sport-activities-features

Release 0.2.15

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

Apr 26, 2022

USER DOCUMENTATION

sport-activities-features is a minimalistic toolbox for extracting features from sport 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

GENERAL OUTLINE OF THE FRAMEWORK

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

The sport-activities-framework on the other side offers a detailed insight into the sport activity files. The framework supports both identification and extraction methods, such as identifying number of hills, extracting the average altitudes of identified hills, measuring 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 were addressed in [1] relevant scientific papers on data mining, also in a combination with the generating/predicting automated sport training sessions.

TWO

DETAILED INSIGHTS

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

- extracting integral metrics, such as total distance, total duration, calories (see example),

- extracting topographic features, such as number of hills, average altitude of identified hills, total distance of identified hills, climbing ratio, 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 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,

- and much more.

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

HISTORICAL WEATHER DATA

Weather data parsed 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 is otherwise operating on a pay-as-you-go model. For pricing and terms of use please read the official documentation of the API provider.

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
```

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	^0.3.0	All
requests	^2.25.1	All
niaaml	^1.1.0	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 localy 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 `sport_activities_features`

class `sport_activities_features.`**`AverageWeather`**(*weather_a: Optional[*[sport_activities_features.weather_objects.Weather.Weather]*]* *= None, weather_b: Optional[*[sport_activities_features.weather_objects.Weather.Weather]*]* *= None, weight_a=None*)

> Bases: `object`

> Class for providing average weather from two Weather objects

class `sport_activities_features.`**`BanisterTRIMP`**(*duration: float, avg_hr: float*)
> Bases: `object`

> Class for calculation of Banisters TRIMP.

> **Args:**

>> **duration (float):** total duration in minutes

>> **avg_hr (float):** average heart rate in bpm

> `calculate_TRIMP`() → float
>> Method for the calculation of TRIMP.

>> **Returns:** float: TRIMP value

class `sport_activities_features.`**`DataAnalysis`**(***kwargs*)
> Bases: `object`

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

> **Args:**

>> ****kwargs:** various arguments

> `analyze_data`(*src: str, fitness_name: str, population_size: numpy.uint64, number_of_evaluations: numpy.uint64, optimization_algorithm: str, classifiers: Iterable, feature_selection_algorithms: Optional[Iterable] = None, feature_transform_algorithms: Optional[Iterable] = None, imputer: Optional[str] = None*) → niaaml.pipeline.Pipeline
>> Method for running AutoML process using NiaAML PipelineOptimizer class instance.

>> **Args:**

>>> **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:** Pipeline: instance of Pipeline object from the NiaAML framework
>
> **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*) → niaaml.pipeline.Pipeline
> Method for loading a NiaAML's pipeline from a binary file.
>
> > **Args:**
> >
> > **file_name (str):** path to a binary pipeline file
> >
> > **Note:** See NiaAML's documentation for more details on the use of the Pipeline class.

class sport_activities_features.**DataExtraction**(*activities: list*)
  Bases: `object`

  Class for storing activities' analysed data in CSV files.

> **Args:**
>
> **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.
>
> > **Args:**
> >
> > **path (str):** absolute path where the CSV files should be saved

class sport_activities_features.**DataExtractionFromCSV**(*activities: Optional[list] = None*)
  Bases: `object`

  Class for extracting data from CSV files.

> **Args:**
>
> **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.
>
> > **Args:**
> >
> > **path (str):** absolute path to the folder with CSV files
>
> **Returns:** list: list of activities

> **from_file**(*path: str*) → list
> Method for extracting data from CSV file to dataframe.
>
> **Args:** path (str): absolute path to the CSV file
>
> **Returns:** list: list of activities

> **select_random_activities**(*number: int*) → list
> Method for selecting random activities.
>
> > **Args:**
> >
> > **number (int):** desired number of random activities
>
> **Returns:** list: list of random activities

