SoftwareX 17 (2022) 100971

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

tsflex: Flexible time series processing & feature extraction

Jonas Van Der Donckt [*,1], Jeroen Van Der Donckt [1], Emiel Deprost, Sofie Van Hoecke

IDLab, Ghent University - imec, Technologiepark Zwijnaarde 126, 9052 Zwijnaarde, Belgium

ARTICLE INFO

Article history:
Received 15 September 2021
Received in revised form 16 December 2021
Accepted 21 December 2021

Keywords:
Time series
Processing
Feature extraction
Machine learning
Python

ABSTRACT

Time series processing and feature extraction are crucial and time-intensive steps in conventional machine learning pipelines. Existing packages are limited in their applicability, as they cannot cope with irregularly-sampled or asynchronous data and make strong assumptions about the data format. Moreover, these packages do not focus on execution speed and memory efficiency, resulting in considerable overhead. We present `tsflex`, a Python toolkit for time series **processing and feature extraction**, that focuses on performance and flexibility, enabling broad applicability. This toolkit leverages window-stride arguments of the same data type as the sequence-index, and maintains the sequence-index through all operations. `tsflex` is **flexible** as it supports (1) multivariate time series, (2) multiple window-stride configurations, and (3) integrates with processing and feature functions from other packages, while (4) making no assumptions about the data sampling regularity, series alignment, and data type. Other functionalities include multiprocessing, detailed execution logging, chunking sequences, and serialization. Benchmarks show that `tsflex` is **faster** and more **memory-efficient** compared to similar packages, while being more permissive and flexible in its utilization.

Code metadata

Current code version	v0.2.3
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-21-00171
Code Ocean compute capsule	Not available
Legal Code License	MIT
Code versioning system used	git
Software code languages, tools, and services used	Python3.7+
Compilation requirements, operating environments & dependencies	poetry is used for dependency management, package installation, and publishing to PyPi. The poetry-toml file is used to install the required packages.
If available Link to developer documentation/manual	https://predict-idlab.github.io/tsflex
Support email for questions	jonvdrdo.vanderdonckt@ugent.be

Software metadata

Current software version	v0.2.3
Permanent link to executables of this version	https://github.com/predict-idlab/tsflex
Legal Software License	MIT
Computing platforms/Operating Systems	Linux, OS X, Microsoft Windows
Installation requirements & dependencies	Python3.7+
If available, link to user manual - if formally published include a reference to the publication in the reference list	https://predict-idlab.github.io/tsflex
Support email for questions	jonvdrdo.vanderdonckt@ugent.be

1. Motivation and significance

Data-driven modelling and forecasting of time series is a major topic of interest in academic research and industrial applications,

* Corresponding author.
E-mail address: jonvdrdo.vanderdonckt@ugent.be (Jonas Van Der Donckt).
[1] Contributed equally.

https://doi.org/10.1016/j.softx.2021.100971

being a key component in various domains such as climate modelling [1], patient monitoring [2], industrial maintenance [3], and decision-making in finance [4].

Two traditional steps in machine learning on time series are (pre)processing and feature extraction, often performed in this order. Processing is concerned with cleaning or transforming the raw data, e.g., filtering noise, detrending, clipping outliers, and resampling. Feature extraction aims to extract a set of characteristics, i.e., the features, with the intention of constructing a relevant (lower-dimensional) representation of the data. Both steps are time-consuming and rather complex, yet they are crucial for a successful machine learning pipeline [5].

In many cases the time series measurements might not necessarily be observed at a regular rate or could be unsynchronized [6]. Moreover, the presence or absence of measurements and the varying sampling rate may carry information on its own [7]. Unfortunately, current Python time series packages such as `seglearn` [8], `tsfresh` [9], `TSFEL` [10], and `kats` [11] make strong assumptions about the sampling rate regularity and the alignment of modalities. Furthermore, to the best of our knowledge, no library today supports multiple strided-window feature extraction, varying data types (e.g., handling categorical data), and chunking of (multiple) time series. These observations highlight the need for a flexible processing and feature extraction package. Therefore, we present `tsflex`, a package designed solely concerning these two steps, as it aims to get the fundamentals right. `tsflex` offers, next to custom functions, seamless integration with other data science packages, e.g., processing or feature functions from libraries such as `NumPy` [12], `SciPy` [13], `seglearn` [8], `tsfresh` [9], and `TSFEL` [10], or machine-learning toolkits like `scikit-learn` [14].

