SoftwareX 22 (2023) 101364

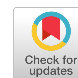
Contents lists available at ScienceDirect

SoftwareX

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

Original software publication

dtwParallel: A Python package to efficiently compute dynamic time warping between time series



Óscar Escudero-Arnanz *, Antonio G. Marques, Cristina Soguero-Ruiz, Inmaculada Mora-Jiménez, Gregorio Robles

Department of Signal Theory and Communications, Telematics and Computing Systems, King Juan Carlos University, Fuenlabrada 28942, Spain

A R T I C L E   I N F O

Article history:
Received 17 August 2022
Received in revised form 29 December 2022
Accepted 8 March 2023

Dataset link: https://github.com/oscarescuderoarnanz/dtwParallel/tree/main/exampleData/Data

Keywords:
DTW
Multivariate Time Series
Kernel-based similarity
Parallelization
Python

A B S T R A C T

`dtwParallel` is a Python package that computes the Dynamic Time Warping (DTW) distance between a collection of (multivariate) time series (MTS). `dtwParallel` incorporates the main functionalities available in current DTW libraries and novel functionalities such as parallelization, computation of similarity (kernel-based) values, and consideration of data with different types of features (categorical, real-valued, ...). A *low-floor*, *high-ceiling*, and *wide-walls* software design principle has been adopted, envisioning uses in education, research, and industry. The source code and documentation of the package are available at https://github.com/oscarescuderoarnanz/dtwParallel.

© 2023 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Code metadata

Current code version	v. 0.9.35
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-22-00246
Permanent link to Reproducible Capsule	N/A
Legal Code License	BSD 2-Clause
Code versioning system	git
Software code languages, tools, and services	Python3
Compilation requirements, operating environments and dependencies	numpy, pandas, matplotlib, seaborn, gower, setuptools, scipy, joblib, numba
If available, link to developer documentation/manual	https://github.com/oscarescuderoarnanz/dtwParallel/blob/main/README.md
Support email for questions	oscar.escudero@urjc.es

1. Motivation and significance

Time series (TS) are pervasive in contemporary applications [1] such as multimedia, medicine, or finance [2–4]. While classical tools to deal with TS exploit stationarity and frequency representations, the explosion in data availability has rendered more heterogeneous and complex TS (e.g., having different duration or including both categorical and numerical data). This has sparked the interest of the data-science and machine learning (ML) communities in dealing with both univariate TS (UTS) and multivariate TS (MTS) [2]. The definition of a well-suited distance (alternatively, a similarity) function is key to many ML approaches [5]. In this context, dynamic time warping (DTW), originally proposed

* Corresponding author.
E-mail addresses: oscar.escudero@urjc.es (Óscar Escudero-Arnanz), antonio.garcia.marques@urjc.es (Antonio G. Marques), cristina.soguero@urjc.es (Cristina Soguero-Ruiz), inmaculada.mora@urjc.es (Inmaculada Mora-Jiménez), gregorio.robles@urjc.es (Gregorio Robles).

https://doi.org/10.1016/j.softx.2023.101364
2352-7110/© 2023 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

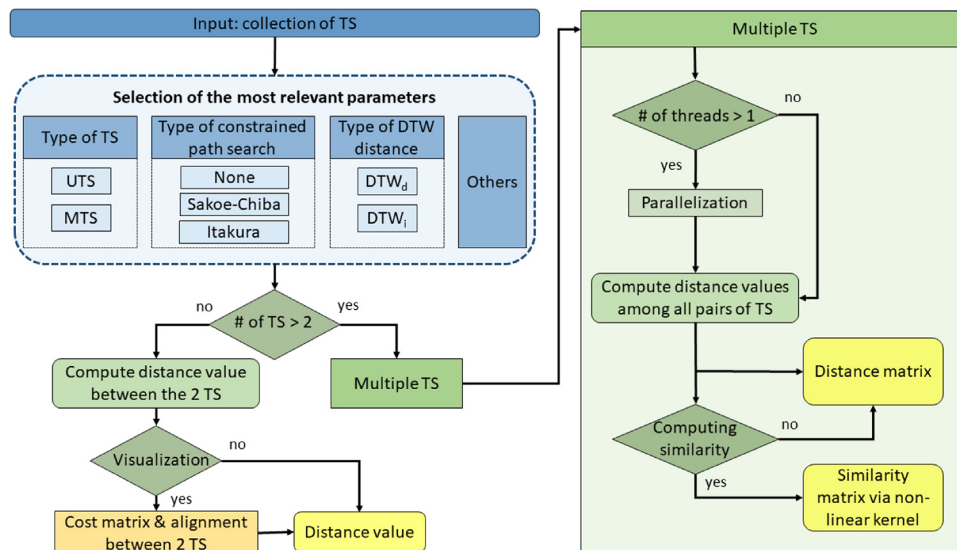


Fig. 1. Flow-diagram representing the most critical parts of the software architecture. Yellow boxes indicate outputs and blue ones input and configuration parameters. Only the most relevant input parameters are shown, for a comprehensive list, check Table 1. Visualization is only available when using `dtwParallel` with an API. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

to handle potential misalignments between univariate speech signals [6], has emerged as one of the most prominent distances for TS [7,8].

