sport-activities-features

Release 0.3.18

Iztok Fister Jr., Luka Lukač, Alen Rajšp, Luka Pečnik, Dušan Fiste

Mar 14, 2024

USER DOCUMENTATION

sport-activities-features is a minimalistic toolbox for extracting features from sports activity files written in Python.

- **Free software:** MIT license
- **Github repository:** https://github.com/firefly-cpp/sport-activities-features
- **Python versions:** 3.6.x, 3.7.x, 3.8.x, 3.9.x, 3.10.x, 3.11.x, 3.12.x

CHAPTER

ONE

GENERAL OUTLINE OF THE FRAMEWORK

Monitoring sports activities produce many geographic, topologic, and personalized data, with a vast majority of details hidden [1]. Thus, a rigorous ex-post data analysis and statistic evaluation are required to extract them. Namely, most mainstream solutions for analyzing sports activities files rely on integral metrics, such as total duration, total distance, and average hearth rate, which may suffer from the "overall (integral) metrics problem". Among others, such problems are expressed in capturing sports activities in general only (omitting crucial components), calculating potentially fallacious and misleading metrics, not recognizing different stages/phases of the sports activity (warm-up, endurance, intervals), and others [2].

The sport-activities-framework, on the other side, offers a detailed insight into the sports activity files. The framework supports both identification and extraction methods, such as identifying the number of hills, extracting the average altitudes of identified hills, measuring the total distance of identified hills, deriving climbing ratios (total distance of identified hills vs. total distance), average/total ascents of hills and so much more. The framework also integrates many other extensions, among others, historical weather parsing, statistical evaluations, and ex-post visualizations. Previous work on these topical questions was addressed in [3] relevant scientific papers on data mining, also in combination with the generating/predicting automated sport training sessions.

CHAPTER

TWO

DETAILED INSIGHTS

The sport-activities-features framework is compatible with TCX & GPX activity files and Overpass API nodes. Current version withholds (but is not limited to) the following functions:

- extracting integral metrics, such as total distance, total duration, calories (see example),
- extracting topographic features, such as number of hills, the average altitude of identified hills, a total distance of identified hills, climbing ratio, the average ascent of hills, total ascent, total descent (see example),
- plotting identified hills (see example),
- extracting the intervals, such as number of intervals, maximum/minimum/average duration of intervals, maximum/minimum/average distance of intervals, maximum/minimum/average heart rate during intervals,
- plotting the identified intervals (see example),
- calculating the training loads, such as Bannister TRIMP, Lucia TRIMP (see example),
- parsing the historical weather data from an external service,
- extracting the integral metrics of the activity inside the area given with coordinates (distance, heartrate, speed) (see example),
- extracting the activities from CSV file(s) and randomly selecting the specific number of activities (see example),
- extracting the dead ends (see example),
- converting TCX to GPX (see example),
- and much more.

The framework comes with two (testing) benchmark datasets, which are freely available to download from: DATASET1, DATASET2.

CHAPTER

THREE

HISTORICAL WEATHER DATA

Weather data is collected from the Visual Crossing Weather API. Please note that this is an external unaffiliated service, and users must register to use the API. The service has a free tier (1000 Weather reports/day) but otherwise operates on a pay-as-you-go model. For pricing and terms of use, please read the official documentation of the API provider.

CHAPTER

FOUR

DOCUMENTATION

The main documentation is organized into a couple of sections:

- *User Documentation*
- *Developer Documentation*
- *About*

4.1 Getting Started

This section is going to show you how to use the sport-activities-features toolbox.

4.1.1 Installation

Firstly, install sport-activities-features package using the following command:

```
pip install sport-activities-features
```

To install sport-activities-features on Fedora, use:

```
dnf install python3-sport-activities-features
```

To install sport-activities-features on Arch Linux, please use an AUR helper:

```
yay -Syyu python-sport-activities-features
```

To install sport-activities-features on Alpine, use:

```
apk add py3-sport-activities-features
```

After the successful installation you are ready to run your first example.

4.1.2 Examples

You can find usage examples here.

4.2 Installation

4.2.1 Setup development environment

Requirements

- Poetry: https://python-poetry.org/docs/

After installing Poetry and cloning the project from GitHub, you should run the following command from the root of the cloned project:

```
$ poetry install
```

All of the project's dependencies should be installed and the project ready for further development. **Note that Poetry creates a separate virtual environment for your project.**

Development dependencies

List of sport-activities-features dependencies:

Package	Version	Platform
geopy	^2.0.0	All
matplotlib	^3.3.3	All
tcxreader	^0.3.10	All
requests	^2.25.1	All
niaaml	^1.1.6	All
overpy	^1.23.1	All
gpxpy	^1.4.2	All
geotiler	^0.14.5	All
numpy	^1.23.1	All
dotmap	^1.3.25	All

List of development dependencies:

Package	Version	Platform
Sphinx	^3.5.1	Any
sphinx-rtd-theme	^0.5.1	Any

4.3 Testing

Before making a pull request, if possible provide tests for added features or bug fixes.