`tsflex` can be employed from prototyping machine learning pipelines to deploying real-world time series projects. Currently, we are amongst others using `tsflex` in real-time data pipelines for the *mBrain* study [15]. Here `tsflex` is used for processing and feature extraction of raw sensor data streams in which gaps, irregular sampling rates and large data chunks occur.

The remainder of this paper is as follows. In Section 2 we elaborate on the software and its functionality. Next on, Section 3 provides an illustrative example. Section 4 stresses the impact of `tsflex` by both positioning our toolkit among existing libraries and benchmarking these libraries against `tsflex`. Finally, we end with a conclusion in Section 5.

2. Software description

`tsflex` is a Python package that leverages (under the hood) efficient `NumPy` [12] data operations on `pandas` [16] data for (pre)processing and extracting features from time series. We opted for pandas data (either `pd.DataFrame` or `pd.Series`) since this is a convenient format for sequence data, and supports amongst others sequence indexing, integrated column names, and various data types. A direct result of complying with the available `pandas` data types is that `tsflex` allows performing operations on numerical, categorical, boolean, time based, and string-like data.

Users can install `tsflex` by using *pip*; `pip install tsflex`, or *conda*; `conda install -c conda-forge tsflex`. Once installed, our documentation together with various examples should enable the user to apply this toolkit for their purpose.

2.1. Software architecture

`tsflex` consists of two separated entities, i.e., a processing and a feature extraction submodule. The following subsections describe the architecture of both submodules, visually aided by Fig. 1.

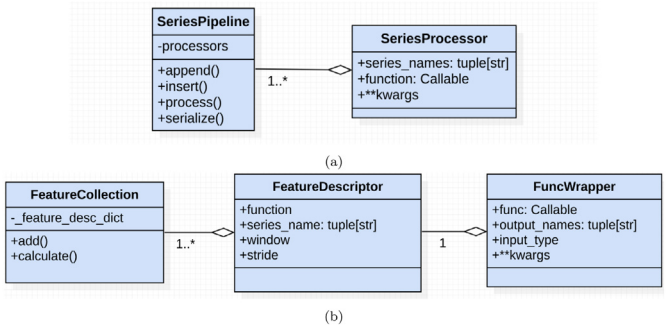

Fig. 1. UML diagram of the (a) `tsflex.processing` and (b) `tsflex.features` submodule.

Remark that these two submodules work on a different scope. The processing submodule works on full sequences, i.e., full scope, whereas the feature extraction submodule works on strided windows, i.e., restricted scope.

2.1.1. Processing submodule

Fig. 1(a) depicts the main components of the `tsflex.processing` submodule. The processing functionality is provided by the `SeriesPipeline` which contains one or multiple `SeriesProcessor` steps. The processing steps are applied sequentially on the data that is passed to the processing pipeline. This sequential order is crucial as the processing operations can create new series or update existing ones, which can be used in the succeeding steps, e.g., first applying a filter-processor and in the next steps decomposing that filtered signal. We summarize the objective of each component:

- `SeriesPipeline`: serves as a pipeline, withholding the to-be-sequentially-applied *processing steps*.
- `SeriesProcessor`: an instance of this class describes a *processing step*.
  A processing step is defined by a `function` (the *Callable* processing-function), `series_names` (the *name(s)* of the series that should be processed), and `**kwargs` (optional keyword arguments for `function`).

