Time Series Compression Survey

GIACOMO CHIAROT and CLAUDIO SILVESTRI, Department of Environmental Sciences, Informatics, and Statistics of Ca' Foscari University of Venice

Smart objects are increasingly widespread and their ecosystem, also known as the Internet of Things (IoT), is relevant in many application scenarios. The huge amount of temporally annotated data produced by these smart devices demands efficient techniques for the transfer and storage of time series data. Compression techniques play an important role toward this goal and, even though standard compression methods could be used with some benefit, there exist several ones that specifically address the case of time series by exploiting their peculiarities to achieve more effective compression and more accurate decompression in the case of lossy compression techniques. This article provides a state-of-the-art survey of the principal time series compression techniques, proposing a taxonomy to classify them considering their overall approach and their characteristics. Furthermore, we analyze the performances of the selected algorithms by discussing and comparing the experimental results that were provided in the original articles.

The goal of this article is to provide a comprehensive and homogeneous reconstruction of the state-of-the-art, which is currently fragmented across many articles that use different notations and where the proposed methods are not organized according to a classification.

CCS Concepts: • **Theory of computation** → **Data compression**; • **Information systems** → *Data streaming;*

Additional Key Words and Phrases: Time series, compression, streams

ACM Reference format:
Giacomo Chiarot and Claudio Silvestri. 2023. Time Series Compression Survey. *ACM Comput. Surv.* 55, 10, Article 198 (February 2023), 32 pages.
https://doi.org/10.1145/3560814

1 INTRODUCTION

Time series are relevant in several contexts, and the **Internet of Things** (**IoT**) ecosystem is among the most pervasive ones. IoT devices, indeed, can be found in different applications, ranging from health care (smart wearables) to industrial ones (smart grids) [1], producing a large amount of time-series data. For instance, a single Boeing 787 fly can produce about half a terabyte of data from sensors [2]. In those scenarios, characterized by high data rates and volumes, time series compression techniques are a sensible choice to increase the efficiency of collection, storage, and

This research was partially funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No 814496.

Authors' address: G. Chiarot, Department of Environmental Sciences, Informatics, and Statistics of Ca' Foscari University, Via Torino 155, Venice, Italy, 30172; email: giacomo.chiarot@unive.it; C. Silvestri (corresponding author), European Centre for Living Technology (ECLT), Ca' Bottacin Dorsoduro 3911, Venice, Italy, 30123; email: silvestri@unive.it.
Updated author affiliation: G. Chiarot, Department of Environmental Sciences, Informatics, and Statistics of Ca' Foscari University, Italy; C. Silvestri, European Centre for Living Technology (ECLT), Italy.

0360-0300/2023/02-ART198 $15.00
https://doi.org/10.1145/3560814

analysis of such data. In particular, the need to include in the analysis both information related to the recent and the history of the data stream leads to considering data compression as a solution to optimize space without losing the most important information. A direct application of time series compression, for example, can be seen in Time Series Management Systems (or Time Series Database) in which compression is one of the most significant steps [3].

There exists an extensive literature on data compression algorithms, both on generic purpose ones for finite-size data and on domain-specific ones, for example, for images, video, and audio data streams. This survey aims at providing an overview of the state-of-the-art in time series compression research, specifically focusing on general-purpose data compression techniques that are either developed for time series or working well with time series.

The algorithms we chose to summarize can deal with the continuous growth of time series over time and are suitable for generic domains (as in the different applications in the IoT). Furthermore, these algorithms take advantage of the peculiarities of time series produced by sensors, such as

— **Redundancy:** some segments of a time series can frequently appear inside the same or other related time series;
— **Approximability:** sensors in some cases produce time series that can be approximated by functions;
— **Predictability:** some time series can be predictable, for example using deep neural network techniques.

The main contribution of this survey is to present a reasoned summary of the state-of-the-art in time series compression algorithms, which are currently fragmented among several sub-domains ranging from databases to IoT sensor management. Moreover, we propose a taxonomy of time series compression techniques based on their approach (dictionary-based, functional approximation, autoencoders, sequential, others) and their properties (adaptiveness, lossless reconstruction, symmetry, tuneability of max error or minimum compression ratio), anticipated in visual form in Figure 1 and discussed in Section 3, that will guide the description of the selected approaches. Finally, we recapitulate the results of performance measurements indicated in the described studies.

1.1 Outline of the Survey

Section 2 describes the method we applied to select the literature included in this survey. Section 3 provides some definitions regarding time series, compression, and quality indexes. Section 4 describes the compression algorithms and is structured according to the proposed taxonomy. Section 5 summarizes the experimental results found in the studies that originally presented the approaches we describe. A summary and conclusions are presented in Section 6.

2 LITERATURE SELECTION

2.1 Background and Motivation

In the scientific literature, many surveys were published related to data compression, addressing particular domains (e.g., images, music, and video) or type of compression (lossy or lossless), but there is not an exhaustive survey considering time series.

2.2 Literature Selection

We conducted an extensive literature search on the ACM Digital Library using the query *[Title: serial sequences time series] AND [Title: compression]*, obtaining 2,097 records. We carefully inspected the remaining articles and filtered out articles that were deemed to not be relevant. Furthermore, the survey also includes several additional articles hand-picked by the authors among the relevant reference of other considered articles, by snowballing sampling, or published in the last five years in the same conference/journal as one of the considered article.

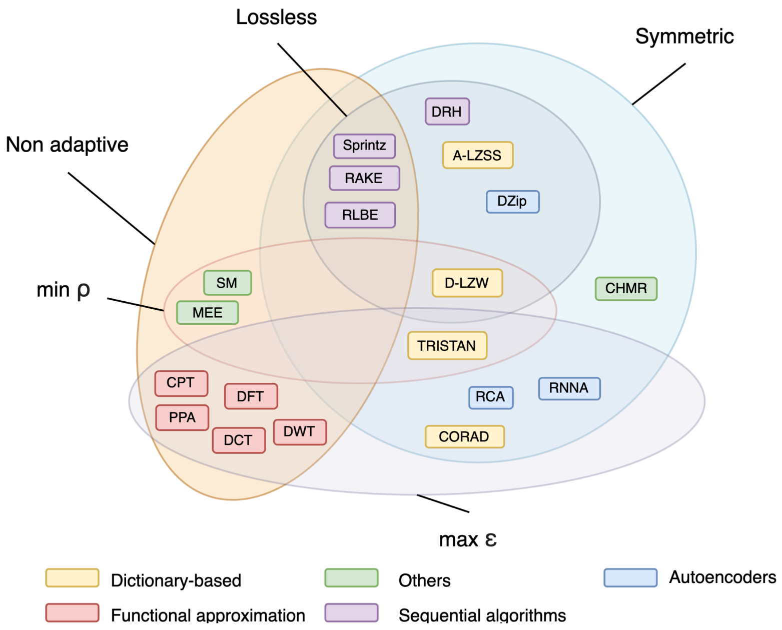


Fig. 1. Visual classification of time series compression algorithms.

2.3 Scope

In time series mining, time series analysis, and machine learning fields, there are many dimensionality reduction techniques that are used to achieve better results in classification and data analysis tasks [40, 41]. An overview of these techniques is presented in [50] with a detailed taxonomy. Most of these techniques are not designed to reconstruct the original data after dimensionality reduction in a way that the error between the original and the reconstructed data is reasonably small, as, for example, natural language representation, trees, and random mapping. Our survey focuses on those techniques that allow using the compressed representation to reconstruct the original data with a reasonably small or zero error.

3 BACKGROUND

3.1 Time Series

Time series are defined as a collection of data, sorted in ascending order according to the timestamp t_i associated with each element. They are divided into

— **Univariate Time Series (UTS):** elements inside the collection are real values;
— **Multivariate Time Series (MTS):** elements inside the collections are arrays of real values, in which each position in the array is associated with a time series feature.

For instance, the temporal evolution of the average daily price of a commodity as the one represented in the plot in Figure 2 can be modeled as a UTS, whereas the summaries of daily exchanges for a stock (including opening price, closing price, the volume of trades and other information) can be modeled as an MTS.

Using a formal notation, time series can be written as

$$TS = [(t_1, x_1), \dots, (t_n, x_n)], x_i \in \mathbb{R}^m, \tag{1}$$

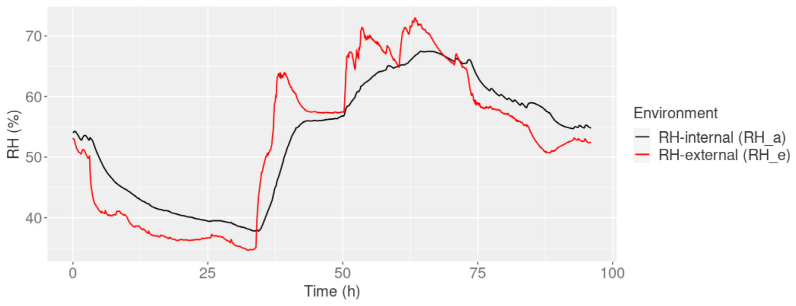


Fig. 2. Example of an MTS representing relative humidity fluctuations inside and outside an artwork's container.

where n is the number of elements inside a time series and m is the vector dimension of MTS. For UTS, $m = 1$. Given $k \in [1, n]$, we write $TS[k]$ to indicate the kth element (t_k, x_k) of the time series TS.

A time series can be divided into segments, defined as a portion of the time series without any missing elements and ordering preserved:

$$TS_{[i,j]} = [(t_i, x_i), \ldots, (t_j, x_j)], \tag{2}$$

where $\forall k \in [i, j], TS[k] = TS_{[i,j]}[k - i + 1]$.

3.2 Compression

Data compression, also known as *source coding*, is defined in [4] as "the process of converting an input data stream (the source stream or the original raw data) into another data stream (the output, the bitstream, or the compressed stream) that has a smaller size". This process can take advantage of the ***Simplicity Power*** (**SP**) theory, formulated in [5], in which the compression goal is to remove redundancy while having high descriptive power.

The decompression process, complementary to the compression one, is indicated also as *source decoding*, and tries to reconstruct the original data stream from its compressed representation.

Compression algorithms can be described with the combination of different classes, shown in the following list:

— **Non-adaptive—adaptive:** a non-adaptive algorithm that does not need a training phase to work efficiently with a particular dataset or domain since the operations and parameters are fixed, while an adaptive one does;
— **Lossy—lossless:** algorithms can be lossy if the decoder does not return a result that is identical to original data, or lossless if the decoder result is identical to original data;
— **Symmetric—non-symmetric:** an algorithm is symmetric if the decoder performs the same operations of the encoder in reverse order, whereas a non-symmetric one uses different operations to encode and decode a time series.

