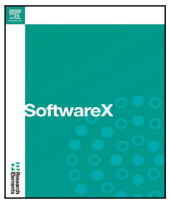
SoftwareX 29 (2025) 102036

Contents lists available at ScienceDirect

SoftwareX

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



Original software publication

Timeseria: An object-oriented time series processing library

Stefano Alberto Russo [a,b,c,*], Giuliano Taffoni [a,b], Luca Bortolussi [c]

[a] Italian National Center For HPC, Big Data and Quantum Computing, Bologna, Italy
[b] INAF - Italian National Institute for Astrophysics - Observatory of Trieste, Italy
[c] University of Trieste - Department of Mathematics, Informatics and Geosciences, Trieste, Italy

ARTICLE INFO

Keywords:
Python
Time series
Data structures
Forecasting
Reconstruction
Anomaly detection

ABSTRACT

Timeseria is an object-oriented time series processing library implemented in Python, which aims at making it easier to manipulate time series data and to build statistical and machine learning models on top of it. Unlike common data analysis frameworks, it builds up from well defined and reusable logical units (objects), which can be easily combined together in order to ensure a high level of consistency. Thanks to this approach, Timeseria can address by design several non-trivial issues which are often underestimated, such as handling data losses, non-uniform sampling rates, differences between aggregated data and punctual observations, time zones, daylight saving times, and more. Timeseria comes with a comprehensive set of base data structures, data transformations for resampling and aggregation, common data manipulation operations, and extensible models for data reconstruction, forecasting and anomaly detection. It also integrates a fully featured, interactive plotting engine capable of handling even millions of data points.

Code metadata

Current code version	v2.0.2
Permanent link to code repository for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-24-00430
Permanent link to reproducible capsule	https://codeocean.com/capsule/94157280-4a93-4fb0-8ddf-d909557d3309
Code license	Apache Licence v2
Code versioning system	Git
Code languages	Python, JavaScript
Operating environments & dependencies	Required environment: Python >= 3.6 Optional environments: Jupyter Lab >=3, Jupyter Notebook >=5 Required Python dependencies: matplotlib >=2.1.2, <4.0.0, numpy >=1.19.5, <2.0.0, scikit-learn >=0.2.2, <2.0.0, pandas >=0.23.4, <2.0.0, chardet >=3.0.4, <4.0.0, convertdate >=2.1.2, <3.0.0, lunarcalendar >=0.0.9, <1.0.0, cython >=0.29.17, <1.0.0, requests >=2.20.0, <3.0.0, h5py >=2.10.0, <4.0.0, scipy >=1.5.4, <2.0.0, pyppeteer>=0.2.6, <1.0.0, fitter==1.7.0, propertime>=1.0.1, <2.0.0 Optional Python dependencies: tensorflow >=1.15.2, <3.0.0, prophet: >=1.1.1, <2.0.0, pmdarima >=1.8, <2.0.0, statsmodels >=0.12.1, <1.0.0 Optional system dependencies: Chrome or Chromium >= 57 for image-based plots (automatically downloaded if not present)
Documentation	https://timeseria.readthedocs.io/en/v2.0.2
Support for questions	https://github.com/sarusso/Timeseria/issues

* Correspondence to: Italian National Center For HPC, Big Data and Quantum Computing, Italy.
E-mail address: stefano.russo@gmail.com (Stefano Alberto Russo).

https://doi.org/10.1016/j.softx.2025.102036
Received 13 August 2024; Received in revised form 7 December 2024; Accepted 3 January 2025
Available online 27 January 2025

1. Motivation and significance

Time series are central to several fields across many disciplines, such as engineering, meteorology, medicine, environmental science, finance, economics, and more. Time series represent the evolution of a phenomena over time, and their analysis is essential to capture the dynamics of the phenomena being studied, understand cause-and-effect relationships, and make predictions.

However, a typical time series processing pipeline — loading a data set, cleaning and plotting it, performing some operations, applying some models and inspecting the results — still feels unnecessarily cumbersome. Scientists, engineers, analysts, and many other professional figures spend a considerable amount of time on repetitive procedures and on getting their code to work, instead of focusing on their core tasks.

Most of current numerical and data analysis frameworks indeed fall short in handling everyday practical challenges when it comes to time series data, and perhaps most importantly, in real-world scenarios. For example, in a recent review [1] that surveyed 40 time series processing packages, within the 17 listed as capable of handling missing data we found that none of them provided a robust solution, but more of a set of workarounds. Similarly, among the packages listed as capable of performing modeling tasks, checking for data consistency was always left to the user.

We also noted a generalized lack of definitions about what constitutes time series data, and how to represent it [2]. This is partly because each format has its own advantages and disadvantages, and partly because the complexity of such data is often underestimated. Simple vector or matrix data structures are usually the default choice, until discovering at later stages that are just not capable of taking into account most of the typical time series characteristics: data losses are represented ambiguously, calendar arithmetic and daylight saving time are handled poorly, there is no support for variable sampling rates, and there is no clear distinction between punctual observations and aggregated data.

In this context, the usual solution to overcome such limitations consists of a series of patches pieced together which eventually backfire, and very few attempts have been made to address the fundamentals properly. The TSflex [3] package, which focuses on time-series pre-processing and feature extraction, is one of such rare examples.

To the best of our knowledge, as of today there is neither research work nor software packages aimed at addressing such issues at their roots. In this work we presents a novel, object-oriented approach for representing and processing time series data, which we believe can address most of the issues outlined above, together with the software library implementing it: *Timeseria*.