In case any of the test cases fails, those should be fixed before we merge your pull request to master branch.

For the purpose of checking if all test are passing locally you can run following command:

```
$ poetry run python -m unittest discover
```

4.4 Documentation

To locally generate and preview documentation run the following command in the project root folder:

```
$ poetry run sphinx-build ./docs ./docs/_build
```

If the build of the documentation is successful, you can preview the documentation in the docs/_build folder by clicking the `index.html` file.

4.5 API

This is the sport-activities-features API documentation, auto generated from the source code.

4.5.1 Activity generator

class sport_activities_features.activity_generator.**SportyDataGen**(***kwargs*)

Bases: `object`

Class that contains selected and modified SportyDataGen methods for generation of sports activity collections.

Parameters
****kwargs** – various arguments

Reference:
Fister Jr., I., Vrbančič, G., Brezočnik, L., Podgorelec, V., & Fister, I. (2018). SportyDataGen: An Online Generator of Endurance

Sports Activity Collections.

In Central European Conference on Information and Intelligent Systems (pp. 171-178). Faculty of Organization and Informatics Varazdin.

Reference URL:
http://www.iztok-jr-fister.eu/static/publications/225.pdf

Note: [WIP] This class is still under developement, therefore its methods may not work as expected.

random_generation_without_clustering(*activities*) → None

Method for the random generation of sport activities (without clustering).

Parameters
activities –

Note:

Select n activities randomly without any special preprocessing tasks.

4.5.2 Area Identification

class sport_activities_features.area_identification.**AreaIdentification**(*positions: array*, *distances: array*, *timestamps: array*, *heart_rates: array*, *area_coordinates: array*)

Bases: object

Area identification based by coordinates.

Parameters

- **positions** (*np.array*) – coordinates of positions as an array of latitudes and longitudes
- **distances** (*np.array*) – cummulative distances as an array of floats
- **timestamps** (*np.array*) – information about time as an array of datetimes
- **heart_rates** (*np.array*) – heart rates as an array of integers
- **area_coordinates** (*np.array*) – coordinates of the area where data is analysed as an array of latitudes and longitudes

Reference:

L. Lukač, "Extraction and Analysis of Sport Activity Data Inside Certain Area", 7th Student Computer Science Research Conference StuCoSReC, 2021, pp. 47-50, doi: https://doi.org/10.18690/978-961-286-516-0.9.

do_two_line_segments_intersect(*p1: array*, *p2: array*, *p3: array*, *p4: array*) → bool

Method for checking whether two line segments have an intersection point.

Parameters

- **p1** (*np.array*) – first point of the first line as a pair of coordinates
- **p2** (*np.array*) – second point of the first line as a pair of coordinates
- **p3** (*np.array*) – first point of the second line as a pair of coordinates
- **p4** (*np.array*) – second point of the second line as a pair of coordinates

Returns
True if the two lines have an intersection point, False otherwise.

Return type
bool

static draw_activities_inside_area_on_map(*activities: array*, *area_coordinates: array*) → None

Static method for drawing all the activities inside of an area on the map.

Parameters

- **activities** (*np.array*) – array of AreaIdentification objects
- **area_coordinates** (*np.array*) – border coordinates of an area as an array of latitudes and longitudes.

draw_map() → None

Method for the visualization of the exercise on the map using Geotiler.

extract_data_in_area() → dict

Method for extracting the data of the identified points in area.

Returns: area_data: {

'distance': distance, 'time': time, 'average_speed': average_speed, 'minimum_heart_rate': minimum_heart_rate, 'maximum_heart_rate': maximum_heart_rate, 'average_heart_rate': average_heart_rate

}.

static get_area_coordinates_around_point(*point: array*, *distance: int*) → array

Static method to get area coordinates around the point on Earth according to given distance. Area limits consist of 4 border points.

Parameters

- **point** (*np.array*) – a pair of Earth coordinates
- **distance** (*int*) – desired distance from given point to area border points

Returns
an array of area coordinates.

Return type
np.array

identify_points_in_area() → None

Method for identifying the measure points of the activity inside of the specified area.

is_equal(*value_1: float*, *value_2: float*) → bool

Method for checking whether the two float values are equal with certain tolerance (because of round error).

Parameters

- **value_1** (*float*) – first value
- **value_2** (*float*) – second value

Returns
True if the two values are equal, false otherwise.

Return type
bool

static plot_activities_inside_area_on_map(*activities: array*, *area_coordinates: array*) → None

Static method for plotting the area borders and the activities (or their parts) inside of an area.

Parameters

- **activities** (*np.array*) – array of AreaIdentification objects

- **area_coordinates** (*np.array*) – border coordinates of an area as an array of latitudes and longitudes.

plot_map() → None

Method for plotting the map using Geotiler according to the object variables.

4.5.3 Classes

class sport_activities_features.classes.**StoredSegments**(*segment*, *ascent*, *average_slope=None*)

Bases: object

Class for stored segments.