In the particular case of time series compression, a compression algorithm (*encoder*) takes in input one Time Series TS of size s and returns its compressed representation TS' of size s', where $s' < s$ and the size is defined as the bits needed to store the time series: $E(TS) = TS'$. From the compressed representation TS', using a *decoder*, it is possible to reconstruct the original time series: $D(TS') = \overline{TS_s}$. If $\overline{TS} = TS_s$ then the algorithm is lossless, otherwise it is lossy.

In Section 4, there are shown the most relevant categories of compression techniques and their implementation.

3.3 Quality Indices

To measure the performances of a compression encoder for time series, three characteristics are considered: compression ratio, speed, and accuracy.

Compression ratio. This metric measures the effectiveness of a compression technique, and it is defined as

$$\rho = \frac{s'}{s}, \tag{3}$$

where s' is the size of the compressed representation and s is the size of the original time series. Its inverse $\frac{1}{\rho}$ is named *compression factor*. An index used for the same purpose is the *compression gain*, defined as

$$c_g = 100 \log_e \frac{1}{\rho}. \tag{4}$$

Accuracy, also called distortion, measures the fidelity of the reconstructed time series with respect to the original. It is possible to use different metrics to determine fidelity [6]:

— **Mean Squared Error:** $MSE = \frac{\sum_{i=1}^{n}(x_i - \overline{x}_i)^2}{n}$.
— **Root Mean Squared Error:** $RMSE = \sqrt{MSE}$.
— **Signal to Noise Ratio:** $SNR = \frac{\sum_{i=1}^{n} x_i^2/n}{MSE}$.
— **Peak Signal to Noise Ratio:** $PSNR = \frac{x_{pick}^2}{MSE}$ where x_{peak} is the maximum value in the original time series.

4 COMPRESSION ALGORITHMS

In this section, we present the most relevant time series compression algorithms by describing in short summaries their principles and peculiarities. We also provide a pseudo-code for each approach, focusing more on style homogeneity than on the faithful reproduction of the original pseudo-code proposed by the authors. For a more detailed description, we refer the reader to the complete details available in the original articles. Below the full list of algorithms described in this section, divided by approach:

(1) **Dictionary-based (DB)**:
  1.1. TRISTAN;
  1.2. CORAD;
  1.3. A-LZSS;
  1.4. D-LZW.
(2) **Functional Approximation (FA)**
  2.1. Piecewise Polynomial Approximation (PPA);
  2.2. Chebyshev Polynomial Transform (CPT);
  2.3. Discrete Wavelet Transform (DWT);
  2.4. Discrete Fourier Transform (DFT);
  2.5. Discrete Cosine Transform (DCT).
(3) **Autoencoders:**
  3.1. Recurrent Neural Network Autoencoder (RNNA);
  3.2. Recurrent Convolutional Autoencoder (RCA);
  3.3. DZip.
(4) **Sequential Algorithms (SA)**:
  4.1. Delta encoding, Run-length, and Huffman (DRH);
  4.2. Sprintz;
  4.3. Run-Length Binary Encoding (RLBE);
  4.4. RAKE.

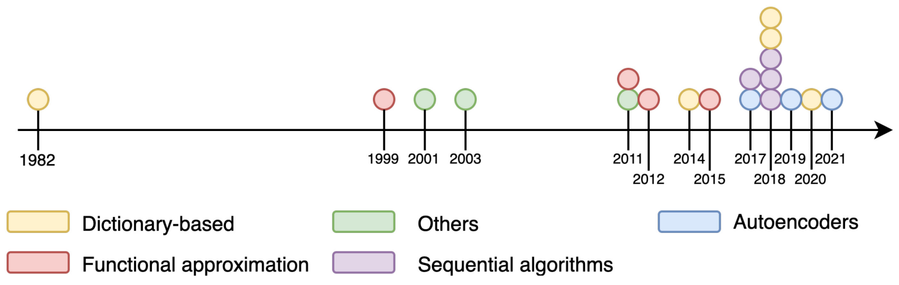


Fig. 3. Chronological trend of compression techniques for time series.

(5) **Others:**
 5.1. Major Extrema Extractor (MEE);
 5.2. Segment Merging (SM);
 5.3. Continuous Hidden Markov Chain (CHMC).

RNNA algorithm can be applied both to univariate and multivariate time series. The other algorithms can also handle MTS by extracting each feature as an independent time series. All the algorithms accept time series represented with real values.

The methods we considered span a temporal interval of more than 20 years, with a significant outlier dating back to the '80s. Figure 3 graphically represents the temporal trend of adoption of the different approaches for the methods we considered.

4.1 Dictionary-based (DB)

This approach is based on the principle that time series share some common segments, without considering timestamps. These segments can be extracted into atoms, such that a time series segment can be represented with a sequence of these atoms. Atoms are then collected into a dictionary that associates each atom with a univocal key used both in the representation of time series and to search efficiently their content. The choice of atoms length should guarantee a low decompression error and maximize the compression factor at the same time. Algorithm 1 shows how the training phase works at a high level: `createDictionary` function computes a dictionary of segments given a dataset composed of time series and a threshold value `th`.

The `find` function searches in the dictionary if there exists a segment that is similar to the segment `s` with a distance lower than threshold `th`; a possible index of distance can be *MSE*. If a match is found, the algorithm merges the segment `s` with the matched one to achieve generalization. Larger `th` value results in higher compression and lower reconstruction accuracy, and a dictionary with lower dimension.

After the segment dictionary is created, the compression phase takes in input one time series, and each segment is replaced with its key in the dictionary. If some segments are not present in the dictionary, they are left uncompressed, or a new entry is added to the dictionary. Algorithm 2 shows how the compression phase works: the `compress` function takes in input a time series to compress, the dictionary created during the training phase, a threshold value, and returns the compressed time series as a list of indices and segments.

The compression achieved by this technique can be either lossy or lossless, depending on the implementation.

The main challenges for this architecture are to

— maximize the searching speed to find time series segments in the dictionary;
— make the time series segments stored in the dictionary more general as possible to minimize the distance in the compression phase.

4.1.1 TRISTAN. One implementation of the DB architecture is TRISTAN [7], an algorithm divided into two phases: the learning phase and the compression phase.

Learning phase. The dictionary used in this implementation can be created by domain experts that add typical patterns or by learning a training set. To learn a dictionary from a training set $T = [t_1, \dots, t_n]$ of n segments, the following minimization problem has to be solved:

$$D = \arg\min_D \sum_{i=1}^{n} \|w_i D - t_i\|_2. \quad \text{Under the constraint: } \|w_i\|_0 \leq sp, \tag{5}$$

where D is the obtained dictionary, sp is a fixed parameter representing sparsity, w_i is the compressed representation of segment i, and $w_i D$ is the reconstructed segment. The meaning of this formulation is that the solution to be found is a dictionary that minimizes the distance between original and reconstructed segments.

The problem shown in Equation (5) is NP-hard [7], thus an approximate result is computed. A technique for approximating this result is shown in [8].

Compression phase. Once the dictionary is built, the compression phase consists in finding w such that:

$$s = w \cdot D, \tag{6}$$

where D is the dictionary, s is a segment and $w \in \{0, 1\}^k$, and k is the length of the compressed representation. An element of w in position i is 1 if the $a_i \in D$ is used to reconstruct the original segment, 0 otherwise.

Finding a solution for Equation (6) is an NP-hard problem and, for this reason, the matching pursuit method [9] is used to approximate the original problem:

$$\overline{s} = \arg\min_w \|wD - s\|_2. \quad \text{Under the constraint: } \|\overline{s}\|_0 \leq sp, \tag{7}$$

where sp is a fixed parameter representing sparsity.

Reconstruction phase. Having a dictionary D and a compressed representation w of a segment s, it is possible to compute $\overline{s}$ as

$$\overline{s} = wD. \tag{8}$$

4.1.2 CORAD. This implementation extends the idea presented in TRISTAN [10]. The main difference is that it adds autocorrelation information to get better performances in terms of compression ratio and accuracy.

Correlation between two time series TS^A and TS^B is measured with the Pearson correlation coefficient:

$$r = \frac{\sum_i^n (x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_i^n (x_i - \overline{x})^2} \sqrt{\sum_i^n (y_i - \overline{y})^2}}, \tag{9}$$

where x_i is an element of TS_n^A, y_i is an element of TS_n^B and $\overline{x}, \overline{y}$ are mean values of the corresponding time series. This coefficient can be applied also to segments that have different ranges of values and $r \in [-1, q]$ where 1 expresses the maximum linear correlation, −1 the maximum linear negative correlation, and 0 no linear correlation.

Time series are divided into segments and time windows are set. For each window, correlation is computed between each segment belonging to it, and results are stored in a correlation matrix $M \in \mathbb{R}^{n \times n}$ where n is the segment number of elements.

Compression phase. During this phase, segments are sorted from the less correlated to the most correlated. To do this, a correlation matrix M is used. The metric used to measure how much one segment is correlated with all the others is the absolute sum of the correlations, computed as the sum of each row of M.

Knowing correlation information, a dictionary is used only to represent segments that are not correlated with others, as in TRISTAN implementation. While the other segments are represented solely using correlation information.

Reconstruction phase. The reconstruction phase starts with the segment represented with dictionary atoms while the others are reconstructed looking at the segment to which they are correlated.

This process is very similar to the one proposed in TRISTAN, with the sole difference that segments represented in the dictionary are managed differently than those that are not.

4.1.3 *Accelerometer LZSS (A-LZSS).*

A-LZSS is an algorithm built on top of the LZSS algorithm [11] for searching matches [12]. A-LZSS algorithm uses Huffman codes, generated offline using frequency distributions. In particular, this technique considers blocks of size $s = 1$ to compute frequencies and build the code: an element of the time series will be replaced by a variable number of bits. Moreover, larger blocks can be considered and, in general, having larger blocks gives better compression performances at the cost of larger Huffman code tables.

The implementation of this technique is shown in Algorithm 3, where:

— `minM`: the minimum match length, which is asserted to be `minM` > `0`;
— `Ln`: determines the lookahead distance, as 2^{Ln};
— `Dn`: determines the dictionary atoms length, as 2^{Dn};
— `longestMatch`: is a function that returns the index I of the found match and the length L of the match. If the length of the match is too small, then the Huffman code representation of `s` is sent as the compression representation, otherwise, the index and the length of the match are sent, and the next L elements are skipped.

This implementation uses a brute-force approach, with complexity $O(2^{Dn} \cdot 2^{Ln})$ but it is possible to improve it by using hashing techniques.

4.1.4 *Differential LZW (D-LZW).*

The core of this technique is the creation of a very large dictionary that grows over time: once the dictionary is created, if a buffer block is found inside the dictionary it is replaced by the corresponding index, otherwise, the new block is inserted in the dictionary as a new entry [13].

