Universidad de la República

Tesis de Maestría

Coding of Multichannel Signals with Irregular Sampling

Autor:
Pablo Cerveñansky

Supervisores:
Álvaro Martín
Gadiel Seroussi

Núcleo de Teoría de la Información
Facultad de Ingeniería

September 3, 2020

Cambios de la versión anterior (13/7/2020) a esta versión (27/7/2020)

Chapter 1: Datasets

- Agregué una introducción especificando lo que va a ir en cada section.
- En la Section 1.1 agregué una tabla con los datasets. Incluye la columna Dataset Characteristic, que también se muestra en la Tabla 3.1 del Chapter 3. Creo que está bueno que esa misma información se presente de la misma manera cuando se habla de los datasets.

Chapter 2: Algorithms

- Agregué una introducción especificando lo que va a ir en cada section.
- En la Section 2.1 agregué una tabla con los algoritmos. Creo que ayuda a entender el tema de los distintos conjuntos de algoritmos y variantes que se definen en el Chapter 3.
- En la segunda página de la Section 2.1 agregué el pseudocódigo genérico para todos los algoritmos de modelo constante y lineal. El algoritmo GAMPS también considera la correlación entre columnas, así que el pseudocódigo es diferente a los demás (todavía no lo hice, lo pensaba poner en la section de GAMPS).
- En la Section 2.3 agregué los pseudocódigos de codificación y decodificación del algoritmo Base.
- En la Section 2.4 agregué los pseudocódigos de codificación del algoritmo PCA, variantes con y sin máscara. Esta semana termino los de decodificación y sigo con APCA, CA, etc.

Chapter 3: Experimental Results

- Introducción: Hice las correcciones marcadas y la separé en párrafos. Marqué un par de dudas con negrita.
- Al principio de la Section 3.1 agregué varias definiciones en un párrafo. No estoy seguro si debería hacer definiciones formales o si está bien que estén todas juntas en un párrafo.
- En las leyendas de las figuras 3.3 y 3.4 está bien poner LOWS y OWS, o debería poner w^*_{global}y w^*_{local}?
- En todas las figuras después de la Section 3.2 creo que debería poner PCA$_M$, APCA$_M$, etc. en vez de PCA, APCA, etc.
- Agregué la Section 3.5 con conclusiones. Algunas cosas quedaron similares a la introducción, pero intenté de que nunca hubiera dos frases iguales.
- Agregué la Section 3.6 con un punteo de ideas de trabajo futuro. Conclusions and Future Work debería ser un capítulo aparte?

Chapter 1

Datasets

In this chapter we present all of the datasets that were compressed in our experimental work, which is described in Chapter 3. In Section 1.1 we give an overview of the different datasets, describing their characteristics in terms of the amount of gaps, and the number of files and data types that each have. In the remaining sections we present in detail each of the datasets.

1.1 Overview

TODO:

- Explain why / how every dataset was transformed into a common format.
- Show example csv (describe header, column names, data rows, etc.)

Dataset	**Dataset Characterstic**	**#Files**	**#Types**	**Data Types**
IRKIS [1]	Many gaps	7	1	VWC
SST [2]	Many gaps	3	1	SST
ADCP [2]	Many gaps	3	1	Vel
Solar [3]	Many gaps	4	3	GHI, DNI, DHI
ElNino [4]	Many gaps	1	7	Lat, Long, Zonal Winds, Merid. Winds, Humidity, Air Temp., SST
Hail [5]	No gaps	1	3	Lat, Long, Size
Tornado [5]	No gaps	1	2	Lat, Long
Wind [5]	No gaps	1	3	Lat, Long, Speed

Table 1.1: Datasets overview. The second column indicates the characteristic of each dataset, in terms of the amount of gaps. The third column shows the number of files. The fourth and fifth columns show the number of data types and their names, respectively.

1.2 IRKIS

1.3 ADCP

1.4 ElNino

1.5 Solar

1.6 Hail

1.7 Tornado

1.8 Wind

Chapter 2

Algorithms

In this chapter we present all of the coding algorithms implemented in the project. In Section 2.1 we give an overview of the different algorithms, describing their features, parameters and masking variants. In Section 2.2 we provide some implementation details, including the pseudocodes for the coding and decoding subroutines that are common for every algorithm. We also describe the implementation of the gap encoding in the masking variant, which applies the KT estimator and arithmetic coding. In Section 2.3 we present algorithm Base, which is a trivial algorithm that serves as a base ground for the compression performance comparison developed in Chapter 3. In the remaining sections we present each of the coding algorithms in detail, including implementation specifics with the coding and decoding pseudocode for each of their variants.

TODO:

- Llamar column.integer entries al iterar en los algoritmos con mask mode y all..entries en otro caso.
- C++: especificar estructura del proyecto? Mencionar los nombres de las clases?

2.1 Introduction

There exists literature analyzing the performance of the state-of-the-art algorithms used for sensor data compression [6, 7]. In their original form, these algorithms assume that the signals have regular sampling and that there are no gaps in the data. However, it is often the case that this is not true for real-world datasets. For example, all of the datasets presented in Chapter 1 consist of signals that have either one or both of these characteristics. We selected a number of the state-of-the-art algorithms and implemented them in a way such that they are also able to encode signals with these characteristics.

