Niimpy Documentation

Release dev

the contributors

May 22, 2023

BASICS

Niimpy is a Python package for analyzing and quantifying behavioral data. It uses pandas to read data from disk, perform basic manipulations, and provides many high-level functions for various types of data.

CHAPTER

ONE

INTRODUCTION

1.1 What

Niimpy is a Python package for analyzing and quantifying behavioral data. It uses pandas to read data from disk, perform basic manipulations, provides explorative data analysis functions, offers many high-level preprocessing functions for various types of data, and has functions for behavioral data analysis.

1.2 For Who

Niimpy is intended for researchers and data scientists analyzing digital behavioral data. Its purpose is to facilitate data analysis by providing a standardized replicable workflow.

1.3 Why

Digital behavioral studies using personal digital devices typically produce rich multi-sensor longitudinal datasets of mixed data types. Analyzing such data requires multidisciplinary expertise and software designed for the purpose. Currently, no standardized workflow or tools exist to analyze such data sets. The analysis requires domain knowledge in multiple fields and programming expertise. Niimpy package is specifically designed to analyze longitudinal, multimodal behavioral data. Niimpy is a user-friendly open-source package that can be easily expanded and adapted to specific research requirements. The toolbox facilitates the analysis phase by providing tools for data management, preprocessing, feature extraction, and visualization. The more advanced analysis methods will be incorporated into the toolbox in the future.

1.4 How

The toolbox is divided into four layers by functionality: 1) reading, 2) preprocessing, 3) exploration, and 4) analysis. For more information about the layers, refer the toolbox *Architecture and workflow* chapter. The *quick start* guide is be a good place to start. More detailed demo Jupyter notebooks are provided in the *user guide* chapter. Instructions for individual functions can be found under API chapter *niimpy package*.

This documentation has following chapters:

- Basic information about the toolbox
- Quickstart guide
- API documentation
- User guide

- Community guide
- Data documentation

Basic information contain this introduction, *installation instructions*, *software architecture and workflow schematics*, and information about *compatible data input-formats* and the *required data schema*.

The *quickstart guide* provides a minimal working analysis example to get you started.

The *API documentation* has all technical details, containing instruction about how to use the toolbox functions, classes, return types, arguments and such.

The *user guide* provide more thorough examples of each toolbox layer functionalities. The examples are in Jupyter notebook format.

The community guide has information about the authors, community rules, *contribution*, and our collaborators.

CHAPTER TWO

INSTALLATION

Niimpy is a normal Python package to install. It is not currently available on PyPi, so you can install it manually from github repository:

```
pip install niimpy
```

Note: only supports Python 3 (tested on 3.6. and above).

CHAPTER

THREE

ARCHITECTURE AND WORKFLOW

Niimpy toolbox functionality is organized into four layers:

1. Data Reading
2. Data Preprocessing
3. Data Exploration
4. Data Analysis.

Each layer in implemented as a module. Following table presents the layer properties.

Layer	Purpose
Reading	Read data from the on-disk formats
Preprocessing	Prepare data for analysis
Exploration	Initial analysis, explorative data analysis
Analysis	Data analysis

3.1 Layer: reading

Data is read from the on-disk formats.

Typical input consists of filenames on disk, and typical output is a pandas.DataFrame with a direct mapping of on-disk formats. For convenience, it may do various other small limiting and preprocessing, but should not look inside the data too much.

These are in `niimpy.reading`.

3.2 Layer: preprocessing

After reading the data for analysis, preprocessing can handle filtering, etc. using the standard schema columns. It does not look at or understand actual sensor values, and the unknown sensor-specific columns are passed straight through to a future layer.

Typical input arguments include the DataFrame, and output is the DataFrame slightly adjusted, without affecting sensor-specific columns.

These are in `niimpy.preprocessing`.

3.3 Layer: exploration

These functions can do data aggregation, basic analysis, and visualization which is not specific to any sensor, instead of to the data type.

These are in `niimpy.exploration`.

3.4 Layer: analysis

These functions understand the sensor values and perform analysis based on them.

These are often in modules specific to the type of analysis.

These are in `niimpy.analysis`.

3.5 Workflow

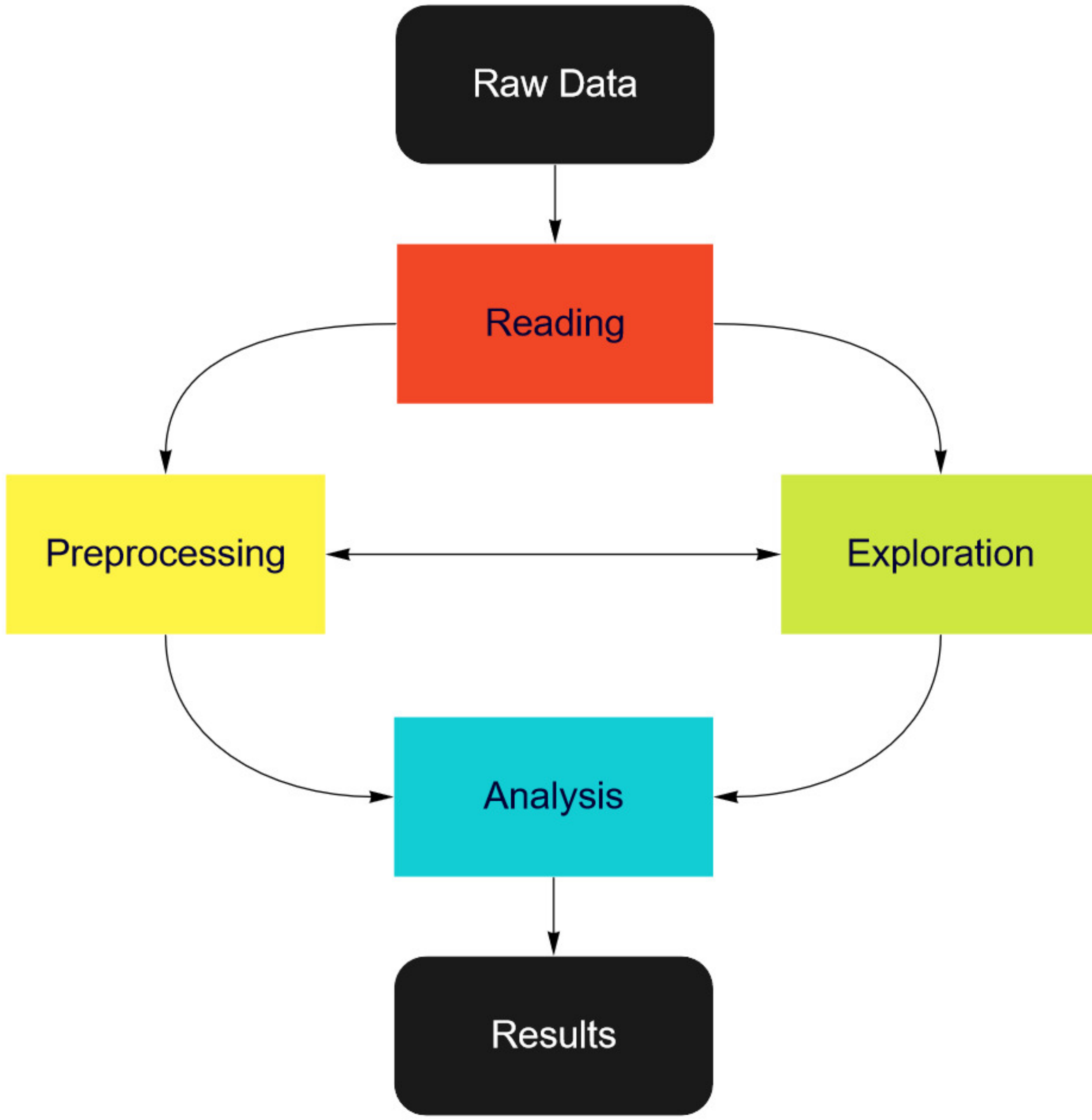
Typical behavioral data analysis workflow consists of following steps:

- Data reading -> Preprocessing -> Explorations -> Analysis

Other possible workflows:

- Data reading -> Exploration -> Preprocessing -> Analysis
- Data reading -> Exploration -> Preprocessing -> Exploration -> Analysis

Niimpy workflow diagram


Raw Data
Reading
Preprocessing
Exploration
Analysis
Results

CHAPTER

FOUR

FILE FORMATS

In principle, Niimpy can deal with any files of any format - you only need to convert them to a DataFrame. Still, it is very useful to have some common formats, so we present two standard formats with default readers:

- **CSV files** are very standard and normal to create and understand, but in order to deal with them everything must be loaded into memory.
- **sqlite3 databases**, which requires sqlite3 to read, but provides more power for filtering and automatic processing without reading everything into memory.

4.1 DataFrame format (in-memory)

In-memory, data is stored in a pandas DataFrame. This is basically a normal dataframe. There are some standardized columns (see the schema) and the index is a DatetimeIndex.

4.2 CSV files

CSV files should have a header that lists the column names and generally be readable by `pandas.read_csv`.

Reading these can be done with `niimpy.read_csv`:

[1]:
```
import os
import niimpy
import niimpy.config as config

# Read the battery data
df= niimpy.read_csv(config.MULTIUSER_AWARE_BATTERY_PATH, tz='Europe/Helsinki')
```

4.3 sqlite3 databases

For the purposes of niimpy, sqlite3 databases can generally be seen as supercharged CSV files.

A single database file could contain multiple datasets within it, thus when reading them a **table name** must be specified.

One reads the entire database into memory using `sqlite.read_sqlite`:

[2]:
```
# Read the sqlite3 data
df= niimpy.read_sqlite(config.SQLITE_SINGLEUSER_PATH, table="AwareScreen", tz='Europe/
↪Helsinki')
```

You can list the tables within a database using `niimpy.reading.read.read_sqlite_tables`:

[3]:
```
niimpy.reading.read.read_sqlite_tables(config.SQLITE_SINGLEUSER_PATH)
```

[3]:
```
{'AwareScreen'}
```

sqlite3 files are highly recommended as a data storage format, since many common exploration options can be done within the database itself without reading the whole data into memory or writing an iterator. However, the interface is more difficult to use. Niimpy (before 2021-07) used this as its primary interface, but since then this interface has been de-emphasized. You can read more in the database section, but this is only recommended if you need efficiency when using massive amounts of data.

4.4 Other formats

You can add readers for any types of formats which you can convert into a Pandas dataframe (so basically anything). For examples of readers, see `niimpy/reading/read.py`. Apply the function `niimpy.preprocessing.util.df_normalize` in order to apply some standardizations to get the standard Niimpy format.

CHAPTER

FIVE

DATA SCHEMA

This page documents the expected data schema of Niimpy. This does *not* extend to the contents of data from sensors (yet), but relates to the metadata applicable to all sensors.

By using a standardized schema (mainly column names), we can promote interoperability of various tools.

5.1 Format

Data is in a tabular (relational) format. A row is an observation, and columns are properties of observations. (At this level of abstraction, an "observation" may be one sensor observation, or some data which contains a package of multiple observations).

In Niimpy, this is internally stored and handled as a pandas.DataFrame. The schema naturally maps to the columns/rows of the DataFrames.

The on-disk format is currently irrelevant, as long as the producers can create a DataFrame of the necessary format. Currently, we provide readers for sqlite3 and csv. Other standards may be implemented later.

5.2 Standard columns in DataFrames

By having standard columns, we can create portable functions that easily operate on diverse data types.

- The **DataFrame index** should be a `pandas.DatetimeIndex`.
- `user`: opaque identifier for the user. Often a string or integer.
- `device`: unique identifier for a user's device (not the device type). For example, a user could have multiple phones, and each would have a separate `device` identifier.
- `time`: timestamp of the observation, in unixtime (seconds since 00:00 on 1970-01-01), stored as an integer. Unixtime is a globally unique measure of an instance of time on Earth, and to get localtime it is combined with a timezone.

  In on-disk formats, `time` is considered the master timestamp, many other time-based properties are computed from it (though you could produce your own DataFrames other ways). In some of the standard formats (CSV/sqlite3), when a file is read, this integer column is automatically converted to the `datetime` column below and the DataFrame index.
- `datetime`: a `DateTime`-compatible object, such as in pandas a numpy.datetime64 object, used only in in-memory representations (not usually written to portable save files). This should be an timezone-aware object, and the data loader handles the timezone conversion. automatically added to DataFrames when loaded.

  It is the responsibility of each loader (or preprocessor) to add this column to the in-memory representation by converting the `time` column to this format. This happens automatically with readers included in `niimpy`.

- `timezone`: Timezone in some format. Not yet used, to be decided.
- For questionaire data
  - `id`: a question identifier. String, should be of form `QUESTIONAIRE_QUESTION`, for example `PHQ9_01`. The common prefix is used to group questions of the same series.
  - `answer`: the answer to the question. Opaque identifier.

Sensor-specific schemas are defined elsewhere. Columns which are not defined here are allowed and considered to be part of the sensors, most APIs should pass through unknown columns for handling in a future layer (sensor analysis).

5.3 Other standard columns in Niimpy

These are not part of the primary schema, but are standard in Niimpy.

- `day`: e.g. `2021-04-09` (str)
- `hour`: hour of day, e.g. `15` (int)

5.4 Standard columns in on-disk formats

For the most part, this maps directly to the columns you see above. An on-disk format should have a `time` column (unixtime, integer) plus whatever else is needed for that particular sensor, based on the above.

CHAPTER

SIX

HOW TO CITE

- Digitraceslab. (n.d.). Digitraceslab/niimpy: Python module for analysis of Behavorial Data. GitHub. Retrieved April 28, 2022, from https://github.com/digitraceslab/niimpy

CHAPTER

SEVEN

SEE ALSO

List of references:

- Aledavood, Talayeh, et al. "Data collection for mental health studies through digital platforms: requirements and design of a prototype." JMIR research protocols 6.6 (2017): e6919. doi:10.2196/resprot.6919
- Triana, Ana María, et al. "Mobile Monitoring of Mood (MoMo-Mood) pilot: A longitudinal, multi-sensor digital phenotyping study of patients with major depressive disorder and healthy controls." medRxiv (2020)

CHAPTER

EIGHT

QUICK START

We will guide you through the main features of `niimpy`. This guide assumes that you have basic knowledge of Python. Also, please refers to the installation page for installing `niimpy`.

This guide provides an example of reading and handling Aware battery data. The tutorial will guide you through 4 basic steps of a data analysis pipeline:

- *Reading*
- *Preprocessing*
- *Visualization*
- *Basic analysis*

[1]:
```
# Setting up plotly environment
import plotly.io as pio
pio.renderers.default = "png"
```

[2]:
```
import numpy as np
import niimpy
from niimpy import config
from niimpy.exploration.eda import punchcard, missingness
from niimpy.preprocessing import battery
```

8.1 Reading

`niimpy` provides a simple function to read data from csv and sqlite database. We will read a csv file containing 1 month of battery data from an individual.

[3]:
```
df = niimpy.read_csv(config.MULTIUSER_AWARE_BATTERY_PATH, tz='Europe/Helsinki')
df.head()
```

[3]:
```
                                           user          device          time  \
2020-01-09 02:20:02.924999936+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:21:30.405999872+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:24:12.805999872+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:35:38.561000192+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09
2020-01-09 02:35:38.953000192+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09

                                     battery_level  battery_status
2020-01-09 02:20:02.924999936+02:00             74               3  \
```

(continues on next page)

(continued from previous page)

```
2020-01-09 02:21:30.405999872+02:00            73               3
2020-01-09 02:24:12.805999872+02:00            72               3
2020-01-09 02:35:38.561000192+02:00            72               2
2020-01-09 02:35:38.953000192+02:00            72               2

                                     battery_health  battery_adaptor
2020-01-09 02:20:02.924999936+02:00               2                0  \
2020-01-09 02:21:30.405999872+02:00               2                0
2020-01-09 02:24:12.805999872+02:00               2                0
2020-01-09 02:35:38.561000192+02:00               2                0
2020-01-09 02:35:38.953000192+02:00               2                2

                                                               datetime
2020-01-09 02:20:02.924999936+02:00 2020-01-09 02:20:02.924999936+02:00
2020-01-09 02:21:30.405999872+02:00 2020-01-09 02:21:30.405999872+02:00
2020-01-09 02:24:12.805999872+02:00 2020-01-09 02:24:12.805999872+02:00
2020-01-09 02:35:38.561000192+02:00 2020-01-09 02:35:38.561000192+02:00
2020-01-09 02:35:38.953000192+02:00 2020-01-09 02:35:38.953000192+02:00
```

8.2 Preprocessing

There are various ways to handle battery data. For example, you can extract the gaps between consecutive battery timestamps.

[4]:
```
gaps = battery.battery_gaps(df, {})
gaps.head()
```

[4]:
```
                                                     battery_gap
user
iGyXetHE3S8u 2019-08-05 14:00:00+03:00    0 days 00:01:18.600000
             2019-08-05 14:30:00+03:00    0 days 00:27:18.396000
             2019-08-05 15:00:00+03:00 0 days 00:51:11.997000192
             2019-08-05 15:30:00+03:00                       NaT
             2019-08-05 16:00:00+03:00 0 days 00:59:23.522999808
```

`niimpy` can also extract the amount of battery data found within an interval.

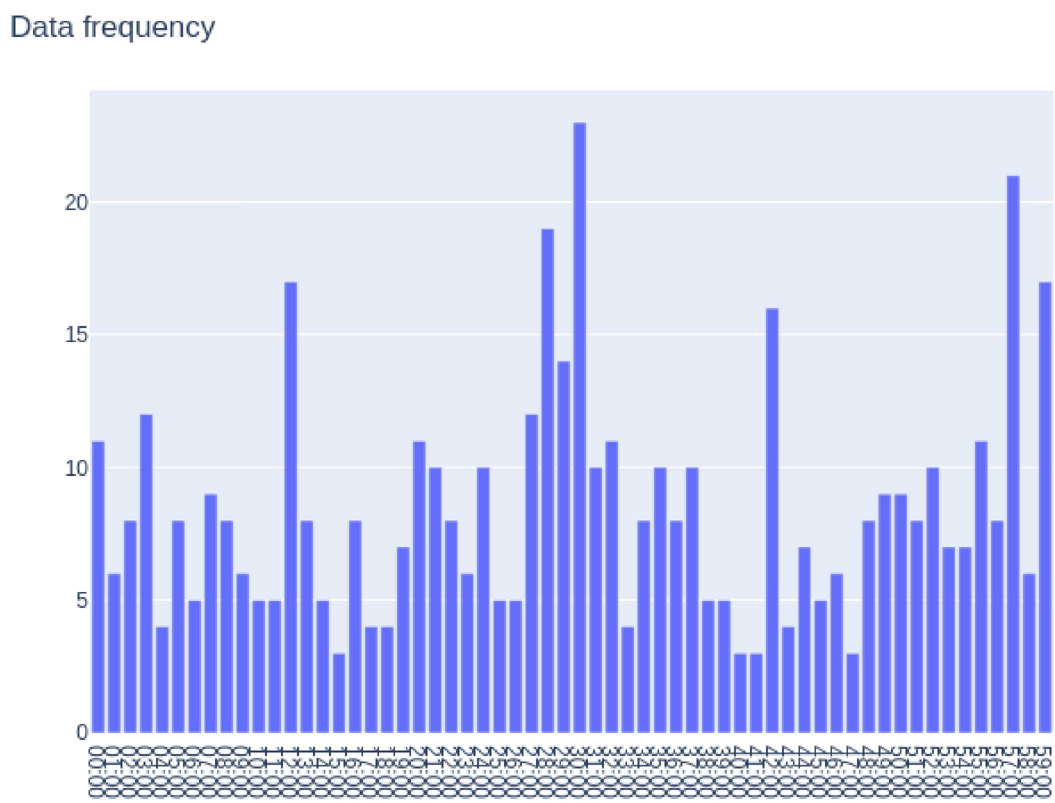
[5]:
```
occurences = battery.battery_occurrences(df, {"resample_args": {"rule": "1H"}})
occurences.head()
```

[5]:
```
                                        occurrences
user
iGyXetHE3S8u 2019-08-05 14:00:00+03:00            3
             2019-08-05 15:00:00+03:00            1
             2019-08-05 16:00:00+03:00            1
             2019-08-05 17:00:00+03:00            1
             2019-08-05 18:00:00+03:00            1
```

8.3 Visualization

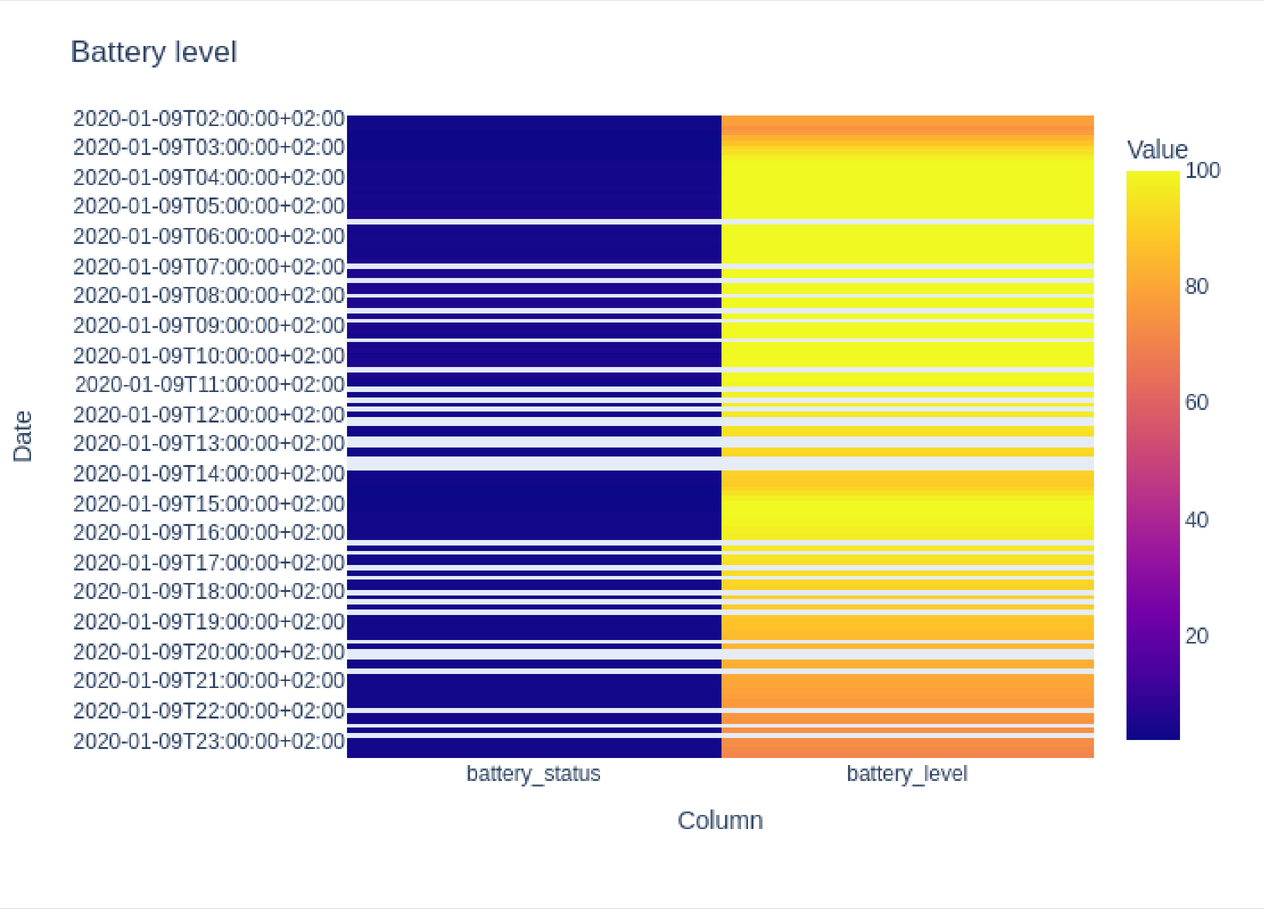
`niimpy` provides a selection of visualization tools curated for exploring behavioural data. For example, you can examine the frenquency of battery level in specified interval.

[6]:
```
fig = missingness.bar_count(df, columns=['battery_level'], sampling_freq='T')
fig.show()
```



In addition, you can analyze the battery level at each sampling interval by using a punchcard plot.

[7]:
```
fig = punchcard.punchcard_plot(df,
                               user_list=['jd9INuQ5BBlW'],
                               columns=['battery_status', 'battery_level'],
                               resample='10T',
                               title="Battery level")
fig.show()
```



For more information, refer to the Exploration section.

CHAPTER

NINE

NIIMPY API DOCS

This section provides function reference for Niimpy. Please refer to the user guide for further details on the function usage.

9.1 niimpy package

9.1.1 Subpackages

niimpy.analysis package

Module contents

niimpy.exploration package

Subpackages

niimpy.exploration.eda package

Submodules

niimpy.exploration.eda.categorical module

Created on Thu Nov 18 14:49:22 2021

@author: arsii

`niimpy.exploration.eda.categorical.`**`categorize_answers`**(*df*, *question*, *answer_column*)

Extract a question answered and count different answers.

Parameters

df
[Pandas Dataframe] Dataframe containing questionnaire data

question
[str] dataframe column sontaining question id

answer_column
[str] dataframe column containing the answer

Returns

category_counts: Pandas Dataframe
Dataframe containing the category counts of answers filtered by the question

`niimpy.exploration.eda.categorical.`**`get_xticks_`**`(`*ser*`)`

Helper function for plot_categories function. Convert series index into xtick values and text.

Parameters

ser
[Pandas series] Series containing the categorized counts

`niimpy.exploration.eda.categorical.`**`plot_categories`**`(`*df*, *title=None*, *xlabel=None*, *ylabel=None*, *width=900*, *height=900*`)`

Create a barplot of categorical data

Parameters

df
[Pandas Dataframe] Dataframe containing categorized data

title
[str] Plot title

xlabel
[str] Plot xlabel

ylabel
[str] Plot ylabel

width
[integer] Plot width

height
[integer] Plot height

Returns

fig: plotly Figure
A barplot of the input data

`niimpy.exploration.eda.categorical.`**`plot_grouped_categories`**`(`*df*, *group*, *title=None*, *xlabel=None*, *ylabel=None*, *width=900*, *height=900*`)`

Plot summary barplot for questionnaire data.

Parameters

df: Pandas DataFrameGroupBy
A grouped dataframe containing categorical data

group: str
Column used to describe group

title
[str] Plot title

xlabel
[str] Plot xlabel

ylabel
[str] Plot ylabel

width
[integer] Plot width

height
[integer] Plot height

Returns

fig: plotly Figure
Figure containing barplots of the data in each group

`niimpy.exploration.eda.categorical.`**`question_by_group`**(*df*, *question*, *id_column='id'*, *answer_column='answer'*, *group='group'*)

Plot summary barplot for questionnaire data.

Parameters

df
[Pandas Dataframe] Dataframe containing questionnaire data

question
[str] question id

answer_column
[str] answer_column containing the answer

group
[str] group by this column

Returns

df
[Pandas DataFrameGroupBy] Dataframe a single answers column filtered by the question parameter and grouped by the group parameter

`niimpy.exploration.eda.categorical.`**`questionnaire_grouped_summary`**(*df*, *question*, *id_column='id'*, *answer_column='answer'*, *group='group'*, *title=None*, *xlabel=None*, *ylabel=None*, *width=900*, *height=900*)

Create a barplot of categorical data

Parameters

df
[Pandas Dataframe] Dataframe containing questionnaire data

question
[str] question id

column
[str] column containing the answer

title
[str] Plot title

xlabel
[str] Plot xlabel

ylabel
[str] Plot ylabel

user
[Bool or str] If str, plot single user data If False, plot group level data

group
[str] group by this column

Returns

fig: plotly Figure
A barplot of the input data

`niimpy.exploration.eda.categorical.`**`questionnaire_summary`**`(`*df*, *question*, *column*, *title=None*, *xlabel=None*, *ylabel=None*, *user=None*, *width=900*, *height=900*`)`

Plot summary barplot for questionnaire data.

Parameters

df
[Pandas Dataframe] Dataframe containing questionnaire data

question
[str] question id

column
[str] column containing the answer

title
[str] Plot title

xlabel
[str] Plot xlabel

ylabel
[str] Plot ylabel

user
[Bool or str] If str, plot single user data If False, plot group level data

Returns

fig: plotly Figure
A barplot summary of the questionnaire

niimpy.exploration.eda.countplot module

Created on Mon Nov 8 14:42:18 2021

@author: arsii

`niimpy.exploration.eda.countplot.`**`barplot_`**`(`*df*, *fig_title*, *xlabel*, *ylabel*`)`

Plot a barplot showing counts for each subjects

A dataframe must have columns named 'user', containing the user id's, and 'values' containing the observation counts.

Parameters

df
[Pandas Dataframe] Dataframe containing the data

fig_title
[str] Plot title

xlabel
[str] Plot xlabel

ylabel
[str] Plot ylabel

Returns

`niimpy.exploration.eda.countplot.`**`boxplot_`**(*df*, *fig_title*, *points='outliers'*, *y='values'*, *xlabel='Group'*, *ylabel='Count'*, *binning=False*)

Plot a boxplot

Parameters

df
[Pandas Dataframe] Dataframe containing the data

fig_title
[str] Plot title

points
[str] If 'all', show all observations next to boxplots If 'outliers', show only outlying points The default is 'outliers'

y: str
A dataframe column to plot

xlabel
[str] Plot xlabel

ylabel
[str] Plot ylabel

Returns

`niimpy.exploration.eda.countplot.`**`calculate_bins`**(*df*, *binning*)

Calculate time index based bins for each observation in the dataframe.

Parameters

df
[Pandas DataFrame]

binning
[str]

to_string
[bool]

Returns

bins
[pandas period index]

`niimpy.exploration.eda.countplot.`**`countplot`**(*df*, *fig_title*, *plot_type='count'*, *points='outliers'*, *aggregation='group'*, *user=None*, *column=None*, *binning=False*)

Create boxplot comparing groups or individual users.

Parameters

df
[pandas DataFrame] A DataFrame to be visuliazed

fig_title
[str] The plot title.

plot_type
[str] If 'count', plot observation count per group (boxplot) or by user (barplot) If 'value', plot observation values per group (boxplot) The default is 'count'

aggregation
[str] If 'group', plot group level summary If 'user', plot user level summary The default is 'group'

user
[str] if given … The default is None

column
[str, optional] if None, count number of rows. If given, count only occurances of that column. The default is None.

Returns

`niimpy.exploration.eda.countplot.`**`get_counts`**`(`*df*, *aggregation*`)`

Calculate datapoint counts by group or by user

Parameters

df
[Pandas DataFrame]

aggregation
[str]

Returns

n_events
[Pandas DataFrame]

niimpy.exploration.eda.lineplot module

Created on Wed Oct 27 09:53:46 2021

@author: arsii

`niimpy.exploration.eda.lineplot.`**`calculate_averages_`**`(`*df*, *column*, *by*`)`

calculate group averages by given timerange

`niimpy.exploration.eda.lineplot.`**`plot_averages_`**`(`*df*, *column*, *by='hour'*`)`

Plot user group level averages by hour or by weekday.

Parameters

df
[Pandas Dataframe] Dataframe containing the data

column
[str] Columns to plot.

by
[str, optional] Indicator for group level averaging. The default is False. If 'hour', hourly averages per group are presented. If 'weekday', daily averages per gruop are presented.

Returns

None.

`niimpy.exploration.eda.lineplot.plot_timeseries_`(*df*, *columns*, *users*, *title*, *xlabel*, *ylabel*, *resample=False*, *interpolate=False*, *window_len=False*, *reset_index=False*)

There goes the text.

Parameters

df
[Pandas Dataframe] Dataframe containing the data

columns
[list or str] Columns to plot.

users
[list or str] Users to plot.

title
[str] Plot title.

xlabel
[str] Plot xlabel.

ylabel
[str] Plot ylabel.

resample
[str, optional] Data resampling frequency. The default is False. For details: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.resample.html

interpolate
[bool, optional] If true, time series will be interpolated using splines. The default is False.

window
[int, optional] Rolling window smoothing window size. The default is False.

reset_index
[bool, optional] If true, dataframe index will be resetted. The default is False.

Returns

None.

`niimpy.exploration.eda.lineplot.resample_data_`(*df*, *resample*, *interpolate*, *window_len*, *reset_index*)

resample dataframe for plotting

`niimpy.exploration.eda.lineplot.timeplot`(*df*, *users*, *columns*, *title*, *xlabel*, *ylabel*, *resample=False*, *interpolate=False*, *window=False*, *reset_index=False*, *by=False*)

Plot a time series plot. Plot selected users and columns or group level averages, aggregated by hour or weekday.

Parameters

df
[Pandas Dataframe] Dataframe containing the data

users
[list or str] Users to plot.

columns
[list or str] Columns to plot.

title
[str] Plot title.

xlabel
[str] Plot xlabel.

ylabel
[str] Plot ylabel.

resample
[str, optional] Data resampling frequency. The default is False. For details: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.resample.html

interpolate
[bool, optional] If true, time series will be interpolated using splines. The default is False.

window
[int, optional] Rolling window smoothing window size. The default is False.

reset_index
[bool, optional] If true, dataframe index will be resetted. The default is False.

by
[str, optional] Indicator for group level averaging. The default is False. If 'hour', hourly averages per group are presented. If 'weekday', daily averages per gruop are presented.

Returns

——-

None.

niimpy.exploration.eda.missingness module

This module is rewritten based on the missingno package. The original files can be found here: https://github.com/ResidentMario/missingno

`niimpy.exploration.eda.missingness.`**`bar`**(*df*, *columns=None*, *title='Data frequency'*, *xaxis_title=''*, *yaxis_title=''*, *sampling_freq=None*, *sampling_method='mean'*)

Display bar chart visualization of the nullity of the given DataFrame.

Parameters

df: pandas Dataframe
Dataframe to plot

columns: list, optional
Columns from input dataframe to investigate missingness. If none is given, uses all columns.

title: str
Figure's title

xaxis_title: str, optional
x_axis's label

yaxis_title: str, optional
y_axis's label

sampling_freq: str, optional
Frequency to resample the data. Requires the dataframe to have datetime-like index. Possible values: 'H', 'T'

sampling_method: str, optional
Resampling method. Possible values: 'sum', 'mean'. Default value is 'mean'.

Returns

——-

fig: Plotly figure.

`niimpy.exploration.eda.missingness.`**`bar_count`**`(`*df*, *columns=None*, *title='Data frequency'*, *xaxis_title=''*, *yaxis_title=''*, *sampling_freq='H'*`)`

Display bar chart visualization of the nullity of the given DataFrame.

Parameters

df: pandas Dataframe
Dataframe to plot

columns: list, optional
Columns from input dataframe to investigate missingness. If none is given, uses all columns.

title: str
Figure's title

xaxis_title: str, optional
x_axis's label

yaxis_title: str, optional
y_axis's label

sampling_freq: str, optional
Frequency to resample the data. Requires the dataframe to have datetime-like index. Possible values: 'H', 'T'

Returns

fig: Plotly figure.

`niimpy.exploration.eda.missingness.`**`heatmap`**`(`*df*, *height=800*, *width=800*, *title=''*, *xaxis_title=''*, *yaxis_title=''*`)`

Return 'plotly' heatmap visualization of the nullity correlation of the Dataframe.

Parameters

df: pandas Dataframe
Dataframe to plot

width: int:
Figure's width

height: int:
Figure's height

Returns

——-

fig: Plotly figure.

`niimpy.exploration.eda.missingness.`**`matrix`**`(`*df*, *height=500*, *title='Data frequency'*, *xaxis_title=''*, *yaxis_title=''*, *sampling_freq=None*, *sampling_method='mean'*`)`

Return matrix visualization of the nullity of data. For now, this function assumes that the data frame is datetime indexed.

Parameters

df: pandas Dataframe
Dataframe to plot

columns: list, optional
Columns from input dataframe to investigate missingness. If none is given, uses all columns.

title: str
Figure's title

xaxis_title: str, optional
x_axis's label

yaxis_title: str, optional
y_axis's label

sampling_freq: str, optional
Frequency to resample the data. Requires the dataframe to have datetime-like index. Possible values: 'H', 'T'

sampling_method: str, optional
Resampling method. Possible values: 'sum', 'mean'. Default value is 'mean'.

Returns

——-

fig: Plotly figure.

niimpy.exploration.eda.punchcard module

Created on Thu Nov 18 16:14:47 2021

@author: arsii

`niimpy.exploration.eda.punchcard.`**`combine_dataframe_`**(*df*, *user_list*, *columns*, *res*, *date_index*, *agg_func=<function mean>*)

resample values from multiple users into new dataframe

Parameters

df
[Pandas Dataframe] Dataframe containing the data

user_list
[list] List containing user names/id's (str)

columns
[list] List of column names (str) to be plotted

res
[str] Resample parameter e.g., 'D' for resampling by day

date_index
[pd.date_range] Date range used as an index

agg_func
[numpy function] Aggregation function used with resample. The default is np.mean

Returns

df_comb
[pd.DataFrame] Resampled and combined dataframe

`niimpy.exploration.eda.punchcard.get_timerange_(df, resample)`

get first and last timepoint from the dataframe, and return a resampled datetimeindex.

Parameters

df
[Pandas Dataframe] Dataframe containing the data

ressample
[str] Resample parameter e.g., 'D' for resampling by day

Returns

date_index
[pd.DatatimeIndex] Resampled DatetimeIndex

`niimpy.exploration.eda.punchcard.punchcard_(df, title, n_xticks, xtitle, ytitle)`

create a punchcard plot

Parameters

df
[Pandas Dataframe] Dataframe containing the data

title
[str] Plot title.

n_xticks
[int or None] Number of xaxis ticks. If None, scaled automatically.

xtitle
[str] Plot xaxis title

ytitle
[str] Plot yaxis title

Returns

fig
[plotly.graph_objs._figure.Figure] Punchcard plot

`niimpy.exploration.eda.punchcard.punchcard_plot(df, user_list=None, columns=None, title='Punchcard Plot', resample='D', normalize=False, agg_func=<function mean>, timerange=False)`

Punchcard plot for given users and column with optional resampling

Parameters

df
[Pandas Dataframe] Dataframe containing the data

user_list
[list, optional] List containing user id's as string. The default is None.

columns
[list, optional] List containing columns as strings. The default is None.

title
[str, optional] Plot title. The default is "Punchcard Plot".

resample
[str, optional] Indicator for resampling frequency. The default is 'D' (day).

agg_func
[numpy function] Aggregation function used with resample. The default is np.mean

normalize
[boolean, optional] If true, data is normalized using min-max-scaling. The default is False.

timerange
[boolean or tuple, optional] If false, timerange is not filtered. If tuple containing timestamps, timerange is filtered. The default is False.

Returns

fig
[plotly.graph_objs._figure.Figure] Punchcard plot

Module contents

Submodules

niimpy.exploration.missingness module

`niimpy.exploration.missingness.`**`missing_data_format`**(*question*, *keep_values=False*)

Returns a series of timestamps in the right format to allow missing data visualization .

Parameters

question: Dataframe

`niimpy.exploration.missingness.`**`missing_noise`**(*database*, *subject*, *start=None*, *end=None*)

Returns a Dataframe with the estimated missing data from the ambient noise sensor.

NOTE: This function aggregates data by day.

Parameters

database: Niimpy database
user: string
start: datetime, optional
end: datetime, optional

Returns

avg_noise: Dataframe

`niimpy.exploration.missingness.`**`screen_missing_data`**(*database*, *subject*, *start=None*, *end=None*)

Returns a DataFrame contanining the percentage (range [0,1]) of loss data calculated based on the transitions of screen status. In general, if screen_status(t) == screen_status(t+1), we declared we have at least one missing point.

Parameters

database: Niimpy database
user: string
start: datetime, optional
end: datetime, optional

Returns

count: Dataframe

niimpy.exploration.setup_dataframe module

`niimpy.exploration.setup_dataframe.`**`create_categorical_dataframe`**`()`

Create a sample Pandas dataframe used by the test functions.

Returns

df
[pandas.DataFrame] Pandas dataframe containing sample data.

`niimpy.exploration.setup_dataframe.`**`create_dataframe`**`()`

Create a sample Pandas dataframe used by the test functions.

Returns

df
[pandas.DataFrame] Pandas dataframe containing sample data.

`niimpy.exploration.setup_dataframe.`**`create_missing_dataframe`**`(`*nrows*, *ncols*, *density=0.9*, *random_state=None*, *index_type=None*, *freq=None*`)`

Create a Pandas dataframe with random missingness.

Parameters

nrows
[int] Number of rows

ncols
[int] Number of columns

density: float
Amount of available data

random_state: float, optional
Random seed. If not given, default to 33.

index_type: float, optional
Accepts the following values: "dt" for timestamp, "int" for integer.

freq: string, optional:
Sampling frequency. This option is only available is index_type is "dt".

Returns

df
[pandas.DataFrame] Pandas dataframe containing sample data with random missing rows.

`niimpy.exploration.setup_dataframe.`**`create_timeindex_dataframe`**`(`*nrows*, *ncols*, *random_state=None*, *freq=None*`)`

Create a datetime index Pandas dataframe

Parameters

nrows
[int] Number of rows

ncols
[int] Number of columns

random_state: float, optional
Random seed. If not given, default to 33.

freq: string, optional:
Sampling frequency.

Returns

——-

df
[pandas.DataFrame] Pandas dataframe containing sample data with random missing rows.

Module contents

niimpy.preprocessing package

Submodules

niimpy.preprocessing.application module

niimpy.preprocessing.application.**app_count**(*df*, *bat*, *screen*, *feature_functions=None*)

This function returns the number of times each app group has been used, within the specified timeframe. The app groups are defined as a dictionary within the feature_functions variable. Examples of app groups are social media, sports, games, etc. If no mapping is given, a default one will be used. If no resampling window is given, the function sets a 30 min default time window. The function aggregates the duration by user, by app group, by timewindow.

Parameters

df: pandas.DataFrame
Input data frame

bat: pandas.DataFrame
Dataframe with the battery information. If no data is available, an empty dataframe should be passed.

screen: pandas.DataFrame
Dataframe with the screen information. If no data is available, an empty dataframe should be passed.

feature_functions: dict, optional
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name "" will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

niimpy.preprocessing.application.**app_duration**(*df*, *bat*, *screen*, *feature_functions=None*)

This function returns the duration of use of different app groups, within the specified timeframe. The app groups are defined as a dictionary within the feature_functions variable. Examples of app groups are social media, sports, games, etc. If no mapping is given, a default one will be used. If no resampling window is given, the function sets a 30 min default time window. The function aggregates the duration by user, by app group, by timewindow.

Parameters

df: pandas.DataFrame
Input data frame

bat: pandas.DataFrame
Dataframe with the battery information. If no data is available, an empty dataframe should be passed.

screen: pandas.DataFrame
Dataframe with the screen information. If no data is available, an empty dataframe should be passed.

feature_functions: dict, optional
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name "application_name" will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.application.classify_app`(*df*, *feature_functions*)

This function is a helper function for other screen preprocessing. The function classifies the screen events into the groups specified by group_map.

Parameters

df: pandas.DataFrame
Input data frame

feature_functions: dict, optional
Dictionary keys containing optional arguments for the computation of screen information. Keys can be column names, other dictionaries, etc. It can contain a dictionary called group_map, which has the mapping to define the app groups. Keys should be the app name, values are the app groups (e.g. 'my_app':'my_app_group')

Returns

df: dataframe
Resulting dataframe

`niimpy.preprocessing.application.extract_features_app`(*df*, *bat*, *screen*, *features=None*)

This function computes and organizes the selected features for application events. The function aggregates the features by user, by app group, by time window. If no time window is specified, it will automatically aggregate the features in 30 mins non-overlapping windows. If no group_map is provided, a default one will be used.

The complete list of features that can be calculated are: app_count, and app_duration.

Parameters

df: pandas.DataFrame
Input data frame

features: dict, optional
Dictionary keys contain the names of the features to compute. If none is given, all features will be computed.

Returns

result: dataframe
Resulting dataframe

niimpy.preprocessing.audio module

niimpy.preprocessing.audio.**audio_count_loud**(*df_u*, *feature_functions=None*)

This function returns the number of times, within the specified timeframe, when there has been some sound louder than 70dB in the environment. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df_u: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

niimpy.preprocessing.audio.**audio_count_silent**(*df_u*, *feature_functions=None*)

This function returns the number of times, within the specified timeframe, when there has been some sound in the environment. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df_u: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

niimpy.preprocessing.audio.**audio_count_speech**(*df_u*, *feature_functions=None*)

This function returns the number of times, within the specified timeframe, when there has been some sound between 65Hz and 255Hz in the environment that could be specified as speech. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df_u: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework

will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.audio.audio_max_db(df_u, feature_functions=None)`

This function returns the maximum decibels of the recorded audio snippets, within the specified timeframe. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df_u: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.audio.audio_max_freq(df_u, feature_functions=None)`

This function returns the maximum frequency of the recorded audio snippets, within the specified timeframe. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df_u: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.audio.audio_mean_db(df_u, feature_functions=None)`

This function returns the mean decibels of the recorded audio snippets, within the specified timeframe. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df_u: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.audio.`**`audio_mean_freq`**`(`*df_u*, *feature_functions=None*`)`

This function returns the mean frequency of the recorded audio snippets, within the specified timeframe. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df_u: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.audio.`**`audio_median_db`**`(`*df_u*, *feature_functions=None*`)`

This function returns the median decibels of the recorded audio snippets, within the specified timeframe. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df_u: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.audio.`**`audio_median_freq`**`(`*df_u*, *feature_functions=None*`)`

This function returns the median frequency of the recorded audio snippets, within the specified timeframe. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df_u: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.audio.`**`audio_min_db`**`(`*df_u*, *feature_functions=None*`)`

This function returns the minimum decibels of the recorded audio snippets, within the specified timeframe. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df_u: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.audio.`**`audio_min_freq`**`(`*df_u*, *feature_functions=None*`)`

This function returns the minimum frequency of the recorded audio snippets, within the specified timeframe. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df_u: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.audio.audio_std_db(df_u, feature_functions=None)`

This function returns the standard deviation of the decibels of the recorded audio snippets, within the specified timeframe. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df_u: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.audio.audio_std_freq(df_u, feature_functions=None)`

This function returns the standard deviation of the frequency of the recorded audio snippets, within the specified timeframe. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df_u: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.audio.extract_features_audio(df, features=None)`

This function computes and organizes the selected features for audio snippets that have been recorded using Aware Framework. The function aggregates the features by user, by time window. If no time window is specified, it will automatically aggregate the features in 30 mins non-overlapping windows.

The complete list of features that can be calculated are: audio_count_silent, audio_count_speech, audio_count_loud, audio_min_freq, audio_max_freq, audio_mean_freq, audio_median_freq, audio_std_freq, audio_min_db, audio_max_db, audio_mean_db, audio_median_db, audio_std_db

Parameters

df: pandas.DataFrame
Input data frame

features: dict, optional
Dictionary keys contain the names of the features to compute. If none is given, all features will be computed.

Returns

result: dataframe
Resulting dataframe

niimpy.preprocessing.battery module

`niimpy.preprocessing.battery.`**`battery_charge_discharge`**`(`*df*, *feature_functions*`)`

Returns a DataFrame showing the mean difference in battery values and mean battery charge/discharge rate within specified time windows. If there is no specified timeframe, the function sets a 30 min default time window. Parameters ———- df: dataframe with date index

`niimpy.preprocessing.battery.`**`battery_discharge`**`(`*df*, *feature_functions*`)`

This function returns the mean discharge rate of the battery within a specified time window. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow. Parameters ———- df: pandas.DataFrame

Dataframe with the battery information

feature_functions: dict, optional
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc.

Returns

result: dataframe

`niimpy.preprocessing.battery.`**`battery_gaps`**`(`*df*, *feature_functions*`)`

Returns a DataFrame with the mean time difference between consecutive battery timestamps. The mean is calculated within intervals specified in feature_functions. The minimum size of the considered deltas can be decided with the min_duration_between parameter.

Parameters

df: pandas.DataFrame
Dataframe with the battery information

feature_functions: dict, optional
Dictionary keys containing optional arguments for the computation of batter information. Keys can be column names, other dictionaries, etc.

Optional arguments in feature_functions:
min_duration_between: Timedelta, for example, pd.Timedelta(minutes=5)

`niimpy.preprocessing.battery.`**`battery_mean_level`**`(`*df*, *feature_functions*`)`

This function returns the mean battery level within the specified timeframe. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow. Parameters ———- df: pandas.DataFrame

Dataframe with the battery information

feature_functions: dict, optional
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc.

Returns

result: dataframe

`niimpy.preprocessing.battery.battery_median_level(df, feature_functions)`

This function returns the median battery level within the specified timeframe. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow. Parameters ——— df: pandas.DataFrame

Dataframe with the battery information

feature_functions: dict, optional
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc.

