



TSFuse: automated feature construction for multiple time series data

Arne De Brabandere¹ · Tim Op De Beéck² · Kilian Hendrickx³ · Wannes Meert¹ · Jesse Davis¹

Received: 31 March 2020 / Revised: 5 May 2021 / Accepted: 26 July 2021 /

Published online: 16 May 2022

© The Author(s), under exclusive licence to Springer Science+Business Media LLC, part of Springer Nature 2022

Abstract

A central paradigm for building predictive models from time series data is to convert the data into a feature vector representation and then apply standard inductive learners. Typically, the conversion is done by manually defining features, which is an extremely time-consuming and error-prone process. This has motivated the development of algorithms that automatically construct features from time series. However, these systems are typically designed for univariate time series data. In contrast, many real-world applications require analyzing time series consisting of data collected by multiple sensors. In this context, it is often useful to derive new series by fusing the collected data both within a sensor and across multiple different sensors. Unfortunately, this poses additional challenges for automated construction as exponentially more operations are possible than in the univariate case. This paper proposes an automated feature construction system called TSFuse, which supports fusion and explores the search space in a computationally efficient way. We perform an empirical evaluation on real-world time series classification datasets and show that our system is able to find a better feature representation compared to existing feature construction systems for univariate time series data.

Keywords Automated data science · Feature construction · Time series analysis · Data fusion

1 Introduction

Time series data arise in a wide array of domains including health (Incel et al., 2013; Decroos et al., 2018), sports (Op De Beéck et al., 2018; Derie et al., 2020), finance (Kim, 2003), and industry (Helwig et al., 2015). A standard analysis task is to learn a predictive model from the collected data where typically each series is subdivided into windows and each window is annotated with a label. A canonical task that

Editors: Tim Verdonck, Bart Baesens, María Óskarsdóttir and Seppe vanden Broucke.

✉ Arne De Brabandere
arne.debrabandere@kuleuven.be

Extended author information available on the last page of the article

fits this description is activity recognition, where the goal is to classify a person's current activity (e.g., walking, sitting, running, biking, etc.) based on data from a sensor such as an accelerometer.

There are three broad categories of approaches for learning a model from time series data. First, the feature-based approach transforms the time series into a feature vector representation and then applies standard supervised learning on top of this representation (Schroeder et al., 2019; Yuan et al., 2019; Koch et al., 2019). Second, it is possible to apply deep learning (Fawaz et al., 2019) to the time series data which entails training an end-to-end neural network model that predicts the label starting from the raw time series data. Finally, it is possible to use a nearest neighbors approach on top of the raw data using a measure such as dynamic time warping to compare different time series (Keogh & Ratanamahatana, 2005).

This paper focuses on the feature-based approach. The key challenge is to define the *right set of features* as the quality of the feature representation strongly affects the predictive performance of the learned model. Typically, the conversion is done by manually defining features, which is an extremely time-consuming and error-prone process. To ease this burden, there has been increased interest in designing approaches for automatically constructing features from time series data (Fulcher, 2017). Typically, these approaches operate by first extracting a large number of predefined features from the data and then selecting a subset of the most relevant ones.

While these algorithms have gained traction for analyzing time series, they suffer from one important drawback: they focus solely on extracting features from univariate time series. In contrast, many real-world applications require analyzing time series consisting of data collected by multiple sensors. These applications are typically based on sensor *networks* where the sensors are positioned at different locations in an environment. This can arise in an industrial context, where a site could be instrumented with temperature, humidity, and air pressure sensors (Grzegorowski & Stawicki, 2015). Sports and health applications often involve attaching sensors on a person. For example, analyzing the movements of a runner requires attaching accelerometers at different locations of the runner's body (Op De Beéck et al., 2018). It is also possible to place sensors both on a person and in the environment. A well-known application is activity recognition, where a person's daily life activities can be monitored using sensors attached on the person as well as on different objects in the person's house (Roggen et al., 2010). For such applications, two types of transformations of the data are extremely important. First, it is often useful to define new series such as deriving the total acceleration from a triaxial accelerometer (Kwapisz et al., 2011) or computing the first-order derivative of the acceleration loadingrate. Second, it is also possible to construct meaningful features by comparing values from different series such as measuring the symmetry between the movement of the left and right leg of a person (Op De Beéck et al., 2018). Current approaches offer no support for these types of transformations and instead push the burden on the user if these are necessary.

In order to fill this gap, we propose an automated feature construction system called TSFuse that automatically transforms multiple time series into an attribute-value format. In contrast to the existing approaches (Christ et al., 2018; Fulcher & Jones, 2017) that only extract a predefined set of features from univariate data, TSFuse is able to both (1) derive new time series and then extract features from them, and (2) construct attributes from one or more existing ones. Supporting this expanded functionality dramatically increases the search space of operations and would make an exhaustive search computationally infeasible. To cope with this challenge, we design a search method that improves the computational efficiency in several ways such as restricting the search space by exploiting

constraints based on domain knowledge and identifying redundant transformations. The contributions of this work are summarized as follows:

1. We propose an automated approach to feature construction and selection that goes beyond existing work by operating on multiple time series and supporting a substantially broader set of features.
2. We propose a data representation that captures contextual information, which can be used to define constraints that reduce the search space to promote efficiency.
3. We perform an extensive empirical evaluation on ten real-world datasets and show that our approach is able to find a better feature representation compared to existing automated feature construction methods for univariate time series data. In addition, we show that our system outperforms the 1-nearest neighbour dynamic time warping baseline, and is able to outperform state-of-the-art deep learning methods in some cases.
4. We release a publicly available Python package called TSFuse¹ that supports the functionality described in this paper.

This paper is organized as follows. First, Sect. 2 describes our data representation. Section 3 describes how the system constructs an attribute-value representation from this data format. Section 4 describes the state-of-the-art of automated feature construction for time series. Finally, Sect. 5 experimentally evaluates our approach and compares the results to the state-of-the-art by applying the system to time series data collected in different application domains.

2 Data representation

In this section, we describe how TSFuse represents data that captures contextual information in real-world heterogeneous time series applications. To illustrate our representation, Fig. 1 shows a setup for monitoring a runner using four wearable sensors: one heart rate monitor on the chest, and three tri-axial accelerometers, one on the back, one on the left shin, and one on the right shin. While the heart rate monitor captures one signal (measured in beats per minute), each accelerometer captures three time series, that is, the acceleration in the x , y , and z direction (each series is measured in g).

Transforming and fusing these ten time series into a feature vector representation is non-trivial for several reasons. First, the sensors may use different sampling rates. For example, the heart rate monitor samples data at a lower frequency compared to the accelerometers. Second, there are a huge number of features that can be derived from these series, particularly if one wants to consider fusing different combinations of the input signals. For example, it may be useful to derive the resultant of the acceleration and extract features from this new signal (De Brabandere et al., 2018). It may also be useful to combine multiple signals at the level of the attributes extracted from the signals. For example, a runner's mean heart rate divided by the mean acceleration measured at the foot is a relevant feature for predicting the runner's physical fitness level (Weyand et al., 2001). However, considering all possible ways of combining multiple signals is computationally expensive as the number of possibilities grows exponentially: there are $O(n^p)$ possibilities when combining

¹ <https://github.com/arnedb/tsfuse>

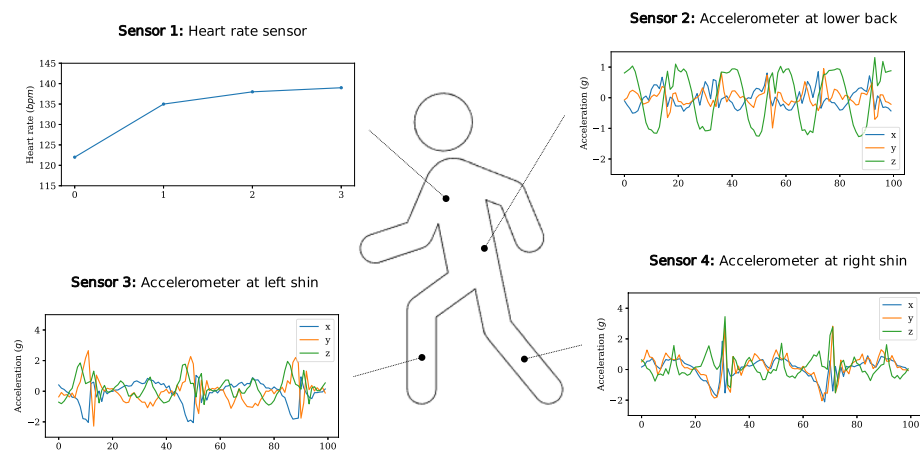


Fig. 1 Example of time series data collected by four sensors on a runner. The heart rate sensor collects a one-dimensional time series, measured in beats per minute. The other three sensors are tri-axial accelerometers: one on the lower back, one on the right shin, and one on the left shin. Each accelerometer collects one time series for each axis, measured in g

at most p out of n signals. Third, we often have domain knowledge about the problem that one can use to guide the search. In our running example, some combinations of signals are more meaningful than others. For example, fusing the accelerations on the left shin with the accelerations on the right shin can capture how symmetrical a runner moves whereas other combinations may not be meaningful from an application perspective.

To overcome these challenges, we define a data format that captures both the time series data as well as contextual information, which our feature construction exploits to improve the efficiency of searching for relevant features. TSFuse works on time series that have been divided into *windows*, where each window defines a start and end point and selects all data that was collected within this time period. Figure 1 shows the data that was collected within one specific window for each of the time series. Each window contains a number of *time series collections*, each of which groups together multiple contextually similar time series (e.g., collected by the same physical sensor or at the same location). Each time series collection has an associated *shape* of $t \times d$ where t is the number of measurements and d is the dimension of the series. Hence, the collections are multivariate. The collections are assumed to be time-synchronized, which means that the i th sample of each collection was recorded at the same time. In order to derive meaningful features, sensors of the same type are assumed to have the same units. Figure 1 shows how ten time series can be grouped into four time series collections (i.e., one per sensor). The data for the collection representing the heart rate has shape 4×1 whereas the three collections representing the acceleration data all have shape 100×3 . Each window is annotated with a label, which is a categorical value in the case of a classification problem, or a real value in the case of a regression problem. For an activity recognition task, the window shown in Fig. 1 could be annotated with a ‘jogging’ class label.

To capture the domain knowledge, TSFuse allows the user to assign *tags* to each time series collection. These tags empower the user to explicitly encode why time series are grouped into the same time series collection and may include information such as the sampling rate, measurement type, and location. For each time series collection x , the tags M_x

are represented as a dictionary $\{k \mapsto v\}$ that maps keys k to values v . For the example in Fig. 1, one could assign the following tags to the four time series collections x_1, x_2, x_3, x_4 :

$$\begin{aligned} M_{x_1} &= \{\text{type} \mapsto \text{heart-rate}; \text{location} \mapsto \text{chest}\} \\ M_{x_2} &= \{\text{type} \mapsto \text{accelerometer}; \text{location} \mapsto \text{back}\} \\ M_{x_3} &= \{\text{type} \mapsto \text{accelerometer}; \text{location} \mapsto \text{shin}; \text{side} \mapsto \text{left}\} \\ M_{x_4} &= \{\text{type} \mapsto \text{accelerometer}; \text{location} \mapsto \text{shin}; \text{side} \mapsto \text{right}\} \end{aligned}$$

This information will be leveraged by our system to improve both the efficiency of the feature construction process and the interpretability of the constructed features.

3 Feature construction

TSFuse addresses the following problem:

- Given:* A dataset with N windows, where each window consists of n time series collections $\{x_i\}_{i=1}^n$ and a label y . In addition to the time series data itself, the user can specify domain knowledge by assigning tags $\{M_{x_i}\}_{i=1}^n$ to the collections and defining constraints on the operations that can be applied to the collections.
- Construct:* An attribute-value representation of the data that enables predicting the label of an unseen window.

In order to construct the attribute-value representation, we define features by combining multiple transformation steps. We represent the features using a *computation graph*. Section 3.1 describes this representation. The feature construction problem involves discovering a good computation graph. This can be thought of as a search problem that requires picking which transformations to use as well as the order in which to apply these transformations. The key challenge that arises when searching for a good computation graph is that there exist a huge number of possible computation graphs. To make the search more efficient, we use constraints that reduce the number of possible graphs. Section 3.2 describes how these constraints are defined. We then describe our search procedure in Sect. 3.3.

3.1 Computation graph

A computation graph G is a directed acyclic graph that transforms a given window with time series collections $\{x_i\}_{i=1}^n$ into an attribute-value representation. Figure 2 shows a computation graph for the data collected in the running application described in Sect. 2. Conceptually, each node in the graph performs an operation to manipulate or *transform* the data in some way. Following the naming conventions of the scikit-learn API (Pedregosa et al., 2011), nodes that transform data are named *transformers*.² Edges in the graph represent the flow of the data and map the output of one node to the input of another one. A computation graph can contain six kinds of nodes:

² Transformers should not be confused with the equally named neural network architecture.

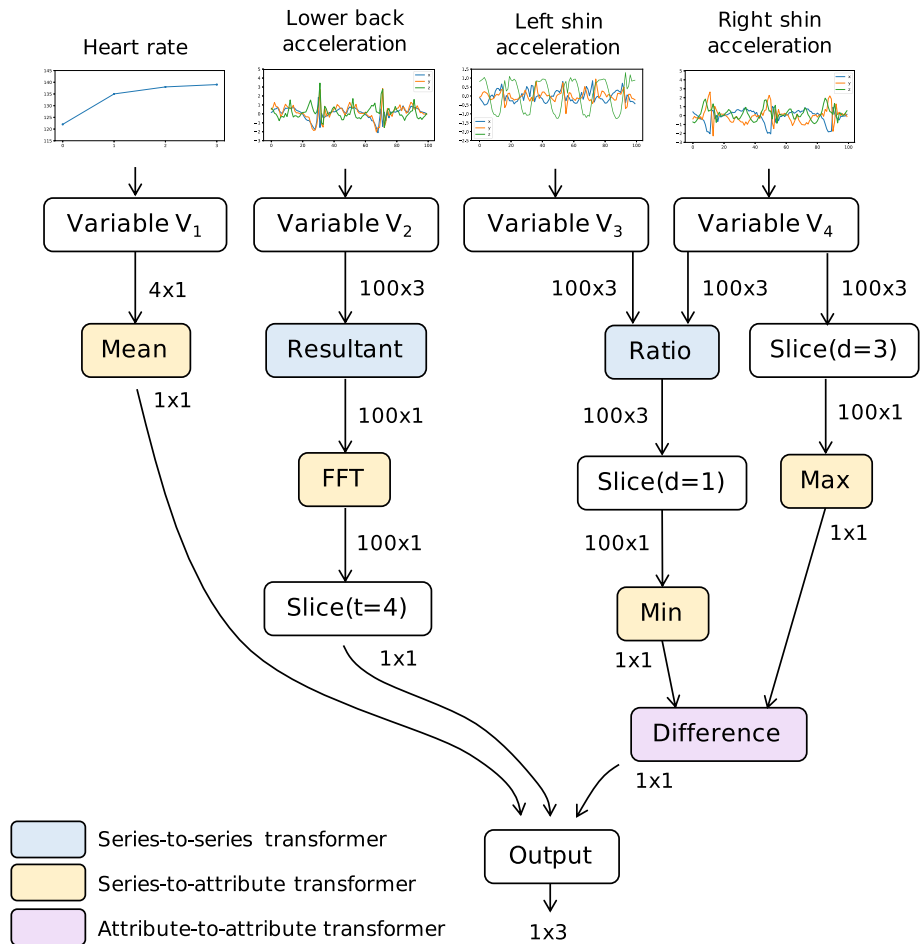


Fig. 2 Example of a computation graph for the running example introduced in Sect. 2. The graph has four variable nodes (V_1, V_2, V_3, V_4), one for each time series collection. The computation graph returns as output a feature vector of length three by concatenating the attribute collections returned by the Mean, Slice($t=4$), and Difference nodes. The figure shows the shape of the data for the outgoing edge of each node, where the data may either represent a time series collection or an attribute-value collection. The data type (attribute or time series) and tags associated with each edge are not shown for readability

Variable V_i

Variable nodes represent time series collections associated with windows. There is one variable node V_i in the graph for each time series collection x_i of a window.

Series-to-series transformer $T_{S \rightarrow S}(x_1, x_2, \dots, x_c)$

This transformer creates a new time series collection by fusing the input collections $x_1, \dots, x_c$. For example, the Ratio transformer in Fig. 2 takes as input the 3-dimensional acceleration signal from the left shin and the

Table 1 Series-to-series transformers. See Appendix A for a detailed description of each transformer

Derivative(x)	First-order derivative of each series
Resultant(x)	ℓ_2 -norm over all series at each time step
Average(x_1, x_2)	Series-wise mean
Difference(x_1, x_2)	Series-wise difference
Ratio(x_1, x_2)	Series-wise division
Angle(x_1, x_2, x_3)	Angle defined by three points at each time step

3-dimensional acceleration signal from the right shin and will compute the ratio of the left to the right acceleration in the (1) x , (2) y , and (3) z directions. The transformer operates in a series-wise fashion, meaning that the first series of the output is the ratio between the left shin's x -acceleration and the right shin's x -acceleration, the second series is the ratio between the y -accelerations, and the third series is the ratio between the z -accelerations. Hence the resulting collection has three signals and has a shape of 100×3 . Visually, a series-to-series node has one incoming edge for each of its inputs. These edges could originate from either a variable node or a series-to-series node. Table 1 shows the series-to-series transformers supported by TSFuse.

Series-to-attribute transformer $T_{S \rightarrow A}(x_1, x_2, \dots, x_c)$

This transformer converts its input time series collections $x_1, \dots, x_c$ into an attribute collection. For example, in Fig. 2, the Mean transformer extracts the mean value of the heart rate signal, and the FFT transformer computes the Fourier transform of the resultant of the lower back acceleration. For most series-to-attribute transformers, like the Mean transformer, the output has shape $1 \times d$. However, transformers that compute coefficients, such as the FFT transformer, have

Table 2 Series-to-attribute transformers. See Appendix A for a detailed description of each transformer

Mean(x)	Mean
Median(x)	Median
Variance(x)	Variance
StandardDeviation(x)	Standard deviation
Skewness(x)	Skewness
Kurtosis(x)	Kurtosis
Min(x)	Minimal value
Max(x)	Maximal value
Sum(x)	Sum of values
Length(x)	Number of time steps
Energy(x)	Sum of squared values
C3(x)	Non-linearity measure as defined in Schreiber and Schmitz (1997)
CID(x)	Complexity estimate as defined in Batista et al. (2014)
CountAboveMean(x)	Number of values greater than the mean
CountBelowMean(x)	Number of values smaller than the mean
EnergyRatio(x)	Relative sum of squares per chunk
BinnedDistribution(x)	Percentage of values in each bin
BinnedEntropy(x)	Entropy of the binned distribution
Quantile(x , $q=q$)	q -quantile, $q \in \{0.1, 0.2, \dots, 0.9\}$
AggregatedLinearTrend(x , size= s)	Linear trend for downsampled series, $s \in \{5, 10, 20\}$
FFT(x)	Fourier transform coefficients
CWT(x)	Wavelet transform coefficients
AutoRegressiveCoefficients(x)	Coefficients of an auto-regressive process
FriedrichCoefficients(x)	Coefficients of a polynomial as described in (Friedrich et al. 2000)
AutoCorrelation(x)	Auto-correlation coefficients
CrossCorrelation(x_1, x_2)	Cross-correlation coefficients

Table 3 Attribute-to-attribute transformers. See Appendix A for a detailed description of each transformer

Average(x_1, x_2)	Attribute-wise mean
Difference(x_1, x_2)	Attribute-wise difference
Ratio(x_1, x_2)	Attribute-wise division

Attribute-to-attribute transformer $T_{A \rightarrow A}(x_1, x_2)$

an output of shape $a \times d$ where a is the number of extracted coefficients. Table 2 lists the series-to-attribute transformers supported by TSFuse. This transformer fuses together two input attribute collections to create a new attribute collection. For example, the Difference(x_1, x_2) node in Fig. 2 computes the difference between the attributes extracted by the Min and Max transformers. Table 3 lists the attribute-to-attribute transformers supported by TSFuse.