DTW has been employed in several ML tasks [1,5], and recent advances have reduced the computational complexity relative to the series length [1]. However, no efforts have been accomplished for setups with multiple MTS, where parallelization is advantageous. Moreover, current software implementations (e.g., [9–16]) exhibit additional limitations since they have been designed to address a specific problem. They may lack simplicity, ease of use, or do not allow working with features of different nature, among others. These limitations motivated `dtwParallel` with the ultimate goal of facilitating and fostering the use of DTW.

From a software-engineering viewpoint, `dtwParallel` is based on the *low-floor*, *high-ceiling*, and *wide-walls* software-design principle [17,18], which comes from educational environments and, in particular, from those that aim to bring programming to young people [18]. As a result, `dtwParallel` aims to: (i) render DTW accessible to everybody, even those with fewer technical and mathematical skills, making it suitable for learning (*low-floor*); (ii) be appropriate for advanced uses, including industry-ready applications (*high-ceiling*); and (iii) be versatile enough for many types of tasks (*wide-walls*).

`dtwParallel` combines the functionalities of the most relevant state-of-the-art DTW software packages (in particular [9–16]) with novel functionalities, such as computation of both independent and dependent DTW for MTS [19], consideration of multimodal data [20,21], parallelization, and computation of both distances and similarities (based on an exponential kernel) [22,23].

2. Software description

2.1. Introduction to `dtwParallel`

We focus first on the simplest case where the inputs x and y are two UTS with length (duration) T and T', respectively. The DTW distance between x and y is computed using `dtwParallel`, supplying $d_{DTW}(x, y)$ as output. Alternatively, if X is a collection of N UTS and Y of M UTS (typically as a matrix), the `dtwParallel` package returns an $N \times M$ matrix whose (i, j)-th entry represents the DTW distance between the i-th UTS in X and the j-th UTS in Y. Finally, if the input of `dtwParallel` is only X, the output is an $N \times N$ symmetric distance matrix whose (i, j)-th entry represents the DTW distance between the i-th and the j-th UTS in X. For MTS, `dtwParallel` works similarly: if two single MTS are used as input (each of them being a matrix with F rows, as many as features, and columns representing time instants), the output is a scalar; if two collections of MTS are used as input (typically as a tensor collecting either N or M matrices, each of the matrices representing a single MTS), the output is a non-symmetric rectangular matrix; and if only one collection of MTS is used as input, the output is a symmetric distance matrix. As detailed next, the additional inputs and configuration parameters allow the implementation of several types of local dissimilarity functions, including the L1-norm, the L2-norm, or the square Euclidean distance [24,25], among others. The transformation of a distance matrix into a similarity matrix via an exponential kernel and the visualization of intermediate results providing additional information on the temporal structure of the inputs is also implemented in `dtwParallel`.

2.2. Software architecture and functionalities

The `dtwParallel` package is designed to be easy to use and interact with via the command line or a Python API. Its operation can be tuned via 14 parameters (see Table 1), most of which activate additional functionalities and advanced options.

The flowchart of the package is detailed in Fig. 1. Some relevant features are summarized next:

- After inputting the set of TS whose distances must be computed, the users can specify the number of configuration parameters. These include the nature of the TS, the type of DTW distance (dependent or independent [19]), or if a constrained search method to accelerate the computation of the warping/alignment path is implemented (two constrained schemes, Itakura parallelogram [26] and Sakoe-Chiba band [6,8] are provided). The list of options, along with the "by default" values, is given in Table 1.
- When considering two TS, users can visualize the cost matrix, the optimal warping path used to compute the distance (for details, see [28]), and the alignment between TS. Visualizing the intermediate results provides insights into the alignment of both TS [8,28].

Table 1
Description of the parameters of `dtwParallel`, including default values.

API	Terminal	Description	Default
type_dtw	-t	Specifies the type of DTW scheme to deal with the multiple features in the MTS. The user can choose between "dependent DTW" (d) or "independent DTW" (i) [19].	d
constrained_path_search	-c	Specifies the global constraint for admissible paths in DTW. The user can choose between "itakura" [26], "sakoe_chiba" [6] or None.	None
local_dissimilarity	-d	Specifies the type of local dissimilarity/distance to be used. Among others, all distances in `scipy.spatial.distance` (see list in [27]), as well as the Gower distance (intended for multimodal features), are allowed.	Euclidean
check_errors	-ce	Enables carrying out an exhaustive error check.	False
visualization	-vis	Enables the visualization of the cost matrix with the optimal route used to find the value of the DTW distance (uses the API).	False
MTS	MTS	Indicates if the computation involves MTS (if true) or UTS (if false).	False
Not possible	-of	Enables saving the output in a *CSV* file.	False
Not possible	-nf	If the previous option is selected, sets the name of the output file.	output.csv
n_threads	-n	Sets the number of threads to parallelize the computation of the DTW distances among multiple pairs of MTS. The value -1 is equivalent to selecting all the available threads of your CPU.	-1
dtw_to_kernel	-k	Generates an additional output quantifying the similarities among the input TS via an (exponential) kernel transformation of the distance matrix entries.	False
sigma_kernel	-s	When performing the kernel transformation, sets the value of the parameter (sigma) specifying the width of the exponential kernel.	1
regular_flag	-rf	Value used to complete irregular MTS. This value is removed transparently to the user.	0
itakura_max_slope	-imx	Maximum slope for the Itakura parallelogram [26].	2
sakoe_chiba_radius	-scr	Radius to be used for Sakoe-Chiba band [6].	1