Returns

result: dataframe

`niimpy.preprocessing.battery.battery_occurrences(df, feature_functions)`

Returns a dataframe showing the amount of battery data points found within a specified time window. If there is no specified timeframe, the function sets a 30 min default time window. Parameters ——— df: pandas.DataFrame

Dataframe with the battery information

feature_functions: dict, optional
Dictionary keys containing optional arguments for the computation of batter information. Keys can be column names, other dictionaries, etc.

`niimpy.preprocessing.battery.battery_shutdown_time(df, feature_functions)`

This function returns the total time the phone has been turned off within a specified time window. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow. Parameters ——— df: pandas.DataFrame

Dataframe with the battery information

feature_functions: dict, optional
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc.

Returns

result: dataframe

`niimpy.preprocessing.battery.battery_std_level(df, feature_functions)`

This function returns the standard deviation battery level within the specified timeframe. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow. Parameters ——— df: pandas.DataFrame

Dataframe with the battery information

feature_functions: dict, optional
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc.

Returns

result: dataframe

`niimpy.preprocessing.battery.`**`extract_features_battery`**(*df*, *feature_functions=None*)

Calculates battery features

Parameters

df
[pd.DataFrame] dataframe of battery data. It must contain these columns: *battery_level* and *battery_status*.

feature_functions
[map (dictionary) of functions that compute features.] it is a map of map, where the keys to the first map is the name of functions that compute features and the nested map contains the keyword arguments to that function. If there is no arguments use an empty map. Default is None. If None, all the available functions are used. Those functions are in the dict *battery.ALL_FEATURE_FUNCTIONS*. You can implement your own function and use it instead or add it to the mentioned map.

Returns

features
[pd.DataFrame] Dataframe of computed features where the index is users and columns are the the features.

`niimpy.preprocessing.battery.`**`find_battery_gaps`**(*battery_df*, *other_df*, *feature_functions*)

Returns a dataframe showing the gaps found only in the battery data. The default interval is 6 hours. Parameters ——— battery_df: Dataframe other_df: Dataframe

The data you want to compare with

`niimpy.preprocessing.battery.`**`find_non_battery_gaps`**(*battery_df*, *other_df*, *feature_functions*)

Returns a dataframe showing the gaps found only in the other data. The default interval is 6 hours. Parameters ——— battery_df: Dataframe other_df: Dataframe

The data you want to compare with

`niimpy.preprocessing.battery.`**`find_real_gaps`**(*battery_df*, *other_df*, *feature_functions*)

Returns a dataframe showing the gaps found both in the battery data and the other data. The default interval is 6 hours. Parameters ——— battery_df: Dataframe other_df: Dataframe

The data you want to compare with

`niimpy.preprocessing.battery.`**`format_battery_data`**(*df*, *feature_functions*)

Returns a DataFrame with battery data for a user. Parameters ——— battery: DataFrame with battery data

`niimpy.preprocessing.battery.`**`shutdown_info`**(*df*, *feature_functions*)

Returns a pandas DataFrame with battery information for the timestamps when the phone has shutdown. This includes both events, when the phone has shut down and when the phone has been rebooted. NOTE: This is a helper function created originally to preprocess the application info data Parameters ——— bat: pandas.DataFrame

Dataframe with the battery information

feature_functions: dict, optional
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc.

Returns

shutdown: pandas series

niimpy.preprocessing.communication module

`niimpy.preprocessing.communication.call_count(df, feature_functions=None)`

This function returns the number of times, within the specified timeframe, when a call has been received, missed, or initiated. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.communication.call_duration_mean(df, feature_functions=None)`

This function returns the average duration of each call type, within the specified timeframe. The call types are incoming, outgoing, and missed. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.communication.call_duration_median(df, feature_functions=None)`

This function returns the median duration of each call type, within the specified timeframe. The call types are incoming, outgoing, and missed. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df: pandas.DataFrame
Input data frame

bat: pandas.DataFrame
Dataframe with the battery information

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.communication.`**`call_duration_std`**`(`*df*, *feature_functions=None*`)`

This function returns the standard deviation of the duration of each call type, within the specified timeframe. The call types are incoming, outgoing, and missed. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.communication.`**`call_duration_total`**`(`*df*, *feature_functions=None*`)`

This function returns the total duration of each call type, within the specified timeframe. The call types are incoming, outgoing, and missed. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.communication.call_outgoing_incoming_ratio`(*df*, *feature_functions=None*)

This function returns the ratio of outgoing calls over incoming calls, within the specified timeframe. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.communication.extract_features_comms`(*df*, *features=None*)

This function computes and organizes the selected features for calls and SMS events. The function aggregates the features by user, by time window. If no time window is specified, it will automatically aggregate the features in 30 mins non-overlapping windows.

The complete list of features that can be calculated are: call_duration_total, call_duration_mean, call_duration_median, call_duration_std, call_count, call_outgoing_incoming_ratio, sms_count

Parameters

df: pandas.DataFrame
Input data frame

features: dict, optional
Dictionary keys contain the names of the features to compute. If none is given, all features will be computed.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.communication.sms_count`(*df*, *feature_functions=None*)

This function returns the number of times, within the specified timeframe, when an SMS has been sent/received. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

niimpy.preprocessing.filter module

Generic DataFrame filtering

This module provides functions for generic DataFrame filtering. In many cases, it is simpler to do these filtering operations yourself directly on the DataFrames, but these functions simplify the operations of standard arguments in other functions.

`niimpy.preprocessing.filter.`**`filter_dataframe`**`(`*df*, *user=None*, *start=None*, *end=None*, *rename_columns={}*`)`

Standard dataframe preprocessing filter.

This implements some standard and common dataframe preprocessing options, which are used in very many functions. It is likely simpler and more clear to do these yourself on the DataFrames directly.

- select only certain user: *df['user'] == user*
- select date range: *df[start:end]*

- column map: *df.rename(columns=rename_columns)*

It returns a new dataframe (and does not modify the passed one in-place).

niimpy.preprocessing.location module

`niimpy.preprocessing.location.`**`cluster_locations`**`(`*lats*, *lons*, *min_samples=5*, *eps=200*`)`

Performs clustering on the locations

Parameters

lats
[pd.DataFrame] Latitudes

lons
[pd.DataFrame] Longitudes

mins_samples
[int] Minimum number of samples to form a cluster. Default is 5.

eps
[float] Epsilone parameter in DBSCAN. The maximum distance between two neighbour samples. Default is 200.

Returns

clusters
[array] Array of clusters. -1 indicates outlier.

`niimpy.preprocessing.location.`**`compute_nbin_maxdist_home`**`(`*lats*, *lons*, *latlon_home*, *home_radius=50*`)`

Computes number of bins in home and maximum distance to home

Parameters

lats
[pd.DataFrame] Latitudes

lons
[pd.DataFrame] Longitudes

latlon_home
[array] A tuple (lat, lon) showing the coordinate of home

Returns

(n_home, max_dist_home)
[tuple] *n_home*: number of bins the person has been near the home *max_dist_home*: maximum distance that the person has been from home

`niimpy.preprocessing.location.`**`distance_matrix`**(*lats*, *lons*)

Compute distance matrix using great-circle distance formula

https://en.wikipedia.org/wiki/Great-circle_distance#Formulae

Parameters

lats
[array] Latitudes

lons
[array] Longitudes

Returns

dists
[matrix] Entry *(i, j)* shows the great-circle distance between point *i* and *j*, i.e. distance between *(lats[i], lons[i])* and *(lats[j], lons[j])*.

`niimpy.preprocessing.location.`**`extract_features_location`**(*df*, *feature_functions=None*)

Calculates location features

Parameters

df
[pd.DataFrame] dataframe of location data. It must contain these columns: *double_latitude*, *double_longitude*, *user*, *group*. *double_speed* is optional. If not provided, it will be computed manually.

speed_threshold
[float] Bins whose speed is lower than *speed_threshold* are considred *static* and the rest are *moving*.

feature_functions
[map (dictionary) of functions that compute features.] it is a map of map, where the keys to the first map is the name of functions that compute features and the nested map contains the keyword arguments to that function. If there is no arguments use an empty map. Default is None. If None, all the available functions are used. Those functions are in the dict *location.ALL_FEATURE_FUNCTIONS*. You can implement your own function and use it instead or add it to the mentioned map.

Returns

features
[pd.DataFrame] Dataframe of computed features where the index is users and columns are the the features.

`niimpy.preprocessing.location.`**`filter_location`**(*location*, *remove_disabled=True*, *remove_zeros=True*, *remove_network=True*)

Remove low-quality or weird location samples

Parameters

location
[pd.DataFrame] DataFrame of locations

remove_disabled
[bool] Remove locations whose *label* is disabled

remove_zerso
[bool] Remove locations which their latitude and longitueds are close to 0

remove_network
[bool] Keep only locations whose *provider* is *gps*

Returns

location
[pd.DataFrame]

`niimpy.preprocessing.location.`**`find_home`**`(`*lats*, *lons*, *times*`)`

Find coordinates of the home of a person

Home is defined as the place most visited between 12am - 6am. Locations within this time period first clustered and then the center of largest clusetr shows the home.

Parameters

lats
[array-like] Latitudes

lons
[array-like] Longitudes

times
[array-like] Time of the recorderd coordinates

Returns

(lat_home, lon_home)
[tuple of floats] Coordinates of the home

`niimpy.preprocessing.location.`**`get_speeds_totaldist`**`(`*lats*, *lons*, *times*`)`

Computes speed of bins with dividing distance by their time difference

Parameters

lats
[array-like] Array of latitudes

lons
[array-like] Array of longitudes

times
[array-like] Array of times associted with bins

Returns

(speeds, total_distances)
[tuple of speeds (array) and total distance travled (float)]

`niimpy.preprocessing.location.`**`location_distance_features`**`(`*df*, *feature_functions={}*`)`

Calculates features related to distance and speed.

Parameters

df: dataframe with date index
feature_functions: A dictionary of optional arguments
Optional arguments in feature_functions:
longitude_column: The name of the column with longitude data in a floating point format. Defaults to 'double_longitude'. latitude_column: The name of the column with latitude data in a floating point format. Defaults to 'double_latitude'. speed_column: The name of the column with speed data in a floating point format. Defaults to 'double_speed'. resample_args: a dictionary of arguments for the Pandas resample function. For example to resample by hour, you would pass {"rule": "1H"}.

`niimpy.preprocessing.location.`**`location_number_of_significant_places`**`(`*df*, *feature_functions={}*`)`

Computes number of significant places

`niimpy.preprocessing.location.`**`location_significant_place_features`**`(`*df*, *feature_functions={}*`)`

Calculates features related to Significant Places.

Parameters

df: dataframe with date index
feature_functions: A dictionary of optional arguments
Optional arguments in feature_functions:
longitude_column: The name of the column with longitude data in a floating point format. Defaults to 'double_longitude'. latitude_column: The name of the column with latitude data in a floating point format. Defaults to 'double_latitude'. speed_column: The name of the column with speed data in a floating point format. Defaults to 'double_speed'. resample_args: a dictionary of arguments for the Pandas resample function. For example to resample by hour, you would pass {"rule": "1H"}.

`niimpy.preprocessing.location.`**`number_of_significant_places`**`(`*lats*, *lons*, *times*`)`

Computes number of significant places.

Number of significant plcaes is computed by first clustering the locations in each month and then taking the median of the number of clusters in each month.

It is assumed that *lats* and *lons* are the coordinates of static points.

Parameters

lats
[pd.DataFrame] Latitudes

lons
[pd.DataFrame] Longitudes

times
[array] Array of times

Returns
[the number of significant places discovered]

niimpy.preprocessing.sampledata module

Sample data of different types

niimpy.preprocessing.screen module

niimpy.preprocessing.screen.**duration_util_screen**(*df*)

This function is a helper function for other screen preprocessing. The function computes the duration of an event, based on the classification function event_classification_screen.

Parameters

df: pandas.DataFrame
Input data frame

Returns

df: dataframe
Resulting dataframe

niimpy.preprocessing.screen.**event_classification_screen**(*df*, *feature_functions*)

This function is a helper function for other screen preprocessing. The function classifies the screen events into four transition types: on, off, in use, and undefined, based on the screen events recorded. For example, if two consecutive events are 0 and 3, there has been a transition from off to unlocked, i.e. the phone has been unlocked and the events will be classified into the "use" transition.

Parameters

df: pandas.DataFrame
Input data frame

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

df: dataframe
Resulting dataframe

niimpy.preprocessing.screen.**extract_features_screen**(*df*, *bat*, *features=None*)

This function computes and organizes the selected features for screen events that have been recorded using Aware Framework. The function aggregates the features by user, by time window. If no time window is specified, it will automatically aggregate the features in 30 mins non-overlapping windows.

The complete list of features that can be calculated are: screen_off, screen_count, screen_duration, screen_duration_min, screen_duration_max, screen_duration_median, screen_duration_mean, screen_duration_std, and screen_first_unlock.

Parameters

df: pandas.DataFrame
Input data frame

features: dict
Dictionary keys contain the names of the features to compute. If none is given, all features will be computed.

Returns

computed_features: dataframe
Resulting dataframe

niimpy.preprocessing.screen.**screen_count**(*df*, *bat*, *feature_functions=None*)

This function returns the number of times, within the specified timeframe, when the screen has turned off, turned on, and been in use. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df: pandas.DataFrame
Input data frame

bat: pandas.DataFrame
Dataframe with the battery information

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

df: dataframe
Resulting dataframe

niimpy.preprocessing.screen.**screen_duration**(*df*, *bat*, *feature_functions=None*)

This function returns the duration (in seconds) of each transition, within the specified timeframe. The transitions are off, on, and in use. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df: pandas.DataFrame
Input data frame

bat: pandas.DataFrame
Dataframe with the battery information

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

niimpy.preprocessing.screen.**screen_duration_max**(*df*, *bat*, *feature_functions=None*)

This function returns the duration (in seconds) of each transition, within the specified timeframe. The transitions are off, on, and in use. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df: pandas.DataFrame
Input data frame

bat: pandas.DataFrame
Dataframe with the battery information

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.screen.screen_duration_mean`(*df*, *bat*, *feature_functions=None*)

This function returns the duration (in seconds) of each transition, within the specified timeframe. The transitions are off, on, and in use. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df: pandas.DataFrame
Input data frame

bat: pandas.DataFrame
Dataframe with the battery information

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.screen.screen_duration_median`(*df*, *bat*, *feature_functions=None*)

This function returns the duration (in seconds) of each transition, within the specified timeframe. The transitions are off, on, and in use. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df: pandas.DataFrame
Input data frame

bat: pandas.DataFrame
Dataframe with the battery information

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

niimpy.preprocessing.screen.**screen_duration_min**(*df*, *bat*, *feature_functions=None*)

This function returns the duration (in seconds) of each transition, within the specified timeframe. The transitions are off, on, and in use. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df: pandas.DataFrame
Input data frame

bat: pandas.DataFrame
Dataframe with the battery information

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

niimpy.preprocessing.screen.**screen_duration_std**(*df*, *bat*, *feature_functions=None*)

This function returns the duration (in seconds) of each transition, within the specified timeframe. The transitions are off, on, and in use. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df: pandas.DataFrame
Input data frame

bat: pandas.DataFrame
Dataframe with the battery information

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

result: dataframe
Resulting dataframe

niimpy.preprocessing.screen.**screen_first_unlock**(*df*, *bat*, *feature_functions*)

This function returns the first time the phone was unlocked each day. The data is aggregated by user, by day.

Parameters

df: pandas.DataFrame
Input data frame

bat: pandas.DataFrame
Dataframe with the battery information

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used.

Returns

result: dataframe
Resulting dataframe

`niimpy.preprocessing.screen.`**`screen_off`**(*df*, *bat*, *feature_functions=None*)

This function returns the timestamps, within the specified timeframe, when the screen has turned off. If there is no specified timeframe, the function sets a 30 min default time window. The function aggregates this number by user, by timewindow.

Parameters

df: pandas.DataFrame
Input data frame

bat: pandas.DataFrame
Dataframe with the battery information

feature_functions: dict, optional
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc.

Returns

df: dataframe
Resulting dataframe

`niimpy.preprocessing.screen.`**`util_screen`**(*df*, *bat*, *feature_functions*)

This function is a helper function for all other screen preprocessing. The function has the option to merge information from the battery sensors to include data when the phone is shut down. The function also detects the missing datapoints (i.e. not allowed transitions like ON to ON).

Parameters

df: pandas.DataFrame
Input data frame

bat: pandas.DataFrame
Dataframe with the battery information

feature_functions: dict
Dictionary keys containing optional arguments for the computation of scrren information. Keys can be column names, other dictionaries, etc. The functions needs the column name where the data is stored; if none is given, the default name employed by Aware Framework will be used. To include information about the resampling window, please include the selected parameters from pandas.DataFrame.resample in a dictionary called resample_args.

Returns

df: dataframe
Resulting dataframe

niimpy.preprocessing.subject_selection module

niimpy.preprocessing.survey module

niimpy.preprocessing.survey.**daily_affect_variability**(*questions*, *subject=None*)

Returns two DataFrames corresponding to the daily affect variability and mean daily affect, both measures defined in the OLO paper available in 10.1371/journal.pone.0110907. In brief, the mean daily affect computes the mean of each of the 7 questions (e.g. sad, cheerful, tired) asked in a likert scale from 0 to 7. Conversely, the daily affect viariability computes the standard deviation of each of the 7 questions.

NOTE: This function aggregates data by day.

Parameters

questions: DataFrame with subject data (or database for backwards compatibility)
subject: string, optional (backwards compatibility only, in the future do filtering before).

Returns

DLA_mean: mean of the daily affect
DLA_std: standard deviation of the daily affect

niimpy.preprocessing.survey.**survey_convert_to_numerical_answer**(*df*, *answer_col*, *question_id*, *id_map*, *use_prefix=False*)

Convert text answers into numerical value (assuming a long dataframe). Use answer mapping dictionaries provided by the users to convert the answers. Can convert multiple questions having the same prefix (e.g., PSS10_1, PSS10_2, . . . ,PSS10_9) if prefix mapping is provided. Function returns original values for the answers that have not been specified for conversion.

Parameters

df
[pandas dataframe] Dataframe containing the questions

answer_col
[str] Name of the column containing the answers

question_id
[str] Name of the column containing the question id.

id_map
[dictionary] Dictionary containing answer mappings (value) for each question_id (key), or a dictionary containing a map for each question id prefix if use_prefix option is used.

use_prefix
[boolean] If False, uses given map (id_map) to convert questions. The default is False. If True, use question id prefix map, so that multiple question_id's having the same prefix may be converted on the same time.

Returns

result
[pandas series] Series containing converted values and original values for aswers hat are not supposed to be converted.

niimpy.preprocessing.survey.**survey_print_statistic**(*df*, *question_id_col='id'*, *answer_col='answer'*, *prefix=None*, *group=None*)

Return survey statistic. Assuming that the question ids are stored in question_id_col and the survey answers are stored in answer_col, this function returns all the relevant statistics for each question. The statistic includes min, max, average and s.d of the scores of each question.

Parameters

df: pandas.DataFrame
Input data frame

question_id_col: string.
Column contains question id.

answer_col: string
Column contains answer in numerical values.

prefix: list, optional
List contains survey prefix. If None is given, search question_id_col for all possible categories.

group: string, optional
Column contains group factor. If this is given, survey statistics for each group will be returned

Returns

——-

dict: dictionary
A dictionary contains summary of each questionaire category. Example: {'PHQ9': {'min': 3, 'max': 8, 'avg': 4.5, 'std': 2}}

niimpy.preprocessing.survey.**survey_sum_scores**(*df*, *survey_prefix=None*, *answer_col='answer'*, *id_column='id'*)

Sum all columns (like `PHQ9_*`) to get a survey score.

Parameters

df: pandas DataFrame
DataFrame should be a DateTime index, an answer_column with numeric scores, and an id_column with question IDs like "PHQ9_1", "PHQ9_2", etc. The given survey_prefix is the "PHQ9" (no underscore) part which selects the right questions (rows not matching this prefix won't be included).

survey_prefix: string
The survey prefix in the 'id' column, e.g. 'PHQ9'. An '_' is appended.

niimpy.preprocessing.tracker module

niimpy.preprocessing.tracker.**extract_features_tracker**(*df*, *features=None*)

This function computes and organizes the selected features for tracker data recorded using Polar Ignite.

The complete list of features that can be calculated are: tracker_daily_step_distribution

Parameters

df: pandas.DataFrame
Input data frame

features: dict, optional
Dictionary keys contain the names of the features to compute. The value of the keys is the list of parameters that will be passed to the function. If none is given, all features will be computed.

Returns

result: dataframe
Resulting dataframe

niimpy.preprocessing.tracker.**step_summary**(*df*, *value_col='values'*, *user_id=None*, *start_date=None*, *end_date=None*)

Return the summary of step count in a time range. The summary includes the following information of step count per day: mean, standard deviation, min, max

Parameters

df
[Pandas Dataframe] Dataframe containing the hourly step count of an individual. The dataframe must be date time index.

value_col: str.
Column contains step values. Default value is "values".

user_id: list. Optional
List of user id. If none given, returns summary for all users.

start_date: string. Optional
Start date of time segment used for computing the summary. If not given, acquire summary for the whole time range.

end_date: string. Optional
End date of time segment used for computing the summary. If not given, acquire summary for the whole time range.

Returns

summary_df: pandas DataFrame
A dataframe containing user id and associated step summary.

niimpy.preprocessing.tracker.**tracker_daily_step_distribution**(*steps_df*)

Return distribution of steps within each day. Assuming the step count is recorded at hourly resolution, this function will compute the contribution of each hourly step count into the daily count (percentage wise).

Parameters

steps_df
[Pandas Dataframe] Dataframe containing the hourly step count of an individual.

Returns

df: pandas DataFrame
A dataframe containing the distribution of step count per day at hourly resolution.

niimpy.preprocessing.util module

niimpy.preprocessing.util.**aggregate**(*df*, *freq*, *method_numerical='mean'*, *method_categorical='first'*, *groups=['user']*, ***resample_kwargs*)

Grouping and resampling the data. This function performs separated resampling for different types of columns: numerical and categorical.

Parameters

df
[pandas Dataframe] Dataframe to resample

freq
[string] Frequency to resample the data. Requires the dataframe to have datetime-like index.

method_numerical
[str] Resampling method for numerical columns. Possible values: 'sum', 'mean', 'median'. Default value is 'mean'.

method_categorical
[str] Resampling method for categorical columns. Possible values: 'first', 'mode', 'last'.

groups
[list] Columns used for groupby operation.

resample_kwargs
[dict] keywords to pass pandas resampling function

Returns

An aggregated and resampled multi-index dataframe.

`niimpy.preprocessing.util.`**`date_range`**`(`*df*, *start*, *end*`)`

Extract out a certain date range from a DataFrame.

Extract out a certain data range from a dataframe. The index must be the dates, and the index must be sorted.

`niimpy.preprocessing.util.`**`df_normalize`**`(`*df*, *tz=None*, *old_tz=None*`)`

Normalize a df (from sql) before presenting it to the user.

This sets the dataframe index to the time values, and converts times to pandas.TimeStamp:s. Modifies the data frame inplace.

`niimpy.preprocessing.util.`**`install_extensions`**`()`

Automatically install sqlite extension functions.

Only works on Linux for now, improvements welcome.

`niimpy.preprocessing.util.`**`occurrence`**`(`*series*, *bin_width=720*, *grouping_width=3600*`)`

Number of 12-minute

This reproduces the logic of the "occurrence" database function, without needing the database.

inputs: pandas.Series of pandas.Timestamps

Output: pandas.DataFrame with timestamp index and 'occurance' column.

TODO: use the grouping_width option.

`niimpy.preprocessing.util.`**`set_tz`**`(`*tz*`)`

Globally set the preferred local timezone

`niimpy.preprocessing.util.`**`tmp_timezone`**`(`*new_tz*`)`

Temporarily override the global timezone for a black.

This is used as a context manager:

```
with tmp_timezone('Europe/Berlin'):
    ....
```

Note: this overrides the global timezone. In the future, there will be a way to handle timezones as non-global variables, which should be preferred.

`niimpy.preprocessing.util.`**`to_datetime`**`(`*value*`)`

`niimpy.preprocessing.util.`**`uninstall_extensions`**`()`

Uninstall any installed extensions

Module contents

niimpy.reading package

Submodules

niimpy.reading.database module

Read data from sqlite3 databases.

Direct use of this module is mostly deprecated.

Read data from sqlite3 databases, both into pandas.DataFrame:s (Database.raw(), among other functions), and Database objects. The Database object does not immediately load data, but provides some methods to load data on demand later, possibly doing various filtering and preprocessing already at the loading stage. This can save memory and processing time, but is much more complex.

This module is mostly out-of-use now: read.read_sqlite is used instead, which wraps the .raw() method and reads all data into memory.

Database format

When reading data, a table name must be specified (which allows multiple datasets to be put in one file). Table column names map to dataframe column names, with various standard processing (for example the 'time' column being converted to the index)

Quick usage

db = database.open(FILE_NAME, tz=TZ) df = db.raw(TABLE_NAME, user=database.ALL)

Recommend usage:

df = niimpy.read_sqlite(FILE_NAME, TABLE_NAME, tz=TZ)

See also

niimpy.reading.read_*: currently recommended functions to access all types of data, including databases.

class `niimpy.reading.database.`**`ALL`**

Bases: `object`

Sentinel value for all users

class `niimpy.reading.database.`**`Data1`**(*db*, *tz=None*)

Bases: `object`

Database wrapper for niimpy data.

This opens a database and provides methods to do common operations.

Methods

`count`(*args, **kwargs)	Return the number of rows
`execute`(*args, **kwargs)	Execute rauw SQL code.
`exists`(*args, **kwargs)	Returns True if any data exists
`first`(table, user[, start, end, offset, ...])	Return earliest data point.
`get_survey_score`(table, user, survey[, ...])	Get the survey results, summing scores.
`last`(*args, **kwargs)	Return the latest timestamp.
`raw`(table, user[, limit, offset, start, end])	Read all data in a table and return it as a DataFrame.
`tables`()	List all tables that are inside of this database.
`user_table_counts`()	Return table of number of data points per user, per table.
`users`([table])	Return set of all users in all tables
`validate_username`(user)	Validate a username, for single/multiuser database and so on.

hourly	
occurrence	
timestamps	

count(**args*, ***kwargs*)

Return the number of rows

See the “first” for more information.

execute(**args*, ***kwargs*)

Execute rauw SQL code.

Execute raw SQL. Smply proxy all arguments to self.conn.execute(). This is simply a convenience shortcut.

exists(**args*, ***kwargs*)

Returns True if any data exists

Follows the same syntax as .first(), .last(), and .count(), but the limit argument is not used.

first(*table*, *user*, *start=None*, *end=None*, *offset=None*, *_aggregate='min'*, *_limit=None*)

Return earliest data point.

Return None if there is no data.

get_survey_score(*table*, *user*, *survey*, *limit=None*, *start=None*, *end=None*)

Get the survey results, summing scores.

survey: The servey prefix in the ‘id’ column, e.g. ‘PHQ9’. An ‘_’ is appended.

hourly(*table*, *user*, *columns=[]*, *limit=None*, *offset=None*, *start=None*, *end=None*)

last(**args*, ***kwargs*)

Return the latest timestamp.

See the “first” for more information.

occurrence(*table*, *user*, *bin_width=720*, *limit=None*, *offset=None*, *start=None*, *end=None*)

raw(*table*, *user*, *limit=None*, *offset=None*, *start=None*, *end=None*)

Read all data in a table and return it as a DataFrame.

This reads all data (subject to several possible filters) and returns it as a DataFrame.

tables()

List all tables that are inside of this database.

Returns a set.

timestamps(*table*, *user*, *limit=None*, *offset=None*, *start=None*, *end=None*)

user_table_counts()

Return table of number of data points per user, per table.

Return a dataframe of row=table, column=user, value=number of counts of that user in that table.

users(*table=None*)

Return set of all users in all tables

validate_username(*user*)

Validate a username, for single/multiuser database and so on.

This function considers if the database is single or multi-user, and ensures a valid username or ALL.

It returns a valid username, so can be used as a wrapper, to handle future special cases, e.g.:

```
user = db.validate_username(user)
```

niimpy.reading.database.**open**(*db*, *tz=None*)

Open a database and return a Data1 object

class niimpy.reading.database.**sqlite3_stdev**

Bases: `object`

Sqlite sample standard deviation function in pure Python.

With *conn.create_aggregate("stdev", 1, sqlite3_stdev)*, this adds a stdev function to sqlite.

Edge cases:

- Empty list = nan (different than C function, which is zero)
- Ignores nan input values (does not count them). (different than numpy: returns nan)
- ignores non-numeric types (no conversion)

Methods

finalize	
step	

finalize()

step(*value*)

niimpy.reading.read module

Read data from various formats, user entery point.

This module contains various functions *read_** which load data from different formats into pandas.DataFrame:s. As a side effect, it provides the authoritative information on how incoming data is converted to dataframes.

`niimpy.reading.read.`**`read_csv`**(*filename*, *read_csv_options={}*, *add_group=None*, *tz=None*)

Read DataFrame from csv file

This will read data from a csv file and then process the result with *niimpy.util.df_normalize*.

Parameters

filename
[str] filename of csv file

read_csv_options: dict
Dictionary of options to pandas.read_csv, if this is necessary for custom csv files.

add_group
[object] If given, add a 'group' column with all values set to this.

`niimpy.reading.read.`**`read_csv_string`**(*string*, *tz=None*)

Parse a string containing CSV and return dataframe

This should not be used for serious reading of CSV from disk, but can be useful for tests and examples. Various CSV reading options are turned on in order to be better for examples:

- Allow comments in the CSV file
- Remove the *datetime* column (redundant with *index* but some older functions break without it, so default readers need to leave it).

Parameters

string
[string containing CSV file]

Returns

df: pandas.DataFrame

`niimpy.reading.read.`**`read_sqlite`**(*filename*, *table*, *add_group=None*, *user=<class 'niimpy.reading.database.ALL'>*, *limit=None*, *offset=None*, *start=None*, *end=None*, *tz=None*)

Read DataFrame from sqlite3 database

This will read data from a sqlite3 file, taking sensor data in a given table, and optionally apply various limits.

Parameters

filename
[str] filename of sqlite3 database

table
[str] table name of data within the database

add_group
[object] If given, add a 'group' column with all values set to this.

user
: [str or database.ALL, optional] If given, return only data matching this user (based an column 'user')

limit
: [int, optional] If given, return only this many rows

offset
: [int, optional] When used with limit, skip this many lines at the beginning

start
: [int or float or str or datetime.datetime, optional] If given, limit to this starting time. Formats can be int/float (unixtime), string (parsed with dateutil.parser.parser, or datetime.datetime.

end
: [int or float or str or datetime.datetime, optional] Same meaning as 'start', but for end time

`niimpy.reading.read.`**`read_sqlite_tables`**`(`*filename*`)`

Return names of all tables in this database

Return a set of all tables contained in this database. This may be useful when you need to see what data is available within a database.

Module contents

9.1.2 Submodules

niimpy.demo module

9.1.3 Module contents

CHAPTER

TEN

DEMO NOTEBOOK FOR NIIMPY EXPLORATION LAYER MODULES

10.1 Introduction

To study and quantify human behavior using longitudinal multimodal digital data, it is essential to get to know the data well first. These data from various sources or sensors, such as smartphones and watches and activity trackers, yields data with different types and properties. The data may be a mixture of categorical, ordinal and numerical data, typically consisting of time series measured for multiple subjetcs from different groups. While the data is typically dense, it is also heterogenous and contains lots of missing values. Therefore, the analysis has to be conducted on many different levels.

This notebook introduces the Niimpy toolbox exploration module, which seeks to address the aforementioned issues. The module has functionalities for exploratory data analysis (EDA) of digital behavioral data. The module aims to produce a summary of the data characteristics, inspecting the structures underlying the data, to detecting patterns and changes in the patterns, and to assess the data quality (e.g., missing data, outliers). This information is highly essential for assessing data validity, data filtering and selection, and for data preprocessing. The module includes functions for *plotting catogorical data*, *data counts*, *timeseries lineplots*, *punchcards* and *visualizing missing data.*

Exploration module functions are supposed to run after data preprocessing, but they can be run also on the raw observations. All the functions are implemented by using Plotly Python Open sourde Library. Plotly enables interactive visualizations which in turn makers it easier to explore different aspects of the data (e.g.,specific timerange and summary statistics).

This notebook uses several sample dataframes for module demonstration. The sample data is already preprocessed, or will be preprocessed in notebook sections before visualizations. When the sample data is loaded, some of the key characteristics of the data are displayed.

All eploration module functions require the data to follow data schema. defined in the Niimpy toolbox documentation. The user must ensure that the input data follows the specified schema.

10.1.1 Sub-module overview

The following table shows accepted data types, visualization functions and the purpose of each exploration sub-module. All submodules are located inside `niimpy/exploration/eda` -folder.

Sub-module	Data type	Functions	For what
catogorical.py	Categorical	Barplot	Observations counts and distributions
countplot.py	Categorical* / Numerical	Barplot/Boxplot	Observation counts and distibutions
lineplot.py	Numerical	Lineplot	Trend, cyclicity, patterns
punchcard.py	Categorical* / Numerical	Heatmap	Temporal patterns of counts or values
missingness.py	Categorical / Numerical	Barplot / Heatmap	Missing data patterns

Data types denoted with * are not compatible with every function within the module. *** ### *NOTES*

This notebook uses following definitions referring to data: * *Feature* refers to dataframe column that stores observations (e.g., numerical sensor values, questionnaire answers) * *User* refers to unique identifier for each subject in the data. Dataframe should also have a column named as `user`. * *Group* refers to unique group idenfier. If subjects are grouped, dataframe shoudl have a column named as `group`.

10.1.2 Imports

Here we import modules needed for running this notebook.

[1]:
```
import numpy as np
import pandas as pd
import plotly
import plotly.graph_objects as go
import plotly.express as px
import plotly.io as pio
import warnings
warnings.filterwarnings("ignore")
import niimpy
from niimpy import config
from niimpy.preprocessing.survey import survey_convert_to_numerical_answer, survey_print_
↪statistic
from niimpy.preprocessing.survey import PHQ2_MAP, PSQI_MAP, PSS10_MAP, PANAS_MAP, GAD2_
↪MAP, ID_MAP_PREFIX
from niimpy.exploration import setup_dataframe
from niimpy.exploration.eda import categorical, countplot, lineplot, missingness,␣
↪punchcard
```

10.1.3 Plotly settings

Next code block defines default settings for plotly visualizations. Feel free to adjust the settings according to your needs.

[2]:
```
pio.renderers.default = "png"
pio.templates.default = "seaborn"
px.defaults.template = "ggplot2"
px.defaults.color_continuous_scale = px.colors.sequential.RdBu
px.defaults.width = 1200
px.defaults.height = 482
warnings.filterwarnings("ignore")
```

10.2 1) Categorical plot

This section introduces categorical plot module visualizes **categorical data**, such as questionnaire data responses. We will demonstrate functions by using a mock survey dataframe, containing answers for: * *Patient Health Questionnaire-2 (PHQ-2)* * *Perceived Stress Scale (PSS10)* * *Generalized Anxiety Disorder-2 (GAD-2)*

The data will be preprocessed, and then it's basic characteristics will be summarized before visualizations.

10.3 1.1) Reading the data

We'll start by importing the data:

[3]:
```
df = niimpy.read_csv(config.SURVEY_PATH, tz='Europe/Helsinki')
df.head()
```

[3]:
```
   user  age  gender Little interest or pleasure in doing things.  \
0     1   20    Male                                 several-days
1     2   32    Male                      more-than-half-the-days
2     3   15    Male                      more-than-half-the-days
3     4   35  Female                                   not-at-all
4     5   23    Male                      more-than-half-the-days

  Feeling down; depressed or hopeless. Feeling nervous; anxious or on edge.  \
0              more-than-half-the-days                           not-at-all
1              more-than-half-the-days                           not-at-all
2                           not-at-all                         several-days
3                    nearly-every-day                           not-at-all
4                           not-at-all              more-than-half-the-days

  Not being able to stop or control worrying.  \
0                            nearly-every-day
1                                several-days
2                                  not-at-all
3                                several-days
4                                several-days

  In the last month; how often have you felt that you were unable to control the␣
↪important things in your life?  \
0                                       almost-never
1                                              never
2                                              never
3                                         very-often
4                                       almost-never

  In the last month; how often have you felt confident about your ability to handle your␣
↪personal problems?  \
0                                          sometimes
1                                              never
2                                         very-often
3                                       fairly-often
4                                         very-often
```

(continues on next page)

(continued from previous page)

```
  In the last month; how often have you felt that things were going your way?  \
0                                      fairly-often
1                                        very-often
2                                        very-often
3                                        very-often
4                                      almost-never

  In the last month; how often have you been able to control irritations in your life?  \
0                                             never
1                                         sometimes
2                                      fairly-often
3                                             never
4                                         sometimes

  In the last month; how often have you felt that you were on top of things?  \
0                                         sometimes
1                                             never
2                                             never
3                                         sometimes
4                                         sometimes

  In the last month; how often have you been angered because of things that were outside␣
↪of your control?  \
0                                        very-often
1                                      fairly-often
2                                             never
3                                             never
4                                        very-often

  In the last month; how often have you felt difficulties were piling up so high that␣
↪you could not overcome them?
0                                      fairly-often
1                                             never
2                                      almost-never
3                                      fairly-often
4                                             never
```

Then check some basic descriptive statistics:

[4]:
```
df.describe()
```

[4]:
```
              user          age
count  1000.000000  1000.000000
mean    500.500000    26.911000
std     288.819436     4.992595
min       1.000000    12.000000
25%     250.750000    23.000000
50%     500.500000    27.000000
75%     750.250000    30.000000
max    1000.000000    43.000000
```

The dataframe's columns are raw questions from a survey. Some questions belong to a specific category, so we will annotate them with ids. The id is constructed from a prefix (the questionnaire category: GAD, PHQ, PSQI etc.), followed by the question number (1,2,3). Similarly, we will also the answers to meaningful numerical values.

Note: It's important that the dataframe follows the below schema before passing into niimpy.

Next, we'll convert the column names to question ID's using predefined maps (python dictionaries) imported from `survey.py` module. Then, we'll transform the data from long to wide format. Finally we'll add a column with id's matching the questions.

```
[5]: # Convert column name to id, based on provided mappers from niimpy
col_id = {**PHQ2_MAP, **PSQI_MAP, **PSS10_MAP, **PANAS_MAP, **GAD2_MAP}
selected_cols = [col for col in df.columns if col in col_id.keys()]

# Convert data frame to long format
m_df = pd.melt(df, id_vars=['user', 'age', 'gender'], value_vars=selected_cols, var_name=
↪'question', value_name='answer')

# Assign questions to codes
m_df['id'] = m_df['question'].replace(col_id)
m_df.head()
```

```
[5]:    user  age  gender                                    question  \
0     1   20    Male  Little interest or pleasure in doing things.
1     2   32    Male  Little interest or pleasure in doing things.
2     3   15    Male  Little interest or pleasure in doing things.
3     4   35  Female  Little interest or pleasure in doing things.
4     5   23    Male  Little interest or pleasure in doing things.

                    answer      id
0             several-days  PHQ2_1
1  more-than-half-the-days  PHQ2_1
2  more-than-half-the-days  PHQ2_1
3               not-at-all  PHQ2_1
4  more-than-half-the-days  PHQ2_1
```

We can use a helper method to convert the answers into numerical value. The pre-defined mapper inside survey.py would be useful for this step. Since all questionaires havin PHQ and PSS prefix in their name, use similar mappings from categorical answer into numerical, we can use `ID_MAP_PREFIX` mapper that converts all the questionaires at same time.

```
[6]: # Transform raw answers to numerical values
m_df['answer'] = survey_convert_to_numerical_answer(m_df,
                                                     answer_col='answer',
                                                     question_id='id',
                                                     id_map=ID_MAP_PREFIX,
                                                     use_prefix=True)
m_df.head()
```

```
[6]:    user  age  gender                                    question answer  \
0     1   20    Male  Little interest or pleasure in doing things.      1
1     2   32    Male  Little interest or pleasure in doing things.      2
2     3   15    Male  Little interest or pleasure in doing things.      2
3     4   35  Female  Little interest or pleasure in doing things.      0
4     5   23    Male  Little interest or pleasure in doing things.      2

       id
0  PHQ2_1
```

(continues on next page)

(continued from previous page)

```
1   PHQ2_1
2   PHQ2_1
3   PHQ2_1
4   PHQ2_1
```

We can also produce a summary of the questionaire's score. This function can describe aggregated score over the whole population, or specific subgroups.

First we'll show statistics for the whole population:

[7]:
```
d1 = survey_print_statistic(m_df)
pd.DataFrame(d1)
```

[7]:

	PHQ2	GAD2	PSS10
min	0.0000	0.000000	4.000000
max	6.0000	6.000000	27.000000
avg	3.0520	3.042000	14.006000
std	1.5855	1.536423	3.687759

Statistics by the group gender:

[8]:
```
d2 = survey_print_statistic(m_df, group='gender')
pd.DataFrame(d2)
```

[8]:

	PHQ2		GAD2		PSS10	
	Female	Male	Female	Male	Female	Male
min	0.000000	0.000000	0.000000	0.000000	4.000000	4.000000
max	6.000000	6.000000	6.000000	6.000000	27.000000	23.000000
avg	3.067210	3.037328	3.087576	2.998035	14.059063	13.954813
std	1.605337	1.567567	1.585157	1.488141	3.783230	3.596247

And finally statistics for PHQ questionnaires by group:

[9]:
```
d3 = survey_print_statistic(m_df, group='gender', prefix='PHQ')
pd.DataFrame(d3)
```

[9]:

	PHQ	
	Female	Male
min	0.000000	0.000000
max	6.000000	6.000000
avg	3.067210	3.037328
std	1.605337	1.567567

10.4 1.1. Questionnaire summary

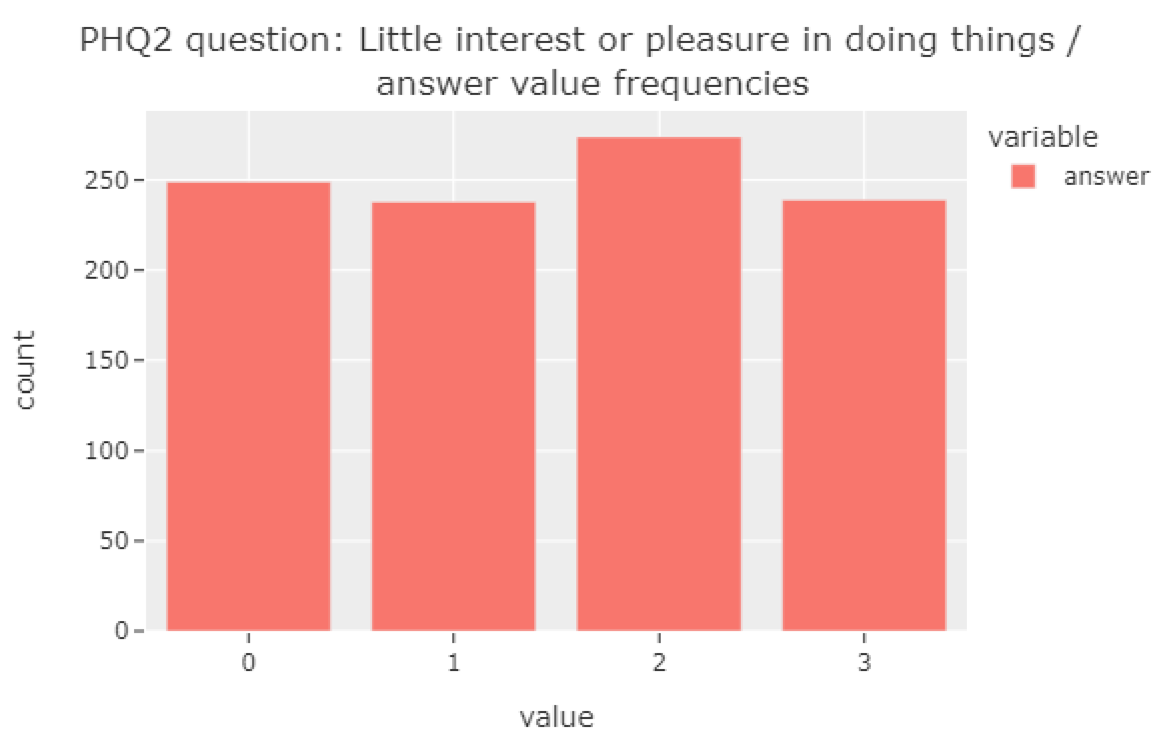
We can now make some plots for the preprocessed data frame. First, we can display the summary for the specific question (*PHQ-2* first question).

[10]:
```
fig = categorical.questionnaire_summary(m_df,
                                        question = 'PHQ2_1',
                                        column = 'answer',
                                        title='PHQ2 question: Little interest or␣
```

(continues on next page)

(continued from previous page)

```
↪pleasure in doing things / <br> answer value frequencies',
                                        xlabel='value',
                                        ylabel='count',
                                        width=600,
                                        height=400)
fig.show()
```

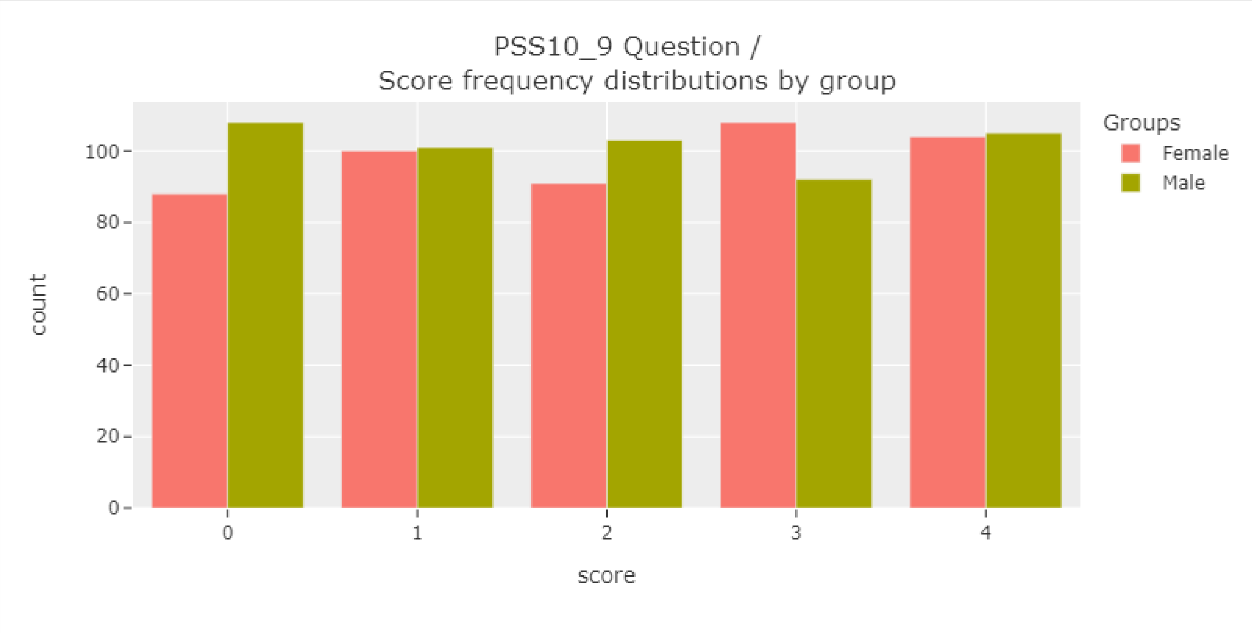


The figure shows that the answer values (from 0 to 3) almost uniform in distribution.

10.5 1.2. Questionnaire grouped summary

We can also display the summary for each subgroup (*gender*).

```
[11]: fig = categorical.questionnaire_grouped_summary(m_df,
                                                question='PSS10_9',
                                                group='gender',
                                                title='PSS10_9 Question / <br> Score␣
↪frequency distributions by group',
                                                xlabel='score',
                                                ylabel='count',
                                                width=800,
                                                height=400)
fig.show()
```



The figure shows that the differences between subgroups are not very large.

10.6 1.3. Questionnaire grouped summary score distribution

With some quick preprocessing, we can display the score distribution of each questionaire.

We'll extract *PSS-10* questionnaire answers from the dataframe, group the data by user and gender, and aggregate the answer scores.

[12]:
```
pss_sum_df = m_df[m_df['id'].str.startswith('PSS')] \
                            .groupby(['user', 'gender']) \
                            .agg({'answer':sum}) \
                            .reset_index()
pss_sum_df['id'] = 'PSS'
```

We'll quickly inspect the preprocessed dataframe.