Slice(x, d, t)

Output

This node reduces the dimensionality of a given collection x by selecting one specific time series or attribute, or one specific time step in each time series. Specifically, it selects the d^{th} time series or attribute of the given collection, and the t^{th} time step if the given collection is a time series collection. For example, in Fig. 2, the *Slice*($d=3$) node selects the z -acceleration of the right shin.

The graph contains a single output node. This node can only have incoming edges from series-to-attribute or attribute-to-attribute transformers. For example, in Fig. 2, the output node has three incoming edges, each having shape 1×1 . Hence, this graph results in a feature vector of length three.

The edges in the graph are annotated with three pieces of information. First, an edge lists the shape of the data it propagates. Second, an edge is labeled as propagating either a time series collection or an attribute collection. Finally, recall from Sect. 2 that each time series collection is annotated with tags that represent meta data about the collection. As these tags play a role in restricting the search space, the edges also must propagate this information. Edges from variable nodes simply are assigned the tags from the time series collection they represent. Edges originating from transformer nodes are annotated with the intersection of the tags from its incoming edges. For the example computation graph shown in Fig. 2, the edges have the following tags:

$$\begin{aligned} M_{\rightarrow V_1} &= M_{x_1} = \{\text{type} \mapsto \text{heart-rate}; \text{location} \mapsto \text{chest}\} \\ M_{\rightarrow V_2} &= M_{x_2} = \{\text{type} \mapsto \text{accelerometer}; \text{location} \mapsto \text{shin}; \text{side} \mapsto \text{left}\} \\ M_{\rightarrow V_3} &= M_{x_3} = \{\text{type} \mapsto \text{accelerometer}; \text{location} \mapsto \text{shin}; \text{side} \mapsto \text{right}\} \\ M_{\rightarrow V_4} &= M_{x_4} = \{\text{type} \mapsto \text{accelerometer}; \text{location} \mapsto \text{back}\} \\ M_{V_1 \rightarrow \text{Mean}} &= M_{\rightarrow V_1} = \{\text{type} \mapsto \text{heart-rate}; \text{location} \mapsto \text{chest}\} \\ M_{V_2 \rightarrow \text{Resultant}} &= M_{\rightarrow V_2} = \{\text{type} \mapsto \text{accelerometer}; \text{location} \mapsto \text{shin}; \text{side} \mapsto \text{left}\} \\ M_{(V_3, V_4) \rightarrow \text{Ratio}} &= M_{\rightarrow V_3} \cap M_{\rightarrow V_4} = \{\text{type} \mapsto \text{accelerometer}; \text{location} \mapsto \text{shin}\} \\ &\text{etc.} \end{aligned}$$

When computing the output of a transformer, the tags assigned to the transformer's outgoing edge are also assigned to the output data. For example, when applying the Mean transformer to input x_1 , the output $x = \text{Mean}(x_1)$ will have the tags $M_x = M_{V_1 \rightarrow \text{Mean}} = \{\text{type} \mapsto \text{heart-rate}; \text{location} \mapsto \text{chest}\}$.

3.2 Constraints

When constructing a computation graph, following a brute-force approach that generates all possible transformers can become prohibitively expensive due to the number of combinations that exist when combining multiple time series or attributes. Moreover, it is

unlikely that all generated transformers result in meaningful features. Therefore, our search procedure uses constraints to remove meaningless features. Specifically, we consider two kinds of constraints:

Precondition constraints Not all time series collections are suitable inputs for each transformer. Therefore, we use preconditions to define which inputs are allowed. We define the precondition p_T for each transformer T separately. For example, the Ratio transformer has the following precondition:

$$p_{\text{Ratio}}(x_1, x_2) = \text{true if } (x_2 \text{ is non-zero}) \text{ else false}$$

Compatibility constraints For transformers that have more than one input, the user can provide a compatibility relation r to specify which combinations of inputs are allowed. This relation defines all pairs of nodes that can be used together as the input nodes of a transformer. For the time series collections in Fig. 1, one could for example define the following compatibility relation:

$$r(x_1, x_2) = \text{true if } (\text{location of } x_1 = \text{location of } x_2) \text{ else false}$$

When adding a transformer to the computation graph, the constraints are used to check whether the transformer's argument list is valid. Algorithm 1 shows the procedure that checks whether a given argument list is valid with respect to the transformer's precondition and the compatibility relation. For the compatibility constraint, each argument should be compatible with at least one other argument. Note that the compatibility relation is only checked when the number of arguments is at least two.

Algorithm 1: Check whether an argument list is valid

Input: Argument list $(x_1, \dots, x_m)$, precondition p and compatibility relation r

Output: True if the argument list is valid with respect to p and r

```

is_valid( $(x_1, \dots, x_m), p, r$ ):
    if  $\neg p(x_1, \dots, x_m)$  then
        return false
    if  $m \geq 2$  then
        for  $i \in \{1, \dots, m\}$  do
            if  $\neg \exists j \in \{1, \dots, m\} \setminus \{i\} : r(x_i, x_j) = \text{true}$  then
                return false
    return true
  
```

3.3 Search procedure

The key challenge for designing a search procedure to find a good computation graph is that there is a huge space of possible computation graphs. This arises because each addition

Depth $d = 0$ Variable V_1 Variable V_2 Variable V_3 Variable V_4

Fig. 3 Initial computation graph with one variable node V_i per collection x_i

of a new transformer to the graph enables multiple new subsequent additions. This is compounded by the fact that not all transformers are symmetric with respect to their argument lists. For example, the series-to-series transformers $\text{Ratio}(x_1, x_2)$ and $\text{Ratio}(x_2, x_1)$ output different time series collections.

To cope with the large search space, we carefully define our search algorithm. It begins by creating a computation graph G that contains one variable node V_i for each time series collection x_i , as shown in Fig. 3 for our illustrative example. To ensure a finite search space, we only consider computation graphs that include nodes of at most depth D , where the depth of a node v is the maximum number of transformers on any path starting at a variable node and ending at node v . We subdivide the search into four separate steps that are executed in sequence:

- Step 1:* Add series-to-series transformers to G
- Step 2:* Add series-to-attribute transformers to G
- Step 3:* Add attribute-to-attribute transformers to G
- Step 4:* Select series-to-attribute and attribute-to-attribute transformers from G

Within each step, we employ a combination of techniques to further restrict the search space, including (1) exploiting constraints to limit which transformations are added, (2) pruning transformations that yield similar outputs, and (3) subsampling the considered transformations. Next, we describe each step in detail.

3.3.1 Step 1: Add series-to-series transformers

The first step adds series-to-series transformers to the computation graph. We add these transformers in two phases. First, the *generate* phase adds all transformers that satisfy the constraints. Second, the *select* phase removes transformers that result in redundant time series collections being constructed, which restricts the number of possible additions in the subsequent steps and hence improves the computational efficiency. Algorithm 2 shows the pseudo-code of this step. The two phases are described in detail below.

3.3.1.1 Phase 1: Generate The first phase generates series-to-series transformers by adding nodes selected from Table 1 to the computation graph and connecting their input arguments to other nodes of the graph. The input nodes can be variable nodes or series-to-series transformers. Since these nodes must already be present in the computation graph, the generation process follows a level-wise approach where transformers at depth d are only generated once all nodes at depth $d - 1$ are added to the graph. The process terminates at depth $D - 1$, because a series-to-series transformer is only useful if it is connected to at least one series-to-attribute transformer.

At each depth d , we generate all valid series-to-series transformers with an incoming edge originating from a node with a depth less than d . A transformer is valid if it respects the provided compatibility relation and satisfies the transformer's preconditions (Algorithm 1). A complicating factor is that most series-to-series transformers are not symmetric with respect to their arguments. This necessitates generating all permutations of each transformer's argument list. To avoid generating duplicate nodes, we require that each argument list contains an input argument originating from a node at depth $d - 1$. To ensure that the number of transformer nodes that can be constructed in this manner does not grow exponentially as the depth increases, we randomly sample n_s possible argument lists for each series-to-series transformer.

3.3.1.2 Phase 2: Select The first phase will likely result in nodes being added to G that produce highly similar time series collections. This redundancy is undesirable for two reasons. On the one hand, it increases the search space for the subsequent steps, which increases the computational cost of the search. On the other hand, attributes extracted from these series will be redundant as well, which can pose problems for learners.

To identify potentially redundant nodes, for all pairs of nodes $v, w \in G$, we compute the Pearson correlation for six summary statistics (minimum, maximum, mean, variance, skewness, and kurtosis) from the time series collections emitted by these nodes.³ If the absolute value of the Pearson correlation for at least one of these six statistics larger than r , then v and w are considered to be redundant and we add the node with the largest depth to R , which is a list containing candidate nodes for removal. We then process the nodes in R in decreasing depth order. At depth d , we remove each node $v \in R$ that has no outgoing edges. In this way, we do not remove a redundant node v if v is used as an input to a non-redundant node.

Figure 4 shows a possible state of the computation graph after step 1. The graph contains four variables nodes: none of them were removed as all variable nodes were found to be either non-redundant or used as input for a non-redundant series-to-series transformer. The variable nodes V_1 and V_3 , the Resultant node with input argument V_2 , and the Ratio node with input arguments (V_3, V_4) pass the redundancy test and will be used when adding series-to-attribute transformers in the next step.

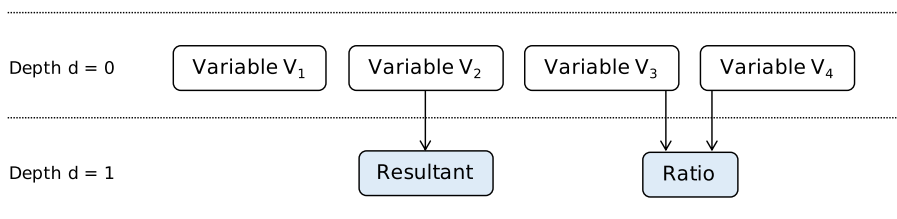


Fig. 4 Computation graph after step 1

³ All these statistics are computed efficiently using only one pass over the data.

Algorithm 2: Add series-to-series transformers to G

Input:

- Compatibility relation r
- Computation graph G initialized with variable nodes

Parameters:

- Series-to-series transformers S
- Maximum computation graph depth D
- Maximum number of series permutations n_s
- Maximum correlation coefficient c

Output: G extended with series-to-series transformers

PHASE 1: GENERATE

```

for depth  $d \in \{1, \dots, D - 1\}$  do
   $V$  = all variable nodes and series-to-series transformers currently in  $G$ 
  for transformer  $T \in S$  do
     $m$  = number of arguments of  $T$ 
     $p$  = preconditions of  $T$ 
    for nodes  $(v_1, \dots, v_m) \in \text{generate\_permutations}(V, d, m, n_s)$  do
       $x_i$  = output of node  $v_i, \forall i \in \{1, \dots, m\}$ 
      if  $\text{is\_valid}((x_1, \dots, x_m), p, r)$  then
        | add node  $T$  and edges  $v_i \rightarrow T$  to  $G$ 

```

PHASE 2: SELECT

```

 $R = \emptyset$ 
 $V$  = all variable nodes and series-to-series transformers currently in  $G$ 
sort  $V$  by increasing depth and randomly shuffle the nodes in  $V$  per depth
for node  $v \in V$  do
  remove  $v$  from  $V$ 
  for node  $w \in V$  do
    | if  $\text{is\_redundant}(v, w, c)$  then
    | | add  $w$  to  $R$ 
sort  $R$  by decreasing depth
for node  $v \in R$  do
  | if  $v$  has no outgoing edges then
  | | remove  $v$  and all its incoming edges from  $G$ 

```

3.3.2 Step 2: Add series-to-attribute transformers

Since the goal is to construct an attribute-value representation, we need to add series-to-attribute nodes to extract attributes from the series, both those from variable nodes and those generated by series-to-series transformers.

Algorithm 3 shows the pseudo-code for this step. First, we select each non-redundant node $v \in G$ (i.e., $v \notin R$). Second, we convert each multivariate time series returned by these nodes to a univariate time series, i.e., a time series collection of shape $t \times 1$, using

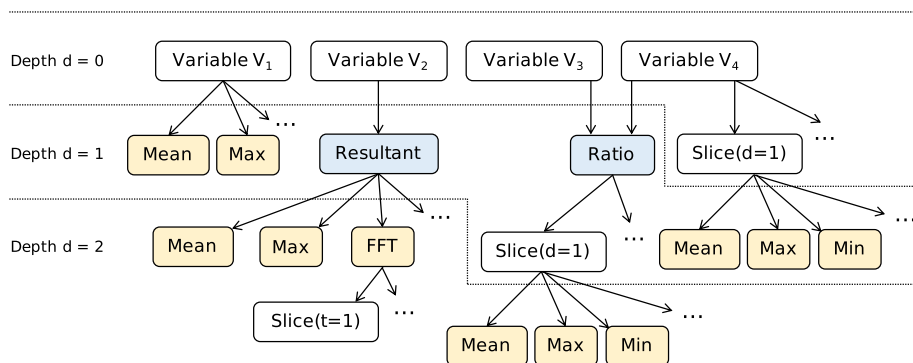


Fig. 5 Computation graph after step 2

$\text{Slice}(d=d)$ nodes which select the d^{th} dimension of a time series collection. Third, we generate all valid permutations of nodes to use as the input arguments of the series-to-attribute transformers. Fourth, we select a percentage q of the largest coefficients for transformers that return coefficients, since these transformers would otherwise result in too many features. Since we do not know yet which features will become important after applying the attribute-to-attribute transformers in step 3, we simply assume that the largest coefficients will be the most important.

Figure 5 continues our illustrative example and shows the computation graph (where many nodes must be omitted for readability) after step 2 of the search. Since variable V_4 and the Ratio node return a 3-dimensional time series collection, each of these nodes is connected to three Slice nodes: $\text{Slice}(d=1)$, $\text{Slice}(d=2)$ and $\text{Slice}(d=3)$. Each non-redundant variable, series-to-series transformer, or Slice node has the same series-to-attribute transformers.

Algorithm 3: Add series-to-attribute transformers to G **Input:**

- Compatibility relation r
- Redundant nodes R found in step 1
- Computation graph G returned by step 1

Parameters:

- Series-to-attribute transformers S
- Maximum number of series permutations n_s
- Percentage of coefficients q

Output: G extended with series-to-attribute transformers

```

 $V$  = all variable nodes and series-to-series transformers in  $G$  but not in  $R$ 
 $V_{1d} = \emptyset$ 
for node  $v \in V$  do
     $d$  = number of dimensions of the output of  $v$ 
    if  $d > 1$  then
        for  $i \in \{1, \dots, d\}$  do
             $v_i = \text{Slice}(d=i)$ 
            add node  $v_i$  to  $V_{1d}$  and  $G$ 
            add edge  $v \rightarrow v_i$  to  $G$ 
        else
            add node  $v$  to  $V_{1d}$ 
for transformer  $T \in S$  do
     $m$  = number of arguments of  $T$ 
     $p$  = preconditions of  $T$ 
    for nodes  $(v_1, \dots, v_m) \in \text{generate\_permutations}(V_{1d}, d, m, n_s)$  do
         $x_i$  = output of node  $v_i$ 
        if  $\text{is\_valid}((x_1, \dots, x_m), p, r)$  then
            add node  $T$  and edges  $v_i \rightarrow T$  to  $G$ 
             $x$  = output of  $T$ 
            if  $T$  returns coefficients then
                for  $i \in \text{largest\_coefficients}(x, q)$  do
                     $T_i = \text{Slice}(t=i)$ 
                    add node  $T_i$  and edge  $T \rightarrow T_i$  to  $G$ 

```

3.3.3 Step 3: Add attribute-to-attribute transformers

Similar to the *generate* phase of step 1, we add all attribute-to-attribute transformers in a level-wise order. However, we now stop at depth D instead of $D - 1$, because an attribute-to-attribute transformer can be directly connected to the output node. We use the transformers listed in Table 3. At each depth d , we again generate all valid attribute-to-attribute transformers with incoming edges from the nodes with a depth less than d . We also take a random subsample of n_a argument lists for each transformer to make sure that the number of added transformers does not grow exponentially for increasing depths. The pseudo-code of this step is shown in Algorithm 4.

Since the number of generated attribute-to-attribute transformers can be much larger than the number of series-to-series transformers, we apply an additional selection step for the generated permutations at each depth. The challenge is to reduce the number of permutations without removing those that could result in relevant features. Related work typically looks for interacting attributes when exploring pairs of attributes (Lou et al., 2013; van den Bosch, 2017). We follow a similar approach and select the transformers for which the

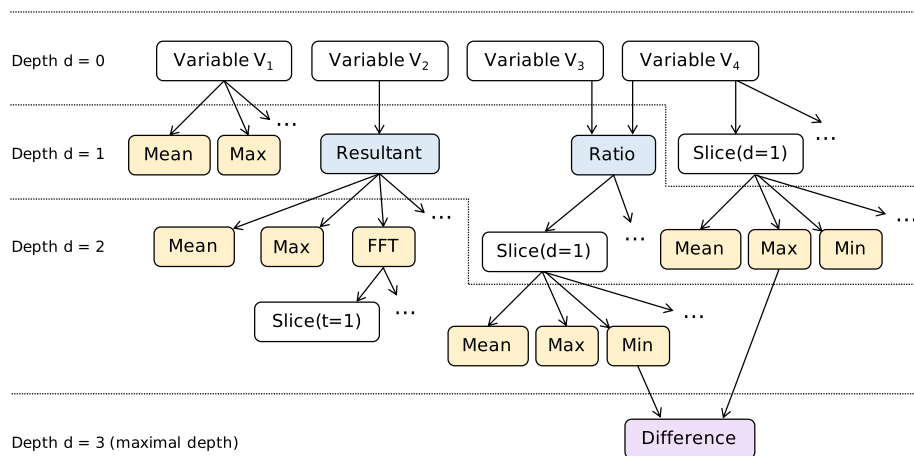


Fig. 6 Computation graph after step 3, where the Difference node is the only attribute-to-attribute transformer that was added

inputs have interacting attributes. Specifically, we compute the H -statistic (Friedman et al., 2008) of the attributes to estimate their interaction strength. For a given pair of attributes (x_j, x_k) , this statistic is computed as follows:

$$H_{jk}^2 = \frac{\sum_{i=1}^N [\hat{F}_{jk}(x_{ij}, x_{ik}) - \hat{F}_j(x_{ij}) - \hat{F}_k(x_{ik})]^2}{\sum_{i=1}^N \hat{F}_{jk}^2(x_{ij}, x_{ik})}$$

where i refers to the instance in the dataset and N is the number of instances (i.e., the number of windows). $\hat{F}_s$ is the estimated partial dependence of a gradient boosted tree model $F(x)$ as defined in Friedman et al. (2008). If H_{jk}^2 is zero, attributes x_j and x_k do not interact. Larger values correspond to a stronger interaction. We select all permutations that have a H -statistic larger than a threshold h .