- When the number of TS exceeds 2, the activation of multi-threaded computation (parallelization) and the number of threads can be specified. Parallelization will speed up computation, especially for a large number of TS (see Section 4.2 for details). The DTW distance between each pair of TS has arranged in a (distance) matrix. Finally, an exponential kernel transformation (the width of the kernel must be indicated) can be applied to the distance matrix [23,29] to get a similarity matrix as an additional output. Similarity matrices are particularly suitable for visualization, supervised classification/regression, or unsupervised clustering.

In a nutshell, the main functionality of `dtwParallel` is to compute the DTW distance between two or more TS, where: (i) the TS can be either uni or multivariate; (ii) the TS can have different lengths; and (iii) for the multivariate case, features can be of the same type or multimodal. Regarding the definition of the DTW distance, several configuration choices are provided, including: (a) the function to quantify the dissimilarity between values of the TS at two time instants (Euclidean, Manhattan, among others); and (b) for the multivariate case, the DTW approach (independent or dependent) to deal with the multiple features in an MTS. From the execution point of view, `dtwParallel` offers the possibility of speeding up computations across pairs of series and features. From an application viewpoint, `dtwParallel` allows the generation of a similarity matrix quantifying the closeness between TS. Finally, our tool provides additional minor functionality such as reading the inputs from *CSV* or *npy* files and, for $N = 2$ input series, visualizing the time-instant-to-time instant cost matrix used to obtain the DTW distance.

E1

```
1  from dtwParallel import dtw_functions
2  from scipy.spatial import distance
3  import numpy as np
4
5  x = [2.5, 4.3, 6.6, 8.0, 1, 0, 0, 1, 5.5, 15.2]
6  y = [12.1, 0, 0, 1, 1, 6.4, 3.5, 1, 0, 0]
7
8  dtw_distance = dtw_functions.dtw(x, y, local_dissimilarity=distance.euclidean, check_errors=True)
9  print("DTW distance: ", np.round(dtw_distance,4))
10
11 dtw_distance = dtw_functions.dtw(x, y, local_dissimilarity=distance.canberra)
12 print("DTW distance: ", np.round(dtw_distance,4))
13
14 dtw_distance = dtw_functions.dtw(x, y, local_dissimilarity=distance.euclidean, get_visualization=True)
15 print("DTW distance: ", np.round(dtw_distance,4))
```

```
DTW distance:  45.4
DTW distance:  5.4687
DTW distance:  45.4
```

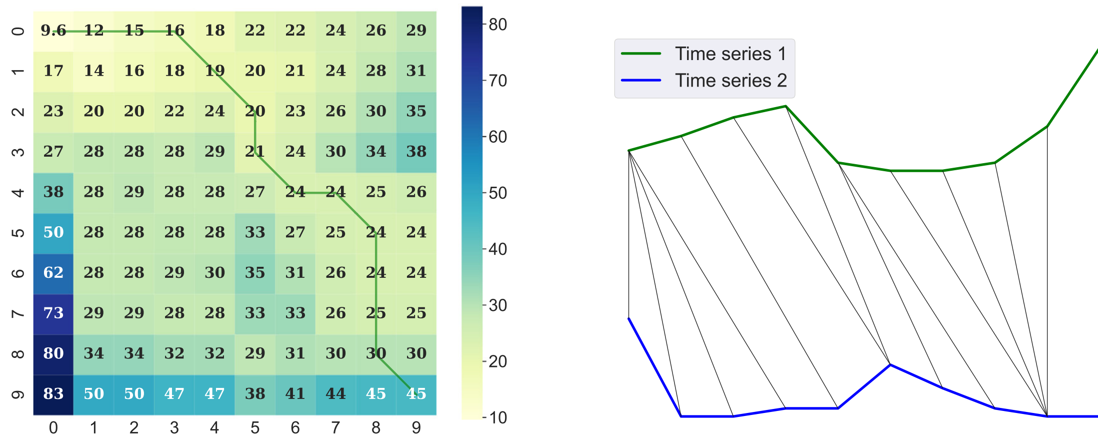


Fig. 2. Example showing the python code (top panel) to obtain the DWT distance for 2 UTS (x and y), visualizing the cost matrix and the path followed to reach the final distance (bottom-left panel), and visualizing the alignment between TS x and y (bottom-right panel). The bottom-right entry of the cost matrix represents $d_{DTW}(x, y)$. .