Parameters

- **()** (*ascent*) –
- **()** –
- **average_slope()** –

Note:

[WIP] This class is still under developement, therefore its methods may not work as expected.

4.5.4 Data Analysis

class sport_activities_features.data_analysis.**DataAnalysis**(***kwargs*)

Bases: object

Class for data analysis that uses automated machine learning to analyze extracted sport features.

Parameters

****kwargs** – various arguments.

analyze_data(*src: str*, *fitness_name: str*, *population_size: uint32*, *number_of_evaluations: uint32*, *optimization_algorithm: str*, *classifiers: Iterable*, *feature_selection_algorithms: Iterable = None*, *feature_transform_algorithms: Iterable = None*, *imputer: str = None*) → Pipeline

Method for running AutoML process using NiaAML PipelineOptimizer class instance.

Parameters

- **src** (*str*) – path to a CSV file
- **fitness_name** (*str*) – name of the fitness class to use as a function
- **population_size** (*uint*) – number of individuals in the optimization process
- **number_of_evaluations** (*uint*) – number of maximum evaluations
- **optimization_algorithm** (*str*) – name of the optimization algorithm to use
- **classifiers** (*Iterable[Classifier]*) – array of names of possible classifiers
- **feature_selection_algorithms** (*Optional[Iterable[str]]*) – array of names of possible feature selection algorithms
- **feature_transform_algorithms** (*Optional[Iterable[str]]*) – array of names of possible feature transform algorithms

- **imputer** (*Optional[str]*) – name of the imputer used for features that contain missing values

Returns
instance of Pipeline object from the NiaAML framework

Return type
Pipeline

Note: See NiaAML's documentation for more details on possible input parameters' values and further usage of the returned Pipeline object.

---

static load_pipeline(*file_name: str*) → Pipeline

Method for loading a NiaAML's pipeline from a binary file.

Parameters
file_name (*str*) – path to a binary pipeline file

Note: See NiaAML's documentation for more details on the use of the Pipeline class.

4.5.5 Data Extraction

class sport_activities_features.data_extraction.**DataExtraction**(*activities: list*)

Bases: object

Class for storing activities' analysed data in CSV files.

Parameters
activities (*list*) – list of activities.

extract_data(*path: str*) → None

This method is used for extracting the data of the activities into separate CSV files.

Parameters
path (*str*) – absolute path where the CSV files should be saved.

4.5.6 Data Extraction from csv files

class sport_activities_features.data_extraction_from_csv.**DataExtractionFromCSV**(*activities: list = None*)

Bases: object

Class for extracting data from CSV files.

Parameters
activities (*list*) – list of activities.

from_all_files(*path: str*) → list

Method for extracting data to list of dataframes from all CSV files in the folder.

Parameters
path (*str*) – absolute path to the folder with CSV files

Returns
list of activities.

Return type
list

from_file(*path: str*) → list

Method for extracting data from CSV file to dataframe.

Parameters
path (*str*) – absolute path to the CSV file

Returns
list of activities.

Return type
list

select_random_activities(*number: int*) → list

Method for selecting random activities.

Parameters
number (*int*) – desired number of random activities

Returns
list of random activities.

Return type
list

4.5.7 Dead End Identification

class sport_activities_features.dead_end_identification.**DeadEndIdentification**(*positions: array*, *distances: array*, *tolerance_degrees: float = 5.0*, *tolerance_position: float = 5.0*, *minimum_distance: int = 500*, *U_turn_allowed_distance: int = 50*)

Bases: object

Class for identifying and visualising dead ends in an exercise. Dead end is a part of an exercise, where an athlete suddenly makes a U-turn takes the same path as before the U-turn is conducted in the opposite direction.

Parameters

- **positions** (*np.array*) – array of positions as pairs of latitudes and longitudes
- **distances** (*np.array*) – array of cummulative distances
- **tolerance_degrees** (*float*) – tolerance of driving the same route in the opposite direction given in degrees

- **tolerance_position** (*float*) – tolerance of positions given in meters
- **minimum_distance** (*int*) – minimum distance of a dead end
- **U_turn_allowed_distance** (*int*) – maximum distance of a U-turn while turning around and starting a dead end

---

Note: [WIP] This class is still under developement, therefore its methods may not work as expected.

---

draw_map() → None

Method for visualisation of the exercise with identified dead ends.

identify_dead_ends() → None

Method for identifying dead ends of the exercise.

is_dead_end(*azimuth_1: float*, *azimuth_2: float*, *tolerance_azimuth: float*) → bool

Method for checking whether two azimuths represent a part of a dead end allowing the given tolerance.

Parameters

- **azimuth_1** (*float*) – first azimuth
- **azimuth_2** (*float*) – second azimuth
- **tolerance_azimuth** (*float*) – difference tolerance of the two azimuths

Returns

True if given azimuths are within the given tolerance,
False otherwise.