Adding new blocks guarantees lossless compression, but has the drawback of having too large dictionaries. This makes the technique suitable only for particular scenarios (i.e, input streams composed of words/characters or when the values inside a block are quantized).

Another drawback of this technique is how the dictionary is constructed: elements are simply appended to the dictionary to preserve the indexing of previous blocks. For a simple implementation of the dictionary, the complexity for each search is $O(n)$ where n is the size of the dictionary. This complexity can be improved by using more efficient data structures.

4.2 Function Approximation (FA)

The main idea behind function approximation is that a time series can be represented as a function of time. Since finding a function that approximates the whole time series is infeasible due to the presence of new values that cannot be handled, the time series is divided into segments and for each of them, an approximating function is found.

Exploring all the possible functions $f : T \to X$ is not feasible, thus implementations consider only one family of functions and try to find the parameters that better approximate each segment. This makes the compression lossy.

A point of strength is that it does not depend on the data domain, so no training phase is required since the regression algorithm considers only single segments in isolation.

4.2.1 Piecewise Polynomial Approximation (PPA). This technique divides a time series into several segments of fixed or variable length and tries to find the best polynomials that approximate segments.

Despite the compression is lossy, a maximum deviation from the original data can be fixed *a priori* to enforce a given reconstruction accuracy.

The implementation of this algorithm is described in [14] where the authors apply a greedy approach and three different online regression algorithms for approximating constant functions, straight lines, and polynomials. These online algorithms are

— the PMR-Midrange algorithm, which approximates using constant functions [15];
— the optimal approximation algorithm, described in [16], which uses linear regression;
— the randomized algorithm presented in [17], which approximates using polynomials.

The algorithm used in [14] for approximating a time series segment is explained in Algorithm 4.

This algorithm finds repeatedly the polynomial of degree between 0 and a fixed maximum that can approximate the longest segment within the threshold error, yielding the maximum local compression factor. After a prefix of the stream has been selected and compressed into a polynomial, the algorithm analyzes the following stream segment. A higher value of ϵ returns higher compression and lower reconstruction accuracy. The fixed maximum polynomial degree ρ affects compression speed, accuracy, and compression ratio: higher values slow down compression and reduce the compression factor, but return higher reconstruction accuracy.

4.2.2 Chebyshev Polynomial Transform (CPT). Another implementation of polynomial compression can be found in [18]. In this article, the authors show how a time series can be compressed in a sequence of finite Chebyshev polynomials.

The principle is very similar to the one shown in Section 4.2.1 but based on the use of a different type of polynomial. Chebyshev Polynomials are of two types, $T_n(x)$, $U_n(x)$, defined as [19]:

$$T_n(x) = \frac{n}{2}\sum_{k=0}^{n/2}(-1)^k \frac{(n-k-1)!}{k!(n-2k)!}(2x)^{n-2k}, |x| < 1, \tag{10}$$

$$U_n(x) = \sum_{k=0}^{n/2}(-1)^k \frac{(n-k)!}{k!(n-2k)!}(2x)^{n-2k}, |x| < 1, \tag{11}$$

where $n \geq 0$ is the polynomial degree.

4.2.3 Discrete Wavelet Transform (DWT). DWT uses wavelet functions to transform time series. Wavelets are functions that, similarly to a wave, start from zero, and end with zero after some oscillation. An application of this technique can be found in [20].

This transformation can be written as

$$\Psi_{m,n}(t) = \frac{1}{\sqrt{a^m}}\Psi\left(\frac{t-nb}{a^m}\right), \tag{12}$$

where $a > 1$, $b > 0$, $m, n \in Z$.

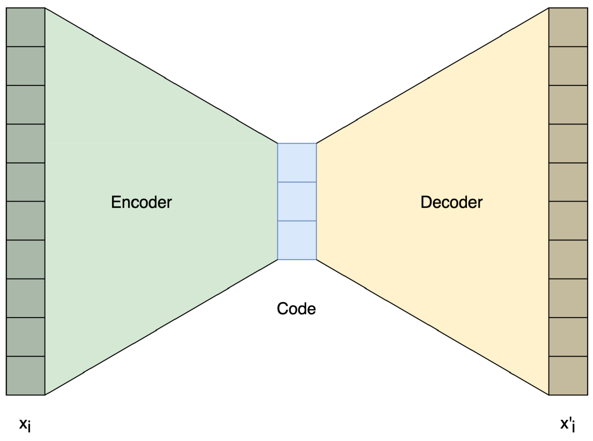


Fig. 4. Simple autoencoder architecture.

To recover one transformed signal, the following formula can be applied:

$$x(t) = \sum_{m \in Z} \sum_{n \in Z} \langle x, \Psi_{m,n} \rangle \cdot \Psi_{m,n}(t). \tag{13}$$

4.2.4 Discrete Fourier Transform (DFT). Together with the DWT, the DFT is commonly used for signal compression. Given a time series $[(t_1, x_1), \ldots, (t_n, x_n)]$, it is defined as

$$X_k = \sum_{n=1}^{N} x_n e^{\frac{-i2\pi kn}{N}}.$$

Where $e^{\frac{i2\pi}{N}}$ is a primitive Nth root of 1.

To efficiently compute this transformation, the Fast Fourier transform algorithm can be used, as the one introduced here [42]. As described in [43], compressing a time series can be done after applying the DFT by cutting coefficients that are close to zero. This can be done also by fixing a compression ratio and discarding all the coefficients after a certain index.

4.2.5 Discrete Cosine Transform (DCT). Given a time series $[(t_1, x_1), \ldots, (t_n, x_n)]$, the DCT is defined as

$$c_k = \sum_{n=0}^{N} x_n \cdot \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right)k\right].$$

The computation of this transformation is computationally expensive, so efficient implementations can be used, as the one presented in [44].

Similarly to the DFT, compression of time series can be achieved by cutting the coefficients that are close to zero or by fixing a compression ratio and discarding all the coefficients after a certain index. An example of this technique can be found in [45].

4.3 Autoencoders

An autoencoder is a particular neural network that is trained to give as output the same values passed as input. Its architecture is composed of two symmetric parts: encoder and decoder. Giving an input of dimension n, the encoder gives as output a vector with dimensionality $m < n$, called code, while the decoder reconstructs the original input from the code, as shown in Figure 4 [21].

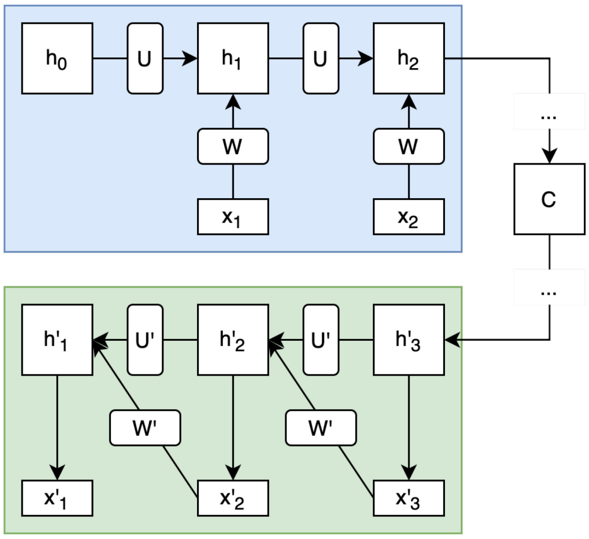


Fig. 5. General structure of an RNN encoder and decoder.

4.3.1 Recurrent Neural Network Autoencoder (RNNA). RNNA compression algorithms exploit recurrent neural networks [22] to achieve a compressed representation of a time series. Figure 5 shows the general unrolled structure of a recurrent neural network encoder and decoder. The encoder takes in input time series elements, which are combined with hidden states. Each hidden state is then computed starting from the new input and the previous state. The last hidden state of the encoder is passed as the first hidden state of the decoder, which applies the same mechanism, with the only difference being that each hidden state provides an output. The output provided by each state is the reconstruction of the relative time series element and is passed to the next state.

New hidden state h_t is obtained by applying:

$$h_t = \phi(Wx_t + Uh_{t-1}), \tag{14}$$

where ϕ is a logistic sigmoid function or the hyperbolic tangent.

One application of this technique is shown in [23], in which also Long Short-Term Memory [24] is considered. This implementation compresses time series segments of different lengths using autoencoders. The compression achieved is lossy and a maximal loss threshold ϵ can be enforced.

The training set is preprocessed considering temporal variations of data applying:

$$\Delta(\mathcal{L}) = \sum_{t \in \mathcal{L}} |x_t - x_{t-1}|, \tag{15}$$

where $\mathcal{L}$ is the local time window.

The value obtained by Equation (15) is then used to partition the time series, such that each segment has a total variation close to a predetermined value τ.

Algorithm 5 shows an implementation of the RNN autoencoder approach, with an error threshold ϵ, where:

- `RAE a` is the recurrent autoencoder trained on a training set, composed of an encoder and a decoder;
- `getError` computes the reconstruction error between the original and reconstructed segment;
- ϵ is the error threshold value.

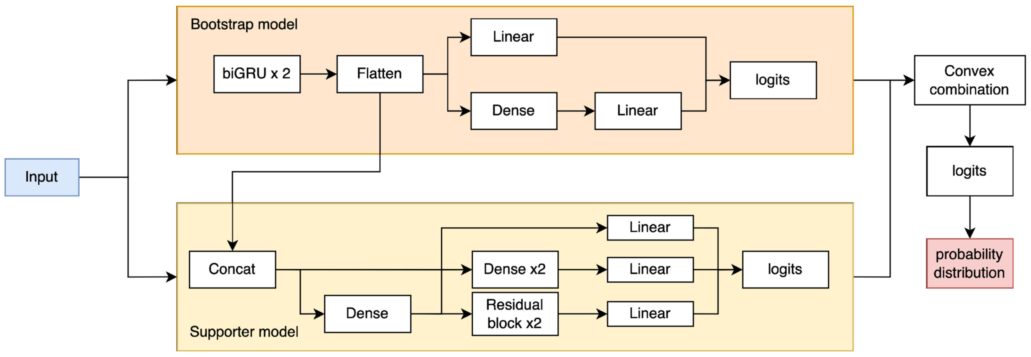


Fig. 6. DZip model infrastructure.

4.3.2 Recurrent Convolutional Autoencoder (RCA). A variation of the technique presented in the previous subsection is proposed in [49] and consists of adding convolutional layers. Convolutional layers are used mainly for images data to extract local features. In the case of time series, this layer allows extracting local fluctuations and addressing complex inputs.

4.3.3 DZip. DZip is a lossless recurrent autoencoder [47]. This model also uses the prediction technique: it tries to predict the next symbol, having, in this case, a fixed vocabulary. To achieve a lossless compression, it combines two modules: the bootstrap model and the supporter model, as shown in Figure 6.