3. Illustrative examples

We present several examples that, with an increasing level of complexity, are intended to gain insights on DTW and become familiar with the `dtwParallel` package. The code of the experiments is available as a Jupyter Notebook at https://github.com/oscarescuderoarnanz/dtwParallel/tree/main/exampleData/CodeExamples.

3.1. Simple example: 2 UTS and optimal path visualization

The first example, labeled as **# E1** and shown in Fig. 2, illustrates how to find the distance between 2 UTS when using a Jupyter Notebook. First, we start with the imports (lines 1–3) and the definition of the input UTS (x and y, both with length $T = 10$) in lines 5 and 6. In line 8, we make the first call to the DTW function, using the Euclidean distance, activating the option to check for errors, and saving the output as a variable. The output is then printed (line 9). Lines 11 and 12 are the counterparts of 8 and 9, but without checking for errors and considering the Canberra distance. Finally, line 14 repeats the computation in line 8, also requesting the visualization of the cost matrix, the optimal warping path, and the alignment between time series. The calling line 14 returns: (i) the computed DTW distance (same value as in line 8), (ii) a colormap image containing both the entries of the 10×10 cost matrix and the optimal warping path (starting in the top left corner and finishing in the bottom right corner), and (iii) the alignment between the time series. The last value of the path in the cost matrix is the DTW distance [30].

3.2. Simple example: 2 MTS and multimodal data

 # E2 illustrates the package's operation when considering two MTS as inputs (X and Y), each of them with $F = 3$ and $T = 5$. The code in Fig. 3 reveals that two of the features are real-valued, while one of them (the second one) is binary. As a result, the Gower distance is used. Since MTS are considered, we specify whether to use the DTW_d or DTW_i approach. The code starts by importing the necessary packages (lines 1 and 2) and defining the MTS X (lines 4–10) and Y (lines 12–18) as 5×3 matrices. Finally, we call the `dtw` function (line 20) to compute the DTW_d between X and Y via the Gower distance.

3.3. Intermediate example: multiple UTS and similarity matrix

 # E3, detailed in Fig. 4, considers the DTW distance between N pairs of UTS, using two "subexamples" **# E3.1** and **# E3.2**. For **# E3.1**, we import the `dtwParallel` package (line 1 and 2), load the N UTS (line 4) onto the variable `data_A` (with size 1×3) and the N UTS (line 5) onto the variable `data_B` (with size 1×4). Both variables have $N = 3$ UTS of length $T = 4$. The distance between each of the N UTS in `data_A` and each of the N UTS in `data_B` is computed, parallelizing the computation with 9 threads (line 7). The output (a 3×3 matrix) is shown in Fig. 4. For **# E3.2**, we consider a single collection of $N = 3$ UTS and compute the DTW distance between those 3 UTS. The result, equivalent to computing the distance from `data_A` to itself, is arranged as a 3×3 symmetric matrix whose diagonal is zero.

E2

```
 1 from dtwParallel import dtw_functions
 2 import numpy as np
 3
 4 X = np.array([
 5               [3.3, 1, 1.75],
 6               [5.1, 1, 3.42],
 7               [1.5, 0, 0.55],
 8               [2.0, 1, 0.85],
 9               [1.2, 0, 2.25],
10             ])
11
12 Y = np.array([
13               [1.3, 1, 0.25],
14               [4.1, 1, 1.42],
15               [2.5, 1, 4.55],
16               [3.0, 1, 2.85],
17               [0.2, 0, 3.25],
18              ])
19
20 dtw_distance = dtw_functions.dtw(X, Y, type_dtw="i", local_dissimilarity="gower", MTS=True)
21 print("DTW distance: ", np.round(dtw_distance, 4))

DTW distance:  11.0
```

Fig. 3. Example code to compute the DTW_i distance between two MTS (X and Y).

E3.1

```
1 from dtwParallel import dtw_functions
2 import pandas as pd
3
4 data_A = pd.read_csv("../../Data/E1_SyntheticData/data_A.csv")
5 data_B = pd.read_csv("../../Data/E1_SyntheticData/data_B.csv")
6
7 dtw_distance = dtw_functions.dtw(data_A, data_B, n_threads=9)
8 print("DTW distance:\n", dtw_distance)

DTW distance:
 [[ 2. 23. 14.]
 [29. 17. 25.]
 [23. 38. 28.]]
```

E3.2

```
1 from dtwParallel import dtw_functions
2 import pandas as pd
3
4 data_A = pd.read_csv("../../Data/E1_SyntheticData/data_A.csv")
5
6 dtw_distance, Kernel_matrix = dtw_functions.dtw(data_A, n_threads=3, dtw_to_kernel=True, sigma_kernel=1.75)
7 print("DTW distance:\n", dtw_distance, "\n\nKernel matrix:\n", Kernel_matrix)

DTW distance:
 [[ 0. 26. 20.]
 [26.  0. 23.]
 [20. 23.  0.]]

Kernel matrix:
 [[1.         0.0143372  0.03818524]
 [0.0143372  1.         0.02339806]
 [0.03818524 0.02339806 1.        ]]
```

Fig. 4. Examples code for N UTS. **# E3.1** uses two collections of UTS (the resultant matrix distance is non-symmetric). **# E3.2** uses a single collection (the distance matrix is symmetric) and illustrates how the associated similarity matrix can be obtained.

