Raising the ClaSS of Streaming Time Series Segmentation

Arik Ermshaus
Humboldt-Universität zu Berlin
Berlin, Germany
ermshaua@informatik.hu-berlin.de

Patrick Schäfer
Humboldt-Universität zu Berlin
Berlin, Germany
patrick.schaefer@hu-berlin.de

Ulf Leser
Humboldt-Universität zu Berlin
Berlin, Germany
leser@informatik.hu-berlin.de

ABSTRACT

Ubiquitous sensors today emit high frequency streams of numerical measurements that reflect properties of human, animal, industrial, commercial, and natural processes. Shifts in such processes, e.g. caused by external events or internal state changes, manifest as changes in the recorded signals. The task of streaming time series segmentation (STSS) is to partition the stream into consecutive variable-sized segments that correspond to states of the observed processes or entities. The partition operation itself must in performance be able to cope with the input frequency of the signals. We introduce ClaSS, a novel, efficient, and highly accurate algorithm for STSS. ClaSS assesses the homogeneity of potential partitions using self-supervised time series classification and applies statistical tests to detect significant change points (CPs). In our experimental evaluation using two large benchmarks and six real-world data archives, we found ClaSS to be significantly more precise than eight state-of-the-art competitors. Its space and time complexity is independent of segment sizes and linear only in the sliding window size. We also provide ClaSS as a window operator with an average throughput of $1k$ data points per second for the Apache Flink streaming engine.

PVLDB Reference Format:
Arik Ermshaus, Patrick Schäfer, and Ulf Leser. Raising the ClaSS of Streaming Time Series Segmentation. PVLDB, 17(8): 1953 - 1966, 2024.
doi:10.14778/3659437.3659450

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ermshaua/classification-score-stream.

1 INTRODUCTION

Over the past two decades, the decreasing costs of sensors and the growing digitalization of industry, science, and society has led to an enormous increase in applications that analyse streams of sensor recordings. For example, modern smartphones contain inertial measurement units (IMUs) with triaxial accelerometers, gyroscopes, and magnetometers that can track human activities [5]. Seismology relies on globally distributed stations to provide high-resolution waveform recordings used for earthquake detection and early warning [64]. In cardiology, electrocardiographs (ECG) capture heart beats from subjects over long periods of time to obtain insights into cardiac dynamics such as arrhythmias [39]. Regardless of the domain, the underlying sensors emit continuous sequences of real-valued measurements at a given frequency, called sensor data (series) or *time series* (TS). The literature offers a rich selection of technologies to store, manage, analyse, visualize and search in collections of TS [1, 15, 37, 62, 66, 67]. Common basic operations are the detection of unusual stretches called *anomalies* [45], of repetitive structures called *motifs* [52], and of homogeneous subsequences called *segments* [23].

TS methods can be broadly classified into batch or streaming. Methods for the batch analysis of TS, used in applications such as gait or sleep stage analysis [30, 57], can largely ignore latency, runtime and memory requirements and use complex preprocessing based on global statistics (e.g. frequency filtering or signal decomposition). This is different in the streaming case, where infinite TS must be processed in real-time relative to the measurement frequency and where the complexity of operations must not depend on the length of sequences [60].

This is especially unfortunate for the task of TS segmentation (TSS), a common preprocessing step between data collection [49] and knowledge discovery from TS [38]. TSS allows inferring the latent states of an underlying process by analysing sensor measurements, as signal shifts from one segment to another are assumed to be caused by state changes in the process being monitored, such as a transition from one human activity to another or from one machine state to another. In the batch case, TSS aims to partition a given TS into consecutive regions such that each region is homogeneous in itself yet sufficiently different from the neighbouring regions. It is typically performed by focusing on the detection of change points (CPs) separating segments [3]. State-of-the-art methods for TSS rely on global statistics of the TS, value distributions, densities or learned features [57], and often exhibit a high computational complexity. Recent accurate contributions, e.g. FLUSS [23] or ClaSP [16], are quadratic in runtime regarding the TS length.

For the streaming case such statistics are not available and such complexities clearly are not feasible, i.e., for segmenting streams of TS (STSS) [27, 41, 43]. Real-time processing is essential for STSS, yet challenging. For instance, IoT devices may emit measurements with hundreds of Hertz (Hz) [4, 24, 54, 64]. STSS requires algorithms that process data points faster than they arrive, utilizing only a constant amount of memory. The problem was first formulated by Kifer et al. in the context of change detection [32]. The basic approach is to constrain the analysis to the last d observations using a sliding window and to continuously emit detected CPs, which each defines the end of a segment [3]. Following this seminal work, many follow-up methods considered stream segmentation as a part of IoT workflows [10, 12, 61] or studied drift or CP detection [3, 20]. The best methods, however, only work on temporal data with a suitable value distribution (e.g. BOCD [2]), can only detect very limited change or drift types (e.g. NEWMA [31]), or rely on thresholding

Proceedings of the VLDB Endowment, Vol. 17, No. 8 ISSN 2150-8097.
doi:10.14778/3659437.3659450

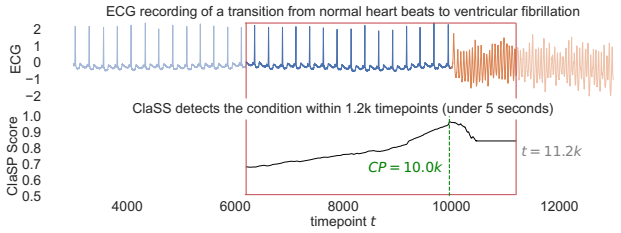

Figure 1: An electrocardiogram (ECG) recording of a human subject demonstrating the transition from normal heartbeats (in blue) to ventricular fibrillations (in orange) [42]. The ClaSS algorithm continuously scores the TS stream within a sliding window (shown in red), and at $t = 11.2k$ a significant change in the signal shape is detected and immediately reported to the user. This split effectively divides the stream into a fully processed segment and one that evolves.

as segmentation procedures (e.g. FLOSS [23]), which is not robust for real-world signals.

We present ClaSS (Classification Score Stream), a domain-agnostic, highly accurate and efficient algorithm that approaches STSS as a self-supervised learning problem. It continuously scores the homogeneity of hypothetical sliding window split points and identifies statistically significant CPs using hypothesis testing to find the ends and beginnings of segments. Notably, the algorithm only scores the last detected segment and tests if it should be further split into two, thereby reducing model complexity and saving runtime. The core of this approach lies in an efficient calculation of ClaSP (Classification Score Profile) [17], which was originally constrained to batch analysis. ClaSS achieves much higher efficiency than ClaSP, as necessary for the streaming case, by efficiently cross-validating a novel streaming k-nearest neighbour (k-NN) that reuses the results of calculations from previous overlapping sliding windows. Time and space complexity of ClaSS are both linearly dependent only on the sliding window size, thus fulfilling the requirements of STSS. While some competitors exhibit sublinear sliding window update runtimes surpassing ClaSS, their segmentation models are restricted to basic methods (e.g. statistical parameter deviation), limiting their accuracy. Conversely, ClaSS trades this runtime gap to incorporate more advanced data mining techniques, resulting in significantly higher accuracy at sufficient speeds.

Figure 1 exemplifies how ClaSS segments a sliding window into homogeneous regions. The data set [42] shows an ECG recording sampled at 250 Hz from a human subject, who experienced ventricular fibrillations after 40 seconds ($t = 10k$). The global maximum in the profile (Figure 1 bottom) captures the start of the condition, and is detected and reported as a significant change around 5 seconds after the ventricular fibrillations begin ($t = 11.2k$); dividing the stream into segments with normal and abnormal cardiac activity.

Specifically, this paper makes the following contributions:

(1) We introduce ClaSS, a novel, efficient and domain-agnostic method for STSS, which scores sliding windows using self-supervised TS classification to detect and report statistically significant CPs with low latency. The scoring process annotates the sliding window with the likelihood of CPs. Besides being necessary for STSS, this makes it easy for humans to understand and suitable for decision-support systems.
(2) We present two technical advancements that enable ClaSS to meet the stringent performance criteria for STSS: the first exact streaming TS k-nearest neighbour algorithm that runs in $O(k \cdot d)$ for a single sliding window (of length d) update, substantially improving upon the current-best $O((k + \log d) \cdot d)$ solution [23], and a novel algorithm for cross-validating a self-supervised k-NN classifier in $O(d)$, outperforming the prior best $O(d^2)$ approach [17].
(3) We analysed the accuracy of ClaSS using 592 real-world TS from two benchmarks and six experimental studies. Compared to eight state-of-the-art competitors (FLOSS, DDM, ChangeFinder, NEWMA, BOCD, HDDM, ADWIN and a sliding window baseline), ClaSS significantly outperforms all other competitors, exhibits the highest overall segmentation accuracy and improves the state of the art by 13.7 pp (percentage points).

We make all of our used source codes, including a stand-alone Python implementation and a comparably fast Apache Flink window operator of ClaSS, the evaluation framework, Jupyter Notebooks, as well as all experiment data and visualizations openly available on our supporting website [11] to foster the reproducibility of our findings and replicability for follow-up works. The rest of the paper is structured as follows. In Section 2, we provide background and definitions for this work. Section 3 introduces our ClaSS algorithm and its components. Section 4 describes results from an extensive experimental evaluation and Section 5 details related work. Finally, we conclude in Section 6.

2 BACKGROUND AND DEFINITIONS

This section formally introduces the concepts of time series streams, sliding windows, subsequences, the streaming time series segmentation (STSS) problem and self-supervision. We also briefly recapitulate the idea behind ClaSP [17] upon which ClaSS is based.

Definition 1. *A TS* stream *S produces a new real-valued data point $t_\tau \in \mathbb{R}$ at evenly spaced time intervals, which enqueues in a continuous sequence $< \ldots, t_{\tau-1}, t_\tau >$ of values. The data points are also called observations or measurements.*

The main characteristic of a TS stream is its infinite length. However, in practice, only a finite number of measurements can be stored and processed at any time (i.e., t_τ). This necessitates efficient data mining techniques that can quickly analyse incoming data. We focus on univariate streams that are sampled at equidistant time stamps (e.g. 50 Hz), with the same temporal duration between consecutive measurements. It is worth emphasizing that any finitely long TS can be treated as a TS stream and analysed accordingly.

Definition 2. *Given a TS stream S, a* sliding window *$S_{\tau-d+1,\tau}$ is a buffer of size d that stores the latest d data points produced by S. As a new data point appears in S, the corresponding $S_{\tau-d+1,\tau}$ expels the oldest observation and appends the youngest one.*

The size of the window d is a hyper-parameter that will be discussed in Subsection 3.5. This choice directly affects the amount of information and thus the runtime and memory of the algorithms applied to $S_{\tau-d+1,\tau}$. For an example of a sliding window, see Figure 2.