The bootstrap model is composed by stacking two bidirectional gated recurrent units, followed by linear and dense layers. The output of this module is a probability distribution, that is, the probability of each symbol in the fixed vocabulary to be the next one.

The supporter model is used to better estimate the probability distribution with respect to the bootstrap model. It is composed by stacking three neural networks which act as independent predictors of varying complexity.

The two models are then combined by applying the following equation:

$$o = \lambda o_b + (1 - \lambda) o_s.$$

Where o_b is the output of the bootstrap model, o_s is the output of the supporter model, and $\lambda \in [0, 1]$ is a learnable parameter.

4.4 Sequential Algorithms

This architecture is characterized by combining sequentially several simple compression techniques. Some of the most used techniques are

— Huffman coding;
— Delta encoding;
— Run-length encoding;
— Fibonacci binary encoding.

These techniques, summarized below, are the building blocks of the methods presented in the following subsections.

Huffman coding. Huffman coding is the basis of many compression techniques since it is one of the necessary steps, as for the algorithm shown in Section 4.4.4.

The encoder creates a dictionary that associates each symbol with a binary representation and replaces each symbol of the original data with the corresponding representation. The compression algorithm is shown in Algorithm 6 [6].

The createPriorityList function creates a list of elements ordered from the less frequent to the most frequent, addTree function adds to the tree a father node with its children and createDictionary assigns 0 and 1, respectively, to the left and right arcs and creates the dictionary assigning to characters the sequence of 0|1 in the route from the root to a leaf that represents the characters. Since the priority list is sorted according to frequency, more frequent characters are inserted closer to the root and they are represented using a shorter code.

The decoder algorithm is very simple since the decoding process is the inverse of the encoding process: the encoded bits are searched in the dictionary and replaced with the corresponding original symbols.

Delta encoding. This technique encodes a target file with respect to one or more reference files [25]. In the particular case of time series, each element at time t is encoded as $\Delta(x_t, x_{t-1})$.

Run-length. In this technique, each run (a sequence in which the same value is repeated consecutively) is substituted with the pair (v_t, o) where v_t is the value at time t and o is the number of consecutive occurrences [26].

Fibonacci binary encoding. This encoding technique is based on the Fibonacci sequence, defined as

$$F(N) = \begin{cases} F(N-1) + F(N-2), & \text{if } N >= 1 \\ N, & \text{N} = -1, \text{N} = 0 \end{cases}. \tag{16}$$

Having $F(N) = a_1 \ldots a_p$ where a_j is the bit at position j of the binary representation of $F(N)$, the Fibonacci binary coding is defined as

$$F_E(N) = \sum_{i=0}^{j} a_i \cdot F(i). \tag{17}$$

Where $a_i \in \{0, 1\}$ is the ith bit of the binary representation of $F(N)$ [27].

4.4.1 Delta Encoding, Run-length, and Huffman (DRH). This technique combines together three well-known compression techniques [28]: delta encoding, run-length encoding, and Huffman code. Since these techniques are all lossless, the compression provided by DRH is lossless too if no quantization is applied.

Algorithm 7 describes the compression algorithm, where Q is the quantization level and is asserted to be ≥ 1: if $Q = 1$ the compression is lossless, and increasing values of Q return a higher compression factor and lower reconstruction accuracy.

The decompression algorithm is the inverse of the compression one: once data is received, it is decoded using the Huffman code and reconstructed using the repetition counter.

Since this kind of algorithm is not computationally expensive, the compression phase can be performed also by low-resource computational units, such as sensor nodes.

4.4.2 Sprintz. Sprintz algorithm [29] is designed for the IoT scenario, in which energy consumption and speed are important factors. In particular, the goal is to satisfy the following requirements:

— Handling of small block size;
— High decompression speed;
— Lossless data reconstruction.

The proposed algorithm is a coder that exploits prediction to achieve better results. In particular, it is based on the following components:

— **Forecasting:** used to predict the difference between new samples and the previous ones through delta encoding or FIRE algorithm [29];

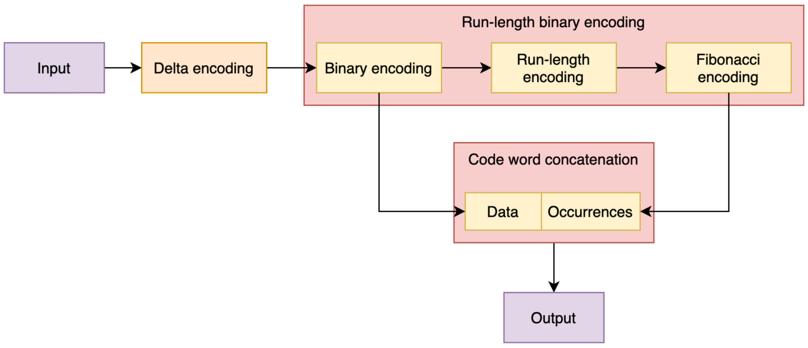


Fig. 7. RLBE compression steps.

Table 1. RAKE Dictionary

symbol	code	length
−1	1	1
+1	01	2
−2	001	3
+2	0001	4
...	...	...
−R	0...1	2R - 1
+R	00...1	2R
0	all zeros	2R

— **Bit packing:** packages are composed of a payload that contains prediction errors and a header that contains information that is used during reconstruction;
— **Run-length encoding:** if a sequence of correct forecasts occurs, the algorithm does not send anything until some error is detected and the length of skipped zero error packages is added as information;
— **Entropy coding:** package headers and payloads are coded using Huffman coding, presented in Section 4.4.

4.4.3 Run-Length Binary Encoding (RLBE). This lossless technique is composed of five steps, combining delta encoding, run-length, and Fibonacci coding, as shown in Figure 7 [30].

This technique is developed specifically for devices characterized by low memory and computational resources, such as IoT infrastructures.

4.4.4 RAKE. RAKE algorithm, presented in [31], exploits sparsity to achieve compression. It is a lossless compression algorithm with two phases: preprocessing and compression.

Preprocessing. In this phase, a dictionary is used to transform original data. For this purpose, many algorithms can be used, such as the Huffman coding presented in Section 4.4, but since the aim is that of obtaining sparsity, the RAKE dictionary uses a code similar to unary coding thus every codeword has at most one bit set to 1. This dictionary does not depend on symbol probabilities, so no learning phase is needed. Table 1 shows a simple RAKE dictionary.

Compression. This phase works, as suggested by the algorithm name, as a rake of n teeth. Figure 8 shows an execution of the compression phase, given a preprocessed input and $n = 4$.

The rake starts at position 0, in case there is no bit set to 1 in the rake interval, then 0 is added to the code, otherwise 1 and followed by the binary representation of relative index for the first

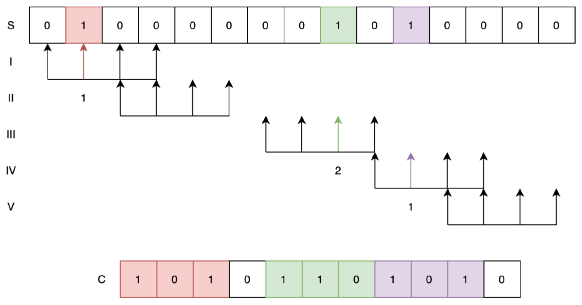


Fig. 8. RAKE algorithm execution.

bit set to 1 in the rake (2 bits in the example in Figure 8). After that, the rake is shifted right of n positions for the first case or starting right to the first found bit. In the figure, the rake is initially at position 0, and the first bit set to 1 is in relative position 1 (output: 1 followed by 01), then the rake advances of 2 positions (after the first 1 in the rake); all bits are set to zero (output: 0) thus that rake is moved forward by 4 places; the first bit set to 1 in the rake has relative index 2 (output: 1 followed by 10) thus the rake is advanced by 3 places and the process continues for the two last rake positions (output: 101 and 0, respectively).

Decompression. The decompression processes the compressed bit stream by replacing 0 with n occurrences of 0 and 1 followed by an offset with a number 0 equal to the offset followed by a bit set to 1. The resulting stream is decoded on the fly using the dictionary.

4.5 Others

In this subsection, we present other time series compression algorithms that cannot be grouped in the previously described categories.

4.5.1 Major Extrema Extractor (MEE). This algorithm is introduced in [32] and exploits time series features (maxima and minima) to achieve compression. For this purpose, strict, left, right, and flat extrema are defined. Considering a time series $TS = [(t_0, x_0), \ldots, (t_n, x_n)]$, x_i is a minimum if it follows these rules:

- **strict:** if $x_i < x_{i-1} \land x_i < x_{i+1}$
- **left:** if $x_i < x_{i-1} \land \exists k > i : \forall j \in [i,k], x_j = x_i \lor x_i < x_{k+1}$
- **right:** if $x_i < x_{i+1} \land \exists k < i : \forall j \in [k,i], x_j = x_i \lor x_i < x_{k-1}$
- **flat:** if $(\exists k > i : \forall j \in [i,k], x_j = x_i \lor x_i < x_{k+1}) \land (\exists k < i : \forall j \in [k,i], x_j = x_i \lor x_i < x_{k-1})$

For maximum points, they are defined similarly.

After defining minimum and maximum extrema, the authors introduce the concept of importance, based on a distance function *dist* and a compression ratio ρ: x_i is an important minimum if

$$\begin{aligned} \exists i_l < i < i_r : \quad & x_i \text{ is minimum in } \{x_l, \ldots, x_r\} \land \\ & dist(x_l, x_i) < \rho \land dist(x_i, x_r) < r. \end{aligned}$$

Important maximum points are defined similarly.

Once important extrema are found, they are used as a compressed representation of the segment. This technique is a lossy compression technique, as the SM one described in the next section. Although the compressed data can be used to obtain original data properties useful for visual data representation (minimum and maximum), it is impossible to reconstruct the original data.

4.5.2 Segment Merging (SM). This technique, presented in [48] and reused in [33] and [50], considers time series with regular timestamps and repeatedly replaces sequences of consecutive

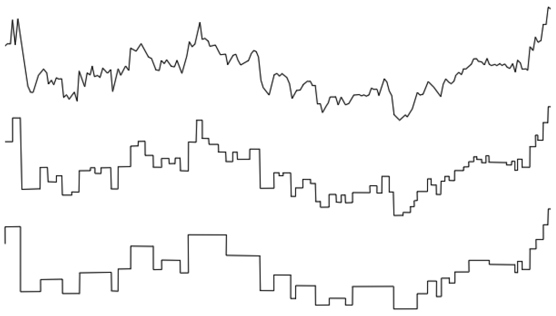


Fig. 9. Example of SM technique, with increasing value time interval.

elements (segments) with a summary consisting of a single value and a representation error, as shown in Figure 9 where the error is omitted.