This example also illustrates how a matrix quantifying the similarity across the input UTS can be obtained. As shown in Fig. 4, the necessary packages (lines 1 and 2) are imported, data are loaded (line 4), and the dtw function is called (line 6) by setting the parameters of the similarity transformation (`dtw_to_kernel` and `sigma_kernel`). Two 3×3 matrices are returned as output and printed (line 7): the distance matrix and the similarity matrix, both shown in Fig. 4.

3.4. Advanced example: real-world dataset

 # E4 considers financial data associated with the stocks of the SP500 index downloaded from Yahoo Finance (https://finance.yahoo.com/) and processed using the code in https://github.com/oscarescuderoarnanz/dtwParallel/tree/main/exampleData/CodeExamples/E2_FinanceData. We obtained 6 features (open-price, close-price, adjusted-close-price, highest-price, lowest-price, volume) for 505 US stocks during a period of 20 trading days. The data is arranged as a tensor of size $505 \times 20 \times 6$ saved as [*FinanceData_20days_norm.npy*]. Our goal is to compute the DTW distance among the 505 MTS series, providing the associated code in Fig. 5. We start by importing the required Python packages (lines 1–2) and creating the Input class (lines 4–18). Then, we load the 505 MTS from the *npy* file (lines 20 and 21), generate the object with the corresponding parameters (line 23) and call the `dtw_tensor_3d` method from `dtwParallel` (line 25), finally printing the symmetric 505×505 distance matrix.

4. Impact

Over time, access to information generated/recorded periodically is pervasive in contemporary data-science and engineering

 # E4

```
import numpy as np
from dtwParallel import dtw_functions as dtw

class Input:
    def __init__(self):
        self.check_errors = False
        self.type_dtw = "d"
        self.constrained_path_search = None
        self.MTS = True
        self.regular_flag = False
        self.n_threads = -1
        self.local_dissimilarity = "norm2"
        self.visualization = False
        self.output_file = True
        self.dtw_to_kernel = False
        self.sigma_kernel = 1
        self.itakura_max_slope = None
        self.sakoe_chiba_radius = None

X = np.load('../../Data/E2_FinanceData/FinanceData_20Days.npy', allow_pickle=True)
Y = X.copy()

input_obj = Input()
# API call.
distance_matrix = dtw_functions.dtw_tensor_3d(X, Y, input_obj)
print("Dimensions:", distance_matrix.shape, "\n")
print(distance_matrix)
```

```
Dimensions: (505, 505)

[[0.00000000e+00 1.15864701e+08 2.52994000e+07 ... 1.47812000e+07
  3.85234000e+07 3.19547002e+07]
 [1.15864701e+08 0.00000000e+00 7.31400001e+07 ... 1.23506201e+08
  1.69340200e+08 1.55661900e+08]
 [2.52994000e+07 7.31400001e+07 0.00000000e+00 ... 2.52220001e+07
  7.02448000e+07 5.84802000e+07]
 ...
 [1.47812000e+07 1.23506201e+08 2.52220001e+07 ... 0.00000000e+00
  3.52968000e+07 3.11542004e+07]
 [3.85234000e+07 1.69340200e+08 7.02448000e+07 ... 3.52968000e+07
  0.00000000e+00 3.56295000e+07]
 [3.19547002e+07 1.55661900e+08 5.84802000e+07 ... 3.11542004e+07
  3.56295000e+07 0.00000000e+00]]
```

Fig. 5. Example code of DTW distance among the 505 stocks in the SP500 index.

applications. Owing to its ability to deal with TS not necessarily synchronized and of different duration, DTW is broadly used to compute distances between TS in many applications [7,31–33].

Thus, it is not surprising that several software implementations of DTW exist, some of them with significant impact both from an academic (number of citations) and practitioners' (number of stars in the GitHub repositories) viewpoint. The two implementations with the largest number of Google Scholar citations are the `dtw` R package [9] and the `fastdtw` Python package [10], which, at the current time (December 2022), accumulate more than 900 and 2100 citations, respectively. Additionally, when a GitHub user adds a repository to his/her/their personal list for later reference, the project gets one GitHub star, prompting this metric to be used as a proxy for popularity among software developers [34]. Taking this into account, it is worth mentioning that there exist 5 DTW packages that currently have more than 100 stars: `tslearn` [16] (2.3k), `pyts` [15] (1.4k), `dtw` [12] (988), `dtaidistance` [13] (816) and `dtw-python` [11] (154).

Our long-term goal with `dtwParallel` is to offer a more comprehensive and easy-to-use DTW package, boosting the impact and adoption of DTW by a larger and more diverse set of users. Next, we delve into some of our comparative advantages.

