PAPER • **OPEN ACCESS**

altx: a python package for adaptive law-based transformation in time series classification

IOP Publishing Mach. Learn.: Sci. Technol. 7 (2026) 015034 https://doi.org/10.1088/2632-2153/ae3e4f

MACHINE LEARNING
Science and Technology

PAPER

altx: a python package for adaptive law-based transformation in time series classification

Balázs P Halmos[1,2], **Balázs Hajós**[2,3], **Vince Á Molnár**[1,2], **Marcell T Kurbucz**[4,*] and **Antal Jakovác**[2,5]

[1] Faculty of Engineering and Natural Sciences, Tampere University, Kalevantie 4, Tampere 33100, Finland
[2] Department of Computational Sciences, Wigner Research Centre for Physics, 29–33 Konkoly-Thege Miklós Street, Budapest 1121, Hungary
[3] Faculty of Science, Eötvös Loránd University, 1/A Pázmány Péter Walkway, Budapest 1117, Hungary
[4] Institute for Global Prosperity, University College London, 9–11 Endsleigh Gardens, London WC1H 0EH, United Kingdom
[5] Department of Statistics, Corvinus University of Budapest, 8 Fővám Square, Budapest 1093, Hungary
[*] Author to whom any correspondence should be addressed.

E-mail: m.kurbucz@ucl.ac.uk

Keywords: software, python package, time series classification, feature extraction, adaptive law-based transformation, machine learning

OPEN ACCESS

RECEIVED
22 July 2025

REVISED
16 January 2026

ACCEPTED FOR PUBLICATION
27 January 2026

PUBLISHED
10 February 2026

Abstract

We introduce altx, an open-source python package for computationally lightweight and transparent time series classification pipelines. The altx package implements the adaptive law-based transformation, a multiscale feature extraction method that maps raw time series to compact tabular feature vectors by pooling class-labeled law responses across windows and scales. The approach extends the linear law-based transformation with a multiscale shifted-window schedule while preserving transparency. The package provides a GPU-capable PyTorch implementation with an estimator-style interface, enabling straightforward integration into modern machine-learning workflows and interoperability with common scientific Python toolkits. We include illustrative examples and summarize representative benchmark results reported in our companion methodological paper.

1. Introduction

Time series classification (TSC) is a core task in machine learning [1, 2], with applications spanning healthcare [3], industrial diagnostics [4], and environmental sensing [5]. Related tasks arise in domain settings such as financial trend prediction [6], the analysis of stochastic dynamics in physics (e.g. anomalous diffusion) [7], and early warning and forecasting of food insecurity [8–11]. Modern TSC methods must cope with noise, temporal misalignment, and discriminative patterns that occur at multiple time scales, while balancing accuracy, interpretability, and computational efficiency. These requirements motivate representation learning methods that are both practical to deploy and easy to integrate with standard classifiers.

Recent progress in TSC has been driven by learning-based approaches such as InceptionTime [12], ensemble methods including HIVE-COTE [13, 14], and tree-based classifiers such as TS-CHIEF [15]. Random-feature methods, notably ROCKET [16] and its extensions MiniROCKET [17] and MultiROCKET [18], have shown that strong performance can be achieved with simple classifiers when paired with appropriate feature representations. Recent work also explores higher-order (hypergraph/simplicial) neural architectures for spatio-temporal forecasting, illustrating the breadth of time-series representation learning [19, 20]. While these approaches often deliver high benchmark accuracy, they may involve substantial computational cost, reduced transparency, or limited control over the extracted representations. These trade-offs motivate complementary feature representations that offer greater transparency and user control while remaining compatible with standard classifiers.

Table 1. Metadata of the `altx` package.

Metadata description	Metadata contents
Current code version:	1.0.0
Permanent link:	https://github.com/dcintlab/altx
Legal code license:	GPL-3.0
Code versioning system:	Git
Software language:	Python
System requirements:	OS agnostic (Linux, macOS, Windows). Python 3.9+.
Required Python packages:	`numpy`, `pandas`, `matplotlib`, `torch` (PyTorch)
Suggested Python packages:	`scikit-learn`, `jupyter`, `aeon`
External dependencies:	None. Any external classifier can be used downstream.
User manual:	https://github.com/dcintlab/altx/blob/main/README.md
Support email:	balazs.halmos@tuni.fi

In earlier work, we introduced the linear law-based transformation (LLT) [21–23] to exploit the observation that many time series exhibit approximately invariant linear relations over embedded trajectories, reminiscent of conserved quantities in dynamical systems. LLT constructs time-delay embeddings and uses spectral decomposition to extract low-variance directions ('laws') from labeled training instances, then represents new series through their responses to these class-labeled laws, yielding compact and inspectable features for downstream classifiers. This law-based view provides a transparent and lightweight alternative to heavily parameterized pipelines, with intuitive robustness to common distortions such as noise and amplitude scaling, and is available as an R package [24]. However, LLT operates at a single, fixed embedding/window scale, which can limit its ability to capture discriminative structure that appears at multiple temporal extents or shifted positions.