After compression, segments are represented by tuples (t, y, δ) where t is the starting time of the segment, y is the constant value associated with the segment, and δ is the segment error. The merging operation can be applied either to a set of elements or to a set of segments, to further compress a previously compressed time series. The case of elements is an instance of the case of segments, and it is immediate to generalize from two to more elements/segments. We limit our description to the merge of two consecutive segments represented by the tuples (t_i, y_i, δ_i) and (t_j, y_j, δ_j), where $i < j$, into a new segment represented by the tuple (t, y, δ) computed as

$$t = t_i,$$

$$y = \frac{\Delta t_i \cdot y_i + \Delta t_j \cdot y_j}{\Delta t_i + \Delta t_j},$$

$$\delta = \sqrt{\frac{\Delta t_i \cdot (y_i^2 + \delta_i^2) + \Delta t_j \cdot (y_i^2 + \delta_i^2)}{\Delta t_i + \Delta t_j} - y^2},$$

where Δt_x is the duration of the segment x, thus $\Delta t_i = t_j - t_i$, $\Delta t_j = t_k - t_j$ and t_k is the timestamp of the segment after the one starting at t_j.

The sets of consecutive segments to be merged are chosen to minimize segment error with constraints on a maximal acceptable error and maximal segment duration.

This compression technique is lossy and the result of the compression phase can be considered both as the compressed representation and as the reconstruction of the original time series, without any additional computations executed by a decoder.

4.5.3 Continuous Hidden Markov Chain (CHMC). The idea behind this algorithm is that the data generation process follows a probabilistic model and can be described with a Markov chain [34]. This means that a system can be represented with a set of finite states S and a set of arcs A for transition probabilities between states.

Once the hidden Markov chain is found using known techniques, such as the one presented in [35], a lossy reconstruction of the original data can be obtained by following the chain probabilities.

4.6 Algorithms Summary

When we apply the taxonomy described in Sections 3.2 and 4 to the above techniques we obtain the classification reported in Table 2 that summarizes the properties of the different implementations. To help the reader in visually grasping the membership of the techniques to different parts of the taxonomy, we report the same classification graphically in Figure 1 using Venn diagrams.

Table 2. Compression Algorithms Classification

		DB techniques			
	Non-adaptive	Lossless	Symmetric	min ρ	max ϵ
TRISTAN	-	-	✓	✓	✓
CORAD	-	-	✓	-	✓
A-LZSS	-	✓	✓	-	NA
D-LZW	-	✓	✓	✓	NA
		Function Approximation			
	Non adaptive	Lossless	Symmetric	min ρ	max ϵ
PPA	✓	-	-	✓	✓
CPT	✓	-	-	✓	✓
DWT	✓	-	-	✓	✓
DFT	✓	-	-	✓	✓
DCT	✓	-	-	✓	✓
		Autoencoders			
	Non adaptive	Lossless	Symmetric	min ρ	max ϵ
RNNA	-	-	✓	-	✓
RCA	-	-	✓	-	✓
DZip	-	✓	✓	-	NA
		Sequential algorithms			
	Non adaptive	Lossless	Symmetric	min ρ	max ϵ
DRH	-	✓	✓	-	NA
SPRINTZ	✓	✓	✓	-	NA
RLBE	✓	✓	✓	-	NA
RAKE	✓	✓	✓	-	NA
		Others			
	Non adaptive	Lossless	Symmetric	min ρ	max ϵ
MEE	✓	-	-	✓	-
SM	✓	-	-	✓	-
CHMC	-	-	✓	-	-

Where ✓ indicates if the related property is true,—if it is not, and NA if it is not applicable. Min ρ and max ϵ indicate, respectively, the possibility to set a minimum compression ratio or a maximum reconstruction error.

5 EXPERIMENTAL RESULTS

In this section, we discuss the different performances of the techniques in Section 4, as reported by their authors. To ensure a homogeneous presentation of the different techniques and for copyright reasons, we do not include figures from the works we are describing. Instead, we redraw the figures using the same graphical style for all of them and maintain a faithful reproduction with respect to the represented values.

The experiments presented in those studies were based on the following datasets:

— ICDM challenge [36]—Traffic flows;
— RTE France[1]—Electrical power consumptions;

[1]https://data.rte-france.com.

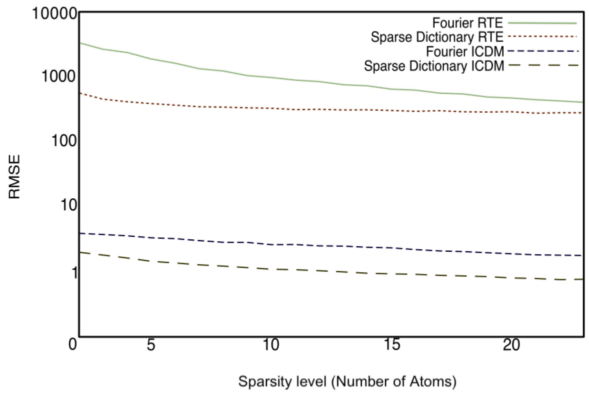


Fig. 10. RMSE results depending on sparsity for TRISTAN [7].

- ACSF1[2]—Power consumptions;
- BAFU[3],—Hydrogeological data;
- GSATM[4]— Dynamic mixtures of carbon monoxide (CO) and humid synthetic air in a gas chamber;
- PigAirwayPressure[5]—vital signs;
- SonyAIBORobotSurface2[5]—X-axis of robot movements;
- SMN [37]—Seismic measures in Nevada;
- HAS [38]—Human activity from smartphone sensors;
- PAMAP [39]—Wearable devices motion and heart frequency measurements;
- MSRC-12[6]—Microsoft Kinect gestures;
- UCI gas dataset[4]—Measuring gas concentration during chemical experiments;
- AMPDs[7]—Measuring maximum consumption of water, electricity, and natural gas in houses.

5.1 Algorithms

To foster the comparison of the different algorithms, we report synoptically the accuracy and compression ratio obtained in the experiments described by their authors.

5.1.1 TRISTAN.

Accuracy. The creation of the dictionary depends on the dictionary sparsity parameter that is directly related to its size and the accuracy can be influenced by this value, as shown in Equation (5). Experimental results are shown in Figure 10, where sparsity is related to RMSE.

Compression ratio. The sparsity parameter also affects the compression ratio. To assess this dependency, the authors execute experiments on two different datasets both containing one measurement per minute with daily segments: RTE and ICDM, using dictionaries consisting respectively of 16 and 131 atoms.

The resulting compression ratios are $\rho = 0.5$ for RTE and $\rho = 0.05$ for ICDM, thus higher sparsity values (larger dictionaries) allow for better compression ratios.

[2]http://www.timeseriesclassification.com.
[3]https://www.bafu.admin.ch.
[4]https://archive.ics.uci.edu.
[5]https://www.cs.ucr.edu/.
[6]https://www.microsoft.com.
[7]http://ampds.org.

Table 3. D-LZW Compression Ratio Comparison

	D-LZW	Bzip2	Gzip
Compression ratio (%)	76.47	60.52	53.63

5.1.2 *CORAD.*

Accuracy. The authors of CORAD measure accuracy using MSE and the best-reported results for their algorithms are 0.03 for the BAFU dataset, 0.11 for the GSATM dataset, and 0.04 for the ACSF1 dataset. As for TRISTAN, accuracy depends on time series segments redundancy and correlation.

Compression ratio. The best reported results for CORAD compression ratio are $\rho = 0.04$ for the ACSF1 dataset, $\rho = 0.07$ for the BAFU dataset and $\rho = 0.06$ for the GSATM dataset.

The compression ratio is affected by the parameters used during training and encoding: error threshold, sparsity, and segment length. Their effects are depicted in the three plots in Figure 11, which represent the compression factor ($\frac{1}{\rho}$) obtained for different values of the parameters.

5.1.3 A-LZSS. The compression ratio obtained by the A-LZSS algorithm depends on two parameters: L_n and D_n, which are respectively the lookahead distance, and the dictionary atoms length. From the experiments conducted by the authors in [12], the best compression ratio is obtained setting $L_n = 5$ and $D_n = 10$. Increasing L_n and decreasing D_n penalize this quality index, as values of $L_n < 4$.

5.1.4 D-LZW. The authors did not provide a detailed study on the compression performances, limiting the experimental results only to EGC and PPG signals. Nevertheless, they did a comparison with Bzip2 and Gzip, giving the results shown in Table 3.

5.1.5 Piecewise Polynomial Approximation (PPA). Accuracy and compression ratio of PPA are affected by the maximum polynomial degree and the maximum allowed deviation, the parameters of the method as described in Section 4.2.

Accuracy. Errors are bounded by the maximum deviation parameter and the maximum polynomial degree can affect accuracy: deviation is always under the threshold and using higher degree polynomials yield more precise reconstructions.

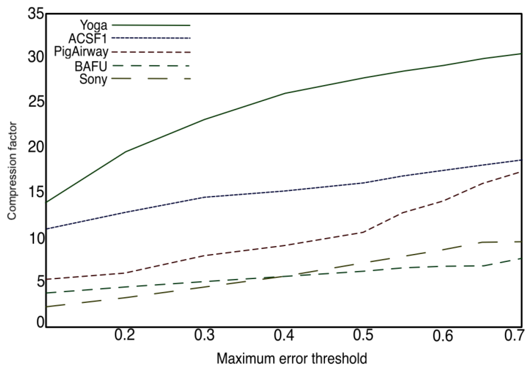
Figure 12 shows the relation between maximum deviation and MSE.

Compression ratio. Both the maximum deviation and polynomial degree parameters affect the compression ratio, as we can observe in Figure 13 where the compression factor ($\frac{1}{\rho}$) is reported for different values of parameters. The best compression ratios, $\rho = 0.06$ for REDD and $\rho = 0.02$ for Office, are achieved for the highest degree polynomials. As expected, also for increasing maximum deviation the compression factor is monotonically increasing.

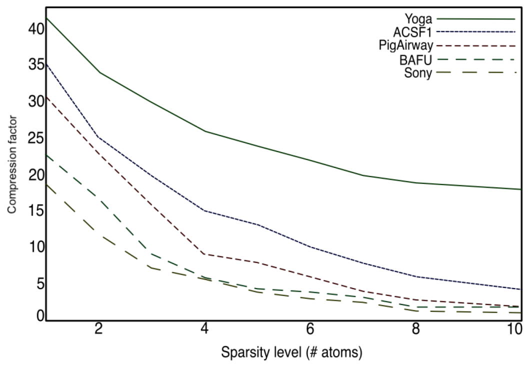
5.1.6 Chebyshev Polynomial Transform (CPT). Despite the absence of experimental results in the original article, some considerations can be done. The algorithm has three parameters:

— Block size;
— Maximum deviation;
— Number of quantization bits.