4.1. Functionalities

We summarize in Table 2 the main functionalities of different DTW implementations. In particular, we compare `dtwParallel` with `dtw` [12], `dtaidistance` [13], `fastdtw` [10], `dtw-python` [11], `tslearn` [16], and `pyts` [15]. Since `pyts` only works with univariate time series, it has not been included in Table 2. The first observation is that, except for `dtwParallel`, none of the state-of-the-art alternatives implement the functionalities considered in this paper. Arguably, the most important difference relative to the state-of-the-art is the ability to compute the DTW distances distributing the load among processors (parallelization) since none of the listed implementations considers this option, challenging their use in tasks with a large number of TS. Note also that, for the MTS case, none of the implementations can compute DTW distances for multimodal MTS. On a related note, it also holds that none of the implementations offers functionalities that, while not difficult to compute, are quite useful in data-driven applications, such as DTW_i and kernel-based similarity measures. Finally, we note that some of the available packages, including `dtw-python`, lack examples and user-friendliness, hindering their adoption by non-expert users.

4.2. Parallelization

The number of entries of the distance matrix grows quadratically with N, the number of input TS. Since this hinders the use of DTW in large datasets, the use of parallelization to distribute the computation among the available threads is well-motivated. To illustrate the associated gains when using the `dtwParallel` package, we present three sets of experiments. The first one evaluates the computational time as the number of threads (a configuration parameter) increases. The second one evaluates the difference, in terms of computational time, between `dtwParallel` and several state-of-the-art DTW implementations. The third one evaluates the computational time of `dtwParallel` and the package providing the lowest computational time in the second experiment when varying the time series length. For all cases, the MTS of the SP500 dataset downloaded from Yahoo Finance (cf.

Table 2
Summary of the functionalities offered by the most relevant DTW implementations, and by `dtwParallel`. The number of GitHub stars (#stars) were obtained using data from December 2022.

Implementation (#stars)	Visualization	Multimodal Features	DTW_d	DTW_i	Parallelization	Kernel transformations
`tslearn` [16] (2.3k)	✓	–	✓	–	–	–
`dtw` [12] (988)	–	–	✓	–	–	–
`dtaidistance` [13] (816)	✓	–	✓	–	–	–
`fastdtw` [10] (650)	–	–	✓	–	–	–
`dtw-python` [11] (154)	✓	–	✓	–	–	–
`dtwParallel`	✓	✓	✓	✓	✓	✓

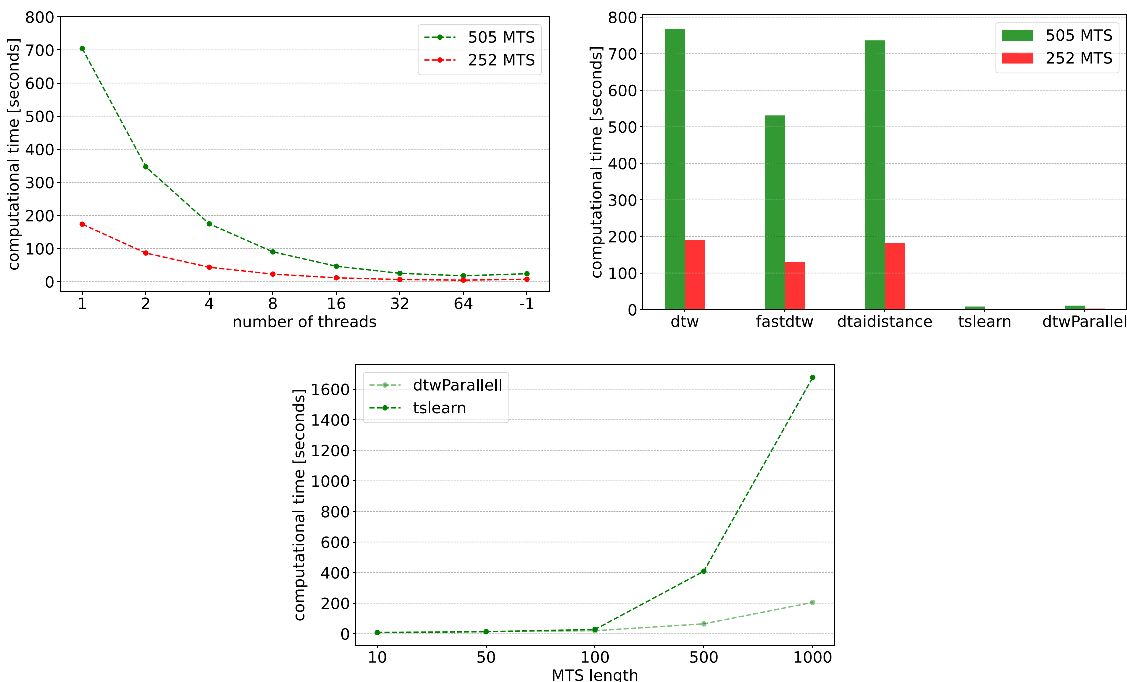


Fig. 6. Computational time (averaged over 30 repetitions) to compute DTW distances for the 505 MTS of the SP500 Yahoo Finance dataset run on an AMD-Ryzen-Threadripper-3990X processor (64 cores/128 threads) and RAM-DDR4-3200MHz-128 GB. Results for: (Top-left) different number of threads, with the value −1 in abscissas standing for "all threads available on the machine" (128 in our case); (Top-right) 5 popular DTW packages (4 packages are single-threaded, and `dtwParallel` is run with 32 threads); and (Bottom) `dtwParallel` and `tslearn` when varying the MTS length. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Section 3.4 for details) was considered, also adopting a dependent DTW approach among features and the Euclidean distance as the local dissimilarity function.