2.1.2. Feature extraction submodule

Fig. 1(b) depicts the `tsflex.features` components. The feature extraction functionality is provided by a `FeatureCollection` that contains one or multiple `FeatureDescriptors`. The features are calculated (possibly in parallel) on the data that is passed to the feature collection. We describe the objective of each component:

- `FeatureCollection`: serves as a registry, withholding the to-be-calculated *features*.
- `FeatureDescriptor`: an instance of this class describes a *feature*.
  A feature is defined by a `series_name` (the *name(s)* of the input series on which the feature-function will operate), `function` (the *Callable* feature-function), and `window` and `stride` (the sequence-index based window and stride range).
- `FuncWrapper`: a wrapper around *Callable* functions, intended for advanced feature-function configuration (e.g., customizing feature output-names, passing `**kwargs` to feature functions), and defining the function input data-type (i.e., `numpy.array` or `pandas.Series`).

2.2. Software functionalities

In the sections below, we further detail the processing and feature extraction functionalities, together with other utilities of `tsflex`.

2.2.1. Processing

The processing functionality is concerned with either transforming (i.e., replacing) sequences or creating new ones. `tsflex` provides flexible processing by accepting a generic processing function prototype. Such processing functions should take one or multiple sequences as input, followed by optional keyword arguments. This generic processing function prototype enables compatibility with many existing libraries, e.g., `scipy.signal` [13], `statsmodels.tsa` [17].[2]

2.2.2. Feature extraction

The feature extraction functionality is concerned with calculating features on strided-rolling windows. `tsflex` was designed to define the window and stride arguments in the same unit as the sequence-index its datatype (e.g., window=*"5min"* and stride=*"30s"* for time-indexed sequences, or window=*300* and stride=*30* for numeric-indexed sequences). As existing libraries define the window and stride in terms of number of samples [8–10], they implicitly assume that the sampling rate is fixed and there are no gaps. `tsflex`'s *flexibility* is a direct consequence of not making such assumptions; by default, features can be extracted on multivariate time series with varying sampling rates and even gaps.[3] In addition, `tsflex` supports a wide range of feature functions, again enabling compatibility with many existing libraries, e.g., `numpy`, `scipy.stats`, `tsfresh.feature_extraction`, `seglearn.feature_functions`, `tsfel.feature_extraction`.

2.2.3. Other functionalities

`tsflex` serves various additional functionalities, such as embedded serialization, execution time logging, native support for categorical and time based data, and handling of time series in chunks. Chunking of sequence data can be performed by calling the `chunk_data` function from the `tsflex.chunking` submodule. Processing and extracting features on chunked data produces lower memory peaks, enabling time series handling in constrained environments (e.g., streaming, edge devices [18]). Additionally, chunking allows parallelizing the sequential processing. Lastly, the `FeatureCollection` its `reduce(feat_cols_to_keep)` method returns a new `FeatureCollection` instance, withholding the subset of features that constitute the output column names listed in `feat_cols_to_keep`.

Unit tests. The provided functionalities of `tsflex` are extensively tested through unit testing. For example, these tests assure that the functions should perform view-based operations, that `tsflex` handles categorical and time-based data, and that feature-functions are not allowed to change the view-based input data. Every claim about `tsflex` we make in this paper is backed by unit testing.

[2] Processing functions can return an arbitrary amount of sequences; `tsflex` supports one-to-one, one-to-many, many-to-one, and many-to-many functions; see https://predict-idlab.github.io/tsflex/processing/index.html#versatile-processing-functions.

[3] It is the feature-function its responsibility to handle such cases correctly. Note that a feature-function can easily be made robust using the `make_robust` wrapper from `tsflex.features.utils`.

2.3. Limitations

Currently, there is no agreed standard for time series in Python [19]. The main cause for this disagreement is that each format has its own benefits and disadvantages. An in-depth discussion about this topic is out of scope for this paper. For `tsflex`, we made the design decision to operate on single-indexed wide/flat data (such as a list of series or a wide-dataframe) whose index represents the sequence-position. In our opinion, this data format is most intuitive to wrangle with, e.g., slicing, visualizing, processing. Therefore, two limitations of `tsflex` are that it (1) does not support long data, nor (2) multi-indexed data (and columns). Remark that a long-dataframe can be transformed into a list of series (that has the same in-memory size). A third limitation is that `tsflex` uses sequence-names as identifiers, resulting in the assumption that each sequence should have a unique name.