[13]:
```
pss_sum_df
```

[13]:
```
     user  gender answer   id
0       1    Male     15  PSS
1       2    Male      9  PSS
2       3    Male     12  PSS
3       4  Female     16  PSS
4       5    Male     14  PSS
..    ...     ...    ...  ...
995   996  Female     17  PSS
996   997  Female     13  PSS
997   998    Male     13  PSS
998   999    Male     21  PSS
999  1000    Male     14  PSS
```

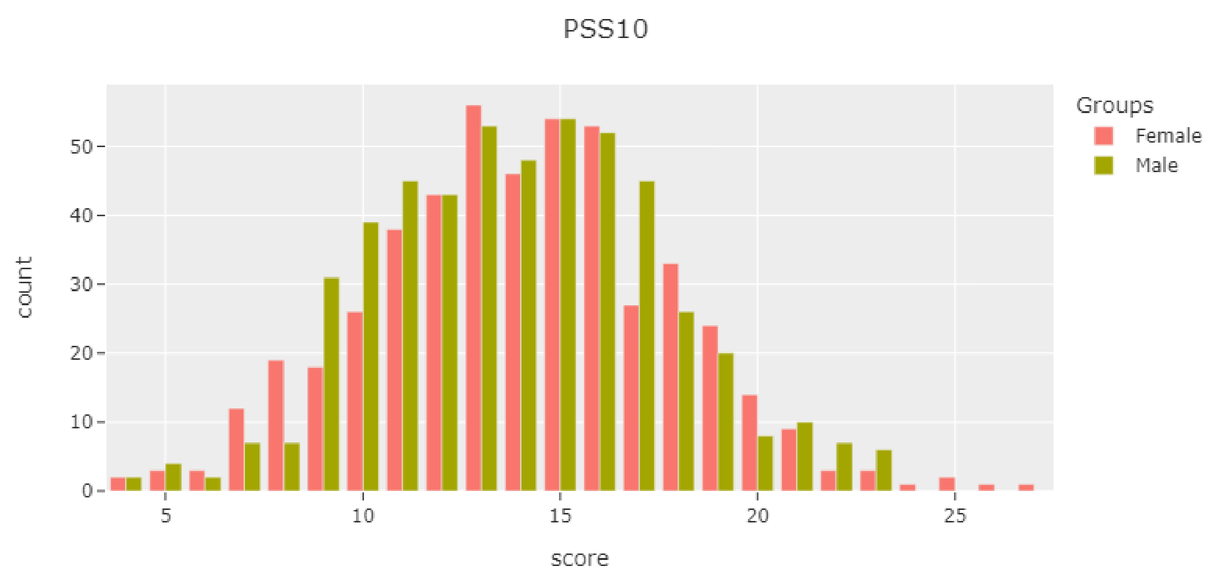
(continues on next page)

(continued from previous page)

```
[1000 rows x 4 columns]
```

And then visualize aggregated summary score distributions, grouped by gender:

[14]:
```
fig = categorical.questionnaire_grouped_summary(pss_sum_df,
                                                question='PSS',
                                                group='gender',
                                                title='PSS10',
                                                xlabel='score',
                                                ylabel='count',
                                                width=800,
                                                height=400)
fig.show()
```



The figure shows that the grouped summary score distrubutions are close to each other.

10.7 2) Countplot

This section introduces Countplot module. The module contain functions for user and group level observation count (number of datapoints per user or group) visualization and observation value distributions. Observation counts use barplots for user level and a boxplots for group level visualizations. Boxplots are used for group level value distributions. The module assumes that the visualized **data is numerical**.

10.7.1 Data

We will use sample from StudentLife dataset to demonstrate the module functions. The sample contains hourly aggregated activity data (values from 0 to 5, where 0 corresponds to no activity, and 5 to high activity) and group information based on pre- and post-study PHQ-9 test scores. Study subjects have been grouped by the depression symptom severity into groups: *none*, *mild*, *moderate*, *moderately severe*, and *severe*. Preprocessed data sample is included in the Niimpy toolbox *sampledata* folder.

[15]:
```
# Load data
sl = niimpy.read_csv(config.SL_ACTIVITY_PATH, tz='Europe/Helsinki')
sl.set_index('timestamp',inplace=True)
sl.index = pd.to_datetime(sl.index)
sl_loc = sl.tz_localize(None)
```

[16]:
```
sl_loc.head()
```

[16]:
```
                    user  activity group
timestamp
2013-03-27 06:00:00  u00         2  none
2013-03-27 07:00:00  u00         1  none
2013-03-27 08:00:00  u00         2  none
2013-03-27 09:00:00  u00         3  none
2013-03-27 10:00:00  u00         4  none
```

Before visualizations, we'll inspect the data.

[17]:
```
sl_loc.describe()
```

[17]:
```
           activity
count  55907.000000
mean       0.750264
std        1.298238
min        0.000000
25%        0.000000
50%        0.000000
75%        1.000000
max        5.000000
```

[18]:
```
sl_loc.group.unique()
```

[18]:
```
array(['none', 'severe', 'mild', 'moderately severe', 'moderate'],
      dtype=object)
```

10.8 2.1. User level observation count

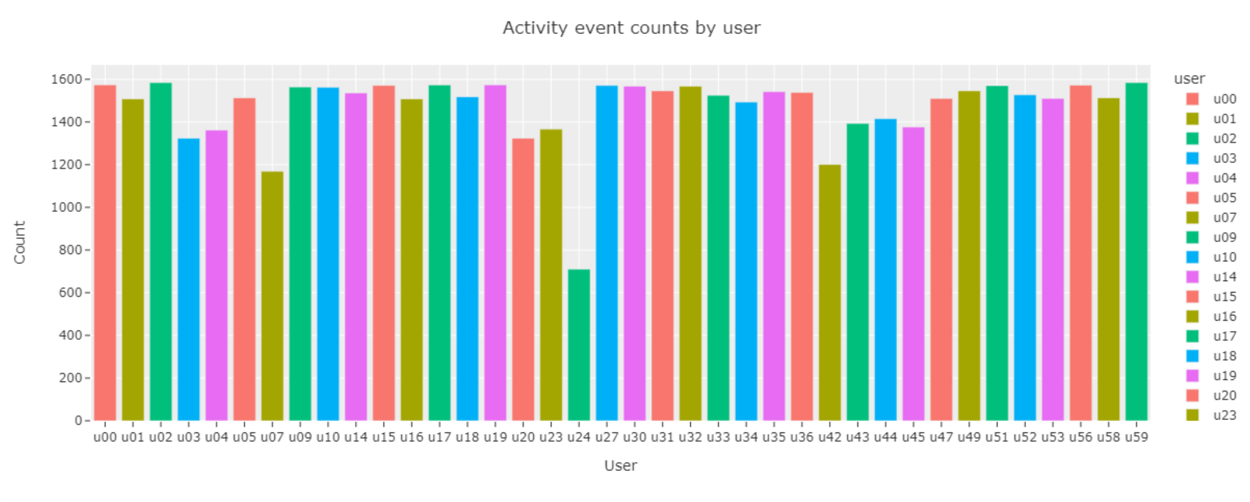
At first we visualize the number of observations for each subject.

[19]:
```
fig = countplot.countplot(sl,
                          fig_title='Activity event counts by user',
                          plot_type='count',
                          points='all',
                          aggregation='user',
```

(continues on next page)

(continued from previous page)

```
                          user=None,
                          column=None,
                          binning=False)

fig.show()
```

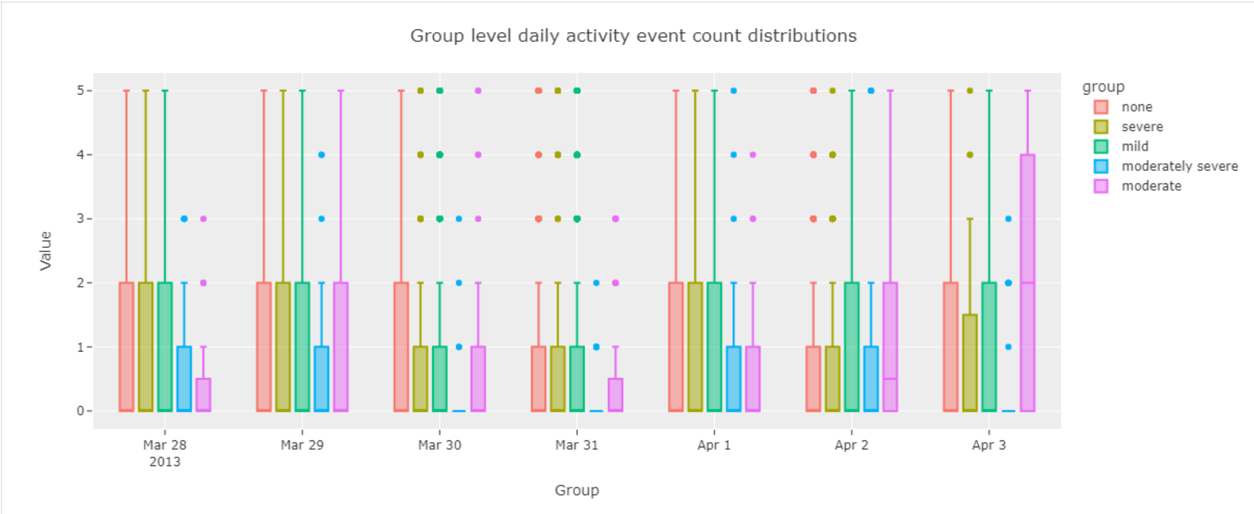


The barplot shows that there are differences in user total activity counts. The user *u24* has the lowest event count of 710 and users *u02* and *u59* have the highest count of 1584.

10.9 2.2. Group level observation count

Next we'll inspect group level daily activity event count distributions by using boxplots. For the improved clarity, we select a timerange of one week from the data.

[20]:
```
sl_one_week = sl_loc.loc['2013-03-28':'2013-4-3']

fig = countplot.countplot(sl_one_week,
                          fig_title='Group level daily activity event count distributions
↪',
                          plot_type='value',
                          points='all',
                          aggregation='group',
                          user=None,
                          column='activity',
                          binning='D')

fig.show()
```

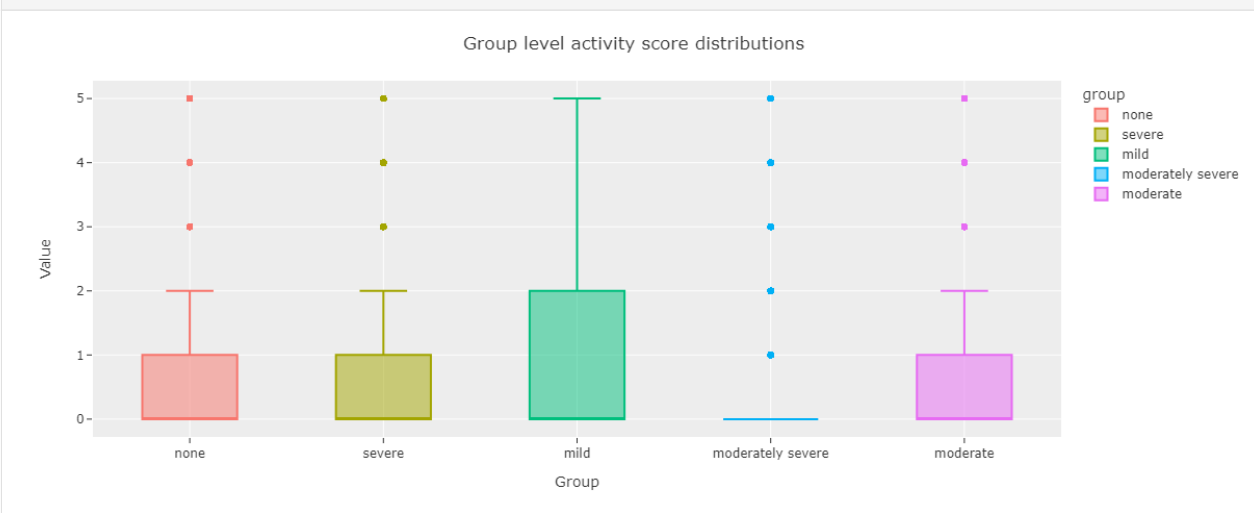


The boxplot shows some variability in group level event count distributions across the days spanning from Mar 28 to Apr 3 2013.

10.10 2.3. Group level value distributions

Finally we visualize group level activity value distributions for whole time range.

[21]:
```
fig = countplot.countplot(sl,
                         fig_title='Group level activity score distributions',
                         plot_type='value',
                         points='outliers',
                         aggregation='group',
                         user=None,
                         column='activity',
                         binning=False)

fig.show()
```



The boxplot shows that activity score distribution for groups *mild* and *moderately severe* differ from the rest.

10.11 3. Lineplot

This section introduces Lineplot module functions. We use the same StudentLife dataset derived activity data as in previous section.

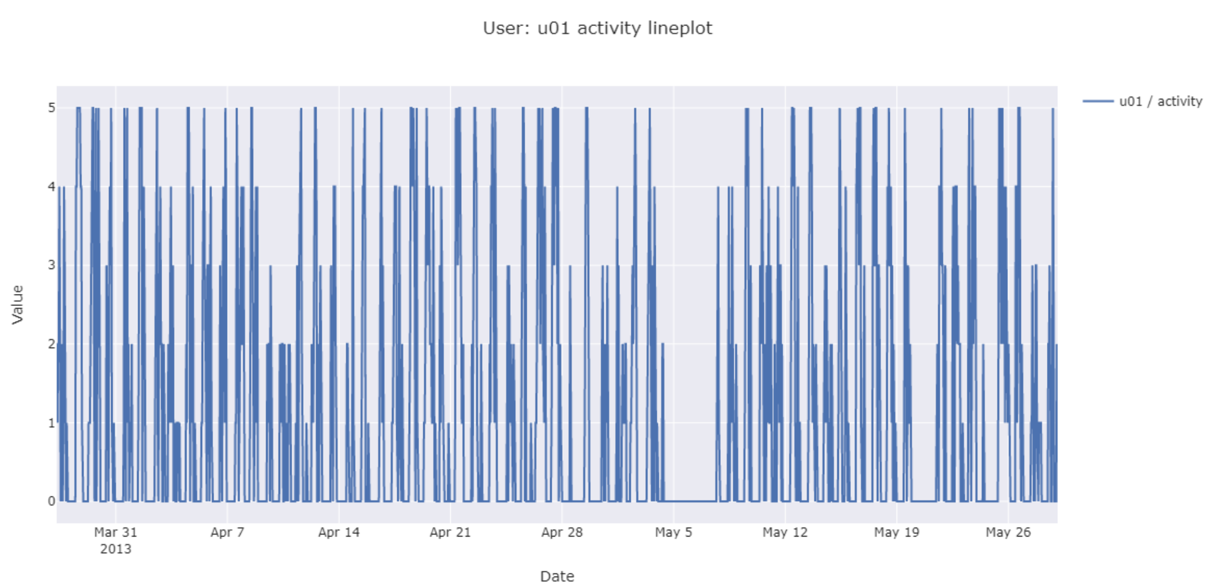
10.12 3.1. Lineplot

Lineplot functions display **numerical feature values** on time axis. The user can optionally resample (downsample) and smoothen the data for better visual clarity.

10.13 3.1.1. Single user single feature

At first, we'll visualize single user single feature data, without resampling or smoothing.

[22]:
```
fig = lineplot.timeplot(sl_loc,
                        users=['u01'],
                        columns=['activity'],
                        title='User: {} activity lineplot'.format('u01'),
                        xlabel='Date',
                        ylabel='Value',
                        resample=False,
                        interpolate=False,
                        window=1,
                        reset_index=False)

fig.show()
```

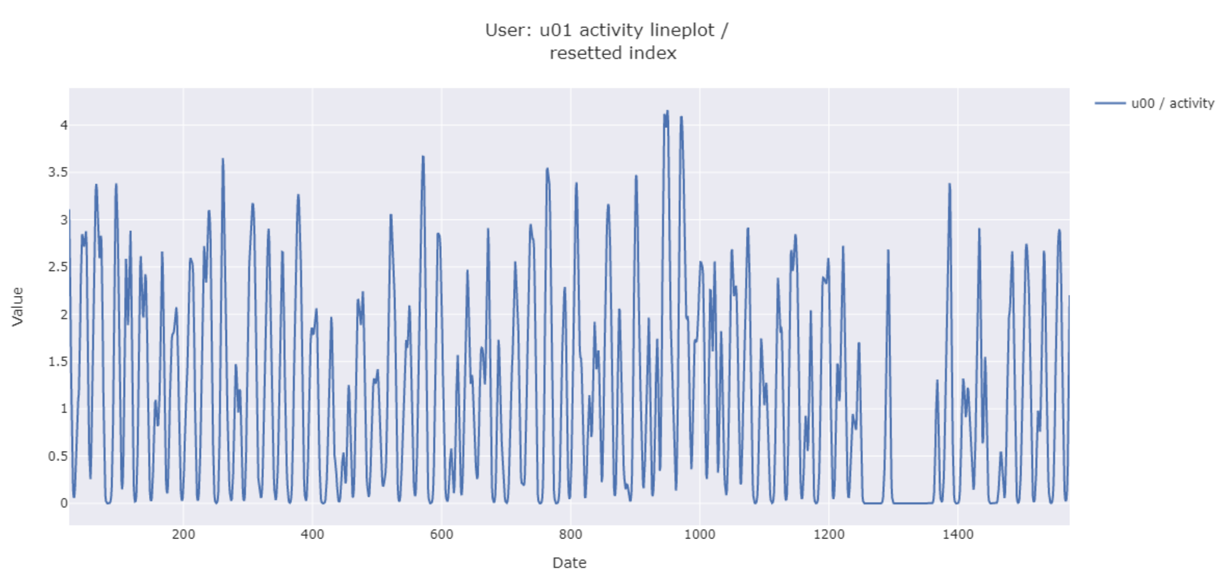


The figure showing all the activity datapoints is difficult to interpet. By zooming in the time range, the daily patters come apparent. There is no or low activity during the night.

10.14 3.1.2. Single user single feature index resetted

Next, we'll plot visualize the same data using resampling by hour, and 24 hour rolling window smoothing for improved visualization clarity. We also reset the index, showing now hours from the first activity feature observation.

[23]:
```
fig = lineplot.timeplot(sl_loc,
                        users=['u00'],
                        columns=['activity'],
                        title='User: {} activity lineplot / <br> resetted index'.format(
↪'u01'),
                        xlabel='Date',
                        ylabel='Value',
                        resample='H',
                        interpolate=True,
                        window=24,
                        reset_index=True)

fig.show()
```



By zooming in the smoothed lineplot, daily activity patterns are easier to detect.

10.15 3.1.3. Single user single feature, aggregated by day

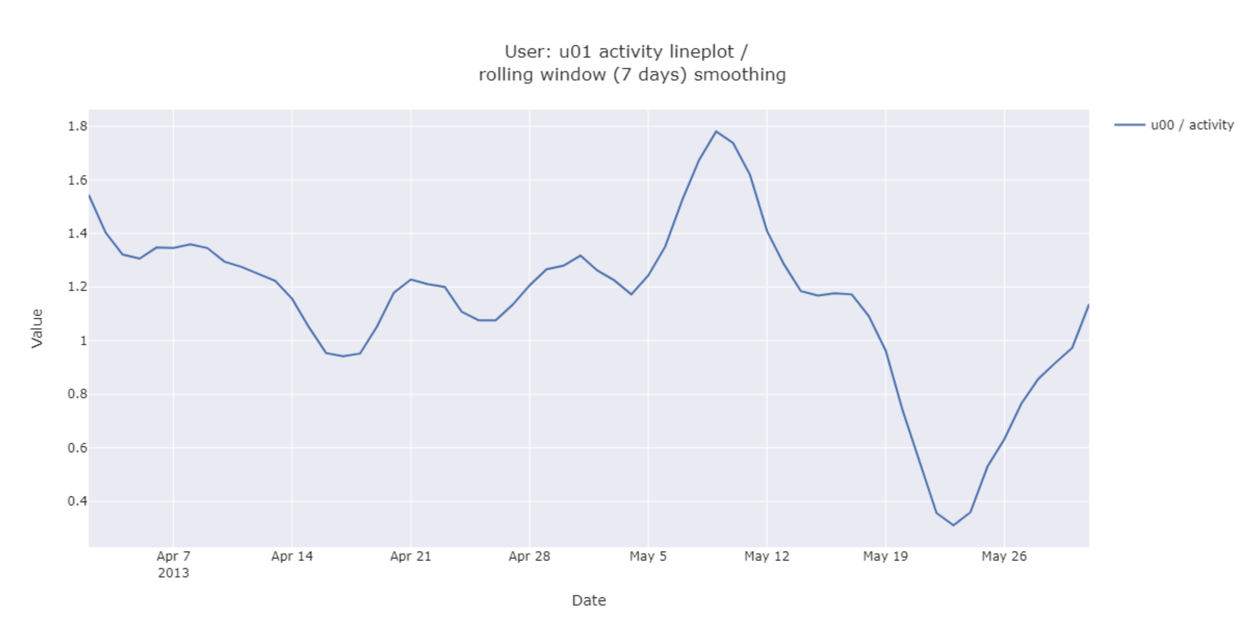
Next visualization shows resamplig by day and 7 day rolling window smoothing, making the activity time series trend visible.

[24]:
```
fig = lineplot.timeplot(sl_loc,
                        users=['u00'],
                        columns=['activity'],
                        title='User: {} activity lineplot / <br> rolling window (7 days)␣
↪smoothing'.format('u01'),
```

(continues on next page)

(continued from previous page)

```
                        xlabel='Date',
                        ylabel='Value',
                        resample='D',
                        interpolate=True,
                        window=7)

fig.show()
```

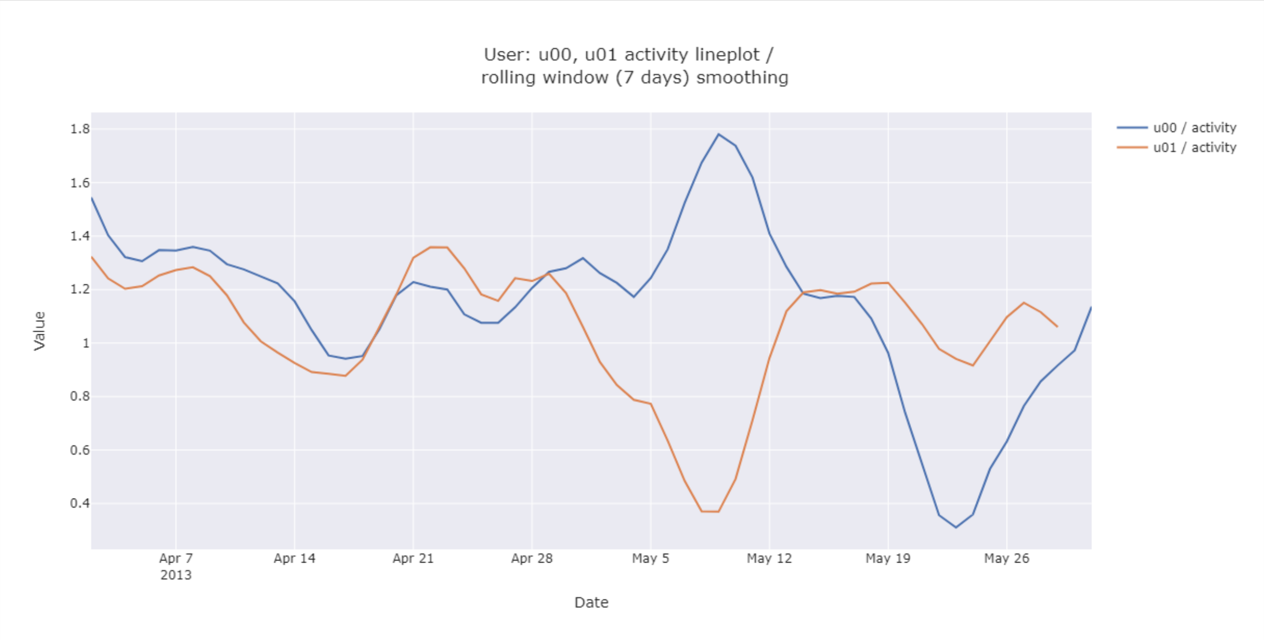


Daily aggregated and smoothed data makes the user activity trend visible. There is a peak at May 9 and the crest at May 23.

10.16 3.2. Multiple subjects single feature

The following visualization superimposes three subject's activity on same figure.

```
[25]: fig = lineplot.timeplot(sl_loc,
                        users=['u00','u01'],
                        columns=['activity'],
                        title='User: {}, {} activity lineplot / <br> rolling window (7␣
↪days) smoothing'.format('u00','u01'),
                        xlabel='Date',
                        ylabel='Value',
                        resample='D',
                        interpolate=True,
                        window=7)

fig.show()
```

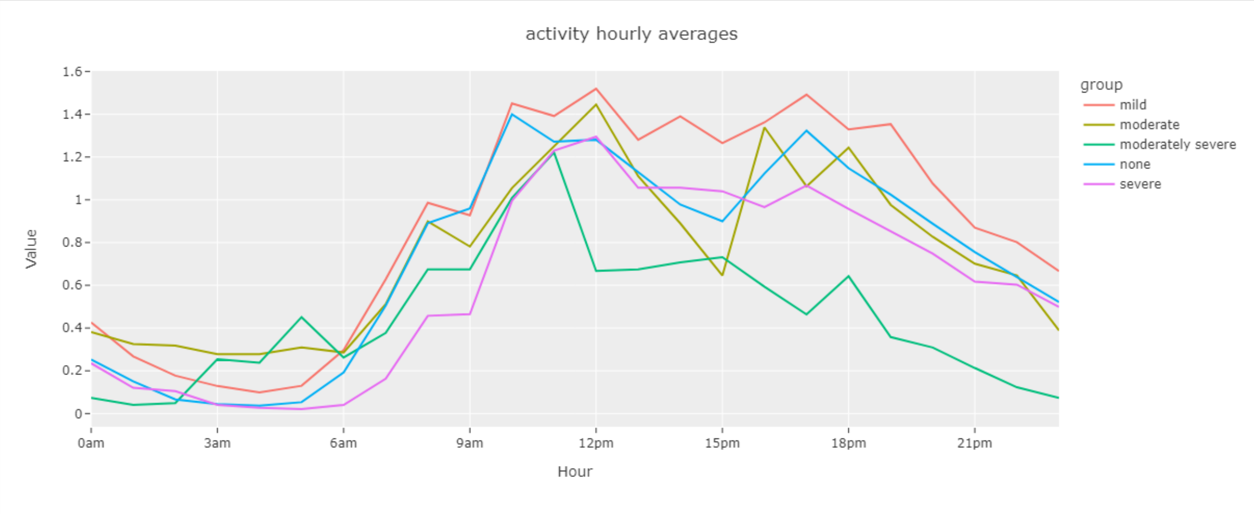


The figure shows that the user daily averaged activity is quite similar in the beginning of inspected time range. In first two weeks of May, the activity shows opposing trends (user *u00* activity increases and user *u01* decreases).

10.17 3.3. Group level hourly averages

Next we'll compare group level hourly average activity.

[26]:
```
fig = lineplot.timeplot(sl_loc,
                        users='Group',
                        columns=['activity'],
                        title='User group activity / <br> hourly averages',
                        xlabel='Date',
                        ylabel='Value',
                        resample='D',
                        interpolate=True,
                        window=7,
                        reset_index=False,
                        by='hour')

fig.show()
```

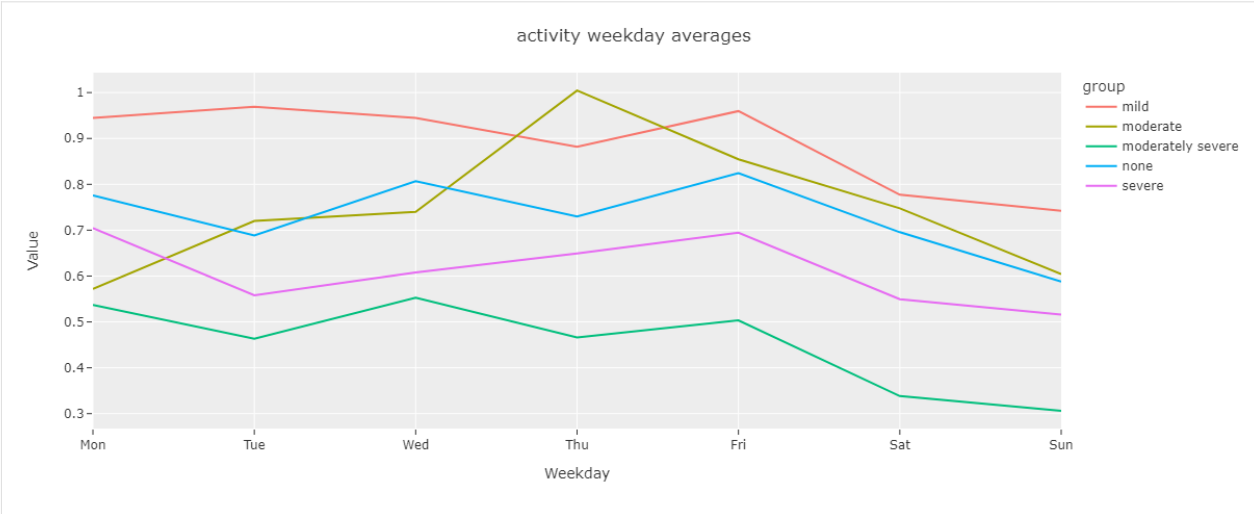


The time plot reveals that the hourly averaged group level activity follows circadian rhytmn (less activity during the night). Moderately severe group seems to be least active group during the latter half of the day.

10.18 3.4. Group level weekday averages

And finally,

[27]:
```
fig = lineplot.timeplot(sl_loc,
                        users='Group',
                        columns=['activity'],
                        title='User Activity',
                        xlabel='Date',
                        ylabel='Value',
                        resample='D',
                        interpolate=True,
                        window=7,
                        reset_index=False,
                        by='weekday')

fig.show()
```



The timeplot shows that there is some differences between the average group level activity, e.g., group *mild* being more active than *moderately severe*. Additionally, activity during Sundays is at lower level in comparison with weekdays.

10.19 4. Punchcard

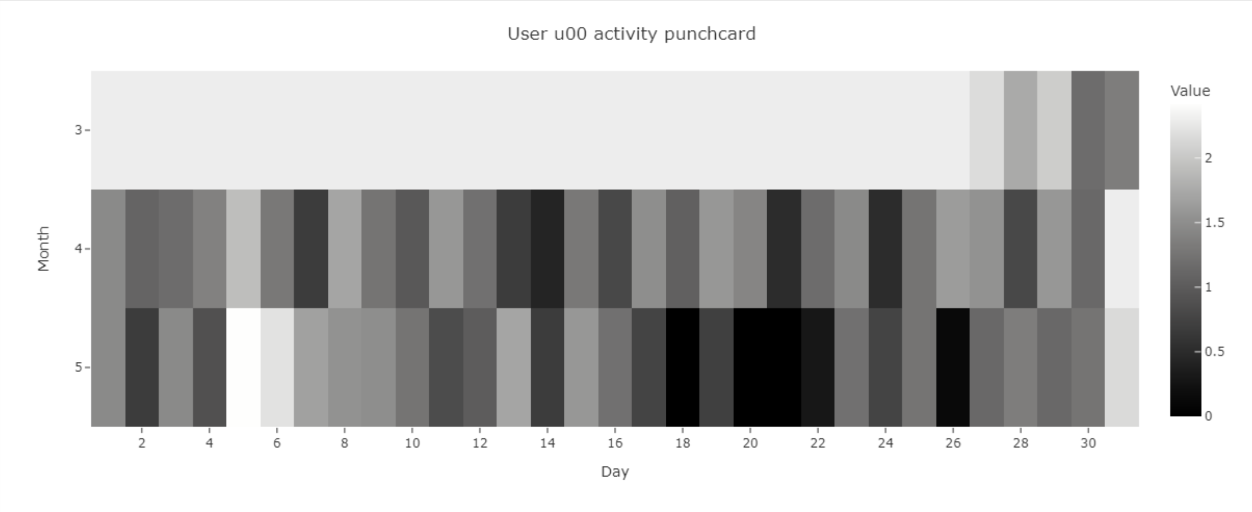
This section introduces Punchcard module functions. The functions aggregate the data and show the averaged value for each timepoint. We use the same StudentLife dataset derived activity data as in two previous sections.

10.20 4.1. Single user punchcard

At first we visualize one daily aggregated mean activity for single subject. We'll change the plot color to grayscale for improved clarity.

[28]:
```
px.defaults.color_continuous_scale = px.colors.sequential.gray
```

[29]:
```
fig = punchcard.punchcard_plot(sl,
                               user_list=['u00'],
                               columns=['activity'],
                               title="User {} activity punchcard".format('u00'),
                               resample='D',
                               normalize=False,
                               agg_func=np.mean,
                               timerange=False)

fig.show()
```

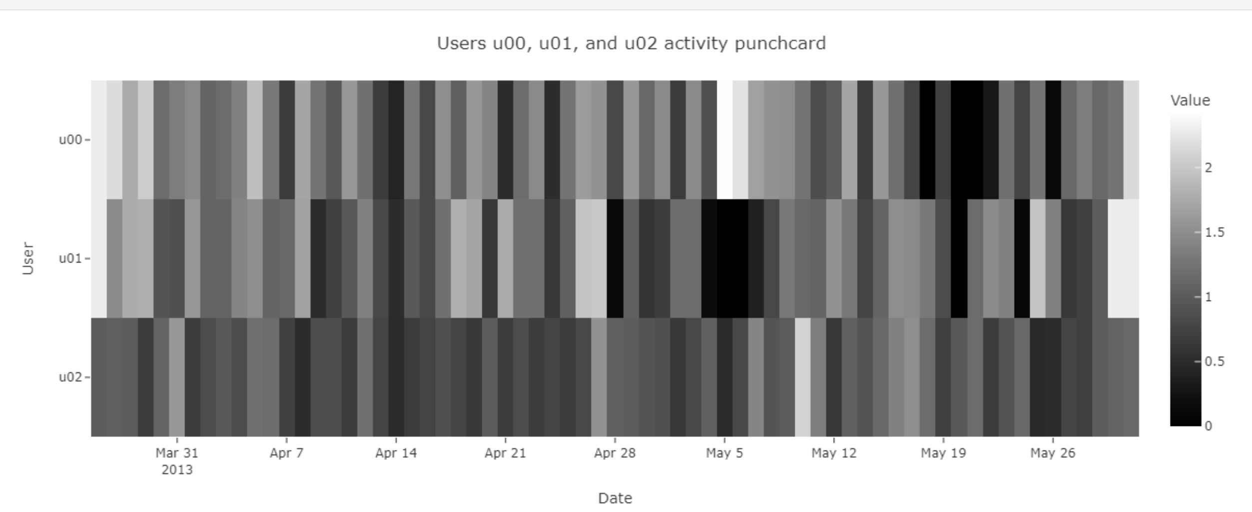


The punchcard reveals that May 5th has the highest average activity and May 18th, 20th, and 21th have the lowest activity.

10.21 4.2. Multiple user punchcard

Next, we'll visualize mean activity for multiple subjects.

[30]:
```
fig = punchcard.punchcard_plot(sl,
                               user_list=['u00','u01','u02'],
                               columns=['activity'],
                               title="Users {}, {}, and {} activity punchcard".format(
↪'u00','u01','u02'),
                               resample='D',
                               normalize=False,
                               agg_func=np.mean,
                               timerange=False)

fig.show()
```



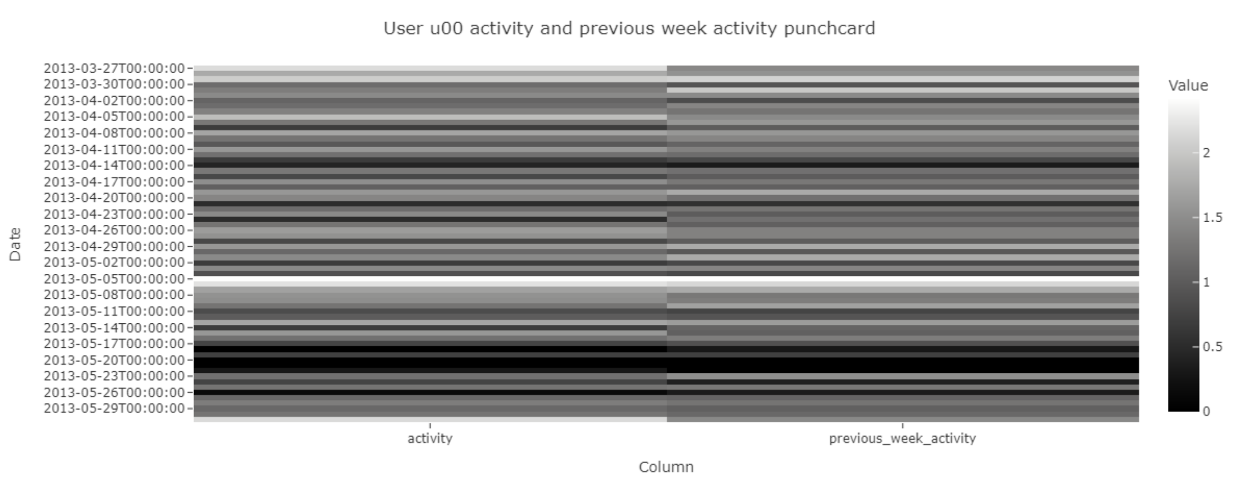
The punchard allows comparison of daily average activity for multiple subjects. It seems that there is not evident common pattern in the activity.

10.22 4.3. Single user punchcard showing two features

Lastly, we'll visualize daily aggregated single user activity side by side with activity of previous week. We start by shifting the activity by one week and by adding it to the original dataframe.

[31]:
```
sl_loc['previous_week_activity'] = sl_loc['activity'].shift(periods=7, fill_value=0)
```

[32]:
```
fig = punchcard.punchcard_plot(sl_loc,
                               user_list=['u00'],
                               columns=['activity','previous_week_activity'],
                               title="User {} activity and previous week activity␣
↪punchcard".format('u00'),
                               resample='D',
                               normalize=False,
                               agg_func=np.mean,
                               timerange=False)

fig.show()
```



The punchcard show weekly repeating patterns in subjects activity.

10.23 5) Missingness

This sections introduces Missingness module for missing data inspection. The module features data missingness visualizations by frequency and by timepoint. Additionally, it offers an option for missing data correlation visualization.

10.23.1 Data

For data missingness visualizations, we'll create a mock dataframe with missing values using `niimpy.util.create_missing_dataframe` function.

[33]:
```
df_m = setup_dataframe.create_missing_dataframe(nrows=2*24*14, ncols=5, density=0.7,␣
↪index_type='dt', freq='10T')
df_m.columns = ['User_1','User_2','User_3','User_4','User_5',]
```

We will quickly inspect the dataframe before the visualizations.

[34]:
```
df_m
```

[34]:
```
                        User_1     User_2     User_3     User_4     User_5
2022-01-01 00:00:00  61.510061        NaN  94.183162        NaN  17.182417
2022-01-01 00:10:00        NaN  79.917067   2.262049        NaN  50.717029
2022-01-01 00:20:00        NaN        NaN  78.738399  62.668739  89.811021
2022-01-01 00:30:00  46.090139  45.456629  89.636218        NaN  84.734977
2022-01-01 00:40:00  67.590468        NaN        NaN  63.255760  52.828918
...                        ...        ...        ...        ...        ...
2022-01-05 15:10:00  57.428122  22.149352  84.623527   5.111538  96.280872
2022-01-05 15:20:00  55.412583        NaN  26.508021  26.090605  49.644855
2022-01-05 15:30:00        NaN  85.720930   7.869486        NaN  80.884746
2022-01-05 15:40:00        NaN  64.156419  19.482492  93.745107  50.000204
2022-01-05 15:50:00  32.042671        NaN  36.168003        NaN  41.869135

[672 rows x 5 columns]
```

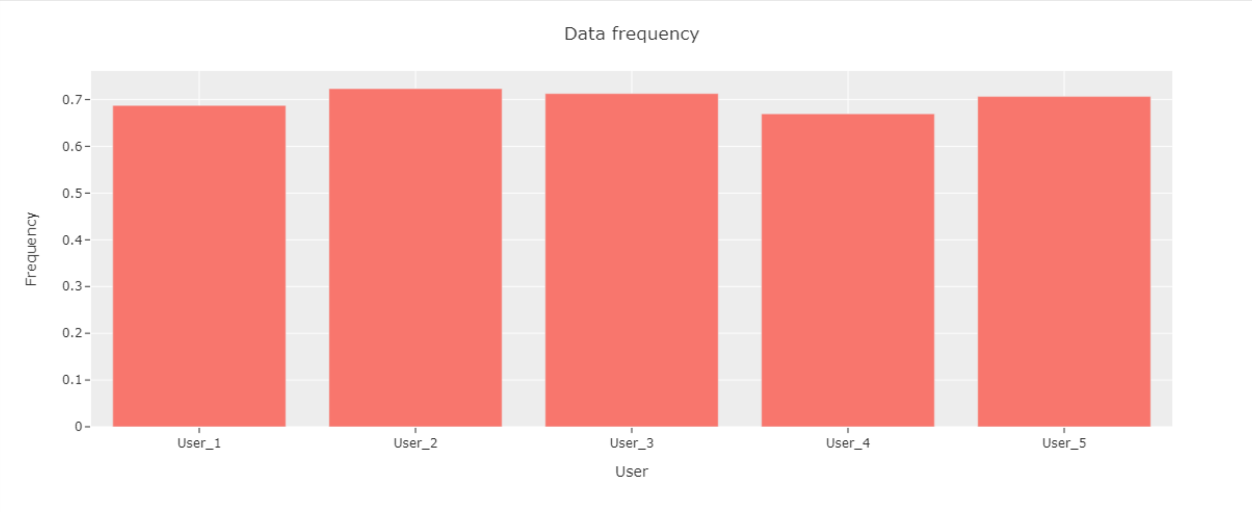
[35]:
```
df_m.describe()
```

[35]:
```
           User_1      User_2      User_3      User_4      User_5
count  462.000000  486.000000  479.000000  450.000000  475.000000
mean    51.094058   51.253925   53.111200   51.446476   51.093412
std     27.747781   28.160343   27.983376   28.974988   28.751183
min      1.277476    1.136281    1.239122    1.272853    1.158063
25%     28.013887   28.529354   29.920174   25.762065   24.852271
50%     53.168911   52.792652   55.010363   51.043262   49.868089
75%     73.824861   75.177829   77.905901   76.606165   76.494873
max     99.723094   99.964264   99.640601   99.791555   99.895162
```

10.24 5.1. Data frequency by feature

First, we create a histogram to visualize data frequency per column. Here, frequency of 1 indicates no missing data points and 0 that all data points are missing.

[36]:
```
fig = missingness.bar(df_m,
                      xaxis_title='User',
                      yaxis_title='Frequency')
fig.show()
```

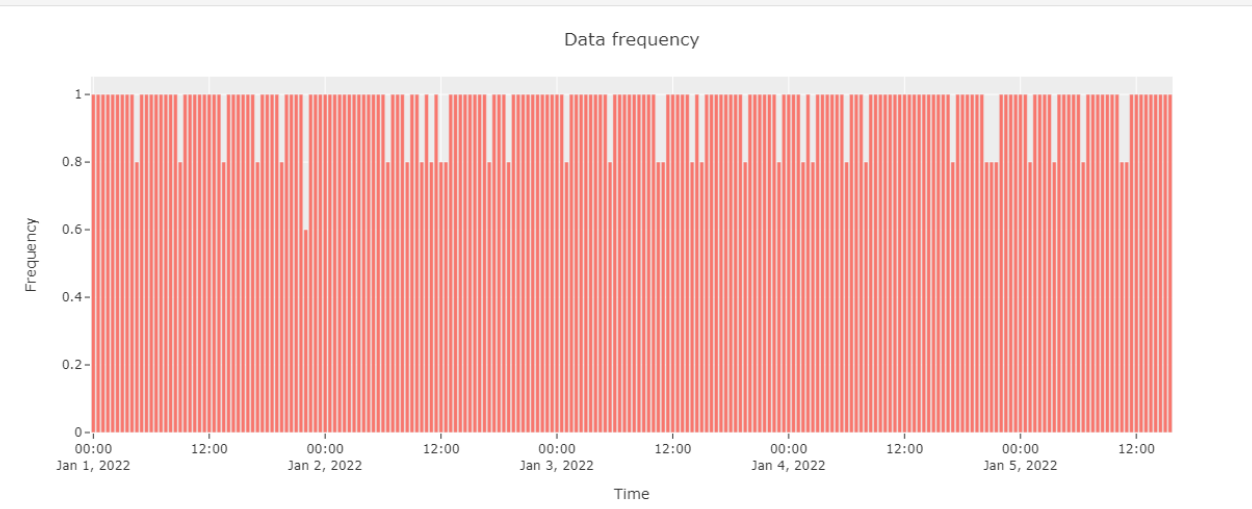


The data frequency is nearly similar for each user, *User_5* having the highest frequency.

10.25 5.2. Average frequency by user

Next, we will show average data frequency for all users.

[37]:
```
fig = missingness.bar(df_m,
                      sampling_freq='30T',
                      xaxis_title='Time',
                      yaxis_title='Frequency')
fig.show()
```

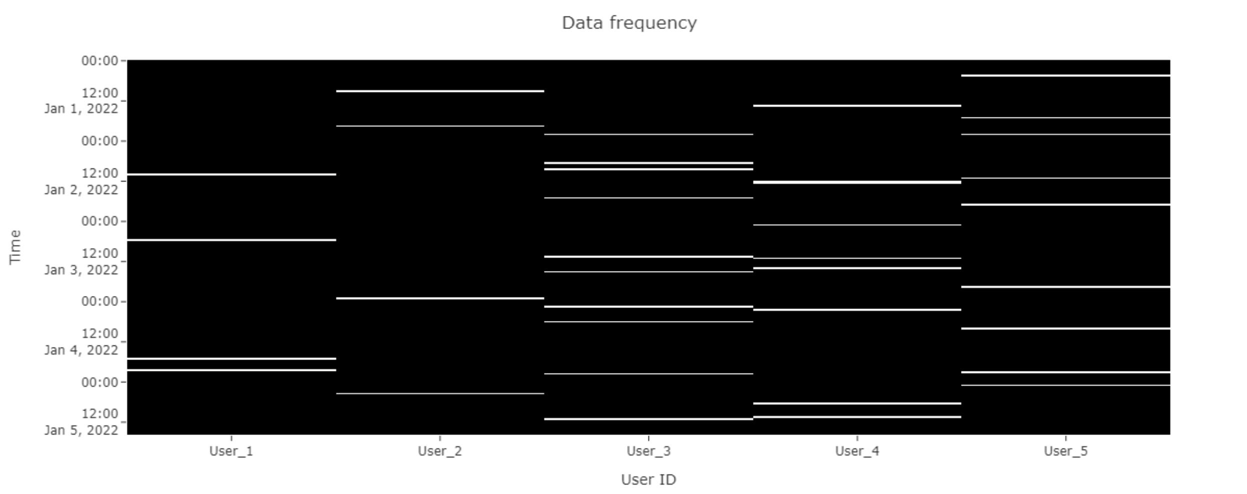


The overall data frequency suggests no clear pattern for data missingness.

10.26 5.3. Missingness matrix

We can also create a missingness matrix visualization for the dataframe. The nullity matrix show data missingess by a timepoint.

[38]:
```
fig = missingness.matrix(df_m,
                         sampling_freq='30T',
                         xaxis_title="User ID",
                         yaxis_title="Time")
fig.show()
```



10.27 5.4. Missing data correlations

Finally, we plot a heatmap to display the correlations between missing data.

Correlation ranges from -1 to 1: * -1 means that if one variable appears then the other will be missing. * 0 means that there is no correlation between the missingness of two variables. * 1 means that the two variables will always appear together.

10.27.1 Data

For the correlations, we use *NYC collision factors* sample data.

[39]:
```
collisions = pd.read_csv("https://raw.githubusercontent.com/ResidentMario/missingno-data/
↪master/nyc_collision_factors.csv")
```

First, we'll inspect the data frame.