In the first experiment, a distance matrix of size 505 × 505 is computed, with each entry representing the DTW distance between 2 MTS with $F = 6$ and $T = 20$. The computational time averaged across 30 repetitions is reported for each particular number of threads [35]. Fig. 6 (top-left panel) shows the simulation results, with the vertical axis representing computational time (in seconds) and the horizontal axis the number of threads from 1 to 128. The number −1 in the abscissas axis refers to the total number of available threads (128 in this case). The results confirm that: (i) the computational time decreases as the number of threads increases; (ii) the savings are significant, going from approximately 665 seconds to around 18 seconds (using 64 threads); and (iii) the marginal savings decrease as the number of threads increase (see how the line flattens after 32). We rerun the experiments using approximately half of the MTS (252), noting that the computational time reduces by a factor of 4, and findings i-ii-iii also hold in this case.

The second experiment compares the computational time of the `dtwParallel` package with that of other DTW packages widely used in the literature. In particular, 4 alternative DTW implementations have been considered: `dtw`, `dtaidistance`, `fastdtw`, and `tslearn` [16] (a package about TS that includes the computation of the DTW distance). `dtw-python`, has not been included due to its more limited use and documentation. The computational times are reported in Fig. 6 (top-right panel), again averaging across 30 repetitions. For the `dtwParallel` package, the number of threads is set to 32. When comparing the computational time among the tested alternatives, we observe that `dtwParallel` and `tslearn` yield the lowest computational time, being 40–50 times faster. Clearly, the actual complexity depends on the value of N and the number of threads. To corroborate this, we also present the results when only 252 of the MTS are considered (see red bars in Fig. 6, top-right panel). In this case, while the computational time is 4 times smaller, the savings of our implementation are still in the order of ×20 than the other three alternatives.

The third experiment delves into the comparison of the computational time of the two fastest architectures: `dtwParallel` and `tslearn`. The number of threads is 32, and N is 505. Since the previous experiment considered MTS with a length of 20, we test a broader range of lengths here. The Fig. 6 (bottom panel) reveals that: (i) for low time lengths `tslearn` is faster than `dtwParallel`; (ii) for a length of 50, the computational time is similar; and (iii) for larger lengths `dtwParallel` is faster (8 times faster for a length of 1000).

5. Conclusions

dtwParallel is a new free/open software evaluating the DTW distance among a collection of UTS or MTS in a computationally efficient way. Under the umbrella of a single off-the-shelf Python package, it provides all the relevant functionalities present in the current state-of-the-art DTW libraries, complementing them with new meaningful functionalities. The latter include, among others: consideration of TS with multiple features, consideration of multiple pairs of series, use of parallelization across features and pairs of series to speed up computation, ability to deal with multimodal setups involving different types of data (categorical, binary, …), and generation of not only distance but also similarity values.

Our ultimate goal was to facilitate and foster the use of DTW distances in a diverse and broad range of data-science environments. Consideration of parallelization, multiple features, and multimodality was key to fostering the application to real-world datasets (which typically involve a large number of MTS and heterogeneous features). Similarly, the implementation of a *low-floor*, *high-ceiling*, and *wide-walls* software design methodology (which resulted in a more comprehensive design and the ability to generate multiple intermediate results and outputs) was aimed at enlarging the set of potential users (including those in education, research, and industry environments) with different levels of expertise.

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.

Data availability

The data used are public and are available in the package repository: https://github.com/oscarescuderoarnanz/dtwParallel/tree/main/exampleData/Data.

Acknowledgments

Work supported by the Spanish NSF (grants PID2019-106623RB-C41/AEI/10.13039/501100011033, PID2019-105032GB-I00AEI/10.13039/501100011033, and PID2019-107768RA-I00/AEI/10.13039/501100011033), and the Community of Madrid, Spain (grants PEJ-2020-AI/TIC-18964, URJC-F661, and e-Madrid-CM-P2018/TCS-4307, co-financed by EU Structural Funds FSE and FEDER, Spain).