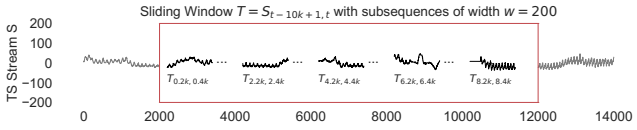

Figure 2: A TS stream S from which the last $d = 10k$ observations are buffered in a sliding window $T = S_{\tau-d+1,\tau}$, depicted as the red frame. Older (or yet to arrive) data points are greyed out. The sliding window is further cut into subsequences of width $w = 200$, to be analysed for segmentation.

Note that we represent a window as a finite TS $T = S_{\tau-d+1,\tau}$ of size d, from which we can access values at offsets $[1 \dots d]$.

Definition 3. *Given a sliding window $T = S_{\tau-d+1,\tau}$, a subsequence $T_{s,e}$ of T with start offset s and end offset e consists of the contiguous values of T from position s to position e, i.e., $T_{s,e} = (t_{\tau-d+s}, \dots, t_{\tau-d+e})$ with $1 \leq s \leq e \leq d$. The length of $T_{s,e}$ is $|T_{s,e}| = e - s + 1$.*

We refer to the length of subsequences as *width*. Figure 2 illustrates subsequences in a sliding window. Periodic TS streams generally repeat a subsequence of values after a constant period of time, which we refer to as a temporal pattern (or period). However, periods can vary or drift, and local parts of TS may differ in terms of period length, shape or amplitude.

Definition 4. *A* segmentation *of a TS stream S produces the latest completed segment of S as a variable-sized interval $s_{-1} = [t_{c_{-2}}, \dots, t_{c_{-1}}]$ where $c_{-2} < c_{-1} \leq \tau$ are the two last discovered change points (or splits). For consistency, we consider the first observed value from S as the first change point.*

The location and amount of CPs in S is unknown and must be inferred by evaluating the last d data points in the sliding window $S_{\tau-d+1,\tau}$. Change points and the respective segments are continuously reported until S is aborted. Note that, by definition, the latest reported segment may stretch until before the current window.

Definition 5. *The problem of streaming time series segmentation (STSS) is to find a meaningful segmentation of a given TS stream S such that the change points between two subsequent segments correspond to state changes in the observed process.*

The notion of being *meaningful* depends on the domain, typically relating to the shape or value distribution of potential segments. Following [22], we assume that a natural process has discrete states that lead to changes in measured values. An example are sequences of human emotional states, that can switch between (a) resting, (b) amused or (c) stressed and influence biosignals, as studied in [54]. The task of STSS would be to track e.g. the last 10 seconds of a subject's respiration signal and report the last completed segment (e.g. resting), as soon as another one (e.g. stressed) emerges. This differs from other problems such as trend detection [57].

STSS algorithms must maintain efficient data structures with constant memory requirements and minimal latency to be able to report segmentations in real-time. They must also possess the ability to decide when they have seen enough data points to predict a CP, using only the limited information available from $S_{\tau-d+1,\tau}$.

2.1 Self-supervised Time Series Segmentation

In order to detect the last completed segment, we must be able to differentiate it from the newly evolving one. ClaSS enumerates potential binary segment candidates (splits of $S_{\tau-d+1,\tau}$ into two parts). We assess the distinctiveness between candidates using a heterogeneity score, selecting the segmentation with the most dissimilar segments, provided the score surpasses a predefined threshold.

Our scoring methodology draws inspiration from self-supervised change analysis, as formulized by Hido et al. [26]. In self-supervised learning, an unsupervised learning variant, the data itself generates supervision labels. Consider two data sets, X_A and X_B, representing subsequences from different segment candidates. We assign label 0 to X_A instances and 1 to X_B instances, facilitating a binary classification evaluation using cross-validation. A TS classifier, trained on labelled subsequences, predicts labels for unlabelled instances. We employ cross-validation, wherein the classifier is trained on $(k-1)$ portions of the data and tested on the remaining part. A k-NN classifier, for instance, collects and aggregates subsequence labels from train instances. This process repeats k times, covering all combinations, with the average of the k evaluation scores (e.g., F1-scores) representing the classifier's performance. This value measures the classifier's ability to distinguish between data sets X_A and X_B. A high score implies high dissimilarity and unique characteristics between the segments, while a lower score indicates similarities, suggesting the data sets may belong to the same segment.

2.2 Classification Score Profile

ClaSS is based on the idea of the self-supervised TSS algorithm "Classification Score Profile" (ClaSP), as introduced for the batch setting in [51]. We briefly recapitulate the concept of ClaSP. In Section 3, we describe how ClaSS efficiently computes ClaSP to address the streaming case.

Definition 6. *Given a TS T, $|T| = n$ and a subsequence width w, a ClaSP is a real-valued sequence of length $n - w + 1$, in which the i-th value is the cross-validation score $c \in [0, 1]$ of a classifier trained on a binary classification problem with overlapping labelled subsequences $[(T_{1,w}, \mathbf{0}), \dots, (T_{i-w+1,i}, \mathbf{0}), (T_{i-w+2,i+1}, \mathbf{1}), \dots (T_{n-w+1,n}, \mathbf{1})]$, with labels* 0 *and* 1.

Conceptually, a ClaSP is the result of a sequence of self-supervised TS classifications, summarized in a profile that annotates T, where every offset (or split point) i reports how well a TS classifier can differentiate the left from the right subsequences (see Figure 1 bottom). ClaSP quantifies the heterogeneity between potential segments as a profile.

The main drawback of ClaSP is its high runtime complexity of $O(n^2)$. Directly applying it for the streaming case on high-frequency streams with a sliding window (of length d) is impracticable, as it requires $O(d^2)$ computations for each new observation in the TS stream S. Furthermore, such an approach would force the method to take decisions on CPs only based on the current sliding window, which leads to false positives.

3 CLASSIFICATION SCORE STREAM

We propose *Classification Score Stream (ClaSS)*, a novel method for fast and accurate STSS. ClaSS uses a sliding window to update a

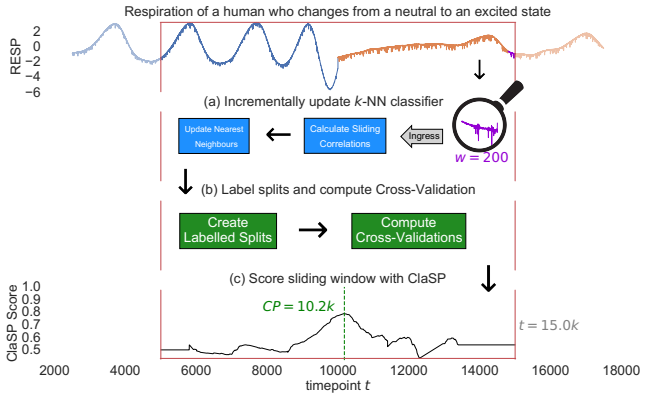

Figure 3: The conceptual ClaSS workflow for a human respiration recording that captures the transition from a neutral to an excited state [54]. (a) The streaming k-NN classifier in ClaSS is updated with the newest subsequence (magenta). (b) For every possible offset, the sliding window (red) is transformed into hypothetical binary classification problems evaluated using cross-validation. (c) The result, ClaSP, annotates the sliding window.

Algorithm 1 Classification Score Stream

```
   this.N ← array of length (d − w + 1) × 3        ▷ k-NN indices
   this.C ← array of length (d − w + 1) × 3        ▷ k-NN correlations
 1: procedure CLASS(S, d)
 2:     cp_l ← d
 3:     w ← LEARN_SUBSEQUENCE_WIDTH(S, d)
 4:     while HAS_NEXT(S) do
 5:         S_{τ−d+1,τ} ← retrieve last d time points from S
 6:         cp_l ← MAX(1, cp_l − 1)    ▷ Account for shift in S_{τ−d+1,τ}
 7:         UPDATE_STREAMING_KNN(this.C, this.N, S_{τ−d+1,τ}, w, 3)
 8:         CLASP ← CROSS_VAL_SCORES(this.N_{cp_l,d−w+1}, w)
 9:         if HAS_SIGNIFICANT_CP(CLASP) then
10:             cp ← cp_l + ARGMAX(CLASP) − 1
11:             REPORT(τ − d + cp)
12:             cp_l ← cp
13:         end if
14:     end while
15: end procedure
```

streaming k-nearest neighbour (k-NN) classifier (with correlation as similarity measure) from continuous TS streams, computing the homogeneity of hypothetical sliding window splits and applying hypothesis testing to determine statistically significant CPs. A high-level overview of the workflow of ClaSS is illustrated in Figure 3 and presented by pseudocode in Algorithm 1.

The method takes a time series stream S and the size of the sliding window d as inputs, and first learns a subsequence width w as a model-parameter from the first d observations in the stream (line 3, see Subsection 3.4). It then processes a single observation from S at a time (line 4). The procedure stores the last d data points, its sliding window, in the sequence $S_{\tau-d+1,\tau}$ (line 5) and updates the position of the last CP cp_l that denotes the beginning index of the yet unsegmented values (line 6). For consistency reasons, we consider the first observed value from S as the first CP. Note that $S_{\tau-d+1,\tau}$ contains a prefix of $d-1$ known data points while only the last measurement is new. We exploit this property later to speed up computation. ClaSS maintains a 2-dimensional k-NN sliding window profile N and pairwise Pearson correlations C (line 7). N maps the i-th subsequence $T_{i,i+w-1}$ to its k-NN subsequences in $S_{\tau-d+1,\tau}$. C stores the pairwise correlations between a subsequence and its k-NNs. ClaSS computes the $d - cp_l - w + 2$ cross-validation scores for the most recent unsegmented observations in S (line 8), leaving only the newest $w-1$ observations unscored. This scoring process enables the method to determine the homogeneity of all hypothetical splits since the last CP cp_l. This step is also the most time-consuming component of the algorithm. Every local maximum in this profile marks a potential CP, as it distinguishes between the TS parts to its left and right with high accuracy. ClaSS checks for a significant CP cp, and if so, immediately reports its time point $\tau - d + cp$ to the user, resetting the last CP cp_l (lines 9–13). The last discovered segment is then easily extracted with the last and current CP. The applied hypothesis testing is conservative in its predictions and reports CPs with high accuracy. The segmentation is then repeated as long as S produces new observations.

In the following subsections, we provide the details of the most important steps performed by ClaSS. We describe in Subsections 3.1 and 3.2 how to update and evaluate the k-NN classifier as new observations arrive, 3.3 shows how we apply statistical tests to filter for false positive CPs, 3.4 explains how ClaSS automatically learns w, and 3.5 discusses the setting of the hyper-parameter d. Finally, Subsection 3.6 analyses the runtime and space complexity of ClaSS.

3.1 Streaming k-Nearest Neighbours

In an offline setting, the computation of the k-NN profile can be delegated to a pre-processing step, which is then used for scoring hypothetical splits of the TS. This computation can be efficiently executed using various exact or approximate optimization techniques [53, 69], GPUs [68], or index structures [36]. However, in a streaming context, such pre-processing becomes challenging as $S_{\tau-d+1,\tau}$ and k-NNs evolve with each time step. Upon the arrival of a new data point, it is ingressed into the sliding window $S_{\tau-d+1,\tau}$ at position d, while all preceding data points shift left by one, possibly egressing the oldest observation. The prevalent optimization techniques fall short in accommodating sliding windows, or require a-priori construction of prediction models or complex data structures, necessitating continuous updates.