Figure 6 shows the new possible state of the computation graph. In our example, only one attribute-to-attribute transformer was added to the graph after applying the interaction-based selection.

Algorithm 4: Add attribute-to-attribute transformers to G **Input:**

- Labels y
- Compatibility relation r
- Computation graph G returned by step 2

Parameters:

- Attribute-to-attribute transformers S
- Maximum computation graph depth D
- Maximum number of attribute permutations n_a
- Attribute interaction threshold h

Output: G extended with attribute-to-attribute transformers

```

for depth  $d \in \{2, \dots, D\}$  do
     $V =$  nodes from  $G$  with a depth less than  $d$  that return an attribute collection
    for transformer  $T \in S$  do
         $p =$  preconditions of  $T$ 
        for nodes  $(v_1, \dots, v_m) \in \text{generate\_permutations}(V, d, 2, n_a)$  do
             $x_i =$  output of node  $v_i, \forall i \in \{1, \dots, m\}$ 
            if  $\text{is\_valid}((x_1, \dots, x_m), p, r)$  then
                if  $\text{interaction\_strength}(x_1, x_2, y) > h$  then
                    add node  $T$  and edges  $v_i \rightarrow T$  to  $G$ 

```

3.3.4 Step 4: Select series-to-attribute and attribute-to-attribute transformers

After step 3, G may still result in a feature vector representation prone to overfitting for two reasons. First, steps 2 and 3 typically construct a large number of attributes, often more than the number of windows in the dataset. Second, there will likely be many attributes that are quite similar to each other (e.g., the various FFT coefficients) which increases the correlation among the attributes. Both these factors increase the risk of overfitting. Therefore, our final step focuses on removing a subset of the series-to-attribute and attribute-to-attribute transformers.

Algorithm 5 shows the pseudo-code for this step. Our goal is to select a diverse set of attributes. To do so, we first group the series-to-attribute and attribute-to-attribute transformers in G by similarity, and then select a small number of nodes per group as follows:

1. *Group similar nodes* We exploit the computation graph structure to find similar attributes. Intuitively, attributes computed using the same transformers are likely to be similar to each other. Therefore, we group the nodes based on the similarity of their parent nodes. For each series-to-attribute and attribute-to-attribute node v in the computation graph, we select the sub-graph that consists of v itself, each parent node v' of v , each parent node v'' of each parent node v' of v , etc. For variable nodes, we remove the name of the variable (e.g., V_1). We then group all nodes that have the same sub-graph. For example, the sub-graph for the Slice($t=1$) node at depth 2 of the computation graph shown in Fig. 6 is Variable $\rightarrow$ Resultant $\rightarrow$ FFT $\rightarrow$ Slice.
2. *Select nodes per group* We use a model-based or *wrapper* feature selection approach (Chandrashekar & Sahin, 2014) to select attributes. For each group, we train a decision tree that predicts the label y of each window using the attributes extracted by the transformers in the group. Since decision trees avoid making redundant splits, the selected attributes are likely to be non-redundant. Only those transformers that produced

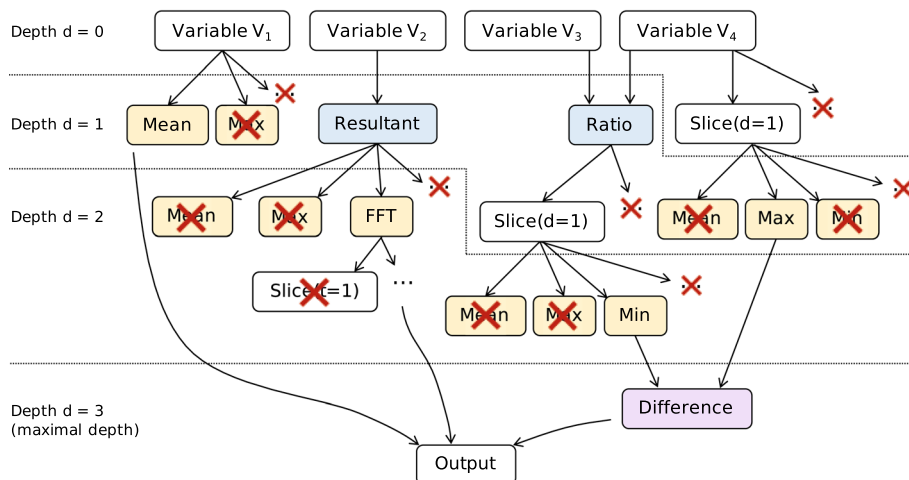


Fig. 7 Computation graph after step 4

a feature used as split criterion in the decision tree are retained in G . These transformers are connected to the output node. All other transformers in the group are removed.

Figure 7 shows a potential state of the computation graph after selecting a subset of the series-to-attribute and attribute-to-attribute transformers. The nodes that are crossed out are removed. The incoming edges connected to these transformers are removed as well. In this example, step 4 removed a large number of nodes. For example, for the group Variable $\rightarrow$ Mean, only the Mean transformer for V_1 was selected. It is also possible that no transformers are selected for a group. For example, this is the case for the group Variable $\rightarrow$ Median, where all attributes in the group are irrelevant with respect to the label y of the windows. Finally, all remaining series-to-attribute and attribute-to-attribute transformers are connected to the output node.

Algorithm 5: Select series- and attribute-to-attribute transformers

Input:

- Labels y
- Computation graph G returned by step 3

Output: G with less series-to-attribute and attribute-to-attribute transformers V = all nodes from G that return an attribute collection

```

for group  $g \in \text{group\_similar\_nodes}(V)$  do
   $V_g$  = all nodes that belong to group  $g$ 
   $X$  = attribute returned by the nodes in  $V_g$ 
   $m$  = decision tree model trained on  $(X, y)$ 
  for node  $v \in V_g$  do
    if output of  $v$  is used in  $m$  then
      add an edge  $v \rightarrow \text{Output}$  to  $G$ 
for node  $v \in V$  do
  if  $v$  has no path to the Output node then
    remove  $v$  and all its incoming edges from  $G$ 
  
```

3.4 Time complexity

We now analyze the time complexity of our system. The complexity depends on the number of time series, which includes the series generated via fusion by our system. Therefore, we first define an upper bound M for sum of the input time series and generated time series:

$$O(n \cdot d + D \cdot |S_{s \rightarrow s}| \cdot n_s)$$

where

- n is the number of input time series collections;
- d is the largest number of series in the input collections;
- D is the maximum computation graph depth;
- $|S_{s \rightarrow s}|$ is the number of different series-to-series transformers;
- n_s is the maximum number of nodes added for each series-to-series transformer per depth.

Per window, the computational cost depends on the total number of constructed attributes, i.e., the number of series-to-attribute nodes and attribute-to-attribute nodes that are connected to the output node of the computation graph. Given N windows, the worst-case behaviour of the algorithm is as follows:

$$O(N \cdot (M \cdot |S_{s \rightarrow a}| + D \cdot |S_{a \rightarrow a}| \cdot n_a))$$

where

- $|S_{s \rightarrow a}|$ is the number of series-to-attribute transformers;
- $|S_{a \rightarrow a}|$ is the number of attribute-to-attribute transformers;
- n_a is the maximum number of nodes added for each attribute-to-attribute transformer per depth.

4 Related work

Within data science, it is recognized that having the *right representation* of the data is crucial in order to be able to learn a performant model. Moreover, coming up with this representation is an extremely complicated and time-consuming manual process (Munson, 2012), which often requires expert knowledge about the application domain. The goal of automated feature construction is to reduce this burden and it has long been recognized as an important problem. The original work dates from the 1980s when it was known as constructive induction (c.f. Rendell (1985)). This original work, as well as recent follow-up work (Khurana et al., 2018; Horn et al., 2019) focused on propositional data. Automated feature construction from relational data has also been studied, where it is often called *propositionalization* (Davis et al., 2005; Krogel, 2005; Kanter & Veeramachaneni, 2015; Lam et al., 2017; Boullé et al., 2019; Schouterden et al., 2019). These systems construct features by transforming a relational dataset to an attribute-value representation.

Constructing features from time series requires fundamentally different transformations than for either propositional or relational data. Hence, there exist approaches specifically designed for time series data (Fulcher & Jones, 2017; Christ et al., 2018). These systems typically extract a large set of features such as the mean, variance, kurtosis, Fourier transform coefficients, auto-correlation, etc. Currently, tsfresh (Christ et al., 2018) is one of the most prominent approaches. It extracts a set of 794 features from each time series. After computing these features, the system uses the FRESH algorithm (FeatuRe Extraction based on Scalable Hypothesis tests) to select the most relevant ones. This algorithm performs multiple hypothesis tests to measure the dependency between each feature and the target data, and selects a subset of the features according to a given false discovery rate. Another popular system is hctsa (Fulcher & Jones, 2017). This system extracts a similar set of features as tsfresh, but uses a different selection method. Instead of hypothesis tests, it applies a forward sequential selection approach using a linear model. Compared to our system, tsfresh and hctsa extract a more extensive set of features. However, these features only consist of *series-to-attribute* transformations. The series-to-attribute transformers of TSFuse are a subset of the tsfresh features: we consider approximately half of the features that tsfresh considers (i.e., TSFuse implements 25 out of 63 functions available in tsfresh). However, TSFuse also considers series-to-series and attribute-to-attribute transformers. These transformers can be used to define new features when combining multiple transformers, which allows our system to extract a larger number of features compared to tsfresh.

The existing feature construction systems for time series data are limited to a pre-defined list of features. Hence, these systems are not able to construct new operations. To overcome these limitations, some studies apply genetic programming to construct features from time series data (Mierswa, 2005; Schuller et al., 2006; Harvey & Todd, 2014). Unfortunately, there exist no publicly available implementations for these studies. The goal of genetic programming is to define features by creating expression trees and applying genetic operations: mutation (changing a part of an expression tree) and cross-over (combining two expression trees to form a new expression tree). The expression trees are similar to our computation graph representation. In (Harvey and Todd, 2014), the operations used for building the expression trees consist of various mathematical functions, such as the derivative of a time series, the square root of a time series, etc. However, while these operations can create new time series and attributes, they do not consider combinations of multiple series and hence cannot take the ratio of two series or the resultant. Another limitation of genetic programming is its a high

computational cost, as evaluating the fitness of the expression trees requires computing a large number of features. For example, (Harvey & Todd, 2014) reported run times that are in the order of hours, even for small datasets, where 80% to 90% was spent on feature evaluation.

Features can also be learned using deep neural networks. The advantage of this approach is that the learned features can combine multiple series. In recent years, a variety of architectures have been developed (Fawaz et al., 2019), e.g., convolutional neural networks (CNN) (Fukushima et al., 1983) and recurrent neural networks (RNN) (Hochreiter & Schmidhuber, 1997; Jaeger & Haas, 2004). The main drawback is that deep learning still requires considerable human effort and expertise for choosing an appropriate architecture. In addition, training neural networks requires large amounts of (labeled) examples, which may not be available. For example, health and sports applications often involve collecting data from human subjects. Typical limitations in these applications are the available time and personnel to collect the data, and the number of subjects that meet the selection criteria of the study. The available time can also be a limitation in other applications where the training examples need to be manually labeled by a human annotator.

5 Experimental evaluation

The primary motivation underlying our work is that fusion of multiple time series and/or attributes can result in a better feature representation compared to univariate time series analysis. Hence, our main research question is the following:

Q1: Does TSFuse’s ability to fuse series and attributes result in a better feature representation?

In addition, we address the following research questions:

Q2: How does TSFuse compare to existing feature construction methods?

Existing automated feature construction methods are only applicable to univariate time series and hence treat signals in a multivariate setting independently. We evaluate whether automatically fusing time series and attributes can result in a better feature representation compared to these methods.

Q3: How does TSFuse compare to operating directly on the raw time series?

We compare our system to two popular approaches that train a model on the raw time series data: dynamic time warping and deep learning.

Q4: How do the system’s parameters affect its performance?

We investigate how the different parameters of TSFuse affect the system’s predictive performance and computational cost.

Q5: How can we interpret the constructed features?

Since interpretability is an advantage of feature-based methods, we show how the computation graph representation of TSFuse helps to get a better understanding of the constructed features.

5.1 Datasets

For each research question, we run experiments on ten popular benchmark time series datasets. The properties of each dataset are shown in Table 4. This section summarizes the datasets, the associated machine learning task, and how we represent the dataset in TSFuse. Appendix B gives a more detailed description of each dataset.

5.1.1 Articulatory word recognition

The goal of this dataset is to predict the word pronounced by a person based on sensor data tracking the movement of the lower lip (LL), the upper lip (UL), and the tongue (T1). Each sensor records the position as (x, y, z) coordinates, resulting in a total of nine dimensions. Each window consists of 144 time steps.

Representation in TSFuse We group each sensor location as a three-dimensional time series collection. Since each collection is collected from the same type of sensor, we do not specify tags. The compatibility relation allows any combination of signals.

5.1.2 Cricket hand signals

This dataset consists of data of people performing cricket hand signals captured by tri-axial accelerometers on the right and left wrist (Ko et al., 2005). We use the data provided by Shokoohi-Yekta et al. (2015). Each window consists of 1197 time steps and corresponds to a single hand signal. The goal is to predict the performed hand signal from the accelerometer data.

Representation in TSFuse We have two time series collections, one per accelerometer, each with three dimensions. The compatibility relation allows any combination of signals.

Table 4 Dataset properties: number of windows (where a window is one instance of an activity, gesture, etc.), time series length (i.e., the number of time steps per window), and number of dimensions (i.e., the number of time series per window)

Dataset	Windows	Length	Dims
1 Articulatory word recognition	575	144	9
2 Cricket hand signal classification	180	1197	6
3 Handwriting character classification	1000	152	6
4 Accelerometer-based gesture recognition	440	315	3
5 Camera-based gesture recognition	360	51	24
6 Atrial fibrillation detection	30	7680	2
7 Machine condition monitoring	226	7–8	3
8 Human activity recognition	840	500	6
9 Indoor exercise monitoring	1791	31–308	75
10 Stock price forecasting	5010	30	4

5.1.3 Handwriting character classification

This dataset was collected using a tri-axial accelerometer and a tri-axial gyroscope on a person's wrist (Shokoohi-Yekta et al., 2015). Each window contains 152 time steps and the task is to detect the character written by the person.

Representation in TSFuse We represent the data as two time series collections: one for the accelerometer with a tag {type $\mapsto$ accelerometer} and another one for the gyroscope with a tag {type $\mapsto$ gyroscope}. We impose a compatibility relation stating that only data from the same sensor type can be combined:

$$r(x_i, x_j) = \text{true if } (\text{type of } x_i = \text{type of } x_j) \text{ else false}$$

5.1.4 Accelerometer-based gesture recognition

This dataset recorded the hand movements of eight gestures using a tri-axial accelerometer (Liu et al., 2009). We use the data from Shokoohi-Yekta et al. (2015). Given the accelerometer data of one window of 315 time steps, the goal is to predict the gesture.

Representation in TSFuse Since only one sensor was used, we represent each axis as a one-dimensional time series collection. The compatibility relation allows any combination of signals.

5.1.5 Camera-based gesture recognition

This dataset recorded the 3D coordinates of a person's body parts using a camera tracking system (Ghouaiei et al., 2017). Given a window of 51 time steps of the positions of eight body parts (left and right hand tips, thumbs, wrists, and elbows) of an aircraft marshaller, the goal is to predict the gesture that was performed.

Representation in TSFuse There are eight time series collections, one per location. Each collection has three dimensions and is assigned a 'location' tag specifying the body part (i.e., left elbow, right elbow, left wrist, etc.). The compatibility relation only allows pairs of collections that have the exact same 'location' tag, have the same kind of body part (e.g., left and right elbow), or are connected by a limb.

5.1.6 Atrial fibrillation detection

This dataset consists of one minute long recordings from a two-channel ECG data recorded at 128Hz, yielding two dimensions and 7680 time steps per recording (Moody, 2004; Goldberger et al., 2000). As opposed to the UEA benchmark version of this dataset (Bagnall et al., 2018), we do not subsample the data but instead create windows of one minute that each contain a complete recording. Given the data of a single recording, the task is to predict if the episode (i) does not terminate, (ii) terminates after more than one minute after the end of the given recording, or (iii) terminates immediately (i.e., in less than one second) after the recording.

Representation in TSFuse We represent the two-channel ECG data as two one-dimensional time series collections. The compatibility relation allows any combination.

5.1.7 Machine condition monitoring

In this dataset, the goal is to detect an electrical failure of a three-phase motor (Hendrickx et al., 2020). The three-phase current amplitude was recorded during linear run-up experiments of 20 seconds. These experiments were split into windows of one second. Given the current of a one-second window, the task is to predict the condition of the motor.

Representation in TSFuse There are three one-dimensional time series collections, where each collection represents the current amplitude of one of the three phases. The compatibility relation allows any combination of collections.

5.1.8 Human activity recognition

Detecting a person's daily life activities based on one or more wearable sensors is a widely studied time series classification task. Here, we use the USC Human Activity Database (HAD) (Zhang & Sawchuk, 2012). The data was collected by an accelerometer and gyroscope attached to a person's hip. Each participant performed multiple trials. We selected the first ten seconds of each trial, such that each window consists of 500 time steps. Given one window of accelerometer and gyroscope data, the task is to predict the activity performed by the person. When deploying an activity recognition system, users can attach the sensors in different ways than intended (Yurtman & Barshan, 2017). To evaluate TSFuse's robustness to the sensor orientation, we randomly rotate the sensor data and consider three variations of the activity recognition dataset: (a) the original *HAD* dataset, (b) the *HAD* dataset rotated with a small perturbation, and (c) the *HAD* dataset rotated with a large perturbation.