[40]:
```
collisions.head()
```

[40]:
```
        DATE      TIME    BOROUGH  ZIP CODE   LATITUDE  LONGITUDE  \
0  11/10/2016  16:11:00   BROOKLYN   11208.0  40.662514 -73.872007
1  11/10/2016  05:11:00  MANHATTAN   10013.0  40.721323 -74.008344
2  04/16/2016  09:15:00   BROOKLYN   11201.0  40.687999 -73.997563
```

(continues on next page)

(continued from previous page)

```
3  04/15/2016  10:20:00     QUEENS   11375.0  40.719228 -73.854542
4  04/15/2016  10:35:00   BROOKLYN   11210.0  40.632147 -73.952731

                  LOCATION  ON STREET NAME CROSS STREET NAME  \
0  (40.6625139, -73.8720068)  WORTMAN AVENUE    MONTAUK AVENUE
1  (40.7213228, -74.0083444)   HUBERT STREET     HUDSON STREET
2  (40.6879989, -73.9975625)    HENRY STREET     WARREN STREET
3  (40.7192276, -73.8545422)             NaN               NaN
4  (40.6321467, -73.9527315)  BEDFORD AVENUE       CAMPUS ROAD

      OFF STREET NAME  ...  CONTRIBUTING FACTOR VEHICLE 1  \
0                 NaN  ...  Failure to Yield Right-of-Way
1                 NaN  ...  Failure to Yield Right-of-Way
2                 NaN  ...             Lost Consciousness
3  67-64 FLEET STREET  ...  Failure to Yield Right-of-Way
4                 NaN  ...  Failure to Yield Right-of-Way

   CONTRIBUTING FACTOR VEHICLE 2  CONTRIBUTING FACTOR VEHICLE 3  \
0                    Unspecified                            NaN
1                            NaN                            NaN
2             Lost Consciousness                            NaN
3  Failure to Yield Right-of-Way  Failure to Yield Right-of-Way
4  Failure to Yield Right-of-Way                            NaN

   CONTRIBUTING FACTOR VEHICLE 4  CONTRIBUTING FACTOR VEHICLE 5  \
0                            NaN                            NaN
1                            NaN                            NaN
2                            NaN                            NaN
3                            NaN                            NaN
4                            NaN                            NaN

   VEHICLE TYPE CODE 1 VEHICLE TYPE CODE 2 VEHICLE TYPE CODE 3  \
0                 TAXI   PASSENGER VEHICLE                 NaN
1    PASSENGER VEHICLE                 NaN                 NaN
2    PASSENGER VEHICLE                 VAN                 NaN
3    PASSENGER VEHICLE   PASSENGER VEHICLE   PASSENGER VEHICLE
4    PASSENGER VEHICLE   PASSENGER VEHICLE                 NaN

  VEHICLE TYPE CODE 4 VEHICLE TYPE CODE 5
0                 NaN                 NaN
1                 NaN                 NaN
2                 NaN                 NaN
3                 NaN                 NaN
4                 NaN                 NaN

[5 rows x 26 columns]
```

[41]:
```
collisions.dtypes
```

[41]:
```
DATE                              object
TIME                              object
BOROUGH                           object
```

(continues on next page)

(continued from previous page)

```
ZIP CODE                         float64
LATITUDE                         float64
LONGITUDE                        float64
LOCATION                          object
ON STREET NAME                    object
CROSS STREET NAME                 object
OFF STREET NAME                   object
NUMBER OF PERSONS INJURED          int64
NUMBER OF PERSONS KILLED           int64
NUMBER OF PEDESTRIANS INJURED      int64
NUMBER OF PEDESTRIANS KILLED       int64
NUMBER OF CYCLISTS INJURED       float64
NUMBER OF CYCLISTS KILLED        float64
CONTRIBUTING FACTOR VEHICLE 1     object
CONTRIBUTING FACTOR VEHICLE 2     object
CONTRIBUTING FACTOR VEHICLE 3     object
CONTRIBUTING FACTOR VEHICLE 4     object
CONTRIBUTING FACTOR VEHICLE 5     object
VEHICLE TYPE CODE 1               object
VEHICLE TYPE CODE 2               object
VEHICLE TYPE CODE 3               object
VEHICLE TYPE CODE 4               object
VEHICLE TYPE CODE 5               object
dtype: object
```

We will then inspect the basic statistics.

[42]:
```
collisions.describe()
```

[42]:
```
           ZIP CODE     LATITUDE    LONGITUDE  NUMBER OF PERSONS INJURED  \
count   6919.000000  7303.000000  7303.000000                7303.000000
mean   10900.746640    40.717653   -73.921406                   0.350678
std      551.568724     0.069437     0.083317                   0.707873
min    10001.000000    40.502341   -74.248277                   0.000000
25%    10310.000000    40.670865   -73.980744                   0.000000
50%    11211.000000    40.723260   -73.933888                   0.000000
75%    11355.000000    40.759527   -73.864463                   1.000000
max    11694.000000    40.909628   -73.702590                  16.000000

       NUMBER OF PERSONS KILLED  NUMBER OF PEDESTRIANS INJURED  \
count               7303.000000                    7303.000000
mean                   0.000959                       0.133644
std                    0.030947                       0.362129
min                    0.000000                       0.000000
25%                    0.000000                       0.000000
50%                    0.000000                       0.000000
75%                    0.000000                       0.000000
max                    1.000000                       3.000000

       NUMBER OF PEDESTRIANS KILLED  NUMBER OF CYCLISTS INJURED  \
count                   7303.000000                         0.0
mean                       0.000822                         NaN
std                        0.028653                         NaN
```

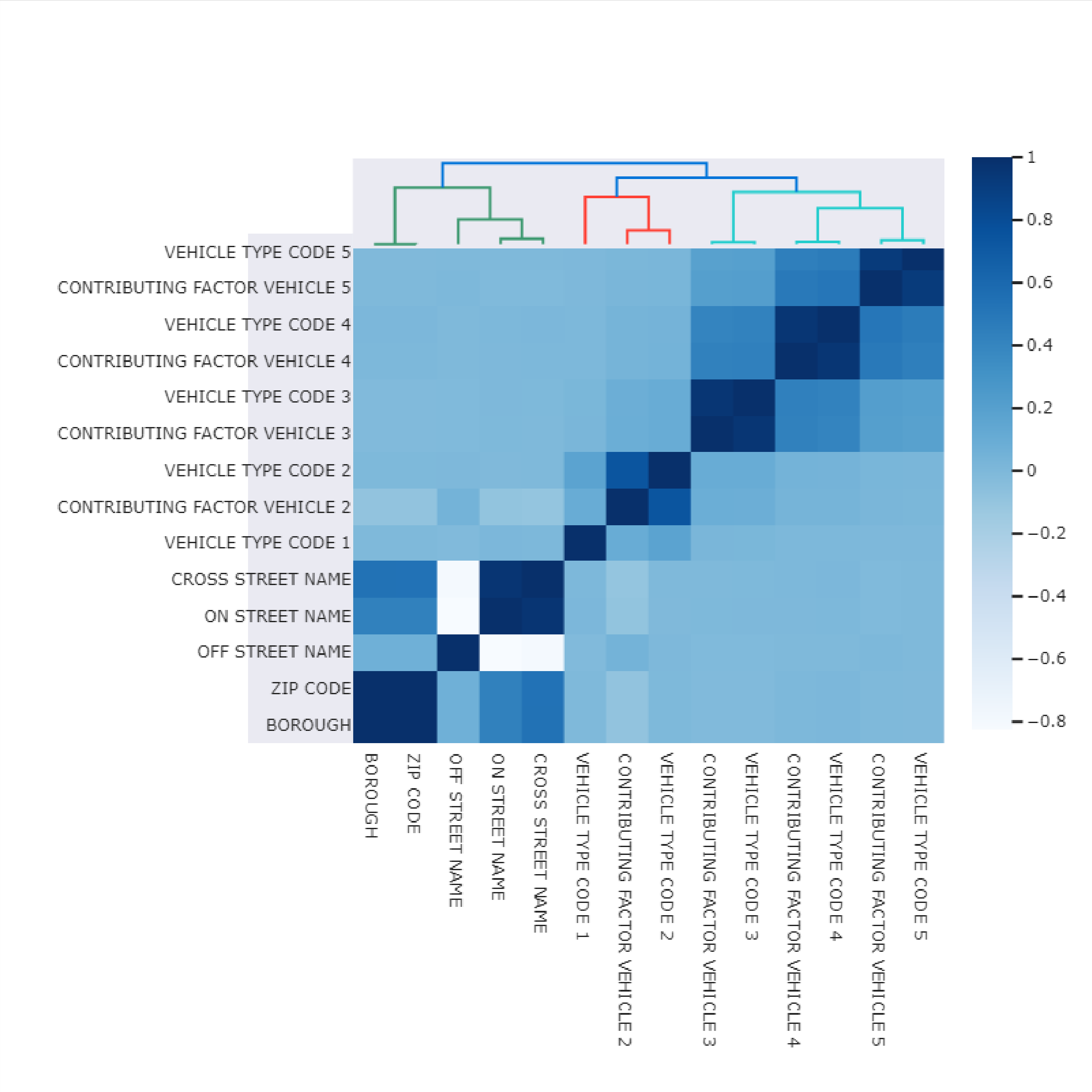
(continues on next page)

(continued from previous page)

```
min                        0.000000                         NaN
25%                        0.000000                         NaN
50%                        0.000000                         NaN
75%                        0.000000                         NaN
max                        1.000000                         NaN

       NUMBER OF CYCLISTS KILLED
count                        0.0
mean                         NaN
std                          NaN
min                          NaN
25%                          NaN
50%                          NaN
75%                          NaN
max                          NaN
```

Finally, we will visualize the nullity (how strongly the presence or absence of one variable affects the presence of another) correlations by a heatmap and a dendrogram.

[43]:
```
fig = missingness.heatmap(collisions)
fig.show()
```



The nullity heatmap and dendrogram reveals a data correlation structure, e.g., *vehicle type codes* and *contributing factor vehicle* are highly correlated. Features having complete data are not shown on the figure.

CHAPTER ELEVEN

DEMO NOTEBOOK FOR ANALYSING LOCATION DATA

11.1 Introduction

GPS location data contain rich information about people's behavioral and mobility patterns. However, working with such data is a challenging task since there exists a lot of noise and missingness. Also, designing relevant features to gain knowledge about the mobility pattern of subjects is a crucial task. To address these problems, `niimpy` provides these main functions to clean, downsample, and extract features from GPS location data:

- `niimpy.preprocessing.location.filter_location`: removes low-quality location data points
- `niimpy.util.aggregate`: downsamples data points to reduce noise
- `niimpy.preprocessing.location.extract_features_location`: feature extraction from location data

In the following, we go through analysing a subset of location data provided in StudentLife dataset.

11.2 Read data

[1]:
```
import niimpy
from niimpy import config
import niimpy.preprocessing.location as nilo
import warnings
warnings.filterwarnings("ignore")
```

[2]:
```
data = niimpy.read_csv(config.GPS_PATH, tz='Europe/Helsinki')
data.shape
```

[2]:
```
(9857, 6)
```

There are 9857 location datapoints with 6 columns in the dataset. Let us have a quick look at the data:

[3]:
```
data.head()
```

[3]:
```
                                 time  double_latitude  double_longitude
2013-03-27 06:03:29+02:00  1364357009        43.706667        -72.289097  \
2013-03-27 06:23:29+02:00  1364358209        43.706637        -72.289066
2013-03-27 06:43:25+02:00  1364359405        43.706678        -72.289018
2013-03-27 07:03:29+02:00  1364360609        43.706665        -72.289087
2013-03-27 07:23:25+02:00  1364361805        43.706808        -72.289370

                           double_speed     user                  datetime
```

(continues on next page)

(continued from previous page)

```
2013-03-27 06:03:29+02:00          0.00  gps_u01 2013-03-27 06:03:29+02:00
2013-03-27 06:23:29+02:00          0.00  gps_u01 2013-03-27 06:23:29+02:00
2013-03-27 06:43:25+02:00          0.25  gps_u01 2013-03-27 06:43:25+02:00
2013-03-27 07:03:29+02:00          0.00  gps_u01 2013-03-27 07:03:29+02:00
2013-03-27 07:23:25+02:00          0.00  gps_u01 2013-03-27 07:23:25+02:00
```

The necessary columns for further analysis are `double_latitude`, `double_longitude`, `double_speed`, and `user`. `user` refers to a unique identifier for a subject.

11.3 Filter data

Three different methods for filtering low-quality data points are implemented in `niimpy`:

- `remove_disabled`: removes data points whose `disabled` column is `True`.
- `remove_network`: removes data points whose `provider` column is `network`. This method keeps only `gps`-derived data points.
- `remove_zeros`: removes data points close to the point <lat=0, lon=0>.

[4]:
```
data = nilo.filter_location(data, remove_disabled=False, remove_network=False, remove_
↪zeros=True)
data.shape
```

[4]:
```
(9857, 6)
```

There is no such data points in this dataset; therefore the dataset does not change after this step and the number of datapoints remains the same.

11.4 Downsample

Because GPS records are not always very accurate and they have random errors, it is a good practice to downsample or aggregate data points which are recorded in close time windows. In other words, all the records in the same time window are aggregated to form one GPS record associated to that time window. There are a few parameters to adjust the aggregation setting:

- `freq`: represents the length of time window. This parameter follows the formatting of the pandas time offset aliases function. For example '5T' means 5 minute intervals.
- `method_numerical`: specifies how numerical columns should be aggregated. Options are 'mean', 'median', 'sum'.
- `method_categorical`: specifies how categorical columns should be aggregated. Options are 'first', 'mode' (most frequent), 'last'.

The aggregation is performed for each `user` (subject) separately.

[5]:
```
binned_data = niimpy.util.aggregate(data, freq='5T', method_numerical='median')
binned_data = binned_data.reset_index(0).dropna()
binned_data.shape
```

[5]:
```
(9755, 6)
```

After binning, the number of datapoints (bins) reduces to 9755.

11.5 Feature extraction

Here is the list of features `niimpy` extracts from location data:

1. Distance based features (`niimpy.preprocessing.location.location_distance_features`):

Feature	Description
`dist_total`	Total distance a person traveled in meters
`variance`, `log_variance`	Variance is defined as sum of variance in latitudes and longitudes
`speed_average`, `speed_variance`, and `speed_max`	Statistics of speed (m/s). Speed, if not given, can be calculated by dividing the distance between two consequitive bins by their time difference
`n_bins`	Number of location bins that a user recorded in dataset

2. Significant place related features (`niimpy.preprocessing.location.location_significant_place_features`):

Feature	Description
`n_static`	Number of static points. Static points are defined as bins whose speed is lower than a threshold
`n_moving`	Number of moving points. Equivalent to `n_bins - n_static`
`n_home`	Number of static bins which are close to the person's home. Home is defined the place most visited during nights. More formally, all the locations recorded during 12 Am and 6 AM are clusterd and the center of largest cluster is assumed to be home
`max_dist_home`	Maximum distance from home
`n_sps`	Bumber of significant places. All of the static bins are clusterd using DBSCAN algorithm. Each cluster represents a Signicant Place (SP) for a user
`n_rare`	Number of rarely visited (referred as outliers in DBSCAN)
`n_transitions`	Number of transitions between significant places
`n_top1`, `n_top2`, `n_top3`, `n_top4`, `n_top5`	: Number of bins in the top `N` cluster. In other words, `n_top1` shows the number of times the person has visited the most freqently visited place
`entropy`, `normalized_entropy`	: Entropy of time spent in clusters. Normalized entropy is the entropy divided by the number of clusters

[6]:
```
import warnings
warnings.filterwarnings('ignore', category=RuntimeWarning)

# extract all the available features
all_features = nilo.extract_features_location(binned_data)
all_features
```

[6]:
```
                                    n_significant_places  n_sps  n_static
user
gps_u00 2013-03-31 00:00:00+02:00                      6    5.0     280.0  \
        2013-04-30 00:00:00+03:00                     10   10.0    1966.0
        2013-05-31 00:00:00+03:00                     15   12.0    1827.0
        2013-06-30 00:00:00+03:00                      1    1.0      22.0
gps_u01 2013-03-31 00:00:00+02:00                      4    2.0     307.0
        2013-04-30 00:00:00+03:00                      4    1.0    1999.0
        2013-05-31 00:00:00+03:00                      2    1.0    3079.0
```

(continues on next page)

(continued from previous page)

```
                                   n_moving  n_rare  n_home  max_dist_home  
user                                                                        
gps_u00 2013-03-31 00:00:00+02:00       8.0     3.0   106.0   2.074186e+04  \
        2013-04-30 00:00:00+03:00      66.0    45.0  1010.0   2.914790e+05   
        2013-05-31 00:00:00+03:00      76.0    86.0  1028.0   1.041741e+06   
        2013-06-30 00:00:00+03:00       2.0    15.0     0.0   2.035837e+04   
gps_u01 2013-03-31 00:00:00+02:00      18.0     0.0   260.0   6.975303e+02   
        2013-04-30 00:00:00+03:00      71.0     1.0  1500.0   1.156568e+04   
        2013-05-31 00:00:00+03:00      34.0     1.0    45.0   3.957650e+03   

                                   n_transitions  n_top1  n_top2  ...  n_top5  
user                                                              ...          
gps_u00 2013-03-31 00:00:00+02:00           48.0   106.0    99.0  ...    18.0  \
        2013-04-30 00:00:00+03:00          194.0  1016.0   668.0  ...    38.0   
        2013-05-31 00:00:00+03:00          107.0  1030.0   501.0  ...    46.0   
        2013-06-30 00:00:00+03:00           10.0    15.0     7.0  ...     0.0   
gps_u01 2013-03-31 00:00:00+02:00            8.0   286.0    21.0  ...     0.0   
        2013-04-30 00:00:00+03:00            2.0  1998.0     1.0  ...     0.0   
        2013-05-31 00:00:00+03:00            2.0  3078.0     1.0  ...     0.0   

                                    entropy  normalized_entropy    dist_total  
user                                                                           
gps_u00 2013-03-31 00:00:00+02:00  5.091668            3.163631  4.132581e+05  \
        2013-04-30 00:00:00+03:00  7.284903            3.163793  2.179693e+06   
        2013-05-31 00:00:00+03:00  6.701177            2.696752  6.986551e+06   
        2013-06-30 00:00:00+03:00  0.000000            0.000000  2.252893e+05   
gps_u01 2013-03-31 00:00:00+02:00  3.044522            4.392317  1.328713e+04   
        2013-04-30 00:00:00+03:00  0.000000            0.000000  1.238429e+05   
        2013-05-31 00:00:00+03:00  0.000000            0.000000  1.228235e+05   

                                   n_bins  speed_average  speed_variance  
user                                                                      
gps_u00 2013-03-31 00:00:00+02:00   288.0       0.033496        0.044885  \
        2013-04-30 00:00:00+03:00  2032.0       0.269932        6.129277   
        2013-05-31 00:00:00+03:00  1903.0       0.351280        7.590639   
        2013-06-30 00:00:00+03:00    24.0       0.044126        0.021490   
gps_u01 2013-03-31 00:00:00+02:00   325.0       0.056290        0.073370   
        2013-04-30 00:00:00+03:00  2070.0       0.066961        0.629393   
        2013-05-31 00:00:00+03:00  3113.0       0.026392        0.261978   

                                   speed_max  variance  log_variance  
user                                                                  
gps_u00 2013-03-31 00:00:00+02:00   1.750000  0.003146     -5.761688  
        2013-04-30 00:00:00+03:00  33.250000  0.237133     -1.439133  
        2013-05-31 00:00:00+03:00  34.000000  8.288687      2.114892  
        2013-06-30 00:00:00+03:00   0.559017  0.014991     -4.200287  
gps_u01 2013-03-31 00:00:00+02:00   2.692582  0.000004    -12.520989  
        2013-04-30 00:00:00+03:00  32.750000  0.000027    -10.510017  
        2013-05-31 00:00:00+03:00  20.250000  0.000012    -11.364454  

[7 rows x 22 columns]
```

[7]:
```
# extract only distance related features
feature_functions = {
    nilo.location_distance_features: {} # arguments
}
distance_features = nilo.extract_features_location(
    binned_data,
    feature_functions=feature_functions)
distance_features
```

[7]:
```
                                     dist_total  n_bins  speed_average  \
user
gps_u00 2013-03-31 00:00:00+02:00  4.132581e+05   288.0       0.033496
        2013-04-30 00:00:00+03:00  2.179693e+06  2032.0       0.269932
        2013-05-31 00:00:00+03:00  6.986551e+06  1903.0       0.351280
        2013-06-30 00:00:00+03:00  2.252893e+05    24.0       0.044126
gps_u01 2013-03-31 00:00:00+02:00  1.328713e+04   325.0       0.056290
        2013-04-30 00:00:00+03:00  1.238429e+05  2070.0       0.066961
        2013-05-31 00:00:00+03:00  1.228235e+05  3113.0       0.026392

                                   speed_variance  speed_max  variance  \
user
gps_u00 2013-03-31 00:00:00+02:00        0.044885   1.750000  0.003146
        2013-04-30 00:00:00+03:00        6.129277  33.250000  0.237133
        2013-05-31 00:00:00+03:00        7.590639  34.000000  8.288687
        2013-06-30 00:00:00+03:00        0.021490   0.559017  0.014991
gps_u01 2013-03-31 00:00:00+02:00        0.073370   2.692582  0.000004
        2013-04-30 00:00:00+03:00        0.629393  32.750000  0.000027
        2013-05-31 00:00:00+03:00        0.261978  20.250000  0.000012

                                   log_variance
user
gps_u00 2013-03-31 00:00:00+02:00     -5.761688
        2013-04-30 00:00:00+03:00     -1.439133
        2013-05-31 00:00:00+03:00      2.114892
        2013-06-30 00:00:00+03:00     -4.200287
gps_u01 2013-03-31 00:00:00+02:00    -12.520989
        2013-04-30 00:00:00+03:00    -10.510017
        2013-05-31 00:00:00+03:00    -11.364454
```

The 2 rows correspond to the 2 users present in the dataset. Each column represents a feature. For example user `gps_u00` has higher variance in speeds (`speed_variance`) and location variance (`variance`) compared to the user `gps_u01`.

11.6 Implementing your own features

If you want to implement a customized feature you can do so with defining a function that accepts a dataframe and returns a dataframe or a series. The returned object should be indexed by `user`. Then, when calling `extract_features_location` function, you add the newly implemented function to the `feature_functions` argument. The default feature functions implemented in `niimpy` are in this variable:

[8]:
```
nilo.ALL_FEATURE_FUNCTIONS
```

```
[8]: {<function niimpy.preprocessing.location.location_number_of_significant_places(df,␣
↪feature_functions={})>: {'resample_args': {'rule': '1M'}},
 <function niimpy.preprocessing.location.location_significant_place_features(df, feature_
↪functions={})>: {'resample_args': {'rule': '1M'}},
 <function niimpy.preprocessing.location.location_distance_features(df, feature_
↪functions={})>: {'resample_args': {'rule': '1M'}}}
```

You can add your new function to the `nilo.ALL_FEATURE_FUNCTIONS` dictionary and call `extract_features_location` function. Or if you are interested in only extracting your desired feature you can pass a dictionary containing just that function, like here:

```
[9]: # customized function
def max_speed(df, feature_arg):
    grouped = df.groupby('user')
    return grouped['double_speed'].max()

customized_features = nilo.extract_features_location(
    binned_data,
    feature_functions={max_speed: {}}
)
customized_features
```

```
[9]:         double_speed
user
gps_u00         34.00
gps_u01         32.75
```

CHAPTER

TWELVE

DEMO NOTEBOOK FOR ANALYZING APPLICATION DATA

12.1 Introduction

Application data refers to the information about which apps are open at a certain time. These data can reveal important information about people's circadian rhythm, social patterns, and activity. Application data is an event data; this means it cannot be sampled at a regular frequency. Instead, we just have information about the events that occured. There are two main issues with application data (1) missing data detection, and (2) privacy concerns.

Regarding missing data detection, we may never know if all events were detected and reported. Unfortunately there is little we can do. Nevertheless, we can take into account some factors that may interfere with the correct detection of all events (e.g. when the phone's battery is depleated). Therefore, to correctly process application data, we need to consider other information like the battery status of the phone. Regarding the privacy concerns, application names can reveal too much about a subject, for example, an uncommon app use may help identify a subject. Consequently, we try anonimizing the data by grouping the apps.

To address both of these issues, `niimpy` includes the function `extract_features_app` to clean, downsample, and extract features from application data while taking into account factors like the battery level and naming groups. In addition, `niimpy` provides a map with some of the common apps for pseudo-anonymization. This function employs other functions to extract the following features:

- `app_count`: number of times an app group has been used
- `app_duration`: how long an app group has been used

The app module has one internal function that help classify the apps into groups.

In the following, we will analyze screen data provided by `niimpy` as an example to illustrate the use of application data.

12.2 2. Read data

Let's start by reading the example data provided in `niimpy`. These data have already been shaped in a format that meets the requirements of the data schema. Let's start by importing the needed modules. Firstly we will import the `niimpy` package and then we will import the module we will use (application) and give it a short name for use convenience.

[1]:
```
import niimpy
from niimpy import config
import niimpy.preprocessing.application as app
import pandas as pd
import warnings
warnings.filterwarnings("ignore")
```

Now let's read the example data provided in `niimpy`. The example data is in `csv` format, so we need to use the `read_csv` function. When reading the data, we can specify the timezone where the data was collected. This will help us handle daylight saving times easier. We can specify the timezone with the argument **tz**. The output is a dataframe. We can also check the number of rows and columns in the dataframe.

[2]:
```
data = niimpy.read_csv(config.SINGLEUSER_AWARE_APP_PATH, tz='Europe/Helsinki')
data.shape
```

[2]:
```
(132, 6)
```

The data was succesfully read. We can see that there are 132 datapoints with 6 columns in the dataset. However, we do not know yet what the data really looks like, so let's have a quick look:

[3]:
```
data.head()
```

[3]:
```
                                           user         device          time  \
2019-08-05 14:02:51.009999872+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565003e+09
2019-08-05 14:02:58.009999872+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565003e+09
2019-08-05 14:03:17.009999872+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565003e+09
2019-08-05 14:02:55.009999872+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565003e+09
2019-08-05 14:03:31.009999872+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565003e+09

                                     application_name  \
2019-08-05 14:02:51.009999872+03:00    Android System
2019-08-05 14:02:58.009999872+03:00    Android System
2019-08-05 14:03:17.009999872+03:00  Google Play Music
2019-08-05 14:02:55.009999872+03:00  Google Play Music
2019-08-05 14:03:31.009999872+03:00              Gmail

                                                 package_name  \
2019-08-05 14:02:51.009999872+03:00                   android
2019-08-05 14:02:58.009999872+03:00                   android
2019-08-05 14:03:17.009999872+03:00  com.google.android.music
2019-08-05 14:02:55.009999872+03:00  com.google.android.music
2019-08-05 14:03:31.009999872+03:00     com.google.android.gm

                                                               datetime
2019-08-05 14:02:51.009999872+03:00 2019-08-05 14:02:51.009999872+03:00
2019-08-05 14:02:58.009999872+03:00 2019-08-05 14:02:58.009999872+03:00
2019-08-05 14:03:17.009999872+03:00 2019-08-05 14:03:17.009999872+03:00
2019-08-05 14:02:55.009999872+03:00 2019-08-05 14:02:55.009999872+03:00
2019-08-05 14:03:31.009999872+03:00 2019-08-05 14:03:31.009999872+03:00
```

By exploring the head of the dataframe we can form an idea of its entirety. From the data, we can see that:

- rows are observations, indexed by timestamps, i.e. each row represents that an app has been prompted to the smartphone screen
- columns are characteristics for each observation, for example, the user whose data we are analyzing
- there is one main column: `application_name`, which stores the Android name for the application.

12.2.1 A few words on missing data

Missing data for application is difficult to detect. Firstly, this sensor is triggered by events (i.e. not sampled at a fixed frequency). Secondly, different phones, OS, and settings change how easy it is to detect apps. Thirdly, events not related to the application sensor may affect its behavior, e.g. battery running out. Unfortunately, we can only correct missing data for events such as the screen turning off by using data from the screen sensor and the battery level. These can be taken into account in `niimpy` if we provide the screen and battery data. We will see some examples below.

12.2.2 A few words on grouping the apps

As previously mentioned, the application name may reveal too much about a subject and privacy problems may arise. A possible solution to this problem is to classify the apps into more generic groups. For example, apps like WhatsApp, Signal, Telegram, etc. are commonly used for texting, so we can group them under the label *texting*. `niimpy` provides a default map, but this should be adapted to the characteristics of the sample, since apps are available depending on countries and populations.

12.2.3 A few words on the role of the battery and screen

As mentioned before, sometimes the screen may be OFF and these events will not be caught by the application data sensor. For example, we can open an app and let it remain open until the phone screen turns off automatically. Another example is when the battery is depleated and the phone is shut down automatically. Having this information is crucial for correctly computing how long a subject used each app group. `niimpy`'s screen module is adapted to take into account both, the screen and battery data. For this example, we have both, so let's load the screen and battery data.

[4]:
```
bat_data = niimpy.read_csv(config.MULTIUSER_AWARE_BATTERY_PATH, tz='Europe/Helsinki')
screen_data = niimpy.read_csv(config.MULTIUSER_AWARE_SCREEN_PATH, tz='Europe/Helsinki')
```

[5]:
```
bat_data.head()
```

[5]:
```
                                             user         device          time  \
2020-01-09 02:20:02.924999936+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:21:30.405999872+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:24:12.805999872+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:35:38.561000192+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09
2020-01-09 02:35:38.953000192+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09

                                     battery_level  battery_status  \
2020-01-09 02:20:02.924999936+02:00             74               3
2020-01-09 02:21:30.405999872+02:00             73               3
2020-01-09 02:24:12.805999872+02:00             72               3
2020-01-09 02:35:38.561000192+02:00             72               2
2020-01-09 02:35:38.953000192+02:00             72               2

                                     battery_health  battery_adaptor  \
2020-01-09 02:20:02.924999936+02:00               2                0
2020-01-09 02:21:30.405999872+02:00               2                0
2020-01-09 02:24:12.805999872+02:00               2                0
2020-01-09 02:35:38.561000192+02:00               2                0
2020-01-09 02:35:38.953000192+02:00               2                2

                                                               datetime
2020-01-09 02:20:02.924999936+02:00 2020-01-09 02:20:02.924999936+02:00
```

(continues on next page)

(continued from previous page)

```
2020-01-09 02:21:30.405999872+02:00 2020-01-09 02:21:30.405999872+02:00
2020-01-09 02:24:12.805999872+02:00 2020-01-09 02:24:12.805999872+02:00
2020-01-09 02:35:38.561000192+02:00 2020-01-09 02:35:38.561000192+02:00
2020-01-09 02:35:38.953000192+02:00 2020-01-09 02:35:38.953000192+02:00
```

The dataframe looks fine. In this case, we are interested in the battery_status information. This is standard information provided by Android. However, if the dataframe stores this information in a column with a different name, we can use the argument `battery_column_name` and input our custom battery column name (again, we will have an example below).

[6]:
```
screen_data.head()
```

[6]:
```
                                             user         device          time  \
2020-01-09 02:06:41.573999872+02:00  jd9INuQ5BBlW  OWd1Uau8POix  1.578528e+09
2020-01-09 02:09:29.152000+02:00     jd9INuQ5BBlW  OWd1Uau8POix  1.578529e+09
2020-01-09 02:09:32.790999808+02:00  jd9INuQ5BBlW  OWd1Uau8POix  1.578529e+09
2020-01-09 02:11:41.996000+02:00     jd9INuQ5BBlW  OWd1Uau8POix  1.578529e+09
2020-01-09 02:16:19.010999808+02:00  jd9INuQ5BBlW  OWd1Uau8POix  1.578529e+09

                                     screen_status  \
2020-01-09 02:06:41.573999872+02:00              0
2020-01-09 02:09:29.152000+02:00                 1
2020-01-09 02:09:32.790999808+02:00              3
2020-01-09 02:11:41.996000+02:00                 0
2020-01-09 02:16:19.010999808+02:00              1

                                                                datetime
2020-01-09 02:06:41.573999872+02:00 2020-01-09 02:06:41.573999872+02:00
2020-01-09 02:09:29.152000+02:00       2020-01-09 02:09:29.152000+02:00
2020-01-09 02:09:32.790999808+02:00 2020-01-09 02:09:32.790999808+02:00
2020-01-09 02:11:41.996000+02:00       2020-01-09 02:11:41.996000+02:00
2020-01-09 02:16:19.010999808+02:00 2020-01-09 02:16:19.010999808+02:00
```

This dataframe looks fine too. In this case, we are interested in the screen_status information, which is also standardized values provided by Android. The column does not need to be name "screen_status" as we can pass the name later on. We will see an example later.

12.3 * TIP! Data format requirements (or what should our data look like)

Data can take other shapes and formats. However, the `niimpy` data scheme requires it to be in a certain shape. This means the application dataframe needs to have at least the following characteristics: 1. One row per app prompt. Each row should store information about one app prompt only 2. Each row's index should be a timestamp 3. There should be at least three columns: - index: date and time when the event happened (timestamp) - user: stores the user name whose data is analyzed. Each user should have a unique name or hash (i.e. one hash for each unique user) - application_name: stores the Android application name 4. Columns additional to those listed in item 3 are allowed 5. The names of the columns do not need to be exactly "user", and "application_name" as we can pass our own names in an argument (to be explained later).

Below is an example of a dataframe that complies with these minimum requirements

[7]:
```
example_dataschema = data[['user','application_name']]
example_dataschema.head(3)
```

[7]:
```
                                          user   application_name
2019-08-05 14:02:51.009999872+03:00  iGyXetHE3S8u     Android System
2019-08-05 14:02:58.009999872+03:00  iGyXetHE3S8u     Android System
2019-08-05 14:03:17.009999872+03:00  iGyXetHE3S8u  Google Play Music
```

Similarly, if we employ screen and battery data, we need to fulfill minimum data scheme requirements. We will briefly show examples of these dataframes that comply with the minimum requirements.

[8]:
```
example_screen_dataschema = screen_data[['user','screen_status']]
example_screen_dataschema.head(3)
```

[8]:
```
                                          user  screen_status
2020-01-09 02:06:41.573999872+02:00  jd9INuQ5BBlW              0
2020-01-09 02:09:29.152000+02:00     jd9INuQ5BBlW              1
2020-01-09 02:09:32.790999808+02:00  jd9INuQ5BBlW              3
```

[9]:
```
example_battery_dataschema = bat_data[['user','battery_status']]
example_battery_dataschema.head(3)
```

[9]:
```
                                          user  battery_status
2020-01-09 02:20:02.924999936+02:00  jd9INuQ5BBlW               3
2020-01-09 02:21:30.405999872+02:00  jd9INuQ5BBlW               3
2020-01-09 02:24:12.805999872+02:00  jd9INuQ5BBlW               3
```

12.4 4. Extracting features

There are two ways to extract features. We could use each function separately or we could use `niimpy`'s ready-made wrapper. Both ways will require us to specify arguments to pass to the functions/wrapper in order to customize the way the functions work. These arguments are specified in dictionaries. Let's first understand how to extract features using stand-alone functions.

We can use `niimpy`'s functions to compute communication features. Each function will require two inputs: - (mandatory) dataframe that must comply with the minimum requirements (see '* TIP! Data requirements above) - (optional) an argument dictionary for stand-alone functions

12.4.1 4.1.1 The argument dictionary for stand-alone functions (or how we specify the way a function works)

In this dictionary, we can input two main features to customize the way a stand-alone function works: - the name of the columns to be preprocessed: Since the dataframe may have different columns, we need to specify which column has the data we would like to be preprocessed. To do so, we can simply pass the name of the column to the argument `app_column_name`.

- the way we resample: resampling options are specified in `niimpy` as a dictionary. `niimpy`'s resampling and aggregating relies on `pandas.DataFrame.resample`, so mastering the use of this pandas function will help us greatly in `niimpy`'s preprocessing. Please familiarize yourself with the pandas resample function before continuing. Briefly, to use the `pandas.DataFrame.resample` function, we need a rule. This rule states the intervals we would like to use to resample our data (e.g., 15-seconds, 30-minutes, 1-hour). Neverthless, we can input more details into the function to specify the exact sampling we would like. For example, we could use the *close* argument if we would like to specify which side of the interval is closed, or we could use the *offset* argument if we would like to start our binning with an offset, etc. There are plenty of options to use this command, so we strongly recommend having `pandas.DataFrame.resample` documentation at hand. All features for the `pandas.DataFrame.resample` will be specified in a dictionary where keys are the arguments' names for the `pandas.DataFrame.resample` function, and the dictionary's values are the values for each of these selected arguments. This dictionary will be passed as a value to the key `resample_args` in `niimpy`.

Let's see some basic examples of these dictionaries:

[10]:
```
feature_dict1:{"app_column_name":"application_name","resample_args":{"rule":"1D"}}
feature_dict2:{"app_column_name":"other_name", "screen_column_name":"screen_name",
↪"resample_args":{"rule":"45T","origin":"end"}}
```

Here, we have two basic feature dictionaries.

- `feature_dict1` will be used to analyze the data stored in the column `application_name` in our dataframe. The data will be binned in one day periods
- `feature_dict2` will be used to analyze the data stored in the column `other_name` in our dataframe. In addition, we will provide some screen data in the column "screen_name". The data will be binned in 45-minutes bins, but the binning will start from the last timestamp in the dataframe.

Default values: if no arguments are passed, `niimpy`'s default values are "application_name" for the app_column_name, "screen_status" for the screen_column_name, and "battery_status" for the battery_column_name. We will also use the default 30-min aggregation bins.

12.4.2 4.1.2 Using the functions

Now that we understand how the functions are customized, it is time we compute our first application feature. Suppose that we are interested in extracting the number of times each app group has been used within 1-minutes bins. We will need `niimpy`'s `app_count` function, the data, and we will also need to create a dictionary to customize our function. Let's create the dictionary first

[11]:
```
function_features={"app_column_name":"application_name","resample_args":{"rule":"1T"}}
```

Now let's use the function to preprocess the data.

[12]:
```
my_app_count = app.app_count(data, bat_data, screen_data, function_features)
```

`my_app_count` is a multiindex dataframe, where the first level is the user, and the second level is the app group. Let's look at some values.

[13]:
```
my_app_count.head()
```

[13]:
```
                                                  count
user         app_group datetime
iGyXetHE3S8u comm      2019-08-05 14:02:00+03:00     28
                       2019-08-05 14:03:00+03:00     58
             leisure   2019-08-05 14:02:00+03:00      3
                       2019-08-05 14:03:00+03:00     17
             na        2019-08-05 14:02:00+03:00     10
```

We see that the bins are indeed 1-minutes bins, however, they are adjusted to fixed, predetermined intervals, i.e. the bin does not start on the time of the first datapoint. Instead, `pandas` starts the binning at 00:00:00 of everyday and counts 1-minutes intervals from there.

If we want the binning to start from the first datapoint in our dataset, we need the origin parameter and a for loop.

[14]:
```
users = list(data['user'].unique())
results = []
for user in users:
    start_time = data[data["user"]==user].index.min()
    function_features={"app_column_name":"application_name","resample_args":{"rule":"1T",
↪"origin":start_time}}
    results.append(app.app_count(data[data["user"]==user],bat_data[bat_data["user
↪"]==user], screen_data[screen_data["user"]==user], function_features))
my_app_count = pd.concat(results)
```

[15]:
```
my_app_count
```

[15]:
```
                                                       count
user         app_group datetime
iGyXetHE3S8u comm      2019-08-05 14:02:42.009999872+03:00     86
             leisure   2019-08-05 14:02:42.009999872+03:00     20
             na        2019-08-05 14:02:42.009999872+03:00     19
             work      2019-08-05 14:02:42.009999872+03:00      7
```

Compare the timestamps and notice the small difference in this example. In the cell 21, the first timestamp is at 14:02:00, whereas in the new app_count, the first timestamp is at 14:02:42

The functions can also be called in absence of battery or screen data. In this case, simply input an empty dataframe in the second or third position of the function. For example,

[16]:
```
empty_bat = pd.DataFrame()
empty_screen = pd.DataFrame()
no_bat = app.app_count(data, empty_bat, screen_data, function_features) #no battery␣
↪information
no_screen = app.app_count(data, bat_data, empty_screen, function_features) #no screen␣
↪information
no_bat_no_screen = app.app_count(data, empty_bat, empty_screen, function_features) #no␣
↪battery and no screen information
```

[17]:
```
no_bat.head()
```

[17]:
```
                                                       count
user         app_group datetime
iGyXetHE3S8u comm      2019-08-05 14:02:42.009999872+03:00     86
             leisure   2019-08-05 14:02:42.009999872+03:00     20
             na        2019-08-05 14:02:42.009999872+03:00     19
             work      2019-08-05 14:02:42.009999872+03:00      7
```

[18]:
```
no_screen.head()
```

[18]:
```
                                                       count
user         app_group datetime
iGyXetHE3S8u comm      2019-08-05 14:02:42.009999872+03:00     86
             leisure   2019-08-05 14:02:42.009999872+03:00     20
             na        2019-08-05 14:02:42.009999872+03:00     19
             off       2019-08-07 10:36:42.009999872+03:00      2
             work      2019-08-05 14:02:42.009999872+03:00      7
```

We see some small differences between these two dataframes. For example, the no_screen dataframe includes the app_group "off", as it has taken into account the battery data and knows when the phone has been shut down.

4.2 Extract features using the wrapper

We can use `niimpy`'s ready-made wrapper to extract one or several features at the same time. The wrapper will require two inputs: - (mandatory) dataframe that must comply with the minimum requirements (see '* TIP! Data requirements above) - (optional) an argument dictionary for wrapper

12.4.3 4.2.1 The argument dictionary for wrapper (or how we specify the way the wrapper works)

This argument dictionary will use dictionaries created for stand-alone functions. If you do not know how to create those argument dictionaries, please read the section **4.1.1 The argument dictionary for stand-alone functions (or how we specify the way a function works)** first.

The wrapper dictionary is simple. Its keys are the names of the features we want to compute. Its values are argument dictionaries created for each stand-alone function we will employ. Let's see some examples of wrapper dictionaries:

[19]:
```
wrapper_features1 = {app.app_count:{"app_column_name":"application_name", "resample_args
↪":{"rule":"1T"}},
            app.app_duration:{"app_column_name":"some_name", "screen_column_name":
↪"screen_name", "battery_column_name":"battery_name", "resample_args":{"rule":"1T"}}}
```

- `wrapper_features1` will be used to analyze two features, `app_count` and `app_duration`. For the feature app_count, we will use the data stored in the column `application_name` in our dataframe and the data will be binned in one-minute periods. For the feature app_duration, we will use the data stored in the column `some_name` in our dataframe and the data will be binned in one day periods. In addition, we will also employ screen and battery data which are stored in the columns `screen_name` and `battery_name`.

[20]:
```
wrapper_features2 = {app.app_count:{"app_column_name":"application_name", "resample_args
↪":{"rule":"1T", "offset":"15S"}},
            app.app_duration:{"app_column_name":"some_name", "screen_column_name":
↪"screen_name", "battery_column_name":"battery_name", "resample_args":{"rule":"30S"}}}
```

- `wrapper_features2` will be used to analyze two features, `app_count` and `app_duration`. For the feature app_count, we will use the data stored in the column `application_name` in our dataframe and the data will be binned in one-minute periods with a 15-seconds offset. For the feature app_duration, we will use the data stored in the column `some_name` in our dataframe and the data will be binned in 30-second periods. In addition, we will also employ screen and battery data which are stored in the columns `screen_name` and `battery_name`.

Default values: if no arguments are passed, `niimpy`'s default values are "application_name" for the app_column_name, "screen_status" for the screen_column_name, "battery_status" for the battery_column_name, and 30-min aggregation bins. Moreover, the wrapper will compute all the available functions in absence of the argument dictionary. Similarly to the use of functions, we may input empty dataframes if we do not have screen or battery data.

12.4.4 4.2.2 Using the wrapper

Now that we understand how the wrapper is customized, it is time we compute our first application feature using the wrapper. Suppose that we are interested in extracting the call total duration every 30 seconds. We will need `niimpy`'s `extract_features_apps` function, the data, and we will also need to create a dictionary to customize our function. Let's create the dictionary first

[21]:
```
wrapper_features1 = {app.app_count:{"app_column_name":"application_name", "resample_args
↪":{"rule":"30S"}}}
```

Now let's use the wrapper

[22]:
```
results_wrapper = app.extract_features_app(data, bat_data, screen_data, features=wrapper_
↪features1)
results_wrapper.head(5)
```

```
computing <function app_count at 0x7f5233babee0>...
```

[22]:
```
                                                 count
user         app_group datetime
iGyXetHE3S8u comm      2019-08-05 14:02:30+03:00     28
                       2019-08-05 14:03:00+03:00     34
                       2019-08-05 14:03:30+03:00     24
             leisure   2019-08-05 14:02:30+03:00      3
                       2019-08-05 14:03:00+03:00     15
```

Our first attempt was succesful. Now, let's try something more. Let's assume we want to compute the app_count and app_duration in 20-seconds bins. Moreover, let's assume we do not want to use the screen or battery data this time. Note that the app_duration values are in seconds.

[23]:
```
wrapper_features2 = {app.app_count:{"app_column_name":"application_name", "resample_args
↪":{"rule":"20S"}},
                     app.app_duration:{"app_column_name":"application_name", "resample_
↪args":{"rule":"20S"}}}
results_wrapper = app.extract_features_app(data, empty_bat, empty_screen,␣
↪features=wrapper_features2)
results_wrapper.head(5)
```

```
computing <function app_count at 0x000001D47C314B80>...
computing <function app_duration at 0x000001D47C314C10>...
```

[23]:
```
                                                 count  duration
user         app_group datetime
iGyXetHE3S8u comm      2019-08-05 14:02:40+03:00     28     600.0
                       2019-08-05 14:03:00+03:00     20      66.0
                       2019-08-05 14:03:20+03:00     31    -719.0
                       2019-08-05 14:03:40+03:00      7    -206.0
             leisure   2019-08-05 14:02:40+03:00      3      93.0
```

Great! Another successful attempt. We see from the results that more columns were added with the required calculations. We also see that some durations are in negative numbers, this may be due to the lack of screen and battery data. This is how the wrapper works when all features are computed with the same bins. Now, let's see how the wrapper performs when each function has different binning requirements. Let's assume we need to compute the app_count every 20 seconds, and the app_duration every 10 seconds with an offset of 5 seconds.

[24]:
```
wrapper_features3 = {app.app_count:{"app_column_name":"application_name", "resample_args
↪":{"rule":"20S"}},
                     app.app_duration:{"app_column_name":"application_name", "resample_
↪args":{"rule":"10S", "offset":"5S"}}}
results_wrapper = app.extract_features_app(data, bat_data, screen_data, features=wrapper_
↪features3)
results_wrapper.head(5)
```

```
computing <function app_count at 0x000001D47C314B80>...
computing <function app_duration at 0x000001D47C314C10>...
```

[24]:
```
                                                 count  duration
user         app_group datetime
```

(continues on next page)

(continued from previous page)

```
iGyXetHE3S8u comm      2019-08-05 14:02:40+03:00   28.0       NaN
                       2019-08-05 14:03:00+03:00   20.0       NaN
                       2019-08-05 14:03:20+03:00   31.0       NaN
                       2019-08-05 14:03:40+03:00    7.0       NaN
             leisure   2019-08-05 14:02:40+03:00    3.0       NaN
```

[25]:
```
results_wrapper.tail(5)
```

[25]:
```
                                                  count  duration
user         app_group datetime
iGyXetHE3S8u work      2019-08-05 14:02:45+03:00    NaN       1.0
                       2019-08-05 14:02:55+03:00    NaN       3.0
                       2019-08-05 14:03:05+03:00    NaN       0.0
                       2019-08-05 14:03:15+03:00    NaN       2.0
                       2019-08-05 14:03:25+03:00    NaN       0.0
```

The output is once again a dataframe. In this case, two aggregations are shown. The first one is the 20-seconds aggregation computed for the `app_count` feature (head). The second one is the 10-seconds aggregation period with 5-seconds offset for the `app_duration` (tail). Because the `app_count` feature is not required to be aggregated every 10 seconds, the aggregation timestamps have a NaN value. Similarly, because the `app_duration` is not required to be aggregated in 20-seconds windows, its values are NaN for all subjects.

12.4.5 4.2.3 Wrapper and its default option

The default option will compute all features in 30-minute aggregation windows. To use the `extract_features_apps` function with its default options, simply call the function.