We propose the first exact streaming TS k-NN algorithm that runs in $O(k \cdot d)$, substantially improving upon the fastest exact algorithm that requires $O((k + \log d) \cdot d)$ [23]. The central idea is to establish and maintain data structures, as the stream evolves, to incrementally compute k-NNs and update C and N accordingly. To do so, we have to perform three key operations: (a) calculate and store the k-NNs for the current (latest) subsequence to insert it into the data structure; (b) shift the existing k-NNs left and deal with out-of-range references that point out of the window; and (c) update the outdated existing k-NNs that may now point to the current subsequence. We first describe the mathematics of the similarity measure computation used for k-NN determination, and then

specify how to efficiently implement steps (a) to (c) in Algorithm 2. The workflow is visualized in Figure 4 (a–b).

Similarity Calculation: For every incoming data point, we determine the k-NNs between the newest subsequence $T_{d-w+1,d}$ and the maximal $d-w+1$ many subsequences (of size w) in $T = S_{\tau-d+1,\tau}$ by calculating their mutual correlations. This can be naively computed in $O(d \cdot w)$ or optimized using Fast Fourier Transform (FFT) in $O(d \log d)$ [66], which is the basis of [23]. However, we can further improve the efficiency of this computation to $O(d)$ by adapting ideas from the STOMP algorithm [69] to the streaming setting. The Pearson correlation $c^w_{i,j}$ (Equation 4) between two w-length subsequences starting at offset i and j in T can be re-written using the dot product $q^w_{i,j}$ [40]. This definition mainly depends on the w-length subsequence means μ^w, standard deviations σ^w and dot products q^w. Rakthanmanon et al. [47] showed that μ^w_l and σ^w_l can be computed in $O(1)$ from μ^w_{l-1} and σ^w_{l-1} (independent of w) using so-called differencing cumulative running sums (Equation 1 and 2).

$$\mu^w_l = \frac{1}{w} \cdot \left(\text{CUMSUM}(T_{1,l+w-1}) - \text{CUMSUM}(T_{1,l-1})\right) \tag{1}$$

$$\sigma^w_l = \sqrt{\frac{1}{w} \cdot \left(\text{CUMSUM}^2(T_{1,l+w-1}) - \text{CUMSUM}^2(T_{1,l-1})\right) - (\mu^w_l)^2} \tag{2}$$

Furthermore, Zhu et al. [69] demonstrated that the dot product $q^w_{i,j}$ can be calculated in $O(1)$ from $q^w_{i-1,j-1}$ (also independent of w) by reusing dot products from previous subsequences. Utilizing these two findings from batch analysis, we establish them for the streaming setting to compute the Pearson correlations between the newest subsequence $T_{d-w+1,d}$ and its $d-w$ predecessors in $O(d)$ based on accessible information from the previous update.

$$q^w_{i,j} = q^{(w-1)}_{i,j} + T_{i+w-1} \cdot T_{j+w-1} \tag{3}$$

$$c^w_{i,j} = \frac{q^w_{i,j} - w\mu^w_i\mu^w_j}{w\sigma^w_i\sigma^w_j} \tag{4}$$

$$q^{(w-1)}_{i,j} = q^w_{i-1,j-1} - T_{i-1} \cdot T_{j-1} \tag{5}$$

To do so, we first compute the means μ^w and standard deviations σ^w from differenced (squared) running sum sliding windows. We then reuse the dot products between the $(w-1)$-length subsequences $T_{i,i+w-2}$ and $T_{d-w+1,d-1}$ from the last update, and add $T_{i+w-1} \cdot T_d$ to obtain the w-length dot products needed for the current iteration (Equation 3). Using the means, standard deviations, and dot products, we calculate the correlations (Equation 4) to determine the k-NNs for the current subsequence $T_{d-w+1,d}$. We then subtract $T_i \cdot T_{d-w+1}$ from the dot products to prepare them for the next update (Equation 5). Keeping the dot products updated, enables us to continuously reuse them to calculate correlations.

The similarity measure used in the streaming k-NN is not necessarily restricted to Pearson correlation; it can easily be adapted to (dis-)similarity functions that can be expressed with dot products, such as (complexity-invariant) Euclidean distance [6]. We implement multiple measures that cover different stream properties.

Incremental k-NN Calculation: In the batch setting, we can calculate initial dot products for all subsequences using FFT [66] and then update them to calculate k-NNs [69]. This preprocessing is not possible in the streaming setting, where we need to compute and update k-NNs as soon as new data points arrive and old ones are evicted. Therefore, we first enlarge the dot products and incrementally update them as outlined.

Algorithm 2 Streaming k-Nearest Neighbors

```
this.R = array of length d                                   ▷ Cumsums
this.R² = array of length d                                  ▷ sqrd. Cumsums
this.Q = array of length d − w + 1                           ▷ Dot products
1:  procedure CALC_KNN(S_{τ−d+1,τ}, w, k)
2:      T, start, end ← S_{τ−d+1,τ}, d − LENGTH(S_{τ−d+1,τ}) + 1, d − w + 1
3:      if LENGTH(T) < w then return null, null end if
4:      μ ← MEAN(this.R)                                     ▷ Eqn. 1
5:      σ ← STD(this.R, this.R²)                             ▷ Eqn. 2
6:      if start > 1 then
7:          this.Q_start ← DOT(T_{start,start+w−2}, T_{end,d−1})
8:      end if
9:      add T_{start+w−1,d} · T_d to this.Q_{start,end}      ▷ Eqn. 3
10:     corr ← PEARSON(this.Q, w, μ, σ)                      ▷ Eqn. 4
11:     knn ← ARGKMAX(corr, k, w)
12:     subtract T_{start,end} · T_end from this.Q_{start,end}   ▷ Eqn. 5
13:     return corr, knn
14: end procedure

15: procedure UPDATE_STREAMING_KNN(C, N, S_{τ−d+1,τ}, w, k)
16:     SHIFT_ADD_LAST(this.R, this.R_d + S_{−1})
17:     SHIFT_ADD_LAST(this.R², this.R²_d + S²_{−1})
18:     corr, knn ← CALC_KNN(S_{τ−d+1,τ}, w, k)
19:     if LENGTH(S_{τ−d+1,τ}) < w + k then return end if
20:     SHIFT_ADD_LAST(C, corr[knn]), SHIFT_ADD_LAST(N, knn)
21:     N_{1,d−w} ← N_{1,d−w} − 1
22:     mask ← CHANGED_NN_POS(C, N, corr, knn)
23:     UPDATE(C, mask, corr), UPDATE(N, mask, d − w + 1)
24: end procedure
```

Algorithm 2 takes the k-NN correlations C and k-NN indices N, the sliding window $S_{\tau-d+1,\tau}$, the subsequence width w and number of neighbours k as input from Algorithm 1. It maintains two (squared) running sum sliding windows R (or rather R^2) as class variables, which it updates and uses to calculate the Pearson correlation efficiently (lines 16–17). Subsequently, the algorithm calculates the correlations between the newest subsequence $T_{d-w+1,d}$ and the maximal $d-w+1$ subsequences (of size w) in $S_{\tau-d+1,\tau}$ to identify its k-NNs. The CALC_KNN subroutine (lines 1–14) first computes the *start* and *end* index of the data contained in the sliding window (line 2) and then calculates the $d-w+1$ means μ and standard deviations σ with R (or rather R^2) (lines 4–5). Similarly, the procedure maintains $d-w+1$ dot products between the $(w-1)$-length subsequences $T_{i,i+w-2}$ and $T_{d-w+1,d-1}$ at the i-th offset as a sliding window class variable Q. For the first d data points, it continuously enlarges Q to include the correct $(w-1)$-length dot products (lines 6–8). The algorithm adds $T_{start+w-1,d} \cdot T_d$ to $Q_{start,end}$ to obtain w-length dot products (line 9). It then calculates the correlations between the newest and all other subsequences and determines its nearest neighbours with k sequential searches,

considering an exclusion radius of the last $\frac{3}{2}w$ observations to avoid trivial matches (lines 10–11). Lastly, the subroutine subtracts $T_{start,end} \cdot T_{end}$ from Q to restore the $(w-1)$-length dot products for the next update and reports the correlations and k-NNs (lines 12–13) to UPDATE_STREAMING_KNN. Being able to provide the dot products in every iteration, is the central optimization that leads to linear runtime, as opposed to recomputing them in log-linear time [23].

Figure 4 (a) shows an example of this process. In the last column, the procedure stores the mean and standard deviation for the newest subsequence (magenta) and in the last two rows its pairwise dot products and correlations with all previous subsequences.

k-NN Shift: Having the k-NNs of the newest subsequence calculated, the procedure updates the correlations C and offsets N for the newest subsequence $T_{d-w+1,d}$ (line 20) and shifts the existing ones left accordingly. This leaves the prior $d-w$ to be adjusted. The algorithm decreases their offsets in N by one, to account for the shift (line 21), potentially producing negative out-of-range indices that point to subsequences outside the sliding window. To avoid this, we could constrain the nearest neighbour direction, as proposed in [23]. However, for the k-NN classifier in ClaSS, we do not even need the actual subsequences, but only their offsets, which by design belong to class zero if they are negative. Thus, we may safely ignore this issue.

k-NN Update: Lastly, the algorithm checks if the newest subsequence is one of the k-NNs of the existing subsequences in T. If so, the correlations and offsets are updated (lines 22–23). This can be done efficiently by locating the offsets that have k-NNs with lower or equal correlations compared to the newest subsequence, and inserting it in order of descending correlation, while expelling the least correlated one. Subsequently, the correlations C and offsets N are updated with the new observation.

Figure 4 (b) illustrates an example of the updated offsets in N. The last column contains the 3-NN for the newest subsequence, and it can now be a NN of previous subsequences.

3.2 Scoring the Sliding Window

The basic idea of self-supervised learning for obtaining scores for hypothetical split points is to first assign artificial ground truth labels to each subsequence up to (after) a split point, assigning them to class zero (one). These labels are then used in a second step to create the predictions of the classifier for every subsequence by the k-NN rule, collecting and aggregating ground truth classes into majority labels. In a third and final step, we calculate a classification score with both the ground truth and predicted labels. This calculation is repeated for all possible splits, thus creating the classification score profile (ClaSP).

For the at most $d-w+1$ subsequences in N, the implementation in [17] computes a single classification score in $O(d)$, re-using the class-independent k-NN offsets and relabelling them according to the changing ground truth labels. However, since this cross-validation is executed $(d-2\cdot w-1)$ times, it becomes inefficient in the streaming setting, resulting in $O(d^2)$ computations at the arrival of a single data point.