2. Related work and comparative analysis

Within the large body of work on libraries, tools, and frameworks for data processing suitable for time series data, we selected the most relevant ones with respect to Timeseria. The key criterion was their usability as building blocks (i.e., as a library), which excluded, among others, GUI (Graphical User Interface) applications, web-based solutions, and closed-source software.

One of such examples is Wolfram Mathematica, a commercial (and closed-source) software solution originally developed for symbolic computation, which provides several functionalities for time series data. Although these functionalities could, in principle, be compared to Timeseria's, the comparison would be challenging on both sides, given their different nature, goals, and target audiences, as discussed in a GitHub issue opened on Timeseria's repository[1]

[1] https://github.com/sarusso/Timeseria/issues/40

Also the R [4] software, which is instead open source, provides functionalities for time series processing that could be compared to Timeseria's. In particular, R provides the `ts` function, which allows to create and manipulate time series objects. In this case as well, a direct comparison would be difficult, in particular because R is designed for interactive analysis and data visualization, lacking the general-purpose features and ecosystem support provided by other languages, which severely limits its usability as a building block for other software.

In addition, several domain-specific tools are available, often provided in the form of GUI or Web-based applications. Some examples include MaD GUI [5] (a tool for creating GUI interfaces for domain-expert time series annotation), GRETA [6] (a web-based solution that produces hourly wind and solar photovoltaic generation time series), and OXI [7] (an online tool for visualization and annotation of satellite time series data). While such tools can undoubtedly outperform more general-purpose solutions (like Timeseria) for the use case or domain they were originally developed for, they are deeply intertwined with their original purpose, making them difficult to adapt, even if heavily modifying the code.

With respect to the solutions that can instead be directly compared to Timeseria, and in particular in the realm of the data structures, Pandas [8] `Series` and `DataFrames` are the most commonly used for representing time series data in the Python ecosystem. They provide basic support for time series data through the `DatetimeIndex`, which enables operations and slicing based on time. Resampling functionalities are also supported, with basic handling of data losses, provided that they have been already marked as null values. Xarray [9] is another widely used option in the Python ecosystem, which provides data structures for n-dimensional labeled arrays. Time series data can be represented using the `DataArray` class, which features interfaces similar to Pandas `Series` or `DataFrames` but is optimized for multi-dimensional data and large datasets.

Timeseria provides instead its own data structures for time series data, which are built mainly from scratch. This design choice was necessary in order to achieve the consistency and abstraction levels Timeseria aimed for. Timeseria data structures are indeed not optimized for performance, but for clarity of abstractions and consistency. This approach made it possible to differentiate between punctual observations and aggregated data, to integrate handling of data losses into the library's foundations, to manage time nuances as time zones, daylight saving time, and calendar time effectively, and more.

Higher-level solutions are built almost exclusively on top of Pandas or Xarray data structures, and thus lack the consistency and abstraction levels that Timeseria can provide. In the following we report the most notable ones, which typically focus on modeling aspects.

SKtime [10] and Tslearn [11] are general-purpose Python libraries for a variety of time series machine learning tasks, offering a combination of tools for pre-processing, feature extraction, clustering, classification, regression and forecasting. TSflex [3] is a Python toolkit for flexible time series processing and feature extraction, claimed to be particularly efficient and making few assumptions about the underlying data. As mentioned earlier, it is one of the few examples that aim to address the fundamentals properly, albeit with a narrow focus. GluonTS [12] is a Python package for probabilistic time series modeling focusing on deep learning models, and AutoTS [13] is a package designed for rapidly deploying high-accuracy forecasts at scale, testing several forecasting models using a genetic approach, and finding the best candidates autonomously. Prophet [14] is instead defined as a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, along with holiday effects, which is implemented in both R and Python. Greykite [15] aims to provides flexible, intuitive and fast forecasts through its flagship algorithm, Silverkite, which is particularly indicated for time series with change points in trend or seasonality, event/holiday effects, and temporal dependencies. PyOD [16], is a comprehensive Python library for detecting outliers and/or anomalies

Table 1
Comparison of Timeseria with similar solutions (Darts, Kats, ETNA). Notes: [1] such methods are intended as functions that perform their task in just one call, without requiring any extra code; [2] by "standard" performance it has to be intended the performance that can be obtained when relying on Pandas or Xarray data structures.

	Timeseria	Darts	Kats	ETNA
Dedicated, object-oriented base data structures	yes	no	no	no
Differentiation between aggregated and punctual observations	yes	no	no	no
Extensive time zone and calendar time support	yes	no	no	no
Extensive support for data losses	yes, with auto-detect	no	no	no
Dedicated, fully featured interactive plotting engine	yes	no	no	no
Forecasting	yes	yes	yes	yes
Reconstruction	yes	no	no	no
Anomaly detection	only model-based	yes, with several techniques	only as outliers	only as outliers
Model apply method[1]	yes, for all models	no	no	no
Model evaluation and cross validation methods[1]	yes	only as utility functions for the residuals	no	yes, as pipeline back testing functionality
Support for custom models	yes, with several functionalities	limited, no significant functionalities	limited, no significant functionalities	yes, with several functionalities
Built-in models	few, mainly as examples	several	several	several
Probabilistic support	under development	yes	only for forecasts	only for forecasts
Computing performance[1]	lower than standard	standard	standard	standard

in multivariate data, while Orion [17] is a machine learning library built for unsupervised time series anomaly detection, mainly using Generative Adversarial Networks, with the goal of identifying rare patterns and flag them for expert review.