Representation in TSFuse We represent the data with two time series collections: one for the accelerometer and one for the gyroscope. We assign a tag 'type' to these collections to specify the sensor type (i.e., accelerometer or gyroscope) and use the following compatibility relation:

$$r(x_i, x_j) = \text{true if } (\text{type of } x_i = \text{type of } x_j) \text{ else false}$$

5.1.9 Indoor exercise monitoring

Monitoring physical therapy exercises can improve the quality of the performed exercises by giving corrective feedback to the patient. This dataset used a Microsoft Kinect camera to capture the positions of 25 joints of a person's body during the execution of different physical therapy exercises (Decroos et al., 2018). Given the joint positions recorded during one exercise repetition, i.e., one window of 31–308 time steps, the task is to predict: (a) the type of exercise, (b) the correctness of the exercise, and (c) the type of mistake that was made.

Representation in TSFuse We represent the data as 25 time series collections, one for each joint, where each collection has three dimensions to represent the 3D coordinates. For each collection, we assign a 'location' tag that specifies the location of the joint. The compatibility relation only includes all pairs of joint locations that are connected to each other directly via a limb.

5.1.10 Stock price forecasting

The goal of stock price forecasting is to estimate the next day's stock price based on historical data. We selected the data of the Dow Jones Industrial Average (*DJI*) from January 1, 2017 until December 31, 2019. We follow an approach similar to Guo et al. (2014) which uses a sliding window approach and splits the data in windows of 30 days with a step size of one day. The task is to predict the closing price on a given day, based on the data of the past 30 days.

Representation in TSFuse We represent the data using four one-dimensional time series collections: the opening price, closing price, highest price, and lowest price. The compatibility relation allows any combination.

5.2 Experimental details

5.2.1 Parameters of TSFuse

Algorithms 2–5 have introduced several parameters. In the experiments for questions Q1, Q2, Q3, and Q5, we use the same settings. These default settings are shown in Table 5. For research question Q4, we analyze the effect of each parameter on these system's performance.

5.2.2 Feature-based methods

TSFuse and all feature-based baselines only return a set of features. Evaluating these methods requires training a model that predicts the target label based on the extracted features. Since the results depend on the model, we evaluate three different classification/regression methods:

- Nearest neighbors
- Logistic/linear regression
- Gradient boosted decision trees

We use the scikit-learn (Pedregosa et al., 2011) implementation for nearest neighbors and logistic/linear regression. For nearest neighbors, we use the default hyperparameter settings of scikit-learn where the number of neighbors k is 5. For logistic regression, we employ L2 regularization and tune the regularization strength using a grid search over $C \in \{0.001, 0.01, \dots, 1000\}$. For linear regression, we use

Table 5 Default parameter settings of TSFuse

Maximum computation graph depth D	3
Maximum number of series permutations n_s	10
Maximum number of attribute permutations n_a	1000
Maximum correlation coefficient c	0.9
Percentage of coefficients q	0.1
Attribute interaction threshold h	0.1

ElasticNet which applies both L1 and L2 regularization. We perform a grid search over the regularization strength ($\alpha \in \{0.001, 0.01, \dots, 1000\}$) and the mixing parameter ($\text{ll_ratio} \in \{0.1, 0.5, 0.7, 0.9, 0.95, 0.99, 1\}$).

For the gradient boosted decision trees, we use the XGBoost (Chen & Guestrin, 2016) implementation with hyperparameter tuning for the most important parameters. Specifically, we first find the three best configurations for the number of trees ($\text{n_estimators} \in \{100, 200, 500\}$), the learning rate ($\text{learning_rate} \in \{0.01, 0.1, 0.2, 0.3\}$ where 0.01 is only included when $\text{n_estimators} = 500$), and the maximal decision tree depth ($\text{max_depth} \in \{4, 6, 8\}$). Then, we tune the instance subsampling parameter ($\text{subsample} \in \{0.5, 0.9, 1.0\}$) for each of these three configurations and select the best configuration. Finally, we tune the feature subsampling parameter ($\text{colsample_bylevel} \in \{0.5, 0.9, 1.0\}$) and select the final best configuration.

5.2.3 Dynamic time warping

Typically, the DTW distance is combined with a 1-nearest neighbor model which returns the label of the most similar instance in the training set. We again use the implementation of scikit-learn and set the number of neighbors k to 1. To compute the DTW distance, we use the `dtaidistance` package (Meert et al., 2020). We compute the non-normalized dependent DTW, which was found to be suitable for multivariate time series benchmarks (Ruiz et al., 2020).

5.2.4 Deep learning

For the deep learning methods, we use the `sktime-dl` (Löning et al., 2019) package which implements all networks evaluated in a recent benchmark study that compares state-of-the-art architectures for multivariate time series classification (Fawaz et al., 2019). Due to the high computational cost of training these networks, we do not tune their hyperparameters and set the same hyperparameter settings as those used in this benchmark study.

5.2.5 Evaluation

To split the data in training and test data, we employ a five times repeated two-fold cross-validation ($5 \times 2cv$) procedure (Dietterich, 1998). For each method, we report the average predictive performance over the ten cross-validation splits. For the classification datasets, we evaluate the predictive performance of each method by reporting the area under the ROC curve (AUC). For the regression dataset, we report the coefficient of determination (R^2). For each dataset, we evaluate whether TSFuse significantly outperforms each baseline using the $5 \times 2cv$ paired t-test (Dietterich, 1998). In addition to the predictive performance, we evaluate the computational cost of each method. Specifically, we measure the feature construction time, model training time, and test time. The feature construction time is the time needed for generating, computing, and selecting features for the training data. The model training time is the run time for tuning the hyperparameters and training the final model with the best (or default) hyperparameters. The test time is the time needed to compute the constructed features for the test data, and to compute the predictions of the model.

5.3 Time series and attribute fusion (Q1)

One of our key claims is that fusion can be important when analyzing multivariate time series datasets. Therefore, we compare three variants of TSFuse:

1. TSFuse that fuses both series and attributes, i.e., it includes both series-to-series and attribute-to-attribute transformers;
2. TSFuse that just fuses series, i.e., it includes series-to-series transformers;
3. TSFuse without fusion.

For each variant, we evaluate the predictive performance by training a model on top of the constructed features. We only show the results for the XGBoost models for the classification datasets and for the ElasticNet models for the regression dataset as these models were the most accurate.

Table 6 shows how including the fusion operations in TSFuse affects the predictive performance of the model learned on top of the constructed feature representation compared to the TSFuse variant that does not include fusion. For most datasets, time series fusion improves performance, with an average relative gain of +3.16% that ranges from −8.74% on the stock price forecasting dataset (dataset 10) to +17.47% on the atrial fibrillation detection dataset (dataset 6). The gains from adding attribute fusion on top of series-to-series transformers are more mixed. On a small number of datasets (5 out of 14 cases) attribute fusion yields slight additional improvements. Therefore, time series

Table 6 Comparison of the predictive performance (AUC of the XGBoost models for datasets 1–9 and R^2 of the linear regression models for dataset 10) for three variants of TSFuse. For all variants with fusion, the table shows the difference with respect the setting to the right

Time series fusion	yes	yes	no
Attribute fusion	yes	no	no
1. Articular word recognition	0.000	0.000	0.998
2. Cricket hand signals	+0.001	0.000	0.996
3. Handwriting character classification	−0.001	−0.011	0.890
4. Accelerometer-based gesture recognition	0.000	+0.007	0.981
5. Camera-based gesture recognition	−0.001	+0.013	0.980
6. Atrial fibrillation detection	−0.013	+0.092	0.527
7. Machine condition monitoring	−0.002	+0.070	0.884
8a. Human activity recognition: original	−0.001	+0.003	0.988
8b. Human activity recognition: small rotation	+0.001	+0.006	0.961
8c. Human activity recognition: heavy rotation	−0.003	+0.022	0.935
9a. Indoor exercise monitoring: exercise	+0.001	+0.006	0.991
9b. Indoor exercise monitoring: correctness	+0.010	+0.104	0.739
9c. Indoor exercise monitoring: mistake type	+0.003	+0.067	0.787
10. Stock price forecasting	−1.197	−0.071	0.810

fusion seems more important than attribute fusion. In some cases, fusion can decrease the performance because generating more features increases the risk of overfitting.

5.4 Comparison to feature-based methods (Q2)

Next, we compare TSFuse to two existing feature-based methods:

tsfresh (Christ et al., 2018) extracts a variety of features commonly used in time series analysis and selects a subset of these features using the FRESH algorithm (Christ et al., 2016). The selection method uses a parameter to control the false discovery rate. We set this parameter to its default value (0.05) for all experiments. For some datasets, we noticed that no features were selected with the default false discovery rate. In this case, we simply selected all features.

catch22 (Lubba et al., 2019) extracts the same fixed set of 22 features for every dataset. These features have been found to be relevant across different time series classification problems.

For TSFuse and *tsfresh*, we compare two sets of transformations: the *minimal* setting, which only uses a small set of simple statistical features, and the *default* setting, which uses all available transformations. For *tsfresh*, the default setting uses the `ComprehensiveFCParameters` and the minimal set uses the `MinimalFCParameters`. Appendix A lists the considered transformations in each setting for TSFuse.

We learn a model on top of the constructed features and consider three different learners: nearest neighbours, logistic/linear regression, and gradient boosted decision trees. Table 7 shows the average ranks for each learner across all datasets. Regardless of the feature construction method, the gradient boosted trees ensemble work best for the classification task. For regression problems, linear regression tends to be the best. For readability, the following discussion focuses on these two methods and Appendix C contains the complete results.

5.4.1 Predictive performance

Table 8a shows the predictive performance in terms of the AUC for the classification datasets (1–9) and the R^2 for the regression dataset (10). For each baseline, we indicate whether the difference with TSFuse (with the default setting) is significant using the 5x2cv paired t-test.

Overall, TSFuse obtains the best predictive performance on 13 out of 14 datasets. None of the baselines significantly outperform TSFuse, which shows that our system is

Table 7 Average rank for the three learners for each feature-based method. GB = gradient boosted decision trees, LR = logistic/linear regression, NN = nearest neighbours

Method	Classification			Regression		
	GB	LR	NN	GB	LR	NN
catch22	1.6	1.9	2.5	3.0	2.0	1.0
tsfresh	1.2	2.3	2.5	2.0	1.0	3.0
TSFuse	1.0	2.9	2.1	2.0	1.0	3.0

Table 8 Results for the feature-based methods: catch22, tsfresh, and TSFuse. For tsfresh and TSFuse, we evaluate the minimal setting (min) and the default setting (def). The table shows the results obtained with XGBoost for datasets 1–9 and with linear regression for dataset 10

(a) AUC for datasets 1–9 and R^2 for dataset 10 (mean $\pm$ std). For each baseline, a * indicates that the difference with TSFuse_{def} is significant ($p < 0.05$).

Dataset	catch22	tsfresh _{min}	tsfresh _{def}	TSFuse _{min}	TSFuse _{def}
1	0.998 $\pm$ 0.001	0.991 $\pm$ 0.003*	0.997 $\pm$ 0.001	0.998 $\pm$ 0.001	0.998 $\pm$ 0.002
2	0.997 $\pm$ 0.002	0.992 $\pm$ 0.007	0.997 $\pm$ 0.003	0.994 $\pm$ 0.004	0.997 $\pm$ 0.003
3	0.832 $\pm$ 0.012*	0.724 $\pm$ 0.009*	0.916 $\pm$ 0.007	0.760 $\pm$ 0.016	0.878 $\pm$ 0.020
4	0.988 $\pm$ 0.004	0.797 $\pm$ 0.013*	0.988 $\pm$ 0.006	0.975 $\pm$ 0.006	0.988 $\pm$ 0.004
5	0.972 $\pm$ 0.005*	0.962 $\pm$ 0.003*	0.977 $\pm$ 0.004*	0.989 $\pm$ 0.002	0.993 $\pm$ 0.002
6	0.575 $\pm$ 0.028	0.439 $\pm$ 0.040	0.548 $\pm$ 0.118	0.416 $\pm$ 0.112	0.605 $\pm$ 0.046
7	0.690 $\pm$ 0.019*	0.864 $\pm$ 0.068*	0.897 $\pm$ 0.074	0.956 $\pm$ 0.045	0.952 $\pm$ 0.049
8a	0.936 $\pm$ 0.010*	0.957 $\pm$ 0.009*	0.981 $\pm$ 0.006*	0.966 $\pm$ 0.007	0.990 $\pm$ 0.005
8b	0.892 $\pm$ 0.005*	0.867 $\pm$ 0.010*	0.932 $\pm$ 0.006*	0.890 $\pm$ 0.020	0.968 $\pm$ 0.007
8c	0.881 $\pm$ 0.008	0.814 $\pm$ 0.017*	0.888 $\pm$ 0.020*	0.865 $\pm$ 0.024	0.954 $\pm$ 0.016
9a	0.500 $\pm$ 0.000*	0.993 $\pm$ 0.008	0.979 $\pm$ 0.021	0.995 $\pm$ 0.005	0.998 $\pm$ 0.002
9b	0.500 $\pm$ 0.000*	0.659 $\pm$ 0.017*	0.842 $\pm$ 0.062	0.819 $\pm$ 0.009	0.853 $\pm$ 0.008
9c	0.500 $\pm$ 0.000*	0.750 $\pm$ 0.011*	0.831 $\pm$ 0.005*	0.858 $\pm$ 0.012	0.858 $\pm$ 0.009
10	−14.2 $\pm$ 19.7	0.740 $\pm$ 0.200*	0.669 $\pm$ 0.417	0.802 $\pm$ 0.149	−0.5 $\pm$ 2.275

(b) Run time (in seconds). The construction time is the time needed to compute and select the features for the training data. The training time is the time needed to train the XGBoost or linear regression model, which includes hyperparameter tuning. The test time is the time needed for computing the features and predicting the labels for the test data.

Dataset	Phase	catch22	tsfresh _{min}	tsfresh _{def}	TSFuse _{min}	TSFuse _{def}
1	construct	7.7	8.1	671.1	1223.1	2601.8
	train	604.8	589.2	33135.2	6238.9	29352.9
	test	7.7	7.5	620.6	16.8	210.5
2	construct	6.4	7.1	972.0	278.6	1762.9
	train	97.5	62.0	4869.3	359.3	7242.1
	test	6.4	5.6	934.2	2.8	48.9
3	construct	8.9	17.3	482.7	5096.5	4921.0
	train	1304.6	918.5	12027.2	76064.0	116320.9
	test	9.1	13.8	404.0	39.7	244.6
4	construct	2.6	3.7	272.8	315.9	563.7
	train	112.1	169.7	25880.8	3298.5	7400.5
	test	2.6	2.9	257.8	11.5	69.7
5	construct	10.3	6.6	664.6	765.6	506.9
	train	397.4	279.0	15790.5	7067.5	4818.7
	test	10.3	5.0	645.2	14.3	126.3
6	construct	2.5	2.2	935.8	82.6	275.4
	train	12.2	22.8	450.9	78.4	50.3
	test	2.5	2.0	932.2	0.7	8.0
7	construct	0.7	0.7	30.3	31.8	127.4
	train	10.3	4.6	215.0	87.3	435.9
	test	0.7	0.5	29.1	3.5	12.4

Table 8 (continued)

(b) Run time (in seconds). The construction time is the time needed to compute and select the features for the training data. The training time is the time needed to train the XGBoost or linear regression model, which includes hyperparameter tuning. The test time is the time needed for computing the features and predicting the labels for the test data.

Dataset	Phase	catch22	tsfresh _{min}	tsfresh _{def}	TSFuse _{min}	TSFuse _{def}
8a	construct	12.4	26.0	2326.4	1990.4	1139.7
	train	634.3	311.1	17971.3	11608.9	15519.7
	test	12.5	24.6	2315.5	21.4	128.8
8b	construct	12.5	11.3	995.6	905.4	1213.4
	train	676.2	555.1	16816.3	9837.2	16017.0
	test	12.5	10.7	981.2	31.7	122.4
8c	construct	12.8	17.9	908.4	806.9	3174.2
	train	703.7	468.5	19411.0	5609.0	59046.4
	test	12.8	17.3	895.9	19.2	116.7
9a	construct	170.3	632.8	91742.4	1429.3	1372.4
	train	94.1	759.1	211115.5	7209.5	4818.9
	test	170.3	630.2	91674.9	99.1	681.0
9b	construct	170.0	435.5	61547.1	294.7	963.2
	train	27.9	40.5	4606.1	3519.8	9310.3
	test	170.0	434.7	61489.6	101.0	892.2
9c	construct	170.6	367.1	61472.9	708.5	4829.7
	train	119.8	2363.0	324076.2	11758.5	86957.5
	test	170.6	365.8	61386.1	109.1	938.6
10	construct	21.8	15.7	1095.0	57.2	506.6
	train	5.4	4.2	151.1	12.0	33.7
	test	21.8	15.5	1093.9	21.2	37.1

able to find a feature representation that is at least as good as existing feature construction methods. TSFuse results in a significantly more accurate model in 8 datasets compared to catch22, and in 4 datasets compared to tsfresh.

Since the series-to-attribute transformers of TSFuse are less extensive than the feature transformations implemented in tsfresh, our system only outperforms tsfresh when fusion is important, i.e., when the series-to-series or attribute-to-attribute transformers result in a better feature representation compared to univariate features. Based on the comparison of the different types of fusion in Table 6, there are two datasets where fusion decreased the predictive performance: handwriting character classification (dataset 3) and stock price forecasting (dataset 10). These are the only two datasets where tsfresh outperforms TSFuse with the default setting. For articulatory word recognition (dataset 1) and cricket hand signals (dataset 2), Table 6 shows that fusion has (almost) no effect. Hence, our system is not able to obtain a substantially better performance on these datasets. For all other datasets, series and/or attribute fusion improved the performance of TSFuse compared to the variant without fusion. For these datasets, our system outperforms tsfresh in 9 out of 10 cases.

5.4.2 Computational cost

Table 8b shows the computational cost of each feature-based method in terms of the feature construction time, model training time, and test time, where the test time consists of the time required for computing the features for the test data and applying the model.

In terms of feature construction time, TSFuse is slower than catch22 on all datasets and slower than tsfresh in 7 out of 14 datasets when using the default setting. For TSFuse, the construction time heavily depends on the number of time series generated in step 1 and the number of interacting attribute pairs found in step 3, and therefore our system is not consistently faster or slower than tsfresh. Similarly, the training time depends on the number of constructed features for TSFuse and the number of selected features for tsfresh. Therefore, we cannot say that one system is slower or faster than the other in general.

At prediction time, TSFuse can efficiently compute the constructed features. Compared to tsfresh, the test time of our system is on average 29.4 times faster. The computation graph representation of TSFuse is helpful here, as it enables reusing intermediate operations and therefore reduces the need for recomputing the same time series and/or attributes multiple times. The catch22 baseline, which sacrifices predictive performance for computational efficiency, is at least an order of magnitude faster than both tsfresh and TSFuse with the default setting.

5.5 Comparison to dynamic time warping and deep learning (Q3)

Next, we compare TSFuse (using XGBoost and hyperparameter tuning) to approaches that operate directly on the raw time series data: 1NN using DTW as well as ResNet and TWIESN, which are two deep learning baselines commonly applied to multivariate time series data.