References

[1] Wang X, Mueen A, Ding H, Trajcevski G, Scheuermann P, Keogh E. Experimental comparison of representation methods and distance measures for time series data. Data Min Knowl Discov 2013;26(2):275–309.
[2] Mikalsen KØ, Bianchi FM, Soguero-Ruiz C, Jenssen R. Time series cluster kernel for learning similarities between multivariate time series with missing data. Pattern Recognit 2018;76:569–81.
[3] Li H. Accurate and efficient classification based on common principal components analysis for multivariate time series. Neurocomputing 2016;171:744–53.
[4] Martínez-Agüero S, Soguero-Ruiz C, Alonso-Moral JM, Mora-Jiménez I, Álvarez-Rodríguez J, Marques AG. Interpretable clinical time-series modeling with intelligent feature selection for early prediction of antimicrobial multidrug resistance. Future Gener Comput Syst 2022;133:68–83.
[5] Li H, Liu J, Yang Z, Liu RW, Wu K, Wan Y. Adaptively constrained dynamic time warping for time series classification and clustering. Inform Sci 2020;534:97–116.
[6] Sakoe H, Chiba S. Dynamic programming algorithm optimization for spoken word recognition. IEEE Trans Acoust Speech Signal Process 1978;26(1):43–9.
[7] Berndt DJ, Clifford J. Using dynamic time warping to find patterns in time series. In: Proceedings of the international conference on knowledge discovery and data mining. 1994, p. 359–70.
[8] Keogh E, Ratanamahatana CA. Exact indexing of dynamic time warping. Knowl Inf Syst 2005;7(3):358–86.
[9] Giorgino T. Computing and visualizing dynamic time warping alignments in R: the dtw package. J Stat Softw 2009;31:1–24.
[10] Salvador S, Chan P. Toward accurate dynamic time warping in linear time and space. Intell Data Anal 2007;11(5):561–80.
[11] https://github.com/DynamicTimeWarping/dtw-python, [Accessed 29 December 2022].
[12] https://github.com/pollen-robotics/dtw, [Accessed 29 December 2022].
[13] https://github.com/wannesm/dtaidistance, [Accessed 29 December 2022].
[14] Folgado D, Barandas M, Antunes M, Nunes ML, Liu H, Hartmann Y, et al. TSSEARCH: Time series subsequence search library. SoftwareX 2022;18:101049.
[15] Faouzi J, Janati H. Pyts: A python package for time series classification. J Mach Learn Res 2020;21:1720–5.
[16] 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.
[17] Shneiderman B, et al. Creativity support tools: Report from a US national science foundation sponsored workshop. Int J Hum-Comput Interact 2006;20(2):61–77.
[18] Resnick M, et al. Scratch: programming for all. Commun ACM 2009;52(11):60–7.
[19] Shokoohi-Yekta M, Hu B, Jin H, Wang J, Keogh E. Generalizing DTW to the multi-dimensional case requires an adaptive approach. Data Min Knowl Discov 2017;31(1):1–31.
[20] Lahat D, Adali T, Jutten C. Multimodal data fusion: an overview of methods, challenges, and prospects. Proc IEEE 2015;103(9):1449–77.
[21] Gao J, Li P, Chen Z, Zhang J. A survey on deep learning for multimodal data fusion. Neural Comput 2020;32(5):829–64.
[22] Plaen HD, Fanuel M, Suykens JAK. Wasserstein exponential kernels. In: 2020 international joint conference on neural networks. IJCNN, 2020, p. 1–6.
[23] Gudmundsson S, Runarsson TP, Sigurdsson S. Support vector machines and dynamic time warping for time series. In: IEEE international joint conference on neural networks. 2008, p. 2772–6.
[24] Vaughan N, Gabrys B. Comparing and combining time series trajectories using dynamic time warping. Procedia Comput Sci 2016;96:465–74.
[25] Serra J, Arcos JL. An empirical evaluation of similarity measures for time series classification. Knowl-Based Syst 2014;67:305–14.
[26] Itakura F. Minimum prediction residual principle applied to speech recognition. IEEE Trans Acoust Speech Signal Process 1975;23(1):67–72.
[27] https://docs.scipy.org/doc/scipy/reference/spatial.distance.html, [Accessed 29 December 2022].
[28] Górecki T, Łuczak M. Multivariate time series classification with parametric derivative dynamic time warping. Expert Syst Appl 2015;42(5):2305–12.
[29] Lei H, Sun B. A study on the dynamic time warping in kernel machines. In: 2007 third international IEEE conference on signal-image technologies and internet-based system. 2007, p. 839–45.
[30] Seto S, Zhang W, Zhou Y. Multivariate time series classification using dynamic time warping template selection for human activity recognition. In: Symposium series on computational intelligence. 2015, p. 1399–406.
[31] Keogh EJ, Pazzani MJ. Scaling up dynamic time warping for datamining applications. In: Proceedings of the ACM SIGKDD international conference on knowledge discovery and data mining. 2000, p. 285–9.
[32] Tomasi G, Van Den Berg F, Andersson C. Correlation optimized warping and dynamic time warping as preprocessing methods for chromatographic data. J Chemometr: J Chemometr Soc 2004;18(5):231–41.
[33] Moor M, Horn M, Rieck B, Roqueiro D, Borgwardt K. Early recognition of sepsis with Gaussian process temporal convolutional networks and dynamic time warping. In: Proceedings of the machine learning for healthcare conference. 2019, p. 2–26.
[34] Borges H, Valente MT. What's in a github star? understanding repository starring practices in a social coding platform. J Syst Softw 2018;146:112–29.
[35] Georges A, Buytaert D, Eeckhout L. Statistically rigorous java performance evaluation. In: Proceedings of the ACM SIGPLAN conference on object-oriented programming systems, languages and applications. 2007, p. 57–76.