3. Illustrative examples

As illustrative example, we provide three snippets containing working code. An online version can be found in the examples folder of the tsflex repository.

Data loading. Listing 1 fetches the data for the examples. In total, three `pd.DataFrames` are loaded, containing multimodal data of different sampling rates. This data is an excerpt of a wrist-worn wearable from the WESAD study [20]. The characteristics of the dataframes are summarized in Table 1. The `df_tmp` dataframe withholds skin temperature data, `df_acc` withholds accelerometer data along the 3 movement axes, and `df_ibi` contains the Inter-Beat-Interval (IBI) data, representing the time between two consecutive heartbeats. Remark that IBI data is only available when two consecutive, successfully detected beats took place, making IBI an irregularly sampled series.

```
from tsflex.utils.data import load_empatica_data
df_tmp, df_acc, df_ibi = load_empatica_data(["tmp", "acc", "ibi"])
```

Listing 1: Data loading code for illustrative example.

Table 1
Properties of the data used in the illustrative example.

	columns	shape	sampling rate
df_tmp	[TMP]	(30200, 1)	4.0 Hz
df_acc	[ACC_x, ACC_y, ACC_z]	(241620, 3)	32.0 Hz
df_ibi	[IBI]	(1230, 1)	Irregularly sampled

3.1. Processing

Listing 2 shows how various processing steps are applied on the loaded data. For each processing step a `SeriesProcessor` object is created, which records the series names[4] (i.e., the names of the sequences that should be processed) and the optional keyword arguments. Observe that the `smv` function creates a new series.

3.2. Feature extraction

Listing 3 shows how feature extraction can be performed on the previously processed data. Two `MultipleFeatureDescriptors`[5] are created; the first defines some general statistical

[4] When a processing function should be applied on multiple series, a list should be passed to the `series_names` argument. When a processing function handles multiple series as input, a tuple (or a list thereof) should be passed to the `series_names` argument.

[5] `MultipleFeatureDescriptors` are a convenient way to define features containing multiple functions, series names, windows, and strides.

```
import pandas as pd; import numpy as np; from scipy.signal import
      savgol_filter
from tsflex.processing import SeriesProcessor, SeriesPipeline

# Create the processing functions
def clip_data(sig: pd.Series, min_val=None, max_val=None) -> np.ndarray:
    return np.clip(sig,  a_min=min_val, a_max=max_val)

def smv(*sigs) -> pd.Series:
    sig_prefixes = set(sig.name.split('_')[0] for sig in sigs)
    result = np.sqrt(np.sum([np.square(sig) for sig in sigs], axis=0))
    return pd.Series(result, index=sigs[0].index, name='|'.join(sig_prefixes
          )+'_'+'SMV')

# Create the series processors (with their keyword arguments)
tmp_clippper = SeriesProcessor(clip_data, series_names="TMP", max_val=35)
acc_savgol = SeriesProcessor(
    savgol_filter, ["ACC_x", "ACC_y", "ACC_z"], window_length=33, polyorder
          =2
)
acc_smv = SeriesProcessor(smv, ("ACC_x", "ACC_y", "ACC_z"))

# Create the series pipeline & process the data
series_pipe = SeriesPipeline([tmp_clippper, acc_savgol, acc_smv])
out_data = series_pipe.process([df_acc, df_tmp, df_ibi])
```