Return type
bool

long_enough_to_be_a_dead_end(*start_distance: float*, *finish_distance: float*) → bool

Method for checking whether a dead end is long enough to be a dead end.

Parameters

- **start_distance** (*float*) – cummulative distance at the start of the dead end
- **finish_distance** (*float*) – cummulative distance at the end of the dead end

Returns
True if dead end is long enough, False otherwise.

Return type
bool

really_is_dead_end(*position1: array*, *position2: array*, *tolerance_coordinates: float*) → bool

Method for checking whether a dead end really is a dead end.

Parameters

- **position1** (*np.array*) – position of the first point
- **position2** (*np.array*) – position of the second point
- **tolerance_coordinates** (*float*) – the tolerance between the two positions in meters

Returns

True if a track segment is a part of dead end,
False otherwise.

Return type
bool

reorganize_exercise_data(*positions: array*, *distances: array*, *interval_distance: int = 1*) → None

Method for reorganising the exercise in the way that the trackpoints are organized in a constant interval of distance.

Parameters

- **positions** (*np.array*) – array of positions as pairs of latitudes and longitudes
- **distances** (*np.array*) – array of cummulative distances
- **interval_distance** (*int*) – desired distance between two neighboring points.

show_map() → <module 'matplotlib.pyplot' from '/usr/lib/python3.12/site-packages/matplotlib/pyplot.py'>

Method for plotting the exercise with dead ends.

Return type
plt.

4.5.8 File Manipulation

class sport_activities_features.file_manipulation.**FileManipulation**

Bases: object

Superclass of GPXFile and TCXFile. Contains common methods of both classes, that have the same implementation. e.g. (filling missing values).

count_missing_values(*list*)

Counts the number of elements with value Nona.

Args:

list (list/ndarray): list to check

returns
number of elements with value None in list.

rtype
(int)

linear_fill_missing_values(*activity*, *key*, *max_seconds=15*)

Function that lineary fills missing values, if the successive missing values are up to (max_seconds) apart.

Args:

activity: TCXReader read file key (str): dictionary key (e.g. 'heartrates', 'distances', ...) max_seconds (int): maximum time between two valid values, to still fill the missing values.

Returns:

/ Transforms the sent array / list.

4.5.9 GPX Manipulation

class sport_activities_features.gpx_manipulation.**GPXFile**

Bases: *FileManipulation*

Class for reading GPX files.

extract_integral_metrics(*filename*) → dict

Method for parsing one GPX file and extracting integral metrics.

Returns: int_metrics: {

"activity_type": activity_type, "distance": distance, "duration": duration, "calories": calories, "hr_avg": hr_avg, "hr_max": hr_max, "hr_min": hr_min, "altitude_avg": altitude_avg, "altitude_max": altitude_max, "altitude_min": altitude_min, "ascent": ascent, "descent": descent,

}.

read_directory(*directory_name: str*) → list

Method for finding all GPX files in a directory.

Parameters
directory_name (*str*) – name of the directory in which to identify GPX files

Returns
array of paths to the identified files.

Return type
list

read_one_file(*filename*, *numpy_array=False*)

Method for parsing one GPX file.

Parameters

- **filename** (*str*) – name of the TCX file to be read
- **numpy_array** (*bool*) – if set to true dictionary lists are transformed into numpy.arrays

Returns: activity: {

'activity_type': activity_type, 'positions': positions, 'altitudes': altitudes, 'distances': distances, 'total_distance': total_distance, 'timestamps': timestamps, 'heartrates': heartrates, 'speeds': speeds

}.

Note:

In the case of missing value in raw data, we assign None.

class sport_activities_features.gpx_manipulation.**GPXTrackPoint**(*longitude: float = None*, *latitude: float = None*, *elevation: float = None*, *time=None*, *distance=None*, *hr_value: int = None*, *cadence=None*, *watts: float = None*, *speed: float = None*)

Bases: object

Class for saving GPX point records.

Parameters

- **longitude** (*float*) – longitude in degrees
- **latitude** (*float*) – latitude in degrees
- **elevation** (*float*) – elevation in meters
- **time** (*datetime*) – datetime of time at the given point
- **distance** (*float*) – total distance travelled until this point
- **hr_value** (*int*) – heart beats per minute at given recording.
- **cadence** (*int*) – cadence
- **watts** (*float*) – watts power rating
- **speed** (*float*) – speed in km/h.

from_GPX(*gpx: GPXTrackPoint*, *hr_value: int = None*, *cadence: int = None*, *watts: int = None*) → None

Helper method for initialising GPXTrackPoint from the gpxpy.gpx.GpxTrackPoint.

Parameters

- **gpx** (*gpxpy.gpx.GPXTrackPoint*) – gpxpy.gpx.GPXTrackPoint not to be confused with class of the same name used in gpx_manipulation
- **hr_value** (*int*) – heart beats per minute at given recording
- **cadence** (*int*) – cadence
- **watts** (*int*) – watts power rating.