We propose a novel algorithm for cross-validating a self-supervised k-NN classifier that runs in $O(d)$ time, exploiting the observation that the label configurations for two consecutive splits

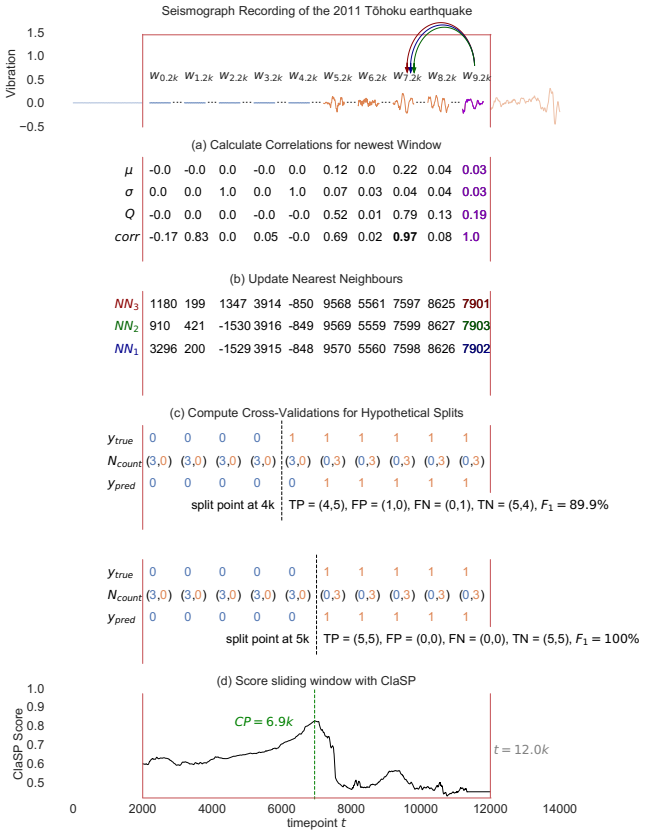

Figure 4: A workflow example for the k-NN classifier update and cross-validation computation in ClaSS. The TS stream contains the beginning of the 2011 Tōhoku earthquake seismogram, captured at Black Forest Observatory [7]. (a) The streaming 3-NN updates its means, standard deviations, and dot products to calculate the correlations between the latest subsequence (magenta) and the previous ones. (b) The 3-NN correlations and offsets are updated with the three highest correlations and their locations. (c) The sliding window is repeatedly divided into hypothetical splits and the updated k-NN classifier is evaluated to calculate the resulting classification scores (d) that form the ClaSP.

only minimally differ. It computes this delta in amortized constant runtime, as opposed to creating new labels for each cross-validation, substantially accelerating the process while being exact. This key idea is implemented in Algorithm 3, and visualized in Figure 4 (c–d).

Algorithm 3 receives the sliding window k-NN indices N and the subsequence width w as input from Algorithm 1. It initializes the ground truth and predicted labels y_{true} and y_{pred} as two arrays of $d-w+1$ ones, and stores the label counts for each subsequence in a 2-dimensional array N_{count} (line 2) of size $(d-w+1)\times 2$. This data structure stores for the i-th subsequence $T_{i,i+w-1}$ the number of 0 and 1 labels within its k-NN. The procedure transposes the k-NN indices N to its reverse NN R, which retrieves all subsequences that have a given offset as their k-NN (line 3). This information is necessary for retrieving the relevant offsets during the relabelling

Algorithm 3 Cross Validation Scores

```
1:  procedure CROSS_VAL_SCORES(N, w)
2:      N_count, y_true, y_pred ← INIT_LABELS(N)
3:      R ← TRANSPOSE(N)                                  ▷ Reverse NN
4:      M ← INIT_CONF_MATRIX(y_true, y_pred)
5:      ClaSP ← initialize array of length d
6:      for i ∈ [w + 1, ..., d − w − 1] do
7:          y_true[i − w] ← 0                             ▷ The updated label
8:          for idx ∈ R[i − w] do                         ▷ Affected NN
9:              zeros, ones ← UPDATE_COUNTS(N_count[idx])
10:             y_pred[idx] ← 0 if zeros ≥ ones else 1
11:             update M with y_pred[idx]
12:         end for
13:         ClaSP[i] ← SCORE_FUNCTION(M)
14:     end for
15:     return ClaSP
16: end procedure
```

process. The algorithm also initializes a confusion matrix to store the true positive (TP), false positive (FP), false negative (FN) and true negative (TN) counts for both labels (line 4). For the 0 class, all measures are initialized to 0 except the TN count, which is $d-w+1$. Conversely for the 1 class, the TP count is $d-w+1$ with all other counts being 0. The confusion matrix is updated between cross-validations and used to calculate the classification scores for each split in constant time.

The relabelling procedure changes the ground truth label in y_{true} from 1 to 0 for a given split index $i \in [w+1, \ldots, d-w-1]$ (line 7), updating the relevant k-NN labels and the confusion matrix in the process (line 8-12). The classification score resulting from this is stored in the ClaSP (line 13). To achieve this efficiently, the procedure retrieves the subsequence offsets with R which have the split index i as one of their k-nearest neighbours (line 8). The count of zero (one) labels is increased (decreased) by one to reflect that the ground truth label changed from 1 to 0 at split point i (line 9). The algorithm then computes their predicted k-NN majority label, updating the predicted labels y_{pred} and confusion matrix M accordingly (line 10–11). This is done by subtracting (adding) the old (new) prediction from (to) M, and replacing it in y_{pred}. The classification score for split point i is computed and stored in ClaSP (line 13). Evaluation scores such as accuracy and F1 can be calculated with M in constant runtime. Lastly, the cross-validation scores are returned, which constitute ClaSP for the current sliding window in ClaSS (line 15). The central optimization in this routine exploits the fact that although a subsequence can be a k-NN to many other subsequences, the total number of nearest neighbours is bound by exactly $k \cdot (d-w+1)$, the size of all lists from R.

Figure 4 (c) exemplifies how the label configuration changes from split point i to $i+1$. At this offset, the ground truth label changes from 1 to 0. For all offsets that have the split point as a NN, the procedure updates the label counts, confusion matrix and computes new predictions. Lastly, it calculates the classification score for the split point that is inserted in ClaSP (d) at offset $i+1$.

3.3 Detecting Significant Changes

In principle, every local maximum in the cross-validation scores is a potential CP because it marks a sliding window split that separates two differently-shaped segments. This observation is useful for domain experts, who can use a visualization tool, such as [11], to assess these points and to spot semantic changes in the incoming stream. However, automatic change point detection (CPD) is essential for stream segmentation to be incorporated as an IoT edge analytics tool [33], or to uncover latent segmentations in signals where no expert with domain knowledge is available.

To implement this in ClaSS (Algorithm 1, line 9), we first locate the global maximum in the classification scores. We then use the non-parametric two-sided Wilcoxon rank-sum test, as suggested in [17], to check whether, for the associated sliding window split i (from Algorithm 3, line 6), the difference in predicted label frequencies y_{pred} after cross-validation between the left $y_{pred}[1 \ldots i-w]$ and right $y_{pred}[i-w+1 \ldots d-w+1]$ segment is likely due to chance or not. In the ablation study (Subsection 4.2), we empirically learn a significance level for this test. However, in the streaming setting we run into the problem that the test statistics are calculated with different sample sizes due to the sliding window procedure, which takes d as a hyper-parameter and only scores the most recent observations beginning at the last CP cp_l (Algorithm 1, line 8). Accordingly, the number of the predicted cross-validation labels in ClaSP, with which the significance test is computed, is variable, resulting in a bias, because the p-value tends to decrease with increasing observations [55]. To control the variable sample size, resampling is used. $1k$ labels are randomly chosen with replacement from the cross-validation labels y_{pred}, maintaining the class distribution, in order to make the significance level independent of the sliding window size and increase accuracy.

3.4 Learning the Subsequence Width

Setting appropriate parameters is a crucial task for unsupervised data mining algorithms in general and for STSS in particular [59]. Therefore, we propose methods to relieve users from this task and study the impact. A model-parameter in ClaSS is the subsequence width w (Algorithm 1, line 3), needed to partition the TS stream into overlapping subsequences that can be classified. By default, we learn a proper value for w on the first d observations, under the assumption that these initial observations are representative of the characteristics of the entire stream. Multiple window size selection (WSS) methods have been developed based on the idea that a temporal pattern approximately repeats throughout a TS [16], a presumption shared by ClaSS. We use the SuSS [17] algorithm for WSS, due to its expected linear (and worst-case log-linear) runtime complexity with respect to TS length. After the subsequence width has been determined at the start of ClaSS, the sliding window segmentation algorithm processes the stream from the first observation onward.

In settings where users expect or encounter concept drifts in the TS stream, the subsequence width w can be periodically relearned. Similar to the algorithm's initial phase, the data points from a newly evolving segment can be utilized to relearn w, and the segmentation process resumes. This behaviour can be activated on demand. Although we do not need to account for concept drifts in

Table 1: Technical specifications of TS used for experiments.

Name	No. TS	TS Length Min/Median/Max	No. Segments Min/Median/Max
TSSB [17]	75	240 / 3.5k / 20.7k	1 / 3 / 9
UTSA [22]	32	2k / 12k / 40k	2 / 2 / 3
mHealth [4]	90	32.2k / 34.3k / 35.5k	12 / 12 / 12
Arr DB [39]	96	650k / 650k / 650k	1 / 10 / 207
VE DB [25]	44	525k / 525k / 525k	2 / 13 / 134
PAMAP [48]	135	37.5k / 132.1k / 175k	2 / 9 / 9
Sleep DB [30]	88	2.7M / 3.1M / 3.9M	83 / 138 / 231
WESAD [54]	32	2M / 2.1M / 2.1M	5 / 5 / 5

our experimental evaluation, it provides flexibility for applications where they are a concern.

3.5 Setting the Sliding Window Size

Like most streaming algorithms [19], ClaSS requires a sliding window size hyper-parameter d (Algorithm 1, line 1). With larger values for d, ClaSS becomes more accurate, albeit slower, as the amount of available information increases. In many real-world data streams, this tradeoff exhibits a diminishing returns effect, where the accuracy of ClaSS initially improves as d increases, but then tapers off for even larger values. This stagnation is expected, as the amount of information in a signal typically does not grow linearly with its size due to the presence of repetitive substructures [16]. Therefore, d should be set to a value that covers multiple instances (10 to 100 times) of temporal patterns. If such knowledge is not available, ClaSS uses a default value of $10k$, which we found to be robust throughout many domains and sensor types in the experiments (see Subsection 4.4), leading to fast and accurate stream segmentations.

3.6 Computational Complexity

In a streaming setting, the runtime and space complexity of a segmentation procedure is of critical importance for its applicability, as it must keep up with real-time requirements. The complexity of ClaSS is mainly determined by the one-time subsequence width selection (Algorithm 1, line 3) and the recurring scoring and extraction of the sliding window (lines 7–9). SuSS requires $O(d \log w)$ to learn the subsequence width from the initial d observations.