Listing 2: Processing example. Continuation of code snippet 1.

```
from tsflex.features import MultipleFeatureDescriptors, FeatureCollection
from tsflex.features.integrations import seglearn_feature_dict_wrapper
from tsflex.features.utils import make_robust

# Import / create the feature functions
from seglearn.feature_functions import base_features
def area(sig: np.ndarray): return np.sum(np.abs(sig))

# Create the feature descriptors
general_feats = MultipleFeatureDescriptors(
        functions=seglearn_feature_dict_wrapper(base_features()) + [area],
        series_names=["ACC_SMV", "TMP"],
        windows=["5min", "2.5min"], strides="2min",
)
ibi_feats = MultipleFeatureDescriptors(
    [make_robust(f) for f in [np.min, np.max, np.mean, np.std]] + [len],
    series_names="IBI", windows="5min", strides="2min"
)

# Create the feature collection & calculate the features
fc = FeatureCollection(feature_descriptors=[general_feats, ibi_feats])
feat_df = fc.calculate(out_data, return_df=True, approve_sparsity=True)
```

Listing 3: Feature extraction example. Continuation of code snippet 2.

and spectral features on the `ACC_SMV` and `TMP` signal for two different windows, and the second defines a robust version of some statistical features (and the number of samples) for the `IBI` signal. Remark that in `general_feats`, `seglearn` feature-functions are imported and wrapped in a convenient manner. These two descriptor objects are enclosed in a feature collection, which is used for extracting (i.e., calculating) the features. The `approve_sparsity` flag enables the user to explicitly acknowledge that there might be sparse data, i.e., irregularly sampled data. Setting this flag avoids warnings that are raised in case of sparsity.

4. Impact

We first indicate the impact of `tsflex` by positioning it among other packages. Then, we present `tsflex`'s performance in terms of memory usage and computation time, and compare these results to related packages. We conclude with some examples and references to notebooks which highlight the cross-domain applicability of `tsflex` for time series.

4.1. Functionalities

Irregularly sampled data is ubiquitous. However, most existing time series toolkits assume that either the user segments the data in valid chunks or that the data is regularly sampled. The former induces a significant user burden, whilst the latter is a fairly strong assumption. By employing a `sequence range based` window-stride approach and thus not a *sample based* one, `tsflex` interoperates natively with irregularly sampled sequence data. We position such functionalities of `tsflex` against other related packages in Table 2. Remark that `tsflex` is the only package that (1) allows defining multiple window-stride combinations, (2) can operate on non-numerical data, and (3) serves time-based chunking functionalities. Moreover, except for `tsfresh`, `tsflex` is the only other library that maintains the index of the data, encouraging index based analysis of the obtained outputs. We refer to example notebooks for more concrete illustrations of these functionalities.

4.2. Feature extraction performance

Considering all Python toolkits, eligible for strided-rolling feature extraction [8–10], only `seglearn` mentions toolkit-performance by comparing their computation time and model accuracy with other packages. However, for real-world applicability, computational efficiency is of utmost importance. Therefore, we benchmarked `tsflex` its memory usage and runtime against other libraries and open-sourced the benchmarking codebase at this repository[6] to encourage effortless benchmarking of `tsflex` on other use cases (e.g., edge devices, extremely large datasets, streaming use cases).

Profiling is realized by using the `VizTracer` [21] package with the `VizPlugins` add-on. The benchmark dataset is a synthetically generated dataframe consisting of 5 channels and spans 1 h. Its values have the numerical `numpy.float32` data type. To comply with the assumptions that other toolkits make, each modality is sampled at 1000 Hz and does not contain gaps. The toolkit are configured to extract the same features using a window-stride of 30 s-10 s, respectively. The benchmark process follows these steps for each toolkit-feature-extraction configuration:

1. Each toolkit feature extraction script is called 20 times to average out the memory usage and runtime.[7]
2. Script execution:
   (a) Construct the synthetic pd.DataFrame benchmark data
   (b) `VizTracer` starts logging
   (c) Create the feature extraction configuration
   (d) Extract and store the features
   (e) `VizTracer` stops logging
   (f) Write the `VizTracer` profile-results to a JSON-file

The profile JSONs were collected on a server with an *Intel Xeon E5-2650 v2 @ 2.60GHz* CPU and SAMSUNG M393B1G73QH0-CMA DDR3 1600MT/s RAM, with *Ubuntu 18.04.5 LTS x86_64* as operating system. Other running processes were limited to a minimum.