4.5.10 Hills

class sport_activities_features.hill_identification.**GradeUnit**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Enum for selecting the type of data we want returned in hill slope calculation (degrees / radians or gradient (%))

class sport_activities_features.hill_identification.**HillIdentification**(*altitudes: List[float]*, *distances: List[float] = None*, *ascent_threshold: float = 30*)

Bases: object

Class for identification of hills from TCX file.

Parameters

- **altitudes** (*list*) – an array of altitude values extracted from TCX file
- **ascent_threshold** (*float*) – parameter that defines the hill (hill >= ascent_threshold)
- **distances** (*list*) – optional, allows calculation of hill grades (steepnes)

identify_hills() → None

Method for identifying hills and extracting total ascent and descent from data.

Note: [WIP] Algorithm is still in its preliminary stage.

return_hill(*ascent: float*, *ascent_threshold: float = 30*) → bool

Method for identifying whether the hill is steep enough to be identified as a hill.

Parameters

- **ascent** (*float*) – actual ascent of the hill
- **ascent_threshold** (*float*) – threshold of the ascent that is used for identifying hills

Returns
True if the hill is recognised, False otherwise

Return type
bool

return_hills() → list

Method for returning identified hills.

Returns
array of identified hills

Return type
list

4.5.11 Intervals

class sport_activities_features.interval_identification.**IntervalIdentificationByHeartRate**(*distances: list*, *timestamps: list*, *altitudes: list*, *heart_rates: list*, *minimum_time: int = 30*)

Bases: object

Class for identifying intervals based on heart rate.

Parameters

- **distances** (*list*) – list of cummulative distances
- **timestamps** (*list*) – list of timestamps
- **altitudes** (*list*) – list of altitudes
- **heart_rates** (*list*) – list of heart rates
- **minimum_time** (*int*) – minimum time of an interval given in seconds

calculate_interval_statistics() → dict

Method for calculating interval statistics.

Returns

data = {
'number_of_intervals': number_of_intervals, 'min_duration_interval': min_duration_interval, 'max_duration_interval': max_duration_interval, 'avg_duration_interval': avg_duration_interval, 'min_distance_interval': min_distance_interval, 'max_distance_interval': max_distance_interval, 'avg_distance_interval': avg_distance_interval, 'min_heartrate_interval': min_heartrate_interval, 'max_heartrate_interval': max_heartrate_interval, 'avg_heartrate_interval': avg_heartrate_interval,

}

identify_intervals() → None

Method for identifying intervals from given data.

return_intervals() → list

Method for retrieving identified intervals.

Returns
identified intervals

Return type
list

class sport_activities_features.interval_identification.**IntervalIdentificationByPower**(*distances: list*, *timestamps: list*, *altitudes: list*, *mass: int*, *minimum_time: int = 30*)

Bases: object

Class for identifying intervals based on power.

Parameters

- **distances** (*list*) – list of cummulative distances
- **timestamps** (*list*) – list of timestamps
- **altitudes** (*list*) – list of altitudes
- **mass** (*int*) – total mass of an athlete given in kilograms
- **minimum_time** (*int*) – minimum time of an interval given in seconds

calculate_interval_statistics() → dict

Method for calculating interval statistics.

Returns

data = {
'number_of_intervals': number_of_intervals, 'min_duration': min_duration_interval, 'max_duration': max_duration_interval, 'avg_duration': avg_duration_interval, 'min_distance': min_distance_interval, 'max_distance': max_distance_interval, 'avg_distance': avg_distance_interval,

}

identify_intervals() → None

Method for identifying intervals from given data.

return_intervals() → list

Method for retrieving identified intervals.

Returns
identified intervals

Return type
list

4.5.12 Missing Elevation Identification

class sport_activities_features.missing_elevation_identification.**ElevationIdentification**(*open_elevation_ap*
str
=
'https://api.open-
elevation.com/api/
po-
si-
tions:
list
=
[])

Bases: `object`

Class for retrieving elevation data using Open Elevation Api.

Parameters

- **open_elevation_api** (*str*) – address of the Open Elevation Api, default https://api.open-elevation.com/api/v1/lookup
- **positions** (*list[(lat1, lon1), (lat2, lon2) ...]*) – list of tuples of latitudes and longitudes.

fetch_elevation_data(*payload_formatting: bool = True*) → list

Method for making requests to the Open Elevation API to retrieve elevation data.

Parameters
payload_formatting (*bool*) – True -> break into chunks, False -> don't break self.positions into chunks

Returns
list of elevations for the given positions.

Return type
list[int]

4.5.13 Overpy Node Manipulation

class sport_activities_features.overpy_node_manipulation.**OverpyNodesReader**(*open_elevation_api:*
str =
'https://api.open-
elevation.com/api/v1/lookup?')

Bases: `object`

Class for working with Overpass nodes (Overpy.node). The purpose is to generate a dictionary object similar to those generated by TCXFile and GPXFile classes.