The total runtime of the k-NN update is dominated by the dot product calculation in $O(w)$ (Algorithm 2, lines 6–8) and k sequential NN searches in $O(k \cdot d)$ (line 11). In the streaming setting, where the routine is called $n \gg d$ times, its time complexity is $O(d)$, since k is a small constant. The runtime of the shift operation (line 20) is dominated by moving the data, which is in $O(k \cdot d) = O(d)$ as both C and N have the dimensionality $(d - w + 1) \times k$. Updating C and N (lines 22–23) relies on replacing and moving values, which is also performed in $O(k \cdot d)$ and hence in $O(d)$.

The complexity of the sliding window scoring depends on relabelling and evaluating k-NN offsets. For a single cross-validation, the amortized runtime is in $O(1)$ (Algorithm 3, lines 7–13), as only counts are updated and the score is calculated for a constrained number of offsets in the reverse NN. For all $(d - 2 \cdot w - 1)$ splits, this leads to $O(d)$ total runtime for the entire algorithm.

Table 2: Specification of competitors; n is the number of all observed values, $c \ll d$ is an adaptive/custom window size.

Competitor	Update Complexity	Segmentation Method
BOCD [2]	$O(n)$	Bayesian probability
FLOSS [22]	$O(d \log d)$	Matrix profile
ClaSS	$O(d)$	Self-supervision
ChangeFinder [65]	$O(c^2)$	Moving averages
Window [57]	$O(c)$	Autoregressive cost
NEWMA [31]	$O(c)$	Moving averages
ADWIN [8]	$O(\log c)$	Adaptive Statistics
DDM [18]	$O(1)$	Model error
HDDM [9]	$O(1)$	Hoeffding's inequality

The Wilcoxon rank-sum test in ClaSS, which is used to detect CPs (Algorithm 1, line 9), can also be implemented in $O(d)$, as it mainly depends on ranking binary classes. This results in an overall amortized runtime complexity of $O(d)$ for processing a single observation and $O(n \cdot d)$ for segmenting n measurements with ClaSS. The space complexity is likewise linearly dependent on the sliding window size.

4 EXPERIMENTAL EVALUATION

To evaluate the characteristics of ClaSS and to compare it to 8 state-of-the-art competitors, we measured accuracy, runtime and scalability on large benchmark data sets as well as real-world annotated data archives from experimental studies. Subsection 4.1 outlines data sets, evaluation metrics and methods. We investigate the influence of different design choices in Subsection 4.2 through an ablation study. Subsections 4.3 and 4.4 further evaluate ClaSS and 8 competitors in terms of accuracy, runtime, and throughput. Lastly, Subsection 4.5 discusses a real-life use case to showcase the features and limitations of ClaSS. Experiments were conducted on an Intel Xeon 8358 with 2.60 GHz, 2 TB RAM, 128 cores, running Python 3.8. To ensure reproducibility and foster follow-up works, all source codes, Jupyter-Notebooks, TS used in the evaluation, visualizations, raw measurement sheets and a technical report are available on our supporting website [11].

4.1 Experiment Setup

Data Sets: We use 592 time series from two public TSS benchmarks and six data archives from experimental studies (see Table 1) to measure the quality of ClaSS and 8 competitors. In the following evaluations, we simulated the streaming setting by processing one data point at a time. The ground truth CP locations, used to evaluate the algorithms, were annotated by domain experts. The benchmark data sets from UTSA [22] and TSSB [17] consist of 107 preprocessed medium to large (240 to 40k) TS, representing a dense collection of diverse problem settings featuring biological, mechanical or synthetic processes from sensor, device, image, spectrogram and simulation signals. The data archives contain 485 very large (32.2k to 3.9M) raw sensor signals from 10 sensors capturing human subjects in experimental studies, such as human activity or emotion recognition, medical condition monitoring, or sleep analysis. These data sets are of particular interest, as they reflect

a common application of STSS, in which researchers must first segment instances of very large, heterogeneous recordings into homogeneous subsequences and then apply advanced data mining algorithms such as anomaly detection, forecasting, or classification. Whilst the benchmarks encompass a variety of domains, the data archives focus on human-centric processes.

Evaluation Metric: The literature contains multiple classification- and clustering-based metrics to assess the quality of segmentations; see [57] for a survey. Specifically, we use the soft evaluation measure Covering [58]. This measure quantifies the exact degree to which predicted vs annotated segments overlap and allows the comparison of different-sized segmentations.

It is defined as follows: Let the interval of successive CPs $[t_{c_i}, \ldots, t_{c_{i+1}}]$ denote a segment in T and let $segs_{pred}$ as well as $segs_T$ be the sets of predicted or ground truth segmentations, respectively. For notational convenience, we always consider $t_{c_1} = 0$ as the first and $t_{c_{|segs_T|}} = n+1$ as the last CP to include the first (last) segment. The Covering score reports the best-scoring weighted overlap between a ground truth and a predicted segmentation (using the Jaccard index) as a normed value in the interval $[0, \ldots, 1]$ with higher being better (Equation 6).

$$\text{Covering} = \frac{1}{\|T\|} \sum_{s \in segs_T} \|s\| \cdot \max_{s' \in segs_{pred}} \frac{\|s \cap s'\|}{\|s \cup s'\|} \tag{6}$$

To aggregate results from multiple data sets into a single ranking, we compute the rank of the score for each method on each TS. We then average the rank of each method across all data sets to obtain its overall rank. Critical Difference (CD) diagrams [13], such as Figure 5 (top), are used to statistically assess differences in the mean ranks. The best approaches, which score the lowest average ranks, are shown to the right of the diagram. Approaches that are not significantly different in their ranks are connected by a bar, based on a Nemenyi two-tailed significance test with $\alpha = 0.05$.

Competitors: We compare ClaSS with 8 state-of-the-art competitors (see Table 2), learning optimal hyper-parameters for all algorithms by testing multiple design choices on 20% randomly chosen benchmark TS (21 out of 107), to prevent overfitting. See the technical report [11] for details. For a fair comparison, we learned the design choices of ClaSS on the same TS, described in Subsection 4.2. Some of the competitors (Window, BOCD, ChangeFinder and FLOSS) do not specify online segmentation procedures, but only present homogeneity scores for sliding window splits. We learned a threshold for these scores and report splits with a certain quality, considering an exclusion zone (as proposed in [23]) to prevent series of closely located splits. Note that BOCD processes all so far observed data points of length n, ClaSS and FLOSS require a sliding window of size d, whereas all other methods either use adaptive / custom-sized sliding windows of length $c \ll d$, or update constant-sized statistics (see Table 2).

4.2 Ablation Study

ClaSS has seven major design choices that determine its performance: (a) sliding window size, (b) subsequence width selection method, (c) similarity measure, (d) number of k neighbours used in the streaming k-NN, (e) classification score to evaluate the cross-validations, (f) significance level and (g) sample size used for the detection of significant CPs. We tested ClaSS on the same randomly chosen 20% of benchmark TS with varying values (or methods) of each parameter while fixing the others to their default values. We summarize the results of these extensive experiments and refer the interested reader to our supporting website [11] for the raw measurements and visualizations.

(a) Sliding Window Size: We computed ClaSS with sliding window sizes ranging between $1k$ to $20k$ (steps of $1k$) data points. This group of design choices does not show statistically significant differences in ranks, ranging between 76.7% and 81.4% average Covering performances, 13.2% and 17.4% standard deviations and 9 to 14 wins. We choose $d = 10k$ as a robust default parameter for many scenarios. The user may adapt this parameter, however, to control the throughput of ClaSS (see Subsection 3.5 and 4.4).

(b) Window Size Selection: We tested two whole-series based methods from [16], the most dominant Fourier frequency (FFT) and the highest autocorrelation offset (ACF), as well as two subsequence-based algorithms, Multi-Window-Finder (MWF) [28] and Summary Statistics Subsequence (SuSS) [17]. Our results show no significant differences between the ranks of the methods. We choose SuSS for WSS in ClaSS, as it achieves the most wins and best mean (standard deviation) Covering performance of 79.1% (15.6%).

(c, d) k-Nearest Neighbours: We evaluated Pearson correlation, Euclidean distance and CID as (dis-)similarity measures and $k \in [1, 3, 5, 7]$. We found no significant differences between rankings; Pearson correlation and a 3-NN score the best ranks and show the best Covering performance. Therefore, we use both as the default. Users may change the similarity measure to fit specific use cases.

(e) Classification Score: For the sliding window scoring, we assessed the F1 score and accuracy. We used the macro formulation for both scores, which computes them per label and then averages the results, to tackle the inherent class imbalance in the ClaSP calculated by ClaSS. We did not test the ROC/AUC score, used in the batch ClaSP, as it is not computable in constant runtime from the confusion matrix, which is a prerequisite for us to keep the linear runtime complexity in ClaSS. F1 ranks are not significantly better than accuracy, but show best results. Thus, we use it as the default classification score.

(f, g) Significance Level: Lastly, we evaluated significance levels in the range 1e-10 to 1e-100 (with steps of 1e-10) and sample sizes (variable, 10, 100, 1k, 10k) for extracting significant CPs in the sliding window stream segmentation. The variable sample size uses the entire label configuration, as proposed in [17]. We found that the range of thresholds between 1e-50 and 1e-100 as well as the variable and 1k sample sizes substantially outperformed the other options. The significance level of 1e-50 with 1k sample size achieved the highest mean Covering score and the lowest standard deviation. Thus, we use this configuration in ClaSS.

We conclude that the choice of the sliding window size, the subsequence width, the k-NN and the classification score in ClaSS have only negligible effects on its performance. For specific domains, users may adjust the significance level to optimize results.

4.3 Quantitative Analysis

We evaluate the performance of ClaSS and its 8 competitors separately for the two benchmark data sets and six data archives from

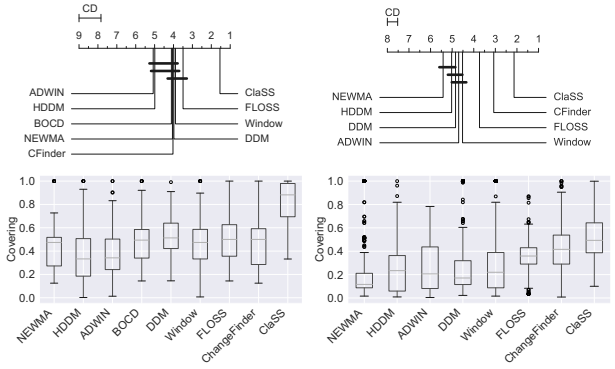

Figure 5: Covering segmentation ranks (top) and box plots (bottom) on the 107 benchmark (left) and 485 archive (right) TS for ClaSS (lowest rank) and the 8 competitors.

Table 3: Summary Covering performances for ClaSS (best results) and its 8 competitors on the two benchmarks and six data archives.