[6] We decided to only benchmark feature extraction, as this is the most advanced functionality of `tsflex`. In our experience, the processing functionality is rather straightforward and thus more dependent on the processing functions. However, also here, experimental results indicate that we have a significant efficiency advantage over other existing packages when parallel processing is performed on chunked data.

[7] Remark that by recalling the script in separate runs, no caching or memory is shared among executions.

Table 2
Comparison of `tsflex` against other relevant packages. The *"X-to-Y functions"* in the Properties column with $X, Y \in \{one, many\}$ represent the feature input-to-output relationship; X = "one" denotes single-series input, whereas X = "many" represents multivariate inputs. When Y = "one" a single feature is returned, whilst the Y = "many" returns multiple features. More info about these versatile functions can be found here. An online version of this table is shown here.

Properties	tsflex	seglearn	tsfresh	TSFEL	kats
General					
Time column requirements	Any - sortable	Any - sorted	Any - sortable	Any - sorted	Datetimeindex
Multivariate time series	✓	✓	✓	✓	✓
Unevenly sampled data	✓	✗	✗	✗	✓
Time column maintenance	✓	✗	✓	✗	✗
Retain output names	✓	✓	✓	✓	✗
Multiprocessing	✓	✗	✓	✓	✗
Operation execution time logging	✓	✗	✗	✗	✗
Chunking (multiple) time series	✓	✗	✗	✗	✗
Feature extraction					
Strided-window definition format	Sequence index range	Sample-based	Sample-based	Sample-based	Na.
Strided-window feature extraction	✓	✓	✓	✓	✗
Multiple stride-window combinations	✓	✗	✗	✗	✗
Custom features	✓	✓	✓	✓	✗
One-to-one functions	✓	✓	✓	✓	✓
One-to-many functions	✓	✓	✓	✓	✓
Many-to-one functions	✓	✓	✗	✗	✗
Many-to-many functions	✓	✗	✗	✗	✗
Categorical data	✓	✗	✗	✗	✗
Datatype preservation	✓	✗	✗	✗	✗