With respect to time series modeling, Timeseria focuses less on providing a wide selection of models, and more on providing well-defined building blocks that allow users to plug-in their own modeling logic with minimal effort, possibly integrating with all of the above solutions (Prophet is indeed used as an example built-in model). The goal of this choice is to bring together the best features of the aforementioned packages with the unique strengths of Timeseria.

Another class of related work are libraries and frameworks targeting the entire data analyzes pipeline, which are generally (if not entirely) mutually exclusive with Timeseria. We identified three of them as the most relevant ones: Darts, Kats, and ETNA. Darts [18] is a Python machine learning library for time series, with a focus on forecasting. It offers a variety of models, from classics such as ARIMA to state-of-the-art deep neural networks. It provides a dedicated main class `TimeSeries` which stores data as a `DataArray` (from the Xarray package). Kats [19] is a light-weight, easy-to-use, extendable, and generalizable framework to perform time series analysis in Python. It aims to provide a one-stop shop for techniques for univariate and multivariate time series including forecasting, anomaly and change point detection, feature extraction, and more. Kats provides a dedicated `TimeSeriesData` structure to represented univariate and multivariate time series, which is based on a tuple of Pandas `Series`, one for the timestamps and one for the values. ETNA [20] is a time series forecasting framework which aims at ease of use. It includes functionalities for time series pre-processing, feature generation, several forecasting models with a unified interface, model combinations and evaluation through back-testing. It makes use of a `TSDataset` format for representing time series data, which is a wrapper around a Pandas `DataFrame`, offering several functionalities for time series data.

While all of the work presented in this section could be compared to Timeseria to some extent, only Darts, Kats and ETNA show significant overlap and can thus be compared in a direct way. We therefore conducted a comparative analysis to highlight their respective differences, with particular attention to Timeseria unique functionalities but also from other angles, that is presented in Table 1.

3. Software description

Timeseria presents itself as a Python library that can be imported and used in interactive data analysis environments (as Jupyter), batch data analysis pipelines, and more structured projects. It supports both univariate and multivariate time series, and provides functionalities for loading, storing, manipulating and plotting the data, as well as fitting, evaluating and applying models.

3.1. Software architecture

The main modules of Timeseria are the *Datastructures*, the *Operations*, the *Transformations* and the *Models*, which are schematized in Fig. 1 together with their main classes, and described in the following.

3.1.1. Data structures

The Datastructures module provides all the base data structures: `Points`, `Slots` and `Series`, together with their specializations.

A `Point` is just a point in any n-dimensional space, which supports common mathematical operations. A `TimePoint` is a point in the time dimension, measured in seconds, where the zero (epoch) corresponds to the midnight of 1st January 1970, UTC (Coordinated Universal Time). Sub-second precision can be achieved with decimals. A `DataPoint` is an extension of the point concept, with some data attached. Such data can be of any kind, however many functionless of Timeseria

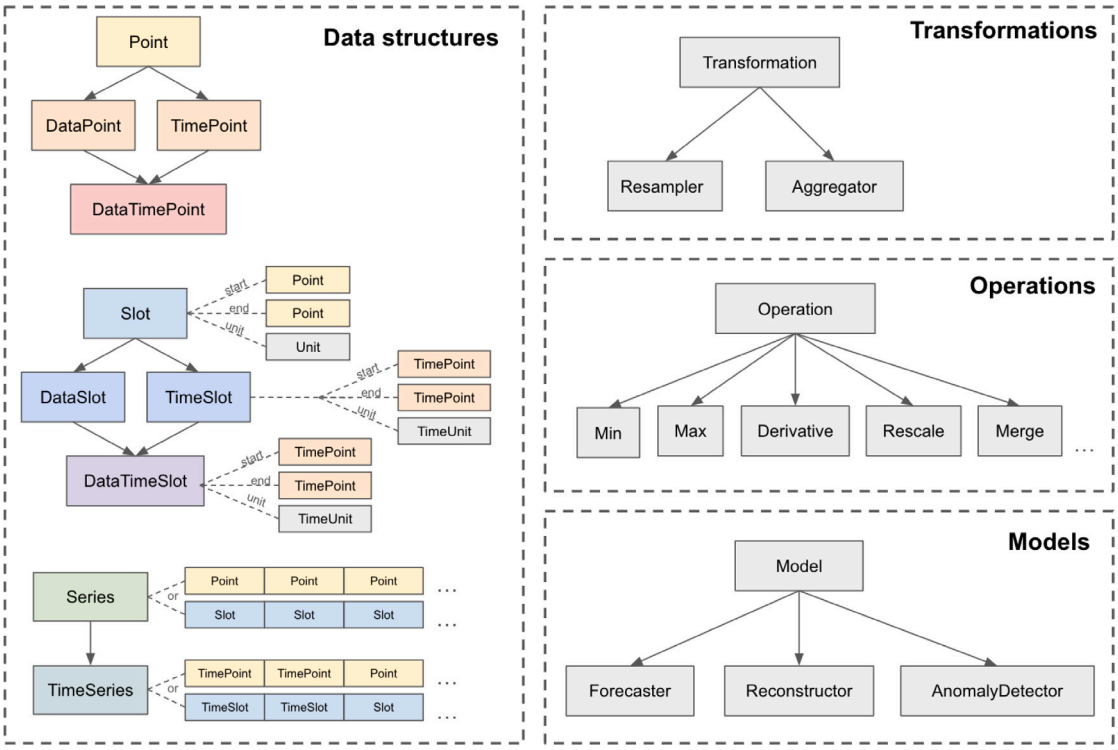


Fig. 1. Timeseria base classes structure.

require indexed or key–value data. To define data points over time, the `DataPoint` and `TimePoint` classes are merged, creating the `DataTimePoint`.