The state-of-the-art algorithms follow a model-based compression approach: they represent the signal data using conventional approximation models whose parameters are then stored as compressed data. These model-based techniques manage to compress the data by exploiting their temporal and (in some cases) spatial correlation. They are popular not only due to their efficient compression performance, but also because they offer many benefits to data processing. For example, they can be used for inferring uncertain sensor readings, detecting outliers, indexing, etc. [6]. The model-based techniques are classified into four different categories, depending on their model type: *Constant models* encode signals using constant functions, *Linear models* use linear functions, *Nonlinear models* use complex nonlinear functions, and *Correlation models* simultaneously encode multiple signals while reducing redundant information. Algorithms in the last category are the only ones that, besides exploiting temporal correlation in the data, also exploit its spatial correlation.

In Table 2.1 we outline the features of every one of the algorithms that were implemented in the project. For each algorithm, the second and third columns indicate whether they support lossless and/or lossy compression, the fourth column shows its model category, the fifth and sixth columns indicate if the masking (*M*) and/or non-masking (*NM*) variants are valid, and the last column specifies if the window size parameter (w) is supported.

Algorithm	**Lossless**	**Lossy**	**Model**	***M***	***NM***	w
Base	x		Constant		x	
PCA [8]	x	x	Constant	x	x	x
APCA [9]	x	x	Constant	x	x	x
CA [10]	x	x	Linear	x	x	x
PWLH [11]/ PWLHInt	x	x	Linear	x	x	x
SF [12]	x	x	Linear	x		
FR [13]	x	x	Linear	x		x
GAMPS [14]/ GAMPSLimit	x	x	Correlation	x	x	x

Table 2.1: Coding algorithms overview. For each algorithm we show whether it supports lossless and/or lossy compression (second and third columns), its model category (fourth column), whether the masking (*M*) and/or non-masking (*NM*) variants are valid (fifth and sixth columns), and whether the window size parameter (w) is supported (last column).

Algorithm Base is a trivial lossless algorithm that is used as a base ground for comparing the performance of the remaining algorithms, all of which support both lossless and lossy encoding while guaranteeing a maximum point-by-point error between the decoded and the original files. This maximum error threshold is specified to the coding routine via the ϵ parameter. We implemented algorithms PWLHInt and GAMPSLimit by modifying the code of algorithms PWLH and GAMPS, respectively, and this is why they appear in the same row.

We implemented at least one algorithm for each of the four categories defined above, except for the Nonlinear model. We decided not to implement algorithms in this category since they do not guarantee a maximum point-by-point error, and they yield poor compression performances [6].

We implemented two variants, masking (*M*) and non-masking (*NM*), which differ in the way they handle the encoding of the gaps in the data. The *M* variant of an algorithm first encodes the position of all the gaps and then proceeds to encode the data values. On the other hand, the *NM* variant encodes the position of the gaps and the data values simultaneously. Implementation details are presented in the remaining sections of the current chapter. It should be pointed out that, for every algorithm, the gaps in the decoded file always must match the gaps in the original file, regardless of the value of the ϵ parameter, which is only considered when encoding the data values. We didn't find an efficient way of implementing the *NM* variant for the algorithms SF and FR, so they only support the *M* variant. In Section 3.2 we compare the compression performance of both variants, *M* and *NM*, for every algorithm that supports them.

Most of the algorithms support a window size parameter, which we label w. This parameter is passed to the coding routine, and it establishes the size of the blocks in which the data are processed and encoded. In algorithm PCA, parameter w defines a *fixed block size*, while in the rest of the algorithms it defines the *maximum block size*. More details on how the data are processed in blocks can be found in the pseudocodes for the algorithms presented in this chapter.

2.2 Implementation details

All of the algorithms are implemented in C++. In every case, we followed the original designs specified in the respective publications and added the new features described in Section 2.1. Algorithms PWLH [11], SF [12] and GAMPS [14] apply certain complex mathematical functions, so when implementing these algorithms we decided to reuse part of the code from the framework linked in [6]. The remaining algorithms [8–10, 13] were implemented entirely on our own.

Figures 2.1 and 2.2 show the coding and decoding pseudocodes, respectively, for every one of the Constant and Linear model algorithms implemented in the project (recall this information from Table 2.1). Both pseudocodes have the same length, and lines with the same number perform opposite actions. Constant and Linear model algorithms only exploit the temporal correlation in the data, thus they iterate through the data columns and encode them independently. Since Correlation models also exploit the spatial correlation (i.e. the data columns are *not* not encoded independently), the pseudocodes for algorithms GAMPS and GAMPSLimit are different to the ones shown in this section, and we present them in Section 2.10.

In Figure 2.1, the inputs for the coding routine are a csv data file with the same format as the datasets presented in Chapter 1, an integer value (*algo_key*) that uniquely describes the selected algorithm, a key (v) that describes its variant (either *M* or *NM*), and the maximum error threshold (ϵ) and window size (w) parameters. The output is a binary file, which represents the input file encoded by the selected algorithm with the specified variant and parameters.

```
input  : in: csv data file to be coded
         algo_key: integer value for the selected algorithm
         v: key for the variant
         ϵ: maximum error threshold
         w: window size
output: out: binary file encoded with the selected algorithm
 1 out = new_binary_file()
 2 out.code_base_2(algo_key, 8)
 3 algo = get_algorithm(algo_key)
 4 if algo.has_window_param? then
 5 |  out.code_base_2(w − 1, 8)
 6 end
 7 out.code_header(in.header)
 8 out.code_base_2(in.count_data_rows(), 24)
 9 out.code_ts_column(in.ts_column)
10 if v == M then
11 |  out.code_gaps(in)
12 |  foreach column in in.data_columns do
13 |  |  algo.code_column_M(column, out, ϵ, w, in.ts_column)
14 |  end
15 else
16 |  foreach column in in.data_columns do
17 |  |  algo.code_column_NM(column, out, ϵ, w, in.ts_column)
18 |  end
19 end
20 out.close_file()
```

FIGURE 2.1: Coding pseudocode for the Constant and Linear model algorithms.