Table 9a shows the predictive performance (AUC or R^2) for all the approaches. 1NN-DTW is uniformly worse than the other approaches on these datasets. There are two main reasons why this baseline performs worse than all other methods. First, 1NN-DTW is the only method in this comparison that is not able to select a subset of the given time series. All series, including those that are not relevant to the given predictive task, are included in the DTW similarity, and therefore these series just add noise. Second, as shown in Section 5.3, time series fusion is important for most datasets, but 1NN-DTW is limited to the original time series only.

ResNet is a strong baseline for multivariate time series classification benchmark problems (Ruiz et al., 2020) and achieves the best predictive performance on 9 out of 14 datasets. The TWIESN architecture is less accurate than ResNet. TSFuse outperforms this baseline on 10 out of 13 classification datasets. Since TWIESN is not suitable for regression tasks, no results are given for stock price forecasting (dataset 10). The deep learning baselines are able to apply multiple operations to the original time series and hence construct features. However, it is hard to interpret these features and examine how they select and fuse the given time series.

Table 9b shows the run time performances of all approaches at both train and test time. 1NN-DTW has no training costs. Among the other approaches, TWIESN has the lowest training cost but produces less accurate models. The training time difference between ResNet and TSFuse is dataset dependent. The training time of the ResNet models mostly depends on the length of the series and the number of dimensions. For TSFuse with XGBoost, the training time heavily depends on the number of constructed features, which

Table 9 Comparison of TSFuse with the 1NN-DTW baseline and the deep learning baselines (ResNet and TWIESN). Note that dataset 10 has no results for TWIESN, as this method is not suitable for regression

(a) AUC for datasets 1–9 and R^2 for dataset 10 (mean $\pm$ std). For each baseline, a * indicates that the difference with TSFuse is significant ($p < 0.05$).

Dataset	1NN-DTW	ResNet	TWIESN	TSFuse
1	0.981 $\pm$ 0.007	1.000 $\pm$ 0.001	0.999 $\pm$ 0.001	0.998 $\pm$ 0.002
2	0.960 $\pm$ 0.006*	1.000 $\pm$ 0.000	1.000 $\pm$ 0.000	0.997 $\pm$ 0.003
3	0.568 $\pm$ 0.007*	0.989 $\pm$ 0.003*	0.766 $\pm$ 0.011*	0.878 $\pm$ 0.020
4	0.938 $\pm$ 0.014*	0.993 $\pm$ 0.003	0.973 $\pm$ 0.010	0.988 $\pm$ 0.004
5	0.882 $\pm$ 0.002*	0.996 $\pm$ 0.002	0.984 $\pm$ 0.004*	0.993 $\pm$ 0.002
6	0.500 $\pm$ 0.000*	0.517 $\pm$ 0.067	0.436 $\pm$ 0.096*	0.605 $\pm$ 0.046
7	0.886 $\pm$ 0.074	0.995 $\pm$ 0.009	0.812 $\pm$ 0.125	0.952 $\pm$ 0.049
8a	0.599 $\pm$ 0.009*	0.961 $\pm$ 0.009	0.939 $\pm$ 0.008*	0.990 $\pm$ 0.005
8b	0.529 $\pm$ 0.010*	0.789 $\pm$ 0.018*	0.673 $\pm$ 0.004*	0.968 $\pm$ 0.007
8c	0.502 $\pm$ 0.014*	0.690 $\pm$ 0.042*	0.385 $\pm$ 0.020*	0.954 $\pm$ 0.016
9a	0.500 $\pm$ 0.000*	1.000 $\pm$ 0.000	1.000 $\pm$ 0.000	0.998 $\pm$ 0.002
9b	0.500 $\pm$ 0.000*	0.891 $\pm$ 0.045	0.718 $\pm$ 0.043*	0.853 $\pm$ 0.008
9c	0.500 $\pm$ 0.000*	0.872 $\pm$ 0.060	0.739 $\pm$ 0.011*	0.858 $\pm$ 0.009
10	−1.4 $\pm$ 1.837	−1.9 $\pm$ 6.069		−0.5 $\pm$ 2.275

(b) Run time (in seconds). The training time is the time needed for constructing the features (only for TSFuse), and for training the model, including hyperparameter tuning. The test time is the time needed for computing the features (only for TSFuse) and predicting the labels for the test data.

Dataset	Phase	1NN-DTW	ResNet	TWIESN	TSFuse
1	train		8963.5	169.8	31954.7
	test	347.2	2.2	3.2	210.5
2	train		13421.8	139.3	9005.1
	test	25.3	2.4	4.1	48.9
3	train		15594.2	271.7	121241.9
	test	429.4	3.3	11.2	244.6
4	train		9151.5	93.0	7964.2
	test	35.2	1.6	4.4	69.7
5	train		1614.9	48.9	5325.6
	test	204.1	0.6	0.7	126.3
6	train		14516.8	178.0	325.7
	test	0.3	2.5	5.6	8.0
7	train		523.3	99.2	563.4
	test	1.4	0.9	0.2	12.4
8a	train		36189.5	288.9	16659.4
	test	80.0	5.8	17.6	128.8
8b	train		24879.5	200.4	17230.4
	test	75.0	3.8	16.5	122.4
8c	train		34477.1	342.1	62220.7
	test	73.8	6.0	22.3	116.7
9a	train		47498.5	291.2	6191.3
	test	83965.5	9.9	8.2	681.0

Table 9 (continued)

(b) Run time (in seconds). The training time is the time needed for constructing the features (only for TSFuse), and for training the model, including hyperparameter tuning. The test time is the time needed for computing the features (only for TSFuse) and predicting the labels for the test data.

Dataset	Phase	1NN-DTW	ResNet	TWIESN	TSFuse
9b	train		34204.1	254.0	10273.5
	test	85335.2	5.8	21.7	892.2
9c	train		45123.9	325.9	91787.2
	test	80833.5	7.6	32.9	938.6
10	train		1924.7		540.3
	test	45.6	0.5		37.1

is unrelated to the time series length and the number of given time series, since TSFuse selects a subset of the given time series and constructs new time series. At prediction time, the deep learning approaches are much faster. 1NN-DTW and TSFuse have a similar run time performance at test time on most datasets except on the indoor exercise monitoring data (datasets 9a, 9b and 9c) where the 1NN-DTW baseline is much slower. These datasets have both a large number of dimensions (75) and windows (1791) which makes computing the DTW score time-consuming compared to the other datasets.

5.6 Parameters of TSFuse (Q4)

Next, we explore the effect on both predictive performance and run time of the three important (classes of) parameters in TSFuse:

1. *Maximum computation graph depth (D)* Larger values of D mean that more complicated features can be constructed but that comes at the cost of an increased run time.
2. *Maximum number of permutations of (n_s, n_a)* Similar to the computation graph depth, more permutations of time series (n_s) and attributes (n_a) result in a larger number of features but also increase the run time.
3. *Threshold parameters (c, q, h)* The impact on performance that varying the values of the maximum correlation coefficient c , the percentage of coefficients q , and the attribute interaction threshold h will have is unclear as there is an interplay between these parameters and the characteristics of the underlying time series themselves.

For each parameter, we evaluate multiple different values while using the default values (listed in Table 5) for all other parameters. As this is a computationally expensive experiment, it focuses on two datasets: atrial fibrillation detection (dataset 6) and machine condition monitoring (dataset 7). For each configuration, we train an XGBoost model with hyperparameter tuning and measure AUC and the sum of the time needed for constructing the features and training the XGBoost models (including hyperparameter tuning).

Maximum computation graph depth (D) Figure 8 shows the predictive performance (left) and computational cost (right) when varying the computation graph depth $D \in \{1, 2, 3, 4\}$. At $D = 1$, TSFuse can only add series-to-attribute transformers to the graph. Series-to-series or attribute-to-attribute transformers can only be added when $D \geq 2$. Using both types of transformers is only possible when $D \geq 3$. In the experiments for Q1–Q3, we set

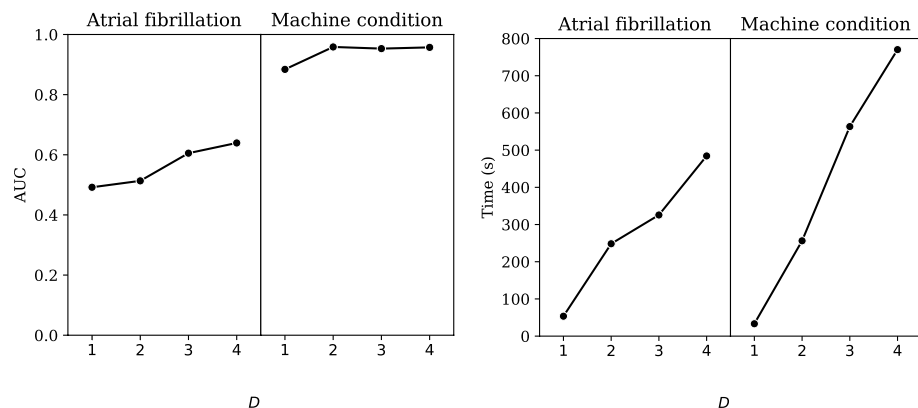


Fig. 8 Predictive performance (AUC) and run time for maximum computation graph depths $D \in \{1, 2, 3, 4\}$. The run time is the time needed for constructing the features and training the XGBoost models (including hyperparameter tuning)

$D = 3$ by default, as the difference in predictive performance with larger depths is usually small. However, a small improvement is observed on the atrial fibrillation dataset. The computational cost heavily depends on the computation graph depth as larger depths lead to more possible nodes to consider. Still, its run time scales linearly as the maximum number of series and attribute permutations parameters limit the maximum number of nodes added at each depth.

Maximum number of permutations of series (n_s) and attributes (n_a) These parameters affect the number of nodes added to the computation graph at a given depth. Figure 9 shows how the performance is affected when we vary $n_s \in \{1, 10, 100\}$ and $n_a \in \{100, 1000, 10000\}$. On these datasets, these parameters have little, if any, effect on the AUC. Increasing the value of n_s leads to slight increases in run time. The value of n_a has a larger impact on the run time, particularly when going from $n_a = 1000$ to $n_a = 10000$.

Threshold parameters (c , q , h) Figure 10 shows the results for three different values for each threshold parameter: the maximum correlation coefficient c (used in step 1), the percentage of coefficients q (used in step 2), and the attribute interaction threshold h (used in step 3). When c is small, many series are considered to be redundant which adversely impacts accuracy by removing series that may be useful. It is harder to gain insights into the effect of q and h as the results depend heavily on the problem. Using a small percentage of coefficients and a small interaction threshold results in better predictive performance for the atrial fibrillation dataset but all parameter settings give similar results for the machine monitoring dataset. While higher values of all three parameters tend to increase run time, they have a much smaller impact on the computational cost than the depth of the computation graph and the number of considered permutations.

5.7 Interpretability of the constructed features (Q5)

One of the advantages of feature construction methods over deep learning is that features are more interpretable than neural networks. One way to analyze the constructed feature set is to let a domain expert analyze the features with the highest feature importance scores.

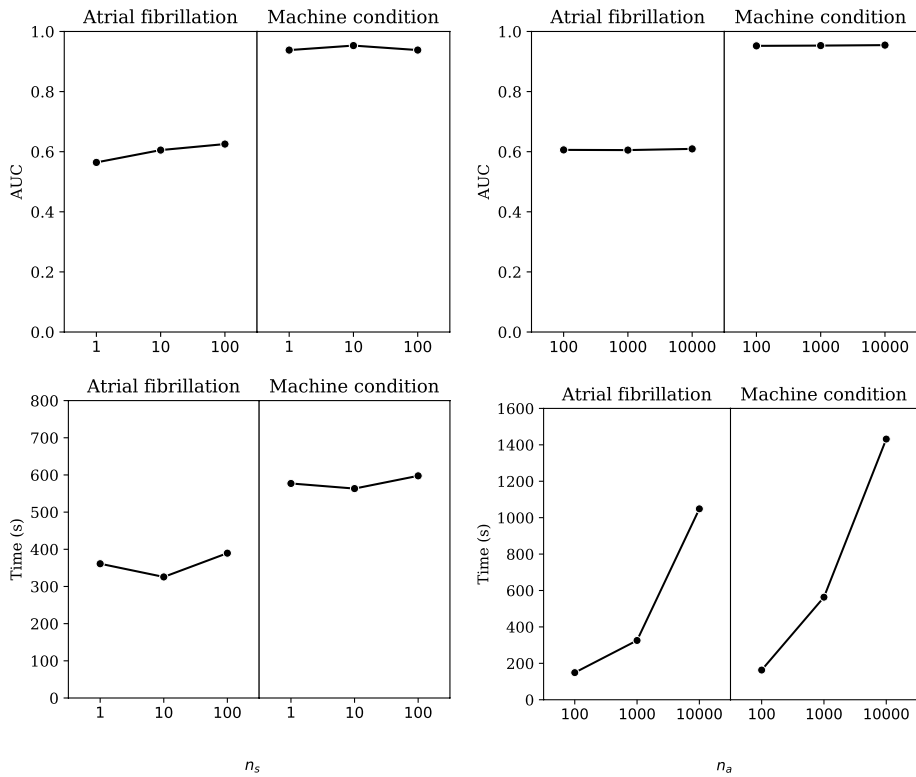


Fig. 9 Predictive performance (AUC) and run time for maximum series permutations $n_s \in \{1, 10, 100\}$ and maximum attribute permutations $n_a \in \{100, 1000, 10000\}$. The run time is the time needed for constructing the features and training the XGBoost models (including hyperparameter tuning)

This can help experts by elucidating new features that may be of importance or simply let the expert check if the constructed features conform to domain knowledge. Here, we analyze the features found in one fold of the machine condition monitoring dataset and show how the computation graph can help to interpret these features.

Feature importance For all constructed features, we rank the features according to their total gain over all trees in the XGBoost model. Figure 11 shows the ten most important features. The most important feature is the median of the ratio of the v variable and the w variable. This can be better understood by inspecting the part of the computation graph responsible for computing this feature, which is shown in Figure 12. First, the variables v and w are combined by the Ratio transformer, which computes the ratio between the time series collections that belong to these variables. Second, the Median transformer extracts the median value of this newly constructed time series. The plots to the right of the computation graph show the u , v , w signals and the ratio between v and w for two runs: one without a failure (left) and one with a failure (right). The interval where the machine fault takes place is highlighted in yellow. In normal conditions, the three signals (i.e., u , v , w) have the same amplitude. In the highlighted interval, we introduced a voltage unbalance. This type of machine fault disturbs the balance of the amplitudes of the three signals. Whereas the ratio between each pair of signals is normally close to one, the voltage

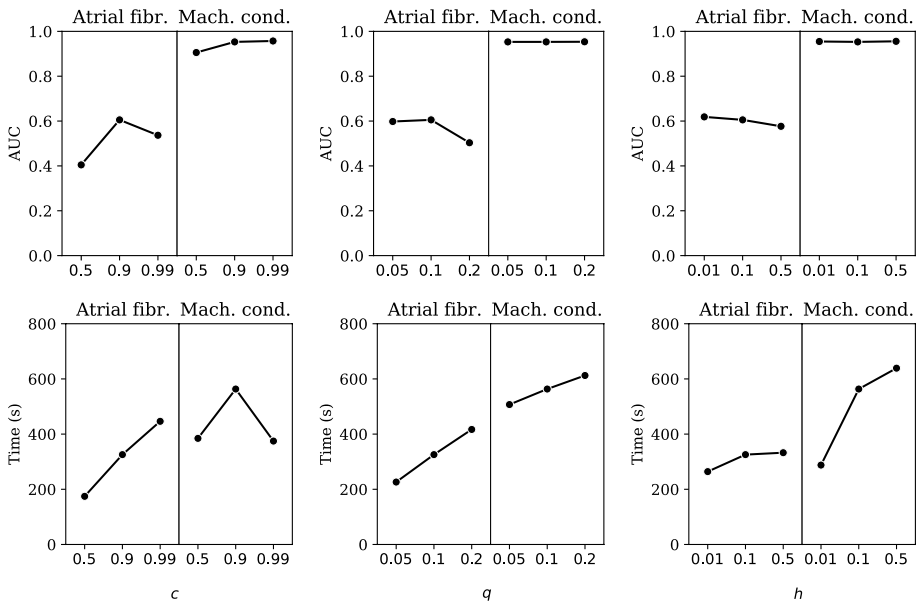


Fig. 10 Predictive performance (AUC) and run time for maximum correlation coefficient $c \in \{0.5, 0.9, 0.99\}$, percentage of coefficients $q \in \{0.05, 0.1, 0.2\}$ and attribute interaction threshold $h \in \{0.01, 0.1, 0.5\}$. The run time is the time needed for constructing the features and training the XGBoost models (including hyperparameter tuning)

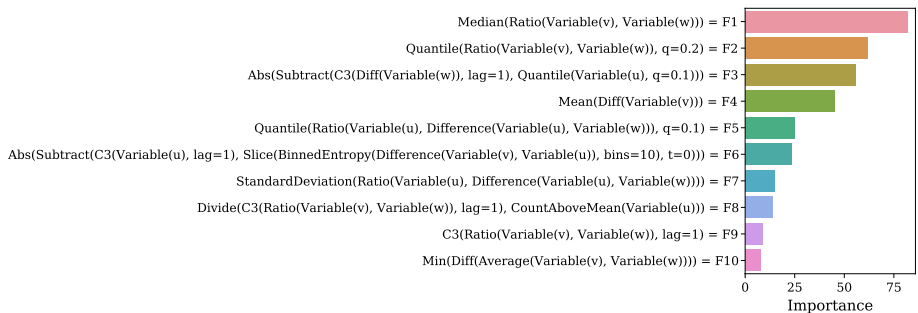


Fig. 11 Most important features for the machine condition monitoring dataset. The feature importance is the total gain over all trees in the XGBoost model

unbalance changes these ratios. The plot shows $\text{Ratio}(v, w)$, i.e., the ratio between v and w , which clearly increased during the fault. For a given window, any tendency measure computed from $\text{Ratio}(v, w)$ can capture this increase. In this case, the median was selected as a relevant feature.