class sport_activities_features.**ElevationIdentification**(*open_elevation_api: str = 'https://api.open-elevation.com/api/v1/lookup', positions: list = []*)

> Bases: object
>
> Class for retrieving elevation data using Open Elevation Api.
>
> **Args:**
>
>> **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.
>>
>> **Args:**
>>
>>> **payload_formatting (bool):** True -> break into chunks, False -> don't break self.positions into chunks
>>
>> **Returns:** list[int]: list of elevations for the given positions

class sport_activities_features.**GPXFile**

> Bases: object
>
> 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.
>>
>> **Args:**
>>
>>> **directory_name (str):** name of the directory in which to identify GPX files
>>
>> **Returns:** list: array of paths to the identified files
>
> **read_one_file**(*filename*)
>> Method for parsing one GPX file.
>>
>> **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.**HillIdentification**(*altitudes: list, ascent_threshold: float*)

> Bases: object
>
> Class for identification of hills from TCX file.

Args:

> **altitudes (list):** an array of altitude values extracted from TCX file
>
> **ascent_threshold (float):** parameter that defines the hill (hill >= ascent_threshold)

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_threshold: float*) → bool
Method for identifying whether the hill is steep enough to be identified as a hill.

> **Args:**
>
> > **ascent_threshold (float):** threshold of the ascent that is used for identifying hills
>
> **Returns:** bool: True if the hill is recognised, False otherwise

return_hills() → list
Method for returning identified hills.

> **Returns:** list: array of identified hills

class sport_activities_features.**InterruptionProcessor**(*time_interval=60*, *min_speed=2*, *overpass_api_url='https://lz4.overpass-api.de/api/interpreter'*)

Bases: object

Class for identifying interruption events (events where the speed dropped below threshold). Args:

> time_interval: Record x seconds before and after the event min_speed: Speed threshold for the event to trigger
>
> > (min_speed = 2 -> trigger if speed less than 2km/h)
>
> **overpass_api_url: Overpass API url. Self-hosting is prefferable** if you want to make a lot of requests.

classify_event(*event: sport_activities_features.interruptions.exercise_event.ExerciseEvent*)
Method that classifies the sent ExerciseEvent. Currently only identifies events which happened in the vicincy of intersections. Args:

> event: ExerciseEvent to be inspected

Returns: ExerciseEvent on which classification has been performed.

events(*lines: list*, *classify=False*) → list
Method that parses events (method parse_events()) and classifies them (classify_events()) if required. Args:

> lines: [TrackSegment] from TCX/GPX data classify: If set to true calls classify_events() method

> **Returns: list of [ExerciseEvent], meaning** parsed (and classified) events.

parse_events(*lines: list*) → list
Parses all events (based on the min_speed parameter in the class initialisation) and returns ExerciseEvent list. Args:

> lines: list of TrackSegment objects

Returns: list of identified ExerciseEvent objects

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

> Bases: object
>
> Class for identifying intervals based on power.
>
> **Args:**
>
>> **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:** list: identified intervals

class sport_activities_features.**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.
>>
>> **Args:**
>>
>>> **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.
>>
>> **Args:**
>>
>>> **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.

> **Args:**

> > **identified_hills (list):** list of identified hills

> **Returns:** list: list of hills

get_positions_of_intervals(*identified_intervals: list*) → list
    Method for retrieving positions of identified intervals.

> **Args:**

> > **identified_intervals (list):** list of identified intervals

> **Returns:** list: list of intervals

plot_basic_map(*altitude: list*, *distance: list*) → <module 'matplotlib.pyplot' from
                '/usr/lib64/python3.10/site-packages/matplotlib/pyplot.py'>
    Method for plotting the whole topographic map.

> **Args:**

> > **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/lib64/python3.10/site-packages/matplotlib/pyplot.py'>
    Method for plotting all hills identified in data on topographic map.

> **Args:**

> > **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/lib64/python3.10/site-packages/matplotlib/pyplot.py'>
    Method for plotting all intervals identified in data on topographic map.

> **Args:**

> > **timestamp (list):** list of timestamps

> > **identified_intervals (list):** list of identified intervals

> **Returns:** plt

class sport_activities_features.**StoredSegments**(*segment*, *ascent*)