[26]:
```
default = app.extract_features_app(data, bat_data, screen_data, features=None)
```

```
computing <function app_count at 0x000001D47C314B80>...
computing <function app_duration at 0x000001D47C314C10>...
```

The function prints the computed features so you can track its process. Now, let's have a look at the outputs

[27]:
```
default.head()
```

[27]:
```
                                                  count  duration
user         app_group datetime
iGyXetHE3S8u comm      2019-08-05 14:00:00+03:00     86      37.0
             leisure   2019-08-05 14:00:00+03:00     20       7.0
             na        2019-08-05 14:00:00+03:00     19       9.0
             work      2019-08-05 14:00:00+03:00      7       6.0
```

12.5 5. Implementing own features

If none of the provided functions suits well, We can implement our own customized features easily. To do so, we need to define a function that accepts a dataframe and returns a dataframe. The returned object should be indexed by user and app_groups (multiindex). To make the feature readily available in the default options, we need add the *app* prefix to the new function (e.g. `app_my-new-feature`). Let's assume we need a new function that computes the maximum duration. Let's first define the function.

```
[28]: import numpy as np
def app_max_duration(df, bat, screen, feature_functions=None):
    if not "group_map" in feature_functions.keys():
        feature_functions['group_map'] = app.MAP_APP
    if not "resample_args" in feature_functions.keys():
        feature_functions["resample_args"] = {"rule":"30T"}

    df2 = app.classify_app(df, feature_functions)
    df2['duration']=np.nan
    df2['duration']=df2['datetime'].diff()
    df2['duration'] = df2['duration'].shift(-1)
    thr = pd.Timedelta('10 hours')
    df2 = df2[~(df2.duration>thr)]
    df2 = df2[~(df2.duration>thr)]
    df2["duration"] = df2["duration"].dt.total_seconds()

    df2.dropna(inplace=True)

    if len(df2)>0:
        df2['datetime'] = pd.to_datetime(df2['datetime'])
        df2.set_index('datetime', inplace=True)
        result = df2.groupby(["user","app_group"])["duration"].resample(**feature_
↪functions["resample_args"]).max()

    return result
```

Then, we can call our new function in the stand-alone way or using the `extract_features_app` function. Because the stand-alone way is the common way to call functions in python, we will not show it. Instead, we will show how to integrate this new function to the wrapper. Let's read again the data and assume we want the default behavior of the wrapper.

```
[29]: customized_features = app.extract_features_app(data, bat_data, screen_data, features=
↪{app_max_duration: {}})
```

```
computing <function app_max_duration at 0x000001D47C44B130>...
```

```
[30]: customized_features.head()
```

```
[30]:
                                                    duration
user        app_group datetime
iGyXetHE3S8u comm      2019-08-05 14:00:00+03:00       59.0
            leisure   2019-08-05 14:00:00+03:00       36.0
            na        2019-08-05 14:00:00+03:00       53.0
            work      2019-08-05 14:00:00+03:00       19.0
```

```
[ ]:
```

CHAPTER

THIRTEEN

DEMO NOTEBOOK FOR ANALYZING AUDIO DATA

13.1 Introduction

Audio data - as recorded by smartphones or other portable devices - can carry important information about individuals' environments. This may give insights about the activity, sleep, and social interaction. However, using these data can be tricky due to privacy concerns, for example, conversations are highly identifiable. A possible solution is to compute more general characteristics (e.g. frequency) and use those instead to extract features. To address this last part, `niimpy` includes the function `extract_features_audio` to clean, downsample, and extract features from audio snippets that have been already anonymized. This function employs other functions to extract the following features:

- `audio_count_silent`: number of times when there has been some sound in the environment
- `audio_count_speech`: number of times when there has been some sound in the environment that matches the range of human speech frequency (65 - 255Hz)
- `audio_count_loud`: number of times when there has been some sound in the environment above 70dB
- `audio_min_freq`: minimum frequency of the recorded audio snippets
- `audio_max_freq`: maximum frequency of the recorded audio snippets
- `audio_mean_freq`: mean frequency of the recorded audio snippets
- `audio_median_freq`: median frequency of the recorded audio snippets
- `audio_std_freq`: standard deviation of the frequency of the recorded audio snippets
- `audio_min_db`: minimum decibels of the recorded audio snippets
- `audio_max_db`: maximum decibels of the recorded audio snippets
- `audio_mean_db`: mean decibels of the recorded audio snippets
- `audio_median_db`: median decibels of the recorded audio snippets
- `audio_std_db`: standard deviations of the recorded audio snippets decibels

In the following, we will analyze audio snippets provided by `niimpy` as an example to illustrate the use of niimpy's audio preprocessing functions.

13.2 2. Read data

Let's start by reading the example data provided in `niimpy`. These data have already been shaped in a format that meets the requirements of the data schema. Let's start by importing the needed modules. Firstly we will import the `niimpy` package and then we will import the module we will use (audio) and give it a short name for use convinience.

```
[1]: import niimpy
     from niimpy import config
     import niimpy.preprocessing.audio as au
     import pandas as pd
     import warnings
     warnings.filterwarnings("ignore")
```

Now let's read the example data provided in `niimpy`. The example data is in `csv` format, so we need to use the `read_csv` function. When reading the data, we can specify the timezone where the data was collected. This will help us handle daylight saving times easier. We can specify the timezone with the argument **tz**. The output is a dataframe. We can also check the number of rows and columns in the dataframe.

```
[2]: data = niimpy.read_csv(config.MULTIUSER_AWARE_AUDIO_PATH, tz='Europe/Helsinki')
     data.shape
```

```
[2]: (33, 7)
```

The data was succesfully read. We can see that there are 33 datapoints with 7 columns in the dataset. However, we do not know yet what the data really looks like, so let's have a quick look:

```
[3]: data.head()
```

```
[3]:
                                           user         device          time  \
2020-01-09 02:08:03.896000+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578528e+09
2020-01-09 02:38:03.896000+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09
2020-01-09 03:08:03.896000+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578532e+09
2020-01-09 03:38:03.896000+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578534e+09
2020-01-09 04:08:03.896000+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578536e+09

                                  is_silent  double_decibels  \
2020-01-09 02:08:03.896000+02:00          0               84
2020-01-09 02:38:03.896000+02:00          0               89
2020-01-09 03:08:03.896000+02:00          0               99
2020-01-09 03:38:03.896000+02:00          0               77
2020-01-09 04:08:03.896000+02:00          0               80

                                  double_frequency  \
2020-01-09 02:08:03.896000+02:00              4935
2020-01-09 02:38:03.896000+02:00              8734
2020-01-09 03:08:03.896000+02:00              1710
2020-01-09 03:38:03.896000+02:00              9054
2020-01-09 04:08:03.896000+02:00             12265

                                                         datetime
2020-01-09 02:08:03.896000+02:00 2020-01-09 02:08:03.896000+02:00
2020-01-09 02:38:03.896000+02:00 2020-01-09 02:38:03.896000+02:00
2020-01-09 03:08:03.896000+02:00 2020-01-09 03:08:03.896000+02:00
2020-01-09 03:38:03.896000+02:00 2020-01-09 03:38:03.896000+02:00
2020-01-09 04:08:03.896000+02:00 2020-01-09 04:08:03.896000+02:00
```

[4]:
```
data.tail()
```

[4]:
```
                                            user         device          time  \
2019-08-13 15:02:17.657999872+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565698e+09
2019-08-13 15:28:59.657999872+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565699e+09
2019-08-13 15:59:01.657999872+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565701e+09
2019-08-13 16:29:03.657999872+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565703e+09
2019-08-13 16:59:05.657999872+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565705e+09

                                     is_silent  double_decibels  \
2019-08-13 15:02:17.657999872+03:00          1               44
2019-08-13 15:28:59.657999872+03:00          1               49
2019-08-13 15:59:01.657999872+03:00          0               55
2019-08-13 16:29:03.657999872+03:00          0               76
2019-08-13 16:59:05.657999872+03:00          0               84

                                     double_frequency  \
2019-08-13 15:02:17.657999872+03:00              2914
2019-08-13 15:28:59.657999872+03:00              7195
2019-08-13 15:59:01.657999872+03:00                91
2019-08-13 16:29:03.657999872+03:00              3853
2019-08-13 16:59:05.657999872+03:00              7419

                                                               datetime
2019-08-13 15:02:17.657999872+03:00 2019-08-13 15:02:17.657999872+03:00
2019-08-13 15:28:59.657999872+03:00 2019-08-13 15:28:59.657999872+03:00
2019-08-13 15:59:01.657999872+03:00 2019-08-13 15:59:01.657999872+03:00
2019-08-13 16:29:03.657999872+03:00 2019-08-13 16:29:03.657999872+03:00
2019-08-13 16:59:05.657999872+03:00 2019-08-13 16:59:05.657999872+03:00
```

By exploring the head and tail of the dataframe we can form an idea of its entirety. From the data, we can see that:

- rows are observations, indexed by timestamps, i.e. each row represents a snippet that has been recorded at a given time and date
- columns are characteristics for each observation, for example, the user whose data we are analyzing
- there are at least two different users in the dataframe
- there are two main columns: `double_decibels` and `double_frequency`.

In fact, we can check the first three elements for each user

[5]:
```
data.drop_duplicates(['user','time']).groupby('user').head(3)
```

[5]:
```
                                            user         device          time  \
2020-01-09 02:08:03.896000+02:00     jd9INuQ5BBlW  3p83yASkOb_B  1.578528e+09
2020-01-09 02:38:03.896000+02:00     jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09
2020-01-09 03:08:03.896000+02:00     jd9INuQ5BBlW  3p83yASkOb_B  1.578532e+09
2019-08-13 07:28:27.657999872+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565671e+09
2019-08-13 07:58:29.657999872+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565672e+09
2019-08-13 08:28:31.657999872+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565674e+09

                                     is_silent  double_decibels  \
2020-01-09 02:08:03.896000+02:00             0               84
2020-01-09 02:38:03.896000+02:00             0               89
```

(continues on next page)

(continued from previous page)

```
2020-01-09 03:08:03.896000+02:00          0              99
2019-08-13 07:28:27.657999872+03:00       0              51
2019-08-13 07:58:29.657999872+03:00       0              90
2019-08-13 08:28:31.657999872+03:00       0              81

                                    double_frequency  \
2020-01-09 02:08:03.896000+02:00                4935
2020-01-09 02:38:03.896000+02:00                8734
2020-01-09 03:08:03.896000+02:00                1710
2019-08-13 07:28:27.657999872+03:00             7735
2019-08-13 07:58:29.657999872+03:00            13609
2019-08-13 08:28:31.657999872+03:00             7690

                                                               datetime
2020-01-09 02:08:03.896000+02:00       2020-01-09 02:08:03.896000+02:00
2020-01-09 02:38:03.896000+02:00       2020-01-09 02:38:03.896000+02:00
2020-01-09 03:08:03.896000+02:00       2020-01-09 03:08:03.896000+02:00
2019-08-13 07:28:27.657999872+03:00 2019-08-13 07:28:27.657999872+03:00
2019-08-13 07:58:29.657999872+03:00 2019-08-13 07:58:29.657999872+03:00
2019-08-13 08:28:31.657999872+03:00 2019-08-13 08:28:31.657999872+03:00
```

Sometimes the data may come in a disordered manner, so just to make sure, let's order the dataframe and compare the results. We will use the columns "user" and "datetime" since we would like to order the information according to firstly, participants, and then, by time in order of happening. Luckily, in our dataframe, the index and datetime are the same.

[6]:
```
data.sort_values(by=['user', 'datetime'], inplace=True)
data.drop_duplicates(['user','time']).groupby('user').head(3)
```

[6]:
```
                                            user         device          time  \
2019-08-13 07:28:27.657999872+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565671e+09
2019-08-13 07:58:29.657999872+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565672e+09
2019-08-13 08:28:31.657999872+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565674e+09
2020-01-09 02:08:03.896000+02:00     jd9INuQ5BBlW  3p83yASkOb_B  1.578528e+09
2020-01-09 02:38:03.896000+02:00     jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09
2020-01-09 03:08:03.896000+02:00     jd9INuQ5BBlW  3p83yASkOb_B  1.578532e+09

                                     is_silent  double_decibels  \
2019-08-13 07:28:27.657999872+03:00          0               51
2019-08-13 07:58:29.657999872+03:00          0               90
2019-08-13 08:28:31.657999872+03:00          0               81
2020-01-09 02:08:03.896000+02:00             0               84
2020-01-09 02:38:03.896000+02:00             0               89
2020-01-09 03:08:03.896000+02:00             0               99

                                     double_frequency  \
2019-08-13 07:28:27.657999872+03:00              7735
2019-08-13 07:58:29.657999872+03:00             13609
2019-08-13 08:28:31.657999872+03:00              7690
2020-01-09 02:08:03.896000+02:00                 4935
2020-01-09 02:38:03.896000+02:00                 8734
2020-01-09 03:08:03.896000+02:00                 1710
```

(continues on next page)

(continued from previous page)

```
                                                            datetime
2019-08-13 07:28:27.657999872+03:00 2019-08-13 07:28:27.657999872+03:00
2019-08-13 07:58:29.657999872+03:00 2019-08-13 07:58:29.657999872+03:00
2019-08-13 08:28:31.657999872+03:00 2019-08-13 08:28:31.657999872+03:00
2020-01-09 02:08:03.896000+02:00       2020-01-09 02:08:03.896000+02:00
2020-01-09 02:38:03.896000+02:00       2020-01-09 02:38:03.896000+02:00
2020-01-09 03:08:03.896000+02:00       2020-01-09 03:08:03.896000+02:00
```

Ok, it seems like our dataframe was in order. We can start extracting features. However, we need to understand the data format requirements first.

13.3 * TIP! Data format requirements (or what should our data look like)

Data can take other shapes and formats. However, the `niimpy` data schema requires it to be in a certain shape. This means the dataframe needs to have at least the following characteristics: 1. One row per call. Each row should store information about one call only 2. Each row's index should be a timestamp 3. The following five columns are required: - index: date and time when the event happened (timestamp) - user: stores the user name whose data is analyzed. Each user should have a unique name or hash (i.e. one hash for each unique user) - is_silent: stores whether the decibel level is above a set threshold (usually 50dB) - double_decibels: stores the decibels of the recorded snippet - double_frequency: the frequency of the recorded snippet in Hz - NOTE: most of our audio examples come from data recorded with the Aware Framework, if you want to know more about the frequency and decibels, please read https://github.com/denzilferreira/com.aware.plugin.ambient_noise 4. Additional columns are allowed. 5. The names of the columns do not need to be exactly "user", "is_silent", "double_decibels" or "double_frequency" as we can pass our own names in an argument (to be explained later).

Below is an example of a dataframe that complies with these minimum requirements

[7]:
```
example_dataschema = data[['user','is_silent','double_decibels','double_frequency']]
example_dataschema.head(3)
```

[7]:
```
                                             user  is_silent  double_decibels  \
2019-08-13 07:28:27.657999872+03:00  iGyXetHE3S8u          0               51
2019-08-13 07:58:29.657999872+03:00  iGyXetHE3S8u          0               90
2019-08-13 08:28:31.657999872+03:00  iGyXetHE3S8u          0               81

                                     double_frequency
2019-08-13 07:28:27.657999872+03:00              7735
2019-08-13 07:58:29.657999872+03:00             13609
2019-08-13 08:28:31.657999872+03:00              7690
```

13.4 4. Extracting features

There are two ways to extract features. We could use each function separately or we could use `niimpy`'s ready-made wrapper. Both ways will require us to specify arguments to pass to the functions/wrapper in order to customize the way the functions work. These arguments are specified in dictionaries. Let's first understand how to extract features using stand-alone functions.

13.4.1 4.1 Extract features using stand-alone functions

We can use `niimpy`'s functions to compute communication features. Each function will require two inputs: - (mandatory) dataframe that must comply with the minimum requirements (see '* TIP! Data requirements above) - (optional) an argument dictionary for stand-alone functions

4.1.1 The argument dictionary for stand-alone functions (or how we specify the way a function works)

In this dictionary, we can input two main features to customize the way a stand-alone function works: - the name of the columns to be preprocessed: Since the dataframe may have different columns, we need to specify which column has the data we would like to be preprocessed. To do so, we can simply pass the name of the column to the argument `audio_column_name`.

- the way we resample: resampling options are specified in `niimpy` as a dictionary. `niimpy`'s resampling and aggregating relies on `pandas.DataFrame.resample`, so mastering the use of this pandas function will help us greatly in `niimpy`'s preprocessing. Please familiarize yourself with the pandas resample function before continuing. Briefly, to use the `pandas.DataFrame.resample` function, we need a rule. This rule states the intervals we would like to use to resample our data (e.g., 15-seconds, 30-minutes, 1-hour). Neverthless, we can input more details into the function to specify the exact sampling we would like. For example, we could use the *close* argument if we would like to specify which side of the interval is closed, or we could use the *offset* argument if we would like to start our binning with an offset, etc. There are plenty of options to use this command, so we strongly recommend having `pandas.DataFrame.resample` documentation at hand. All features for the `pandas.DataFrame.resample` will be specified in a dictionary where keys are the arguments' names for the `pandas.DataFrame.resample`, and the dictionary's values are the values for each of these selected arguments. This dictionary will be passed as a value to the key `resample_args` in `niimpy`.

Let's see some basic examples of these dictionaries:

[8]:
```
feature_dict1:{"audio_column_name":"double_frequency","resample_args":{"rule":"1D"}}
feature_dict2:{"audio_column_name":"random_name","resample_args":{"rule":"30T"}}
feature_dict3:{"audio_column_name":"other_name","resample_args":{"rule":"45T","origin":
↪"end"}}
```

Here, we have three basic feature dictionaries.

- `feature_dict1` will be used to analyze the data stored in the column `double_frequency` in our dataframe. The data will be binned in one day periods
- `feature_dict2` will be used to analyze the data stored in the column `random_name` in our dataframe. The data will be aggregated in 30-minutes bins
- `feature_dict3` will be used to analyze the data stored in the column `other_name` in our dataframe. The data will be binned in 45-minutes bins, but the binning will start from the last timestamp in the dataframe.

Default values: if no arguments are passed, `niimpy`'s will aggregate the data in 30-min bins, and will select the audio_column_name according to the most suitable column. For example, if we are computing the minimum frequency, `niimpy` will select *double_frquency* as the column name.

4.1.2 Using the functions

Now that we understand how the functions are customized, it is time we compute our first audio feature. Suppose that we are interested in extracting the total number of times our recordings were loud every 50 minutes. We will need `niimpy`'s `audio_count_loud` function, the data, and we will also need to create a dictionary to customize our function. Let's create the dictionary first

[9]:
```
function_features={"audio_column_name":"double_decibels","resample_args":{"rule":"50T"}}
```

Now let's use the function to preprocess the data.

[10]:
```
my_loud_times = au.audio_count_loud(data, function_features)
```

`my_loud_times` is a multiindex dataframe, where the first level is the user, and the second level is the aggregated timestamp. Let's look at some values for one of the subjects.

[11]:
```
my_loud_times.xs("jd9INuQ5BBlW", level="user")
```

[11]:
```
                           audio_count_loud
2020-01-09 01:40:00+02:00                 1
2020-01-09 02:30:00+02:00                 2
2020-01-09 03:20:00+02:00                 2
2020-01-09 04:10:00+02:00                 0
2020-01-09 05:00:00+02:00                 1
2020-01-09 05:50:00+02:00                 1
2020-01-09 06:40:00+02:00                 1
2020-01-09 07:30:00+02:00                 0
2020-01-09 08:20:00+02:00                 1
2020-01-09 09:10:00+02:00                 1
2020-01-09 10:00:00+02:00                 2
```

Let's remember how the original data looks like for this subject

[12]:
```
data[data["user"]=="jd9INuQ5BBlW"].head(7)
```

[12]:
```
                                          user         device          time  \
2020-01-09 02:08:03.896000+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578528e+09
2020-01-09 02:38:03.896000+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09
2020-01-09 03:08:03.896000+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578532e+09
2020-01-09 03:38:03.896000+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578534e+09
2020-01-09 04:08:03.896000+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578536e+09
2020-01-09 04:38:03.896000+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578537e+09
2020-01-09 05:08:03.896000+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578539e+09

                                  is_silent  double_decibels  \
2020-01-09 02:08:03.896000+02:00          0               84
2020-01-09 02:38:03.896000+02:00          0               89
2020-01-09 03:08:03.896000+02:00          0               99
2020-01-09 03:38:03.896000+02:00          0               77
2020-01-09 04:08:03.896000+02:00          0               80
2020-01-09 04:38:03.896000+02:00          0               52
2020-01-09 05:08:03.896000+02:00          0               63

                                  double_frequency  \
2020-01-09 02:08:03.896000+02:00              4935
```

(continues on next page)

(continued from previous page)

```
2020-01-09 02:38:03.896000+02:00           8734
2020-01-09 03:08:03.896000+02:00           1710
2020-01-09 03:38:03.896000+02:00           9054
2020-01-09 04:08:03.896000+02:00          12265
2020-01-09 04:38:03.896000+02:00           7281
2020-01-09 05:08:03.896000+02:00          14408

                                                          datetime
2020-01-09 02:08:03.896000+02:00 2020-01-09 02:08:03.896000+02:00
2020-01-09 02:38:03.896000+02:00 2020-01-09 02:38:03.896000+02:00
2020-01-09 03:08:03.896000+02:00 2020-01-09 03:08:03.896000+02:00
2020-01-09 03:38:03.896000+02:00 2020-01-09 03:38:03.896000+02:00
2020-01-09 04:08:03.896000+02:00 2020-01-09 04:08:03.896000+02:00
2020-01-09 04:38:03.896000+02:00 2020-01-09 04:38:03.896000+02:00
2020-01-09 05:08:03.896000+02:00 2020-01-09 05:08:03.896000+02:00
```

We see that the bins are indeed 50-minutes bins, however, they are adjusted to fixed, predetermined intervals, i.e. the bin does not start on the time of the first datapoint. Instead, `pandas` starts the binning at 00:00:00 of everyday and counts 50-minutes intervals from there.

If we want the binning to start from the first datapoint in our dataset, we need the origin parameter and a for loop.

[13]:
```
users = list(data['user'].unique())
results = []
for user in users:
    start_time = data[data["user"]==user].index.min()
    function_features={"audio_column_name":"double_decibels","resample_args":{"rule":"50T
↪","origin":start_time}}
    results.append(au.audio_count_loud(data[data["user"]==user], function_features))
my_loud_times = pd.concat(results)
```

[14]:
```
my_loud_times
```

[14]:
```
                                                 audio_count_loud
user
iGyXetHE3S8u 2019-08-13 07:28:27.657999872+03:00                1
             2019-08-13 08:18:27.657999872+03:00                1
             2019-08-13 09:08:27.657999872+03:00                0
             2019-08-13 09:58:27.657999872+03:00                2
             2019-08-13 10:48:27.657999872+03:00                2
             2019-08-13 11:38:27.657999872+03:00                1
             2019-08-13 12:28:27.657999872+03:00                0
             2019-08-13 13:18:27.657999872+03:00                0
             2019-08-13 14:08:27.657999872+03:00                1
             2019-08-13 14:58:27.657999872+03:00                0
             2019-08-13 15:48:27.657999872+03:00                1
             2019-08-13 16:38:27.657999872+03:00                1
jd9INuQ5BBlW 2020-01-09 02:08:03.896000+02:00                   2
             2020-01-09 02:58:03.896000+02:00                   2
             2020-01-09 03:48:03.896000+02:00                   1
             2020-01-09 04:38:03.896000+02:00                   0
             2020-01-09 05:28:03.896000+02:00                   2
             2020-01-09 06:18:03.896000+02:00                   0
```

(continues on next page)

(continued from previous page)

```
            2020-01-09 07:08:03.896000+02:00                    1
            2020-01-09 07:58:03.896000+02:00                    0
            2020-01-09 08:48:03.896000+02:00                    1
            2020-01-09 09:38:03.896000+02:00                    2
            2020-01-09 10:28:03.896000+02:00                    1
```

13.4.2 4.2 Extract features using the wrapper

We can use `niimpy`'s ready-made wrapper to extract one or several features at the same time. The wrapper will require two inputs: - (mandatory) dataframe that must comply with the minimum requirements (see '* TIP! Data requirements above) - (optional) an argument dictionary for wrapper

4.2.1 The argument dictionary for wrapper (or how we specify the way the wrapper works)

This argument dictionary will use dictionaries created for stand-alone functions. If you do not know how to create those argument dictionaries, please read the section **4.1.1 The argument dictionary for stand-alone functions (or how we specify the way a function works)** first.

The wrapper dictionary is simple. Its keys are the names of the features we want to compute. Its values are argument dictionaries created for each stand-alone function we will employ. Let's see some examples of wrapper dictionaries:

[15]:
```
wrapper_features1 = {au.audio_count_loud:{"audio_column_name":"double_decibels",
↪"resample_args":{"rule":"1D"}},
                     au.audio_max_freq:{"audio_column_name":"double_frequency","resample_
↪args":{"rule":"1D"}}}
```

- `wrapper_features1` will be used to analyze two features, `audio_count_loud` and `audio_max_freq`. For the feature audio_count_loud, we will use the data stored in the column `double_decibels` in our dataframe and the data will be binned in one day periods. For the feature audio_max_freq, we will use the data stored in the column `double_frequency` in our dataframe and the data will be binned in one day periods.

[16]:
```
wrapper_features2 = {au.audio_mean_db:{"audio_column_name":"random_name","resample_args":
↪{"rule":"1D"}},
                     au.audio_count_speech:{"audio_column_name":"double_decibels",
↪"audio_freq_name":"double_frequency", "resample_args":{"rule":"5H","offset":"5min"}}}
```

- `wrapper_features2` will be used to analyze two features, `audio_mean_db` and `audio_count_speech`. For the feature audio_mean_db, we will use the data stored in the column `random_name` in our dataframe and the data will be binned in one day periods. For the feature audio_count_speech, we will use the data stored in the column `double_decibels` in our dataframe and the data will be binned in 5-hour periods with a 5-minute offset. Note that for this feature we will also need another column named "audio_freq_column", this is because the speech is not only defined by the amplitude of the recording, but the frequency range.

[17]:
```
wrapper_features3 = {au.audio_mean_db:{"audio_column_name":"one_name","resample_args":{
↪"rule":"1D","offset":"5min"}},
                     au.audio_min_freq:{"audio_column_name":"one_name","resample_args":{
↪"rule":"5H"}},
                     au.audio_count_silent:{"audio_column_name":"another_name","resample_
↪args":{"rule":"30T","origin":"end_day"}}}
```

- `wrapper_features3` will be used to analyze three features, `audio_mean_db`, `audio_min_freq`, and `audio_count_silent`. For the feature audio_mean_db, we will use the data stored in the column `one_name` and the data will be binned in one day periods with a 5-min offset. For the feature audio_min_freq, we will use the data stored in the column `one_name` in our dataframe and the data will be binned in 5-hour periods. Finally, for the feature audio_count_silent, we will use the data stored in the column `another_name` in our dataframe and the data will be binned in 30-minute periods and the origin of the bins will be the ceiling midnight of the last day.

Default values: if no arguments are passed, `niimpy`'s default values are either "double_decibels", "double_frequency", or "is_silent" for the communication_column_name, and 30-min aggregation bins. The column name depends on the function to be called. Moreover, the wrapper will compute all the available functions in absence of the argument dictionary.

4.2.2 Using the wrapper

Now that we understand how the wrapper is customized, it is time we compute our first communication feature using the wrapper. Suppose that we are interested in extracting the audio_count_loud duration every 50 minutes. We will need `niimpy`'s `extract_features_audio` function, the data, and we will also need to create a dictionary to customize our function. Let's create the dictionary first

[18]:
```
wrapper_features1 = {au.audio_count_loud:{"audio_column_name":"double_decibels",
↪"resample_args":{"rule":"50T"}}}
```

Now, let's use the wrapper

[19]:
```
results_wrapper = au.extract_features_audio(data, features=wrapper_features1)
results_wrapper.head(5)
```

```
computing <function audio_count_loud at 0x0000021328494C10>...
```

[19]:
```
                                          audio_count_loud
user
iGyXetHE3S8u 2019-08-13 07:30:00+03:00                  1
             2019-08-13 08:20:00+03:00                  1
             2019-08-13 09:10:00+03:00                  1
             2019-08-13 10:00:00+03:00                  1
             2019-08-13 10:50:00+03:00                  2
```

Our first attempt was succesful. Now, let's try something more. Let's assume we want to compute the audio_count_loud and audio_min_freq in 1-hour bins.

[20]:
```
wrapper_features2 = {au.audio_count_loud:{"audio_column_name":"double_decibels",
↪"resample_args":{"rule":"1H"}},
                     au.audio_min_freq:{"audio_column_name":"double_frequency",
↪"resample_args":{"rule":"1H"}}}
results_wrapper = au.extract_features_audio(data, features=wrapper_features2)
results_wrapper.head(5)
```

```
computing <function audio_count_loud at 0x0000021328494C10>...
computing <function audio_min_freq at 0x0000021328494CA0>...
```

[20]:
```
                                          audio_count_loud  audio_min_freq
user
iGyXetHE3S8u 2019-08-13 07:00:00+03:00                  1          7735.0
             2019-08-13 08:00:00+03:00                  1          7690.0
             2019-08-13 09:00:00+03:00                  1           756.0
             2019-08-13 10:00:00+03:00                  2          3059.0
             2019-08-13 11:00:00+03:00                  2         12278.0
```

Great! Another successful attempt. We see from the results that more columns were added with the required calculations. This is how the wrapper works when all features are computed with the same bins. Now, let's see how the wrapper performs when each function has different binning requirements. Let's assume we need to compute the audio_count_loud every day, and the audio_min_freq every 5 hours with an offset of 5 minutes.

[21]:
```
wrapper_features3 = {au.audio_count_loud:{"audio_column_name":"double_decibels",
↪"resample_args":{"rule":"1D"}},
                     au.audio_min_freq:{"audio_column_name":"double_frequency",
↪"resample_args":{"rule":"5H", "offset":"5min"}}}
results_wrapper = au.extract_features_audio(data, features=wrapper_features3)
results_wrapper.head(5)
```

```
computing <function audio_count_loud at 0x0000021328494C10>...
computing <function audio_min_freq at 0x0000021328494CA0>...
```

[21]:
```
                                        audio_count_loud  audio_min_freq
user
iGyXetHE3S8u 2019-08-13 00:00:00+03:00              10.0             NaN
jd9INuQ5BBlW 2020-01-09 00:00:00+02:00              12.0             NaN
iGyXetHE3S8u 2019-08-13 05:05:00+03:00               NaN           756.0
             2019-08-13 10:05:00+03:00               NaN          2914.0
             2019-08-13 15:05:00+03:00               NaN            91.0
```

The output is once again a dataframe. In this case, two aggregations are shown. The first one is the daily aggregation computed for the `audio_count_loud` feature. The second one is the 5-hour aggregation period with 5-min offset for the `audio_min_freq`. We must note that because the `audio_min_freq`feature is not required to be aggregated daily, the daily aggregation timestamps have a NaN value. Similarly, because the `audio_count_loud`is not required to be aggregated in 5-hour windows, its values are NaN for all subjects.

4.2.3 Wrapper and its default option

The default option will compute all features in 30-minute aggregation windows. To use the `extract_features_audio` function with its default options, simply call the function.

[22]:
```
default = au.extract_features_audio(data, features=None)
```

```
computing <function audio_count_silent at 0x00000213674A35B0>...
computing <function audio_count_speech at 0x0000021328494B80>...
computing <function audio_count_loud at 0x0000021328494C10>...
computing <function audio_min_freq at 0x0000021328494CA0>...
computing <function audio_max_freq at 0x0000021328494D30>...
computing <function audio_mean_freq at 0x0000021328494DC0>...
computing <function audio_median_freq at 0x0000021328494E50>...
computing <function audio_std_freq at 0x0000021328494EE0>...
computing <function audio_min_db at 0x0000021328494F70>...
computing <function audio_max_db at 0x0000021328495000>...
computing <function audio_mean_db at 0x0000021328495090>...
computing <function audio_median_db at 0x0000021328495120>...
computing <function audio_std_db at 0x00000213284951B0>...
```

[23]:
```
default.head()
```

[23]:
```
                                        audio_count_silent  \
user
iGyXetHE3S8u 2019-08-13 07:00:00+03:00                   0
```

(continues on next page)

(continued from previous page)

```
             2019-08-13 07:30:00+03:00                   0
             2019-08-13 08:00:00+03:00                   0
             2019-08-13 08:30:00+03:00                   0
             2019-08-13 09:00:00+03:00                   1

                                        audio_count_speech  audio_count_loud  \
user
iGyXetHE3S8u 2019-08-13 07:00:00+03:00                 NaN               NaN
             2019-08-13 07:30:00+03:00                 NaN               1.0
             2019-08-13 08:00:00+03:00                 NaN               1.0
             2019-08-13 08:30:00+03:00                 NaN               0.0
             2019-08-13 09:00:00+03:00                 NaN               0.0

                                        audio_min_freq  audio_max_freq  \
user
iGyXetHE3S8u 2019-08-13 07:00:00+03:00          7735.0          7735.0
             2019-08-13 07:30:00+03:00         13609.0         13609.0
             2019-08-13 08:00:00+03:00          7690.0          7690.0
             2019-08-13 08:30:00+03:00          8347.0          8347.0
             2019-08-13 09:00:00+03:00         13592.0         13592.0

                                        audio_mean_freq  audio_median_freq  \
user
iGyXetHE3S8u 2019-08-13 07:00:00+03:00           7735.0             7735.0
             2019-08-13 07:30:00+03:00          13609.0            13609.0
             2019-08-13 08:00:00+03:00           7690.0             7690.0
             2019-08-13 08:30:00+03:00           8347.0             8347.0
             2019-08-13 09:00:00+03:00          13592.0            13592.0

                                        audio_std_freq  audio_min_db  \
user
iGyXetHE3S8u 2019-08-13 07:00:00+03:00             NaN          51.0
             2019-08-13 07:30:00+03:00             NaN          90.0
             2019-08-13 08:00:00+03:00             NaN          81.0
             2019-08-13 08:30:00+03:00             NaN          58.0
             2019-08-13 09:00:00+03:00             NaN          36.0

                                        audio_max_db  audio_mean_db  \
user
iGyXetHE3S8u 2019-08-13 07:00:00+03:00          51.0           51.0
             2019-08-13 07:30:00+03:00          90.0           90.0
             2019-08-13 08:00:00+03:00          81.0           81.0
             2019-08-13 08:30:00+03:00          58.0           58.0
             2019-08-13 09:00:00+03:00          36.0           36.0

                                        audio_median_db  audio_std_db
user
iGyXetHE3S8u 2019-08-13 07:00:00+03:00             51.0           NaN
             2019-08-13 07:30:00+03:00             90.0           NaN
             2019-08-13 08:00:00+03:00             81.0           NaN
             2019-08-13 08:30:00+03:00             58.0           NaN
             2019-08-13 09:00:00+03:00             36.0           NaN
```

13.5 5. Implementing own features

If none of the provided functions suits well, We can implement our own customized features easily. To do so, we need to define a function that accepts a dataframe and returns a dataframe. The returned object should be indexed by user and timestamps (multiindex). To make the feature readily available in the default options, we need add the *audio* prefix to the new function (e.g. `audio_my-new-feature`). Let's assume we need a new function that counts sums all frequencies. Let's first define the function

[24]:
```
def audio_sum_freq(df,feature_functions=None):
    if not "audio_column_name" in feature_functions:
        col_name = "double_frequency"
    else:
        col_name = feature_functions["audio_column_name"]
    if not "resample_args" in feature_functions.keys():
        feature_functions["resample_args"] = {"rule":"30T"}

    if len(df)>0:
        result = df.groupby('user')[col_name].resample(**feature_functions["resample_args
↪"]).sum()
        result = result.to_frame(name='audio_sum_freq')
    return result
```

Then, we can call our new function in the stand-alone way or using the `extract_features_audio` function. Because the stand-alone way is the common way to call functions in python, we will not show it. Instead, we will show how to integrate this new function to the wrapper. Let's read again the data and assume we want the default behavior of the wrapper.

[25]:
```
customized_features = au.extract_features_audio(data, features={audio_sum_freq: {}})
```

```
computing <function audio_sum_freq at 0x0000021328557BE0>...
```

[26]:
```
customized_features.head()
```

[26]:
```
                                          audio_sum_freq
user
iGyXetHE3S8u 2019-08-13 07:00:00+03:00             7735
             2019-08-13 07:30:00+03:00            13609
             2019-08-13 08:00:00+03:00             7690
             2019-08-13 08:30:00+03:00             8347
             2019-08-13 09:00:00+03:00            13592
```

[ ]:
```
```

CHAPTER

FOURTEEN

DEMO NOTEBOOK: ANALYSING BATTERY DATA

14.1 Read data

[1]:
```
import pandas as pd
import niimpy
import niimpy.preprocessing.battery as battery
from niimpy import config
import warnings
warnings.filterwarnings("ignore")
```

[2]:
```
data = niimpy.read_csv(config.MULTIUSER_AWARE_BATTERY_PATH, tz='Europe/Helsinki')
data.shape
```

[2]:
```
(505, 8)
```

14.2 Introduction

In this notebook , we will extract battery data from the Aware platform and infer users' behavioral patterns from their interaction with the phone. The below functions will be described in this notebook:

- `niimpy.preprocessing.battery.battery_shutdown_info`: returns the timestamp when the device is shutdown or rebooted
- `niimpy.preprocessing.battery.battery_occurrences`: returns the number of battery samples within a time range
- `niimpy.preprocessing.battery.battery_gaps`: returns the time gaps between two battery sample

[3]:
```
data.head()
```

[3]:
```
                                            user         device          time  
2020-01-09 02:20:02.924999936+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09   \
2020-01-09 02:21:30.405999872+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:24:12.805999872+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:35:38.561000192+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09
2020-01-09 02:35:38.953000192+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09

                                     battery_level  battery_status  
2020-01-09 02:20:02.924999936+02:00             74               3  \
2020-01-09 02:21:30.405999872+02:00             73               3
```

(continues on next page)

(continued from previous page)

```
2020-01-09 02:24:12.805999872+02:00          72               3
2020-01-09 02:35:38.561000192+02:00          72               2
2020-01-09 02:35:38.953000192+02:00          72               2

                                     battery_health  battery_adaptor
2020-01-09 02:20:02.924999936+02:00               2                0  \
2020-01-09 02:21:30.405999872+02:00               2                0
2020-01-09 02:24:12.805999872+02:00               2                0
2020-01-09 02:35:38.561000192+02:00               2                0
2020-01-09 02:35:38.953000192+02:00               2                2

                                                                datetime
2020-01-09 02:20:02.924999936+02:00 2020-01-09 02:20:02.924999936+02:00
2020-01-09 02:21:30.405999872+02:00 2020-01-09 02:21:30.405999872+02:00
2020-01-09 02:24:12.805999872+02:00 2020-01-09 02:24:12.805999872+02:00
2020-01-09 02:35:38.561000192+02:00 2020-01-09 02:35:38.561000192+02:00
2020-01-09 02:35:38.953000192+02:00 2020-01-09 02:35:38.953000192+02:00
```

CHAPTER

FIFTEEN

FEATURE EXTRACTION

By default, Niimpy data should be ordered by the timestamp in ascending order. We start by sorting the data to make sure it's compatible.

[4]:
```
data = data.sort_index()
```

Next, we will use Niimpy to extract features from the data. These are useful for inspecting the data and can be part of a full analysis workflow.

Usin the `battery_occurences` function, we can count the amount the battery samples every 10 minutes. This function requires the index to be sorted.

[5]:
```
battery.battery_occurrences(data, {"resample_args": {"rule": "10T"}})
```

[5]:
```
                                          occurrences
user
iGyXetHE3S8u 2019-08-05 14:00:00+03:00            2
             2019-08-05 14:10:00+03:00            0
             2019-08-05 14:20:00+03:00            0
             2019-08-05 14:30:00+03:00            1
             2019-08-05 14:40:00+03:00            0
...                                             ...
jd9INuQ5BBlW 2020-01-09 22:50:00+02:00            0
             2020-01-09 23:00:00+02:00            1
             2020-01-09 23:10:00+02:00            1
             2020-01-09 23:20:00+02:00            1
             2020-01-09 23:30:00+02:00            2

[626 rows x 1 columns]
```

The above dataframe gives the battery information of all users. You can also get the information for an individual by passing a filtered dataframe.

[6]:
```
f = niimpy.preprocessing.battery.battery_occurrences
data_filtered = data.query('user == "jd9INuQ5BBlW"')
individual_occurences = battery.extract_features_battery(data_filtered, feature_
↪functions={f: {"resample_args": {"rule": "10T"}}})
individual_occurences.head()
```

```
<function battery_occurrences at 0x7fd699bf72e0> {'resample_args': {'rule': '10T'}}
```

[6]:
```
                                          occurrences
user
jd9INuQ5BBlW 2020-01-09 02:00:00+02:00            3
```

(continues on next page)

(continued from previous page)

```
                 2020-01-09 02:10:00+02:00           1
                 2020-01-09 02:20:00+02:00           5
                 2020-01-09 02:30:00+02:00          16
                 2020-01-09 02:40:00+02:00          14
```

Next, you can extract the gaps between two consecutive battery samples with the `battery_gaps` function.

[7]:
```
f = niimpy.preprocessing.battery.battery_gaps
gaps = battery.battery_gaps(data, {})
gaps
```

[7]:
```
                                                   battery_gap
user
iGyXetHE3S8u 2019-08-05 14:00:00+03:00    0 days 00:01:18.600000
             2019-08-05 14:30:00+03:00    0 days 00:27:18.396000
             2019-08-05 15:00:00+03:00 0 days 00:51:11.997000192
             2019-08-05 15:30:00+03:00                       NaT
             2019-08-05 16:00:00+03:00 0 days 00:59:23.522999808
...                                                        ...
jd9INuQ5BBlW 2020-01-09 21:30:00+02:00 0 days 00:05:41.859499968
             2020-01-09 22:00:00+02:00 0 days 00:14:10.238500096
             2020-01-09 22:30:00+02:00 0 days 00:21:09.899999744
             2020-01-09 23:00:00+02:00 0 days 00:13:20.001333418
             2020-01-09 23:30:00+02:00 0 days 00:08:26.416999936

[210 rows x 1 columns]
```

Knowing when the phone is shutdown is essential if we want to infer the usage behaviour of the subjects. This can be done by calling the `shutdown_info` function. The function returns the timestamp when the phone is shut down or rebooted (e.g: battery_status = -1).

[8]:
```
shutdown = battery.shutdown_info(data, feature_functions={'battery_column_name':
↪'battery_status'})
shutdown
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/tmp/ipykernel_306302/2493873374.py in ?()
----> 1 shutdown = battery.shutdown_info(data, feature_functions={'battery_column_name':
↪'battery_status'})
      2 shutdown

~/src/niimpy/niimpy/preprocessing/battery.py in ?(df, feature_functions)
     29
     30     df[col_name] = pd.to_numeric(df[col_name]) #convert to numeric in case it is␣
↪not
     31
     32     shutdown = df[df[col_name].between(-3, 0, inclusive="neither")]
---> 33     return shutdown[col_name].to_dataframe()

~/miniconda3/envs/niimpy/lib/python3.11/site-packages/pandas/core/generic.py in ?(self,␣
↪name)
   5985             and name not in self._accessors
```

(continues on next page)

(continued from previous page)

```
   5986             and self._info_axis._can_hold_identifiers_and_holds_name(name)
   5987         ):
   5988             return self[name]
-> 5989         return object.__getattribute__(self, name)

AttributeError: 'Series' object has no attribute 'to_dataframe'
```

15.1 Extracting features with the extract_features call

We have seen above how to extract battery features using `niimpy`. Sometimes, we need more than one features and it would be inconvenient to extract everything one by one. `niimpy` provides a `extract_feature` call to allow you extracting all the features available and combining them into a single data frame. The extractable features must start with the prefix `battery_`.

```
[ ]: # Start by defining the feature name
f0 = niimpy.preprocessing.battery.battery_occurrences
f1 = niimpy.preprocessing.battery.battery_gaps
f2 = niimpy.preprocessing.battery.battery_charge_discharge

# The extract_feature function requires a feature_functions parameter.
# This parameter accepts a dictionary where the key is the feature name and value
# is a dictionary containing values passed to the function.
features = battery.extract_features_battery(
    data,
    feature_functions={f0: {'rule': "10min"},
    f1: {},
    f2: {}
})
features.head()
```

```
<function battery_occurrences at 0x7f15ba5bb2e0> {'rule': '10min'}
<function battery_gaps at 0x7f15ba5bb380> {}
<function battery_charge_discharge at 0x7f15ba5bb420> {}
```

```
                                         occurrences               battery_gap
user
iGyXetHE3S8u 2019-08-05 14:00:00+03:00            2    0 days 00:01:18.600000  \
             2019-08-05 14:30:00+03:00            1    0 days 00:27:18.396000
             2019-08-05 15:00:00+03:00            1 0 days 00:51:11.997000192
             2019-08-05 15:30:00+03:00            0                       NaT
             2019-08-05 16:00:00+03:00            1 0 days 00:59:23.522999808

                                         bdelta  charge/discharge
user
iGyXetHE3S8u 2019-08-05 14:00:00+03:00     -0.5         -0.006361
             2019-08-05 14:30:00+03:00     -1.0         -0.000610
             2019-08-05 15:00:00+03:00     -1.0         -0.000326
             2019-08-05 15:30:00+03:00      NaN               NaN
             2019-08-05 16:00:00+03:00     -1.0         -0.000281
```

CHAPTER

SIXTEEN

BASIC TRANSFORMATIONS

This page shows some basic transformations you can do once you have read data. Really, it is simply a pandas crash course, since pandas provides all the operations you may need and there is no need for us to re-invent things. Pandas provides a solid but flexible base for us to build advanced operations on top of.

You can read more at the Pandas documentation.

16.1 Extracting single rows and columns

Let's first import mobile phone battery status data.

[1]:
```
TZ = 'Europe/Helsinki'
```

[2]:
```
import niimpy
from niimpy import config
import warnings
warnings.filterwarnings("ignore")
```

[3]:
```
# Read the data
df = niimpy.read_csv(config.MULTIUSER_AWARE_BATTERY_PATH, tz='Europe/Helsinki')
```

Then check first rows of the dataframe.

[4]:
```
df.head()
```

[4]:
```
                                            user         device          time  \
2020-01-09 02:20:02.924999936+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:21:30.405999872+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:24:12.805999872+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:35:38.561000192+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09
2020-01-09 02:35:38.953000192+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09

                                     battery_level  battery_status  \
2020-01-09 02:20:02.924999936+02:00             74               3
2020-01-09 02:21:30.405999872+02:00             73               3
2020-01-09 02:24:12.805999872+02:00             72               3
2020-01-09 02:35:38.561000192+02:00             72               2
2020-01-09 02:35:38.953000192+02:00             72               2

                                     battery_health  battery_adaptor  \
```

(continues on next page)

(continued from previous page)

```
2020-01-09 02:20:02.924999936+02:00              2                0
2020-01-09 02:21:30.405999872+02:00              2                0
2020-01-09 02:24:12.805999872+02:00              2                0
2020-01-09 02:35:38.561000192+02:00              2                0
2020-01-09 02:35:38.953000192+02:00              2                2

                                                                datetime
2020-01-09 02:20:02.924999936+02:00 2020-01-09 02:20:02.924999936+02:00
2020-01-09 02:21:30.405999872+02:00 2020-01-09 02:21:30.405999872+02:00
2020-01-09 02:24:12.805999872+02:00 2020-01-09 02:24:12.805999872+02:00
2020-01-09 02:35:38.561000192+02:00 2020-01-09 02:35:38.561000192+02:00
2020-01-09 02:35:38.953000192+02:00 2020-01-09 02:35:38.953000192+02:00
```

Get a single column, in this case all **users**:

[5]:
```
df['user']
```

[5]:
```
2020-01-09 02:20:02.924999936+02:00    jd9INuQ5BBlW
2020-01-09 02:21:30.405999872+02:00    jd9INuQ5BBlW
2020-01-09 02:24:12.805999872+02:00    jd9INuQ5BBlW
2020-01-09 02:35:38.561000192+02:00    jd9INuQ5BBlW
2020-01-09 02:35:38.953000192+02:00    jd9INuQ5BBlW
                                           ...
2019-08-09 00:30:48.073999872+03:00    iGyXetHE3S8u
2019-08-09 00:32:40.717999872+03:00    iGyXetHE3S8u
2019-08-09 00:34:23.114000128+03:00    iGyXetHE3S8u
2019-08-09 00:36:05.505000192+03:00    iGyXetHE3S8u
2019-08-09 00:37:37.671000064+03:00    iGyXetHE3S8u
Name: user, Length: 505, dtype: object
```

Get a single row, in this case the **5th** (the first row is zero):

[6]:
```
df.iloc[4]
```

[6]:
```
user                                      jd9INuQ5BBlW
device                                    3p83yASkOb_B
time                                    1578530138.953
battery_level                                       72
battery_status                                       2
battery_health                                       2
battery_adaptor                                      2
datetime               2020-01-09 02:35:38.953000192+02:00
Name: 2020-01-09 02:35:38.953000192+02:00, dtype: object
```

16.2 Listing unique users

We can list unique users by using `pandas.unique()` function.

[7]:
```
df['user'].unique()
```

[7]:
```
array(['jd9INuQ5BBlW', 'iGyXetHE3S8u'], dtype=object)
```

16.3 List unique values

Same applies to other data features/columns.