Since points are intended to represent punctual observations, to represent aggregated data Timeseria defines a new data structure: the `Slots`. These are "containers" that span from a *start* to an *end* (which are again `Point` objects), and that are naturally associated with a `Unit`, a dedicated object that represents their span. Similarly to the points, also for the slots Timeseria defines the `DataSlots`, `TimeSlots` and the `DataTimeSlots`.

In `TimeSlots` and `DataTimeSlots`, the unit becomes a `TimeUnit`, which implements one of the most distinctive design principles of Timeseria: to allow for time units of *variable* length. This allows to effectively model years and months, which can have a variable number of days, and days themselves, which can have a variable number of hours (due to the daylight saving time). It does indeed *not* hold true that a time unit of 24 h ("24h") is equal to a time unit of one day ("1D") : the first is always exactly 24 h, while the second can assume different lengths: 24, 24 or 25 h, depending on whether there is a daylight saving time switch, and in which direction.

Both data points and slots support a `data_loss` attribute, which is a key feature in Timeseria. This represents the data loss that can occur when computing a new point during resampling (e.g. when its nearest neighbors are too far), or when computing a new slot during aggregation (e.g. due to missing punctual observations).

The data loss is a particular case of a more generic concept: the *data indexes*. These are indicators in the 0 - 1 range of some property of the data attached to a data point or slot, and accessible with the `data_indexes` attribute. Example data indexes include, besides the data loss, the *data reconstructed* index, the *forecast* index, and the *anomaly* index.

`Series` are defined as a list of items coming one after another, where every item is guaranteed to be of the same type and following an order or a succession. Ordering is checked with the standard comparison operators, while it is up to the items of the series to define a succession logic, if any. In Timeseria, points only implement an ordering, while slots also define a succession: in other words, if a series is slotted, it must be dense, with no gaps.

Series can carry virtually any data type, but Timeseria focuses on the time domain and thus defines a dedicated specialization: the `TimeSeries`. This is a collection, again in order or succession, of points or slots over time (i.e. `DataTimePoints` or `DataTimeSlots`). Time series also have a `resolution` attribute, which represents their temporal resolution, and that is rendered using a `TimeUnit`. This concept is similar to a sampling interval, but generalized in order to make it compatible with the slots as well. For example, a point time series sampled at 1-minute intervals has a resolution corresponding to the time unit "60s", while a one-day slots time series has a resolution corresponding to the time unit "1D". Time series with a variable sampling rate report instead "variable" as their resolution.

3.1.2. Transformations

Transformations are meant to change the temporal resolution of a time series, and to modify its data accordingly. The base `Transformation` class provides a generic transformation concept, which is then specialized into a `Resampler` and an `Aggregator`.

A `Resampler` transforms a point series in another point series with a different sampling interval, while an `Aggregator` transforms a point series in a slot series, aggregating its data in slots of a given unit. The main difference between the two is that, while a resampling process is supposed to produce an alternative "view" of the original data, following the Nyquist–Shannon theorem, an aggregation is instead intended to compute statistical indicators, the default one being the average. Both transformations also detect missing data, impute it using a given interpolation logic while setting the `data_loss` attribute accordingly, and recompute the data indexes in order to bring them forward.

3.1.3. Operations

Operations are meant to manipulate the data of a series, and can return other series, a scalar, a list of items, or any other valid data type. An operation, if returning a series, it never changes its temporal

resolution, which is a task reserved for the transformations. Timeseria provides several built-in operations, as computing the average, minimum and maximum values, merging two series, applying an offset, normalizing the data, and more. Custom operations can be defined as well.

3.1.4. Models

Timeseria defines a base `Model` class which can represent different kind of models. At a higher level, it differentiates between parametric and non parametric models. The first have all of the modeling information coded in the model itself, while the second foresees a (potentially very large) number of parameters that can be either set manually or learned from the data.

All models expose a `predict()` and an `apply()` method. Parametric models also provide `save()` and `load()` methods to store and load their parameters, and an optional `fit()` method if such parameters are to be learned from the data. For models supporting the evaluation, `evaluate()` and `cross_validate()` methods are available as well.

Models are further sub-divided in three main categories: *forecasters*, *reconstructors*, and *anomaly detectors*.