The aforementioned binary file and the variant key are the only inputs for the decoding routine, in Figure 2.2. This routine also requires parameters *algo_key* and w, but they are implicit inputs since they are already encoded in the binary file (lines 2 and 5). The output is a csv data file with the same headers as the original file (line 7), where the maximum absolute difference between any pair of decoded and original data values is equal to ϵ, and the position of the data gaps is the same as in the original file.

```
input  : in: coded binary file
         v: key for the variant
output: out: decoded csv data file
1  out = new_csv_file()
2  algo_key = in.decode_base_2(8)
3  algo = get_algorithm(algo_key)
4  if algo.has_window_param? then
5  |  w = in.decode_base_2(8) + 1
6  end
7  out.decode_header(in)
8  count_data_rows = in.decode_base_2(24)
9  ts_column = out.decode_ts_column(in)
10 if v == M then
11 |  out.decode_gaps(in)
12 |  foreach column in out.data_columns do
13 |  |  algo.decode_column_M(column, out, w, ts_column)
14 |  end
15 else
16 |  foreach column in out.data_columns do
17 |  |  algo.decode_column_NM(column, out, w, ts_column)
18 |  end
19 end
20 out.close_file()
```

Figure 2.2: Decoding pseudocode for the Constant and Linear model algorithms.

The timestamp column, which is comprised of integers, is the first column in every csv data file, and it's also the first column to be encoded (line 9). This is done using a lossless schema, in which every integer is coded independently and using a fixed number of bits. It is worth mentioning that this coding schema could be further optimized. However, our project is focused in studying the compression of the data columns (i.e. the rest of the columns in the data file), since these are the only ones that could potentially have gaps.

When $v = M$, the masking variant of the algorithm is executed. First, the positions of the gaps in all of the data columns are encoded (line 11); implementation details are shown in Subsection 2.2.1. Then, the code_column_*M* subroutine is executed for every data column (lines 12-14). This subroutine has a different implementation depending on the specific algorithm, and the respective pseudocodes, both for the coder and the decoder, are presented in the following sections. Notice that these subroutines only encode the integer values in the data columns, since the positions of the gaps were already encoded in the previous step.

On the other hand, when $v = NM$, the non-masking variant of the algorithm is executed. In this case, the code_column_*NM* subroutine is executed for every data column (lines 16-18). Again, this subroutine is different for each specific algorithm, and we present the respective pseudocodes, for the coder and the decoder, in the following sections. Note that these subroutines must encode both the integer values *and the position of the gaps* in the data columns.

Before finishing this section, it should be pointed out that some of the implementation details were omitted in the pseudocodes for clarity. For instance, the key for the variant is not actually passed as an argument to the coding and decoding executables. Instead, we used a C++ macro and compiled two different executable files, one for each variant. This approach is useful both for optimizing the code and for making it more readable. Also, the coding and decoding subroutines (for both variants) do not always require all the arguments, and which ones are expected varies depending on the specific algorithm. In particular, the timestamp column is only used by the Linear model algorithms, and whether arguments ϵ and w are required depends on the algorithm features, which are outlined in Table 2.1. Besides, the actual implementation allows to set different maximum error thresholds to encode different data columns, so ϵ is actually a vector with a maximum error threshold parameter for each column in the input csv.

2.2.1 Gap Encoding in the Masking Variant

As we recall from previous sections, the masking variant of an algorithm first encodes the position of all the gaps in the data using a lossless schema, and afterwards it encodes the data values, using a lossless or lossy schema based on the value of the ϵ parameter. These two actions correspond to lines 11, and 12-14, respectively, of the coding and decoding pseudocodes presented in Figures 2.1 and 2.2.

We decided to encode the position of the data gaps using a general purpose lossless algorithm called Arithmetic Coding (AC) [15, 16]. This algorithm sequentially encodes the symbols produced by a source, and it becomes more efficient the closer its model distribution is to the the empirical distribution of the source [17]. In our case, the source is binary, with “0” denoting the presence of an integer data value, and “1” denoting the presence of a gap. As we recall from Chapter 1, the positions of the gaps follow different patterns for different datasets, but in general the gaps occur in bursts, and the amount of gaps is considerably less than the amount of data values. With this in mind, we thought it fitting to model the source distribution through a first-order Markov process with the Krichevsky–Trofimov estimator [18], which we define next in its binary alphabet form.

Definition 2.2.1. Given a source π with alphabet $A = \{0, 1\}$, and a string s generated by π, with $|s_0|$ zeroes and $|s_1|$ ones, the *Krichevsky–Trofimov estimator (KT)* assigns the following estimates $p_i(s)$ to the probability of each symbol $i \in A$:

$$p_0(s) = \frac{|s_0| + 1/2}{|s_0| + |s_1| + 1}, \qquad p_1(s) = \frac{|s_1| + 1/2}{|s_0| + |s_1| + 1}. \tag{2.1}$$

The first-order Markov process has two states, S_0 and S_1, where the current state is S_i iff the last read symbol is $i \in A$. We set S_1 as the initial state. In Figure 2.3 we display the diagram for this Markov process. To calculate the probabilities defined by (2.1) we need to keep a pair of counters (one for counting zeroes and another one for counting ones) *in each state*.