[8]:
```
df['battery_status'].unique()
```

[8]:
```
array([ 3,  2,  5, -1, -3], dtype=int64)
```

16.4 Extract data of only one subject

We can extract data of only one subject by following:

[9]:
```
df[df['user'] == 'jd9INuQ5BBlW']
```

[9]:
```
                                          user        device          time  \
2020-01-09 02:20:02.924999936+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:21:30.405999872+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:24:12.805999872+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:35:38.561000192+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09
2020-01-09 02:35:38.953000192+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09
...                                           ...           ...           ...
2020-01-09 23:02:13.938999808+02:00  jd9INuQ5BBlW  OWd1Uau8POix  1.578604e+09
2020-01-09 23:10:37.262000128+02:00  jd9INuQ5BBlW  OWd1Uau8POix  1.578604e+09
2020-01-09 23:22:13.966000128+02:00  jd9INuQ5BBlW  OWd1Uau8POix  1.578605e+09
2020-01-09 23:32:13.959000064+02:00  jd9INuQ5BBlW  OWd1Uau8POix  1.578606e+09
2020-01-09 23:39:06.800000+02:00     jd9INuQ5BBlW  OWd1Uau8POix  1.578606e+09

                                     battery_level  battery_status  \
2020-01-09 02:20:02.924999936+02:00             74               3
2020-01-09 02:21:30.405999872+02:00             73               3
2020-01-09 02:24:12.805999872+02:00             72               3
2020-01-09 02:35:38.561000192+02:00             72               2
2020-01-09 02:35:38.953000192+02:00             72               2
...                                            ...             ...
2020-01-09 23:02:13.938999808+02:00             73               3
2020-01-09 23:10:37.262000128+02:00             73               3
2020-01-09 23:22:13.966000128+02:00             72               3
2020-01-09 23:32:13.959000064+02:00             71               3
2020-01-09 23:39:06.800000+02:00                71               3

                                     battery_health  battery_adaptor  \
2020-01-09 02:20:02.924999936+02:00               2                0
```

(continues on next page)

(continued from previous page)

```
2020-01-09 02:21:30.405999872+02:00               2                0
2020-01-09 02:24:12.805999872+02:00               2                0
2020-01-09 02:35:38.561000192+02:00               2                0
2020-01-09 02:35:38.953000192+02:00               2                2
...                                             ...              ...
2020-01-09 23:02:13.938999808+02:00               2                0
2020-01-09 23:10:37.262000128+02:00               2                0
2020-01-09 23:22:13.966000128+02:00               2                0
2020-01-09 23:32:13.959000064+02:00               2                0
2020-01-09 23:39:06.800000+02:00                  2                0

                                                               datetime
2020-01-09 02:20:02.924999936+02:00 2020-01-09 02:20:02.924999936+02:00
2020-01-09 02:21:30.405999872+02:00 2020-01-09 02:21:30.405999872+02:00
2020-01-09 02:24:12.805999872+02:00 2020-01-09 02:24:12.805999872+02:00
2020-01-09 02:35:38.561000192+02:00 2020-01-09 02:35:38.561000192+02:00
2020-01-09 02:35:38.953000192+02:00 2020-01-09 02:35:38.953000192+02:00
...                                                                 ...
2020-01-09 23:02:13.938999808+02:00 2020-01-09 23:02:13.938999808+02:00
2020-01-09 23:10:37.262000128+02:00 2020-01-09 23:10:37.262000128+02:00
2020-01-09 23:22:13.966000128+02:00 2020-01-09 23:22:13.966000128+02:00
2020-01-09 23:32:13.959000064+02:00 2020-01-09 23:32:13.959000064+02:00
2020-01-09 23:39:06.800000+02:00       2020-01-09 23:39:06.800000+02:00

[373 rows x 8 columns]
```

16.5 Renaming a column or columns

Dataframe column can be renamed using `pandas.DataFrame.rename()` function.

[10]:
```
df.rename(columns={'time': 'timestamp'}, inplace=True)
df.head()
```

[10]:
```
                                             user         device     timestamp  \
2020-01-09 02:20:02.924999936+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:21:30.405999872+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:24:12.805999872+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:35:38.561000192+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09
2020-01-09 02:35:38.953000192+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09

                                     battery_level  battery_status  \
2020-01-09 02:20:02.924999936+02:00             74               3
2020-01-09 02:21:30.405999872+02:00             73               3
2020-01-09 02:24:12.805999872+02:00             72               3
2020-01-09 02:35:38.561000192+02:00             72               2
2020-01-09 02:35:38.953000192+02:00             72               2

                                     battery_health  battery_adaptor  \
2020-01-09 02:20:02.924999936+02:00               2                0
2020-01-09 02:21:30.405999872+02:00               2                0
```

(continues on next page)

(continued from previous page)

```
2020-01-09 02:24:12.805999872+02:00              2                0
2020-01-09 02:35:38.561000192+02:00              2                0
2020-01-09 02:35:38.953000192+02:00              2                2

                                                              datetime
2020-01-09 02:20:02.924999936+02:00 2020-01-09 02:20:02.924999936+02:00
2020-01-09 02:21:30.405999872+02:00 2020-01-09 02:21:30.405999872+02:00
2020-01-09 02:24:12.805999872+02:00 2020-01-09 02:24:12.805999872+02:00
2020-01-09 02:35:38.561000192+02:00 2020-01-09 02:35:38.561000192+02:00
2020-01-09 02:35:38.953000192+02:00 2020-01-09 02:35:38.953000192+02:00
```

16.6 Change datatypes

Let's then check the dataframe datatypes:

[11]:
```
df.dtypes
```

[11]:
```
user                                           object
device                                         object
timestamp                                     float64
battery_level                                   int64
battery_status                                  int64
battery_health                                  int64
battery_adaptor                                 int64
datetime           datetime64[ns, Europe/Helsinki]
dtype: object
```

We can change the datatypes with `pandas.astype()` function. Here we change **battery_health** datatype to categorical:

[12]:
```
df.astype({'battery_health': 'category'}).dtypes
```

[12]:
```
user                                           object
device                                         object
timestamp                                     float64
battery_level                                   int64
battery_status                                  int64
battery_health                               category
battery_adaptor                                 int64
datetime           datetime64[ns, Europe/Helsinki]
dtype: object
```

16.7 Transforming a column to a new value

Dataframe values can be transformed (decoded etc.) into new values by using `pandas.transform()` function.

Here we add one to the column values.

[13]:
```
df['battery_adaptor'].transform(lambda x: x + 1)
```

[13]:
```
2020-01-09 02:20:02.924999936+02:00    1
2020-01-09 02:21:30.405999872+02:00    1
2020-01-09 02:24:12.805999872+02:00    1
2020-01-09 02:35:38.561000192+02:00    1
2020-01-09 02:35:38.953000192+02:00    3
                                      ..
2019-08-09 00:30:48.073999872+03:00    2
2019-08-09 00:32:40.717999872+03:00    2
2019-08-09 00:34:23.114000128+03:00    2
2019-08-09 00:36:05.505000192+03:00    2
2019-08-09 00:37:37.671000064+03:00    2
Name: battery_adaptor, Length: 505, dtype: int64
```

16.8 Resample

Dataframe down/upsampling can be done with `pandas.resample()` function.

Here we downsample the data by hour and aggregate the mean:

[14]:
```
df['battery_level'].resample('H').agg("mean")
```

[14]:
```
2019-08-05 14:00:00+03:00    46.000000
2019-08-05 15:00:00+03:00    44.000000
2019-08-05 16:00:00+03:00    43.000000
2019-08-05 17:00:00+03:00    42.000000
2019-08-05 18:00:00+03:00    41.000000
                               ...
2020-01-09 19:00:00+02:00    86.166667
2020-01-09 20:00:00+02:00    82.000000
2020-01-09 21:00:00+02:00    78.428571
2020-01-09 22:00:00+02:00    75.000000
2020-01-09 23:00:00+02:00    72.000000
Freq: H, Name: battery_level, Length: 3779, dtype: float64
```

16.9 Groupby

For groupwise data inspection, we can use `pandas.DataFrame.groupby()` function.

Let's first load dataframe having several subjects belonging to different groups.

[15]:
```
df = niimpy.read_csv(config.SL_ACTIVITY_PATH, tz='Europe/Helsinki')
df
```

[15]:

```
                         timestamp user  activity group
0      2013-03-27 06:00:00-05:00  u00         2  none
1      2013-03-27 07:00:00-05:00  u00         1  none
2      2013-03-27 08:00:00-05:00  u00         2  none
3      2013-03-27 09:00:00-05:00  u00         3  none
4      2013-03-27 10:00:00-05:00  u00         4  none
...                          ...  ...       ...   ...
55902  2013-05-31 18:00:00-05:00  u59         5  mild
55903  2013-05-31 19:00:00-05:00  u59         5  mild
55904  2013-05-31 20:00:00-05:00  u59         4  mild
55905  2013-05-31 21:00:00-05:00  u59         5  mild
55906  2013-05-31 22:00:00-05:00  u59         1  mild

[55907 rows x 4 columns]
```

We can summarize the data by grouping the observations by **group** and **user**, and then aggregating the mean:

[16]:

```
df.groupby(['group','user']).agg("mean")
```

[16]:

```
                          activity
group              user
mild               u02    0.922348
                   u04    1.466960
                   u07    0.914457
                   u16    0.702918
                   u20    0.277946
                   u24    0.938028
                   u27    0.653724
                   u31    0.929495
                   u35    0.519455
                   u43    0.809045
                   u49    1.159767
                   u58    0.620621
                   u59    1.626263
moderate           u18    0.445323
                   u52    1.051735
moderately severe u17    0.489510
                   u23    0.412884
none               u00    1.182973
                   u03    0.176737
                   u05    0.606742
                   u09    1.095908
                   u10    0.662612
                   u14    1.005859
                   u15    0.295990
                   u30    0.933036
                   u32    1.113593
                   u36    0.936281
                   u42    0.378851
                   u44    0.292580
                   u47    0.396026
                   u51    0.828662
                   u56    0.840967
```

(continues on next page)

(continued from previous page)

```
severe             u01    1.063660
                   u19    0.571792
                   u33    0.733115
                   u34    0.454789
                   u45    0.441134
                   u53    0.389404
```

16.10 Summary statistics

There are many ways you may want to get an overview of your data.

Let's first load mobile phone screen activity data.

[17]:
```
df = niimpy.read_csv(config.MULTIUSER_AWARE_SCREEN_PATH, tz='Europe/Helsinki')
```

[18]:
```
df
```

[18]:
```
                                            user        device          time  \
2020-01-09 02:06:41.573999872+02:00  jd9INuQ5BBlW  OWd1Uau8POix  1.578528e+09
2020-01-09 02:09:29.152000+02:00     jd9INuQ5BBlW  OWd1Uau8POix  1.578529e+09
2020-01-09 02:09:32.790999808+02:00  jd9INuQ5BBlW  OWd1Uau8POix  1.578529e+09
2020-01-09 02:11:41.996000+02:00     jd9INuQ5BBlW  OWd1Uau8POix  1.578529e+09
2020-01-09 02:16:19.010999808+02:00  jd9INuQ5BBlW  OWd1Uau8POix  1.578529e+09
...                                          ...           ...           ...
2019-09-08 17:17:14.216000+03:00     iGyXetHE3S8u  Cq9vueHh3zVs  1.567952e+09
2019-09-08 17:17:31.966000128+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.567952e+09
2019-09-08 20:50:07.360000+03:00     iGyXetHE3S8u  Cq9vueHh3zVs  1.567965e+09
2019-09-08 20:50:08.139000064+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.567965e+09
2019-09-08 20:53:12.960000+03:00     iGyXetHE3S8u  Cq9vueHh3zVs  1.567965e+09

                                     screen_status  \
2020-01-09 02:06:41.573999872+02:00              0
2020-01-09 02:09:29.152000+02:00                 1
2020-01-09 02:09:32.790999808+02:00              3
2020-01-09 02:11:41.996000+02:00                 0
2020-01-09 02:16:19.010999808+02:00              1
...                                            ...
2019-09-08 17:17:14.216000+03:00                 1
2019-09-08 17:17:31.966000128+03:00              0
2019-09-08 20:50:07.360000+03:00                 3
2019-09-08 20:50:08.139000064+03:00              1
2019-09-08 20:53:12.960000+03:00                 0

                                                                datetime
2020-01-09 02:06:41.573999872+02:00 2020-01-09 02:06:41.573999872+02:00
2020-01-09 02:09:29.152000+02:00       2020-01-09 02:09:29.152000+02:00
2020-01-09 02:09:32.790999808+02:00 2020-01-09 02:09:32.790999808+02:00
2020-01-09 02:11:41.996000+02:00       2020-01-09 02:11:41.996000+02:00
2020-01-09 02:16:19.010999808+02:00 2020-01-09 02:16:19.010999808+02:00
...                                                                  ...
2019-09-08 17:17:14.216000+03:00       2019-09-08 17:17:14.216000+03:00
```

(continues on next page)

(continued from previous page)

```
2019-09-08 17:17:31.966000128+03:00 2019-09-08 17:17:31.966000128+03:00
2019-09-08 20:50:07.360000+03:00       2019-09-08 20:50:07.360000+03:00
2019-09-08 20:50:08.139000064+03:00 2019-09-08 20:50:08.139000064+03:00
2019-09-08 20:53:12.960000+03:00       2019-09-08 20:53:12.960000+03:00

[277 rows x 5 columns]
```

16.11 Hourly data

It is easy to get the amount of data (observations) in each hour

[19]:
```
hourly = df.groupby([df.index.date, df.index.hour]).size()
hourly
```

[19]:
```
2019-08-05  14    19
2019-08-08  21     6
            22    12
2019-08-09  7      6
2019-08-10  15     3
2019-08-12  22     3
2019-08-13  7     12
            8      3
            9      5
2019-08-14  23     3
2019-08-15  12     3
2019-08-17  15     6
2019-08-18  19     3
2019-08-24  8      3
            9      3
            12     3
            13     3
2019-08-25  11     5
            12     4
2019-08-26  11     6
2019-08-31  19     3
2019-09-05  23     3
2019-09-07  8      3
2019-09-08  11     3
            17     6
            20     3
2020-01-09  2     27
            10     6
            11     3
            12     3
            14    17
            15    35
            16     4
            17     8
            18     4
            20     4
```

(continues on next page)

(continued from previous page)

```
            21    19
            22     3
            23    12
dtype: int64
```

[20]:
```
# The index is the (day, hour) pairs and the
# value is the number at that time
print('%s had %d data points'%(hourly.index[0], hourly.iloc[0]))
```

```
(datetime.date(2019, 8, 5), 14) had 19 data points
```

16.12 Occurence

In niimpy, **occurence** is a way to see the completeness of data.

Occurence is defined as such: * Divides all time into hours * Divides all hours into five 12-minute intervals * Count the number of 12-minute intervals that have data. This is *occurrence* * For each hour, report *occurrence*. "5"is taken to mean that data is present somewhat regularly, while "0" means we have no data.

This isn't the perfect measure, but is reasonably effective and simple to calculate. For data which isn't continuous (like screen data we are actually using), it shows how much the sensor has been used.

Column meanings: `day` is the date, `hour` is hour of day, `occurrence` is the measure described above, `count` is total number of data points in this hour, `withdata` is which of the 12-min intervals (0-4) have data.

Note that the "uniformly present data" is not true for all data sources.

[21]:
```
occurrences = niimpy.util.occurrence(df.index)
occurrences.head()
```

[21]:
```
                            day  hour  occurrence
2019-08-05 14:00:00  2019-08-05    14           4
2019-08-08 21:00:00  2019-08-08    21           1
2019-08-08 22:00:00  2019-08-08    22           2
2019-08-09 07:00:00  2019-08-09     7           2
2019-08-10 15:00:00  2019-08-10    15           1
```

We can create a simplified presentation (pivot table) for the data by using `pandas.pivot()`function:

[22]:
```
occurrences.pivot('hour', 'day')
```

[22]:
```
     occurrence                                                           \
day  2019-08-05 2019-08-08 2019-08-09 2019-08-10 2019-08-12 2019-08-13
hour
2           NaN        NaN        NaN        NaN        NaN        NaN
7           NaN        NaN        2.0        NaN        NaN        1.0
8           NaN        NaN        NaN        NaN        NaN        1.0
9           NaN        NaN        NaN        NaN        NaN        1.0
10          NaN        NaN        NaN        NaN        NaN        NaN
11          NaN        NaN        NaN        NaN        NaN        NaN
12          NaN        NaN        NaN        NaN        NaN        NaN
13          NaN        NaN        NaN        NaN        NaN        NaN
14          4.0        NaN        NaN        NaN        NaN        NaN
```

(continues on next page)

(continued from previous page)

```
15          NaN        NaN        NaN        1.0        NaN        NaN
16          NaN        NaN        NaN        NaN        NaN        NaN
17          NaN        NaN        NaN        NaN        NaN        NaN
18          NaN        NaN        NaN        NaN        NaN        NaN
19          NaN        NaN        NaN        NaN        NaN        NaN
20          NaN        NaN        NaN        NaN        NaN        NaN
21          NaN        1.0        NaN        NaN        NaN        NaN
22          NaN        2.0        NaN        NaN        1.0        NaN
23          NaN        NaN        NaN        NaN        NaN        NaN

                                                                      \
day   2019-08-14 2019-08-15 2019-08-17 2019-08-18 2019-08-24 2019-08-25
hour
2           NaN        NaN        NaN        NaN        NaN        NaN
7           NaN        NaN        NaN        NaN        NaN        NaN
8           NaN        NaN        NaN        NaN        1.0        NaN
9           NaN        NaN        NaN        NaN        1.0        NaN
10          NaN        NaN        NaN        NaN        NaN        NaN
11          NaN        NaN        NaN        NaN        NaN        2.0
12          NaN        1.0        NaN        NaN        1.0        1.0
13          NaN        NaN        NaN        NaN        2.0        NaN
14          NaN        NaN        NaN        NaN        NaN        NaN
15          NaN        NaN        3.0        NaN        NaN        NaN
16          NaN        NaN        NaN        NaN        NaN        NaN
17          NaN        NaN        NaN        NaN        NaN        NaN
18          NaN        NaN        NaN        NaN        NaN        NaN
19          NaN        NaN        NaN        1.0        NaN        NaN
20          NaN        NaN        NaN        NaN        NaN        NaN
21          NaN        NaN        NaN        NaN        NaN        NaN
22          NaN        NaN        NaN        NaN        NaN        NaN
23          1.0        NaN        NaN        NaN        NaN        NaN


day   2019-08-26 2019-08-31 2019-09-05 2019-09-07 2019-09-08 2020-01-09
hour
2           NaN        NaN        NaN        NaN        NaN        4.0
7           NaN        NaN        NaN        NaN        NaN        NaN
8           NaN        NaN        NaN        1.0        NaN        NaN
9           NaN        NaN        NaN        NaN        NaN        NaN
10          NaN        NaN        NaN        NaN        NaN        2.0
11          1.0        NaN        NaN        NaN        2.0        1.0
12          NaN        NaN        NaN        NaN        NaN        1.0
13          NaN        NaN        NaN        NaN        NaN        NaN
14          NaN        NaN        NaN        NaN        NaN        2.0
15          NaN        NaN        NaN        NaN        NaN        3.0
16          NaN        NaN        NaN        NaN        NaN        1.0
17          NaN        NaN        NaN        NaN        1.0        2.0
18          NaN        NaN        NaN        NaN        NaN        1.0
19          NaN        1.0        NaN        NaN        NaN        NaN
20          NaN        NaN        NaN        NaN        1.0        1.0
21          NaN        NaN        NaN        NaN        NaN        4.0
22          NaN        NaN        NaN        NaN        NaN        1.0
```

(continues on next page)

(continued from previous page)

```
23          NaN        NaN        1.0        NaN        NaN        1.0
```

CHAPTER

SEVENTEEN

DEMO NOTEBOOK FOR ANALYZING CALLS AND SMS DATA

17.1 1. Introduction

In niimpy, communication data includes calls and SMS information. These data can reveal important information about people's circadian rhythm, social patterns, and activity, just to mention a few. Therefore, it is important to organize this information for further processing and analysis. To address this, niimpy includes a set of functions to clean, downsample, and extract features from communication data. The available features are:

- call_duration_total: duration of incoming and outgoing calls
- call_duration_mean: mean duration of incoming and outgoing calls
- call_duration_median: median duration of incoming and outgoing calls
- call_duration_std: standard deviation of incoming and outgoing calls
- call_count: number of calls within a time window
- call_outgoing_incoming_ratio: number of outgoing calls divided by the number of incoming calls
- sms_count: count of incoming and outgoing text messages
- extract_features_comms: wrapper to extract several features at the same time

In the following, we will analyze call logs provided by niimpy as an example to illustrate the use of niimpy's communication preprocessing functions.

17.2 2. Read data

Let's start by reading the example data provided in niimpy. These data have already been shaped in a format that meets the requirements of the data schema. Let's start by importing the needed modules. Firstly we will import the niimpy package and then we will import the module we will use (communication) and give it a short name for use convinience.

[1]:
```
import niimpy
import niimpy.preprocessing.communication as com
from niimpy import config
import pandas as pd
import warnings
warnings.filterwarnings("ignore")
```

Now let's read the example data provided in niimpy. The example data is in csv format, so we need to use the read_csv function. When reading the data, we can specify the timezone where the data was collected. This will help us handle daylight saving times easier. We can specify the timezone with the argument **tz**. The output is a dataframe. We can also check the number of rows and columns in the dataframe.

[2]:
```python
data = niimpy.read_csv(config.MULTIUSER_AWARE_CALLS_PATH, tz='Europe/Helsinki')
data.shape
```

[2]:
```
(38, 6)
```

The data was succesfully read. We can see that there are 38 datapoints with 6 columns in the dataset. However, we do not know yet what the data really looks like, so let's have a quick look:

[3]:
```python
data.head()
```

[3]:
```
                                            user         device          time  \
2020-01-09 02:08:03.896000+02:00     jd9INuQ5BBlW  3p83yASkOb_B  1.578528e+09
2020-01-09 02:49:44.969000192+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578531e+09
2020-01-09 02:22:57.168999936+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:27:21.187000064+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09
2020-01-09 02:47:16.176999936+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578531e+09

                                    call_type  call_duration  \
2020-01-09 02:08:03.896000+02:00     incoming           1079
2020-01-09 02:49:44.969000192+02:00  outgoing            174
2020-01-09 02:22:57.168999936+02:00  outgoing            890
2020-01-09 02:27:21.187000064+02:00  outgoing           1342
2020-01-09 02:47:16.176999936+02:00  incoming            645

                                                               datetime
2020-01-09 02:08:03.896000+02:00       2020-01-09 02:08:03.896000+02:00
2020-01-09 02:49:44.969000192+02:00 2020-01-09 02:49:44.969000192+02:00
2020-01-09 02:22:57.168999936+02:00 2020-01-09 02:22:57.168999936+02:00
2020-01-09 02:27:21.187000064+02:00 2020-01-09 02:27:21.187000064+02:00
2020-01-09 02:47:16.176999936+02:00 2020-01-09 02:47:16.176999936+02:00
```

[4]:
```python
data.tail()
```

[4]:
```
                                            user         device          time  \
2019-08-12 22:10:21.504000+03:00     iGyXetHE3S8u  Cq9vueHh3zVs  1.565637e+09
2019-08-12 22:27:19.923000064+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565638e+09
2019-08-13 07:01:00.960999936+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565669e+09
2019-08-13 07:28:27.657999872+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565671e+09
2019-08-13 07:21:26.436000+03:00     iGyXetHE3S8u  Cq9vueHh3zVs  1.565670e+09

                                    call_type  call_duration  \
2019-08-12 22:10:21.504000+03:00     incoming            715
2019-08-12 22:27:19.923000064+03:00  outgoing            225
2019-08-13 07:01:00.960999936+03:00  outgoing           1231
2019-08-13 07:28:27.657999872+03:00  incoming            591
2019-08-13 07:21:26.436000+03:00     outgoing            375

                                                               datetime
2019-08-12 22:10:21.504000+03:00       2019-08-12 22:10:21.504000+03:00
2019-08-12 22:27:19.923000064+03:00 2019-08-12 22:27:19.923000064+03:00
2019-08-13 07:01:00.960999936+03:00 2019-08-13 07:01:00.960999936+03:00
2019-08-13 07:28:27.657999872+03:00 2019-08-13 07:28:27.657999872+03:00
2019-08-13 07:21:26.436000+03:00       2019-08-13 07:21:26.436000+03:00
```

By exploring the head and tail of the dataframe we can form an idea of its entirety. From the data, we can see that:

- rows are observations, indexed by timestamps, i.e. each row represents a call that was received/done/missed at a given time and date
- columns are characteristics for each observation, for example, the user whose data we are analyzing
- there are at least two different users in the dataframe
- there are two main columns: `call_type` and `call_duration`. In this case, the `call_type` columns stores information about whether the call was *incoming*, *outgoing* or *missed*; and the `call_duration` contains the duration of the call in seconds

In fact, we can check the first three elements for each user

[5]:
```
data.drop_duplicates(['user','call_duration']).groupby('user').head(3)
```

[5]:
```
                                            user         device          time  \
2020-01-09 02:08:03.896000+02:00     jd9INuQ5BB1W  3p83yASkOb_B  1.578528e+09
2020-01-09 02:49:44.969000192+02:00  jd9INuQ5BB1W  3p83yASkOb_B  1.578531e+09
2020-01-09 02:22:57.168999936+02:00  jd9INuQ5BB1W  3p83yASkOb_B  1.578529e+09
2019-08-08 22:32:25.256999936+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565293e+09
2019-08-08 22:53:35.107000064+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565294e+09
2019-08-08 22:31:34.540000+03:00     iGyXetHE3S8u  Cq9vueHh3zVs  1.565293e+09

                                    call_type  call_duration  \
2020-01-09 02:08:03.896000+02:00     incoming           1079
2020-01-09 02:49:44.969000192+02:00  outgoing            174
2020-01-09 02:22:57.168999936+02:00  outgoing            890
2019-08-08 22:32:25.256999936+03:00  incoming           1217
2019-08-08 22:53:35.107000064+03:00  incoming            383
2019-08-08 22:31:34.540000+03:00     incoming           1142

                                                               datetime
2020-01-09 02:08:03.896000+02:00       2020-01-09 02:08:03.896000+02:00
2020-01-09 02:49:44.969000192+02:00 2020-01-09 02:49:44.969000192+02:00
2020-01-09 02:22:57.168999936+02:00 2020-01-09 02:22:57.168999936+02:00
2019-08-08 22:32:25.256999936+03:00 2019-08-08 22:32:25.256999936+03:00
2019-08-08 22:53:35.107000064+03:00 2019-08-08 22:53:35.107000064+03:00
2019-08-08 22:31:34.540000+03:00       2019-08-08 22:31:34.540000+03:00
```

Sometimes the data may come in a disordered manner, so just to make sure, let's order the dataframe and compare the results. We will use the columns "user" and "datetime" since we would like to order the information according to firstly, participants, and then, by time in order of happening. Luckily, in our dataframe, the index and datetime are the same.

[6]:
```
data.sort_values(by=['user', 'datetime'], inplace=True)
data.drop_duplicates(['user','call_duration']).groupby('user').head(3)
```

[6]:
```
                                            user         device          time  \
2019-08-08 22:31:34.540000+03:00     iGyXetHE3S8u  Cq9vueHh3zVs  1.565293e+09
2019-08-08 22:32:25.256999936+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565293e+09
2019-08-08 22:43:45.834000128+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.565293e+09
2020-01-09 01:55:16.996000+02:00     jd9INuQ5BB1W  3p83yASkOb_B  1.578528e+09
2020-01-09 02:06:09.790999808+02:00  jd9INuQ5BB1W  3p83yASkOb_B  1.578528e+09
2020-01-09 02:08:03.896000+02:00     jd9INuQ5BB1W  3p83yASkOb_B  1.578528e+09

                                    call_type  call_duration  \
2019-08-08 22:31:34.540000+03:00     incoming           1142
```

(continues on next page)

(continued from previous page)

```
2019-08-08 22:32:25.256999936+03:00  incoming          1217
2019-08-08 22:43:45.834000128+03:00  incoming          1170
2020-01-09 01:55:16.996000+02:00     outgoing          1256
2020-01-09 02:06:09.790999808+02:00  outgoing           271
2020-01-09 02:08:03.896000+02:00     incoming          1079

                                                            datetime
2019-08-08 22:31:34.540000+03:00       2019-08-08 22:31:34.540000+03:00
2019-08-08 22:32:25.256999936+03:00 2019-08-08 22:32:25.256999936+03:00
2019-08-08 22:43:45.834000128+03:00 2019-08-08 22:43:45.834000128+03:00
2020-01-09 01:55:16.996000+02:00       2020-01-09 01:55:16.996000+02:00
2020-01-09 02:06:09.790999808+02:00 2020-01-09 02:06:09.790999808+02:00
2020-01-09 02:08:03.896000+02:00       2020-01-09 02:08:03.896000+02:00
```

By comparing the last two dataframes, we can see that sorting the values was a good move. For example, in the unsorted dataframe, the earliest date for the user *iGyXetHE3S8u* was 2019-08-08 22:32:25; instead, for the sorted dataframe, the earliest date for the user *iGyXetHE3S8u* is 2019-08-08 22:31:34. Small differences, but still important.

17.3 * TIP! Data format requirements (or what should our data look like)

Data can take other shapes and formats. However, the `niimpy` data scheme requires it to be in a certain shape. This means the dataframe needs to have at least the following characteristics: 1. One row per call. Each row should store information about one call only 2. Each row's index should be a timestamp 3. There should be at least four columns: - index: date and time when the event happened (timestamp) - user: stores the user name whose data is analyzed. Each user should have a unique name or hash (i.e. one hash for each unique user) - call_type: stores whether the call was incoming, outgoing, or missed. The exact words *incoming*, *outgoing*, and *missed* should be used - call_duration: the duration of the call in seconds 4. Columns additional to those listed in item 3 are allowed 5. The names of the columns do not need to be exactly "user", "call_type" or "call_duration" as we can pass our own names in an argument (to be explained later).

Below is an example of a dataframe that complies with these minimum requirements

[7]:
```
example_dataschema = data[['user','call_type','call_duration']]
example_dataschema.head(3)
```

[7]:
```
                                             user call_type  call_duration
2019-08-08 22:31:34.540000+03:00     iGyXetHE3S8u  incoming           1142
2019-08-08 22:32:25.256999936+03:00  iGyXetHE3S8u  incoming           1217
2019-08-08 22:43:45.834000128+03:00  iGyXetHE3S8u  incoming           1170
```

17.4 4. Extracting features

There are two ways to extract features. We could use each function separately or we could use `niimpy`'s ready-made wrapper. Both ways will require us to specify arguments to pass to the functions/wrapper in order to customize the way the functions work. These arguments are specified in dictionaries. Let's first understand how to extract features using stand-alone functions.

17.4.1 4.1 Extract features using stand-alone functions

We can use `niimpy`'s functions to compute communication features. Each function will require two inputs: - (mandatory) dataframe that must comply with the minimum requirements (see '* TIP! Data requirements above) - (optional) an argument dictionary for stand-alone functions

4.1.1 The argument dictionary for stand-alone functions (or how we specify the way a function works)

In this dictionary, we can input two main features to customize the way a stand-alone function works: - the name of the columns to be preprocessed: Since the dataframe may have different columns, we need to specify which column has the data we would like to be preprocessed. To do so, we can simply pass the name of the column to the argument `communication_column_name`.

- the way we resample: resampling options are specified in `niimpy` as a dictionary. `niimpy`'s resampling and aggregating relies on `pandas.DataFrame.resample`, so mastering the use of this pandas function will help us greatly in `niimpy`'s preprocessing. Please familiarize yourself with the pandas resample function before continuing. Briefly, to use the `pandas.DataFrame.resample` function, we need a rule. This rule states the intervals we would like to use to resample our data (e.g., 15-seconds, 30-minutes, 1-hour). Neverthless, we can input more details into the function to specify the exact sampling we would like. For example, we could use the *close* argument if we would like to specify which side of the interval is closed, or we could use the *offset* argument if we would like to start our binning with an offset, etc. There are plenty of options to use this command, so we strongly recommend having `pandas.DataFrame.resample` documentation at hand. All features for the `pandas.DataFrame.resample` will be specified in a dictionary where keys are the arguments' names for the `pandas.DataFrame.resample`, and the dictionary's values are the values for each of these selected arguments. This dictionary will be passed as a value to the key `resample_args` in `niimpy`.

Let's see some basic examples of these dictionaries:

[8]:
```
feature_dict1:{"communication_column_name":"call_duration","resample_args":{"rule":"1D"}}
feature_dict2:{"communication_column_name":"random_name","resample_args":{"rule":"30T"}}
feature_dict3:{"communication_column_name":"other_name","resample_args":{"rule":"45T",
↪"origin":"end"}}
```

Here, we have three basic feature dictionaries.

- `feature_dict1` will be used to analyze the data stored in the column `call_duration` in our dataframe. The data will be binned in one day periods
- `feature_dict2` will be used to analyze the data stored in the column `random_name` in our dataframe. The data will be aggregated in 30-minutes bins
- `feature_dict3` will be used to analyze the data stored in the column `other_name` in our dataframe. The data will be binned in 45-minutes bins, but the binning will start from the last timestamp in the dataframe.

Default values: if no arguments are passed, `niimpy`'s default values are "call_duration" for the communication_column_name, and 30-min aggregation bins.

4.1.2 Using the functions

Now that we understand how the functions are customized, it is time we compute our first communication feature. Suppose that we are interested in extracting the total duration of outgoing calls every 20 minutes. We will need `niimpy`'s `call_duration_total` function, the data, and we will also need to create a dictionary to customize our function. Let's create the dictionary first

[9]:
```
function_features={"communication_column_name":"call_duration","resample_args":{"rule":
↪"20T"}}
```

Now let's use the function to preprocess the data.

[10]:
```
my_call_duration = com.call_duration_total(data, function_features)
```

`my_call_duration` is a multiindex dataframe, where the first level is the user, and the second level is the aggregated timestamp. Let's look at some values for one of the subjects.

[11]:
```
my_call_duration.xs("jd9INuQ5BBlW", level="user")
```

[11]:
```
                           outgoing_duration_total  incoming_duration_total  \
2020-01-09 01:40:00+02:00                   1256.0                      0.0
2020-01-09 02:00:00+02:00                   2192.0                   1079.0
2020-01-09 02:20:00+02:00                   3696.0                   4650.0
2020-01-09 02:40:00+02:00                    174.0                    645.0
2020-01-09 03:00:00+02:00                      0.0                    269.0

                           missed_duration_total
2020-01-09 01:40:00+02:00                    0.0
2020-01-09 02:00:00+02:00                    0.0
2020-01-09 02:20:00+02:00                    0.0
2020-01-09 02:40:00+02:00                    0.0
2020-01-09 03:00:00+02:00                    0.0
```

Let's remember how the original data looked like for this subject

[12]:
```
data[data["user"]=="jd9INuQ5BBlW"].head(7)
```

[12]:
```
                                             user        device          time  \
2020-01-09 01:55:16.996000+02:00     jd9INuQ5BBlW  3p83yASkOb_B  1.578528e+09
2020-01-09 02:06:09.790999808+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578528e+09
2020-01-09 02:08:03.896000+02:00     jd9INuQ5BBlW  3p83yASkOb_B  1.578528e+09
2020-01-09 02:10:06.573999872+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:11:37.648999936+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:12:31.164000+02:00     jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:21:45.877000192+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09

                                    call_type  call_duration  \
2020-01-09 01:55:16.996000+02:00     outgoing           1256
2020-01-09 02:06:09.790999808+02:00  outgoing            271
2020-01-09 02:08:03.896000+02:00     incoming           1079
2020-01-09 02:10:06.573999872+02:00    missed              0
2020-01-09 02:11:37.648999936+02:00  outgoing           1070
2020-01-09 02:12:31.164000+02:00     outgoing            851
2020-01-09 02:21:45.877000192+02:00  incoming           1489
```

(continues on next page)

(continued from previous page)

```
                                                               datetime
2020-01-09 01:55:16.996000+02:00       2020-01-09 01:55:16.996000+02:00
2020-01-09 02:06:09.790999808+02:00 2020-01-09 02:06:09.790999808+02:00
2020-01-09 02:08:03.896000+02:00       2020-01-09 02:08:03.896000+02:00
2020-01-09 02:10:06.573999872+02:00 2020-01-09 02:10:06.573999872+02:00
2020-01-09 02:11:37.648999936+02:00 2020-01-09 02:11:37.648999936+02:00
2020-01-09 02:12:31.164000+02:00       2020-01-09 02:12:31.164000+02:00
2020-01-09 02:21:45.877000192+02:00 2020-01-09 02:21:45.877000192+02:00
```

We see that the bins are indeed 20-minutes bins, however, they are adjusted to fixed, predetermined intervals, i.e. the bin does not start on the time of the first datapoint. Instead, `pandas` starts the binning at 00:00:00 of everyday and counts 20-minutes intervals from there.

If we want the binning to start from the first datapoint in our dataset, we need the origin parameter and a for loop.

[13]:
```
users = list(data['user'].unique())
results = []
for user in users:
    start_time = data[data["user"]==user].index.min()
    function_features={"communication_column_name":"call_duration","resample_args":{"rule
↪":"20T","origin":start_time}}
    results.append(com.call_duration_total(data[data["user"]==user], function_features))
my_call_duration = pd.concat(results)
```

[14]:
```
my_call_duration
```

[14]:
```
                                               outgoing_duration_total  \
user
iGyXetHE3S8u 2019-08-09 07:11:34.540000+03:00                   1322.0
             2019-08-09 07:31:34.540000+03:00                    959.0
             2019-08-09 07:51:34.540000+03:00                      0.0
             2019-08-09 08:11:34.540000+03:00                      0.0
             2019-08-09 08:31:34.540000+03:00                      0.0
...                                                                ...
             2019-08-09 06:51:34.540000+03:00                      0.0
jd9INuQ5BBlW 2020-01-09 01:55:16.996000+02:00                   3448.0
             2020-01-09 02:15:16.996000+02:00                   3078.0
             2020-01-09 02:35:16.996000+02:00                    792.0
             2020-01-09 02:55:16.996000+02:00                      0.0

                                               incoming_duration_total  \
user
iGyXetHE3S8u 2019-08-09 07:11:34.540000+03:00                        0
             2019-08-09 07:31:34.540000+03:00                     1034
             2019-08-09 07:51:34.540000+03:00                      921
             2019-08-09 08:11:34.540000+03:00                        0
             2019-08-09 08:31:34.540000+03:00                        0
...                                                                ...
             2019-08-09 06:51:34.540000+03:00                        0
jd9INuQ5BBlW 2020-01-09 01:55:16.996000+02:00                     1079
             2020-01-09 02:15:16.996000+02:00                     1897
             2020-01-09 02:35:16.996000+02:00                     3398
             2020-01-09 02:55:16.996000+02:00                      269
```

(continues on next page)

(continued from previous page)

```
                                                missed_duration_total
user
iGyXetHE3S8u 2019-08-09 07:11:34.540000+03:00                     0.0
             2019-08-09 07:31:34.540000+03:00                     0.0
             2019-08-09 07:51:34.540000+03:00                     0.0
             2019-08-09 08:11:34.540000+03:00                     0.0
             2019-08-09 08:31:34.540000+03:00                     0.0
...                                                               ...
             2019-08-09 06:51:34.540000+03:00                     0.0
jd9INuQ5BBlW 2020-01-09 01:55:16.996000+02:00                     0.0
             2020-01-09 02:15:16.996000+02:00                     0.0
             2020-01-09 02:35:16.996000+02:00                     0.0
             2020-01-09 02:55:16.996000+02:00                     0.0

[319 rows x 3 columns]
```

17.4.2 4.2 Extract features using the wrapper

We can use `niimpy`'s ready-made wrapper to extract one or several features at the same time. The wrapper will require two inputs: - (mandatory) dataframe that must comply with the minimum requirements (see '* TIP! Data requirements above) - (optional) an argument dictionary for wrapper

4.2.1 The argument dictionary for wrapper (or how we specify the way the wrapper works)

This argument dictionary will use dictionaries created for stand-alone functions. If you do not know how to create those argument dictionaries, please read the section **4.1.1 The argument dictionary for stand-alone functions (or how we specify the way a function works)** first.

The wrapper dictionary is simple. Its keys are the names of the features we want to compute. Its values are argument dictionaries created for each stand-alone function we will employ. Let's see some examples of wrapper dictionaries:

[15]:
```
wrapper_features1 = {com.call_duration_total:{"communication_column_name":"call_duration
↪","resample_args":{"rule":"1D"}},
                     com.call_count:{"communication_column_name":"call_duration",
↪"resample_args":{"rule":"1D"}}}
```

- `wrapper_features1` will be used to analyze two features, `call_duration_total` and `call_count`. For the feature call_duration_total, we will use the data stored in the column `call_duration` in our dataframe and the data will be binned in one day periods. For the feature call_count, we will use the data stored in the column `call_duration` in our dataframe and the data will be binned in one day periods.

[16]:
```
wrapper_features2 = {com.call_duration_mean:{"communication_column_name":"random_name",
↪"resample_args":{"rule":"1D"}},
                     com.call_duration_median:{"communication_column_name":"random_name",
↪"resample_args":{"rule":"5H","offset":"5min"}}}
```

- `wrapper_features2` will be used to analyze two features, `call_duration_mean` and `call_duration_median`. For the feature call_duration_mean, we will use the data stored in the column `random_name` in our dataframe and the data will be binned in one day periods. For the feature call_duration_median, we will use the data stored in the column `random_name` in our dataframe and the data will be binned in 5-hour periods with a 5-minute offset.

[17]:
```
wrapper_features3 = {com.call_duration_total:{"communication_column_name":"one_name",
↪"resample_args":{"rule":"1D","offset":"5min"}},
                     com.call_count:{"communication_column_name":"one_name","resample_
↪args":{"rule":"5H"}},
                     com.call_duration_mean:{"communication_column_name":"another_name",
↪"resample_args":{"rule":"30T","origin":"end_day"}}}
```

- `wrapper_features3` will be used to analyze three features, `call_duration_total`, `call_count`, and `call_duration_mean`. For the feature call_duration_total, we will use the data stored in the column `one_name` and the data will be binned in one day periods with a 5-min offset. For the feature call_count, we will use the data stored in the column `one_name` in our dataframe and the data will be binned in 5-hour periods. Finally, for the feature call_duration_mean, we will use the data stored in the column `another_name` in our dataframe and the data will be binned in 30-minute periods and the origin of the bins will be the ceiling midnight of the last day.

Default values: if no arguments are passed, `niimpy`'s default values are "call_duration" for the communication_column_name, and 30-min aggregation bins. Moreover, the wrapper will compute all the available functions in absence of the argument dictionary.

4.2.2 Using the wrapper

Now that we understand how the wrapper is customized, it is time we compute our first communication feature using the wrapper. Suppose that we are interested in extracting the call total duration every 20 minutes. We will need `niimpy`'s `extract_features_comms` function, the data, and we will also need to create a dictionary to customize our function. Let's create the dictionary first

[18]:
```
wrapper_features1 = {com.call_duration_total:{"communication_column_name":"call_duration
↪","resample_args":{"rule":"20T"}}}
```

Now let's use the wrapper

[19]:
```
results_wrapper = com.extract_features_comms(data, features=wrapper_features1)
results_wrapper.head(5)
```

```
computing <function call_duration_total at 0x000002521D883AC0>...
```

[19]:
```
                                         outgoing_duration_total  \
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00                   1322.0
             2019-08-09 07:20:00+03:00                    959.0
             2019-08-09 07:40:00+03:00                      0.0
             2019-08-09 08:00:00+03:00                      0.0
             2019-08-09 08:20:00+03:00                      0.0

                                         incoming_duration_total  \
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00                      0.0
             2019-08-09 07:20:00+03:00                   1034.0
             2019-08-09 07:40:00+03:00                    790.0
             2019-08-09 08:00:00+03:00                    131.0
             2019-08-09 08:20:00+03:00                      0.0

                                         missed_duration_total
user
```

(continues on next page)

(continued from previous page)

```
iGyXetHE3S8u 2019-08-09 07:00:00+03:00                   0.0
             2019-08-09 07:20:00+03:00                   0.0
             2019-08-09 07:40:00+03:00                   0.0
             2019-08-09 08:00:00+03:00                   0.0
             2019-08-09 08:20:00+03:00                   0.0
```

Our first attempt was succesful. Now, let's try something more. Let's assume we want to compute the call_duration and call_count in 20-minutes bin.

[20]:
```
wrapper_features2 = {com.call_duration_total:{"communication_column_name":"call_duration
↪","resample_args":{"rule":"20T"}},
                     com.call_count:{"communication_column_name":"call_duration",
↪"resample_args":{"rule":"20T"}}}
results_wrapper = com.extract_features_comms(data, features=wrapper_features2)
results_wrapper.head(5)
```

```
computing <function call_duration_total at 0x000002521D883AC0>...
computing <function call_count at 0x000002525E874790>...
```

[20]:
```
                                        outgoing_duration_total  \
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00                   1322.0
             2019-08-09 07:20:00+03:00                    959.0
             2019-08-09 07:40:00+03:00                      0.0
             2019-08-09 08:00:00+03:00                      0.0
             2019-08-09 08:20:00+03:00                      0.0

                                        incoming_duration_total  \
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00                      0.0
             2019-08-09 07:20:00+03:00                   1034.0
             2019-08-09 07:40:00+03:00                    790.0
             2019-08-09 08:00:00+03:00                    131.0
             2019-08-09 08:20:00+03:00                      0.0

                                        missed_duration_total  outgoing_count  \
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00                    0.0             1.0
             2019-08-09 07:20:00+03:00                    0.0             1.0
             2019-08-09 07:40:00+03:00                    0.0             0.0
             2019-08-09 08:00:00+03:00                    0.0             0.0
             2019-08-09 08:20:00+03:00                    0.0             0.0

                                        incoming_count  missed_count
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00             0.0           0.0
             2019-08-09 07:20:00+03:00             1.0           1.0
             2019-08-09 07:40:00+03:00             1.0           0.0
             2019-08-09 08:00:00+03:00             1.0           0.0
             2019-08-09 08:20:00+03:00             0.0           0.0
```

Great! Another successful attempt. We see from the results that more columns were added with the required calculations. This is how the wrapper works when all features are computed with the same bins. Now, let's see how the wrapper performs when each function has different binning requirements. Let's assume we need to compute the call_duration_mean every day, and the call_duration_median every 5 hours with an offset of 5 minutes.

[21]:
```
wrapper_features3 = {com.call_duration_mean:{"communication_column_name":"call_duration",
↪"resample_args":{"rule":"1D"}},
                     com.call_duration_median:{"communication_column_name":"call_duration
↪","resample_args":{"rule":"5H","offset":"5min"}}}
results_wrapper = com.extract_features_comms(data, features=wrapper_features3)
results_wrapper.head(5)
```

```
computing <function call_duration_mean at 0x000002525E8745E0>...
computing <function call_duration_median at 0x000002525E874670>...
```

[21]:
```
                                        outgoing_duration_mean  \
user
iGyXetHE3S8u 2019-08-09 00:00:00+03:00                  1140.5
             2019-08-10 00:00:00+03:00                  1363.0
             2019-08-11 00:00:00+03:00                     0.0
             2019-08-12 00:00:00+03:00                   209.0
             2019-08-13 00:00:00+03:00                   803.0

                                        incoming_duration_mean  \
user
iGyXetHE3S8u 2019-08-09 00:00:00+03:00              651.666667
             2019-08-10 00:00:00+03:00             1298.000000
             2019-08-11 00:00:00+03:00                0.000000
             2019-08-12 00:00:00+03:00              715.000000
             2019-08-13 00:00:00+03:00              591.000000

                                        missed_duration_mean  \
user
iGyXetHE3S8u 2019-08-09 00:00:00+03:00                   0.0
             2019-08-10 00:00:00+03:00                   0.0
             2019-08-11 00:00:00+03:00                   0.0
             2019-08-12 00:00:00+03:00                   0.0
             2019-08-13 00:00:00+03:00                   0.0

                                        outgoing_duration_median  \
user
iGyXetHE3S8u 2019-08-09 00:00:00+03:00                       NaN
             2019-08-10 00:00:00+03:00                       NaN
             2019-08-11 00:00:00+03:00                       NaN
             2019-08-12 00:00:00+03:00                       NaN
             2019-08-13 00:00:00+03:00                       NaN

                                        incoming_duration_median  \
user
iGyXetHE3S8u 2019-08-09 00:00:00+03:00                       NaN
             2019-08-10 00:00:00+03:00                       NaN
             2019-08-11 00:00:00+03:00                       NaN
             2019-08-12 00:00:00+03:00                       NaN
             2019-08-13 00:00:00+03:00                       NaN

                                        missed_duration_median
user
```

(continues on next page)

(continued from previous page)

```
iGyXetHE3S8u 2019-08-09 00:00:00+03:00                     NaN
             2019-08-10 00:00:00+03:00                     NaN
             2019-08-11 00:00:00+03:00                     NaN
             2019-08-12 00:00:00+03:00                     NaN
             2019-08-13 00:00:00+03:00                     NaN
```

[22]:
```
results_wrapper.tail(5)
```

[22]:
```
                                        outgoing_duration_mean  \
user
iGyXetHE3S8u 2019-08-12 09:05:00+03:00                     NaN
             2019-08-12 14:05:00+03:00                     NaN
             2019-08-12 19:05:00+03:00                     NaN
             2019-08-13 00:05:00+03:00                     NaN
             2019-08-13 05:05:00+03:00                     NaN

                                        incoming_duration_mean  \
user
iGyXetHE3S8u 2019-08-12 09:05:00+03:00                     NaN
             2019-08-12 14:05:00+03:00                     NaN
             2019-08-12 19:05:00+03:00                     NaN
             2019-08-13 00:05:00+03:00                     NaN
             2019-08-13 05:05:00+03:00                     NaN

                                        missed_duration_mean  \
user
iGyXetHE3S8u 2019-08-12 09:05:00+03:00                   NaN
             2019-08-12 14:05:00+03:00                   NaN
             2019-08-12 19:05:00+03:00                   NaN
             2019-08-13 00:05:00+03:00                   NaN
             2019-08-13 05:05:00+03:00                   NaN

                                        outgoing_duration_median  \
user
iGyXetHE3S8u 2019-08-12 09:05:00+03:00                       0.0
             2019-08-12 14:05:00+03:00                       0.0
             2019-08-12 19:05:00+03:00                       0.0
             2019-08-13 00:05:00+03:00                       0.0
             2019-08-13 05:05:00+03:00                       0.0

                                        incoming_duration_median  \
user
iGyXetHE3S8u 2019-08-12 09:05:00+03:00                       0.0
             2019-08-12 14:05:00+03:00                       0.0
             2019-08-12 19:05:00+03:00                     715.0
             2019-08-13 00:05:00+03:00                       0.0
             2019-08-13 05:05:00+03:00                     591.0

                                        missed_duration_median
user
iGyXetHE3S8u 2019-08-12 09:05:00+03:00                     0.0
             2019-08-12 14:05:00+03:00                     0.0
```

(continues on next page)

(continued from previous page)

```
             2019-08-12 19:05:00+03:00                     0.0
             2019-08-13 00:05:00+03:00                     0.0
             2019-08-13 05:05:00+03:00                     0.0
```

The output is once again a dataframe. In this case, two aggregations are shown. The first one is the daily aggregation computed for the `call_duration_mean` feature (head). The second one is the 5-hour aggregation period with 5-min offset for the `call_duration_median` (tail). We must note that because the `call_duration_median`feature is not required to be aggregated daily, the daily aggregation timestamps have a NaN value. Similarly, because the `call_duration_mean`is not required to be aggregated in 5-hour windows, its values are NaN for all subjects.