- **Forecasters** are designed to make predictions for n steps ahead, either using their internal logic to predict all of the steps in one go, or by recursively re-applying them n times on previously predicted data. Forecasters can operate with or without an input window, and edge cases (as applying a forecaster on a time series without enough data when using a window) are automatically handled by the base `Forecaster` class. The `apply()` method performs the forecast for a given time series and automatically adds to the tail, marking it with the `forecasted` data index.
- **Reconstructors** are intended to reconstructing missing data, or in other words to fill *gaps*. Gaps need a "next" element to be defined, and are detected using the `data_loss` attribute. By default, only gaps with a full data loss are reconstructed. Reconstructors can operate with or without an input window, on either sides of the gap. Edge cases (as applying a reconstructor on a time series without enough data before or after a gap) are automatically handled by the base `Reconstructor` class, whose `apply()` method reconstructs the missing data of a given time series marking it with the `reconstructed` data index.
- **Anomaly detectors** aim at spotting anomalies in the data. There are several ways to achieve this goal, and the topic is particularly complex and somewhat controversial [21], given that anomalies are basically ill-defined and that multiple ground truths can co-exists at the same time. A structured exposition of the topic is therefore beyond the scope of this article. As of today, Timeseria implements only *model-based* anomaly detection [22], which mainly delegates the detection capabilities to an underlying model. In this mode, anomalies are detected by evaluating the error between the actual values and the values predicted by the model: if this is "too big", then it is considered as a symptom of anomaly. How to define when an error is "too big" is part of the complexity of the topic, and to avoid making strong assumptions in this regard, Timeseria provides an *anomaly index* rather than a score or a binary classification. Such index is a floating point number in the 0-1 range representing the likelihood of a point or slot to be anomalous, computed taking into account the overall error distribution of the model together with some probabilistic considerations, and following either an unsupervised or semi-supervised logic. If required, a scoring or thresholding mechanism can be then applied on top of the index, ex-post. More details about how the anomaly index is computed can be found in the reference documentation [23]. Besides the generic base `AnomalyDetector` class, Timeseria provides the `ModelBasedAnomalyDetector` class, whose `fit()` and `apply()` methods implement all of the logic to compute and assign the above introduced anomaly index (as the `anomaly` data index), for each data point or slot of a given time series. Any model sub-classing the base `Model` class can then be used as its engine, and in particular any `Forecaster` or `Reconstructor`, including custom ones.

3.2. Software functionalities

As previously mentioned, Timeseria provides functionalities for loading, storing, manipulating, and plotting time series data, as well for fitting, evaluating and applying models. In the following, we provide an overview of the main ones, along with their respective methods and classes.

3.2.1. Loading and storing data

Timeseria supports loading and storing data directly from the `TimeSeries` class. The `from_csv()` method allows to load CSV (Comma-Separated Values) files with automatic encoding and format detection, and supports many parameters if auto-detection fails. For example, field and newline separators, column mapping, and timestamp formats can all be set, as well as many others. Time series can be saved either as standard CSV files using the `to_csv()` method, or using a dedicated format with the `save()` method, that includes some metadata and that fully supports the data indexes. Such format can then be loaded using the `load()` method, and is particularly convenient for storing and loading time series within Timeseria. Lastly, Pandas DataFrames are also supported, using the `to_df()` and `from_df()` methods.

3.2.2. Manipulating and inspecting data

Data can be manipulated using the transformations and the operations, which can be accessed directly as methods of the `TimeSeries` class. Besides the two main `resample()` and `aggregate()` transformations, Timeseria provides a number of built-in operations which include: `min()`, `max()`, `avg()`, `sum()`, `derivative()`, `integral()`, `diff()`, `csum()` (for the cumulative sum), `mavg()` (for the moving average), `normalize()`, `offset()` and `rescale()`, which all operate on the data values, plus the `filter()` operation to filter by label, the `slice()` operation to cut a portion of a time series and the `merge()` operation to merge two or more time series.

The `TimeSeries` class also supports the square brackets notation for quickly accessing its elements by position (e.g. `timeseries[2]`), timestamp (e.g. `timeseries[datetime(2015,10,25,6,0,0)]`), and for filtering on a specific data label if using key–value data (e.g. `timeseries["temperature"]`). Also the standard Python slicing notation is supported, either by position or timestamp.

Lastly, plotting a time series can be achieved by simply invoking the `plot()` method of the `TimeSeries` class, which generates interactive plots using a hybrid Python-JavaScript engine based on Dygraphs [24]. Such plots fully support Timeseria data structures, including the data indexes, and can be saved both as images and self-contained HTML pages. Above a given amount of data, the plotting engine automatically performs an aggregation step, in order to speed up the plotting process and to prevent overwhelming interactive analysis environments (and Web browsers) with too much data. In this case, the plot is rendered as a line chart overlying an area chart which represents the dispersion of the original data, thus allowing to retain visual information about peaks that would otherwise get smoothed.

3.2.3. Fitting, evaluating and applying models

Timeseria provides a set of built-in models to be intended primarily for demonstrative purposes and simple tasks, rather than for real-world applications. They include: the `PeriodicAverageForecaster`, the `PeriodicAverageReconstructor` and the `PeriodicAverageAnomalyDetector`, which are all based on computing periodic averages over historical data; the `AARIMAForecaster` and the `AARIMAReconstructor`, based on Automatic Auto-Regressive Integrated Moving Averages; and the `LSTMForecaster` together with the `LSTMAnomalyDetector`, which make use of a LSTM (Long-Short Term Memory) neural network.

The true goal of Timeseria is instead to provide a framework where to easily plug-in custom modeling logic by extending the base classes. As of today, such classes include the `Forecaster`, the `Reconstructor`, and the `ModelBasedAnomalyDetector`. When extending base classes, a set of methods and decorators is automatically inherited, so that only model-specific logic needs to be coded. For example, the base `Forecaster` class provides the `apply()`, `evaluate()` and `cross_validate()` methods, together with two dedicated decorators: the `@Forecaster.fit_method` and the `@Forecaster.predict_method`, to be used to decorate the custom `fit()` and `predict()`. All common functionalities, including the `save()` and `load()` methods, as well as the data consistency checks, are then automatically inherited.

4. Illustrative examples

The following examples can be reproduced with a clean Timeseria installation either in a Jupyter environment or with a standard Python script. In this case, plots must be generated as images and saved to file, adding the following extra arguments to each `plot()` call: `image=True, save_to="plot.png"`.