	Benchmarks / Data Archives		
	mean (in %)	median (in %)	std (in %)
ClaSS	**81.2** / **51.5**	**88.2** / **49.3**	19.0 / 17.1
ChangeFinder	47.3 / 42.3	50.0 / 41.6	23.5 / 19.7
FLOSS	52.1 / 35.6	50.0 / 35.9	22.7 / 13.0
Window	46.1 / 29.1	47.4 / 22.0	24.7 / 27.7
DDM	53.5 / 26.2	51.3 / 17.1	16.9 / 24.5
BOCD	48.1 / -	49.4 / -	19.0 / -
ADWIN	38.3 / 26.2	34.2 / 20.6	20.6 / 20.5
HDDM	36.5 / 24.6	33.3 / 23.4	24.8 / 18.5
NEWMA	43.4 / 21.5	47.4 / 11.6	20.6 / 26.2

experimental studies. We remark that the data archives are, by far, the harder scenario as they contain ambiguities, anomalies and signal noise and are up to two orders of magnitude larger than the benchmarks; note that algorithms are not fine-tuned to these conditions. Detailed measurements and visualizations are reported on our supporting website [11].

Benchmark Data Sets: The CD diagram in Figure 5 (top left) illustrates the mean Covering ranks. Best results are obtained by ClaSS (1.5) followed by FLOSS (3.5), Window (3.9), DDM (4.0), ChangeFinder (4.0), NEWMA and BOCD (4.1), HDDM (5.0) and ADWIN (5.1). The performance advance of ClaSS is statistically significant, while the differences between the 2nd to 7th-ranking approaches are not. Considering both benchmarks separately, ClaSS still achieves the best performances, with an insignificant advance for UTSA (32 TS), but a significant advance for TSSB (75 TS).

ClaSS wins or ties in 78 of the 107 cases, followed by FLOSS, Window and ChangeFinder each with each 12 wins/ties, DDM (10), NEWMA (9), BOCD (8), HDDM (5) and ADWIN (4). Counts do not sum up to 107 due to ties. ClaSS achieves first place in four subcases of STSS: TS with one segment (6 instances), two segments (46 instances), at least three segments (55 instances), and reoccurring sub-segments (10 instances). In a pairwise comparison of ClaSS against every competitor, ClaSS outperforms all competitors in at least 77% of all cases (see [11]).

ClaSS achieves a mean Covering performance of 81.2%, with a standard deviation of 19.0%. Figure 5 (bottom left) and Table 3 show this to be the highest score, with a large margin of 27.7 pp over the second-best method. The differences in median results are even more pronounced. The summary statistics of the performances are quite stable across the UTSA and TSSB data sets (data shown on [11]). This shows that ClaSS is able to segment TS streams more accurately than its counterparts on the benchmark data sets.

Data Archive Sets: For the 485 time series (TS) from the six data archives, ClaSS (2.1) ranks first, followed by ChangeFinder (3.1), FLOSS (3.7), Window (4.5), ADWIN (4.7), DDM (4.8), HDDM (5.0) and NEWMA (5.4) (see Figure 5 top right). BOCD did not finish within days, and was excluded. Again, ClaSS significantly outperforms its competitors, with the 2nd and 3rd-best competitor ChangeFinder and FLOSS also significantly outperforming the rest. ClaSS ranks first in 5 out of 6 data archives, with 1 significant lead on mHealth and 4 insignificant advances for WESAD, SleepDB, MIT-BIH-VE DB and MIT-BIH-VE-Arr DB. ChangeFinder ranks first on PAMAP, but only with an insignificant difference to ClaSS. We aggregated the average Covering ranks by sensor type and found that ClaSS outperforms its rivals for 7 out of 10 sensors (1 significant); the 3 it performs worse for are electrodermal activity, respiration and body temperature, which are all contained in the WESAD archive and represent just 4 TS per sensor. More annotated TS from these sensors are needed to give a conclusive result on their specific segmentation performance. In a pairwise comparison of ClaSS against the 7 competitors on the data archives, ClaSS achieves the best segmentations in at least 69% instances.

Considering the summary statistics in Figure 5 (bottom right) and Table 3, all methods drop in mean and median Covering performance but keep similar standard deviations on the data archives compared to the benchmark results. ClaSS scores the highest mean Covering performance of 51.5% and the second-smallest standard deviation of 17.1%. The performance improvement of 9.2 pp compared to the second-best method is substantial, however 18.5 pp less than for the benchmark results.

Discussion: Our performance analysis shows that ClaSS outperforms 8 other methods in 305 of 592 TS data sets. This superior performance is attributed to two key characteristics:

(a) ClaSS uses a self-supervised, non-linear k-NN classifier for segmenting TS streams, evaluating the likelihood of potential sliding window prefixes being a completed segment. This method is adept at understanding the diverse semantics of signals, unlike the auto-regressive or statistical deviation models used by our competitors, with the exception of FLOSS.

(b) For identifying CPs, ClaSS utilizes a non-parametric significance test rather than a fixed threshold. This approach allows for more flexible adaptation to different data sets, a strategy not adopted by our competitors except for HDDM.

A real-world example of the impact of these two design choices of ClaSS is explored in Subsection 4.5.

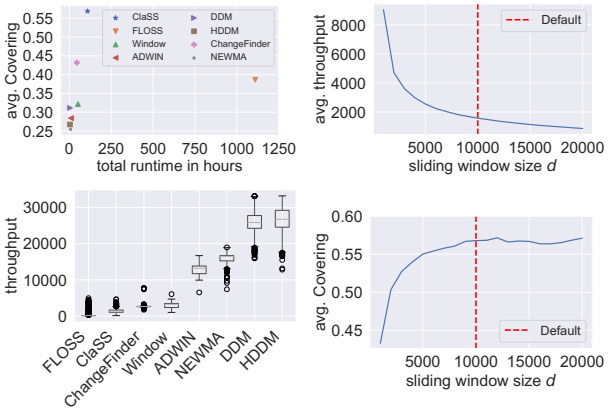

Figure 6: Runtime comparison regarding total time spent vs quality (top left) and standalone data throughput (bottom left) of competitors. Change in throughput (top right) and Covering (bottom right) for different sliding window sizes.

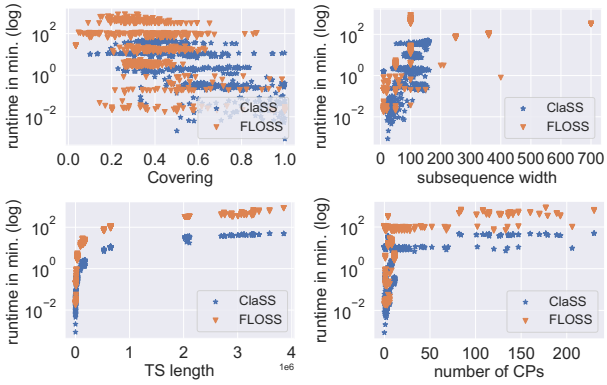

Figure 7: Scalability of ClaSS vs FLOSS considering Covering performance (top left), subsequence width (top right), TS length (bottom left), and number of CPs (bottom right).

4.4 Runtime and Throughput

STSS methods need to process sensor streams in real-time to be useful in practice. We conducted experiments to measure the relationship between runtime and quality as well as data throughput of ClaSS and its competitors on all 592 TS data sets.

Runtime: As shown in Figure 6 (top left), HDDM is the fastest method (total of 4 hours), followed by DDM (5 hours), NEWMA (7 hours), ADWIN (10 hours), ChangeFinder (45 hours), Window (52 hours), ClaSS (109 hours) and FLOSS (1109 hours), for a total of 3.5 GB of 64-bit floating-point TS data on a single core. This ranking is roughly aligned with the computational complexities and sliding window sizes of the methods (see Table 2). The 4 fastest methods build a cluster (bottom left) and produce low average Covering results from 25.4% to 31.1%. ChangeFinder and ClaSS trade runtime to score substantially higher average Covering performances of 43.2% and 56.9%, while being one order of magnitude slower. However, ClaSS is more than 10 times faster and 18.3 pp more accurate than FLOSS, although both methods process the same sliding window. ClaSS needs 36/109 hours for the bespoke k-NN updates. Recomputing dot products increases this runtime to 212 hours; naive distance calculations take 2513 hours. Additionally, ClaSS spends 55/109 hours for the bespoke cross-validations. Using the original implementation from [17] was stopped after 5755 hours, segmenting 7 out of 8 data sets. This empirically validates the massive runtime improvements of the central components.

Throughput: Figure 6 (bottom left) provides a visual representation of the methods' data throughputs when operated in isolation. On average, HDDM and DDM process 26,458 and 26,031 observations per second, followed by NEWMA and ADWIN with 15,949 and 12,958 measurements as well as Window, ChangeFinder, ClaSS and FLOSS with 2,991 down to 378 data points. ClaSS, with an average of 1,408 measurements per second, reaches a maximum of 4,660 observations at times, as its segmentation procedure only scores the unsegmented data points, which leads to throughput peaks. Experiments with ClaSS in Apache Flink show comparable results, averaging $1,004 \pm 310$ values per second (see [11]). This demonstrates that ClaSS can segment data streams with hundreds of points per second with default parameters using a single core. This is sufficient to segment many IoT or medical sensors that output values in this range.

Sliding Window Size: The line plots in Figure 6 (right) illustrate the change in average throughput (top) and Covering (bottom) for sliding window sizes from 1k to 20k (steps of 1k) in ClaSS. The default value of 10k (red line) roughly marks the beginning of converging Covering performance between 56% and 57%. Cutting it in half to 5k, increases throughput to 2548 data points per second (1.8x), loosing 1.8pp accuracy. Similarly, doubling the default sliding window size to 20k decreases throughput to 863 data points per second (0.6x), gaining 0.3pp of accuracy. We find that sliding window sizes between 5k and 10k retain most of the accuracy while providing scope for throughput optimizations.

Scalability: Figure 7 shows the scalability of ClaSS vs FLOSS per TS in relation to Covering score, subsequence width, TS length and amount of CPs. We omit a comparison against batch ClaSP, as its quadratic runtime prohibits an application in our experiments (TS with up to 3.9M values). FLOSS and ClaSS share a similar dispersion of runtimes for the variables, shifted by their total difference. For Covering and subsequence width, we do not observe clear relationships. Conversely, for increasing TS length or number of CPs, both algorithms need more runtime. As expected, ClaSS is consistently faster than FLOSS for large TS. On [11], we additionally show that ClaSS its runtime scales linearly for increasing TS length, which empirically validates its time complexity stated in Subsection 3.6. The runtime of ClaSS for segmenting very large offline data archives can probably be accurately predicted using regression.

4.5 Human Activity Recognition Use Case

We explore the segmentation results of an interesting use case from the PAMAP [48] data archive to show the characteristics of ClaSS compared to FLOSS and Window (2nd and 3rd best benchmark competitors). Human activity recognition (HAR) is an important subfield of ubiquitous sensing, with applications in medical condition monitoring and decision-support in tactical scenarios [35].