The adaptive law-based transformation (ALT) [25] extends the LLT principle to a multiscale setting by scanning time series with a schedule of shifted windows at multiple lengths. In this way, ALT retains the same core logic—deriving class-labeled laws from embedded windows and representing new series through their responses to these laws—while broadening the representation to capture scale- and position-dependent patterns. The methodological development, theoretical analysis, and comprehensive empirical evaluation of ALT are presented in [25].

The contribution of the present paper is software-centric: we introduce `altx`, a research-grade, graphics processing unit (GPU)-capable Python implementation of the ALT. The package is designed for integration into the contemporary Python TSC ecosystem, enabling use in `scikit-learn`-style pipelines and optional GPU acceleration via PyTorch. The package provides a transparent, model-agnostic feature transformation that interfaces naturally with standard classifiers and existing libraries such as `aeon`, `sktime`, and `tslearn`. Illustrative examples and representative benchmark results are included to characterize typical usage and practical behavior. Metadata for `altx` are summarized in table 1.

The remainder of the paper is organized as follows. Section 2 describes the ALT and its core computational steps. Section 3 presents the software design, implementation details, and the user-facing interface. Section 4 provides an end-to-end illustrative example demonstrating typical usage. Section 5 summarizes representative benchmark results and discusses computational complexity and practical scaling behavior. Finally, section 6 concludes the paper and outlines directions for future work.

2. Algorithm

ALT is a feature-based transformation for TSC that explicitly separates representation learning from downstream classification. Given labeled training data, ALT constructs class-labeled law dictionaries. Any standard learner can then be trained on the extracted features; class specificity enters through class-wise column selection during pooling (via $\pi^j_{r,l,k}$). Dictionary construction scans each channel with a multiscale, shifted-window schedule $\mathcal{R} \subset \mathbb{N}^3$ (a finite set of triplets (r,l,k)). For a fixed triplet (r,l,k), set $s = (r-1)/(2l-2)$ (implying $r = s(2l-2)+1$), so that each length-r window contributes $2l-1$ evenly spaced samples. Each window is embedded into an $l \times l$ symmetric delay-embedding matrix S, and the eigenvector associated with the eigenvalue of minimal magnitude is extracted as a local law vector. The resulting law vectors form the columns of $P^j_{r,l,k}$ and carry labels stored in $\pi^j_{r,l,k}$; the dictionary is thus class-labeled rather than split into separate per-class matrices.

At transformation time, a new instance z is embedded using the same schedule. For each (r,l,k) and channel j, ALT forms a row-embedding matrix A^j by sliding a length-l embedding with stride k and spacing s, and computes the law-response matrix $M^j = A^j P^j_{r,l,k}$. These responses quantify the alignment

Algorithm 1. Adaptive law-based transformation (ALT).

Require: Training series $\{x^{i,j}_{1:h_i}\}$ with labels y^i
Schedule $\mathcal{R} = \{(r,l,k)\}$ with $r = s(2l-2)+1$ and $s = (r-1)/(2l-2) \in \mathbb{N}$
Optional law-training ratio $\rho \in (0,1]$; pooling operators $\mathcal{E}$
Ensure Class-labeled law dictionaries $\{(P^j_{r,l,k}, \pi^j_{r,l,k})\}$ and feature map $\Phi(\cdot)$
Train (law dictionary construction)
Select training indices $\mathcal{I}_{LT}$ (optionally a fraction ρ of the training instances)
for each $(r,l,k) \in \mathcal{R}$ and channel j **do**
  $s \leftarrow (r-1)/(2l-2)$
  for each $i \in \mathcal{I}_{LT}$ and $t \leftarrow 1, k+1, 2k+1, \ldots, h_i - r + 1$ **do**
    Construct $S \in \mathbb{R}^{l\times l}$ with $S[p,q] = x^{i,j}_{t+(p+q-2)s}$ ▷ symmetric embedding
    $v \leftarrow$ eigenvector of S with minimal absolute eigenvalue
    Append v to $P^j_{r,l,k}$ and append label y^i to $\pi^j_{r,l,k}$
  end for
end for
Transform (feature extraction for a new instance z)
Let H be the length of z
for each $(r,l,k) \in \mathcal{R}$ and channel j **do**
  $s \leftarrow (r-1)/(2l-2)$; $\Delta \leftarrow (l-1)s+1$
  $o \leftarrow \left\lfloor \frac{H-\Delta}{k} \right\rfloor + 1$
  Form $A^j \in \mathbb{R}^{o\times l}$ with $A^j_{u,q} = z^j_{(u-1)k+(q-1)s+1}$ ▷ row embeddings
  $M^j \leftarrow A^j P^j_{r,l,k}$
  for each class c **do**
    $M^{j,(c)} \leftarrow$ columns of M^j with label c (via $\pi^j_{r,l,k}$)
    Compute energy responses and apply two-stage pooling: reduce over laws, then aggregate over windows using $\mathcal{E}$
  end for
end for
return the concatenated feature vector $\Phi(z)$ over all (r,l,k), channels j, classes, and pooling operators