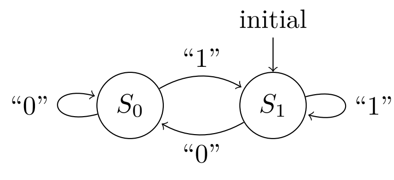


FIGURE 2.3: Markov process diagram

The version of the AC algorithm we used in our project is the CACM87 implementation [19, 20]. It is written in C and it is one of the most standard implementations. One of its advantages is that it allows to effortlessly set a custom model for the source. However, we had to overcome a minor obstacle to make it work within our scheme. In the CACM87 implementation, the coder closes the encoded file after it has encoded the last symbol. This implies that the decoder recognizes that there are no more symbols left to decode once it reads the last byte of the encoded file. But this is not the case in our masking variant scheme, since after the AC coder has encoded the position of all the gaps in the data, our coding algorithm still has to encode all the data values before closing the encoded file. The problem materialized in the decoding process, because after the AC decoder had decoded the last byte corresponding to the position of the gaps (i.e. the last byte encoded by the AC coder), our decoding algorithm would occasionally continue processing bytes corresponding to the encoded data values, which naturally resulted in an error. The solution we found was to flush the current byte in the stream, before and after executing the AC algorithm, both in the coding and the decoding routines.

It is worth mentioning that we also experimented using Golomb coding [21] to encode the position of the gaps in the masking variant. We did this for every dataset and compared the results with those obtained by the AC algorithm. In most of the cases, the AC algorithm outperforms Golomb coding, and the few cases in which Golomb coding obtains a better compression, the difference is relatively small. Therefore, we decided to make AC the default algorithm for encoding the gaps in the masking variant.

2.3 Algorithm Base

Algorithm Base is a trivial lossless algorithm that serves as a base ground for comparing the performance of the rest of the algorithms. In particular, we reference it when defining the compression ratio metric (see Definition 3.1.4), which we use to assess the compression performance of an algorithm in Chapter 3.

Figures 2.4 and 2.5 show the pseudocode for the code_column_*NM* and decode_column_*NM* subroutines, respectively, for algorithm Base. The coding subroutine iterates through every entry in the column of the csv data file, which is one of the inputs. Since algorithm Base only supports the *NM* variant, these entries can be either the character "N", which represents a gap in the data (line 3), or an integer value representing an actual data measurement (line 5). Every column entry is encoded independently and using a fixed number of bits, namely *column*.total_bits (line 7), which varies *for each data type of each dataset*. In every case, the number of bits used for encoding a data value ultimately depends on the range and accuracy of the instrument that was used for measuring and storing the data. Therefore, without loss of generality, we can assume that *column*.total_bits is known by both the coding and decoding subroutines. The same can be said regarding *column*.no_data_int, a special integer reserved for encoding a gap, and *column*.offset, an offset that is added to the original value of a column entry to obtain a non-negative integer that can be encoded in base 2.

```
input  : column: column of the csv data file to be coded
         out: binary file encoded with algorithm Base
1 foreach entry in column.entries do
2   if entry == "N" then
3     value = column.no_data_int
4   else
5     value = entry + column.offset
6   end
7   out.code_base_2(value, column.total_bits)
8 end
```

FIGURE 2.4: Base.code_column_*NM* pseudocode.

The decoding subroutine keeps running until every entry in the column has been decoded. For decoding an entry, the first step is reading a fixed number of bits from the input binary file (line 2). Depending on the value that is read, either the character "N" (line 4) or an offset integer (line 6) is written into the decoded csv data file.

```
input  : column: column encoded with algorithm Base
         out: decoded csv data file
1 foreach entry in column.entries do
2   value = in.decode_base_2(column.total_bits)
3   if value == column.no_data_int then
4     out.write_string("N")
5   else
6     out.write_string(value − column.offset)
7   end
8 end
```

FIGURE 2.5: Base.decode_column_*NM* pseudocode.

In the remaining sections of this chapter, with the purpose of simplifying the pseudocodes and the descriptions of the algorithms, we always omit the offset operations. Yet the related comments made in this section apply to every algorithm.

2.4 Algorithm PCA

Algorithm PCA [8], also known as Piecewise Constant Approximation, supports lossless and lossy compression, with both variants (M and NM), and it has a window size parameter (w) that establishes a fixed block size in which the data are processed and encoded. It is a Constant model algorithm, so it encodes signals using constant functions.

Figure 2.6 shows the pseudocode for the code_column_M subroutine for algorithm PCA. We observe that the data entries are added to the window (line 3) until its size is equal to w (i.e. the condition in line 4 becomes false). When that happens, the window is coded... TODO: continue

```
input  : column: column of the csv data file to be coded
         out: binary file encoded with algorithm PCA
         ϵ: maximum error threshold
         w: fixed window size
1  win = new_window()
2  foreach entry in column.entries do
3      win.push(entry)
4      if win.size < w then
5          continue
6      end
7      if |win.max − win.min| ≤ 2 ∗ ϵ then
8          out.code_bit(0)
9          value = (win.min + win.max)/2
10         out.code_base_2(value, column.total_bits)
11     else
12         out.code_bit(1)
13         foreach win_val in win.values do
14             out.code_base_2(win_val, column.total_bits)
15         end
16     end
17     win = new_window()
18 end
```

FIGURE 2.6: PCA.code_column_M pseudocode.