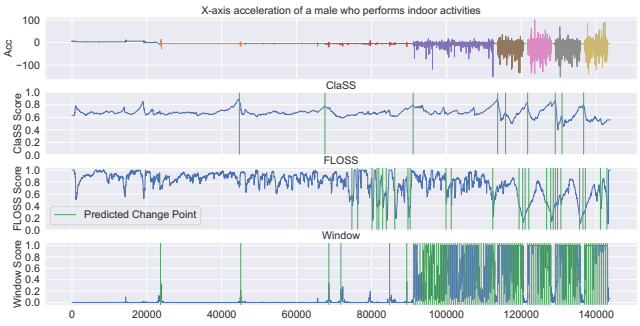

Figure 8: The TS (top) captures the X-axis acceleration of human activity movement [49]. The aggregated profiles for ClaSS, FLOSS and Window (2nd from top to bottom) are illustrated with predicted CPs (green). See [11] for a video.

Figure 8 illustrates an example of HAR, showing accelerometer readings from a 26-year-old male performing a sequence of 9 activities (top). We computed the profiles and predicted CPs using ClaSS, FLOSS and Window (2nd from top to bottom) and visualized the max-aggregated scores for ClaSS and Window and the min-aggregated values for FLOSS. A video of ClaSS' real-time segmentation is available online [11]. ClaSS has a smooth score profile with accurate predictions, missing only a very subtle change from lying to sitting and having two false positives. FLOSS generated a noisy arc curve with more false positives, related to its greedy CP extraction algorithm. Window accurately detected the first four activity transitions, but then had many false positives due to the discrepancy measure misjudging the signal. This use case highlights the accuracy and adaptability of ClaSS, as well as its interpretability for human inspection.

5 RELATED WORK

In the last two decades, a wealth of benchmarks [45], databases [46], indices [15], compression algorithms [37], and data analytics [44] have been developed by TS management and mining research. This research is driven by the rapidly growing amount of sensor data from IoT devices in *smart* applications for environments, healthcare or factories [33]. Sensors, found in wearables or fixed installations, include e.g. accelerometers, thermometers, or optical sensors. Their data, sampled at varying rates, is wirelessly transmitted via Wi-Fi, Bluetooth, or NFC to edge analytics for initial pre-processing and fusion, before being sent to the cloud for advanced analysis [50].

STSS is a complex preprocessing step in many IoT workflows and has been extensively researched in different settings, e.g. on edge devices [12], for smart homes [61], and as a part in integrated HAR systems [10, 34, 56]. Such workflows are typically implemented with streaming platforms such as Apache Flink, Spark, or Storm to manage and process vast amounts of TS data in real-time. Stream processing systems mainly differ regarding their processing models, such as one-at-a-time and micro-batch, issued value delivery guarantees (at-least vs. exactly-once) and order [63]. Karimov et al. [29] conducted a benchmark comparing these systems in terms of data skew, data arrival fluctuations, latency, and throughput. They found that no single platform consistently outperformed the others, with each possessing unique advantages and disadvantages. For instance, Flink exhibits the lowest average latency but is less effective at handling skewed data compared to Spark. Gehring et al. [21] explored qualitative criteria, such as functionality, simplicity, and documentation, when developing TS analytics using Flink and Spark. Their findings indicate that Flink's development API, evaluation, and visualization functionality are better suited for TS analysis workflows. Consequently, we implemented ClaSS in Flink for integration with its stream processing system.

Besides its practical application, STSS has been studied conceptually as CP and drift detection problems [20, 32]. These formalizations focus on the point at which one segment changes into another. Algorithms monitor the shape or value distributions of sliding windows from a TS stream and report CPs once they substantially differ. The literature differentiates multiple categories of methods [3, 57]. Parametric techniques measure the change in a signal's assumed probability distribution. Implementations include Window [57], DDM [18], ADWIN [8] or BOCD [2], which estimates the posterior distribution for the observations since the last CP and can be extended to accommodate for short, gradual changes [14]. Non-parametric approaches do not assume a specific model, and instead compute kernels, distances or rankings on the stream to quantify drift. Algorithms include FLOSS [23], which estimates the density of homogenous regions using nearest-neighbour arcs, or our proposed method ClaSS, which utilizes self-supervised learning [26], and makes few assumptions about segments, e.g. being mutually dissimilar. This is in contrast to the aforementioned existing methods, which are either domain-specific, parametric or lack a robust segmentation procedure to handle observed noise.

6 CONCLUSION

We proposed ClaSS, a novel algorithm with minimal assumptions for streaming time series segmentation (STSS) that is amenable to human inspection. Our extensive experiments demonstrate that it sets the new state of the art on two benchmarks with 107 TS, five out of six data archives from experimental studies with 485 TS, and is fast and scalable. In addition to the streaming setting, ClaSS can also be used for very long TS in the batch scenario where computationally expensive TSS algorithms become infeasible.

Besides its strengths, weaknesses include the initial time points needed to determine the subsequence width and the dependence on the predictive power of the k-NN classifier. ClaSS is solely applicable to univariate TS. Many real-world use cases, however, capture processes with a multitude of sensors, where temporal patterns are distributed across various channels. It also has less throughput compared to some competitors, which restricts its applicability for sensors with extremely high sampling rates.

In future work, we plan to extend ClaSS to the multivariate setting, exploring sensor fusion and dimension selection to improve accuracy. We also plan to accelerate the streaming k-NN calculation and significance test by simultaneously working on sliding window partitions using multi-threading or GPUs.

REFERENCES

[1] Colin Adams, Luis Alonso, Benjamin Atkin, John P. Banning, Sumeer Bhola, Richard W. Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, Nick Sakharov,

George Talbot, Nick Taylor, and Adam Tart. 2020. Monarch: Google's Planet-Scale In-Memory Time Series Database. *Proc. VLDB Endow.* 13 (2020), 3181–3194.
[2] Ryan Prescott Adams and David JC MacKay. 2007. Bayesian online changepoint detection. *arXiv preprint arXiv:0710.3742* (2007).
[3] Samaneh Aminikhanghahi and Diane Joyce Cook. 2016. A survey of methods for time series change point detection. *Knowledge and Information Systems* 51 (2016), 339–367.
[4] Oresti Baños, Rafael García, Juan Antonio Holgado Terriza, Miguel Damas, Héctor Pomares, Ignacio Rojas, Alejandro Saez, and Claudia Villalonga. 2014. mHealthDroid: A Novel Framework for Agile Development of Mobile Health Applications. In *IWAAL*.
[5] Oresti Baños, Claudia Villalonga, Rafael García, Alejandro Saez, Miguel Damas, Juan Antonio Holgado-Terriza, Sungyong Lee, Héctor Pomares, and Ignacio Rojas. 2015. Design, implementation and validation of a novel open framework for agile development of mobile health applications. *BioMedical Engineering OnLine* 14 (2015), S6 – S6.
[6] Gustavo E. A. P. A. Batista, Eamonn J. Keogh, Oben M. Tataw, and Vinicius M. A. Souza. 2013. CID: an efficient complexity-invariant distance for time series. *Data Mining and Knowledge Discovery* 28 (2013), 634–669.
[7] M. Beyreuther, Robert Barsch, Lion Krischer, Tobias Megies, Yannik Behr, and Joachim Wassermann. 2010. ObsPy: A Python Toolbox for Seismology. *Seismological Research Letters* 81 (2010), 530–533.
[8] Albert Bifet and Ricard Gavaldà. 2007. Learning from Time-Changing Data with Adaptive Windowing. In *SDM*.
[9] Isvani Inocencio Frías Blanco, José del Campo-Ávila, Gonzalo Ramos-Jiménez, Rafael Morales Bueno, Agustín Alejandro Ortiz Díaz, and Yailé Caballero Mota. 2015. Online and Non-Parametric Drift Detection Methods Based on Hoeffding's Bounds. *IEEE Transactions on Knowledge and Data Engineering* 27 (2015), 810–823.
[10] Hyunjeong Cho, Jihoon An, Intaek Hong, and Younghee Lee. 2015. Automatic Sensor Data Stream Segmentation for Real-time Activity Prediction in Smart Spaces. *Proceedings of the 2015 Workshop on IoT challenges in Mobile and Industrial Systems* (2015).
[11] ClaSS Code and Raw Results. 2023. https://github.com/ermshaua/classification-score-stream.
[12] Roman Dębski and Rafał Dreżewski. 2021. Adaptive Segmentation of Streaming Sensor Data on Edge Devices. *Sensors (Basel, Switzerland)* 21 (2021).
[13] Janez Demšar. 2006. Statistical Comparisons of Classifiers over Multiple Data Sets. *The Journal of Machine Learning Research* 7 (2006), 1–30.
[14] Erick Draayer, Huiping Cao, and Yifan Hao. 2021. Reevaluating the Change Point Detection Problem with Segment-based Bayesian Online Detection. *Proceedings of the 30th ACM International Conference on Information & Knowledge Management* (2021).
[15] Karima Echihabi, Panagiota Fatourou, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2022. Hercules Against Data Series Similarity Search. *Proc. VLDB Endow.* 15 (2022), 2005–2018.
[16] Arik Ermshaus, Patrick Schäfer, and Ulf Leser. 2022. Window Size Selection In Unsupervised Time Series Analytics: A Review and Benchmark. *7th Workshop on Advanced Analytics and Learning on Temporal Data* (2022).
[17] Arik Ermshaus, Patrick Schäfer, and Ulf Leser. 2023. ClaSP: parameter-free time series segmentation. *Data Mining and Knowledge Discovery* 37 (2023), 1262 – 1300.
[18] João Gama, Pedro Medas, Gladys Castillo, and Pedro Pereira Rodrigues. 2004. Learning with Drift Detection. In *Brazilian Symposium on Artificial Intelligence*.
[19] João Gama and Pedro Pereira Rodrigues. 2007. Data stream processing. In *Learning from Data Streams*. Springer, 25–39.
[20] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and A. Bouchachia. 2014. A survey on concept drift adaptation. *ACM Computing Surveys (CSUR)* 46 (2014), 1 – 37.
[21] Melissa Gehring, Marcela Charfuelan, and Volker Markl. 2019. A Comparison of Distributed Stream Processing Systems for Time Series Analysis. In *Datenbanksysteme für Business, Technologie und Web*.
[22] Shaghayegh Gharghabi, Yifei Ding, Chin-Chia Michael Yeh, Kaveh Kamgar, Liudmila Ulanova, and Eamonn Keogh. 2017. Matrix profile VIII: domain agnostic online semantic segmentation at superhuman performance levels. In *ICDM*. IEEE, 117–126.
[23] Shaghayegh Gharghabi, Chin-Chia Michael Yeh, Yifei Ding, Wei Ding, Paul R. Hibbing, Samuel R LaMunion, Andrew Kaplan, Scott E. Crouter, and Eamonn J. Keogh. 2018. Domain agnostic online semantic segmentation for multi-dimensional time series. *Data Mining and Knowledge Discovery* 33 (2018), 96 – 130.
[24] Ary L. Goldberger, Luis A. Nunes Amaral, L Glass, Jeffrey M. Hausdorff, Plamen Ch. Ivanov, Roger G. Mark, Joseph E. Mietus, George B. Moody, Chung-Kang Peng, and Harry Eugene Stanley. 2000. PhysioBank, PhysioToolkit, and PhysioNet: components of a new research resource for complex physiologic signals. *Circulation* 101 23 (2000), E215–20.
[25] Scott D Greenwald. 1986. The development and analysis of a ventricular fibrillation detector.
[26] Shohei Hido, Tsuyoshi Idé, Hisashi Kashima, Harunobu Kubo, and Hirofumi Matsuzawa. 2008. Unsupervised Change Analysis Using Supervised Learning. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*.
[27] Inseok Hwang, Sungwan Kim, Youdan Kim, and Chze Eng Seah. 2010. A Survey of Fault Detection, Isolation, and Reconfiguration Methods. *IEEE Transactions on Control Systems Technology* 18 (2010), 636–653.
[28] Shima Imani and Eamonn Keogh. 2021. Multi-Window-Finder: Domain Agnostic Window Size for Time Series Data. *MileTS'21: 7th KDD Workshop on Mining and Learning from Time Series* (2021).
[29] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman S. Samarev, Henri Heiskanen, and Volker Markl. 2019. Benchmarking Distributed Stream Data Processing Systems. *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (2019), 1507–1518.
[30] Bob Kemp, Aeilko H. Zwinderman, Bert Tuk, Hilbert A. C. Kamphuisen, and Josefien J. L. Oberye. 2000. Analysis of a sleep-dependent neuronal feedback loop: the slow-wave microcontinuity of the EEG. *IEEE Transactions on Biomedical Engineering* 47 (2000), 1185–1194.
[31] Nicolas Keriven, Damien Garreau, and Iacopo Poli. 2018. NEWMA: A New Method for Scalable Model-Free Online Change-Point Detection. *IEEE Transactions on Signal Processing* 68 (2018), 3515–3528.
[32] Daniel Kifer, Shai Ben-David, and Johannes Gehrke. 2004. Detecting Change in Data Streams. In *Very Large Data Bases Conference*.
[33] Rajalakshmi Krishnamurthi, Adarsh Kumar, Dhanalekshmi Gopinathan, Anand Nayyar, and Basit Qureshi. 2020. An Overview of IoT Sensor Data Processing, Fusion, and Analysis Techniques. *Sensors (Basel, Switzerland)* 20 (2020).
[34] Javier Ortiz Laguna, Angel Garcia-Olaya, and Daniel Borrajo. 2011. A dynamic sliding window approach for activity recognition. In *User Modeling, Adaptation, and Personalization*.
[35] Oscar D. Lara and Miguel A. Labrador. 2013. A Survey on Human Activity Recognition using Wearable Sensors. *IEEE Communications Surveys & Tutorials* 15 (2013), 1192–1209.
[36] Oleksandra Levchenko, Boyan Kolev, Djamel Edine Yagoubi, Reza Akbarinia, Florent Masseglia, Themis Palpanas, Dennis Shasha, and Patrick Valduriez. 2020. BestNeighbor: efficient evaluation of kNN queries on large time series databases. *Knowledge and Information Systems* 63 (2020), 349 – 378.
[37] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: Efficient Lossless Floating Point Compression for Time Series Databases. *Proc. VLDB Endow.* 15 (2022), 3058–3070.
[38] Yasuko Matsubara, Yasushi Sakurai, and Christos Faloutsos. 2014. AutoPlait: automatic mining of co-evolving time sequences. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (2014).
[39] George B. Moody and Roger G. Mark. 2001. The impact of the MIT-BIH Arrhythmia Database. *IEEE Engineering in Medicine and Biology Magazine* 20 (2001), 45–50.
[40] Abdullah Al Mueen, Hossein Hamooni, and Trilce Estrada. 2014. Time Series Join on Subsequence Correlation. *2014 IEEE International Conference on Data Mining* (2014), 450–459.
[41] Jannes Munchmeyer, Dino Bindi, Ulf Leser, and Frederik Tilmann. 2020. The transformer earthquake alerting model: a new versatile approach to earthquake early warning. *Geophysical Journal International* (2020).
[42] FM Nolle, FK Badura, JM Catlett, RW Bowser, and MH Sketch. 1986. CREI-GARD, a new concept in computerized arrhythmia monitoring systems. *Computers in Cardiology* 13, 1 (1986), 515–518.
[43] Jiapu Pan and Willis J. Tompkins. 1985. A Real-Time QRS Detection Algorithm. *IEEE Transactions on Biomedical Engineering* BME-32 (1985), 230–236.
[44] Shuye Pan, Peng Wang, Chen Wang, Wei Wang, and Jianmin Wang. 2022. NLC: Search Correlated Window Pairs on Long Time Series. *Proc. VLDB Endow.* 15 (2022), 1363–1375.
[45] John Paparrizos, Yuhao Kang, Paul Boniol, Ruey Tsay, Themis Palpanas, and Michael J. Franklin. 2022. TSB-UAD: An End-to-End Benchmark Suite for Univariate Time-Series Anomaly Detection. *Proc. VLDB Endow.* 15 (2022), 1697–1711.
[46] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *Proc. VLDB Endow.* 8 (2015), 1816–1827.
[47] Thanawin Rakthanmanon, Bilson J. L. Campana, Abdullah Al Mueen, Gustavo E. A. P. A. Batista, M. Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn J. Keogh. 2012. Searching and Mining Trillions of Time Series Subsequences under Dynamic Time Warping. *KDD : proceedings. International Conference on Knowledge Discovery & Data Mining* 2012 (2012), 262 – 270.
[48] Attila Reiss and Didier Stricker. 2011. Towards global aerobic activity monitoring. In *PETRA '11*.
[49] Attila Reiss and Didier Stricker. 2012. Creating and benchmarking a new dataset for physical activity monitoring. In *PETRA '12*.
[50] Ivo Santos, Marcel Tilly, Badrish Chandramouli, and Jonathan Goldstein. 2013. DiAl: Distributed Streaming Analytics Anywhere, Anytime. *Proc. VLDB Endow.* 6 (2013), 1386–1389.