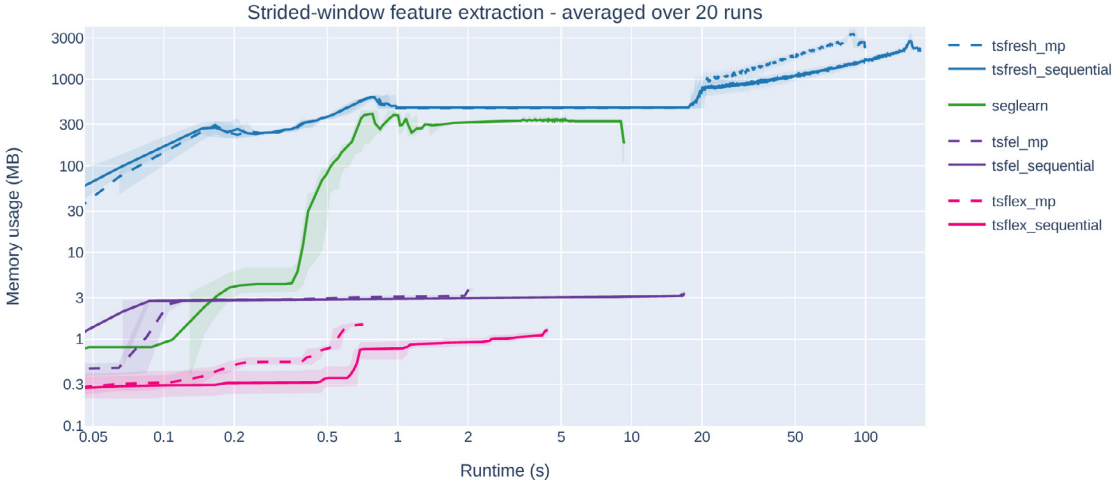

Fig. 2. Average memory usage over time for a feature extraction task on evenly sampled data with a fixed window and stride. The origin of the runtime and memory usage axis starts directly after the synthetic data was constructed; the feature extraction configuration is then initialized and called on the data. As noted in Table 2, only `seglearn v1.2.3` [8], `tsfresh v0.18.0,christ2018tsfresh`, and TSFEL `v0.1.4,barandas2020tsfel` support defining a (sample-based) window and stride, making this comparison fair as the data for this benchmark is evenly sampled. For reference, the allocated memory for the data was 96.4 MB. An interactive version (where you can switch to linear axes) of this figure is shown here.

Fig. 2 depicts the aggregated JSON-file results and Table 3 summarizes the main outcomes of this visualization. For this use-case, `tsflex` is $\sim$ 3$\times$ faster than its closest competitor in both the sequential and multiprocessing variant. The peak memory usage is of particular interest, as this determines the minimum amount of RAM a system should have. `tsflex` and TSFEL apply view-based operations on the data, making them significantly more memory efficient than other packages. Here again, `tsflex` requires $\sim$ 2.5$\times$ less memory than TSFEL. Note that `tsfresh` first expands the data into a tsfresh-compatible format before applying feature extraction. This results in a slope in the logarithmic domain from second 15 to second 80–150.

Table 3
Tabular summary of `VizTracer` benchmarks, depicted in Fig. 2.

	tsflex	TSFEL	seglearn	tsfresh
mean peak memory usage (**MB** $\pm$ **std**)				
sequential	**1.3** $\pm$ **0.1**	3.5 $\pm$ 0.3	435.3 $\pm$ 1.5	3540 $\pm$ 13.9
multiprocessing	**1.5** $\pm$ **0.1**	3.7 $\pm$ 0.1	/	4044 $\pm$ 14.4
mean runtime (**s** $\pm$ **std**)				
sequential	**4.3** $\pm$ **0.1**	16.4 $\pm$ 0.8	9.2 $\pm$ 0.1	169.8 $\pm$ 1.6
multiprocessing	**0.7** $\pm$ **0.0**	2.1 $\pm$ 0.0	/	98.5 $\pm$ 1.2

4.3. Applicability

`tsflex` is a domain-independent package, enabling broad applicability.[8] For example, this package is already used in multiple projects such as wearable-based stress monitoring, automatic sleep staging, occupancy detection in buildings, and anomaly detection. `tsflex`'s computational efficiency (in both execution time and memory usage) also paves the way towards applicability in constrained environments, such as streaming or edge computing [18].

[8] The cross-domain applicability is highlighted by the examples: https://github.com/predict-idlab/tsflex/tree/main/examples.

5. Conclusions

Time series processing and feature extraction are arguably the most important steps in classical machine learning pipelines. However, existing packages are limited in their applicability as they make strong assumptions about the underlying data types and data structure. Furthermore, these toolkits do not prioritize memory and runtime efficiency, creating unnecessary overheads. These existing packages also tend to focus on including numerous feature functions instead of conveniently integrating with other libraries. We argue that there is a need for a more permissive toolkit, which concentrates on the essentials. Therefore, we present `tsflex`, a Python package that focuses on processing and feature extraction for time series. This paper describes the functionalities and performance of `tsflex` and compares it to other packages. We show that `tsflex` is more permissive than existing Python toolkits, and benchmarking indicates it is over 50% more efficient than comparable work in both runtime and memory usage. The increased flexibility is realized by leveraging sequence-index based arguments and is reflected in the few assumptions that this library makes. We believe that `tsflex`'s integration with other libraries, together with its advanced functionalities, e.g., chunking, comprehensible feature output names, enables real-world, cross-domain applicability.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

Jonas Van Der Donckt and Emiel Deprost are funded by a doctoral fellowship of the Research Foundation – Flanders (FWO), Belgium. Part of this work is done in the scope of the imec.ICON COSMO (HBC.2018.0531), imec.AAA Context-aware health monitoring, and VLAIO PoC Nervocity.