4.2.3 Wrapper and its default option

The default option will compute all features in 30-minute aggregation windows. To use the `extract_features_comms` function with its default options, simply call the function.

[23]:
```
default = com.extract_features_comms(data, features=None)
```

```
computing <function call_duration_total at 0x000002521D883AC0>...
computing <function call_duration_mean at 0x000002525E8745E0>...
computing <function call_duration_median at 0x000002525E874670>...
computing <function call_duration_std at 0x000002525E874700>...
computing <function call_count at 0x000002525E874790>...
computing <function call_outgoing_incoming_ratio at 0x000002525E874820>...
```

The function prints the computed features so you can track its process. Now let's have a look at the outputs

[24]:
```
default.head()
```

[24]:
```
                                        outgoing_duration_total  \
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00                   1322.0
             2019-08-09 07:30:00+03:00                    959.0
             2019-08-09 08:00:00+03:00                      0.0
             2019-08-09 08:30:00+03:00                      0.0
             2019-08-09 09:00:00+03:00                      0.0

                                        incoming_duration_total  \
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00                      0.0
             2019-08-09 07:30:00+03:00                   1824.0
             2019-08-09 08:00:00+03:00                    131.0
             2019-08-09 08:30:00+03:00                      0.0
             2019-08-09 09:00:00+03:00                      0.0

                                        missed_duration_total  \
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00                    0.0
             2019-08-09 07:30:00+03:00                    0.0
             2019-08-09 08:00:00+03:00                    0.0
             2019-08-09 08:30:00+03:00                    0.0
             2019-08-09 09:00:00+03:00                    0.0

                                        outgoing_duration_mean  \
```

(continues on next page)

(continued from previous page)

```
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00                  1322.0
             2019-08-09 07:30:00+03:00                   959.0
             2019-08-09 08:00:00+03:00                     0.0
             2019-08-09 08:30:00+03:00                     0.0
             2019-08-09 09:00:00+03:00                     0.0

                                        incoming_duration_mean  \
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00                     0.0
             2019-08-09 07:30:00+03:00                   912.0
             2019-08-09 08:00:00+03:00                   131.0
             2019-08-09 08:30:00+03:00                     0.0
             2019-08-09 09:00:00+03:00                     0.0

                                        missed_duration_mean  \
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00                   0.0
             2019-08-09 07:30:00+03:00                   0.0
             2019-08-09 08:00:00+03:00                   0.0
             2019-08-09 08:30:00+03:00                   0.0
             2019-08-09 09:00:00+03:00                   0.0

                                        outgoing_duration_median  \
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00                    1322.0
             2019-08-09 07:30:00+03:00                     959.0
             2019-08-09 08:00:00+03:00                       0.0
             2019-08-09 08:30:00+03:00                       0.0
             2019-08-09 09:00:00+03:00                       0.0

                                        incoming_duration_median  \
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00                       0.0
             2019-08-09 07:30:00+03:00                     912.0
             2019-08-09 08:00:00+03:00                     131.0
             2019-08-09 08:30:00+03:00                       0.0
             2019-08-09 09:00:00+03:00                       0.0

                                        missed_duration_median  \
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00                     0.0
             2019-08-09 07:30:00+03:00                     0.0
             2019-08-09 08:00:00+03:00                     0.0
             2019-08-09 08:30:00+03:00                     0.0
             2019-08-09 09:00:00+03:00                     0.0

                                        outgoing_duration_std  \
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00                    0.0
             2019-08-09 07:30:00+03:00                    0.0
             2019-08-09 08:00:00+03:00                    0.0
```

(continues on next page)

(continued from previous page)

```
            2019-08-09 08:30:00+03:00                   0.0
            2019-08-09 09:00:00+03:00                   0.0

                                      incoming_duration_std  \
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00              0.000000
             2019-08-09 07:30:00+03:00            172.534055
             2019-08-09 08:00:00+03:00              0.000000
             2019-08-09 08:30:00+03:00              0.000000
             2019-08-09 09:00:00+03:00              0.000000

                                      missed_duration_std  outgoing_count  \
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00                0.0             1.0
             2019-08-09 07:30:00+03:00                0.0             1.0
             2019-08-09 08:00:00+03:00                0.0             0.0
             2019-08-09 08:30:00+03:00                0.0             0.0
             2019-08-09 09:00:00+03:00                0.0             0.0

                                      incoming_count  missed_count  \
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00           0.0           0.0
             2019-08-09 07:30:00+03:00           2.0           1.0
             2019-08-09 08:00:00+03:00           1.0           0.0
             2019-08-09 08:30:00+03:00           0.0           0.0
             2019-08-09 09:00:00+03:00           0.0           0.0

                                      outgoing_incoming_ratio
user
iGyXetHE3S8u 2019-08-09 07:00:00+03:00                      inf
             2019-08-09 07:30:00+03:00                      0.5
             2019-08-09 08:00:00+03:00                      0.0
             2019-08-09 08:30:00+03:00                      0.0
             2019-08-09 09:00:00+03:00                      0.0
```

17.4.3 4.3 SMS computations

`niimpy` includes one function to count the outgoing and incoming SMS. This function is not automatically called by `extract_features_comms`, but it can be used as a standalone. Let's see a quick example where we will upload the SMS data and preprocess it.

[25]:
```
data = niimpy.read_csv(config.MULTIUSER_AWARE_MESSAGES_PATH, tz='Europe/Helsinki')
data.head()
```

[25]:
```
                                          user          device          time  \
2020-01-09 02:34:46.644999936+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09
2020-01-09 02:34:58.803000064+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09
2020-01-09 02:35:37.611000064+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09
2020-01-09 02:55:40.640000+02:00     jd9INuQ5BBlW  3p83yASkOb_B  1.578531e+09
2020-01-09 02:55:50.914000128+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578531e+09
```

(continues on next page)

(continued from previous page)

```
                                       message_type  \
2020-01-09 02:34:46.644999936+02:00     incoming
2020-01-09 02:34:58.803000064+02:00     outgoing
2020-01-09 02:35:37.611000064+02:00     outgoing
2020-01-09 02:55:40.640000+02:00        outgoing
2020-01-09 02:55:50.914000128+02:00     incoming

                                                               datetime
2020-01-09 02:34:46.644999936+02:00 2020-01-09 02:34:46.644999936+02:00
2020-01-09 02:34:58.803000064+02:00 2020-01-09 02:34:58.803000064+02:00
2020-01-09 02:35:37.611000064+02:00 2020-01-09 02:35:37.611000064+02:00
2020-01-09 02:55:40.640000+02:00       2020-01-09 02:55:40.640000+02:00
2020-01-09 02:55:50.914000128+02:00 2020-01-09 02:55:50.914000128+02:00
```

[26]:
```
sms = com.sms_count(data, feature_functions={})
sms.head()
```

[26]:
```
                                        outgoing_count  incoming_count
user
iGyXetHE3S8u 2019-08-13 08:30:00+03:00               1             1.0
             2019-08-13 09:00:00+03:00               0             0.0
             2019-08-13 09:30:00+03:00               2             1.0
             2019-08-13 10:00:00+03:00               0             0.0
             2019-08-13 10:30:00+03:00               0             0.0
```

Similar to the calls functions, we need to define the `feature_functions` dictionary. Likewise, if we leave it empty, then all data is aggregated in 30-minutes bins. We see that the function also differentiates between the incoming and outgoing messages. Let's quickly summarize the data requirements for SMS

17.5 * TIP! Data format requirements for SMS (special case)

Data can take other shapes and formats. However, the `niimpy` data scheme requires it to be in a certain shape. This means the dataframe needs to have at least the following characteristics: 1. One row per call. Each row should store information about one call only 2. Each row's index should be a timestamp 3. There should be at least four columns: - index: date and time when the event happened (timestamp) - user: stores the user name whose data is analyzed. Each user should have a unique name or hash (i.e. one hash for each unique user) - message_type: determines if the message was sent (outgoing) or received (incoming) 4. Columns additional to those listed in item 3 are allowed 5. The names of the columns do not need to be exactly "user", "message_type"

17.6 5. Implementing own features

If none of the provided functions suits well, We can implement our own customized features easily. To do so, we need to define a function that accepts a dataframe and returns a dataframe. The returned object should be indexed by user and timestamps (multiindex). To make the feature readily available in the default options, we need add the *call* prefix to the new function (e.g. `call_my-new-feature`). Let's assume we need a new function that counts all calls, independent of their direction (outgoing, incoming, etc.). Let's first define the function

[27]:
```
def call_count_all(df,feature_functions=None):
    if not "communication_column_name" in feature_functions:
```

(continues on next page)

(continued from previous page)

```
        col_name = "call_duration"
    else:
        col_name = feature_functions["communication_column_name"]
    if not "resample_args" in feature_functions.keys():
        feature_functions["resample_args"] = {"rule":"30T"}

    if len(df)>0:
        result = df.groupby("user")[col_name].resample(**feature_functions["resample_args
↪"]).count()
        result.rename("call_count_all", inplace=True)
        result.to_frame()
    return result
```

Then, we can call our new function in the stand-alone way or using the `extract_features_comms` function. Because the stand-alone way is the common way to call functions in python, we will not show it. Instead, we will show how to integrate this new function to the wrapper. Let's read again the data and assume we want the default behavior of the wrapper.

[28]:
```
data = niimpy.read_csv(config.MULTIUSER_AWARE_CALLS_PATH, tz='Europe/Helsinki')
customized_features = com.extract_features_comms(data, features={call_count_all: {}})
```

```
computing <function call_count_all at 0x000002525EBBD900>...
```

[29]:
```
customized_features.head()
```

[29]:
```
                                            call_count_all
user
iGyXetHE3S8u 2019-08-08 22:30:00+03:00               5
             2019-08-08 23:00:00+03:00               0
             2019-08-08 23:30:00+03:00               0
             2019-08-09 00:00:00+03:00               0
             2019-08-09 00:30:00+03:00               0
```

[ ]:

CHAPTER

EIGHTEEN

DEMO NOTEBOOK FOR ANALYZING SCREEN ON/OFF DATA

18.1 Introduction

Screen data refers to the information about the status of the screen as reported by Android. These data can reveal important information about people's circadian rhythm, social patterns, and activity. Screen data is an event data, this means that it cannot be sampled at a regular frequency. We just have information about the events that occured. However, some factors may interfere with the correct detection of all events (e.g. when the phone's battery is depleated). Therefore, to correctly process screen data, we need to take into account other information like the battery status of the phone. This may complicate the preprocessing. To address this, `niimpy` includes a set of functions to clean, downsample, and extract features from screen data while taking into account factors like the battery level. The functions allow us to extract the following features:

- `screen_off`: reports when the screen has been turned off
- `screen_count`: number of times the screen has turned on, off, or has been in use
- `screen_duration`: duration in seconds of the screen on, off, and in use statuses
- `screen_duration_min`: minimum duration in seconds of the screen on, off, and in use statuses
- `screen_duration_max`: maximum duration in seconds of the screen on, off, and in use statuses
- `screen_duration_median`: median duration in seconds of the screen on, off, and in use statuses
- `screen_duration_mean`: mean duration in seconds of the screen on, off, and in use statuses
- `screen_duration_std`: standard deviation of the duration in seconds of the screen on, off, and in use statuses
- `screen_first_unlock`: reports the first time when the phone was unlocked every day
- `extract_features_screen`: wrapper-like function to extract several features at the same time

In addition, the screen module has three internal functions that help classify the events and calculate their status duration.

In the following, we will analyze screen data provided by `niimpy` as an example to illustrate the use of screen data.

18.2 2. Read data

Let's start by reading the example data provided in `niimpy`. These data have already been shaped in a format that meets the requirements of the data schema. Let's start by importing the needed modules. Firstly we will import the `niimpy` package and then we will import the module we will use (screen) and give it a short name for use convenience.

[1]:
```
import niimpy
from niimpy import config
import niimpy.preprocessing.screen as s
```

(continues on next page)

(continued from previous page)

```
import pandas as pd
import warnings
warnings.filterwarnings("ignore")
```

Now let's read the example data provided in `niimpy`. The example data is in `csv` format, so we need to use the `read_csv` function. When reading the data, we can specify the timezone where the data was collected. This will help us handle daylight saving times easier. We can specify the timezone with the argument **tz**. The output is a dataframe. We can also check the number of rows and columns in the dataframe.

[2]:
```
data = niimpy.read_csv(config.MULTIUSER_AWARE_SCREEN_PATH, tz='Europe/Helsinki')
data.shape
```

[2]:
```
(277, 5)
```

The data was succesfully read. We can see that there are 277 datapoints with 5 columns in the dataset. However, we do not know yet what the data really looks like, so let's have a quick look:

[3]:
```
data.head()
```

[3]:
```
                                            user       device          time  \
2020-01-09 02:06:41.573999872+02:00  jd9INuQ5BBlW  OWd1Uau8POix  1.578528e+09
2020-01-09 02:09:29.152000+02:00     jd9INuQ5BBlW  OWd1Uau8POix  1.578529e+09
2020-01-09 02:09:32.790999808+02:00  jd9INuQ5BBlW  OWd1Uau8POix  1.578529e+09
2020-01-09 02:11:41.996000+02:00     jd9INuQ5BBlW  OWd1Uau8POix  1.578529e+09
2020-01-09 02:16:19.010999808+02:00  jd9INuQ5BBlW  OWd1Uau8POix  1.578529e+09

                                     screen_status  \
2020-01-09 02:06:41.573999872+02:00              0
2020-01-09 02:09:29.152000+02:00                 1
2020-01-09 02:09:32.790999808+02:00              3
2020-01-09 02:11:41.996000+02:00                 0
2020-01-09 02:16:19.010999808+02:00              1

                                                               datetime
2020-01-09 02:06:41.573999872+02:00 2020-01-09 02:06:41.573999872+02:00
2020-01-09 02:09:29.152000+02:00       2020-01-09 02:09:29.152000+02:00
2020-01-09 02:09:32.790999808+02:00 2020-01-09 02:09:32.790999808+02:00
2020-01-09 02:11:41.996000+02:00       2020-01-09 02:11:41.996000+02:00
2020-01-09 02:16:19.010999808+02:00 2020-01-09 02:16:19.010999808+02:00
```

[4]:
```
data.tail()
```

[4]:
```
                                            user       device          time  \
2019-09-08 17:17:14.216000+03:00     iGyXetHE3S8u  Cq9vueHh3zVs  1.567952e+09
2019-09-08 17:17:31.966000128+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.567952e+09
2019-09-08 20:50:07.360000+03:00     iGyXetHE3S8u  Cq9vueHh3zVs  1.567965e+09
2019-09-08 20:50:08.139000064+03:00  iGyXetHE3S8u  Cq9vueHh3zVs  1.567965e+09
2019-09-08 20:53:12.960000+03:00     iGyXetHE3S8u  Cq9vueHh3zVs  1.567965e+09

                                     screen_status  \
2019-09-08 17:17:14.216000+03:00                 1
2019-09-08 17:17:31.966000128+03:00              0
2019-09-08 20:50:07.360000+03:00                 3
2019-09-08 20:50:08.139000064+03:00              1
```

(continues on next page)

(continued from previous page)

```
2019-09-08 20:53:12.960000+03:00                0

                                                              datetime
2019-09-08 17:17:14.216000+03:00       2019-09-08 17:17:14.216000+03:00
2019-09-08 17:17:31.966000128+03:00 2019-09-08 17:17:31.966000128+03:00
2019-09-08 20:50:07.360000+03:00       2019-09-08 20:50:07.360000+03:00
2019-09-08 20:50:08.139000064+03:00 2019-09-08 20:50:08.139000064+03:00
2019-09-08 20:53:12.960000+03:00       2019-09-08 20:53:12.960000+03:00
```

By exploring the head and tail of the dataframe we can form an idea of its entirety. From the data, we can see that:

- rows are observations, indexed by timestamps, i.e. each row represents a screen event at a given time and date
- columns are characteristics for each observation, for example, the user whose data we are analyzing
- there are at least two different users in the dataframe
- the main column is `screen_status`. This screen status is coded in numbers as: 0=off, 1=on, 2=locked, 3=unlocked.

18.3 * TIP! Data format requirements (or what should our data look like)

Data can take other shapes and formats. However, the `niimpy` data scheme requires it to be in a certain shape. This means the dataframe needs to have at least the following characteristics: 1. One row per screen status. Each row should store information about one screen status only 2. Each row's index should be a timestamp 3. There should be at least three columns: - index: date and time when the event happened (timestamp) - user: stores the user name whose data is analyzed. Each user should have a unique name or hash (i.e. one hash for each unique user) - screen_status: stores the screen status (0,1,2, or 3) as defined by Android. 4. Columns additional to those listed in item 3 are allowed 5. The names of the columns do not need to be exactly "screen_status" as we can pass our own names in an argument (to be explained later).

Below is an example of a dataframe that complies with these minimum requirements

[5]:
```
example_dataschema = data[['user','screen_status']]
example_dataschema.head(3)
```

[5]:
```
                                            user  screen_status
2020-01-09 02:06:41.573999872+02:00  jd9INuQ5BBlW              0
2020-01-09 02:09:29.152000+02:00     jd9INuQ5BBlW              1
2020-01-09 02:09:32.790999808+02:00  jd9INuQ5BBlW              3
```

18.3.1 A few words on missing data

Missing data for screen is difficult to detect. Firstly, this sensor is triggered by events and not sampled at a fixed frequency. Secondly, different phones, OS, and settings change how the screen is turned on/off; for example, one phone may go from OFF to ON to UNLOCKED, while another phone may go from OFF to UNLOCKED directly. Thirdly, events not related to the screen may affect its behavior, e.g. battery running out. Neverthless, there are some events transitions that are impossible to have, like a status to itself (e.g. two consecutive 0s). These *imposible* statuses helps us determine the missing data.

18.3.2 A few words on the classification of the events

We can know the status of the screen at a certain timepoint. However, we need a bit more to know the duration and the meaning of it. Consequently, we need to look at the numbers of two consecutive events and classify the transitions (going from one state to another consecutively) as: - from 3 to 0,1,2: the phone was in use - from 1 to 0,1,3: the phone was on - from 0 to 1,2,3: the phone was off

Other transitions are irrelevant.

18.3.3 A few words on the role of the battery

As mentioned before, battery statuses can affect the screen behavior. In particular, when the battery is depleated and the phone is shut down automatically, the screen sensor does not cast any events, so even when the screen is technically OFF because the phone does not have any battery left, we will not see that 0 in the screen status column. Thus, it is important to take into account the battery information when analyzing screen data. `niimpy`'s screen module is adapted to take into account the battery data. Since we do have some battery data, we will load it.

[6]:
```
bat_data = niimpy.read_csv(config.MULTIUSER_AWARE_BATTERY_PATH, tz='Europe/Helsinki')
bat_data.head()
```

[6]:
```
                                             user         device          time  \
2020-01-09 02:20:02.924999936+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:21:30.405999872+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:24:12.805999872+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578529e+09
2020-01-09 02:35:38.561000192+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09
2020-01-09 02:35:38.953000192+02:00  jd9INuQ5BBlW  3p83yASkOb_B  1.578530e+09

                                     battery_level  battery_status  \
2020-01-09 02:20:02.924999936+02:00             74               3
2020-01-09 02:21:30.405999872+02:00             73               3
2020-01-09 02:24:12.805999872+02:00             72               3
2020-01-09 02:35:38.561000192+02:00             72               2
2020-01-09 02:35:38.953000192+02:00             72               2

                                     battery_health  battery_adaptor  \
2020-01-09 02:20:02.924999936+02:00               2                0
2020-01-09 02:21:30.405999872+02:00               2                0
2020-01-09 02:24:12.805999872+02:00               2                0
2020-01-09 02:35:38.561000192+02:00               2                0
2020-01-09 02:35:38.953000192+02:00               2                2

                                                                datetime
2020-01-09 02:20:02.924999936+02:00 2020-01-09 02:20:02.924999936+02:00
2020-01-09 02:21:30.405999872+02:00 2020-01-09 02:21:30.405999872+02:00
2020-01-09 02:24:12.805999872+02:00 2020-01-09 02:24:12.805999872+02:00
2020-01-09 02:35:38.561000192+02:00 2020-01-09 02:35:38.561000192+02:00
2020-01-09 02:35:38.953000192+02:00 2020-01-09 02:35:38.953000192+02:00
```

In this case, we are interested in the battery_status information. This is standard information provided by Android. However, if the dataframe has this information in a column with a different name, we can use the argument `battery_column_name` similarly to the use of `screen_column_name` (more info about this topic below).

18.4 4. Extracting features

There are two ways to extract features. We could use each function separately or we could use niimpy's ready-made wrapper. Both ways will require us to specify arguments to pass to the functions/wrapper in order to customize the way the functions work. These arguments are specified in dictionaries. Let's first understand how to extract features using stand-alone functions.

We can use niimpy's functions to compute communication features. Each function will require two inputs: - (mandatory) dataframe that must comply with the minimum requirements (see '* TIP! Data requirements above) - (optional) an argument dictionary for stand-alone functions

18.4.1 4.1.1 The argument dictionary for stand-alone functions (or how we specify the way a function works)

In this dictionary, we can input two main features to customize the way a stand-alone function works: - the name of the columns to be preprocessed: Since the dataframe may have different columns, we need to specify which column has the data we would like to be preprocessed. To do so, we can simply pass the name of the column to the argument `screen_column_name`.

- the way we resample: resampling options are specified in `niimpy` as a dictionary. `niimpy`'s resampling and aggregating relies on `pandas.DataFrame.resample`, so mastering the use of this pandas function will help us greatly in `niimpy`'s preprocessing. Please familiarize yourself with the pandas resample function before continuing. Briefly, to use the `pandas.DataFrame.resample` function, we need a rule. This rule states the intervals we would like to use to resample our data (e.g., 15-seconds, 30-minutes, 1-hour). Neverthless, we can input more details into the function to specify the exact sampling we would like. For example, we could use the *close* argument if we would like to specify which side of the interval is closed, or we could use the *offset* argument if we would like to start our binning with an offset, etc. There are plenty of options to use this command, so we strongly recommend having `pandas.DataFrame.resample` documentation at hand. All features for the `pandas.DataFrame.resample` will be specified in a dictionary where keys are the arguments' names for the `pandas.DataFrame.resample`, and the dictionary's values are the values for each of these selected arguments. This dictionary will be passed as a value to the key `resample_args` in `niimpy`.

Let's see some basic examples of these dictionaries:

[7]:
```
feature_dict1:{"screen_column_name":"screen_status","resample_args":{"rule":"1D"}}
feature_dict2:{"screen_column_name":"random_name","resample_args":{"rule":"30T"}}
feature_dict3:{"screen_column_name":"other_name","resample_args":{"rule":"45T","origin":
↪"end"}}
```

Here, we have three basic feature dictionaries.

- `feature_dict1` will be used to analyze the data stored in the column `screen_status` in our dataframe. The data will be binned in one day periods
- `feature_dict2` will be used to analyze the data stored in the column `random_name` in our dataframe. The data will be aggregated in 30-minutes bins
- `feature_dict3` will be used to analyze the data stored in the column `other_name` in our dataframe. The data will be binned in 45-minutes bins, but the binning will start from the last timestamp in the dataframe.

Default values: if no arguments are passed, `niimpy`'s default values are "screen_status" for the screen_column_name, and 30-min aggregation bins.

18.4.2 4.1.2 Using the functions

Now that we understand how the functions are customized, it is time we compute our first communication feature. Suppose that we are interested in extracting the total duration of outgoing calls every 20 minutes. We will need `niimpy`'s `screen_count` function, the data, and we will also need to create a dictionary to customize our function. Let's create the dictionary first

[8]:
```
function_features={"screen_column_name":"screen_status","resample_args":{"rule":"20T"}}
```

Now let's use the function to preprocess the data.

[9]:
```
my_screen_count = s.screen_count(data, bat_data, function_features)
```

`my_screen_count` is a multiindex dataframe, where the first level is the user, and the second level is the aggregated timestamp. Let's look at some values for one of the subjects.

[10]:
```
my_screen_count.xs("jd9INuQ5BBlW", level="user")
```

[10]:
```
                           screen_on_count  screen_off_count  screen_use_count
2020-01-09 02:00:00+02:00                2                 2                 2
2020-01-09 02:20:00+02:00                3                 4                 2
2020-01-09 02:40:00+02:00                2                 2                 1
2020-01-09 03:00:00+02:00                0                 0                 0
2020-01-09 03:20:00+02:00                0                 0                 0
...                                    ...               ...               ...
2020-01-09 21:40:00+02:00                1                 1                 0
2020-01-09 22:00:00+02:00                1                 1                 0
2020-01-09 22:20:00+02:00                0                 0                 0
2020-01-09 22:40:00+02:00                0                 0                 0
2020-01-09 23:00:00+02:00                4                 3                 0

[64 rows x 3 columns]
```

Let's remember how the original data looked like for this subject

[11]:
```
data[data["user"]=="jd9INuQ5BBlW"].head(7)
```

[11]:
```
                                             user        device          time  \
2020-01-09 02:06:41.573999872+02:00  jd9INuQ5BBlW  OWd1Uau8POix  1.578528e+09
2020-01-09 02:09:29.152000+02:00     jd9INuQ5BBlW  OWd1Uau8POix  1.578529e+09
2020-01-09 02:09:32.790999808+02:00  jd9INuQ5BBlW  OWd1Uau8POix  1.578529e+09
2020-01-09 02:11:41.996000+02:00     jd9INuQ5BBlW  OWd1Uau8POix  1.578529e+09
2020-01-09 02:16:19.010999808+02:00  jd9INuQ5BBlW  OWd1Uau8POix  1.578529e+09
2020-01-09 02:16:29.648999936+02:00  jd9INuQ5BBlW  OWd1Uau8POix  1.578529e+09
2020-01-09 02:16:29.657999872+02:00  jd9INuQ5BBlW  OWd1Uau8POix  1.578529e+09

                                     screen_status  \
2020-01-09 02:06:41.573999872+02:00              0
2020-01-09 02:09:29.152000+02:00                 1
2020-01-09 02:09:32.790999808+02:00              3
2020-01-09 02:11:41.996000+02:00                 0
2020-01-09 02:16:19.010999808+02:00              1
2020-01-09 02:16:29.648999936+02:00              0
2020-01-09 02:16:29.657999872+02:00              2
```

(continues on next page)

(continued from previous page)

```
                                                         datetime
2020-01-09 02:06:41.573999872+02:00 2020-01-09 02:06:41.573999872+02:00
2020-01-09 02:09:29.152000+02:00       2020-01-09 02:09:29.152000+02:00
2020-01-09 02:09:32.790999808+02:00 2020-01-09 02:09:32.790999808+02:00
2020-01-09 02:11:41.996000+02:00       2020-01-09 02:11:41.996000+02:00
2020-01-09 02:16:19.010999808+02:00 2020-01-09 02:16:19.010999808+02:00
2020-01-09 02:16:29.648999936+02:00 2020-01-09 02:16:29.648999936+02:00
2020-01-09 02:16:29.657999872+02:00 2020-01-09 02:16:29.657999872+02:00
```

We see that the bins are indeed 20-minutes bins, however, they are adjusted to fixed, predetermined intervals, i.e. the bin does not start on the time of the first datapoint. Instead, `pandas` starts the binning at 00:00:00 of everyday and counts 20-minutes intervals from there.

If we want the binning to start from the first datapoint in our dataset, we need the origin parameter and a for loop.

[12]:
```
users = list(data['user'].unique())
results = []
for user in users:
    start_time = data[data["user"]==user].index.min()
    function_features={"screen_column_name":"screen_status","resample_args":{"rule":"20T
↪","origin":start_time}}
    results.append(s.screen_count(data[data["user"]==user],bat_data[bat_data["user
↪"]==user], function_features))
my_screen_count = pd.concat(results)
```

[13]:
```
my_screen_count
```

[13]:
```
                                                  screen_on_count  \
user
jd9INuQ5BBlW 2020-01-09 02:06:41.573999872+02:00                4
             2020-01-09 02:26:41.573999872+02:00                2
             2020-01-09 02:46:41.573999872+02:00                2
             2020-01-09 03:06:41.573999872+02:00                0
             2020-01-09 03:26:41.573999872+02:00                0
...                                                           ...
iGyXetHE3S8u 2019-09-08 19:22:41.009999872+03:00                0
             2019-09-08 19:42:41.009999872+03:00                0
             2019-09-08 20:02:41.009999872+03:00                0
             2019-09-08 20:22:41.009999872+03:00                0
             2019-09-08 20:42:41.009999872+03:00                0

                                                  screen_off_count  \
user
jd9INuQ5BBlW 2020-01-09 02:06:41.573999872+02:00                 3
             2020-01-09 02:26:41.573999872+02:00                 3
             2020-01-09 02:46:41.573999872+02:00                 2
             2020-01-09 03:06:41.573999872+02:00                 0
             2020-01-09 03:26:41.573999872+02:00                 0
...                                                            ...
iGyXetHE3S8u 2019-09-08 19:22:41.009999872+03:00                 0
             2019-09-08 19:42:41.009999872+03:00                 0
             2019-09-08 20:02:41.009999872+03:00                 0
             2019-09-08 20:22:41.009999872+03:00                 0
```

(continues on next page)

(continued from previous page)

```
             2019-09-08 20:42:41.009999872+03:00                 0

                                                    screen_use_count
user
jd9INuQ5BBlW 2020-01-09 02:06:41.573999872+02:00                 3
             2020-01-09 02:26:41.573999872+02:00                 1
             2020-01-09 02:46:41.573999872+02:00                 1
             2020-01-09 03:06:41.573999872+02:00                 0
             2020-01-09 03:26:41.573999872+02:00                 0
...                                                            ...
iGyXetHE3S8u 2019-09-08 19:22:41.009999872+03:00                 0
             2019-09-08 19:42:41.009999872+03:00                 0
             2019-09-08 20:02:41.009999872+03:00                 0
             2019-09-08 20:22:41.009999872+03:00                 0
             2019-09-08 20:42:41.009999872+03:00                 1

[2533 rows x 3 columns]
```

The functions can also be called in absence of a feature_functions dictionary. In this case, the binning will be automatically set to 30-minutes.

[14]:
```
my_screen_count = s.screen_count(data, bat_data, {})
my_screen_count.head()
```

[14]:
```
                                          screen_on_count  screen_off_count  \
user
iGyXetHE3S8u 2019-08-05 14:00:00+03:00                4                 4
             2019-08-05 14:30:00+03:00                2                 2
             2019-08-05 15:00:00+03:00                0                 0
             2019-08-05 15:30:00+03:00                0                 0
             2019-08-05 16:00:00+03:00                0                 0

                                          screen_use_count
user
iGyXetHE3S8u 2019-08-05 14:00:00+03:00                 4
             2019-08-05 14:30:00+03:00                 2
             2019-08-05 15:00:00+03:00                 0
             2019-08-05 15:30:00+03:00                 0
             2019-08-05 16:00:00+03:00                 0
```

In case we do not have battery data, the functions can also be called without it. In this case, simply input an empty dataframe in the second position of the function. For example,

[15]:
```
empty_bat = pd.DataFrame()
no_bat = s.screen_count(data, empty_bat, function_features) #no battery information
no_bat.head()
```

[15]:
```
                                                    screen_on_count  \
user
iGyXetHE3S8u 2019-08-05 14:02:41.009999872+03:00                3
             2019-08-05 14:22:41.009999872+03:00                2
             2019-08-05 14:42:41.009999872+03:00                1
             2019-08-05 15:02:41.009999872+03:00                0
```

(continues on next page)

(continued from previous page)

```
             2019-08-05 15:22:41.009999872+03:00                0

                                                 screen_off_count  \
user
iGyXetHE3S8u 2019-08-05 14:02:41.009999872+03:00                 3
             2019-08-05 14:22:41.009999872+03:00                 2
             2019-08-05 14:42:41.009999872+03:00                 1
             2019-08-05 15:02:41.009999872+03:00                 0
             2019-08-05 15:22:41.009999872+03:00                 0

                                                 screen_use_count
user
iGyXetHE3S8u 2019-08-05 14:02:41.009999872+03:00                 3
             2019-08-05 14:22:41.009999872+03:00                 2
             2019-08-05 14:42:41.009999872+03:00                 1
             2019-08-05 15:02:41.009999872+03:00                 0
             2019-08-05 15:22:41.009999872+03:00                 0
```

4.2 Extract features using the wrapper

We can use `niimpy`'s ready-made wrapper to extract one or several features at the same time. The wrapper will require two inputs: - (mandatory) dataframe that must comply with the minimum requirements (see '* TIP! Data requirements above) - (optional) an argument dictionary for wrapper

18.4.3 4.2.1 The argument dictionary for wrapper (or how we specify the way the wrapper works)

This argument dictionary will use dictionaries created for stand-alone functions. If you do not know how to create those argument dictionaries, please read the section **4.1.1 The argument dictionary for stand-alone functions (or how we specify the way a function works)** first.

The wrapper dictionary is simple. Its keys are the names of the features we want to compute. Its values are argument dictionaries created for each stand-alone function we will employ. Let's see some examples of wrapper dictionaries:

[16]:
```
wrapper_features1 = {s.screen_count:{"screen_column_name":"screen_status","resample_args
↪":{"rule":"1D"}},
                     s.screen_duration_min:{"screen_column_name":"screen_status",
↪"resample_args":{"rule":"1D"}}}
```

- `wrapper_features1` will be used to analyze two features, `screen_count` and `screen_duration_min`. For the feature screen_count, we will use the data stored in the column `screen_status` in our dataframe and the data will be binned in one day periods. For the feature screen_duration_min, we will use the data stored in the column `screen_status` in our dataframe and the data will be binned in one day periods.

[17]:
```
wrapper_features2 = {s.screen_count:{"screen_column_name":"screen_status", "battery_
↪column_name":"battery_status", "resample_args":{"rule":"1D"}},
                     s.screen_duration:{"screen_column_name":"random_name","resample_args
↪":{"rule":"5H","offset":"5min"}}}
```

- `wrapper_features2` will be used to analyze two features, `screen_status` and `screen_duration`. For the feature screen_status, we will use the data stored in the column `screen_status` in our dataframe and the data will be binned in one day periods. In addition, we will use battery data stored in a column called "battery_status". For the feature screen_duration, we will use the data stored in the column `random_name` in our dataframe and the data will be binned in 5-hour periods with a 5-minute offset.

[18]:
```
wrapper_features3 = {s.screen_count:{"screen_column_name":"one_name","resample_args":{
↪"rule":"1D","offset":"5min"}},
                     s.screen_duration:{"screen_column_name":"one_name", "battery_column
↪":"some_column","resample_args":{}},
                     s.screen_duration_min:{"screen_column_name":"another_name",
↪"resample_args":{"rule":"30T","origin":"end_day"}}}
```

- `wrapper_features3` will be used to analyze three features, `screen_count`, `screen_duration`, and `screen_duration_min`. For the feature screen_count, we will use the data stored in the column `one_name` and the data will be binned in one day periods with a 5-min offset. For the feature screen_duration, we will use the data stored in the column `one_name` in our dataframe and the data will be binned using the default settings, i.e. 30-min bins. In addition, we will use data from the battery sensor, which will be passed in a column called "some_column". Finally, for the feature screen_duration_min, we will use the data stored in the column `another_name` in our dataframe and the data will be binned in 30-minute periods and the origin of the bins will be the ceiling midnight of the last day.

Default values: if no arguments are passed, `niimpy`'s default values are "screen_status" for the screen_column_name, and 30-min aggregation bins. Moreover, the wrapper will compute all the available functions in absence of the argument dictionary.

18.4.4 4.2.2 Using the wrapper

Now that we understand how the wrapper is customized, it is time we compute our first communication feature using the wrapper. Suppose that we are interested in extracting the call total duration every 50 minutes. We will need `niimpy`'s `extract_features_comms` function, the data, and we will also need to create a dictionary to customize our function. Let's create the dictionary first

[19]:
```
wrapper_features1 = {s.screen_duration:{"screen_column_name":"screen_status","resample_
↪args":{"rule":"50T"}}}
```

Now, let's use the wrapper

[20]:
```
results_wrapper = s.extract_features_screen(data, bat_data, features=wrapper_features1)
results_wrapper.head(5)
```

```
computing <function screen_duration at 0x000001EDD30E8160>...
```

[20]:
```
                                          screen_on_durationtotal  \
user
iGyXetHE3S8u 2019-08-05 13:20:00+03:00                     78.193
             2019-08-05 14:10:00+03:00                    198.189
             2019-08-05 15:00:00+03:00                      0.000
             2019-08-05 15:50:00+03:00                      0.000
             2019-08-05 16:40:00+03:00                      0.000

                                          screen_off_durationtotal  \
user
iGyXetHE3S8u 2019-08-05 13:20:00+03:00                     546.422
             2019-08-05 14:10:00+03:00                  286720.506
             2019-08-05 15:00:00+03:00                       0.000
```

(continues on next page)

(continued from previous page)

```
             2019-08-05 15:50:00+03:00                     0.000
             2019-08-05 16:40:00+03:00                     0.000

                                        screen_use_durationtotal
user
iGyXetHE3S8u 2019-08-05 13:20:00+03:00                     0.139
             2019-08-05 14:10:00+03:00                     1.050
             2019-08-05 15:00:00+03:00                     0.000
             2019-08-05 15:50:00+03:00                     0.000
             2019-08-05 16:40:00+03:00                     0.000
```

Our first attempt was succesful. Now, let's try something more. Let's assume we want to compute the screen_duration and screen_count in 50-minutes bin.

[21]:
```
wrapper_features2 = {s.screen_duration:{"screen_column_name":"screen_status","resample_
↪args":{"rule":"50T"}},
                     s.screen_count:{"screen_column_name":"screen_status","resample_args
↪":{"rule":"50T"}}}
results_wrapper = s.extract_features_screen(data, bat_data, features=wrapper_features2)
results_wrapper.head(5)
```

```
computing <function screen_duration at 0x000001EDD30E8160>...
computing <function screen_count at 0x000001EDD30E80D0>...
```

[21]:
```
                                        screen_on_durationtotal  \
user
iGyXetHE3S8u 2019-08-05 13:20:00+03:00                   78.193
             2019-08-05 14:10:00+03:00                  198.189
             2019-08-05 15:00:00+03:00                    0.000
             2019-08-05 15:50:00+03:00                    0.000
             2019-08-05 16:40:00+03:00                    0.000

                                        screen_off_durationtotal  \
user
iGyXetHE3S8u 2019-08-05 13:20:00+03:00                   546.422
             2019-08-05 14:10:00+03:00                286720.506
             2019-08-05 15:00:00+03:00                     0.000
             2019-08-05 15:50:00+03:00                     0.000
             2019-08-05 16:40:00+03:00                     0.000

                                        screen_use_durationtotal  \
user
iGyXetHE3S8u 2019-08-05 13:20:00+03:00                     0.139
             2019-08-05 14:10:00+03:00                     1.050
             2019-08-05 15:00:00+03:00                     0.000
             2019-08-05 15:50:00+03:00                     0.000
             2019-08-05 16:40:00+03:00                     0.000

                                        screen_on_count  screen_off_count  \
user
iGyXetHE3S8u 2019-08-05 13:20:00+03:00                1                 1
             2019-08-05 14:10:00+03:00                5                 5
             2019-08-05 15:00:00+03:00                0                 0
```

(continues on next page)

(continued from previous page)

```
              2019-08-05 15:50:00+03:00                0                 0
              2019-08-05 16:40:00+03:00                0                 0

                                        screen_use_count
user
iGyXetHE3S8u 2019-08-05 13:20:00+03:00                 1
             2019-08-05 14:10:00+03:00                 5
             2019-08-05 15:00:00+03:00                 0
             2019-08-05 15:50:00+03:00                 0
             2019-08-05 16:40:00+03:00                 0
```

Great! Another successful attempt. We see from the results that more columns were added with the required calculations. This is how the wrapper works when all features are computed with the same bins. Now, let's see how the wrapper performs when each function has different binning requirements. Let's assume we need to compute the screen_duration every day, and the screen_count every 5 hours with an offset of 5 minutes.

[22]:
```
wrapper_features3 = {s.screen_duration:{"screen_column_name":"screen_status","resample_
↪args":{"rule":"1D"}},
                     s.screen_count:{"screen_column_name":"screen_status","resample_args
↪":{"rule":"5H","offset":"5min"}}}
results_wrapper = s.extract_features_screen(data, bat_data, features=wrapper_features3)
results_wrapper.head(5)
```

```
computing <function screen_duration at 0x000001EDD30E8160>...
computing <function screen_count at 0x000001EDD30E80D0>...
```

[22]:
```
                                        screen_on_durationtotal  \
user
iGyXetHE3S8u 2019-08-05 00:00:00+03:00                 276.382
             2019-08-06 00:00:00+03:00                   0.000
             2019-08-07 00:00:00+03:00                   0.000
             2019-08-08 00:00:00+03:00                  98.228
             2019-08-09 00:00:00+03:00                   8.136

                                        screen_off_durationtotal  \
user
iGyXetHE3S8u 2019-08-05 00:00:00+03:00             287266.927999
             2019-08-06 00:00:00+03:00                  0.000000
             2019-08-07 00:00:00+03:00                  0.000000
             2019-08-08 00:00:00+03:00              34238.356000
             2019-08-09 00:00:00+03:00             114869.103000

                                        screen_use_durationtotal  \
user
iGyXetHE3S8u 2019-08-05 00:00:00+03:00                     1.189
             2019-08-06 00:00:00+03:00                     0.000
             2019-08-07 00:00:00+03:00                     0.000
             2019-08-08 00:00:00+03:00                     2.866
             2019-08-09 00:00:00+03:00                     0.516

                                        screen_on_count  screen_off_count  \
user
iGyXetHE3S8u 2019-08-05 00:00:00+03:00              NaN               NaN
```

(continues on next page)

(continued from previous page)

```
                2019-08-06 00:00:00+03:00                NaN              NaN
                2019-08-07 00:00:00+03:00                NaN              NaN
                2019-08-08 00:00:00+03:00                NaN              NaN
                2019-08-09 00:00:00+03:00                NaN              NaN

                                          screen_use_count
user
iGyXetHE3S8u 2019-08-05 00:00:00+03:00                 NaN
             2019-08-06 00:00:00+03:00                 NaN
             2019-08-07 00:00:00+03:00                 NaN
             2019-08-08 00:00:00+03:00                 NaN
             2019-08-09 00:00:00+03:00                 NaN
```

[23]:
```
results_wrapper.tail(5)
```

[23]:
```
                                          screen_on_durationtotal  \
user
jd9INuQ5BBlW 2020-01-09 00:05:00+02:00                      NaN
             2020-01-09 05:05:00+02:00                      NaN
             2020-01-09 10:05:00+02:00                      NaN
             2020-01-09 15:05:00+02:00                      NaN
             2020-01-09 20:05:00+02:00                      NaN

                                          screen_off_durationtotal  \
user
jd9INuQ5BBlW 2020-01-09 00:05:00+02:00                       NaN
             2020-01-09 05:05:00+02:00                       NaN
             2020-01-09 10:05:00+02:00                       NaN
             2020-01-09 15:05:00+02:00                       NaN
             2020-01-09 20:05:00+02:00                       NaN

                                          screen_use_durationtotal  \
user
jd9INuQ5BBlW 2020-01-09 00:05:00+02:00                       NaN
             2020-01-09 05:05:00+02:00                       NaN
             2020-01-09 10:05:00+02:00                       NaN
             2020-01-09 15:05:00+02:00                       NaN
             2020-01-09 20:05:00+02:00                       NaN

                                          screen_on_count  screen_off_count  \
user
jd9INuQ5BBlW 2020-01-09 00:05:00+02:00                7.0               8.0
             2020-01-09 05:05:00+02:00                0.0               0.0
             2020-01-09 10:05:00+02:00                9.0               9.0
             2020-01-09 15:05:00+02:00               17.0              17.0
             2020-01-09 20:05:00+02:00               12.0              11.0

                                          screen_use_count
user
jd9INuQ5BBlW 2020-01-09 00:05:00+02:00                 5.0
             2020-01-09 05:05:00+02:00                 0.0
             2020-01-09 10:05:00+02:00                 3.0
```

(continues on next page)

(continued from previous page)

```
                2020-01-09 15:05:00+02:00              7.0
                2020-01-09 20:05:00+02:00              3.0
```

The output is once again a dataframe. In this case, two aggregations are shown. The first one is the daily aggregation computed for the `screen_duration` feature (head). The second one is the 5-hour aggregation period with 5-min offset for the `screen_count` (tail). We must note that because the `screen_count`feature is not required to be aggregated daily, the daily aggregation timestamps have a NaN value. Similarly, because the `screen_duration`is not required to be aggregated in 5-hour windows, its values are NaN for all subjects.

18.4.5 4.2.3 Wrapper and its default option

The default option will compute all features in 30-minute aggregation windows. To use the `extract_features_comms` function with its default options, simply call the function.