[51] Patrick Schäfer, Arik Ermshaus, and Ulf Leser. 2021. ClaSP - Time Series Segmentation. *Proceedings of the 30th ACM International Conference on Information & Knowledge Management* (2021).

[52] Patrick Schäfer and Ulf Leser. 2022. Motiflets - Simple and Accurate Detection of Motifs in Time Series. *Proc. VLDB Endow.* 16 (2022), 725–737.

[53] Zachary Schall-Zimmerman, Nader Shakibay Senobari, Gareth J. Funning, Evangelos E. Papalexakis, Samet Oymak, Philip Brisk, and Eamonn J. Keogh. 2019. Matrix Profile XVIII: Time Series Mining in the Face of Fast Moving Streams using a Learned Approximate Matrix Profile. *2019 IEEE International Conference on Data Mining (ICDM)* (2019), 936–945.

[54] Philip Schmidt, Attila Reiss, Robert Dürichen, Claus Marberger, and Kristof Van Laerhoven. 2018. Introducing WESAD, a Multimodal Dataset for Wearable Stress and Affect Detection. *Proceedings of the 20th ACM International Conference on Multimodal Interaction* (2018).

[55] Matthew S. Thiese, Brenden B Ronna, and Ulrike Ott. 2016. P value interpretations and considerations. *Journal of thoracic disease* 8 9 (2016), E928–E931.

[56] Darpan Triboan, Liming Luke Chen, Feng Chen, and Zumin Wang. 2017. Semantic segmentation of real-time sensor data stream for complex activity recognition. *Personal and Ubiquitous Computing* 21 (2017), 411–425.

[57] Charles Truong, Laurent Oudre, and Nicolas Vayatis. 2020. Selective review of offline change point detection methods. *Signal Processing* 167 (2020), 107299.

[58] Gerrit JJ van den Burg and Christopher KI Williams. 2020. An evaluation of change point detection algorithms. *arXiv preprint arXiv:2003.06222* (2020).

[59] Jan N. van Rijn and Frank Hutter. 2017. Hyperparameter Importance Across Datasets. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2017).

[60] Shikhar Verma, Yuichi Kawamoto, Zubair Md. Fadlullah, Hiroki Nishiyama, and Nei Kato. 2017. A Survey on Network Methodologies for Real-Time Analytics of Massive IoT Data and Open Research Issues. *IEEE Communications Surveys & Tutorials* 19 (2017), 1457–1477.

[61] Jie Wan, Michael J. O'Grady, and Gregory M. P. O'Hare. 2015. Dynamic sensor event segmentation for real-time activity recognition in a smart home context. *Personal and Ubiquitous Computing* 19 (2015), 287–301.

[62] Qingsong Wen, Jingkun Gao, Xiaomin Song, Liang Sun, Huan Xu, and Shenghuo Zhu. 2019. RobustSTL: A Robust Seasonal-Trend Decomposition Algorithm for Long Time Series. *Proceedings of the AAAI Conference on Artificial Intelligence* 33, 01 (Jul. 2019), 5409–5416. https://doi.org/10.1609/aaai.v33i01.33015409

[63] Wolfram Wingerath, Felix Gessert, Steffen Friedrich, and Norbert Ritter. 2016. Real-time stream processing for Big Data. *it - Information Technology* 58 (2016), 186 – 194.

[64] J. H. Woollam, Jannes Munchmeyer, Frederik Tilmann, Andreas Rietbrock, Dietrich Lange, Thomas Bornstein, Tobias Diehl, Carlo Giunchi, Florian Haslinger, Dario Jozinovi'c, Alberto Michelini, Joachim Saul, and Hugo Soto. 2022. SeisBench—A Toolbox for Machine Learning in Seismology. *Seismological Research Letters* (2022).

[65] Kenji Yamanishi and Jun'ichi Takeuchi. 2002. A unifying framework for detecting outliers and change points from non-stationary time series data. *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining* (2002).

[66] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Diego Furtado Silva, Abdullah Al Mueen, and Eamonn J. Keogh. 2016. Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View That Includes Motifs, Discords and Shapelets. *2016 IEEE 16th International Conference on Data Mining (ICDM)* (2016), 1317–1322.

[67] Liang Zhang, Noura A. Alghamdi, Huayi Zhang, Mohamed Y. Eltabakh, and Elke A. Rundensteiner. 2022. PARROT: pattern-based correlation exploitation in big partitioned data series. *The VLDB Journal* (2022).

[68] Yan Zhu, Zachary Schall-Zimmerman, Nader Shakibay Senobari, Chin-Chia Michael Yeh, Gareth J. Funning, Abdullah Al Mueen, Philip Brisk, and Eamonn J. Keogh. 2016. Matrix Profile II: Exploiting a Novel Algorithm and GPUs to Break the One Hundred Million Barrier for Time Series Motifs and Joins. *2016 IEEE 16th International Conference on Data Mining (ICDM)* (2016), 739–748.

[69] Yan Zhu, Zachary Schall-Zimmerman, Nader Shakibay Senobari, Chin-Chia Michael Yeh, Gareth J. Funning, Abdullah Al Mueen, Philip Brisk, and Eamonn J. Keogh. 2017. Exploiting a novel algorithm and GPUs to break the ten quadrillion pairwise comparisons barrier for time series motifs and joins. *Knowledge and Information Systems* 54 (2017), 203–236.