Correlation Even though TSFuse attempts to eliminate redundant features, some features are still correlated to each other. Figure 13 visualizes these correlations by plotting the absolute value of the Pearson correlation between each pair of features. The figure shows that two groups of highly correlated features exist. The first group consists of features F1, F2 and F9:

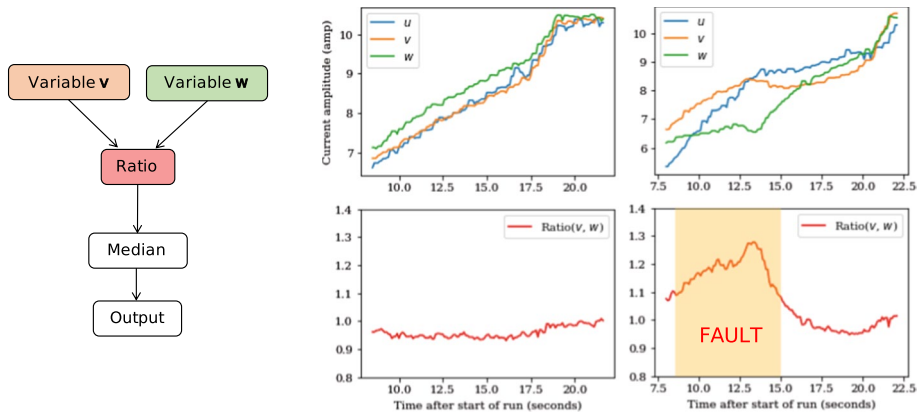


Fig. 12 Subgraph of the computation graph that extracts the $\text{Median}(\text{Ratio}(\text{Variable}(v), \text{Variable}(w)))$ feature. The time series returned by the Variable nodes and the Ratio transformer are shown on the right, for two runs with the same motor

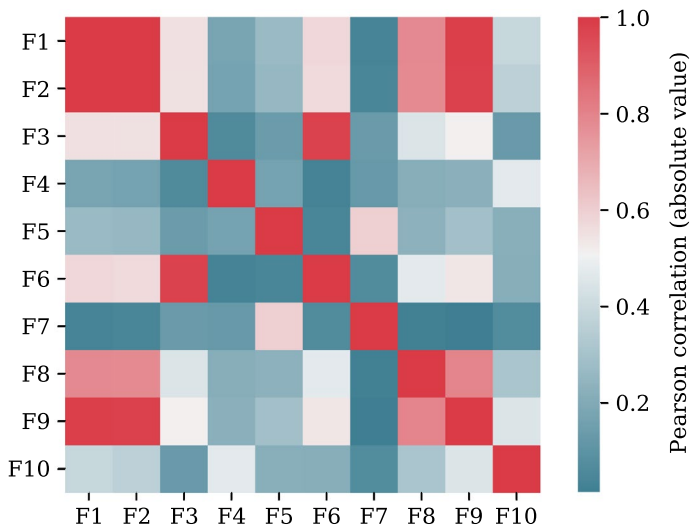


Fig. 13 Correlation matrix for the most important features. For each pair of features, the figure shows the absolute value of the Pearson correlation. See Fig. 11 for the full names of features F1, ..., F10

- $\text{Median}(\text{Ratio}(\text{Variable}(v), \text{Variable}(w)))$
- $\text{Quantile}(\text{Ratio}(\text{Variable}(v), \text{Variable}(w)), q=0.2)$
- $\text{C3}(\text{Ratio}(\text{Variable}(v), \text{Variable}(w)), \text{lag}=1)$

The second group consists of features F3 and F6:

- $\text{Abs}(\text{Subtract}(\text{C3}(\text{Diff}(\text{Variable}(w)), \text{lag}=1), \text{Quantile}(\text{Variable}(u), q=0.1)))$
- $\text{Abs}(\text{Subtract}(\text{C3}(\text{Variable}(u), \text{lag}=1), \text{Slice}(\text{BinnedEntropy}(\text{Difference}(\text{Variable}(v), \text{Variable}(u)), \text{bins}=10), t=0)))$

The correlation between the features within each group is 0.9 or larger. Within these groups, the features are constructed from the same signals using similar transformers, which indicates that these features describe collinear characteristics. It is important to be cognizant of these correlations when interpreting the meaning of a feature.

6 Conclusion

This paper presented TSFuse, a system that automatically constructs features from multiple time series data. Our system goes beyond existing automated feature construction systems in two important ways. First, it provides functionality to derive new series by fusing together signals from multiple series. For example, this would enable computing the ratio between two signals or deriving the resultant of a tri-axial accelerometer. Second, it supports constructing new attributes by comparing the values of already computed attributes. While this functionality is important, it makes the feature construction problem substantially more complex as it entails considering a much bigger set of possible operations. Hence, an exhaustive search is no longer feasible. To address this challenge, we designed a computationally efficient search method that reduces the search space by exploiting constraints based on contextual information and by removing redundant operations. To evaluate our system, we compared the predictive performance and computational cost on ten real-world time series datasets to univariate feature construction systems. In the experimental evaluation, we showed that our system is able to find a better feature representation than existing feature construction systems when fusion operations are important.

Appendix A

Transformers

In this section, we define the output and preconditions of the transformers implemented in our system. Each transformer takes one or more time series or attribute collections as input and returns one new time series or attribute collection. We represent each collection x as a two-dimensional matrix of shape (t, d) where t is the number of time steps ($t = 1$ for attribute collections) and d is the number of series or attributes. We use the following notation for selecting a specific part of a collection x :

- x_i selects the i^{th} time series or attribute;
- $x(j)$ selects the j^{th} time step.

Series-to-series (and attribute-to-attribute) transformers

Since the attribute-to-attribute transformers (Average, Difference, Ratio) are a subset of the series-to-series transformers, we only list the series-to-series transformers. For each transformer, we give a short description of what the transformer computes.

- Derivative(x)

Output: first-order derivative of each time series of collection x

$$\text{Derivative}(x) = [x(i+1) - x(i) \mid i \in \{1, \dots, t-1\}]$$

- Resultant(x)

Output: resultant of all time series of collection x

Precondition: x consists of $d > 1$ time series

$$\text{Resultant}(x) = \sqrt{x_1^2 + \dots + x_d^2}$$

- Average(x, y)

Output: average of all values in x and y

Precondition: x and y have the same number of series d and time steps t

$$\text{Average}(x, y) = \frac{x + y}{2}$$

- Difference(x, y)

Output: absolute difference of all values in x and y

Precondition: x and y have the same number of series d and time steps t

$$\text{Difference}(x, y) = |x - y|$$

- Ratio(x, y)

Output: ratio between all values in x and y

Precondition: x and y have the same number of series d and time steps t , and y is non-zero

$$\text{Ratio}(x, y) = \frac{x}{y}$$

- Angle(x, y, z)

Output: angle defined by three collections (each collection represents 2D or 3D positions)

Precondition: x, y and z have the same number of series d and time steps t , and each collection has $d > 1$ time series

$$\text{Angle}(x, y, z) = \arccos\left(\frac{\overline{xy} \cdot \overline{yz}}{||\overline{xy}|| \ ||\overline{yz}||}\right)$$

Series-to-attribute transformers

We list the series-to-attribute transformers of the two settings: *minimal* and *default*.

Minimal

- Mean(x)

Output: Arithmetic mean of each series x_i

$$\text{Mean}(x) = \left[\frac{x_i(1) + \dots + x_i(t)}{t} \right]_{i \in \{1, \dots, d\}}$$

– Median(x)

Output: Middle value of the sorted values s_i of each series x_i (if t is odd) or the average of the two middle values (if t is even)

$$\text{Median}(x) = \left[\begin{cases} s_i((t+1)/2) & \text{if } t \text{ is odd} \\ (s_i(t/2) + s_i(t/2 + 1))/2 & \text{if } t \text{ is even} \end{cases} \right]_{i \in \{1, \dots, d\}}$$

– Variance(x)

Output: Variance of each series x_i

$$\text{Variance}(x) = \left[\frac{(x_i(1) - \bar{x}_i)^2 + \dots + (x_i(t) - \bar{x}_i)^2}{t} \right]_{i \in \{1, \dots, d\}}$$

– StandardDeviation(x)

Output: Square root of the variance of x

$$\text{StandardDeviation}(x) = \sqrt{\text{Variance}(x)}$$

– Skewness(x)

Output: Sample skewness of each series x_i

– Kurtosis(x)

Output: Kurtosis of each series x_i (using Fisher's definition)

– Min(x)

Output: Smallest value of each series x_i

– Max(x)

Output: Largest value of each series x_i

– Sum(x)

Output: Sum of each series x_i

– Length(x)

Output: Number of time steps t

Default

All minimal transformers, extended with the following transformers:

- Energy(x)

Output: Sum of the squared values of each series x_i

$$\text{Energy}(x) = \left[(x_i(1))^2 + \dots + (x_i(t))^2 \right]_{i \in \{1, \dots, d\}}$$

- C3(x)

Output: Non-linearity of each series x_i with lag $l=1$ (Schreiber and Schmitz, 1997)

$$C3(x) = \left[\sum_{j=1}^{t-2l} x_i(j+2l) \cdot x_i(j+l) \cdot x_i(j) / (t-2l) \right]_{i \in \{1, \dots, d\}}$$

- CID(x)

Output: Time series complexity estimate of each series x_i (Batista et al. 2014)

$$CID(x) = \left[\sum_{j=1}^{t-1} (x_i(j+1) - x_i(j))^2 \right]_{i \in \{1, \dots, d\}}$$

- CountAboveMean(x)

Output: Number of values greater than the mean of x_i

$$\text{CountAboveMean}(x) = \left[\sum_{j=1}^t (x_i(j) > \bar{x}_i) \right]_{i \in \{1, \dots, d\}}$$

- CountBelowMean(x)

Output: Number of values smaller than the mean of x_i

$$\text{CountAboveMean}(x) = \left[\sum_{j=1}^t (x_i(j) < \bar{x}_i) \right]_{i \in \{1, \dots, d\}}$$

- Quantile(x, q)

Output: q -quantile of each series x_i

- EnergyRatio(x)

Output: For each series x_i , this transformer divides the values into 10 bins and computes the sum of squares of each bin divided by the total sum of squares of all bins.

- BinnedDistribution(x)

Output: For each series x_i , this transformer divides the values into 10 bins and computes the fraction of values in each bin.

- BinnedEntropy(x)

Output: For each series x_i , this transformer divides the values into 10 bins and computes the entropy of the values in each bin.

- AggregatedLinearTrend(x, size)

Output: For each series x_i , this transformer aggregates the series by splitting the series in windows of the given size and computing the mean of each window. It then calculates a linear least-squares regression line for x = the window identifiers (i.e., 0, 1, 2, ...) and y = the average of each window. The transformer then returns six statistics of the fitted regression line: the slope, the intercept, the correlation coefficient, the p-value of a two-sided hypothesis test where the null hypothesis is that the slope is zero, the standard error the slope, and the standard error of the intercept.

- FFT(x)

Output: Absolute value of each coefficient of a fast Fourier transform of each series x_i

- CWT(x)

Output: Coefficients of a continuous Ricker wavelet transform of each series x_i

- AutoRegressiveCoefficients(x)

Output: For each series x_i , this transformer fits an autoregressive (AR) model using `statsmodels.tsa.ar_model.AR` and returns the coefficients of the model.

- `FriedrichCoefficients(x)`

Output: For each series x_i , this transformer fits a polynomial to the deterministic dynamics of a Langevin model (Friedrich et al., 2000) and returns the coefficients of the polynomial.

- `AutoCorrelation(x)`

Output: For each series x_i , this transformer computes the auto-correlation coefficient for lags $\{1, \dots, t\}$ and returns the coefficient for each lag.

- `CrossCorrelation(x, y)`

Output: For each dimension $i \in \{1, \dots, d\}$, this transformer computes the cross-correlation coefficient between x_i and y_i for lags $\{1, \dots, t\}$ and returns the coefficient for each lag. *Precondition:* x and y have the same number of series d and time steps t

Appendix B

Datasets

Articulatory word recognition

By converting articulatory data into text, silent speech interfaces can facilitate communication in persons with voice and speech problems. In this dataset, the goal is to recognize the word pronounced by a person based on the movement of the person's tongue and lips (Wang et al., 2013). We use the data provided by Shokoohi-Yekta et al. (2015) which consists of a selection of three sensor locations: the lower lip (LL), the upper lip (UL), and one of the four sensors on the tongue (T1). Each sensor records the position as (x, y, z) coordinates, resulting in a total of nine dimensions. To collect the data, multiple native English speakers were instructed to pronounce sequences of 25 words. Given the LL, UL, and T1 coordinates collected during a window of 144 samples, the task is to predict the word that was pronounced.

Representation in TSFuse We group each sensor location as a three-dimensional time series collection. Since all time series are collected from the same types of sensors, we do not specify tags. The compatibility relation allows any combination of signals.

Cricket hand signals

In cricket, players can signal events using hand motions. This dataset consists of the movements of 12 hand signals recorded using two accelerometers attached to the wrists of a player: one on the left and one on the right (Ko et al., 2005). Similar to the articulatory word recognition dataset, we use the data provided by Shokoohi-Yekta et al. (2015). There are six dimensions: one per axis (x, y, z) per accelerometer. Each window consists of 1197 samples and corresponds to a single hand signal. Given the accelerometer data, the aim is to predict which hand signal was performed.

Representation in TSFuse We group the data per accelerometer. Thus, we have two time series collections, each with three dimensions. Since there are only two sensors, the compatibility relation allows any combination.

Handwriting character classification

Similar to the cricket hand signals, this dataset was collected from sensors positioned on a person's wrist (Shokoohi-Yekta et al., 2015). The sensors consist of a tri-axial accelerometer and a tri-axial gyroscope. Hence, there are six dimensions in this dataset. Given a window of 152 samples, the task is to detect the letter written by the person. There are 26 classes: one for each character of the alphabet.

Representation in TSFuse Each window consists of two time series collections x_1 and x_2 , where x_1 groups the three time series generated by the accelerometer and x_2 groups the three time series generated by the gyroscope. We assign the following tags to these collections to specify the sensor types:

$$M_{x_1} = \{\text{type} \mapsto \text{accelerometer}\} \quad M_{x_2} = \{\text{type} \mapsto \text{gyroscope}\}$$

Since it is hard to interpret features that fuse the accelerometer and gyroscope data, we define the following compatibility relation for this dataset to avoid such combinations:

$$r(x_i, x_j) = \text{true if } (\text{type of } x_i = \text{type of } x_j) \text{ else false}$$

Accelerometer-based gesture recognition

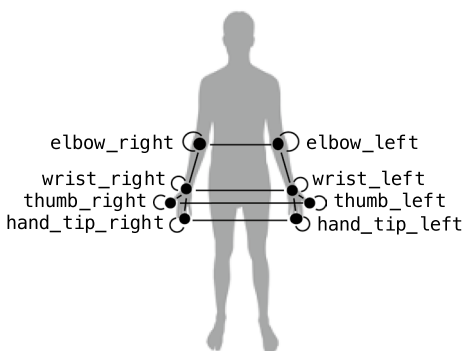
In this dataset, a single tri-axial accelerometer was used to record the hand movements of eight gestures (Liu et al., 2009). We again used the data provided by (Shokoohi-Yekta et al. 2015). Given the accelerometer data of one window of 315 samples, the goal is to predict the gesture.

Representation in TSFuse Since only one sensor was used, we represent each axis as a one-dimensional time series collection. The compatibility relation allows any combination of signals.

Camera-based gesture recognition Instead of accelerometers, gesture recognition can also be based on a camera tracking system that records the positions of a person's body parts. This dataset consists of six gestures performed by an aircraft marshaller (Ghouaiel et al., 2017). Each trial captures the 3D position of eight body parts (left and right hand tips, thumbs, wrists, and elbows) with 51 samples collected during approximately 2.34 seconds. In total, the dataset consists of 360 trials. Each trial corresponds to one window. Given the positions collected during one window, the task is to detect the gesture performed by an aircraft marshaller.

Representation in TSFuse Each window consists of eight time series collections $x_1, \dots, x_8$ (i.e., one for each body part) and each time series collection contains 3 series (i.e., x , y , z coordinates) of length 51. We assign the following tags to the collections:

Fig. 14 Compatible pairs of locations for the camera-based gesture recognition dataset



$$M_{x_1} = \{\text{location} \mapsto \text{elbow_right}\}$$

$$M_{x_2} = \{\text{location} \mapsto \text{elbow_left}\}$$

$$M_{x_3} = \{\text{location} \mapsto \text{wrist_right}\}$$

$$M_{x_4} = \{\text{location} \mapsto \text{wrist_left}\}$$

$$M_{x_5} = \{\text{location} \mapsto \text{thumb_right}\}$$

$$M_{x_6} = \{\text{location} \mapsto \text{thumb_left}\}$$

$$M_{x_7} = \{\text{location} \mapsto \text{hand_tip_right}\}$$

$$M_{x_8} = \{\text{location} \mapsto \text{hand_tip_left}\}$$

Based on these tags, we define the following compatibility relation to specify that it only makes sense to combine the coordinates of (1) the same body part, (2) the corresponding body part at the opposite side of the body, or (3) body parts that are connected through limbs. The compatible pairs of locations are shown in Figure 14.

Atrial fibrillation detection

Atrial fibrillation (AF) is an irregular heartbeat that increases the risk of heart failure, stroke, and dementia. In this dataset, the challenge is to detect if and when an AF episode will end based on ECG measurements (Moody, 2004; Goldberger et al., 2000). Each recording consists of two-channel ECG data recorded at 128Hz during one minute. Thus, there are two dimensions and 7680 samples per recording. As opposed to the UEA benchmark version of this dataset (Bagnall et al., 2018), we do not subsample the data and create windows of one minute that each contain a complete recording. Given the data of a single window, the task is to predict the type of episode. There are three classes: (i) non-terminating, (ii) self-terminating after at least one minute after the end of the recording, and (iii) self-terminating immediately (i.e., in less than one second) after the recording.

Representation in TSFuse The two-channel ECG data is represented as two one-dimensional time series collections. The compatibility relation allows any combination.

Machine condition monitoring

Machine failures can cause high costs for companies and society. Therefore, a lot of companies aim to automatically detect these faults as early as possible by monitoring their machines using various sensors, for example, accelerometers and electrical current sensors. In this dataset, the goal is to detect an electrical failure of a three-phase motor, more specifically a voltage imbalance (Hendrickx et al., 2020). As this fault increases machine wear and temperature, early detection will extend the lifetime of the motor. The dataset consists of two motors, each performing multiple linear run-ups of 20 seconds in both healthy and faulty conditions. Every trial is represented by three time series, the peak amplitudes of each motor phase current (sampled at 3 Hz). We further divide these trails into 226 windows of one second with 50% overlap and label each window with the health condition of the machine during that second. Note that the three current signals are out-of-phase. Therefore, we used sample-and-hold to combine the phase current amplitudes. This explains the 7–8 time stamps in each window in Table 4, even though we sample the signals at 3Hz. To evaluate the generalization to unseen operational conditions, we use different speed regimes in the training and test data. Specifically, we divide the data into six speed regimes defined by the rotations per minute (RPM) of the motor: [600, 700]; [700, 800]; ...; [1100, 1200]. The cross-validation splits are made based on these groups.

Representation in TSFuse Each window is represented as a set of three time series collections u , v , w . Each collection consists of one time series, corresponding to the peak amplitude of the current in one of the three phases. The compatibility relation allows any combination of collections.