    Bases: object

    Class for stored segments.

    Args: segment ():

> ascent ():

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

class sport_activities_features.**TCXFile**

   Bases: object

   Class for reading TCX files.

   extract_integral_metrics(*filename: str*) → dict
      Method for parsing one TCX file and extracting integral metrics.

      Args:

         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,

         }

   read_directory(*directory_name: str*) → list
      Method for finding all TCX files in a directory.

      Args:

         directory_name (str): name of the directory in which to identify TCX files

      Returns: str: array of paths to the identified files

   read_one_file(*filename: str*) → dict
      Method for parsing one TCX file using the TCXReader.

      Args:

         filename (str): name of the TCX file to be read

      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.**TopographicFeatures**(*identified_hills: list*)

   Bases: object

   Class for feature extraction from topographic maps.

   Args:

      identified_hills (list): identified hills are now passed to this class

   ascent(*altitude_data: list*) → float
      Method for ascent calculation.

      Args:

         altitude_data (list): list of altitudes

      Returns: float: total ascent

      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.

Args:

>   **alts (list):** list of altitudes

Returns: float: average altitude

avg_ascent_of_hills(*alts: list*) → float
  Method for calculating the average ascent of all hills in sport activity.

Args:

>   **alts (list):** list of altitudes

Returns: float: average ascent

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.

Args:

>   **latitude_1 (float):** first latitude
>
>   **latitude_2 (float):** second latitude
>
>   **longitude_1 (float):** first longitude
>
>   **longitude_2 (float):** second longitude

Returns: float: distance in kilometers

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.

Args:

>   **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: float: gradient in degrees

descent(*altitude_data: list*) → float
  Method for descent calculation.

Args:

>   **altitude_data (list):** list of altitudes

Returns: float: total descent

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.

Args:

>   **positions (list):** list of positions

Returns: float: distance of hills

num_of_hills() → int
Method for calculating the number of identified hills in sport activity.

Returns: int: number of hills

share_of_hills(*hills_dist: float*, *total_dist: float*) → float
Method for calculating the share of hills in sport activity (percentage).

Args:

hills_dict (float): distance of all hills

total_dist (float): total distance

Returns: float: share of hills

class sport_activities_features.**Weather**(*temperature: float = None*, *maximum_temperature: float = None*, *minimum_temperature: float = None*, *wind_chill: float = None*, *heat_index: float = None*, *precipitation: float = None*, *snow_depth: float = None*, *wind_speed: float = None*, *wind_direction=None*, *wind_gust=None*, *visibility: float = None*, *cloud_cover: float = None*, *relative_humidity: float = None*, *weather_type: str = None*, *sea_level_pressure=None*, *dew_point=None*, *solar_radiation=None*, *conditions: str = None*, *date: <module 'datetime' from '/usr/lib64/python3.10/datetime.py'> = None*, *location=None*, *index: int = None*)

Bases: object

A class for the Weather object files. Args reported based on VisualCrossing API description.

class sport_activities_features.**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/).

Args:

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 TCXFile/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.

Args:

timestamps (list[datetime]): datetime recordings from the TCXFile parsed data

weather (list[Weather]): list of weather objects retrieved from VisualCrossing API

Returns:

list[AverageWeather]: list which is an AverageWeather object for each of the given timestamps

get_weather(*time_delta: int = 30*) → list
>   Method that queries the VisualCrossing weather API for meteorological data at provided (minute) time intervals.

>   **Args:**

>>   **time_delta (int):** time between two measurements, default 30 mins

>   **Returns:**

>>   **list[Weather]: list of Weather objects from the nearest** meteorological station for every interval (time_delta minutes) of training

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

- Dušan Fister
- Luka Lukač
- Luka Pečnik
- Špela Pečnik
- Alen Rajšp

4.9 License

```
MIT License

Copyright (c) 2020-2022 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] 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.

[2] 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