between local segments of z and the learned laws at the corresponding scales and positions. Responses are then processed class-wise by selecting the columns associated with each class via $\pi^j_{r,l,k}$ and computing energy responses (squared magnitudes). Features are obtained by two-stage pooling: first reducing across laws (columns) within each class (e.g. mean or a chosen percentile), and then aggregating across window positions (rows) using the operators $\mathcal{E}$ (e.g. mean, variance, or higher-order moments). Concatenating the resulting scalars over all $(r,l,k) \in \mathcal{R}$, channels j, and classes yields the final fixed-length feature vector $\Phi(z)$. Implementation details such as using a designated subset of the training instances for dictionary construction and the choice of window-indexing convention do not affect the underlying ALT mapping.

Algorithm 1 summarizes the complete ALT pipeline, including dictionary construction (*train*) and feature extraction for a new instance (*transform*), following the ALT formulation in [25].

Since ALT is model-agnostic and outputs a fixed-length feature map $\Phi(\cdot)$, the resulting vectors can be used with any standard downstream classifier—e.g. linear models, support vector machines (SVMs), or k-nearest neighbors (k-NN).

3. Software description

`altx` is a lightweight Python package for adaptive law-based feature extraction in TSC. The package follows an estimator-style workflow: it first constructs class-labeled law dictionaries from labeled training data (`train`), and then transforms unseen instances into fixed-length, tabular feature vectors (`transform`/`transform_set`) that can be used with any downstream classifier. In the following, we describe installation and the typical usage workflow, and then summarize the public interface and parameters.

3.1. Installation

The `altx` package is implemented in Python and distributed as an open-source project. It is compatible with Linux, macOS, and Windows systems running Python 3.9 or later. The core dependencies are

limited to standard scientific libraries, most notably `NumPy` and `PyTorch`. Optional GPU acceleration is available via CUDA-enabled PyTorch installations.

The package can be installed directly from the public GitHub repository:

```
pip install git+https://github.com/dcintlab/altx.git
```

Optional packages such as `scikit-learn`, `aeon`, or `jupyter` are not required for the transformation itself but are commonly used for downstream classification, evaluation, and interactive experimentation.

3.2. Workflow

From a user perspective, interaction with `altx` follows a simple two-stage workflow aligned with section 2.

(a) **Law dictionary construction (`train`):** starting from labeled training data and a user-defined multiscale schedule (window length, embedding dimension, stride), `altx` scans the series, forms symmetric delay-embedding matrices, and extracts local law vectors via eigendecomposition. The extracted laws are stored together with their class labels in reusable dictionaries.

(b) **Feature transformation (`transform` / `transform_set`):** unseen time series are embedded using the same schedule, projected onto the stored laws, and summarized into fixed-length feature vectors through two-stage pooling across laws and window positions, yielding a tabular representation suitable for any downstream classifier.

This explicit `train`/`transform` separation prevents test-time leakage and allows the learned law dictionaries to be saved and reused for inference.

3.3. Practical parameter guidance

While ALT exposes only a small set of core parameters (r, l, k), their choice affects both representation quality and computational cost. As a simple empirical starting point, we recommend selecting a small embedding dimension l (e.g. $l \in \{3, \ldots, 6\}$) to keep dictionary construction efficient, and using the default relation $r = 2l - 1$ to define the corresponding window length. If longer-range temporal structure is expected, additional scales with larger r values can be included.

The stride parameter k controls the density of the window schedule and thus the number of extracted laws. Setting $k = 1$ yields the densest dictionaries and typically the strongest performance at the expense of higher runtime and memory usage, whereas larger values of k reduce both training cost and dictionary size. In practice, a robust workflow is to start with a small number of scales, increase $|\mathcal{R}|$ only if performance saturates, and adjust k to meet available runtime and memory constraints.

3.4. Structure

The `altx` package provides a compact, estimator-style public API for adaptive law-based feature extraction. Table 2 summarizes the main functions (upper panel) and the consolidated parameter definitions shared across them (lower panel).

Internally, dominant computations—including batches of small symmetric eigendecompositions and matrix multiplications—are implemented using PyTorch tensor operations. Consequently, the same code path supports both CPU execution and optional GPU acceleration without changes to the user-facing interface.