The pseudocode for the decode_column_M subroutine for algorithm PCA is shown in Figure 2.7. TODO: continue...

```
input : column: column encoded with algorithm PCA
        out: decoded csv data file
        w: fixed window size
foreach entry in column.entries do
    bit = in.decode_bit()
    if bit == 0 then
        value = in.decode_base_2(column.total_bits)
        repeat w times
            out.write_string(value)
        end
    else
        repeat w times
            value = in.decode_base_2(column.total_bits)
            out.write_string(value)
        end
    end
end
```

FIGURE 2.7: PCA.decode_column_M pseudocode.

TODO: describe example in Figure 2.8.

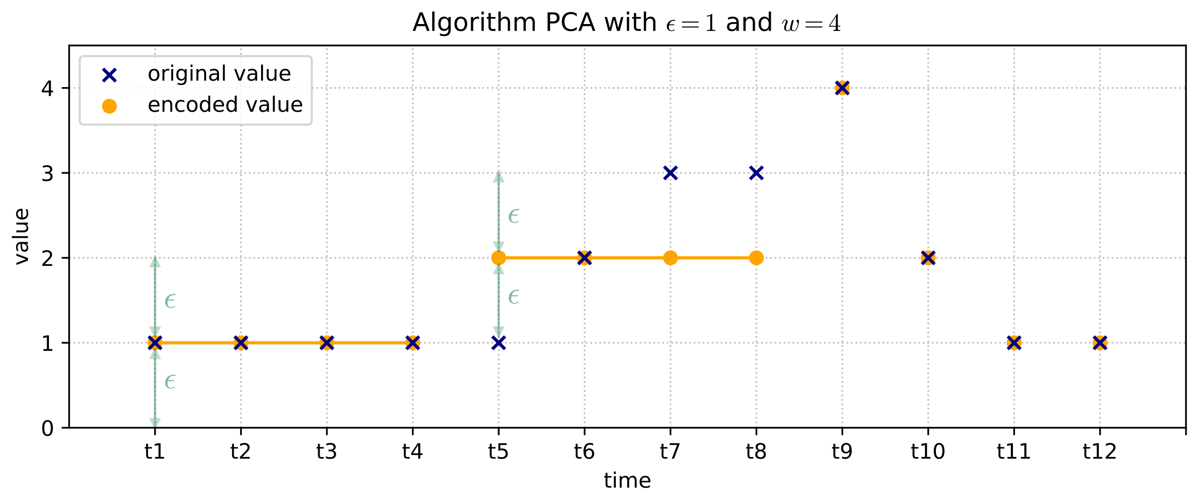


FIGURE 2.8: Example PCA

Figures 2.9 and 2.10 show the pseudocode for the code_column_*NM* and decode_column_*NM* subroutines, respectively, for algorithm PCA.

```
input : column: column of the csv data file to be coded
        out: binary file encoded with algorithm PCA
        ϵ: maximum error threshold
        w: fixed window size
win = new_window()
foreach entry in column.entries do
    win.push(entry)
    if win.size < w then
        continue
    end
    if win.all_entries_are_no_data then
        out.code_bit(0)
        out.code_base_2(column.no_data_int, column.total_bits)
    else if win.all_entries_are_integers and |win.max − win.min| ≤ 2 ∗ ϵ then
        out.code_bit(0)
        value = (win.min + win.max)/2
        out.code_base_2(value, column.total_bits)
    else
        out.code_bit(1)
        foreach win_val in win.values do
            value = (win_val == “N”) ? column.no_data_int : win_val
            out.code_base_2(value, column.total_bits)
        end
    end
    win = new_window()
end
```

FIGURE 2.9: PCA.code_column_*NM* pseudocode.

Notice that the pseudocode for subroutine decode_column_*NM*, presented in Figure 2.10, is very similar to the pseudocode for subroutine decode_column_*M*, presented in Figure 2.7. The only difference is that the former has two additional lines (5 and 12), which are necessary for decoding the gaps.

```
input : column: column encoded with algorithm PCA
        out: decoded csv data file
        w: fixed window size
1  foreach entry in column.entries do
2      bit = in.decode_bit()
3      if bit == 0 then
4          value = in.decode_base_2(column.total_bits)
5          value = (value == column.no_data_int) ? "N" : value
6          repeat w times
7              out.write_string(value)
8          end
9      else
10         repeat w times
11             value = in.decode_base_2(column.total_bits)
12             value = (value == column.no_data_int) ? "N" : value
13             out.write_string(value)
14         end
15     end
16 end
```

Figure 2.10: PCA.decode_column_*NM* pseudocode.

2.5 APCA

Algorithm APCA [9], also known as Adaptive Piecewise Constant Approximation, is a Constant model algorithm, so it encodes signals using constant functions. It operates similarly to algorithm PCA, but the blocks in which the data are processed and encoded have a variable size. The window size parameter (w) establishes the maximum block size allowed for algorithm APCA. This algorithm supports lossless and lossy compression, with both variants (M and NM).