Parameters
open_elevation_api (*str*) – location of the Open Elevation Api.

read_nodes(*nodes: Node*, *cumulative_distances: bool = True*) → dict

Method for reading overpy.Node nodes and generating a TCXFile/GPXFile like dictionary of objects.

Parameters

- **nodes** (*list*) – list of overpy.Node objects

- **cumulative_distances** (*bool*) – If set to True, distance equals previous point distance + distance between the nodes, else tells actual distance between two points.

Returns: dictionary of nodes.

{
'activity_type': str, 'positions': [...], 'altitudes': [...], 'distances': [...], 'total_distance': float
}

4.5.14 Plot data

class sport_activities_features.plot_data.**PlotData**

Bases: object

Class for plotting the extracted data.

draw_basic_map() → None

Method for plotting the whole topographic map and rendering the plot.

draw_hills_in_map(*altitude: list*, *distance: list*, *identified_hills: list*) → None

Method for plotting all hills identified in data on topographic map and rendering the plot.

Parameters

- **altitude** (*list*) – list of altitudes
- **distance** (*list*) – list of distances
- **identified_hills** (*list*) – list of identified hills.

draw_intervals_in_map(*timestamp: list*, *distance: list*, *identified_intervals: list*) → None

Method for plotting all intervals identified in data on topographic map and rendering the plot.

Parameters

- **timestamp** (*datetime*) – list of timestamps
- **distance** (*float*) – list of distances
- **identified_intervals** (*list*) – list of identified intervals.

get_positions_of_hills(*identified_hills: list*) → list

Method for retrieving positions of identified hills.

Parameters
identified_hills (*list*) – list of identified hills

Returns
list of hills.

Return type
list

get_positions_of_intervals(*identified_intervals: list*) → list

Method for retrieving positions of identified intervals.

Parameters
identified_intervals (*list*) – list of identified intervals

Returns
list of intervals.

Return type
list

plot_basic_map(*altitude: list*, *distance: list*) → <module 'matplotlib.pyplot' from '/usr/lib/python3.12/site-packages/matplotlib/pyplot.py'>

Method for plotting the whole topographic map.

Parameters

- **altitude** (*list*) – list of altitudes
- **distance** (*list*) – list of distances

Returns
plt.

plot_hills_on_map(*altitude: list*, *distance: list*, *identified_hills: list*) → <module 'matplotlib.pyplot' from '/usr/lib/python3.12/site-packages/matplotlib/pyplot.py'>

Method for plotting all hills identified in data on topographic map.

Parameters

- **altitude** (*list*) – list of altitudes
- **distance** (*list*) – list of distances
- **identified_hills** (*list*) – list of identified hills

Returns
plt.

plot_intervals_in_map(*timestamp: list*, *identified_intervals: list*) → <module 'matplotlib.pyplot' from '/usr/lib/python3.12/site-packages/matplotlib/pyplot.py'>

Method for plotting all intervals identified in data on topographic map.

Parameters

- **timestamp** (*list*) – list of timestamps
- **identified_intervals** (*list*) – list of identified intervals

Returns
plt.

4.5.15 TCX manipulation

class sport_activities_features.tcx_manipulation.**TCXFile**

Bases: *FileManipulation*

Class for reading TCX files.

create_gps_object(*path_to_the_file*)

Convert TCX file to GPX file.

extract_integral_metrics(*filename: str*) → dict

Method for parsing one TCX file and extracting integral metrics.

Parameters
filename (*str*) – name of the TCX file to be read

Returns

int_metrics = {
'activity_type': activity_type, 'distance': distance, 'duration': duration, 'calories': calories, 'hr_avg': hr_avg, 'hr_max': hr_max, 'hr_min': hr_min, 'altitude_avg': altitude_avg, 'altitude_max': altitude_max, 'altitude_min': altitude_min, 'ascent': ascent, 'descent': descent, 'steps' : steps

}.

read_directory(*directory_name: str*) → list

Method for finding all TCX files in a directory.

Parameters
directory_name (*str*) – name of the directory in which to identify TCX files

Returns
array of paths to the identified files.

Return type
str

read_one_file(*filename: str*, *numpy_array=False*) → dict

Method for parsing one TCX file using the TCXReader.

Parameters

- **filename** (*str*) – name of the TCX file to be read
- **numpy_array** (*bool*) –

  if set to true dictionary lists are
  transformed into numpy.arrays

Returns

activity = {
'activity_type': activity_type, 'positions': positions, 'altitudes': altitudes, 'distances': distances, 'total_distance': total_distance, 'timestamps': timestamps, 'heartrates': heartrates, 'speeds': speeds

}.

Note:

In the case of missing value in raw data, we assign None.

write_gpx(*gps_object*, *output_file_name=None*)

Write GPX object to file. if output_file_name is not specified, the output file name will be the same as the input file name, but with .gpx extension.

4.5.16 Topographic features

class sport_activities_features.topographic_features.**TopographicFeatures**(*identified_hills: list*)

Bases: object

Class for feature extraction from topographic maps.