4. Illustrative example

This section illustrates a typical use of the `altx` package on the *GunPoint* dataset from the UCR TSC Archive [26, 27]. The purpose of the example is to demonstrate the end-to-end workflow of the transformation—from law dictionary construction to feature extraction and downstream classification—rather than to optimize predictive performance.

Table 2. Main `altx` functions and consolidated parameter definitions.

Functions

Name	Type	Parameters	Description
`ALT`	Constructor	1–7	Initializes the transformation by storing the multiscale (r, l, k) schedule, training data, and execution device.
`train`	Method	8	Constructs class-labeled law dictionaries from the training set.
`transform`	Method	9–10	Transforms a single (uni- or multivariate) time series into an ALT feature vector.
`transform_set`	Method	9, 11–15	Transforms a collection of time series into a feature matrix and optionally exports the results.
`save`	Method	13	Serializes learned law dictionaries to disk for reuse without access to the original training data.
`load`	Static method	16	Loads a previously saved ALT model for inference-only use.
`print_number_of_laws`	Utility	—	Reports the number of extracted laws per class and scale (available after training).

Parameters

ID	Name	Type	Description
1	`train_set`	*numpy.ndarray* *torch.Tensor*	Training instances used for law dictionary construction.
2	`train_classes`	*numpy.ndarray* *torch.Tensor*	One-dimensional array of class labels corresponding to `train_set`.
3	`train_length`	*numpy.ndarray* *torch.Tensor*	Optional vector specifying instance lengths for variable-length training data.
4	`R`	*int* / *list[int]*	Window lengths defining temporal scales. If omitted, defaults to $r = 2l - 1$ for each embedding dimension.
5	`L`	*int* / *list[int]*	Embedding dimensions controlling the dimension of extracted laws.
6	`K`	*int* / *list[int]*	Window translation strides controlling the density of extracted laws and responses.
7	`device`	*torch.device* / *str*	Execution device (e.g. `cpu` or `cuda`).
8	`cleanup`	*bool*	If `TRUE`, removes raw training data from memory after dictionary construction.
9	`extr_methods`	*list* / *list[list]*	Feature extraction specification; each entry is either `[`mean_all']` or `[method, q]` with a percentile $q \in [0, 1]$ (e.g. `[`mean', 0.05]`).
10	`z`	*numpy.ndarray* *torch.Tensor*	A single instance to be transformed by `transform`.
11	`test_set`	*numpy.ndarray* *torch.Tensor*	A collection of instances to be transformed by `transform_set`.
12	`test_length`	*numpy.ndarray* *torch.Tensor*	Optional vector specifying instance lengths for variable-length test data.
13	`save_file_name`	*string*	Path used by `save` to write a serialized model, and by `transform_set` to export features (CSV).
14	`save_file_mode`	*string*	Export mode for feature saving: `` `New file' ``, `` `Append feature' ``, or `` `Append instance' ``.
15	`test_classes`	*numpy.ndarray* *torch.Tensor*	Class labels for the transformed set; required only when exporting features.
16	`load_file_name`	*string*	Path used by `load` to restore a serialized model.

Note: Internal helper routines (prefixed with _) implement embedding, eigenvector extraction, dictionary assembly, matrix multiplication, pooling, and export utilities. These routines are not part of the public API and may change without affecting user-facing functionality.

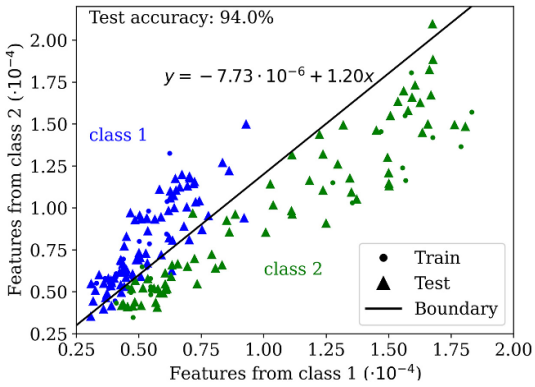

Figure 1. Two representative ALT features on *GunPoint* using a single-scale schedule ($r = 25$, $l = 4$, $k = 1$) and extraction including `["mean", 0.05]`. Train instances are shown as circles and test instances as triangles. The shown linear decision boundary (and reported accuracy) corresponds to a simple linear separator fitted on this 2D projection for illustration only; classification is performed externally to `altx`.