References

[1] Pieters O, Deprost E, Van Der Donckt J, Brosens L, Sanczuk P, Vangansbeke P, et al. MIRRA: A modular and cost-effective microclimate monitoring system for real-time remote applications. Sensors 2021;21(13):4615.

[2] Topol EJ. High-performance medicine: the convergence of human and artificial intelligence. Nat Med 2019;25(1):44–56.

[3] Cook AA, Mısırlı G, Fan Z. Anomaly detection for IoT time-series data: A survey. IEEE Internet Things J 2019;7(7):6481–94.

[4] Taylor SJ. Modelling financial time series. World Scientific; 2008.

[5] Domingos P. A few useful things to know about machine learning. Commun ACM 2012;55(10):78–87.

[6] Yadav P, Steinbach M, Kumar V, Simon G. Mining electronic health records (EHRs) a survey. ACM Comput Surv 2018;50(6):1–40.

[7] Little RJ, Rubin DB. Statistical analysis with missing data, Vol. 793. John Wiley & Sons; 2019.

[8] Burns DM, Whyne CM. Seglearn: a python package for learning sequences and time series. J Mach Learn Res 2018;19(1):3238–44.

[9] Christ M, Braun N, Neuffer J, Kempa-Liehr AW. Time series feature extraction on basis of scalable hypothesis tests (tsfresh–a python package). Neurocomputing 2018;307:72–7.

[10] Barandas M, Folgado D, Fernandes L, Santos S, Abreu M, Bota P, et al. Tsfel: Time series feature extraction library. SoftwareX 2020;11:100456.

[11] Kats - one stop schop for time series analysis in Python. 2021, URL https://facebookresearch.github.io/Kats/.

[12] Harris CR, Millman KJ, van der Walt SJ, Gommers R, Virtanen P, Cournapeau D, et al. Array programming with NumPy. Nature 2020;585(7825):357–62. http://dx.doi.org/10.1038/s41586-020-2649-2.

[13] Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, et al. SciPy 1.0: Fundamental algorithms for scientific computing in python. Nature Methods 2020;17:261–72. http://dx.doi.org/10.1038/s41592-019-0686-2.

[14] Buitinck L, Louppe G, Blondel M, Pedregosa F, Mueller A, Grisel O, et al. API design for machine learning software: experiences from the scikit-learn project. In: ECML PKDD workshop: Languages for data mining and machine learning. 2013, p. 108–22.

[15] Brouwer MD, Vandenbussche N, Steenwinckel B, Stojchevska M, Donckt JVD, Degraeve V, et al. MBrain: towards the continuous follow-up & headache classification of primary headache disorder patients. BMC Med Inf Decis Making 2021.

[16] pandas development team T. Pandas-dev/pandas: Pandas. 2020, http://dx.doi.org/10.5281/zenodo.3509134.

[17] Seabold S, Perktold J. Statsmodels: Econometric and statistical modeling with python. In: Proceedings of the 9th python in science Conference, Vol. 57. Austin, TX; 2010, p. 61.

[18] Shi W, Dustdar S. The promise of edge computing. Computer 2016;49(5):78–81.

[19] Christ M. Awesome time series in python - standardize time series formats. 2020, URL https://github.com/MaxBenChrist/awesome_time_series_in_python/blob/master/standardize_time_series_formats.md.

[20] Schmidt P, Reiss A, Duerichen R, Marberger C, Van Laerhoven K. Introducing WESAD, a multimodal dataset for wearable stress and affect detection. In: Proceedings of the 20th ACM international conference on multimodal interaction. 2018, p. 400–8.

[21] Gao T. Viztracer: a low-overhead logging, debugging, and profiling tool to trace and visualize python code execution.. 2020, URL https://github.com/gaogaotiantian/viztracer/.