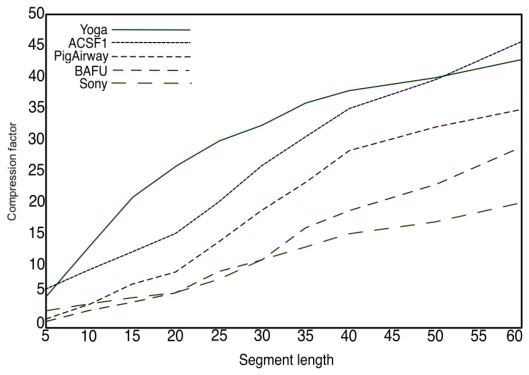
Similarly to PPA, in CPT it is possible to obtain higher compression ratios at the cost of higher MSE by increasing the maximum deviation parameter. The same holds for the block size: larger blocks correspond to better compression ratios and higher decompression errors [18]. Similarly, we can suppose that using fewer quantization bits would entail a less accurate decompression and a better compression ratio.



(a) Varying error threshold



(b) Varying dictionary atoms



(c) Varying segment length

Fig. 11. Compression factor ($\frac{1}{\rho}$) with varying parameters for CORAD [10].

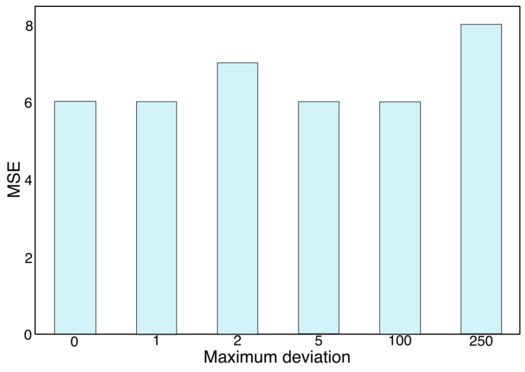
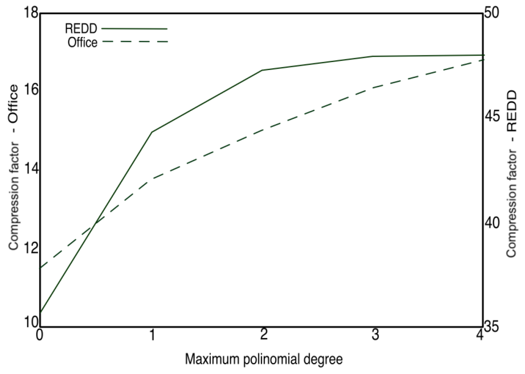
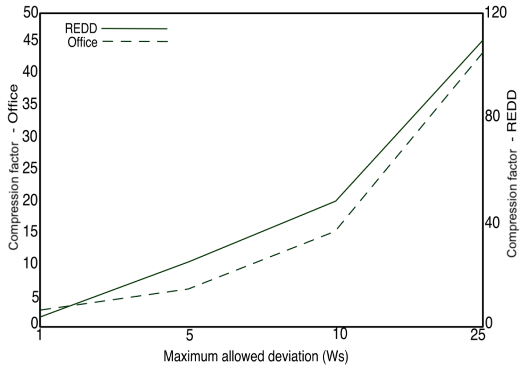


Fig. 12. Relation between MSE and maximum deviation for PPA [23].



(a) Varying maximum polynomial degree



(b) Varying maximum deviation

Fig. 13. PPA [14] compression factor ($\frac{1}{\rho}$) for varying parameters.

Table 4. DCT Experimental Results

	Compression ratio (%)	MSE
A5 temperature sensor	49.60	0.0009
	82.60	0.03
A5 vibration sensor	3.10	0.04
	7.10	0.10
	22.90	0.30
AISat temperature sensor	2.30	6.50
	5.20	92.80
	8.80	317.92

5.1.7 Discrete Wavelet Transform (DWT). As for CPT, experimental results are not provided, but also DWT performances follow the same rules as for PPA since the compression ratio and threshold errors are strictly correlated. The threshold error can be fixed *a priori* to have a higher compression ratio or a higher accuracy.

5.1.8 Discrete Fourier Transform (DFT). The authors tested the performance of the DFT algorithm using three types of signals: Block, Heavy Sine, and Mishmash. The compression ratio depends on the percentage of coefficients that are removed. The optimum threshold is derived by iterative tests, and it depends on the signal type. As stated by the authors, the results obtained by DWT are comparable to the ones obtained by DWT: the choice of the algorithm highly depends on the type of signal that has to be compressed [43].

5.1.9 Discrete Cosine Transform (DCT). The accuracy obtained by this algorithm depends on the chosen compression ratio. The authors used the ARIANE 5 (temperature sensor, and vibration sensor) and AISat temperature sensor for their experiments. The results show that the performances of this algorithm are not homogeneous, as they highly depend on the input dataset. Even if the input data seems to be similar, belonging to the same domain: the performances obtained on the ARIANE 5 and AISat temperature sensors are completely different, as shown in Table 4 [45].

5.1.10 Recurrent Neural Network Autoencoder (RNNA).

Accuracy. One of the parameters of the RNNA-based methods is the threshold for the maximally allowed deviation, whose value directly affects accuracy. For the experiments on UTS presented in [23] the authors report, considering the optimal value for deviation parameter, an RMSE value in the range $[0.02, 0.07]$ for different datasets. In the same experiments, RMSE is in the range $[0.02, 0.05]$ for MTS datasets. Thus, there is no significant difference between univariate and multivariate time series.

Compression ratio. Similarly, the maximally allowed deviation affects the compression ratio. According to the results reported by the authors, in this case, there is not a significant difference between univariate and multivariate time series: ρ is in the range $[0.01, 0.08]$ for the UTS dataset and in the range, $[0.31, 0.05]$ for the multivariate one.

5.1.11 Recurrent Convolutional Autoencoder (RCA). The experimental results reveal a slight improvement with respect to the RNNA model over the traffic flow dataset [49]: without the convolutional layer, the RMSE value is 10.37, while with the convolutional layer, the RMSE decreases to 8.17.

Since the original article is focused on the prediction of the next values only, the compression ratio is missing.

5.1.12 DZip. The authors tested the model with real and synthetic datasets and compared the results with the state-of-the-art compression algorithms, including RNNA. The results revealed

Table 5. DRH Compression Factors

	$^1/_\rho$
Indoor temperature	92.2
Outdoor temperature	91.17
Pressure	82.55
Light	83.53

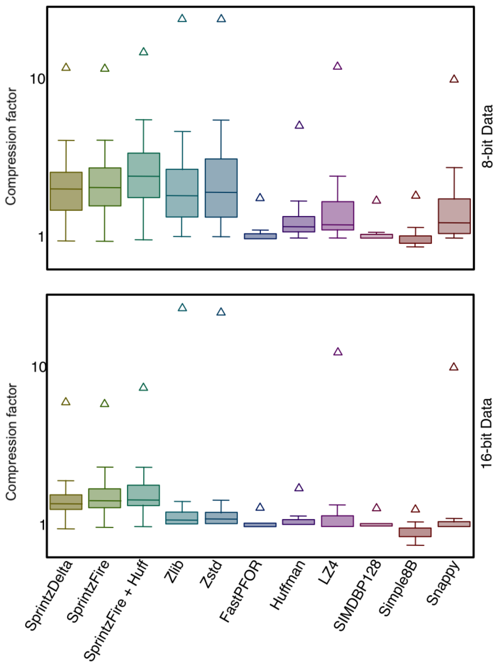


Fig. 14. Sprintz compression factors versus other lossless compression algorithms [29].

that it outperforms the RNNA autoencoder. The compression ratio depends on the vocabulary size, which is negatively affected by larger sizes. Another factor that influences this parameter is the size of the original data: the larger is the time series to be compressed, the better is the compression ratio [47].

5.1.13 Delta Encoding, Run-Length, and Huffman (DRH). DRH, as most of the following algorithms, is a lossless algorithm, thus the only relevant performance metric is the compression ratio.

Compression ratio. This parameter highly depends on the dataset: run-length algorithm combined with delta encoding achieves good results if the time series is composed of long segments that are always increasing or decreasing of the same value or constant. Experimental results are obtained on temperature, pressure, and light measures datasets. For this type of data, DRH appears to be appropriate, since the values are not highly variable and have a relatively long interval with constant values. The resulting compression factors are reported in Table 5 [28].

5.1.14 Sprintz.

Compression ratio. This algorithm is tested over the datasets of the UCR time series archive, which contains data coming from different domains. In the boxplots in Figure 14, compression factors are shown for 16 and 8-bit data and compared with other lossless compression algorithms.

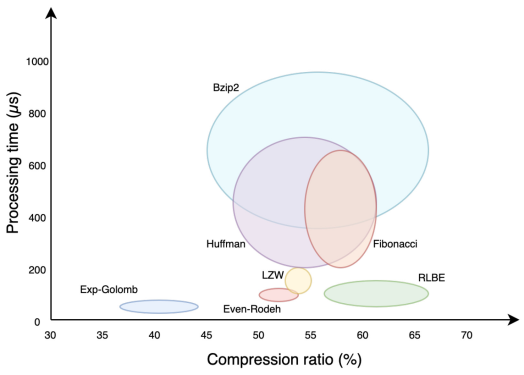


Fig. 15. RLBE compression ratio versus other lossless compression algorithms [30].

Table 6. Compression Factor Results for RAKE Versus other Lossless Compression Algorithms

	p							
Algorithm	0.002	0.005	0.01	0.05	0.1	0.15	0.2	0.25
OPT	48.0	22.0	12.4	3.5	2.1	1.6	1.4	1.2
RAKE	47.4	21.5	12.0	3.5	2.1	1.6	1.4	1.2
RAR	22.1	12.1	7.4	2.6	1.7	1.4	1.2	1.2
GZIP	21.4	11.8	7.4	2.6	1.8	1.4	1.3	1.2
BZIP2	26.0	14.2	8.7	2.6	1.7	1.3	1.2	1.1
PE	29.5	11.9	5.9	1.2	0.6	0.4	0.3	0.2
RLE	35.7	15.7	8.4	2.1	1.2	0.9	0.8	0.6

From this graph, it is possible to see that all algorithms achieve better (higher) compression factors on the 8-bit dataset. This could be due to the fact that 16-bit data are more likely to have higher differences between consecutive values.

Another interesting result is that the FIRE forecasting technique improves this compression, especially if combined with Huffman coding when compared with Delta coding [29].

5.1.15 *Run-Length Binary Encoding (RLBE).*

Compression ratio. The author of this algorithm in [30] report compression ratio and processing time of RLBE with respect to those of other algorithms as shown in Figure 15 where ellipses represent the inter-quartile range of compression ratios and processing times. RLBE achieves good compression ratios with a 10% variability and with low processing time.