We begin by loading the dataset using the `aeon` interface and selecting a small subset of instances for law dictionary construction. The remaining instances are reserved for feature extraction and classification.

```
from aeon.datasets import load_classification
import numpy as np
import torch
import altx

X, y = load_classification("GunPoint")
y = y.astype(np.int64)

learn_X, learn_y = X[:10], y[:10]
test_X, test_y = X[10:], y[10:]
```

Next, we configure the ALT using a single-scale schedule and construct the class-labeled law dictionaries from the selected training subset.

```
alt = altx.ALT(
    train_set=learn_X,
    train_classes=learn_y,
    R=25, L=4, K=1,
    device=torch.device("cpu")
)

alt.train(cleanup=False)
```

After training, the learned dictionaries are applied to the remaining instances to obtain fixed-length feature vectors. In this example, two simple pooling configurations are used: the mean over all responses and the mean of the lower 5% quantile.

```
X_alt = alt.transform_set(
    test_X,
    extr_methods=[["mean_all"], ["mean", 0.05]]
)
```

The resulting feature matrix is a standard tabular representation and can be passed directly to any downstream classifier. For illustration, we combine the extracted features with a simple nearest-neighbor classifier implemented using `scikit-learn`.

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier

clf = make_pipeline(
    StandardScaler(),
    KNeighborsClassifier(n_neighbors=1)
)