Parameters
identified_hills (*list*) – identified hills are now passed to this class.

ascent(*altitude_data: list*) → float

Method for ascent calculation.

Parameters
altitude_data (*list*) – list of altitudes

Returns
total ascent

Return type
float

Note: [WIP] This method should be improved.

avg_altitude_of_hills(*alts: list*) → float

Method for calculating the average altitude of all identified hills in sport activity.

Parameters
alts (*list*) – list of altitudes

Returns
average altitude.

Return type
float

avg_ascent_of_hills(*alts: list*) → float

Method for calculating the average ascent of all hills in sport activity.

Parameters
alts (*list*) – list of altitudes

Returns
average ascent.

Return type
float

calculate_distance(*latitude_1: float*, *latitude_2: float*, *longitude_1: float*, *longitude_2: float*) → float

Method for calculating the distance between the two pairs of coordinates.

Parameters

- **latitude_1** (*float*) – first latitude
- **latitude_2** (*float*) – second latitude
- **longitude_1** (*float*) – first longitude
- **longitude_2** (*float*) – second longitude

Returns
distance in kilometers.

Return type
float

`calculate_hill_gradient`(*latitude_1: float*, *latitude_2: float*, *longitude_1: float*, *longitude_2: float*, *height_1: float*, *height_2: float*) → float

Method for calculation of the hill gradient in percent.

Parameters

- **latitude_1** (*float*) – first latitude
- **latitude_2** (*float*) – second latitude
- **longitude_1** (*float*) – first longitude
- **longitude_2** (*float*) – second longitude
- **height_1** (*float*) – first altitude
- **height_2** (*float*) – second altitude

Returns
gradient in degrees.

Return type
float

`descent`(*altitude_data: list*) → float

Method for descent calculation.

Parameters
altitude_data (*list*) – list of altitudes

Returns
total descent

Return type
float

Note: [WIP] This method should be improved.

`distance_of_hills`(*positions: list*) → float

Method for calculating the total distance of all identified hills in sport activity.

Parameters
positions (*list*) – list of positions

Returns
distance of hills.

Return type
float

`num_of_hills`() → int

Method for calculating the number of identified hills in sport activity.

Returns
number of hills.

Return type
int

share_of_hills(*hills_dist: float*, *total_dist: float*) → float

Method for calculating the share of hills in sport activity (percentage).

Parameters

- **hills_dict** (*float*) – distance of all hills
- **total_dist** (*float*) – total distance

Returns
share of hills.

Return type
float

4.5.17 Training loads

This class is used for calculation of training loads.

class sport_activities_features.training_loads.**BanisterTRIMPv1**(*duration: float*, *average_heart_rate: float*)

Bases: object

Class for calculation of simple Banister's TRIMP.

Reference paper:
Banister, Eric W. "Modeling elite athletic performance." Physiological testing of elite athletes 347 (1991): 403-422.

Parameters

- **duration** (*float*) – total duration in seconds
- **average_heart_rate** (*float*) – average heart rate in beats per minute.

calculate_TRIMP() → float

Method for the calculation of the TRIMP.

Returns
Banister TRIMP value.

Return type
float

class sport_activities_features.training_loads.**BanisterTRIMPv2**(*duration: float*, *average_heart_rate: float*, *min_heart_rate: float*, *max_heart_rate: float*, *gender:* Gender = *Gender.male*)

Bases: object

Class for calculation of Banister's TRIMP. .

Reference paper:
Banister, Eric W. "Modeling elite athletic performance." Physiological testing of elite athletes 347 (1991): 403-422.

Args:

duration (float):
total duration in seconds

average_heart_rate (float):
average heart rate in beats per minute

min_heart_rate (float):
minimum heart rate in beats per minute

max_heart_rate (float):
maximum heart rate in beats per minute

gender (Gender):
gender enum of athlete (default male, female)

`calculate_TRIMP()` → float

Calculate TRIMP.

Returns
float

Return type
Banister TRIMP value.

`calculate_delta_hr_ratio()` → float

Calculate the delta heart rate.

The ratio ranges from a low to a high value (i.e., ~ 0.2 — 1.0) for a low or a high raw heart rate, respectively.

Returns
float

Return type
delta heart rate.

`calculate_weighting_factor(`*delta_hr_ratio: float*`)` → float

Calculate the weighting factor.

Returns
float

Return type
weighting factor (Y).

`class sport_activities_features.training_loads.EdwardsTRIMP(`*heart_rates: list*, *timestamps: list*, *max_heart_rate: int = 200*`)`

Bases: `object`

Class for calculation of Edwards TRIMP.

Reference paper:
https://www.frontiersin.org/articles/10.3389/fphys.2020.00480/full

Parameters

- **heart_rates** (*list[int]*) – list of heart rates in beats per minute
- **timestamps** (*list[timestamp]*) – list of timestamps
- **max_heart_rate** (*int*) – maximum heart rate of an athlete.

calculate_TRIMP() → float

Method for the calculation of the TRIMP.

