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

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.

© 2022 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Motivation and significance

Data-driven modelling and forecasting of time series is a major topic of interest in academic research and industrial applications,
being a key component in various domains such as climate modeling [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.

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., scikit-learn [13], statsmodels.tsa [17], scipy.signal [13], and pandas [18], which were used as the building blocks for tsflex.

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 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. 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, and 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’s 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.

2.3. 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.DataFrame objects 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>
<thead>
<tr>
<th>Table 1</th>
<th>Properties of the data used in the illustrative example.</th>
</tr>
</thead>
<tbody>
<tr>
<td>columns</td>
<td>shape</td>
</tr>
<tr>
<td>df_tmp</td>
<td>[TMP]</td>
</tr>
<tr>
<td>df_acc</td>
<td>[ACC_x, ACC_y, ACC_z]</td>
</tr>
<tr>
<td>df_ibi</td>
<td>[IBI]</td>
</tr>
</tbody>
</table>

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 (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 are created; the first defines some general statistical

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’s 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.

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.
and spectral features on the ACC_SHV and TMP signal for two different windows, and the second defines a robust version of its sparsity. Remark that in different windows, and these second defines a robust version of its sparsity.

Jonas Van Der Donckt, Jeroen Van Der Donckt, Emiel Deprost et al. SoftwareX 17 (2022) 100971

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 repository6 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 is 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 ∈ \{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.

<table>
<thead>
<tr>
<th>Properties</th>
<th>tsflex</th>
<th>seglearn</th>
<th>tsfresh</th>
<th>TSFEL</th>
<th>kats</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time column requirements</td>
<td>Any - sortable</td>
<td>Any - sorted</td>
<td>Any - sortable</td>
<td>Any - sorted</td>
<td>DatetimeIndex</td>
</tr>
<tr>
<td>Multivariate time series</td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unevenly sampled data</td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time column maintenance</td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Retain output names</td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiprocessing</td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operation execution time logging</td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chunking (multiple) time series</td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feature extraction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strided-window definition format</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strided-window feature extraction</td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple stride-window combinations</td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Custom features</td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>One-to-one functions</td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>One-to-many functions</td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Many-to-one functions</td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Many-to-many functions</td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Categorical data</td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Datatype preservation</td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

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

<table>
<thead>
<tr>
<th></th>
<th>tsflex</th>
<th>TSFEL</th>
<th>seglearn</th>
<th>tsfresh</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean peak memory usage (MB ± std)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sequential</td>
<td>1.3 ± 0.1</td>
<td>3.5 ± 0.3</td>
<td>435.3 ± 1.5</td>
<td>3540 ± 13.9</td>
</tr>
<tr>
<td>multiprocessing</td>
<td>1.5 ± 0.1</td>
<td>3.7 ± 0.1</td>
<td>/</td>
<td>4044 ± 14.4</td>
</tr>
<tr>
<td>mean runtime (s ± std)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sequential</td>
<td>4.3 ± 0.1</td>
<td>16.4 ± 0.8</td>
<td>9.2 ± 0.1</td>
<td>169.8 ± 1.6</td>
</tr>
<tr>
<td>multiprocessing</td>
<td>0.7 ± 0.0</td>
<td>2.1 ± 0.0</td>
<td>/</td>
<td>98.5 ± 1.2</td>
</tr>
</tbody>
</table>

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