Human activity recognition Detecting a person’s daily life activities based on one or more wearable sensors is a widely studied time series classification task. In this paper, we use the USC Human Activity Database (HAD) (Zhang and Sawchuk, 2012). This dataset consists of activity data for twelve different activities. Fourteen subjects performed each of the activities five times, resulting in 840 trials in total. During every trial, a tri-axial accelerometer and tri-axial gyroscope were attached near the subject’s front right hip, generating six time series per trial. The data was sampled at 50Hz. For each trail, we selected the first 10 seconds, resulting in windows of length 500.

When deploying an activity recognition system, users can attach the sensors in different ways than intended. For example, when using a mobile phone, the orientation of the phone in the person’s pocket may vary. Since the different orientation may affect the predictive accuracy of the model, it is of practical interest to have a model that is invariant to the orientation of the sensor (Yurtman and Barshan, 2017). To evaluate TSFuse’s robustness in this regard, we rotate the sensor data by multiplying the accelerometer and gyroscope measurements with randomly generated rotation matrices. Using a perturbation parameter δ , we control the amount of rotation. In the experiments, we consider three variations of the activity recognition dataset:

- a. The original *HAD* dataset
- b. The *HAD* dataset rotated with a small perturbation ($\delta = 0.5$)
- c. The *HAD* dataset rotated with a large perturbation ($\delta = 1.0$)

In order to rotate the (x, y, z) axes of the accelerometer and gyroscope measurements, we sample a, b, c uniformly from $[0, 1]$ and define the following variables:⁴

$$\theta = a \cdot 2\pi\delta$$

$$\phi = b \cdot 2\pi$$

$$z = c \cdot 2\delta$$

where θ is the rotation about the pole (Z), ϕ is the direction of the pole deflection, and z is the magnitude of the pole deflection. For the perturbation parameter, $\delta = 0$ corresponds to no rotation and $\delta = 1$ corresponds to maximal rotation. We define the rotation matrix R as follows:

$$R = \left(\begin{bmatrix} \sin(\phi\sqrt{z}) \\ \cos(\phi\sqrt{z}) \\ \sqrt{2-2z\delta} \end{bmatrix} \otimes \begin{bmatrix} \sin(\phi\sqrt{z}) \\ \cos(\phi\sqrt{z}) \\ \sqrt{2-z} \end{bmatrix} - I_3 \right) \cdot \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where $\otimes$ is the tensor product. To rotate the (x, y, z) accelerometer and gyroscope data, we multiply R with the $[x_t, y_t, z_t]$ vector that contains the sensor measurements at each time stamp t .

Representation in TSFuse Similar to the handwriting character classification dataset, we group the data in two time series collections x_1 (accelerometer) and x_2 (gyroscope) and define the following tags and compatibility relation:

$$M_{x_1} = \{\text{type} \mapsto \text{accelerometer}\} \quad M_{x_2} = \{\text{type} \mapsto \text{gyroscope}\}$$

$$r(x_i, x_j) = \text{true if } (\text{type of } x_i = \text{type of } x_j) \text{ else false}$$

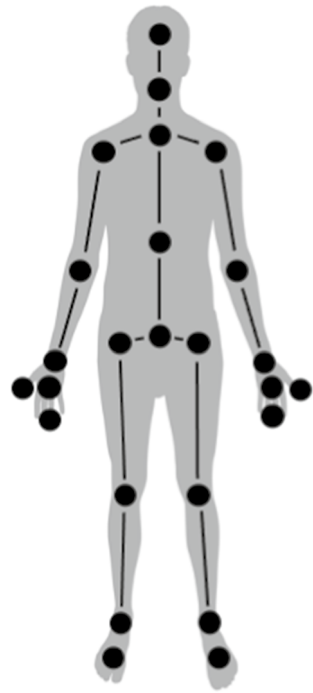
Indoor exercise monitoring Monitoring of indoor rehabilitation exercises can improve the quality of the performed exercises by giving feedback to the patient. In this dataset, a Microsoft Kinect camera was used to capture the positions of 25 joints of the patient's body during the execution of an exercise (Decroos et al., 2018). The positions are 3D coordinates that were derived automatically from the camera images. Ten subjects performed multiple repetitions of three exercises, with and without mistakes. Each repetition corresponds to a window of 31–308 samples. With a sample rate of 30Hz, these windows have a duration of 1–10s. There are three target variables:

- The type of exercise: squat, forward lunge, or sideward lunge;
- The correctness of the exercise: correct or incorrect,
- The type of mistake that was made: none, knee over toes, knock knee, or forward trunk lean.

Representation in TSFuse We represent the data as 25 time series collections, one for each joint, where each collection has three dimensions to represent the 3D coordinates. For each collection, we assign a 'location' tag that specifies the location of the joint. The compatibility relation only includes all pairs of joint locations that are connected to each

⁴ Arvo, J.: Fast random rotation matrices. In: Graphics Gems III, pp. 117–120 (1992)

Fig. 15 25 joint locations of the indoor exercise monitoring data-set. The lines connect all pairs of compatible locations



other directly via a limb. Figure 15 visually shows the location tags and the compatibility relation.

Stock price forecasting The goal of stock price forecasting is to estimate the next day's stock price based on historical data. We selected the data of the Dow Jones Industrial Average (*DJI*) from January 1, 2017 until December 31, 2019. On each day, this dataset consists of the opening price, closing price, highest price, and lowest price. We follow an approach similar to (Guo et al., 2014) which uses a sliding window approach and splits the data in windows of 30 days with a step size of one day. The task is to predict the closing price on a given day, based on the data of the past 30 days.

Representation in TSFuse Each window is represented as a set of 4 time series collections (i.e., opening price, closing price, highest price, and lowest price). Each time series collection consists of one time series with 30 time steps (i.e., the price on each day). The compatibility relation allows any combination of these time series collections.

Results for the feature-based methods

Dataset 1	Error rate	AUC
catch22 + 5NN	0.266 ± 0.037	0.949 ± 0.002
catch22 + LR	0.182 ± 0.015	0.977 ± 0.002
catch22 + XGBoost	0.073 ± 0.013	0.998 ± 0.001
tsfresh _{min} + 5NN	0.148 ± 0.006	0.982 ± 0.000
tsfresh _{min} + LR	0.116 ± 0.007	0.993 ± 0.002
tsfresh _{min} + XGBoost	0.158 ± 0.026	0.991 ± 0.003
tsfresh _{def} + 5NN	0.082 ± 0.005	0.985 ± 0.001
tsfresh _{def} + LR	0.069 ± 0.029	0.998 ± 0.002
tsfresh _{def} + XGBoost	0.087 ± 0.009	0.997 ± 0.001
TSFuse _{min} + 5NN	0.961 ± 0.013	0.505 ± 0.015
TSFuse _{min} + LR	0.960 ± 0.000	0.500 ± 0.000
TSFuse _{min} + XGBoost	0.071 ± 0.019	0.998 ± 0.001
TSFuse _{def} + 5NN	0.914 ± 0.028	0.575 ± 0.035
TSFuse _{def} + LR	0.960 ± 0.000	0.500 ± 0.000
TSFuse _{def} + XGBoost	0.067 ± 0.021	0.998 ± 0.002
Dataset 2	Error rate	AUC
catch22 + 5NN	0.200 ± 0.002	0.974 ± 0.010
catch22 + LR	0.115 ± 0.031	0.983 ± 0.012
catch22 + XGBoost	0.063 ± 0.022	0.997 ± 0.002
tsfresh _{min} + 5NN	0.077 ± 0.017	0.995 ± 0.003
tsfresh _{min} + LR	0.061 ± 0.001	0.997 ± 0.001
tsfresh _{min} + XGBoost	0.096 ± 0.052	0.992 ± 0.007
tsfresh _{def} + 5NN	0.185 ± 0.018	0.980 ± 0.003
tsfresh _{def} + LR	0.154 ± 0.039	0.976 ± 0.019
tsfresh _{def} + XGBoost	0.060 ± 0.029	0.997 ± 0.003
TSFuse _{min} + 5NN	0.835 ± 0.035	0.648 ± 0.057
TSFuse _{min} + LR	0.791 ± 0.043	0.675 ± 0.080
TSFuse _{min} + XGBoost	0.055 ± 0.022	0.994 ± 0.004
TSFuse _{def} + 5NN	0.778 ± 0.004	0.718 ± 0.022
TSFuse _{def} + LR	0.851 ± 0.066	0.643 ± 0.143
TSFuse _{def} + XGBoost	0.052 ± 0.027	0.997 ± 0.003
Dataset 3	Error rate	AUC
catch22 + 5NN	0.928 ± 0.013	0.563 ± 0.006
catch22 + LR	0.895 ± 0.017	0.655 ± 0.005
catch22 + XGBoost	0.679 ± 0.020	0.832 ± 0.012
tsfresh _{min} + 5NN	0.912 ± 0.010	0.570 ± 0.007
tsfresh _{min} + LR	0.901 ± 0.009	0.657 ± 0.008
tsfresh _{min} + XGBoost	0.824 ± 0.015	0.724 ± 0.009
tsfresh _{def} + 5NN	0.817 ± 0.013	0.673 ± 0.006
tsfresh _{def} + LR	0.601 ± 0.020	0.880 ± 0.004
tsfresh _{def} + XGBoost	0.498 ± 0.023	0.916 ± 0.007

Dataset 3	Error rate	AUC
TSFuse _{min} + 5NN	0.959 ± 0.002	0.505 ± 0.008
TSFuse _{min} + LR	0.956 ± 0.000	0.500 ± 0.000
TSFuse _{min} + XGBoost	0.796 ± 0.017	0.760 ± 0.016
TSFuse _{def} + 5NN	0.956 ± 0.008	0.505 ± 0.009
TSFuse _{def} + LR	0.956 ± 0.000	0.500 ± 0.000
TSFuse _{def} + XGBoost	0.599 ± 0.046	0.878 ± 0.020
Dataset 4	Error rate	AUC
catch22 + 5NN	0.516 ± 0.028	0.818 ± 0.015
catch22 + LR	0.443 ± 0.045	0.893 ± 0.016
catch22 + XGBoost	0.123 ± 0.019	0.988 ± 0.004
tsfresh _{min} + 5NN	0.639 ± 0.029	0.759 ± 0.016
tsfresh _{min} + LR	0.593 ± 0.021	0.823 ± 0.010
tsfresh _{min} + XGBoost	0.604 ± 0.014	0.797 ± 0.013
tsfresh _{def} + 5NN	0.190 ± 0.025	0.958 ± 0.011
tsfresh _{def} + LR	0.147 ± 0.023	0.975 ± 0.008
tsfresh _{def} + XGBoost	0.111 ± 0.028	0.988 ± 0.006
TSFuse _{min} + 5NN	0.844 ± 0.023	0.550 ± 0.023
TSFuse _{min} + LR	0.868 ± 0.022	0.514 ± 0.031
TSFuse _{min} + XGBoost	0.163 ± 0.029	0.975 ± 0.006
TSFuse _{def} + 5NN	0.805 ± 0.039	0.583 ± 0.036
TSFuse _{def} + LR	0.875 ± 0.002	0.500 ± 0.000
TSFuse _{def} + XGBoost	0.115 ± 0.030	0.988 ± 0.004
Dataset 5	Error rate	AUC
catch22 + 5NN	0.369 ± 0.008	0.908 ± 0.008
catch22 + LR	0.236 ± 0.014	0.951 ± 0.000
catch22 + XGBoost	0.181 ± 0.019	0.972 ± 0.005
tsfresh _{min} + 5NN	0.339 ± 0.000	0.926 ± 0.002
tsfresh _{min} + LR	0.232 ± 0.007	0.966 ± 0.001
tsfresh _{min} + XGBoost	0.229 ± 0.009	0.962 ± 0.003
tsfresh _{def} + 5NN	0.328 ± 0.028	0.904 ± 0.012
tsfresh _{def} + LR	0.307 ± 0.029	0.898 ± 0.007
tsfresh _{def} + XGBoost	0.138 ± 0.016	0.977 ± 0.004
TSFuse _{min} + 5NN	0.691 ± 0.057	0.686 ± 0.075
TSFuse _{min} + LR	0.681 ± 0.068	0.670 ± 0.077
TSFuse _{min} + XGBoost	0.117 ± 0.016	0.989 ± 0.002
TSFuse _{def} + 5NN	0.751 ± 0.021	0.633 ± 0.018
TSFuse _{def} + LR	0.813 ± 0.021	0.543 ± 0.043
TSFuse _{def} + XGBoost	0.087 ± 0.012	0.993 ± 0.002
Dataset 6	Error rate	AUC
catch22 + 5NN	0.600 ± 0.000	0.580 ± 0.047
catch22 + LR	0.533 ± 0.000	0.674 ± 0.028
catch22 + XGBoost	0.600 ± 0.112	0.575 ± 0.028
tsfresh _{min} + 5NN	0.767 ± 0.033	0.462 ± 0.028

Dataset 6	Error rate	AUC
tsfresh _{min} + LR	0.767 ± 0.033	0.503 ± 0.010
tsfresh _{min} + XGBoost	0.727 ± 0.036	0.439 ± 0.040
tsfresh _{def} + 5NN	0.600 ± 0.000	0.582 ± 0.012
tsfresh _{def} + LR	0.733 ± 0.052	0.365 ± 0.030
tsfresh _{def} + XGBoost	0.620 ± 0.067	0.548 ± 0.118
TSFuse _{min} + 5NN	0.713 ± 0.031	0.457 ± 0.033
TSFuse _{min} + LR	0.780 ± 0.043	0.398 ± 0.066
TSFuse _{min} + XGBoost	0.740 ± 0.096	0.416 ± 0.112
TSFuse _{def} + 5NN	0.753 ± 0.095	0.493 ± 0.109
TSFuse _{def} + LR	0.633 ± 0.095	0.546 ± 0.135
TSFuse _{def} + XGBoost	0.573 ± 0.100	0.605 ± 0.046
Dataset 7	Error rate	AUC
catch22 + 5NN	0.406 ± 0.041	0.640 ± 0.030
catch22 + LR	0.365 ± 0.031	0.668 ± 0.013
catch22 + XGBoost	0.336 ± 0.017	0.690 ± 0.019
tsfresh _{min} + 5NN	0.187 ± 0.086	0.839 ± 0.088
tsfresh _{min} + LR	0.224 ± 0.049	0.725 ± 0.033
tsfresh _{min} + XGBoost	0.188 ± 0.088	0.864 ± 0.068
tsfresh _{def} + 5NN	0.258 ± 0.016	0.806 ± 0.009
tsfresh _{def} + LR	0.464 ± 0.055	0.501 ± 0.066
tsfresh _{def} + XGBoost	0.199 ± 0.075	0.897 ± 0.074
TSFuse _{min} + 5NN	0.416 ± 0.089	0.607 ± 0.119
TSFuse _{min} + LR	0.319 ± 0.063	0.736 ± 0.083
TSFuse _{min} + XGBoost	0.058 ± 0.059	0.956 ± 0.045
TSFuse _{def} + 5NN	0.315 ± 0.054	0.800 ± 0.093
TSFuse _{def} + LR	0.365 ± 0.119	0.522 ± 0.328
TSFuse _{def} + XGBoost	0.073 ± 0.049	0.952 ± 0.049
Dataset 8a	Error rate	AUC
catch22 + 5NN	0.574 ± 0.024	0.802 ± 0.008
catch22 + LR	0.503 ± 0.049	0.888 ± 0.026
catch22 + XGBoost	0.417 ± 0.029	0.936 ± 0.010
tsfresh _{min} + 5NN	0.443 ± 0.017	0.882 ± 0.002
tsfresh _{min} + LR	0.440 ± 0.012	0.925 ± 0.020
tsfresh _{min} + XGBoost	0.294 ± 0.027	0.957 ± 0.009
tsfresh _{def} + 5NN	0.512 ± 0.010	0.869 ± 0.013
tsfresh _{def} + LR	0.543 ± 0.034	0.848 ± 0.018
tsfresh _{def} + XGBoost	0.197 ± 0.041	0.981 ± 0.006
TSFuse _{min} + 5NN	0.617 ± 0.058	0.778 ± 0.033
TSFuse _{min} + LR	0.878 ± 0.059	0.539 ± 0.060
TSFuse _{min} + XGBoost	0.265 ± 0.037	0.966 ± 0.007
TSFuse _{def} + 5NN	0.731 ± 0.026	0.705 ± 0.009
TSFuse _{def} + LR	0.917 ± 0.000	0.500 ± 0.000
TSFuse _{def} + XGBoost	0.108 ± 0.048	0.990 ± 0.005

Dataset 8b	Error rate	AUC
catch22 + 5NN	0.663 ± 0.008	0.730 ± 0.013
catch22 + LR	0.671 ± 0.010	0.814 ± 0.023
catch22 + XGBoost	0.509 ± 0.011	0.892 ± 0.005
tsfresh _{min} + 5NN	0.642 ± 0.018	0.734 ± 0.006
tsfresh _{min} + LR	0.711 ± 0.039	0.773 ± 0.039
tsfresh _{min} + XGBoost	0.550 ± 0.019	0.867 ± 0.010
tsfresh _{def} + 5NN	0.700 ± 0.002	0.719 ± 0.006
tsfresh _{def} + LR	0.704 ± 0.038	0.719 ± 0.021
tsfresh _{def} + XGBoost	0.418 ± 0.017	0.932 ± 0.006
TSFuse _{min} + 5NN	0.767 ± 0.042	0.670 ± 0.058
TSFuse _{min} + LR	0.884 ± 0.034	0.594 ± 0.078
TSFuse _{min} + XGBoost	0.480 ± 0.043	0.890 ± 0.020
TSFuse _{def} + 5NN	0.842 ± 0.027	0.624 ± 0.027
TSFuse _{def} + LR	0.917 ± 0.000	0.500 ± 0.000
TSFuse _{def} + XGBoost	0.232 ± 0.011	0.968 ± 0.007
Dataset 8c	Error rate	AUC
catch22 + 5NN	0.723 ± 0.015	0.711 ± 0.004
catch22 + LR	0.694 ± 0.023	0.810 ± 0.015
catch22 + XGBoost	0.526 ± 0.013	0.881 ± 0.008
tsfresh _{min} + 5NN	0.742 ± 0.006	0.666 ± 0.002
tsfresh _{min} + LR	0.751 ± 0.037	0.730 ± 0.016
tsfresh _{min} + XGBoost	0.668 ± 0.044	0.814 ± 0.017
tsfresh _{def} + 5NN	0.769 ± 0.002	0.695 ± 0.001
tsfresh _{def} + LR	0.758 ± 0.011	0.723 ± 0.026
tsfresh _{def} + XGBoost	0.538 ± 0.033	0.888 ± 0.020
TSFuse _{min} + 5NN	0.829 ± 0.036	0.625 ± 0.034
TSFuse _{min} + LR	0.893 ± 0.029	0.549 ± 0.061
TSFuse _{min} + XGBoost	0.535 ± 0.072	0.865 ± 0.024
TSFuse _{def} + 5NN	0.806 ± 0.003	0.631 ± 0.002
TSFuse _{def} + LR	0.917 ± 0.000	0.500 ± 0.000
TSFuse _{def} + XGBoost	0.308 ± 0.078	0.954 ± 0.016
Dataset 9a	Error rate	AUC
catch22 + 5NN	0.666 ± 0.010	0.500 ± 0.000
catch22 + LR	0.670 ± 0.005	0.500 ± 0.000
catch22 + XGBoost	0.670 ± 0.005	0.500 ± 0.000
tsfresh _{min} + 5NN	0.079 ± 0.068	0.963 ± 0.036
tsfresh _{min} + LR	0.037 ± 0.025	1.000 ± 0.000
tsfresh _{min} + XGBoost	0.041 ± 0.038	0.993 ± 0.008
tsfresh _{def} + 5NN	0.111 ± 0.060	0.954 ± 0.040
tsfresh _{def} + LR	0.048 ± 0.040	0.991 ± 0.009
tsfresh _{def} + XGBoost	0.069 ± 0.069	0.979 ± 0.021
TSFuse _{min} + 5NN	0.460 ± 0.092	0.723 ± 0.079
TSFuse _{min} + LR	0.428 ± 0.130	0.773 ± 0.069