Figure 2.11 shows the pseudocode for the code_column_M subroutine for algorithm APCA. TODO: continue...

```
input  : column: column of the csv data file to be coded
         out: binary file encoded with algorithm APCA
         ϵ: maximum error threshold
         w: maximum window size
win = new_window()
foreach entry in column.entries do
    if win.size == 0 then
        win.push(entry)
        continue
    end
    code_window = false
    if win.size == w then
        code_window = true
    else
        win.push(entry)
        if |win.max − win.min| ≤ 2 ∗ ϵ then
            win.code_value = (win.min + win.max)/2
        else
            entry = win.unpush()
            code_window = true
        end
    end
    if code_window then
        out.code_base_2(win.size − 1, log2 w)
        out.code_base_2(win.code_value, column.total_bits)
        win = new_window()
        win.push(entry)
    end
end
```

FIGURE 2.11: APCA.code_column_M pseudocode.

TODO: describe example in Figure 2.12.

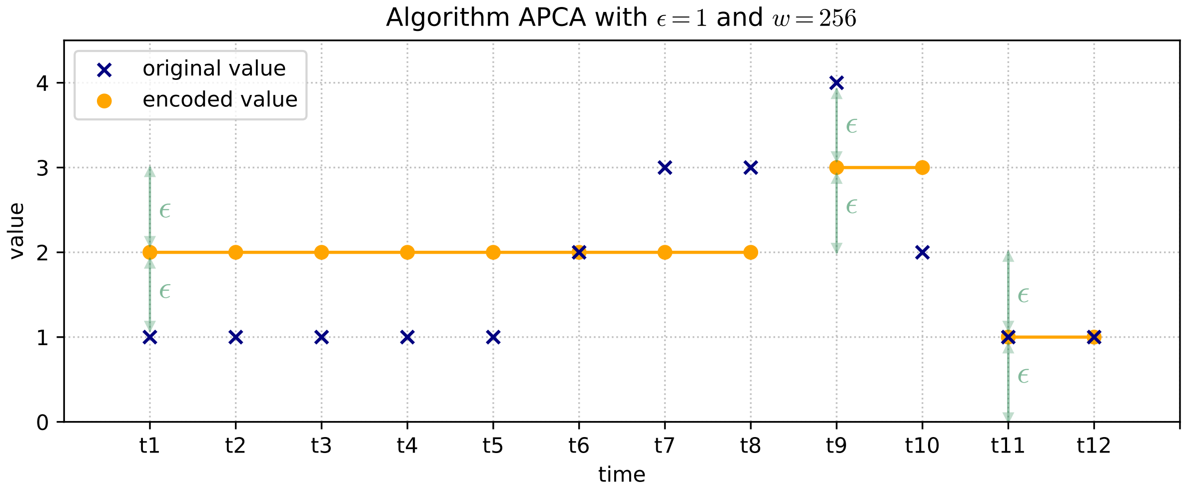


FIGURE 2.12: Example APCA

2.6 Algorithm CA

Algorithm CA [10], also known as Critical Aperture, supports lossless and lossy compression, with both variants (M and NM), and it has a window size parameter (w) that establishes the maximum block size in which the data are processed and encoded. It is a Linear model algorithm, so it encodes signals using linear functions.

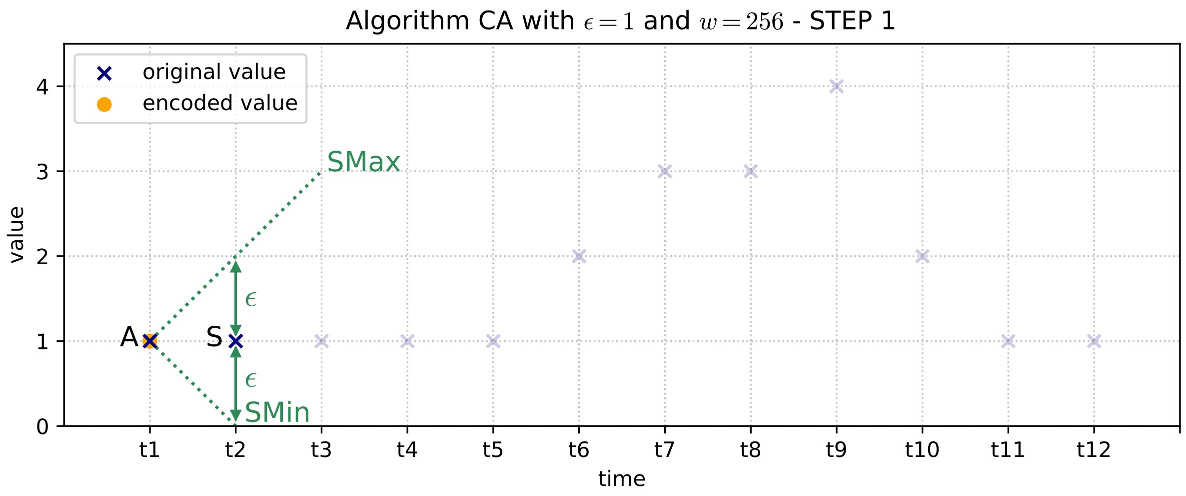
Figure 2.13 shows the pseudocode for the code_column_M subroutine for algorithm CA. TODO: continue...