5.1.16 *RAKE.*

Compression ratio. The authors of RAKE in [31] run experiments on sparse time series with different sparsity ratios p and different algorithms. The corresponding compression factors are reported in Table 6. We can notice that the compression factor is highly dependent on sparsity and lower sparsity values correspond to higher compression factors (better compression). Thus, the RAKE algorithm is more suitable for sparse time series.

5.1.17 Major Extrema Extractor (MEE). The compression ratio is one of the parameters of the MEE algorithm, thus the only relevant performance metric is the accuracy.

Accuracy. Even though MEE is a lossy compression algorithm, the authors have not provided any information about accuracy. Since the accuracy depends on the compression ratio, the authors

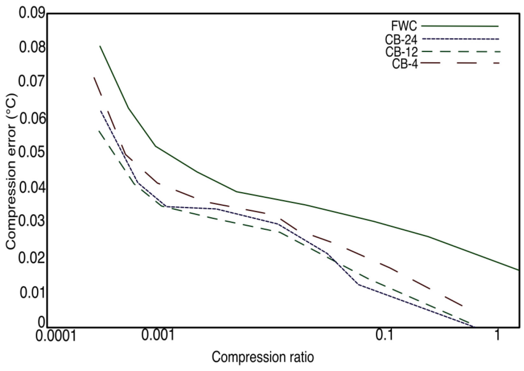


Fig. 16. Correlation between compression ratio and error on a temperature dataset [33]

recommend choosing a value that is not smaller than the percentage of non-extremal points, to obtain an accurate reconstruction of the compressed data [32].

5.1.18 Segment Merging (SM). Since this algorithm is used for visualization purposes, a first evaluation can be a qualitative observation. In Figure 9, we can see how a light sensor time series is compressed considering different time series segment lengths. For this algorithm, compression ratio and error are strictly correlated, as shown in Figure 16 where FWC and CB-*m* correspond to different versions of the algorithm and different parameters [33]. We can observe that compression ratio values close to 1 correspond to error values close to zero.

The dependence of compression ratio and error changes for different datasets. Knowing the function that correlates the compression ratio and error for a given dataset, it is possible to set the compression ratio that corresponds to the desired maximum compression error.

5.1.19 Continuous Hidden Markov Chain (CHMC). For this algorithm, the authors have not provided any quantitative evaluation for compression performances: their assessment is qualitative and mainly focused on the compression and reconstruction of humanoid robot motions after a learning phase [34].

5.2 Intergroup Comparison

After passing through the single techniques, we propose an intergroup comparison to address the general characteristics of each group.

5.2.1 Dictionary-based. The DB approaches include both lossy and lossless compression. The compression ratio depends on the atom's size in the dictionary and on the dataset's repetitiveness. For large datasets, the storage needed for the dictionary can be considered irrelevant, while for smaller datasets it can be inefficient. The training phase can be inefficient for databases with low repetitiveness.

5.2.2 Functional Approximation. FA includes only lossy techniques. The compression ratio can be set in advance, as the maximum reconstruction error. They work better with regular data with low fluctuations and peaks.

5.2.3 Autoencoders. Autoencoders include lossy and lossless techniques. The error threshold can be set in advance, while the compression ratio depends on the quality of the training phase and on the regularity of the dataset. If the autoencoder receives data with different seasonability and distribution with respect to the ones found in the training set, its prediction capacity becomes low. A weak point is the computational cost of the training set: recurrent neural networks are more expensive to be trained respect to classic neural network autoencoders and need a large amount of data in the training set.

5.2.4 Sequential Algorithms. The selected techniques in this group include lossless techniques only. The compression ratio depends on the data sparsity and regularity, especially for those that use delta and run-length encoding. They do not need a training phase and work well with low-computational resources.

5.2.5 Others. This group includes lossy techniques, and their peculiarities depend on the single technique.

6 CONCLUSIONS

The amount of time series data produced in several contexts requires specialized time series compression algorithms to store them efficiently. In this article, we provide an analysis of the most relevant ones, we propose a taxonomy to classify them, and report synoptically the accuracy and compression ratio published by their authors for different datasets. Even if this is a meta-analysis, it is informative for the reader about the suitability of a technique for specific contexts, for example, when a fixed compression ratio or accuracy is required, or for a dataset having particular characteristics.

The selection of the most appropriate compression technique highly depends on the time series characteristics, the constraints required for the compression, or the purpose.

Starting from the first case, time series can be characterized by their regularity (that means, having or not having fixed time intervals between two consecutive measurements), sparsity, fluctuations, and periodicity. Considering **regularity**, only techniques based on function approximation can be applied directly to the time series without the need for transformations. This is because that approach approximates a function passing through the given points, having in the x-axis time, so the time-distance between consecutive points can be arbitrary. All the other techniques assume the time-distance between two points to be fixed. To apply them, some transformations on the original time series are needed. Some techniques work better in input time series with high **sparsity**, as the ones proposed in Section 4.4. Another characteristic to consider is **fluctuations**. The lossy compression techniques presented in this survey cannot achieve good accuracy results in input time series with high non-regular fluctuations. This is caused by the fact that FA and autoencoder techniques tend to flatten peaks, while the ones based on dictionaries would need overly large dictionaries. On the other hand, FA and lossy autoencoders can be taken into consideration if it is sufficient to reconstruct the general trend of the original time series. Lastly, DB techniques work well with time series with high **periodicity**. In this scenario, a relatively small dictionary could be enough to represent the whole time series by combining atoms. Lossy autoencoders also improve their reconstruction accuracy, in input periodical time series.

The constraints required for compression include compression ratio, reconstruction accuracy, speed, and availability of a training dataset. As shown in Table 2, some techniques allow to specify a minimum **compression ratio** and a maximum **reconstruction accuracy** error. This could be necessary for some contexts in which space or accuracy are given as constraints. Moreover, compression **speed** can be relevant, in particular when hardware with low computation capacity is involved. In this context, SA can compress at high-speed, without needing powerful devices. Depending on the presence or not of a **training set**, all the techniques that need a training phase must be discarded.

The different purposes for time series compression include analysis, classification, and visualization. For time series **analysis**, FA and lossy autoencoders can help to reduce noise and outlayers, for those time series that have many fluctuations. Autoencoders can be used also to more deep analysis, as anomaly detections, starting from the compressed representation, as shown in [46]. The CHMC technique can also be used for this aim, since time series can be represented with a probability graph. Autoencoders can be used for **classification** too, since the autoencoder can be used to reduce the dimensionality of the input. Lastly, for **visualization** purposes, MEE and SM can be used to have different definitions of long time series.

APPENDIX

A ALGORITHMS

ALGORITHM 1. DB—Training

```
createDictionary(Stream S, Threshold th, int segmentLength) {
    Dictionary d;
    Segment s;
    while (S.isNotEmpty()) {
        s.append(S.read());
        if (s.length == segmentLength) {
            if (d.find(s, th)) {
                d.merge(s);
            } else {
                d.add(s);
            }
            s.empty();
        }
    }
    return d;
}
```

ALGORITHM 2. DB—Compression

```
compress(Stream S, Dictionary d, Threshold th, int segmentLength) {
    Segment s;
    while (S.isNotEmpty()) {
        s.append(S.read());
        if (s.length == segmentLength) {
            if (d.find(s, th)) {
                send(d.getIndex(s));
            } else {
                send(s);
            }
            s.empty();
        }
    }
}
```

ALGORITHM 3. A-LZSS algorithm

```
compress(Stream S, int minM, Dictionary d, int Ln, int Dn) {
    foreach(s in S) {
        I, L = d.longestMatch(s, Ln, Dn);
        if (L < minM) {
            send(getHuffCode(s));
        } else {
            send((I, L));
            s.skip(L);
        }
    }
}
```

ALGORITHM 4. PPA algorithm

```
compress(int ρ, Stream S, float ϵ) {
    Segment s = S.read();
    while (S.isNotEmpty()) {
        Polynomial p;
        int l = 0;
        foreach (k in [0 : ρ]) {
            float currErr;
            Polynomial currP;
            bool continueSearch = true;
            int i = 0;
            int currL;
            while (continueSearch && i < len(s)) {
                currErr, currP = approx(k, s.getPrefix(i));
                if (currErr < ϵ) {
                    i += 1;
                    currL = i;
                } else {
                    continueSearch = false;
                }
            }
            while (continueSearch && S.isNotEmpty()) {
                s.append(S.read());
                currErr, currP = approx(k, s);
                if (currErr < ϵ) {
                    currL = len(s);
                } else {
                    currL = len(s) - 1;
                    continueSearch = false;
                }
            }
            if (currErr < ϵ && currL > l) {
                p = currP;
                l = currL;
            }
        }
        if (len(s) > 0) {
            send(p, l);
            s.removePrefix(l);
        } else {
            throw "Error:␣exceeded␣error␣threshold␣value";
        }
    }
}
```

Where:

- S is the input time series;
- ρ is the maximum polynomial degree;
- ϵ is the error threshold.

ALGORITHM 5. RNN compression algorithm

```
compress(Stream S, float $\epsilon$, RAE a) {
    Segment s = Null;
    while (S.isNotEmpty()) {
        Segment aux = s;
        Element e = S.read();
        aux.append(e);
        if (getError(aux, a.decode(a.encode(aux)) < $\epsilon$) {
            s = aux
        } else {
            send(a.encode(s));
            s.empty();
            s.append(e);
        }
    }
}
```

ALGORITHM 6. Huffman Code compression

```
createDictionary(StreamPrefix S) {
    Tree T = new Tree();
    PriorityList L = createPriorityList(S);
    foreach (i in [0 : L.length − 1]) {
        Node n = new Node();
        Element el1 = L.extractMin();
        Element el2 = L.extractMin();
        n.frequency = el1.frequency + el2.frequency;
        T.addTree(n, (el1, el2));
        L.add(n);
    }
    return L.toDictionary();
}

compress(Stream S, int prefixLen) {
    StreamPrefix s = S.prefix(prefixLen);
    Dictionary D = createDictionary(s);
    CompressedRepresentation R = [];
    while (S.isNotEmpty()) {
        Element e = S.read();
        send(D[e]);
    }
}
```

ALGORITHM 7. DRH nodel level

```
compress(Stream S, float Q) {
 Element lastValue = null;
 Element lastDelta = null;
 int counter = 0;
 foreach (s in S) {
   if (last != null && lastDelta != null) {
      float delta = 0;
      if (Q > 1) {
         delta = (int)(lastValue − s) / Q;
      } else {
         delta = lastValue − s;
      }
      lastValue = s;
      if (delta == lastDelta) {
         counter += 1;
      } else {
         encoded = huffmanEncode(lastDelta);
         send((encoded, counter));
         lastDelta = delta;
         counter = 0;
      }
   } else {
      lastValue = s;
      lastDelta = 0;
   }
 }
}
```

REFERENCES

[1] P. Asghari, A. M. Rahmani, and H. Javadi. 2019. Internet of things applications: A systematic review. *Computer Networks* 148, (2019), 241–261.