clf.fit(X_alt, test_y)
```

Figure 1 shows a two-dimensional slice of the ALT feature space obtained from the configuration above. Each point corresponds to one transformed instance (circles: training subset used for feature

extraction; triangles: held-out instances), plotted using two selected ALT features computed from the pooled law responses. The plot illustrates that ALT tends to map instances from the same class into compact regions in feature space, which is beneficial for simple downstream classifiers.

5. Benchmark and complexity

The `altx` package is intended as a reusable feature extraction front-end rather than a stand-alone classifier. Its empirical value is therefore best assessed by how reliably it improves the performance and robustness of standard downstream learners. A detailed methodological evaluation is provided in the companion paper [25]. Here we summarize the results most relevant for software users, together with the expected computational costs.

5.1. Benchmark performance and practical regimes

We evaluated ALT on selected datasets from the UCR TSC Archive using lightweight downstream classifiers, comparing performance on raw time series and on ALT-derived features. The numerical results summarized in this section are reproduced from the companion methodology paper [25] and reformatted here for software users. In that study, evaluation protocols differ across methods due to computational cost, with repeated resampling used for KNN-based models and cross-validation without repetition for heavier baselines.

Table 3 presents median test accuracy and indicative runtimes on five representative datasets. Results contrast raw-series KNN with KNN trained on ALT features, and include contextual raw-series baselines based on an optimizable feed-forward neural network and the ROCKET method. For KNN+ALT, we distinguish between downstream model-selection/training time (once features are available) and total end-to-end time, which additionally includes the one-off ALT transformation on the training and test splits.

Across these representative tasks, ALT typically improves KNN test accuracy relative to raw-input classification, with the largest gains observed on noisy and industrial-scale series (*Epilepsy*, *FordA*, *FordB*). Downstream training remains lightweight and is typically comparable to, or faster than, raw KNN. End-to-end runtime depends on whether the ALT transformation cost can be amortized across repeated runs (e.g. model selection, repeated evaluation, or streaming updates). Compared with feed-forward neural networks, ALT achieves substantially higher accuracy on the more challenging benchmarks while requiring orders-of-magnitude less training time; relative to ROCKET, ALT remains particularly strong on the industrial-scale datasets.

Further results on additional UCR datasets, corresponding SVM analyses, corrected significance tests, and the LLT–ALT noise-robustness experiment are reported in [25].

5.2. Robustness and downstream efficiency

Beyond accuracy, two additional properties are particularly relevant for practical use. First, ALT exhibits improved robustness to additive noise: when Gaussian noise is injected into the input signals, ALT features typically degrade more gracefully than raw-input baselines and outperform single-scale law-based transformations across a wide noise range. This behavior is consistent with the law-based aggregation mechanism, whereby local perturbations tend to average out when responses are pooled across many embeddings and laws.

Second, classifiers trained on ALT features operate on compact, tabular representations rather than high-dimensional sequences. As a result, training times for standard models such as SVMs are often substantially reduced. Although dictionary construction incurs an up-front cost, this cost is paid only once per configuration. Learned dictionaries can be saved and reused across cross-validation folds, repeated experiments, or deployment scenarios.

5.3. Computational complexity and scaling behavior

From an implementation perspective, ALT consists of two computational phases with distinct scaling characteristics: dictionary construction (`train`) and feature extraction (`transform` / `transform_set`).

Dictionary construction:
During training, ALT slides a window over the law-training instances for each (r, l, k) triplet and channel, constructs symmetric $l \times l$ embedding matrices, and extracts a single law vector per window via eigendecomposition. If N_{LT} denotes the number of law-training instances, m the number of channels, and

Table 3. Representative comparison on selected UCR datasets (medians). Values are reproduced from [25] and reformatted for software users. For KNN+ALT, Downstream time excludes ALT preprocessing; Total time includes the one-off ALT transformation on the train+test splits. NN and ROCKET are trained on raw series; reported times include the full optimization/training procedures (cross-validation with Bayesian optimization, without repeated resampling). Reproduced from [25] CC BY 4.0.

	KNN + ALT			KNN (Raw)		NN (Raw)		ROCKET (Raw)	
Dataset	Test acc. (%)	Downstr. (s)	Total (s)	Test acc. (%)	Time (s)	Test acc. (%)	Time (s)	Test acc. (%)	Time (s)
BasicMotions	100.0	4.6	7.0	82.5	4.8	87.5	1941.7	95.0	265.9
Epilepsy	96.4	4.1	84.4	70.3	5.3	67.4	2745.3	90.6	781.3
Epilepsy2	93.8	3.8	124.5	84.6	3.7	89.9	1248.9	91.7	830.0
FordA	96.8	4.6	2016.8	71.5	8.0	72.0	7347.9	90.5	963.6
FordB	90.4	4.1	4654.1	65.7	6.7	66.0	5211.7	89.3	1146.2

$w_i(r,k)$ the number of windows for instance i at scale (r,k), then the dominant cost scales as

$$O\left(m \sum_{(r,l,k)\in\mathcal{R}} \sum_{i=1}^{N_{\mathrm{LT}}} w_i\,(r,k)\, l^3\right),$$

where the cubic dependence on l arises from eigendecomposition of $l \times l$ matrices. Using $W_{\max}$ and $L_{\max}$ as upper bounds on the number of windows and embedding dimension, this can be summarized as

$$O\big(N_{\mathrm{LT}}\, m\, |\mathcal{R}|\, W_{\max}\, L_{\max}^3\big)\,.$$

Memory complexity:
The dominant persistent memory cost (after training, assuming the raw training data are discarded via `cleanup`) is storing the learned law dictionaries $\{P^j_{r,l,k}\}$ (and label vectors $\{\pi^j_{r,l,k}\}$). If $n^j_{r,l,k}$ denotes the number of extracted laws at scale (r,l,k) for channel j, then the dictionary storage scales as