[24]:
```
default = s.extract_features_screen(data, bat_data)
```

```
computing <function screen_off at 0x000001EDD30E8040>...
computing <function screen_count at 0x000001EDD30E80D0>...
computing <function screen_duration at 0x000001EDD30E8160>...
computing <function screen_duration_min at 0x000001EDD30E81F0>...
computing <function screen_duration_max at 0x000001EDD30E8280>...
computing <function screen_duration_mean at 0x000001EDD30E8310>...
computing <function screen_duration_median at 0x000001EDD30E83A0>...
computing <function screen_duration_std at 0x000001EDD30E8430>...
computing <function screen_first_unlock at 0x000001EDD30E84C0>...
```

The function prints the computed features so you can track its process. Now let's have a look at the outputs

[25]:
```
default.tail(10)
```

[25]:
```
                                        screen_off  screen_on_count  \
user
jd9INuQ5BBlW 2020-01-09 19:30:00+02:00         NaN              0.0
             2020-01-09 20:00:00+02:00         NaN              0.0
             2020-01-09 20:30:00+02:00         NaN              1.0
             2020-01-09 21:00:00+02:00         NaN              2.0
             2020-01-09 21:30:00+02:00         NaN              4.0
             2020-01-09 22:00:00+02:00         NaN              1.0
             2020-01-09 22:30:00+02:00         NaN              0.0
             2020-01-09 23:00:00+02:00         NaN              4.0
iGyXetHE3S8u 2019-08-05 00:00:00+03:00         NaN              NaN
jd9INuQ5BBlW 2020-01-09 00:00:00+02:00         NaN              NaN

                                        screen_off_count  screen_use_count  \
user
jd9INuQ5BBlW 2020-01-09 19:30:00+02:00               0.0               0.0
             2020-01-09 20:00:00+02:00               0.0               0.0
             2020-01-09 20:30:00+02:00               1.0               1.0
             2020-01-09 21:00:00+02:00               1.0               1.0
             2020-01-09 21:30:00+02:00               5.0               1.0
             2020-01-09 22:00:00+02:00               1.0               0.0
             2020-01-09 22:30:00+02:00               0.0               0.0
```

(continues on next page)

(continued from previous page)

```
             2020-01-09 23:00:00+02:00             3.0             0.0
iGyXetHE3S8u 2019-08-05 00:00:00+03:00             NaN             NaN
jd9INuQ5BBlW 2020-01-09 00:00:00+02:00             NaN             NaN

                                        screen_on_durationtotal  \
user
jd9INuQ5BBlW 2020-01-09 19:30:00+02:00                    0.000
             2020-01-09 20:00:00+02:00                    0.000
             2020-01-09 20:30:00+02:00                    8.253
             2020-01-09 21:00:00+02:00                   11.158
             2020-01-09 21:30:00+02:00                  376.930
             2020-01-09 22:00:00+02:00                  154.643
             2020-01-09 22:30:00+02:00                    0.000
             2020-01-09 23:00:00+02:00                    6.931
iGyXetHE3S8u 2019-08-05 00:00:00+03:00                      NaN
jd9INuQ5BBlW 2020-01-09 00:00:00+02:00                      NaN

                                        screen_off_durationtotal  \
user
jd9INuQ5BBlW 2020-01-09 19:30:00+02:00                  0.000000
             2020-01-09 20:00:00+02:00                  0.000000
             2020-01-09 20:30:00+02:00                  0.005000
             2020-01-09 21:00:00+02:00                  0.010000
             2020-01-09 21:30:00+02:00                 46.027999
             2020-01-09 22:00:00+02:00                  0.011000
             2020-01-09 22:30:00+02:00                  0.000000
             2020-01-09 23:00:00+02:00                  0.025000
iGyXetHE3S8u 2019-08-05 00:00:00+03:00                       NaN
jd9INuQ5BBlW 2020-01-09 00:00:00+02:00                       NaN

                                        screen_use_durationtotal  \
user
jd9INuQ5BBlW 2020-01-09 19:30:00+02:00                     0.000
             2020-01-09 20:00:00+02:00                     0.000
             2020-01-09 20:30:00+02:00                    28.930
             2020-01-09 21:00:00+02:00                    39.087
             2020-01-09 21:30:00+02:00                   101.062
             2020-01-09 22:00:00+02:00                       NaN
             2020-01-09 22:30:00+02:00                       NaN
             2020-01-09 23:00:00+02:00                       NaN
iGyXetHE3S8u 2019-08-05 00:00:00+03:00                       NaN
jd9INuQ5BBlW 2020-01-09 00:00:00+02:00                       NaN

                                        screen_on_durationminimum  \
user
jd9INuQ5BBlW 2020-01-09 19:30:00+02:00                        NaN
             2020-01-09 20:00:00+02:00                        NaN
             2020-01-09 20:30:00+02:00                      8.253
             2020-01-09 21:00:00+02:00                      5.234
             2020-01-09 21:30:00+02:00                     33.834
             2020-01-09 22:00:00+02:00                    154.643
             2020-01-09 22:30:00+02:00                        NaN
```

(continues on next page)

(continued from previous page)

```
              2020-01-09 23:00:00+02:00                        2.079
iGyXetHE3S8u 2019-08-05 00:00:00+03:00                          NaN
jd9INuQ5BBlW 2020-01-09 00:00:00+02:00                          NaN

                                        screen_off_durationminimum  \
user
jd9INuQ5BBlW 2020-01-09 19:30:00+02:00                          NaN
             2020-01-09 20:00:00+02:00                          NaN
             2020-01-09 20:30:00+02:00                        0.005
             2020-01-09 21:00:00+02:00                        0.010
             2020-01-09 21:30:00+02:00                        0.006
             2020-01-09 22:00:00+02:00                        0.011
             2020-01-09 22:30:00+02:00                          NaN
             2020-01-09 23:00:00+02:00                        0.008
iGyXetHE3S8u 2019-08-05 00:00:00+03:00                          NaN
jd9INuQ5BBlW 2020-01-09 00:00:00+02:00                          NaN

                                        screen_use_durationminimum  ...  \
user                                                                ...
jd9INuQ5BBlW 2020-01-09 19:30:00+02:00                          NaN  ...
             2020-01-09 20:00:00+02:00                          NaN  ...
             2020-01-09 20:30:00+02:00                       28.930  ...
             2020-01-09 21:00:00+02:00                       39.087  ...
             2020-01-09 21:30:00+02:00                      101.062  ...
             2020-01-09 22:00:00+02:00                          NaN  ...
             2020-01-09 22:30:00+02:00                          NaN  ...
             2020-01-09 23:00:00+02:00                          NaN  ...
iGyXetHE3S8u 2019-08-05 00:00:00+03:00                          NaN  ...
jd9INuQ5BBlW 2020-01-09 00:00:00+02:00                          NaN  ...

                                        screen_on_durationmean  \
user
jd9INuQ5BBlW 2020-01-09 19:30:00+02:00                      NaN
             2020-01-09 20:00:00+02:00                      NaN
             2020-01-09 20:30:00+02:00                 8.253000
             2020-01-09 21:00:00+02:00                 5.579000
             2020-01-09 21:30:00+02:00                94.232500
             2020-01-09 22:00:00+02:00               154.643000
             2020-01-09 22:30:00+02:00                      NaN
             2020-01-09 23:00:00+02:00                 2.310333
iGyXetHE3S8u 2019-08-05 00:00:00+03:00                      NaN
jd9INuQ5BBlW 2020-01-09 00:00:00+02:00                      NaN

                                        screen_off_durationmean  \
user
jd9INuQ5BBlW 2020-01-09 19:30:00+02:00                       NaN
             2020-01-09 20:00:00+02:00                       NaN
             2020-01-09 20:30:00+02:00                  0.005000
             2020-01-09 21:00:00+02:00                  0.010000
             2020-01-09 21:30:00+02:00                  9.205600
             2020-01-09 22:00:00+02:00                  0.011000
             2020-01-09 22:30:00+02:00                       NaN
```

(continues on next page)

(continued from previous page)

```
             2020-01-09 23:00:00+02:00                  0.008333
iGyXetHE3S8u 2019-08-05 00:00:00+03:00                       NaN
jd9INuQ5BBlW 2020-01-09 00:00:00+02:00                       NaN

                                        screen_use_durationmean  \
user
jd9INuQ5BBlW 2020-01-09 19:30:00+02:00                       NaN
             2020-01-09 20:00:00+02:00                       NaN
             2020-01-09 20:30:00+02:00                    28.930
             2020-01-09 21:00:00+02:00                    39.087
             2020-01-09 21:30:00+02:00                   101.062
             2020-01-09 22:00:00+02:00                       NaN
             2020-01-09 22:30:00+02:00                       NaN
             2020-01-09 23:00:00+02:00                       NaN
iGyXetHE3S8u 2019-08-05 00:00:00+03:00                       NaN
jd9INuQ5BBlW 2020-01-09 00:00:00+02:00                       NaN

                                        screen_on_durationmedian  \
user
jd9INuQ5BBlW 2020-01-09 19:30:00+02:00                        NaN
             2020-01-09 20:00:00+02:00                        NaN
             2020-01-09 20:30:00+02:00                     8.2530
             2020-01-09 21:00:00+02:00                     5.5790
             2020-01-09 21:30:00+02:00                    73.2835
             2020-01-09 22:00:00+02:00                   154.6430
             2020-01-09 22:30:00+02:00                        NaN
             2020-01-09 23:00:00+02:00                     2.2620
iGyXetHE3S8u 2019-08-05 00:00:00+03:00                        NaN
jd9INuQ5BBlW 2020-01-09 00:00:00+02:00                        NaN

                                        screen_off_durationmedian  \
user
jd9INuQ5BBlW 2020-01-09 19:30:00+02:00                         NaN
             2020-01-09 20:00:00+02:00                         NaN
             2020-01-09 20:30:00+02:00                       0.005
             2020-01-09 21:00:00+02:00                       0.010
             2020-01-09 21:30:00+02:00                       0.012
             2020-01-09 22:00:00+02:00                       0.011
             2020-01-09 22:30:00+02:00                         NaN
             2020-01-09 23:00:00+02:00                       0.008
iGyXetHE3S8u 2019-08-05 00:00:00+03:00                         NaN
jd9INuQ5BBlW 2020-01-09 00:00:00+02:00                         NaN

                                        screen_use_durationmedian  \
user
jd9INuQ5BBlW 2020-01-09 19:30:00+02:00                         NaN
             2020-01-09 20:00:00+02:00                         NaN
             2020-01-09 20:30:00+02:00                      28.930
             2020-01-09 21:00:00+02:00                      39.087
             2020-01-09 21:30:00+02:00                     101.062
             2020-01-09 22:00:00+02:00                         NaN
             2020-01-09 22:30:00+02:00                         NaN
```

(continues on next page)

(continued from previous page)

```
             2020-01-09 23:00:00+02:00                          NaN
iGyXetHE3S8u 2019-08-05 00:00:00+03:00                          NaN
jd9INuQ5BBlW 2020-01-09 00:00:00+02:00                          NaN

                                        screen_on_durationstd  \
user
jd9INuQ5BBlW 2020-01-09 19:30:00+02:00                      NaN
             2020-01-09 20:00:00+02:00                      NaN
             2020-01-09 20:30:00+02:00                      NaN
             2020-01-09 21:00:00+02:00                 0.487904
             2020-01-09 21:30:00+02:00                71.990324
             2020-01-09 22:00:00+02:00                      NaN
             2020-01-09 22:30:00+02:00                      NaN
             2020-01-09 23:00:00+02:00                 0.258906
iGyXetHE3S8u 2019-08-05 00:00:00+03:00                      NaN
jd9INuQ5BBlW 2020-01-09 00:00:00+02:00                      NaN

                                        screen_off_durationstd  \
user
jd9INuQ5BBlW 2020-01-09 19:30:00+02:00                       NaN
             2020-01-09 20:00:00+02:00                       NaN
             2020-01-09 20:30:00+02:00                       NaN
             2020-01-09 21:00:00+02:00                       NaN
             2020-01-09 21:30:00+02:00                 20.561987
             2020-01-09 22:00:00+02:00                       NaN
             2020-01-09 22:30:00+02:00                       NaN
             2020-01-09 23:00:00+02:00                  0.000577
iGyXetHE3S8u 2019-08-05 00:00:00+03:00                       NaN
jd9INuQ5BBlW 2020-01-09 00:00:00+02:00                       NaN

                                        screen_use_durationstd  \
user
jd9INuQ5BBlW 2020-01-09 19:30:00+02:00                       NaN
             2020-01-09 20:00:00+02:00                       NaN
             2020-01-09 20:30:00+02:00                       NaN
             2020-01-09 21:00:00+02:00                       NaN
             2020-01-09 21:30:00+02:00                       NaN
             2020-01-09 22:00:00+02:00                       NaN
             2020-01-09 22:30:00+02:00                       NaN
             2020-01-09 23:00:00+02:00                       NaN
iGyXetHE3S8u 2019-08-05 00:00:00+03:00                       NaN
jd9INuQ5BBlW 2020-01-09 00:00:00+02:00                       NaN

                                                     datetime
user
jd9INuQ5BBlW 2020-01-09 19:30:00+02:00                    NaT
             2020-01-09 20:00:00+02:00                    NaT
             2020-01-09 20:30:00+02:00                    NaT
             2020-01-09 21:00:00+02:00                    NaT
             2020-01-09 21:30:00+02:00                    NaT
             2020-01-09 22:00:00+02:00                    NaT
             2020-01-09 22:30:00+02:00                    NaT
```

(continues on next page)

(continued from previous page)

```
              2020-01-09 23:00:00+02:00                                    NaT
iGyXetHE3S8u 2019-08-05 00:00:00+03:00 2019-08-05 14:03:42.322000128+03:00
jd9INuQ5BBlW 2020-01-09 00:00:00+02:00 2020-01-09 02:16:19.010999808+02:00

[10 rows x 23 columns]
```

18.5 Implementing own features

If none of the provided functions suits well, We can implement our own customized features easily. To do so, we need to define a function that accepts a dataframe and returns a dataframe. The returned object should be indexed by user and timestamps (multiindex). To make the feature readily available in the default options, we need add the *screen* prefix to the new function (e.g. `screen_my-new-feature`). Let's assume we need a new function that detects the last time the screen is unlocked. Let's first define the function

[26]:
```
def screen_last_unlock(df, bat, feature_functions=None):
    if not "screen_column_name" in feature_functions:
        col_name = "screen_status"
    else:
        col_name = feature_functions["screen_column_name"]
    if not "resample_args" in feature_functions.keys():
        feature_functions["resample_args"] = {"rule":"30T"}

    df2 = s.util_screen(df, bat, feature_functions)
    df2 = s.event_classification_screen(df2, feature_functions)

    result = df2[df2.on==1].groupby("user").resample(rule='1D').max()
    result = result[["datetime"]]

    return result
```

Then, we can call our new function in the stand-alone way or using the `extract_features_screen` function. Because the stand-alone way is the common way to call functions in python, we will not show it. Instead, we will show how to integrate this new function to the wrapper. Let's read again the data and assume we want the default behavior of the wrapper.

[27]:
```
customized_features = s.extract_features_screen(data, bat_data, features={screen_last_
↪unlock: {}})
```

```
computing <function screen_last_unlock at 0x000001EDD3534F70>...
```

[28]:
```
customized_features.head()
```

[28]:
```
                                                                  datetime
user
iGyXetHE3S8u 2019-08-05 00:00:00+03:00 2019-08-05 14:49:45.596999936+03:00
             2019-08-06 00:00:00+03:00                                 NaT
             2019-08-07 00:00:00+03:00                                 NaT
             2019-08-08 00:00:00+03:00 2019-08-08 22:44:13.834000128+03:00
             2019-08-09 00:00:00+03:00    2019-08-09 07:50:33.224000+03:00
```

```
[ ]:
```

CHAPTER

NINETEEN

SURVEYS

Surveys consist of columns * `id` for the question identifier * `answer` for the answer of the question * `q` which is the text of the question presented to the user (optional) * As usual, the DataFrame index is the timestamp of the answer. It is the convention that all responses in a one single survey instance have the same timestamp, and this is used to link surveys together.

The raw on-disk format is "long", that is, one row per answer, which is "tidy data". This provides the most flexible format, but often you need to do other transformations.

19.1 Load data

[1]:
```
# Artificial example survey data
import niimpy
from niimpy import config
import niimpy.preprocessing.survey as survey
from niimpy.preprocessing.survey import *
import warnings
warnings.filterwarnings("ignore")
```

[2]:
```
df = niimpy.read_csv(config.SURVEY_PATH, tz='Europe/Helsinki')
df.head()
```

[2]:
```
   user  age  gender Little interest or pleasure in doing things.  \
0     1   20    Male                                 several-days
1     2   32    Male                      more-than-half-the-days
2     3   15    Male                      more-than-half-the-days
3     4   35  Female                                   not-at-all
4     5   23    Male                      more-than-half-the-days

  Feeling down; depressed or hopeless. Feeling nervous; anxious or on edge.  \
0              more-than-half-the-days                           not-at-all
1              more-than-half-the-days                           not-at-all
2                           not-at-all                         several-days
3                    nearly-every-day                           not-at-all
4                           not-at-all              more-than-half-the-days

  Not being able to stop or control worrying.  \
0                            nearly-every-day
1                                several-days
2                                  not-at-all
```

(continues on next page)

(continued from previous page)

```
3                                 several-days
4                                 several-days

  In the last month; how often have you felt that you were unable to control the␣
↪important things in your life?  \
0                                        almost-never
1                                               never
2                                               never
3                                          very-often
4                                        almost-never

  In the last month; how often have you felt confident about your ability to handle your␣
↪personal problems?  \
0                                           sometimes
1                                               never
2                                          very-often
3                                        fairly-often
4                                          very-often

  In the last month; how often have you felt that things were going your way?  \
0                                        fairly-often
1                                          very-often
2                                          very-often
3                                          very-often
4                                        almost-never

  In the last month; how often have you been able to control irritations in your life?  \
0                                               never
1                                           sometimes
2                                        fairly-often
3                                               never
4                                           sometimes

  In the last month; how often have you felt that you were on top of things?  \
0                                           sometimes
1                                               never
2                                               never
3                                           sometimes
4                                           sometimes

  In the last month; how often have you been angered because of things that were outside␣
↪of your control?  \
0                                          very-often
1                                        fairly-often
2                                               never
3                                               never
4                                          very-often

  In the last month; how often have you felt difficulties were piling up so high that␣
↪you could not overcome them?
0                                        fairly-often
1                                               never
```

(continues on next page)

(continued from previous page)

```
2                                        almost-never
3                                        fairly-often
4                                               never
```

19.2 Preprocessing

The dataframe's columns are raw questions from a survey. Some questions belong to a specific category, so we will annotate them with ids. The id is constructed from a prefix (the questionnaire category: GAD, PHQ, PSQI etc.), followed by the question number (1,2,3). Similarly, we will also the answers to meaningful numerical values.

Note: It's important that the dataframe follows the below schema before passing into niimpy.

[3]:
```
# Convert column name to id, based on provided mappers from niimpy
col_id = {**PHQ2_MAP, **PSQI_MAP, **PSS10_MAP, **PANAS_MAP, **GAD2_MAP}
selected_cols = [col for col in df.columns if col in col_id.keys()]

# Convert from wide to long format
transformed_df = pd.melt(df, id_vars=['user', 'age', 'gender'], value_vars=selected_cols,
↪ var_name='question', value_name='raw_answer')

# Assign questions to codes
transformed_df['id'] = transformed_df['question'].replace(col_id)
transformed_df.head()
```

[3]:
```
   user  age  gender                                      question  \
0     1   20    Male  Little interest or pleasure in doing things.
1     2   32    Male  Little interest or pleasure in doing things.
2     3   15    Male  Little interest or pleasure in doing things.
3     4   35  Female  Little interest or pleasure in doing things.
4     5   23    Male  Little interest or pleasure in doing things.

               raw_answer      id
0            several-days  PHQ2_1
1  more-than-half-the-days  PHQ2_1
2  more-than-half-the-days  PHQ2_1
3              not-at-all  PHQ2_1
4  more-than-half-the-days  PHQ2_1
```

Moreover, `niimpy` can convert the raw answers to numerical values for further analysis. For this, we need a mapping `{raw_answer: numerical_answer}`, which `niimpy` provides within the `survey` module that you can easily adjust to your own needs.

Based on the question's id, `niimpy` maps the raw answers to their numerical presentation.

[4]:
```
# Transform raw answers to numerical values
transformed_df['answer'] = survey.survey_convert_to_numerical_answer(transformed_df,␣
↪answer_col = 'raw_answer',
                                                                    question_id = 'id',␣
↪id_map=ID_MAP_PREFIX, use_prefix=True)
transformed_df.head()
```

[4]:
```
   user  age  gender                                        question  \
0     1   20    Male  Little interest or pleasure in doing things.
1     2   32    Male  Little interest or pleasure in doing things.
2     3   15    Male  Little interest or pleasure in doing things.
3     4   35  Female  Little interest or pleasure in doing things.
4     5   23    Male  Little interest or pleasure in doing things.

                raw_answer      id answer
0             several-days  PHQ2_1      1
1  more-than-half-the-days  PHQ2_1      2
2  more-than-half-the-days  PHQ2_1      2
3               not-at-all  PHQ2_1      0
4  more-than-half-the-days  PHQ2_1      2
```

19.3 Print survey statistics

Now that we have finally preprocessed the survey, we can extract some meaningful statistic from it.

First, we can compute the mean, standard deviation, min, and max values of all questionnaires.

[5]:
```
d = survey.survey_print_statistic(transformed_df, question_id_col = 'id', answer_col =
↪'answer')
pd.DataFrame(d)
```

[5]:
```
       PHQ2      PSS10      GAD2
min  0.0000   4.000000  0.000000
max  6.0000  27.000000  6.000000
avg  3.0520  14.006000  3.042000
std  1.5855   3.687759  1.536423
```

You can specify the questionnaire that you want statistics of by passing a value into the `prefix` parameter.

[6]:
```
d = survey.survey_print_statistic(transformed_df, question_id_col = 'id', answer_col =
↪'answer', prefix='PHQ')
pd.DataFrame(d)
```

[6]:
```
        PHQ
avg  3.0520
max  6.0000
min  0.0000
std  1.5855
```

CHAPTER

TWENTY

DEMO NOTEBOOK: ANALYSING TRACKER DATA

20.1 Introduction

Fitness tracker is a rich source of longitudinal data captured at high frequency. Those can include step counts, heart rate, calories expenditure, or sleep time. This notebook explains how we can use `niimpy` to extract some basic statistic and features from step count data.

20.2 Read data

[1]:
```
import niimpy
import pandas as pd
import niimpy.preprocessing.tracker as tracker
from niimpy import config
import warnings
warnings.filterwarnings("ignore")
```

[2]:
```
data = pd.read_csv(config.STEP_SUMMARY_PATH, index_col=0)
# Converting the index as date
data.index = pd.to_datetime(data.index)
data.shape
```

[2]:
```
(73, 4)
```

[3]:
```
data.head()
```

[3]:

	user	date	time	steps
2021-07-01 00:00:00	wiam9xme	2021-07-01	00:00:00.000	0
2021-07-01 01:00:00	wiam9xme	2021-07-01	01:00:00.000	0
2021-07-01 02:00:00	wiam9xme	2021-07-01	02:00:00.000	0
2021-07-01 03:00:00	wiam9xme	2021-07-01	03:00:00.000	0
2021-07-01 04:00:00	wiam9xme	2021-07-01	04:00:00.000	0

20.3 Getting basic statistics

Using `niimpy` we can extract a user's step count statistic within a time window. The statistics include:

- `mean`: average number of steps taken within the time range
- `standard deviation`: standard deviation of steps
- `max`: max steps taken within a day during the time range
- `min`: min steps taken within a day during the time range

[4]:
```
tracker.step_summary(data, value_col='steps')
```

[4]:
```
       user  median_sum_step  avg_sum_step  std_sum_step  min_sum_step  \
0  wiam9xme           6480.0   8437.383562   3352.347745          5616

   max_sum_step
0         13025
```

20.4 Feature extraction

Assuming that the step count comes in at hourly resolution, we can compute the distribution of daily step count at each hour. The daily distribution is helpful to look at if for example, we want to see at what hours a user is most active at.

[5]:
```
f = tracker.tracker_daily_step_distribution
step_distribution = tracker.extract_features_tracker(data, features={f: {}})
step_distribution
```

```
{<function tracker_daily_step_distribution at 0x00000190D69F45E0>: {}}
```

[5]:
```
                date                 time  steps  daily_sum  hour  month  day  \
user
wiam9xme  2021-07-01 2021-07-01 00:00:00      0       5616     0      7    1
wiam9xme  2021-07-01 2021-07-01 01:00:00      0       5616     1      7    1
wiam9xme  2021-07-01 2021-07-01 02:00:00      0       5616     2      7    1
wiam9xme  2021-07-01 2021-07-01 03:00:00      0       5616     3      7    1
wiam9xme  2021-07-01 2021-07-01 04:00:00      0       5616     4      7    1
...              ...                 ...    ...        ...   ...    ...  ...
wiam9xme  2021-07-03 2021-07-03 19:00:00    302      12002    19      7    3
wiam9xme  2021-07-03 2021-07-03 20:00:00     12      12002    20      7    3
wiam9xme  2021-07-03 2021-07-03 21:00:00    354      12002    21      7    3
wiam9xme  2021-07-03 2021-07-03 22:00:00      0      12002    22      7    3
wiam9xme  2021-07-03 2021-07-03 23:00:00      0      12002    23      7    3

          daily_distribution
user
wiam9xme            0.000000
wiam9xme            0.000000
wiam9xme            0.000000
wiam9xme            0.000000
wiam9xme            0.000000
...                      ...
wiam9xme            0.025162
```

(continues on next page)

(continued from previous page)

```
wiam9xme            0.001000
wiam9xme            0.029495
wiam9xme            0.000000
wiam9xme            0.000000

[72 rows x 8 columns]
```

```
[ ]:
```

CHAPTER

TWENTYONE

DEMO NOTEBOOK ON READING AND EXPLORING THE STUDENTLIFE DATASET

In this example we download, preprocess and explore the StudentLife Dataset[1].

1.: Wang, Rui, Fanglin Chen, Zhenyu Chen, Tianxing Li, Gabriella Harari, Stefanie Tignor, Xia Zhou, Dror Ben-Zeev, and Andrew T. Campbell. "StudentLife: Assessing Mental Health, Academic Performance and Behavioral Trends of College Students using Smartphones." In Proceedings of the ACM Conference on Ubiquitous Computing. 2014.

[1]:
```
import plotly.express as px
import plotly.io as pio
import warnings
from math import nan, inf
import pandas as pd
import niimpy
from niimpy.exploration.eda import countplot
from niimpy.preprocessing import survey
from niimpy.exploration.eda import categorical
from kaggle.api.kaggle_api_extended import KaggleApi
import zipfile

# Plotly settings. Feel free to adjust to your needs.
pio.renderers.default = "png"
pio.templates.default = "seaborn"
px.defaults.template = "ggplot2"
px.defaults.color_continuous_scale = px.colors.sequential.RdBu
px.defaults.width = 1200
px.defaults.height = 482
warnings.filterwarnings("ignore")

api = KaggleApi()
api.authenticate()

api.dataset_download_files('dartweichen/student-life', path=".")
archive = zipfile.ZipFile('student-life.zip', 'r')
```

[2]:
```
survey_file = archive.open(f"dataset/survey/PHQ-9.csv")
survey_data = pd.read_csv(survey_file)
survey_data = survey_data.rename(columns={'uid': 'user'})
```

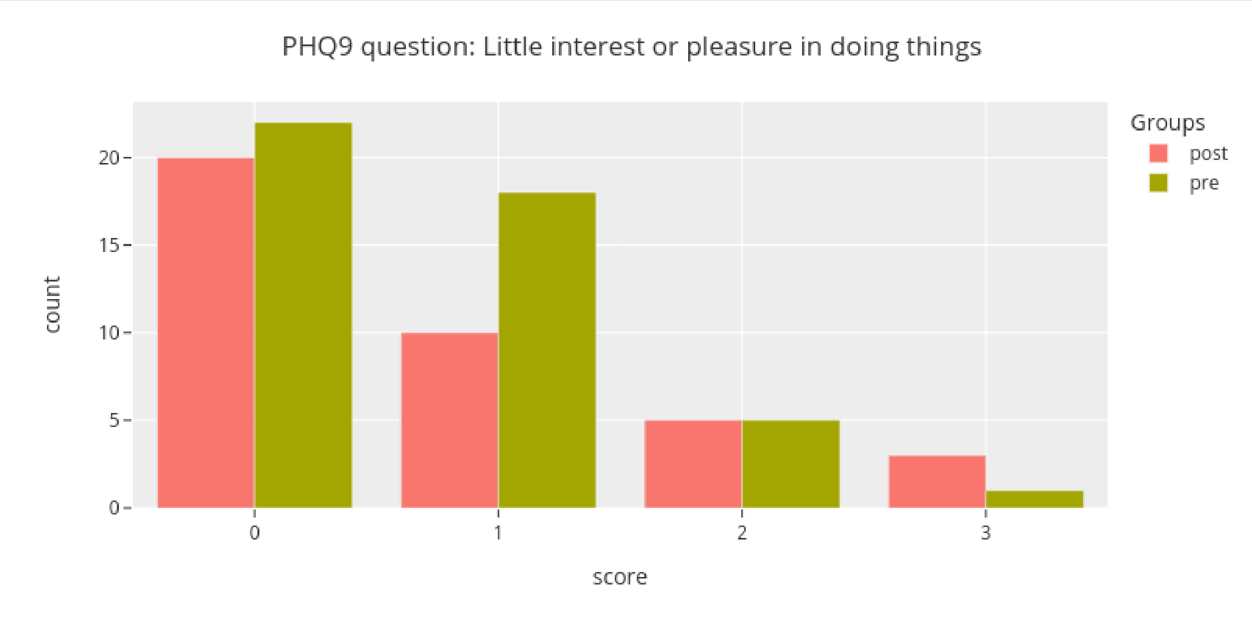
[3]:
```
PHQ9_MAP = {
    'Little interest or pleasure in doing things': "PHQ9_1",
```

(continues on next page)

(continued from previous page)

```
    'Feeling down, depressed, hopeless.': "PHQ9_2",
    'Trouble falling or staying asleep, or sleeping too much.': "PHQ9_3",
    'Feeling tired or having little energy': "PHQ9_4",
    'Poor appetite or overeating': "PHQ9_5",
    'Feeling bad about yourself or that you are a failure or have let yourself or your␣
↪family down': "PHQ9_6",
    'Trouble concentrating on things, such as reading the newspaper or watching␣
↪television': "PHQ9_7",
    'Moving or speaking so slowly that other people could have noticed. Or the opposite␣
↪being so figety or restless that you have been moving around a lot more than usual':
↪"PHQ9_8",
    'Thoughts that you would be better off dead, or of hurting yourself': "PHQ9_9",
}
PHQ9_ANSWER_MAP = {
    "Not at all": 0,
    "Several days": 1,
    "More than half the days": 2,
    "Nearly every day": 3
}
selected_cols = [col for col in survey_data.columns if col in PHQ9_MAP.keys()]
transformed_df = pd.melt(survey_data, id_vars=['user', 'type'], value_vars=selected_cols,
↪ var_name='question', value_name='raw_answer')
transformed_df['id'] = transformed_df['question'].replace(PHQ9_MAP)
transformed_df['answer'] = survey.survey_convert_to_numerical_answer(
    transformed_df, answer_col = 'raw_answer', question_id = 'id',
    id_map={"PHQ9": PHQ9_ANSWER_MAP}, use_prefix=True
)
transformed_df = transformed_df.set_index("user")
```

[4]:
```
fig = categorical.questionnaire_grouped_summary(
    transformed_df,
    question='PHQ9_1',
    group='type',
    title='PHQ9 question: Little interest or pleasure in doing things',
    xlabel='score',
    ylabel='count',
    width=800,
    height=400
)
fig.show()
```



[5]:
```
pre_study_survey = transformed_df[transformed_df["type"] == "pre"]
scores = survey.survey_sum_scores(pre_study_survey, "PHQ9")
```

[6]:
```
def PHQ9_sum_to_group(sum):
    if sum < 5:
        return "minimal"
    elif sum < 10:
        return "mild"
    elif sum < 15:
        return "moderate"
    elif sum < 20:
        return "moderately severe"
    else:
        return "severe"

scores = scores.reset_index()
scores["group"] = scores["score"].apply(PHQ9_sum_to_group)
```

[7]:
```
activity_data = []
for user_number in range(60):
    user = f"u{user_number:02}"
    try:
        csvfile = archive.open(f"dataset/sensing/activity/activity_{user}.csv")
        user_activity = pd.read_csv(csvfile)
        user_activity["user"] = user
        activity_data.append(user_activity)
    except:
        pass
activity_data = pd.concat(activity_data)

# Reduce activity data to whether the user was active or not. Values 1 and 2 represent␣
```

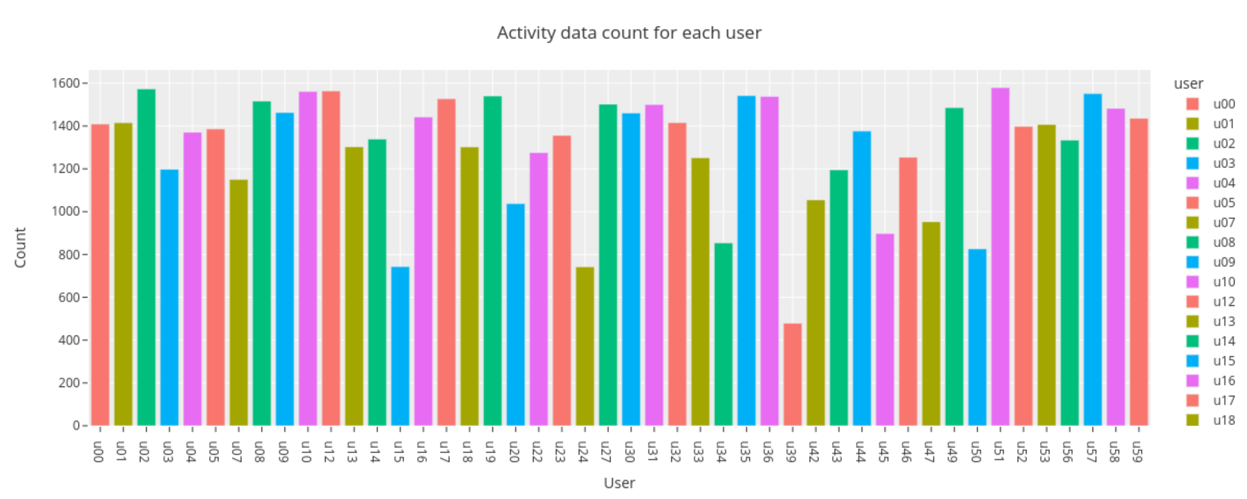
(continues on next page)

(continued from previous page)

```
↪activity.
activity_data = activity_data[activity_data[" activity inference"] != 3]
activity_data["activity"] = activity_data[" activity inference"].isin([1,2]).astype(int)


activity_data.set_index('timestamp',inplace=True)
activity_data.index = pd.to_datetime(activity_data.index, unit='s')

activity_data = niimpy.util.aggregate(activity_data, "1H")
activity_data = activity_data.reset_index("user")
activity_data = activity_data.replace([inf, -inf], nan).dropna(axis=0)
activity_data["activity"] = (activity_data["activity"]*5+0.5).round(0).astype(int)
```

[8]:
```
activity_data = activity_data.reset_index().merge(
    scores[["user", "group"]],
    how="inner",
    on="user",
).set_index("timestamp")
```

[9]:
```
fig = countplot.countplot(activity_data,
                          fig_title='Activity data count for each user',
                          plot_type='count',
                          points='outliers',
                          aggregation='user',
                          user=None,
                          column='activity',
                          binning=False)

fig.show()
```

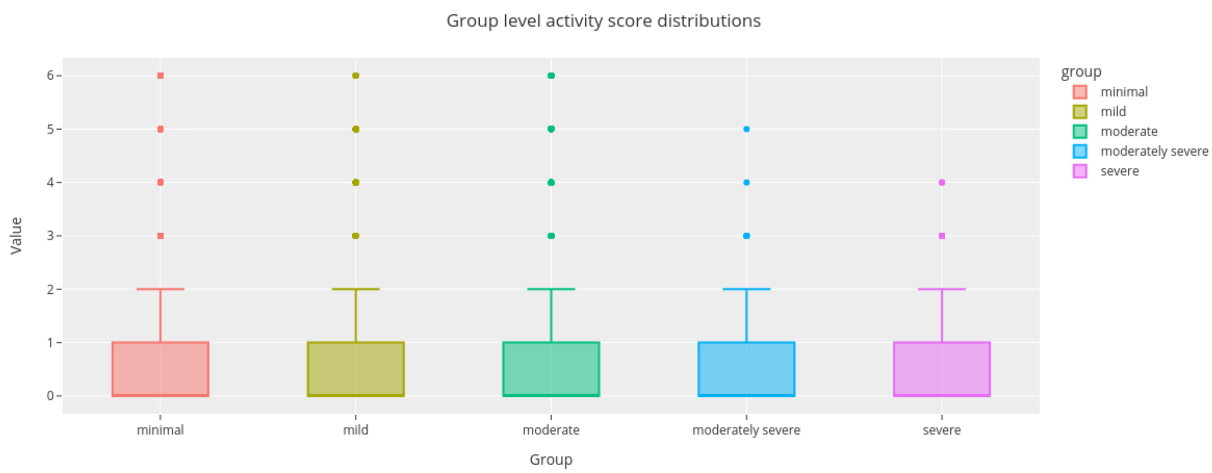


[10]:
```
fig = countplot.countplot(activity_data,
                          fig_title='Group level activity score distributions',
                          plot_type='value',
                          points='outliers',
                          aggregation='group',
```

(continues on next page)

(continued from previous page)

```
                                user=None,
                                column='activity',
                                binning=False)

fig.show()
```



CHAPTER

TWENTYTWO

ADDING FEATURES

22.1 General principles

niimpy is an open source project and general open source contribution guidelines apply - there is no need for us to repeat them right now. Please use Github for communication.

Contributions are welcome and encouraged. * You don't need to be perfect. Suggest what you can and we will help it improve.

22.2 Adding an analysis

- Please add documentatation to a sensor page when you add a new analysis. This should include enough description so that someone else can understand and reproduce all relevant features - enough to describe the method for a scientific article.
- Please add unit tests which test each relevant feature (and each claimed method feature) with a minimal example. Each function can have multiple tests. For examples of unit tests, see below or `niimpy/test_screen.py`. You can create some sample data within each test module which can be used both during development and for tests.

22.3 Common things to note

- You should always use the DataFrame index to retrieve data/time values, not the `datetime` column (which is a convenience thing but not guaranteed to be there).
- Don't require `datetime` in your input
- Have any times returned in the index (unless each row needs multiple times, then do what you need)
- Don't fail if there are extra columns passed (or missing some non-essential columns). Look at what columns/data is passed and and use that, but don't do anything unexpected if someone makes a mistake with input data
- Group by 'user' and 'device' columns if they are present in the input
- Use `niimpy.util._read_sqlite_auto` function for getting data from input
- Use `niimpy.filter.filter_dataframe` to do basic initial filterings based on standard arguments.
- The Zen of Python is always good advice

22.4 Improving old functions

- Add tests for existing functionality
- For every functionality it claims, there should be a minimal test for it.
- Use `read._get_dataframe` and `filter.filter_dataframe` to handle standard arguments
- Don't fail if unnecessary columns are not there (don't drop unneeded columns, select only the needed ones).
- Make sure it uses the index, not the `datetime` column. Some older functions mays still expect it so we have a difficult challenge.
- Improve the docstring of the function: we use the numpydoc format
- Add a documentation page for these sensors, document each function and include an example.
- Document what parameters it groups by when analyzing
- For example an ideal case is that any 'user' and 'device' columns are grouped by in the final output.
- When there are things that don't work yet, you can put a TODO in the docstring to indicate that someone should come back to it later.

22.5 Example unit test

You can read about testing in general in the CodeRefinery testing lesson.

First you would define some sample data. You could reuse existing data (or data from `niimpy.sampledata`), but if data is reused too much then it becomes hard to improve test B because it will affect the data of test A. (do share data when possible but split it when it's relevant).

```
@pytest.fixture
def screen1():
    return niimpy.read_csv(io.StringIO("""\
time,screen_status
0,1
60,0
"""))
```

Then you can make a test function:

```
def test_screen_off(screen1):
    off = niimpy.preprocess.screen_off(screen1)
    assert pd.Timestamp(60,  unit='s', tz=TZ) in off.index
```

`assert` statemnts run the tested functions - when there are errors `pytest` will provide much more useful error messages than you might expect. You can have multiple asserts within a function, to test multiple things.

You run tests with `pytest niimpy/` or `pytest niimpy/test_screen.py`. You can limit to certain tests with `-k` and engage a debugger on errors with `--pdb`.

22.6 Documentation notes

- You can use Jupyter or ReST. ReST is better for narritive documentation.

[ ]:

```

```

CHAPTER TWENTYTHREE

ABOUT DATA SOURCES

This section contains documentation about the contents of various data sources. This is not strictly a task of niimpy: niimpy analyzes any data streams, and you should find the best documentation of the input data from wherever you get that data, and then combine that source knowledge with niimpy analysis documentation to do what you need.

But still, the niimpy developers have their own data sources, and it is useful to include all this information in one place (when we don't have a better place to put it). Other third-party data sources could also be documented here if it proves useful to someone.

CHAPTER

TWENTYFOUR

AWARE

You can read upstream information about Aware sensors from http://www.awareframework.com/sensors/ . This page elaborates the material found there, in particular how the koota-server project processes the data. Still, most of this information could be a useful hints to others using the Aware data.

You can find our previous information on the koota-server wiki, but this information is now being moved here.

Section names in general correspond to the koota-server converter name.

24.1 Standard columns

Some columns that are stored in all the tables.

- `time`: unixtime, time of observation.
- `datetime`: time when the data instance was collected.
- `user`: a unique key to identify a user.
- `device`: a unique key to identify a mobile device.

24.2 AwareAccelerometer

accelerometer data is collected using the phones' accelerometer sensors. The data is used to measure the acceleration of the the phone in any direction of the 3D environment. The coordinate-system is defined relative to the screen of the phone in its default orientation (facing the user). The axis are not swapped when the device's screen orientation changes. The X axis is horizontal and points to the right, the Y axis is vertical and points up and the Z axis points towards the outside of the front face of the screen. In this system, coordinates behind the screen have negative Z axis. The accelerometer sensor measures acceleration and inclues the acceleration due to the force of gravity into consideration. So, if the phone is idle, the accelerometer reads the acceleration of gravity 9.81m/s and if the phone is in free-fall towards the ground, the accelerometer reads 0m/s. The frequency of the data collected can vary largely. It can be in the range of 0 to hundreds of data instances per hour.

- `double_values_0`: acceleration values of X axis.
- `double_values_1`: acceleration values of Y axis.
- `double_values_2`: acceleration values of Z axis.

24.3 AwareApplicationCrashes

contains information about crashed applications. This data is logged whenever any application crashes, which can occur from zero to several times per hour.

- `application_name`: application's localized name.
- `package_name`: application's package name.
- `error_short`: short description of the error.
- `error_long`: more verbose version of the error description.
- `application_version`: version code of the crashed application.
- `error_condition`: type of error has occurred to the application. 1=code error, 2=Application Not Responding (ANR) error

24.4 AwareApplicationNotifications

contains the log of notifications the device has received. This data is logged whenever the phone receives a notification so the frequency of this data can range from zero to hundreds per hour. of times per hour.

- `application_name`: application's localized name.
- `package_name`: application's package name.
- `sound`: notification's sound source.
- `vibrate`: notification's vibration patterns.
- `defaults`: 0=default color, -1=default all, 1=default sound, 2=default vibrate, 3=?, 4=default lights, 6=?, 7=?

24.5 AwareBattery

provides information about the battery and monitors power related events such as phone shutting down or rebooting or charging. The frequency of data sent by battery sensor can be from 0 to tens of times per hour.

- `battery_level`: marks the current percentage of battery charge remaining.
- `battery_status`: 1=unknown, 2=charging, 3=discharging, 4=not charging, 5=full, -1=shut down, -3=reboot.
- `battery_health`: 1=unknown, 2=good, 3=overheat, 4=dead, 5=over voltage, 6=unspecified failure, 7=unknown, 9=?.
- `batery_adaptor`: 0=?, 1=AC, 2=USB, 4=wireless adaptor.

24.6 AwareCalls

logs incoming and outgoing call details. The frequency of AwareMessages data depends upon number of calls the users get so it can be from 0 to tens of times per hour.

- `call_type`: ‘incoming’, ‘outgoing’, ‘missed’.
- `call_duration`: call duration in seconds.
- `trace`: SHA-1 one-way source/target of the call.

24.7 AwareESM

This table provides information about the ESM sensor which adds support for user-provided context by leveraging mobile Experience Sampling Method (ESM). The ESM questionnaires can be triggered by context, time or on-demand, locally or remotely (within your study on AWARE’s dashboard). Although user-subjective, this sensor allows crowd sourcing information that is challenging to instrument with sensors. Depending upon the number of time the users attempt to answer the questions, the frequency can vary from 0 to tens of times per hour.

- `time_asked`: unixtime of the moment the question was asked.
- `id`: the id of the question asked.
- `answer`: the answer to the question asked.
- `type`: 1=text, 2=radio buttons, 3=checkbox, 4=likert scale, 5=quich answer, 6=scale, 9=numeric, 10=web.
- `title`: title of the ESM.
- `instructions`: instructions to answer the ESM.
- `submit`: status of the submission.
- `notification_timeout`: time after which the ESM notification is dismissed and the whole ESM queue

expires (in case expiration threshold is set to 0).

24.8 AwareLocationDayOld

This table ptakes one-day chunks of data and does some processing, for cases where we can’t give raw location data. A day goes from 04:00 one day to 04:00 the next day. Since the information is reliant upon location services being enabled, the frequency can range from zero to several thousands per hour. * `day`: day which is being analyzed, format YYYY-MM-DD. * `totdist`: total distance traveled during the day, meters. * `locstd`: radius of gyration (standard deviation of location throughout the day), meters. * `n_bins`: number of 10-minute intervals with data, including things. * `n_bins_nonnan`: number of these 10-minute intervals with data. * `transtime`: does not work (was supposed to be amount of time you are moving between clusters). * `numclust`: does not work (number of clusters determined with a k-means algorithm, in other words the number of locations they visited. Number of clusters increased until maximum radius is 500m. But maximum number of clusters is 20. This measure may not be accurate). * `entropy`: does not work (was supposed to be p*log(p) of all the cluster memberships. * `normentropy`: does not work.

24.9 AwareLocationDay

This table ptakes one-day chunks of data and does some processing, for cases where we can't give raw location data. A day goes from 04:00 one day to 04:00 the next day. Since the information is reliant upon location services being enabled, the frequency can range from zero to several thousands per hour. * `day`: day which is being analyzed, format YYYY-MM-DD. * `n_points`: the number of raw datapoints. * `n_bins_nonnan`: number of these 10-minute intervals with data. * `n_bins_paired`: the number of bins that also have data right after them. * `ts_min`: first timestamp of any data point of the day (unixtime seconds) * `ts_max`: last timestamp of any data point of the day (unixtime seconds)" (subtracting these two gives the range of data covered which can be contrasted with the next item) * `ts_std`: standard deviation of all timestamps (seconds)". Note that standard deviations of timestamps doesn't actually make that much sense, but combined with the range of timestamps can give you an idea of how spread out through the day the data points are. * `totdist`: total distance covered throughout the day, looking at only the binned averages. If there are large gaps in data, pretend those gaps don't exist and find the distances anyway (meters). * `totdist_raw`: total distance considering every data point (not binned). Probably larger than totdist, more affected by random fluctuations (meters). * `locstd`: Radius of gyration of locations, after the binning (meters). * `radius_mean`: this isn't exactly a radius, but the longest distance between any point an the mean location (both mean location and other points after binning). This measure may not make the most sense, but can be compared to locstd. * `diameter`: Not implemented, always nan. * `n_bins_moving`: number of bins which are considered to be moving. Each bin is compared to the one after to determine an average speed, and n_bins_moving is the number of bins above some threshold. * `n_bins_moving_speed`: number of bins which are moving, using the self-reported speed from Aware. Probably more accurate than the previous. * `n_points_moving_speed`: number of data points (non-binned) which are have a speed above the speed threshold.

24.10 AwareLocationSafe

This table provides information about the users' current location. Since the information is reliant upon location services being enabled, the frequency can range from zero to several thousands per hour.

- `accuracy`: approximate accuracy of the location in meters.
- `double_speed`: users' speed in meters/second over the ground.
- `double_bearing`: location's bearing, in degrees.
- `provider`: describes whether the location information was provided by network or GPS.
- `label`: provides information whether location services was enabled or disabled.

24.11 AwareMessagess

logs incoming and outgoing message details. The frequency of AwareMessages data depends upon number of messages the users get so it can be from 0 to tens of times per hour.

- `message_type`: 'incoming', 'outgoing'.
- `trace`: SHA-1 one-way source/target of the call.

24.12 AwareScreen

This table provides information about the screen status. The number of times this data is collected can range from zero to several hundreds per hour.

- `screen_status`: 0=off, 1=on, 2=locked, 3=unlocked.

24.13 AwareTimestamps

This table lists all the timestamps collected from every data packet that was sent. The frequency of data, since logged for every data packet sent, can range from 0 to tens of thousands per hour.

- `packet_time`: the unixtime of the moment each packet was sent.
- `table`: provides information about which table did the data packet belong to. In other words it describes the kind of data that was being transferred in the packet.

CHAPTER TWENTYFIVE

SURVEY

Provides details about the active data collected from the participants in the form of questionnaires. The survey tables are given below.

25.1 MMMBackgroundAnswers, MMMBaselineAnswers, MMMDiagnosticPatientAnswers, MMMFeedbackPostActiveAnswers, MMMPostActiveAnswers, MMMSurveyAllAnswers

These 6 tables provide details about questions and answers that were asked to the participants of this study. The answers in each of these 6 tables were collected only once per user.

- `id`: uniquely identifies which question was asked to the participant.
- `access_time`: unixtime of the moment when the participant started answering the questions.
- `question`: describes the question that was asked.
- `answer`: provides the participants' answer to the questions. The answers can be of several types.

They can be numbers, small texts or identifier representing a choice for multiple choice questions. * `order`: provides an integer value which represents the number of questions asked before that particular question giving the order of the entire questionnaire. * `choice_text`: represents the texts in the choices of the multiple choice questions which the users selected as answers.

MMMBackgroundMeta, MMMBaselineMeta, MMMDiagnosticPatientMeta, MMMFeedbackPostActiveMeta, MMMPostActiveMeta, MMMSurveyAllMeta —

These 6 tables provide meta data for their respective set of questionnaires' answers. Each of these tables summerize the overall information gathered per user for that particular set of questionnaire. All of the data in these 6 tables were collected only once.

- `name`: the name of the survey.
- `access_time`: unixtime of the moment when the participant started answering the questions.
- `seconds`: describes the time (in seconds) it took for the user to provide the answers.
- `n_questions`: number of questions to be answered in the survey.

CHAPTER

TWENTYSIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

n

INDEX

A

B

C

N

O

P

Q