The dataset used in these illustrative examples is an indoor air temperature and humidity dataset, sampled at about ten minutes intervals, which except from its size presents some of the most common challenges in time series processing. These include: a variable sampling rate, several data losses, and the dependency on "human" time through the time zone and the daylight saving time.

4.1. Load and inspect the data

The data is loaded as the `TimeSeries` object, and timestamps are assumed on UTC, since not otherwise specified. The time series is then printed to quickly inspect it.

```
from timeseria import TEST_DATASETS_PATH as PATH
from timeseria.datastructures import TimeSeries

timeseries = TimeSeries.from_csv(PATH + "humitemp.csv")
print(timeseries)
```

```
Time series of #14000 points at variable resolution (~615s), from point @
1546475294.0 (2019- 01-03 00:28:14 +00:00) to point @ 1555544819.0 (2019
-04-17 23:46:59 +00:00)
```

The average values of temperature and humidity can be obtained using the `avg()` operation:

```
print(timeseries.avg())
```

```
{'humidity[RH]': 43.870, 'temperature[C]': 22.488}
```

The time series can be plotted with the built-in plotting engine (Fig. 2), which will automatically aggregate it by a factor of ten to speed up the process:

```
timeseries.plot()
```

```
[INFO] Aggregating by "10" for improved plotting
```

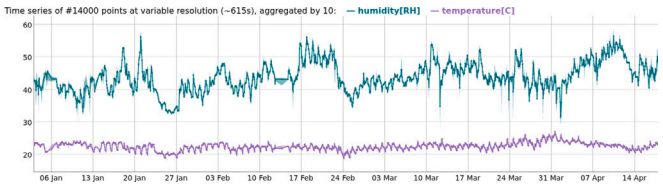


Fig. 2. The time series plotted with an automatic aggregation factor of ten. The lighter shadow underlying the line chart represents the dispersion of the original data.

4.2. Resample to hourly data

The time series is resampled with a sampling interval of one hour. Gaps are filled by linear interpolation and the `data_loss` index is added. Then, the time series in printed and plotted (Fig. 3).

```
timeseries = timeseries.resample("1h")
print(timeseries)
timeseries.plot()
```

```
[INFO] Using auto-detected sampling interval: 615.0s
[INFO] Resampled 14000 DataTimePoints in 2519 DataTimePoints

Time series of #2519 points at 1h resolution, from point @ 1546477200.0 (2019
-01-03 01:00:00 +00:00) to point @ 1555542000.0 (2019-04-17 23:00:00 +00:00)
```

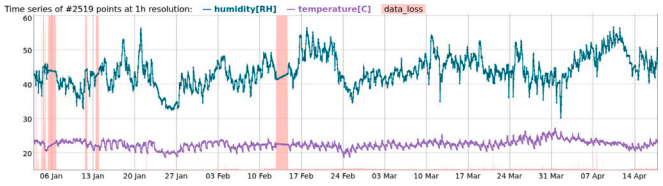


Fig. 3. The resampled time series plotted together with the data loss index, which is rendered as a red area chart.

4.3. Three-day forecast

First of all, the time zone of the time series is changed to the right one for this dataset. This is important because of the strong correlation of the data with the time of the day: leaving the timestamps on UTC would not allow the model to learn such dependency accurately, given that the daylight saving time switch in March would introduce an offset of one hour.

```
timeseries.change_tz("Europe/Rome")
```

A forecasting model based on a LSTM neural network is now instantiated, cross-validated, and fitted. Then, it is applied with a forecasting horizon of 72 (hourly) steps, for a total of three days, and the results plotted (Fig. 4). Data marked as missing is excluded from both the training and the evaluation of the model.

```
from timeseria.models import LSTMForecaster
LSTM_forecaster = LSTMForecaster(window=24, \
  neurons=256, features=["values", "hours"])

crossval=LSTM_forecaster.cross_validate(timeseries, \
  rounds=3, evaluate_error_metrics=["RMSE", "MAPE"]))
print(crossval)
```

```
[INFO] Cross validation round 1/3: validate from 1546477200.0 (2019-01-03
01:00:00+00:00) to 1549497600.0 (2019-02-07 00:00:00+00:00), fit on the rest.
[INFO] Cross validation round 2/3: validate from 1549497600.0 (2019-02-07
00:00:00+00:00) to 1552518000.0 (2019-03-13 23:00:00+00:00), fit on the rest.
[INFO] Cross validation round 3/3: validate from 1552518000.0 (2019-03-13
23:00:00+00:00) to 1555538400.0 (2019-04-17 22:00:00+00:00), fit on the rest.

{'humidity[RH]_RMSE_avg': 1.1284, 'humidity[RH]_RMSE_stdev': 0.1547,
 'humidity[RH]_MAPE_avg': 0.01815, 'humidity[RH]_MAPE_stdev': 0.0013,
 'temperature[C]_RMSE_avg': 0.3584, 'temperature[C]_RMSE_stdev': 0.0152,
 'temperature[C]_MAPE_avg': 0.0118, 'temperature[C]_MAPE_stdev': 0.0010}
```

```
forecaster.fit(timeseries, epochs=100, \
  reproducible=True)
forecaster.apply(timeseries, steps=72).plot()
```

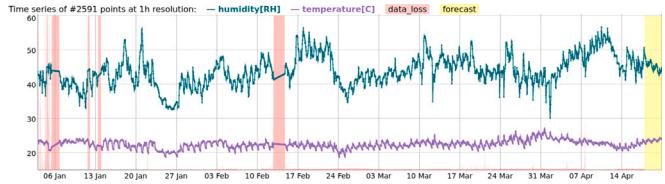


Fig. 4. The time series plotted together with the three days (72 hourly steps) forecast, which is highlighted in yellow using a "forecast" data index.