$$O\left(\sum_{(r,l,k)\in\mathcal{R}} \sum_{j=1}^{m} l\, n^j_{r,l,k}\right)$$

floating-point values (plus $O(\sum_{(r,l,k),j} n^j_{r,l,k})$ integers for labels). In practice, this footprint grows linearly with the number of windows processed during `train`, and can be controlled by reducing $|\mathcal{R}|$, increasing the stride k, lowering l, or limiting the law-training subset size. Temporary tensors created during `transform` (e.g. embeddings and response matrices) may increase peak memory usage but are not stored persistently.

Feature extraction:
At inference time, ALT avoids eigendecompositions and instead relies on batched matrix multiplications between embedding tensors and stored law matrices, followed by lightweight pooling operations. For a given scale, the cost scales approximately linearly in the number of extracted windows and the number of stored laws. Consequently, feature extraction is typically faster than training for fixed configurations and benefits strongly from GPU acceleration, particularly for dense schedules or multi-channel data.

Practical trade-offs:
The package exposes several interpretable knobs that allow users to balance accuracy against runtime and memory usage: reducing the number of scales $|\mathcal{R}|$, decreasing the embedding dimension l, increasing the stride k, or limiting the number of law-training instances. These parameters directly control the dominant terms in the above complexity expressions and reflect the design philosophy of ALT: enabling improved classification performance through multiscale evidence while keeping computational costs transparent and user-controllable.

6. Conclusion and future work

We introduced `altx`, an open-source Python package implementing the ALT as a transparent, multiscale feature extractor for TSC. The software provides an estimator-style workflow (`train`/`transform`), optional GPU acceleration via PyTorch, and outputs compact tabular representations that integrate naturally with standard machine-learning pipelines.

`altx` is designed as a reusable preprocessing front-end rather than a stand-alone classifier: it constructs class-labeled law dictionaries from embedded windows and represents unseen series through pooled law responses across scales and positions. This yields a controllable and inspectable feature space that supports lightweight downstream learners (e.g. k-NN or SVM) while keeping modeling and evaluation choices with the user.

In practical terms, `altx` offers the following value propositions:

- **Transparent multiscale representation:** features are derived from class-labeled laws and pooled responses, making the representation inspectable and tunable through a small set of schedule parameters (r, l, k).
- **Model-agnostic interoperability:** the extracted vectors can be used with any downstream classifier or evaluation protocol.
- **Efficient reuse:** learned dictionaries can be serialized and reused across experiments or deployments, avoiding repeated training costs when schedules are fixed.

Future work will focus on usability and scalability extensions while preserving transparency. We plan optional utilities for data-driven schedule selection and reproducible tuning of (r, l, k), and will investigate pruning or compression of redundant laws to reduce memory and runtime for dense schedules. We also aim to strengthen multivariate support through richer cross-channel embeddings and pooling schemes, and to evaluate the law-response representation in related tasks such as anomaly detection and segmentation. In addition, we plan diagnostic utilities (e.g. visual summaries of pooled law-response contributions across scales and positions, and per-class law usage) to help users identify which temporal regions and scales drive downstream decisions.

Finally, we will explore how the proposed representations could support causal analysis in stochastic settings, drawing on Markov-chain-based causal discovery approaches that target directed interactions and hidden common drivers [28], as well as on recent work on state-space reconstruction of Markov chains from autocorrelation structure [29]. We also plan to study integration with our degrees-of-freedom-based causal inference framework [30] and to assess these directions on real-world case studies (see, e.g. [31, 32]).

Data availability statement

The `altx` package is available under the GPL-3.0 license at https://github.com/dcintlab/altx. The benchmark datasets are from the UCR Time Series Classification Archive [26, 27]. The experimental protocol and full numerical results are reported in [25].

The data that support the findings of this study are openly available at the following URL/DOI:. Data: www.cs.ucr.edu/ eamonn/time_series_data_2018/ [27]; Code: https://github.com/dcintlab/altx. [33]

Acknowledgments

The research was supported by the Hungarian Government and the European Union in the framework of a Grant Agreement No. MILAB RRF-2.3.1-21-2022-00004. Project No. PD142593 was implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development, and Innovation Fund, financed under the PD_22 'OTKA' funding scheme. We also gratefully acknowledge the UCR Time Series Classification Archive team for curating and maintaining the benchmark datasets used in this study and for making them freely accessible to the research community.

Author contributions

Balázs P Halmos 0009-0006-6736-4572
Conceptualization (equal), Formal analysis (equal), Methodology (equal), Software (equal), Validation (equal), Visualization (equal), Writing – original draft (equal), Writing – review & editing (equal)

Balázs Hajós 0009-0005-2735-2836
Conceptualization (equal), Formal analysis (equal), Methodology (equal), Software (equal), Validation (equal), Visualization (equal), Writing – original draft (equal), Writing – review & editing (equal)

Vince Á Molnár 0009-0006-2859-3743
Conceptualization (equal), Formal analysis (equal), Methodology (equal), Software (equal), Validation (equal), Visualization (equal), Writing – original draft (equal), Writing – review & editing (equal)

Marcell T Kurbucz 0000-0002-0121-6781