[2] J. Ronkainen and A. Iivari. 2015. Designing a data management pipeline for pervasive sensor communication systems. *Procedia Computer Science* 56, (2015), 183–188.

[3] S. K. Jensen, T. B. Pedersen, and C. Thomsen. 2017. Time series management systems: A survey. *IEEE Transactions on Knowledge and Data Engineering* 29, 11 (2017), 2581–2600.

[4] D. Salomon. 2007. *Data Compression: The Complete Reference* (4th. ed.). Springer.

[5] J. Wolff. 2003. Wolff - 1990 - Simplicity and Power - Some Unifying Ideas in Computing. *The Computer Journal* 33, 6 (1990), 518–534.

[6] K. Sayood. 2006. *Introduction to Data Compression* (3rd. ed.). Morgan Kaufmann.

[7] A. Marascu, P. Pompey, E. Bouillet, M. Wurst, O. Verscheure, M. Grund, and P. Cudre-Mauroux. 2014. TRISTAN: Real-time analytics on massive time series using sparse dictionary compression. In *Proceedings of the 2014 IEEE International Conference on Big Data.* 291–300.

[8] J. Mairal, F. Bach, J. Ponce, and G. Sapiro. 2009. Online learning for matrix factorization and sparse coding. *Journal of Machine Learning Research* 11, (2009), 19–60.

[9] S. Mallat. 1993. Zhifeng Zhang: Matching pursuits with time-frequency dictionaries. *IEEE Transactions on Signal Processing* 41, 12 (1993), 3397–3415.

[10] A. Khelifati, M. Khayati, and P. Cudre-Mauroux. 2019. Corad: Correlation-aware compression of massive time series using sparse dictionary coding. In *Proceedings of the 2019 IEEE International Conference on Big Data.* 2289–2298.

[11] J. A. Storer and T. G. Szymanski. 1982. Data compression via textual substitution. *Journal of the ACM* 29, 4 (1982), 928–951.

[12] J. Pope, A. Vafeas, A. Elsts, G. Oikonomou, R. Piechocki, and I. Craddock. 2018. An accelerometer lossless compression algorithm and energy analysis for IoT devices. In *Proceedings of the 2018 IEEE Wireless Communications and Networking Conference Workshops.* 396–401.

[13] T. L. Le and M. Vo. 2018. Lossless data compression algorithm to save energy in wireless sensor network. In *Proceedings of the 2018 4th International Conference on Green Technology and Sustainable Development*. 597–600.

[14] F. Eichinger, P. Efros, S. Karnouskos, and K. Böhm. 2015. A time-series compression technique and its application to the smart grid. *The VLDB Journal* 24, 2 (2015), 193–218.

[15] I. Lazaridisand and S. Mehrotra. 2003. Capturing sensor-generated time series with quality guarantees. In *Proceedings of the 19th International Conference on Data Engineering*. 429–440.

[16] M. Dalai and R. Leonardi. 2006. Approximations of one-dimensional digital signals under the$l^{i}nfty$norm. *IEEE Transactions on Signal Processing* 54, 8 (2006), 3111–3124.

[17] R. Seidel. 1991. Small-dimensional linear programming and convex hulls made easy. *Discrete and Computational Geometry* 6, 3 (1991), 423–434.

[18] S. E. I. Hawkins and E. H. Darlington. 2012. Algorithm for compressing time-series data. *NASA Tech Briefs* 36, 6 (2012), 7–8. Retrieved from https://ntrs.nasa.gov/search.jsp?R=20120010460.

[19] X. Lv and S. Shen. 2017. On chebyshev polynomials and their applications. *Advances in Difference Equations* 1, 343 (2017), 1–6.

[20] M. Abo-Zahhad. 2011. ECG signal compression using discrete wavelet transform. In *Proceedings of the Discrete Wavelet Transforms - Theory and Applications*. J. T. Olkkonen (Ed.), InTech.

[21] I. Goodfellow, Y. Bengio, and A. Courville. 2016. *Deep Learning*. MIT Press. Retrieved from http://www.deeplearningbook.org.

[22] A. Sherstinsky. 2020. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *Physica D: Nonlinear Phenomena* 404, 132306 (2020).

[23] D. Hsu. Time series compression based on adaptive piecewise recurrent autoencoder. arXiv:1707.07961. Retrieved from https://arxiv.org/abs/1707.07961.

[24] S. Hochreiter and J. Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (1997), 1735–80.

[25] T. Suel. 2018. Delta compression techniques. In *Proceedings of the Encyclopedia of Big Data Technologies*. S. Sakr and A. Zomaya (Eds.), Springer International Publishing.

[26] S. M. Hardi, B. Angga, M. S. Lydia, I. Jaya, and J. T. Tarigan. 2019. Comparative analysis run-length encoding algorithm and fibonacci code algorithm on image compression. *Journal of Physics: Conference Series* 1235, 012107 (2019).

[27] J. Walder, M. Krátký, and J. Platos. 2010. Fast fibonacci encoding algorithm. In *Proceedings of the Dateso 2010 Annual International Workshop on DAtabases, TExts, Specifications and Objects, Stedronin-Plazy, Czech Republic*. J. Pokorný, V. Snásel, and K. Richta (Eds.), April 21–23, 2010, 72–83. CEUR-WS.org. Retrieved from http://ceur-ws.org/Vol-567/paper14.pdf.

[28] H. S. Mogahed and A. G. Yakunin. 2018. Development of a lossless data compression algorithm for multichannel environmental monitoring systems. In *Proceedings of the 2018 XIV International Scientific-Technical Conference on Actual Problems of Electronics Instrument Engineering*. 483–486.

[29] D. Blalock, S. Madden, and J. Guttag. 2018. Sprintz: Time series compression for the internet of things. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 3 (2018), 1–23.

[30] J. Spiegel, P. Wira, and G. Hermann. 2018. A comparative experimental study of lossless compression algorithms for enhancing energy efficiency in smart meters. In *Proceedings of the 2018 IEEE 16th International Conference on Industrial Informatics*. 447–452.

[31] G. Campobello, A. Segreto, S. Zanafi, and S. Serrano. 2017. RAKE: A simple and efficient lossless compression algorithm for the Internet of Things. In *Proceedings of the 2017 25th European Signal Processing Conference*. 2581–2585.

[32] E. Fink and H. S. Gandhi. 2011. Compression of time series by extracting major extrema. *Journal of Experimental and Theoretical Artificial Intelligence* 23, 2 (2011), 255–270.

[33] R. Goldstein, M. Glueck, and A. Khan. 2011. Real-time compression of time series building performance data. In *Proceedings of the IBPSA-AIRAH Building Simulation Conference*.

[34] T. Inamura, H. Tanie, and Y. Nakamura. 2003. Keyframe compression and decompression for time series data based on the continuous hidden Markov model. In *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 1487–1492.

[35] Y. Steve, E. Gunnar, G. Mark, H. Thomas, K. Dan, L. Xunying Andrew, M. Gareth, O. Julian, O. Dave, V. Valtcho, and W. Phil. 2006. *The HTK Book*. Microsoft.

[36] M. Wojnarski, P. Gora, M. Szczuka, H. S. Nguyen, J. Swietlicka, and D. Zeinalipour. 2010. Ieee icdm 2010 contest: Tomtom traffic prediction for intelligent gps navigation. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*. 1372–1376.

[37] K. Smith, D. Depolo, J. Torrisi, N. Edwards, G. Biasi, and D. Slater. 2008. The 2008 mw 6.0 wells, nevada earthquake sequence. *AGU Fall Meeting Abstracts* (2008).

[38] M. Shoaib, S. Bosch, O. Incel, H. Scholten, and P. Havinga. 2014. Fusion of smartphone motion sensors for physical activity recognition. *Sensors (Basel, Switzerland)* 14, 6 (2014), 10146–10176.

[39] A. Reiss and D. Stricker. 2011. Towards global aerobic activity monitoring. In *Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments. Association for Computing Machinery.*

[40] M. Krawczak and G. Szkatułá. 2014. An approach to dimensionality reduction in time series. In *Proceedings of the 2014 Information Sciences.* 15–36.

[41] M. Ali, R. Borgo, and M. W. Jones. 2021. Concurrent time-series selections using deep learning and dimension reduction. In *Proceedings of the 2021 Knowledge-based Systems.*

[42] J. Cooley and J. W. Tukey. 1965. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation* 19 (1965), 297–301.

[43] M. Ali, R. Borgo, and M. W. Jones. 2011. Wavelet transform and fast fourier transform for signal compression: A comparative study. In *Proceedings of the International Conference on Electronic Devices, Systems, and Applications.* 280–285.

[44] Y. M. Huang and J. L. Wu. 1999. A refined fast 2-d discrete cosine transform algorithm. In *Proceedings of the IEEE Transactions on Signal Processing.* 904–907.

[45] J. G. Mess, R. Schmidt, G. Fey, and F. Dannemann. On the compression of spacecraft housekeeping data using discrete cosine transforms. In *International Workshop on Tracking, Telemetry and Command Systems for Space Applications (TTC'16).* 1–8.

[46] O. I. Provotar, Y. M. Linder, G. Fey, and M. M. Veres. 2019. Unsupervised anomaly detection in time series using LSTM-based autoencoders. In *Proceedings of the IEEE International Conference on Advanced Trends in Information Theory.*

[47] M. Goyal, K. Tatwawadi, S. Chandak, and I. Ochoa. 2021. DZip: improved general-purpose loss less compression based on novel neural network modeling. In *Proceedings of the 2021 Data Compression Conference.* 153–162. Retrieved from https://ieeexplore.ieee.org/document/9418692/.

[48] Kaushik Chakrabarti, Eamonn Keogh, Sharad Mehrotra, and Michael Pazzani. 2002. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM Trans. Database Syst.* 27, 2 (June 2002), 188–228. https://doi.org/10.1145/568518.568520

[49] X. Zhao, X. Han, W. Su, and Z. Yan. 2019. Time series prediction method based on convolutional autoencoder and LSTM. In *Proceedings of the 2019 Chinese Automation Congress.* 5790–5793. Retrieved from https://ieeexplore.ieee.org/document/8996842/.

[50] J. Lin, E. Keogh, S. Lonardi, and B. Chiu. 2003. A symbolic representation of time series, with implications for streaming algorithms. In *Proceedings of the 8th ACM SIGMOD Workshop On Research Issues in Data Mining and Knowledge Discovery.* 2–11. DOI: https://doi.org/10.1145/882082.882086

Received 22 September 2021; revised 5 July 2022; accepted 21 August 2022