4.4. Perform anomaly detection

This example fits and apply a simple anomaly detection model based on periodic averages. By default the anomaly detectors assumes to work in unsupervised mode, using some "sane defaults". Data marked as missing is excluded from the anomaly detection process, in order to prevent false positives. The resulting time series is then plotted together with the `anomaly` data index (Fig. 5), which allows for visual inspection: the bigger anomaly is detected around the 25th of January, when there is an unexpected drop in both temperature and humidity with respect to the "normal" behavior.

```
from timeseria.models import \
  PeriodicAverageAnomalyDetector
anomaly_detector = PeriodicAverageAnomalyDetector()
anomaly_detector.fit(timeseries, periodicity=24)
```

```
[INFO] Using a window of "24" for "humidity[RH]"
[INFO] Using a window of "24" for "temperature[C]"
[INFO] Predictive model(s) fitted, now evaluating...
[INFO] Computing actual vs predicted for "humidity[RH]"...
[INFO] Computing actual vs predicted for "temperature[C]"...
[INFO] Model(s) evaluated, now computing the error distribution(s)...
[INFO] Anomaly detector fitted
```

```
anomaly_detector.apply(timeseries).plot()
```

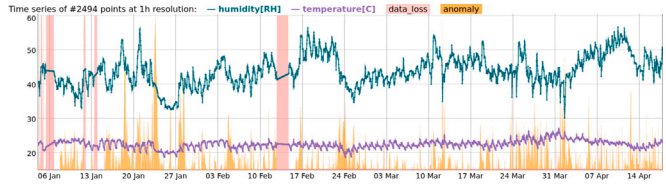


Fig. 5. The time series plotted together with the anomaly index: the closer the index is to the top of the plot (corresponding to the value "1"), the bigger the anomaly is.

4.5. Aggregate to daily data

The time series is aggregated in daily data, correctly handling the daylight saving time change in March. In the aggregation process, the minimum and maximum operations are also computed, besides the (default) average one. The time series is then plotted (Fig. 6).

```
timeseries = timeseries.aggregate("1D", \
  operations=["min","max","avg"])
timeseries.plot()
```

```
[INFO] Using auto-detected sampling interval: 3600.0s
[INFO] Aggregated 2494 points in 103 slots
```

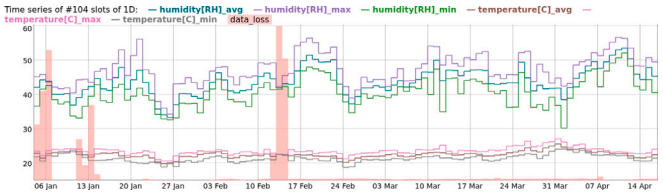


Fig. 6. The time series aggregated in daily slots, together with the data loss index.

4.6. Using custom models

The examples above made use of Timeseria's built-in models. However, as previously mentioned, the main goal of Timeseria is to let users plug-in their own modeling logic with minimal effort. This example thus demonstrated how to define a custom forecaster, by just implementing and decorating the `fit()` and `predict()` methods. The forecaster has a window of two elements, and a trivial logic which uses the average values computed during the fit phase averaged with the window elements to compute the prediction:

```
from timeseria.models import Forecaster

class MyForecaster(Forecaster):

  window = 2

  @Forecaster.fit_method
  def fit(self, series, verbose=False):
    self.data["avg"] = series.avg()

  @Forecaster.predict_method
  def predict(self, series, steps=1):

    if steps > 1:
      raise NotImplementedError()

    prediction = {}
    for label in self.data["avg"].keys():
      prediction[label] = (series[-1].data[label] + \
                  series[-2].data[label] + \
                  self.data["avg"][label]) / 3
    return prediction
```

The forecaster automatically inherited the `apply()`, `evaluate()`, `cross_validate()`, `save()`, and `load()` methods, which can all be used as already introduced. In a similar way, also custom model-based anomaly detectors can be defined, but they just need the model class to be set, as it follows:

```
from timeseria.models import ModelBasedAnomalyDetector

class MyAnomalyDetector(ModelBasedAnomalyDetector):
  model_class = MyForecaster
```

The anomaly detector defined in this way inherited all the main functionalities from the base class, as the `fit()`, `apply()`, `save()`, and `load()` methods, and can be used as previously illustrated, with no further effort.

5. Impact

As of today, Timeseria is used for ongoing research in environmental science, astrophysics and economics, and in a few private companies in the water and energy management space, while being progressively rolled out. We believe that Timeseria can impact many fields, and let researchers and professionals focus more on their core tasks and less on repetitive, low added-value ones, such as handling data consistency, time and calendar arithmetic, edge cases, model evaluation, inspection of the results, and more. We also believe that a structured approach as proposed in this work can impact several aspects of time series analysis overall, as improving reproducibility, reducing false positives and negatives, preventing misleading conclusions, and in general to keep analysis process under control with the ultimate goal of fostering new research and applications.

6. Conclusions

We presented Timeseria, an object-oriented time series processing library implemented in Python. Our goal was to make it easier to manipulate time series data and to build statistical and machine learning models on top of it. We tackled the various issues that arise in performing such tasks at their roots, defining a novel representation for time series data based on well defined logical units (objects), which we then used to build the library upon. Thanks to this approach, we were able to handle most of the issues that arise in processing time series data by design, with particular attention to practical challenges and real-world scenarios.