Conceptualization (equal), Funding acquisition (equal), Methodology (equal), Supervision (equal), Validation (equal), Writing – original draft (equal), Writing – review & editing (equal)

Antal Jakovác 0000-0002-7410-0093
Conceptualization (equal), Funding acquisition (equal), Methodology (equal), Supervision (equal), Validation (equal), Writing – original draft (equal), Writing – review & editing (equal)

References

[1] Esling P and Agon C 2012 Time-series data mining *ACM Comput. Surv.* **45** 1–34
[2] Fawaz H I, Forestier G, Weber J, Idoumghar L and Muller P-A 2019 Deep learning for time series classification: a review *Data Min. Knowl. Discov.* **33** 917–63
[3] Morid M A, Sheng O R L and Dunbar J 2023 Time series prediction using deep learning methods in healthcare *ACM Trans. Manag. Inf. Syst.* **14** 1–29
[4] van den Hoogen J, Bloemheuvel S and Atzmueller M 2021 Classifying multivariate signals in rolling bearing fault detection using adaptive wide-kernel CNNs *Appl. Sci.* **11** 11429
[5] Zaini N, Ean L W, Ahmed A N and Malek M A 2022 A systematic literature review of deep learning neural network for time series air quality forecasting *Enviro. Sci. Pollut. Res.* **29** 4958–90
[6] Sezer O B, Gudelek M U and Ozbayoglu A M 2020 Financial time series forecasting with deep learning: a systematic literature review: 2005–2019 *Appl. Soft Comput.* **90** 106181
[7] Cai W, Hu Y, Qu X, Zhao H, Wang G, Li J and Huang Z 2025 Machine learning analysis of anomalous diffusion *Eur. Phys. J. Plus* **140** 183
[8] Andree B P J, Chamorro A, Kraay A, Spencer P and Wang D 2020 Predicting food crises World Bank Policy Research Working Paper (No. 9412) (https://doi.org/10.1596/1813-9450-9412)
[9] Andree B P J 2022 Machine learning guided outlook of global food insecurity consistent with macroeconomic forecasts World Bank Policy Research Working Paper (No. 10202) (https://doi.org/10.1596/1813-9450-10202)
[10] Wang D, Andrée B P J, Chamorro A F and Spencer P G 2022 Transitions into and out of food insecurity: a probabilistic approach with panel data evidence from 15 countries *World Dev.* **159** 106035
[11] Penson S, Lomme M, Carmichael Z, Manni A, Shrestha S and Andrée B P J 2024 A data-driven approach for early detection of food insecurity in Yemen's humanitarian crisis World Bank Policy Research Working Paper (No. 10768) (https://doi.org/10.1596/1813-9450-10768)
[12] Fawaz H I, Lucas B, Forestier G, Pelletier C, Schmidt D F, Weber J, Webb G I, Idoumghar L, Muller P-A and Petitjean F 2020 Inceptiontime: finding alexnet for time series classification *Data Min. Knowl. Discov.* **34** 1936–62
[13] Lines J, Taylor S and Bagnall A 2016 Hive-cote: the hierarchical vote collective of transformation-based ensembles for time series classification *2016 IEEE 16th Int.l Conf. on Data Mining (ICDM)* (IEEE) pp 1041–6
[14] Middlehurst M, Large J, Flynn M, Lines J, Bostrom A and Bagnall A 2021 Hive-cote 2.0: a new meta ensemble for time series classification *Mach. Learn.* **110** 3211–43
[15] Shifaz A, Pelletier C, Petitjean F and Webb G I 2020 Ts-chief: a scalable and accurate forest algorithm for time series classification *Data Min. Knowl. Discov.* **34** 742–75
[16] Dempster A, Petitjean F and Webb G I 2020 Rocket: exceptionally fast and accurate time series classification using random convolutional kernels *Data Min. Knowl. Discov.* **34** 1454–95
[17] Dempster A, Schmidt D F and Webb G I 2021 Minirocket: a very fast (almost) deterministic transform for time series classification *Proc. 27th ACM SIGKDD Conf. on Knowledge Discovery & Data Mining* pp 248–57
[18] Tan C W, Dempster A, Bergmeir C and Webb G I 2022 Multirocket: multiple pooling operators and transformations for fast and effective time series classification *Data Mini. Knowl. Discov.* **36** 1623–46
[19] Chen Y, Segovia I and Gel Y R 2021 Z-gcnets: time zigzags at graph convolutional networks for time series forecasting *Int. Conf. on Machine Learning* (PMLR) pp 1684–94 (available at: https://proceedings.mlr.press/v139/chen21o.html)
[20] Chen Y, Dominguez I S, Coskunuzer B and Gel Y 2022 Tamp-s2gcnets: Coupling time-aware multipersistence knowledge representation with spatio-supra graph convolutional networks for time-series forecasting *The Int. Conf. on Learning Representations* (ICLR) (available at: https://par.nsf.gov/servlets/purl/10333715)
[21] Jakovac A 2021 Time series analysis with dynamic law exploration (arXiv:2104.10970)
[22] Kurbucz M T, Pósfay P and Jakovác A 2022 Facilitating time series classification by linear law-based feature space transformation *Sci. Rep.* **12** 18026
[23] Pósfay P, Kurbucz M T, Kovács P and Jakovác A 2025 Lightweight ECG signal classification via linear law-based feature extraction *Mach. Learn.: Sci. Technol* **6** 035001
[24] Kurbucz M T, Pósfay P and Jakovác A 2024 LLT: an r package for linear law-based feature space transformation *SoftwareX* **25** 101623
[25] Kurbucz M T, Hajós B, Halmos B P, Molnár V A and Jakovác A 2025 Adaptive law-based feature representation for time series classification *Sci. Rep.* **15** 41775
[26] Dau H A, Bagnall A, Kamgar K, Yeh C-C M, Zhu Y, Gharghabi S, Ratanamahatana C A and Keogh E 2019 The UCR time series archive *IEEE/CAA J. Autom. Sin.* **6** 1293–305

[27] Dau H A, et al 2018 The ucr time series classification archive (available at: https://www.cs.ucr.edu/~eamonn/time_series_data_2018/)
[28] Stippinger M, Bencze A, Zlatniczki A, Somogyvári Z and Telcs A 2023 Causal discovery of stochastic dynamical systems: a Markov chain approach *Mathematics* **11** 852
[29] Jakovác A, Kurbucz M T and Telcs A 2024 State space reconstruction of markov chains via autocorrelation structure *J. Phys. A: Math. Theor.* **57** 315701
[30] Telcs A, Kurbucz M T and Jakovác A 2025 Unified causality analysis based on the degrees of freedom *Phys. Rev. E* **112** 014204
[31] Magyar M, Kovács L and Burka D 2021 Forecasting the spread of the covid-19 pandemic based on the communication of coronavirus sceptics *Eng. Proc.* **5** 35
[32] Kovács L and Büki F 2025 Regional differences in modelling covid-19 infections using Google trends data: evidence from Hungary *Reg. Stat.* **15** 1073–102
[33] Halmos B P, Hajós B, Molnár V Á, Kurbucz M T and Jakovác A 2025 altx: Adaptive law-based transformation *GitHub* (available at: https://github.com/dcintlab/altx)