Dataset 9a	Error rate	AUC
TSFuse _{min} + XGBoost	0.053 ± 0.054	0.995 ± 0.005
TSFuse _{def} + 5NN	0.443 ± 0.005	0.737 ± 0.006
TSFuse _{def} + LR	0.657 ± 0.008	0.500 ± 0.000
TSFuse _{def} + XGBoost	0.067 ± 0.063	0.998 ± 0.002
Dataset 9b	Error rate	AUC
catch22 + 5NN	0.499 ± 0.013	0.500 ± 0.000
catch22 + LR	0.511 ± 0.007	0.500 ± 0.000
catch22 + XGBoost	0.511 ± 0.007	0.500 ± 0.000
tsfresh _{min} + 5NN	0.442 ± 0.024	0.595 ± 0.020
tsfresh _{min} + LR	0.420 ± 0.004	0.640 ± 0.013
tsfresh _{min} + XGBoost	0.381 ± 0.017	0.659 ± 0.017
tsfresh _{def} + 5NN	0.459 ± 0.013	0.551 ± 0.014
tsfresh _{def} + LR	0.391 ± 0.003	0.651 ± 0.013
tsfresh _{def} + XGBoost	0.250 ± 0.059	0.842 ± 0.062
TSFuse _{min} + 5NN	0.483 ± 0.020	0.531 ± 0.022
TSFuse _{min} + LR	0.493 ± 0.018	0.506 ± 0.041
TSFuse _{min} + XGBoost	0.269 ± 0.010	0.819 ± 0.009
TSFuse _{def} + 5NN	0.486 ± 0.012	0.508 ± 0.022
TSFuse _{def} + LR	0.493 ± 0.011	0.500 ± 0.000
TSFuse _{def} + XGBoost	0.238 ± 0.013	0.853 ± 0.008
Dataset 9c	Error rate	AUC
catch22 + 5NN	0.517 ± 0.069	0.500 ± 0.000
catch22 + LR	0.493 ± 0.011	0.500 ± 0.000
catch22 + XGBoost	0.493 ± 0.011	0.500 ± 0.000
tsfresh _{min} + 5NN	0.532 ± 0.009	0.619 ± 0.012
tsfresh _{min} + LR	0.467 ± 0.082	0.738 ± 0.002
tsfresh _{min} + XGBoost	0.465 ± 0.023	0.750 ± 0.011
tsfresh _{def} + 5NN	0.539 ± 0.013	0.605 ± 0.017
tsfresh _{def} + LR	0.539 ± 0.016	0.656 ± 0.022
tsfresh _{def} + XGBoost	0.346 ± 0.012	0.831 ± 0.005
TSFuse _{min} + 5NN	0.544 ± 0.014	0.565 ± 0.024
TSFuse _{min} + LR	0.526 ± 0.028	0.544 ± 0.024
TSFuse _{min} + XGBoost	0.300 ± 0.024	0.858 ± 0.012
TSFuse _{def} + 5NN	0.563 ± 0.010	0.534 ± 0.004
TSFuse _{def} + LR	0.493 ± 0.011	0.500 ± 0.000
TSFuse _{def} + XGBoost	0.310 ± 0.015	0.858 ± 0.009
Dataset 10	RMSE	R ²
catch22 + 5NN	3042.795 ± 1684.703	-12.578 ± 14.613
catch22 + LR	3001.971 ± 1651.478	-14.174 ± 19.705
catch22 + XGBoost	3138.783 ± 1741.022	-14.428 ± 17.573
tsfresh _{min} + 5NN	1820.854 ± 1296.948	-2.275 ± 1.795
tsfresh _{min} + LR	424.670 ± 204.346	0.740 ± 0.200
tsfresh _{min} + XGBoost	1522.980 ± 1251.112	-1.342 ± 1.668

Dataset 10	RMSE	R ²
tsfresh _{def} + 5NN	1530.495 ± 1286.550	−1.280 ± 1.618
tsfresh _{def} + LR	482.626 ± 343.695	0.669 ± 0.417
tsfresh _{def} + XGBoost	1537.993 ± 1338.438	−1.468 ± 1.899
TSFuse _{min} + 5NN	1989.740 ± 1255.597	−3.454 ± 2.681
TSFuse _{min} + LR	419.366 ± 231.101	0.802 ± 0.149
TSFuse _{min} + XGBoost	1625.676 ± 1362.207	−1.832 ± 2.088
TSFuse _{def} + 5NN	1672.127 ± 1279.657	−1.691 ± 1.581
TSFuse _{def} + LR	807.447 ± 764.891	−0.458 ± 2.275
TSFuse _{def} + XGBoost	1515.881 ± 1295.376	−1.409 ± 1.812

Author Contributions Conceptualization: ADB, TODB, WM, JD; Methodology: ADB, TODB, WM, JD; Formal analysis and investigation: ADB, KH; Writing - original draft preparation: ADB; Writing - review and editing: TODB, KH, WM, JD; Funding acquisition: WM, JD; Resources: WM, JD; Supervision: JD.

Funding This research received funding from the KU Leuven Research Fund (C14/17/070), the Interuniversity Special Research Fund (IBOF/21/075), and the Flemish Government under the “Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen” program.

Availability of data and material The non-public datasets used in this work are available upon request.

Declarations

Conflicts of interest The authors declare that they have no conflict of interest.

Code availability TSFuse is released as a Python package: <https://github.com/arnedb/tsfuse>

References

- Bagnall, A., Dau, H.A., Lines, J., Flynn, M., Large, J., Bostrom, A., Southam, P., & Keogh, E. (2018). The UEA multivariate time series classification archive, 2018. arXiv preprint [arXiv:1811.00075](https://arxiv.org/abs/1811.00075)
- Batista, G. E., Keogh, E. J., Tataw, O. M., & De Souza, V. M. (2014). CID: an efficient complexity-invariant distance for time series. *Data Mining and Knowledge Discovery*, 28(3), 634–669.
- Boullé, M., Charnay, C., & Lachiche, N. (2019). A scalable robust and automatic propositionalization approach for bayesian classification of large mixed numerical and categorical data. *Machine Learning*, 108(2), 229–266.
- Chandrashekar, G., & Sahin, F. (2014). A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1), 16–28.
- Chen, T., & Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In: Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining, pp 785–794
- Christ, M., Kempa-Liehr, A.W., & Feindt, M. (2016). Distributed and parallel time series feature extraction for industrial big data applications. arXiv preprint [arXiv:1610.07717](https://arxiv.org/abs/1610.07717)
- Christ, M., Braun, N., Neuffer, J., & Kempa-Liehr, A. W. (2018). Time series feature extraction on basis of scalable hypothesis tests (tsfresh-a python package). *Neurocomputing*, 307, 72–77.
- Davis, J., Burnside, E.S., de Castro Dutra, I., Page, D., Ramakrishnan, R., Costa, V.S., & Shavlik, J.W. (2005). View learning for statistical relational learning: With an application to mammography. In: IJCAI, Citeseer, pp. 677–683
- De Brabandere, A., Op De Beëck, T., Schütte, K. H., Meert, W., Vanwanseele, B., & Davis, J. (2018). Data fusion of body-worn accelerometers and heart rate to predict VO2max during submaximal running. *PloS One*, 13(6), e0199509.

- Decroos, T., Schütte, K., De Beéck, TO., Vanwanseele, B., & Davis, J. (2018). AMIE: Automatic monitoring of indoor exercises. In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Springer, pp. 424–439
- Derie, R., Robberechts, P., Van den Berghe, P., Gerlo, J., De Clercq, D., Segers, V., & Davis, J. (2020). Tibial acceleration-based prediction of maximal vertical loading rate during overground running: A machine learning approach. *Frontiers in Bioengineering and Biotechnology*, 8, 33.
- Dietterich, T. G. (1998). Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation*, 10(7), 1895–1923.
- Fawaz, H. I., Forestier, G., Weber, J., Idoumghar, L., & Muller, P. A. (2019). Deep learning for time series classification: A review. *Data Mining and Knowledge Discovery*, 33(4), 917–963.
- Friedman, J. H., Popescu, B. E., et al. (2008). Predictive learning via rule ensembles. *Annals of Applied Statistics*, 2(3), 916–954.
- Friedrich, R., Siegert, S., Peinke, J., Siefert, M., Lindemann, M., Raethjen, J., et al. (2000). Extracting model equations from experimental data. *Physics Letters A*, 271(3), 217–222.
- Fukushima, K., Miyake, S., & Ito, T. (1983). Neocognitron: A neural network model for a mechanism of visual pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 5, 826–834.
- Fulcher, B.D. (2017). Feature-based time-series analysis. arXiv preprint [arXiv:1709.08055](https://arxiv.org/abs/1709.08055)
- Fulcher, B. D., & Jones, N. S. (2017). hctsa: A computational framework for automated time-series phenotyping using massive feature extraction. *Cell Systems*, 5(5), 527–531.
- Ghouaïel, N., Marteau, P. F., & Dupont, M. (2017). Continuous pattern detection and recognition in stream-a benchmark for online gesture recognition. *International Journal of Applied Pattern Recognition*, 4(2), 146–160.
- Goldberger, A. L., Amaral, L. A., Glass, L., Hausdorff, J. M., Ivanov, P. C., Mark, R. G., et al. (2000). PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals. *Circulation*, 101(23), e215–e220.
- Grzegorowski, M., & Stawicki, S. (2015). Window-based feature engineering for prediction of methane threats in coal mines. In: *Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing*, Springer, pp. 452–463
- Guo, Z., Wang, H., Liu, Q., & Yang, J. (2014). A feature fusion based forecasting model for financial time series. *PloS one*, 9(6), e101113.
- Harvey, D. Y., & Todd, M. D. (2014). Automated feature design for numeric sequence classification by genetic programming. *IEEE Transactions on Evolutionary Computation*, 19(4), 474–489.
- Helwig, N., Pignaneli, E., & Schütze, A. (2015). Condition monitoring of a complex hydraulic system using multivariate statistics. In: *2015 IEEE International Instrumentation and Measurement Technology Conference (I2MTC) Proceedings*, IEEE, pp. 210–215
- Hendrickx, K., Meert, W., Mollet, Y., Gyselinck, J., Cornelis, B., Gryllias, K., & Davis, J. (2020). A general anomaly detection framework for fleet-based condition monitoring of machines. *Mechanical Systems and Signal Processing*, 139, 106585.
- Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780.
- Horn, F., Pack, R., & Rieger, M. (2019). The autofeat Python library for automated feature engineering and selection. In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Springer, pp. 111–120
- Incel, O. D., Kose, M., & Ersoy, C. (2013). A review and taxonomy of activity recognition on mobile phones. *BioNanoScience*, 3(2), 145–171.
- Jaeger, H., & Haas, H. (2004). Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304(5667), 78–80.
- Kanter, J.M., & Veeramachaneni, K. (2015). Deep feature synthesis: Towards automating data science endeavors. In: *2015 IEEE international conference on data science and advanced analytics (DSAA)*, IEEE, pp. 1–10
- Keogh, E., & Ratanamahatana, C. A. (2005). Exact indexing of dynamic time warping. *Knowledge and Information Systems*, 7(3), 358–386.
- Khurana, U., Samulowitz, H., & Turaga, D. (2018). Feature engineering for predictive modeling using reinforcement learning. In: *Proceedings of the AAAI Conference on Artificial Intelligence*, vol 32
- Kim, K. J. (2003). Financial time series forecasting using support vector machines. *Neurocomputing*, 55(1–2), 307–319.
- Ko, M. H., West, G., Venkatesh, S., Kumar, M., (2005). Online context recognition in multisensor systems using dynamic time warping. (2005). *International Conference on Intelligent Sensors* (pp. 283–288). IEEE: Sensor Networks and Information Processing.

- Koch, M., Geraedts, V., Wang, H., Tannemaat, M., & Bäck, T. (2019). Automated machine learning for EEG-based classification of parkinson's disease patients. In: *2019 IEEE International Conference on Big Data (Big Data)*, IEEE, pp. 4845–4852
- Kroegel, M. A. (2005). On propositionalization for knowledge discovery in relational databases
- Kwapisz, J. R., Weiss, G. M., & Moore, S. A. (2011). Activity recognition using cell phone accelerometers. *ACM SIGKDD Explorations Newsletter*, 12(2), 74–82.
- Lam, H.T., Thiebaut, J.M., Sinn, M., Chen, B., Mai, T., & Alkan, O. (2017). *One button machine for automating feature engineering in relational databases*. arXiv preprint [arXiv:1706.00327](https://arxiv.org/abs/1706.00327)
- Liu, J., Zhong, L., Wickramasuriya, J., & Vasudevan, V. (2009). uWave: Accelerometer-based personalized gesture recognition and its applications. *Pervasive and Mobile Computing*, 5(6), 657–675.
- Löning, M., Bagnall, A., Ganesh, S., Kazakov, V., Lines, J., & Király, F. J. (2019). *sktime: A unified interface for machine learning with time series*. arXiv preprint [arXiv:1909.07872](https://arxiv.org/abs/1909.07872)
- Lou, Y., Caruana, R., Gehrke, J., & Hooker, G. (2013). Accurate intelligible models with pairwise interactions. In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 623–631
- Lubba, C. H., Sethi, S. S., Knaute, P., Schultz, S. R., Fulcher, B. D., & Jones, N. S. (2019). catch22: Canonical time-series characteristics. *Data Mining and Knowledge Discovery*, 33(6), 1821–1852.
- Meert, W., Hendrickx, K., & Van Craenendonck, T. (2020). wannesm/tdistance (version v2.0.0). <https://doi.org/10.5281/zenodo.3981067>
- Mierswa, I. (2005). Automatic feature extraction from large time series. In: *Classification—the Ubiquitous Challenge*, Springer, pp. 600–607
- Moody, G. (2004). Spontaneous termination of atrial fibrillation: a challenge from PhysioNet and Computers in Cardiology 2004. In: *Computers in Cardiology, 2004*, IEEE, pp. 101–104
- Munson, M. A. (2012). A study on the importance of and time spent on different modeling steps. *ACM SIGKDD Explorations Newsletter*, 13(2), 65–71.
- Op De Beéck, T., Meert, W., Schütte, K., Vanwanseele, B., & Davis, J. (2018). Fatigue prediction in outdoor runners via machine learning and sensor fusion. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 606–615
- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V. et al. (2011). Scikit-learn: Machine learning in Python. *The Journal of machine Learning research* 12:2825–2830
- Rendell, L. (1985). Substantial constructive induction using layered information compression: Tractable feature formation in search, vol 1198. Citeseer
- Roggen, D., Calatroni, A., Rossi, M., Holleczer, T., Förster, K., Tröster, G., Lukowicz, P., Bannach, D., Pirkel, G., & Ferscha, A. et al. (2010). Collecting complex activity datasets in highly rich networked sensor environments. In: *2010 Seventh international conference on networked sensing systems (INSS)*, IEEE, pp. 233–240
- Ruiz, A.P., Flynn, M., Large, J., Middlehurst, M., Bagnall, A. (2020). The great multivariate time series classification bake off: a review and experimental evaluation of recent algorithmic advances. *Data Mining and Knowledge Discovery* pp. 1–49
- Schouterden, J., Davis, J., Blockeel, H. (2019). LazyBum: Decision tree learning using lazy propositionalization. In: *International Conference on Inductive Logic Programming*, Springer, pp. 98–113
- Schreiber, T., & Schmitz, A. (1997). Discrimination power of measures for nonlinearity in a time series. *Physical Review E*, 55(5), 5443.
- Schroeder, V., Evans, E. D., Wu, Y. C. M., Voll, C. C. A., McDonald, B. R., Savagatrup, S., & Swager, T. M. (2019). Chemiresistive sensor array and machine learning classification of food. *ACS Sensors*, 4(8), 2101–2108.
- Schuller, B., Reiter, S., & Rigoll, G. (2006). Evolutionary feature generation in speech emotion recognition. In: *2006 IEEE International Conference on Multimedia and Expo, IEEE*, pp. 5–8
- Shokoohi-Yekta, M., Hu, B., Jin, H., Wang, J., & Keogh, E. (2015). Generalizing dynamic time warping to the multi-dimensional case requires an adaptive approach. Citeseer
- van den Bosch, S. (2017). Automatic feature generation and selection in predictive analytics solutions. *Master's thesis, Faculty of Science, Radboud University*, 3(1), 1–3.
- Wang, J., Balasubramanian, A., Mojica de la Vega, L., Green, J. R., Samal, A., & Prabhakaran, B. (2013). Word recognition from continuous articulatory movement time-series data using symbolic representations
- Weyand, P. G., Kelly, M., Blackadar, T., Darley, J. C., Oliver, S. R., Ohlenbusch, N. E., et al. (2001). Ambulatory estimates of maximal aerobic power from foot-ground contact times and heart rates in running humans. *Journal of Applied Physiology*, 91(1), 451–458.

- Yuan, B., Tan, Y. J., Mudunuru, M. K., Marcillo, O. E., Delorey, A. A., Roberts, P. M., et al. (2019). Using machine learning to discern eruption in noisy environments: A case study using CO₂-driven cold-water geyser in chimayó, new mexico. *Seismological Research Letters*, 90(2A), 591–603.
- Yurtman, A., & Barshan, B. (2017). Activity recognition invariant to sensor orientation with wearable motion sensors. *Sensors*, 17(8), 1838.
- Zhang, M., & Sawchuk, A. A. (2012). USC-HAD: a daily activity dataset for ubiquitous activity recognition using wearable sensors. In: *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pp. 1036–1043

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Arne De Brabandere¹ · Tim Op De Beéck² · Kilian Hendrickx³ · Wannes Meert¹ · Jesse Davis¹

Tim Op De Beéck
tim@runeasi.ai

Kilian Hendrickx
kilian.hendrickx@siemens.com

Wannes Meert
wannes.meert@kuleuven.be

Jesse Davis
jesse.davis@kuleuven.be

¹ Department of Computer Science, Leuven, Belgium

² RunEASI NV, Boutersem, Belgium

³ Siemens Digital Industries Software, Leuven, Belgium