Early adoption of Timeseria proved it to be a valid tool for everyday work, preventing both naive (yet hard to debug) and more substantial errors, while speeding up all the various activities. Future work include adding support for probabilistic data structures and models (under development), adding more built-in models, and improving the performance with Cython, Numba, PyPy or similar solutions.

CRediT authorship contribution statement

Stefano Alberto Russo: Writing – review & editing, Writing – original draft, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Giuliano Taffoni:** Project administration, Funding acquisition. **Luca Bortolussi:** Writing – review & editing, Validation.

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.

Acknowledgments

Supported by the Italian Research Center on High Performance Computing, Big Data and Quantum Computing (ICSC), funded by the European Union - NextGenerationEU - and the Italian National Recovery and Resilience Plan (NRRP) - Mission 4, Component 2 - within the activities of the Spoke 3 (Astrophysics and Cosmos Observations).

References

[1] Siebert J, Groß J, Schroth C. A systematic review of packages for time series analysis. Eng Proc 2021;5(1):22.
[2] Christ M. Standardize time series formats. 2021, https://github.com/MaxBenChrist/awesome_time_series_in_python/blob/master/standardize_time_series_formats.md. [Accessed: 24 January 2025].
[3] Van Der Donckt J, Van Der Donckt J, Deprost E, Van Hoecke S. Tsflex: Flexible time series processing & feature extraction. SoftwareX 2022;17:100971.
[4] Cowpertwait PS, Metcalfe AV. Introductory time series with R. Springer Science & Business Media; 2009.
[5] Ollenschläger M, Küderle A, Mehringer W, Seifer A-K, Winkler J, Gaß ner H, et al. MaD GUI: An open-source python package for annotation and analysis of time-series data. Sensors 2022;22(15):5849.
[6] McPherson M, Sotiropoulos-Michalakakos T, Harvey LD, Karney B. An open-access web-based tool to access global, hourly wind and solar PV generation time-series derived from the MERRA reanalysis dataset. Energies 2017;10(7):1007.
[7] Ruszczak B, Kotowski K, Andrzejewski J, Haskamp C, Nalepa J. OXI: An online tool for visualization and annotation of satellite time series data. SoftwareX 2023;23:101476.
[8] McKinney W, et al. Pandas: a foundational python library for data analysis and statistics. Python High Perform Sci Comput 2011;14(9):1–9.
[9] Hoyer S, Hamman J. Xarray: ND labeled arrays and datasets in python. J Open Res Softw 2017;5(1).
[10] Löning M, Bagnall A, Ganesh S, Kazakov V, Lines J, Király FJ. Sktime: A unified interface for machine learning with time series. 2019, arXiv preprint 1909.07872.
[11] Tavenard R, Faouzi J, Vandewiele G, Divo F, Androz G, Holtz C, et al. Tslearn, a machine learning toolkit for time series data.. J Mach Learn Res 2020;21(118):1–6.
[12] Alexandrov A, Benidis K, Bohlke-Schneider M, Flunkert V, Gasthaus J, Januschowski T, et al. GluonTS: Probabilistic and neural time series modeling in Python. J Mach Learn Res 2020;21(116):1–6.
[13] Khider D, Zhu F, Gil Y. Autots: Automated machine learning for time series analysis. In: AGU fall meeting abstracts, vol. 2019, 2019, p. PP43D–1637.
[14] Taylor SJ, Letham B. Forecasting at scale. Amer Statist 2018;72(1):37–45.
[15] Hosseini R, Chen A, Yang K, Patra S, Su Y, Al Orjany SE, et al. Greykite: Deploying flexible forecasting at scale at linkedin. In: Proceedings of the 28th ACM SIGKDD conference on knowledge discovery and data mining. 2022, p. 3007–17.
[16] Zhao Y, Nasrullah Z, Li Z. PyOD: A python toolbox for scalable outlier detection. J Mach Learn Res 2019;20(96):1–7.
[17] Alnegheimish S, Liu D, Sala C, Berti-Equille L, Veeramachaneni K. Sintel: A machine learning framework to extract insights from signals. In: Proceedings of the 2022 international conference on management of data. Association for Computing Machinery; 2022, p. 1855–65.
[18] Herzen J, Lässig F, Piazzetta SG, Neuer T, Tafti L, Raille G, Pottelbergh TV, et al. Darts: User-friendly modern machine learning for time series. J Mach Learn Res 2022;23(124):1–6.
[19] Facebook Research. Kats - a one stop shop for time series analysis in python. 2021, https://github.com/facebookresearch/Kats. [Accessed: 24 January 2025].
[20] Tinkoffru Artificial Intelligence Center. ETNA - predict your time series the easiest way. 2024, https://github.com/tinkoff-ai/etna. [Accessed: 24 January 2025].
[21] Wu R, Keogh EJ. Current time series anomaly detection benchmarks are flawed and are creating the illusion of progress. IEEE Trans Knowl Data Eng 2021;35(3):2421–9.
[22] Blázquez-García A, Conde A, Mori U, Lozano JA. A review on outlier/anomaly detection in time series data. ACM Comput Surv 2021;54(3):1–33.
[23] Russo SA. Timeseria: an object-oriented time series processing library, reference documentation. 2024, https://timeseria.readthedocs.io/en/latest/. [Accessed: 24 January 2025].
[24] Dygraphs - a fast, flexible open source JavaScript charting library. 2013, https://dygraphs.com/. [Accessed: 24 January 2025].