Returns
Edwards TRIMP value.

Return type
float

class sport_activities_features.training_loads.**Gender**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Gender Enum.

class sport_activities_features.training_loads.**LuciaTRIMP**(*heart_rates: list*, *timestamps: list*, *VT1: int = 160*, *VT2: int = 180*)

Bases: object

Class for calculation of Lucia's TRIMP.

Reference:
https://www.trainingimpulse.com/lucias-trimp-0

Parameters

- **heart_rates** (*list[int]*) – list of heart rates in beats per minute
- **timestamps** (*list[timestamp]*) – list of timestamps
- **VT1** (*int*) – ventilatory threshold to divide the low and the moderate zone
- **VT2** (*int*) – ventilatory threshold to divide the moderate and the high zone.

calculate_TRIMP() → float

Method for the calculation of the TRIMP.

Returns
Lucia's TRIMP value.

Return type
float

4.5.18 Weather Identification

class sport_activities_features.weather_identification.**WeatherIdentification**(*locations: list*, *timestamps: list*, *vc_api_key: str*, *unit_group='metric'*)

Bases: object

A class used for identification of Weather data from TCX file. For identification of weather an external API is used (https://www.visualcrossing.com/).

Parameters

- **locations** (*list[(float, float)]*) – coordinates of exercise recordings, found in TCXFile/GPXFile generated dictionary under "positions"

- **timestamps** (*list[datetime]*) – timestamps of exercise recordings, found in TCX-File/GPXFile generated dictionary under "timestamps"
- **vc_api_key** (*str*) – API key for accessing VisualCrossing weather data
- **unit_group** (*str*) – Unit group of data recieved. Possible options 'metric' (default), 'us', 'uk', 'base'.

Warnings:

vc_api_key: api key is required.

classmethod get_average_weather_data(*timestamps: list*, *weather: list*) → list

Method generates average weather for each of the timestamps in training by averaging the weather before and after the timestamp, using the __find_nearest_weathers() method.

Parameters

- **timestamps** (*list[datetime]*) – datetime recordings from the TCXFile parsed data
- **weather** (*list[Weather]*) – list of weather objects retrieved from VisualCrossing API

Returns

list which is an AverageWeather object
for each of the given timestamps.

Return type
list[AverageWeather]

get_weather(*time_delta: int = 30*) → list

Method that queries the VisualCrossing weather API for meteorological data at provided (minute) time intervals.

Parameters
time_delta (*int*) – time between two measurements, default 30 mins

Returns

list of Weather objects from the nearest
meteorological station for every interval (time_delta minutes) of training.

Return type
list[Weather]

4.6 Contributing to sport-activities-features

First off, thanks for taking the time to contribute!

4.6.1 Code of Conduct

This project and everyone participating in it is governed by the *Contributor Covenant Code of Conduct*. By participating, you are expected to uphold this code. Please report unacceptable behavior to iztok.fister1@um.si.

4.6.2 How Can I Contribute?

Reporting Bugs

Before creating bug reports, please check existing issues list as you might find out that you don't need to create one. When you are creating a bug report, please include as many details as possible in the issue template.

Suggesting Enhancements

Open new issue using the feature request template.

Pull requests

Fill in the pull request template and make sure your code is documented.

4.7 Contributor Covenant Code of Conduct

4.7.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

4.7.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission

- Other conduct which could reasonably be considered inappropriate in a professional setting

4.7.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

4.7.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

4.7.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at iztok.fister1@um.si. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

4.7.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

4.7.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

4.8 Contributors

4.8.1 Credits

Maintainers

- Iztok Fister, Jr.

Contributors (alphabetically)

- Dušan Fister
- Nejc Graj
- Rok Kukovec
- Tadej Lahovnik
- Zala Lahovnik
- Luka Lukač
- Luka Pečnik
- Špela Pečnik
- Alen Rajšp

4.9 License

```
MIT License

Copyright (c) 2020-2023 Iztok Fister Jr. et al.

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

BIBLIOGRAPHY

[1] Alen Rajšp and Iztok Fister Jr. A systematic literature review of intelligent data analysis methods for smart sport training. *Applied Sciences*, 10(9):3013, 2020.

[2] Iztok Fister, Iztok Fister Jr, and Dušan Fister. *Computational intelligence in sports*. Volume 22. Springer, 2019.

[3] Iztok Fister Jr., Iztok Fister, Dušan Fister, and Simon Fong. Data mining in sporting activities created by sports trackers. In *2013 international symposium on computational and business intelligence*, 88–91. IEEE, 2013.

[4] Iztok Fister Jr., Luka Lukač, Alen Rajšp, Iztok Fister, Luka Pečnik, and Dušan Fister. A minimalistic toolbox for extracting features from sport activity files. In *2021 IEEE 25th International Conference on Intelligent Engineering Systems (INES)*, 000121–000126. IEEE, 2021.

PYTHON MODULE INDEX

S

INDEX

M

N

O

P

R

S

T

W