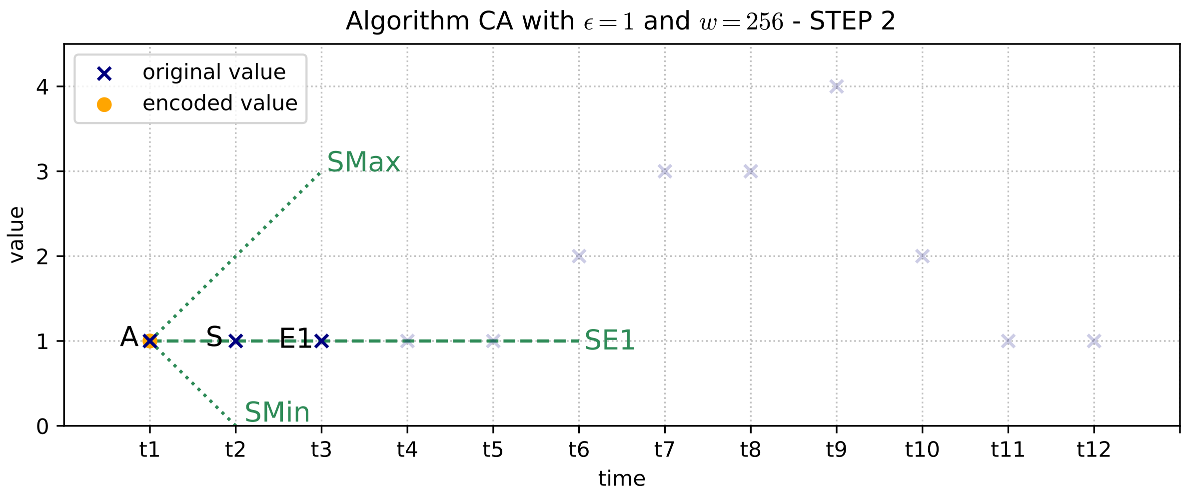
```
input  : column: column of the csv data file to be coded
         out: binary file encoded with algorithm CA
         ϵ: maximum error threshold
         w: maximum window size
win = new_window()
foreach entry in column.entries do
    code_window = false
    code_value = false
    if entry == column.entries[0] then
        code_value = true
    else if win.size == 0 then
        win.add_incoming_point(entry)
        continue
    else if win.size == w  or not win.CA_condition_holds(entry, ϵ) then
        code_window = true
        code_value = true
    end
    if code_window then
        out.code_base_2(win.size − 1, log₂ w)
        out.code_base_2(win.code_value, column.total_bits)
    end
    if code_value then
        out.code_base_2(0, log₂ w)
        out.code_base_2(entry.value, column.total_bits)
        win.set_archived_point(entry)
    end
end
```

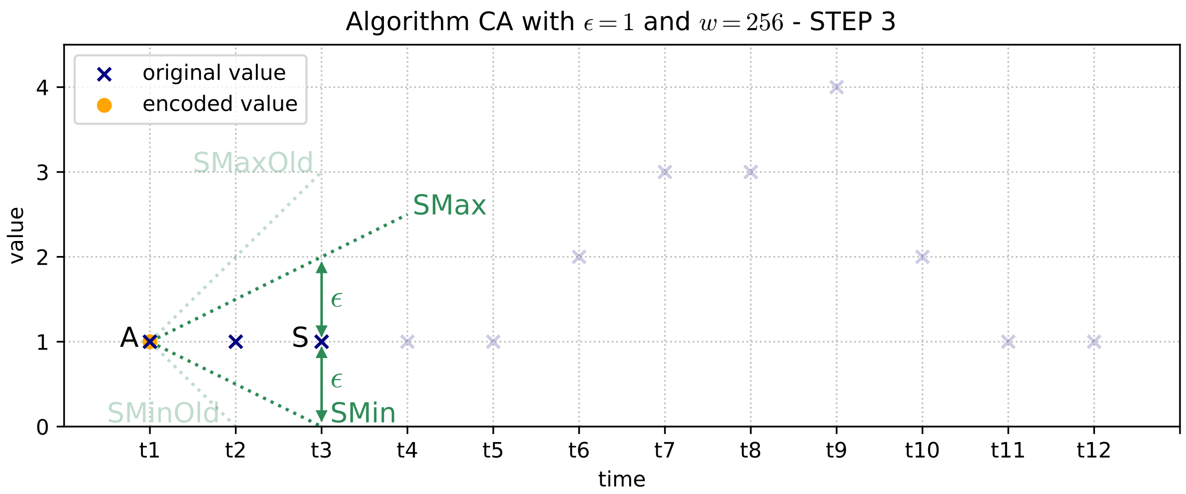
Figure 2.13: CA.code_column_M pseudocode.

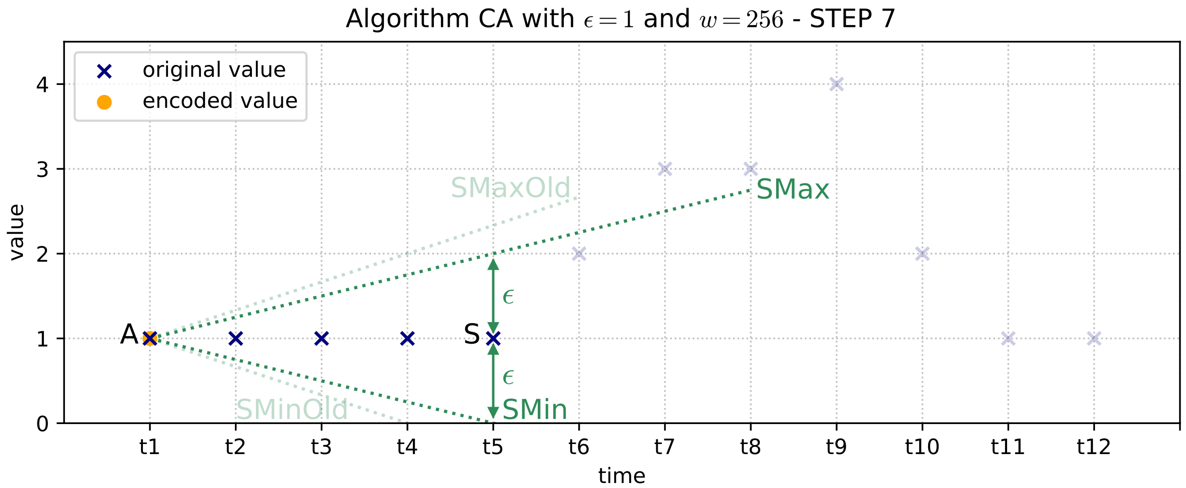
TODO:

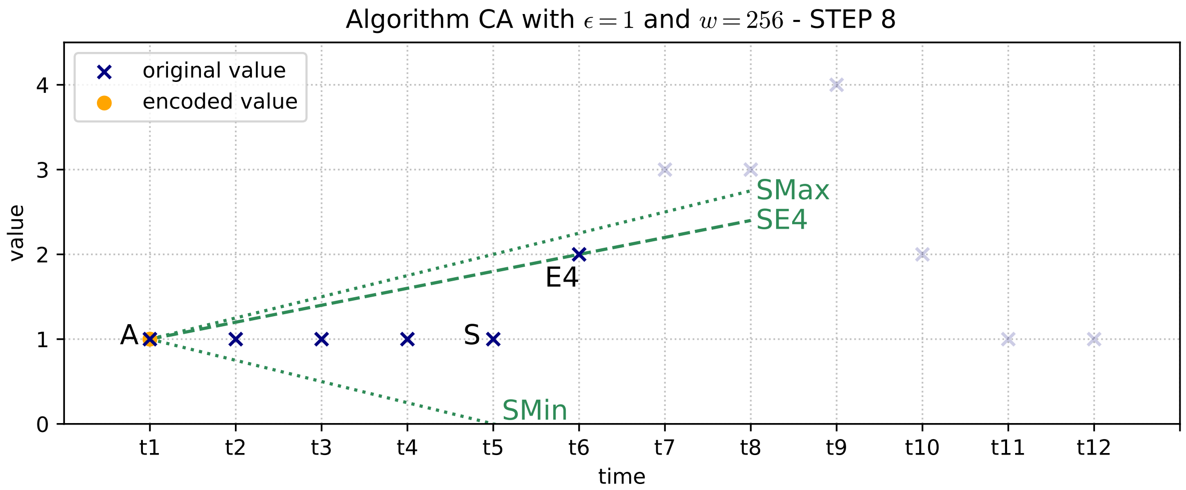
- describir el ejemplo de las próximas 8 imágenes.
- mencionar que se tienen en cuenta los delta pero se omite en el pseudocódigo.
- mencionar que se tiene en cuenta el caso delta = 0 => por esto es que en el STEP 11 el valor de t7 se codifica aparte (si delta = 0, entonces t6 = t7, pero no necesariamente f(t6) = f(t7))
- mencionar que maximum window size tiene que ser mayor a 1 para que code value funcione. - este comentario no sé si aplica...

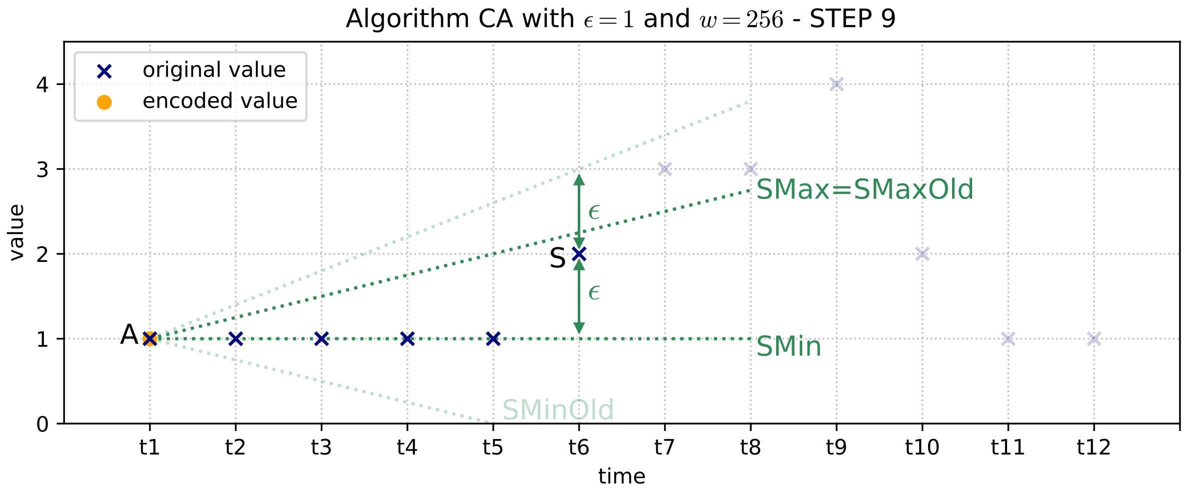

Algorithm CA with $\epsilon = 1$ and $w = 256$ - STEP 1
original value
encoded value
SMax
SMin
A
S
ϵ
ϵ
value
time
t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12
0 1 2 3 4

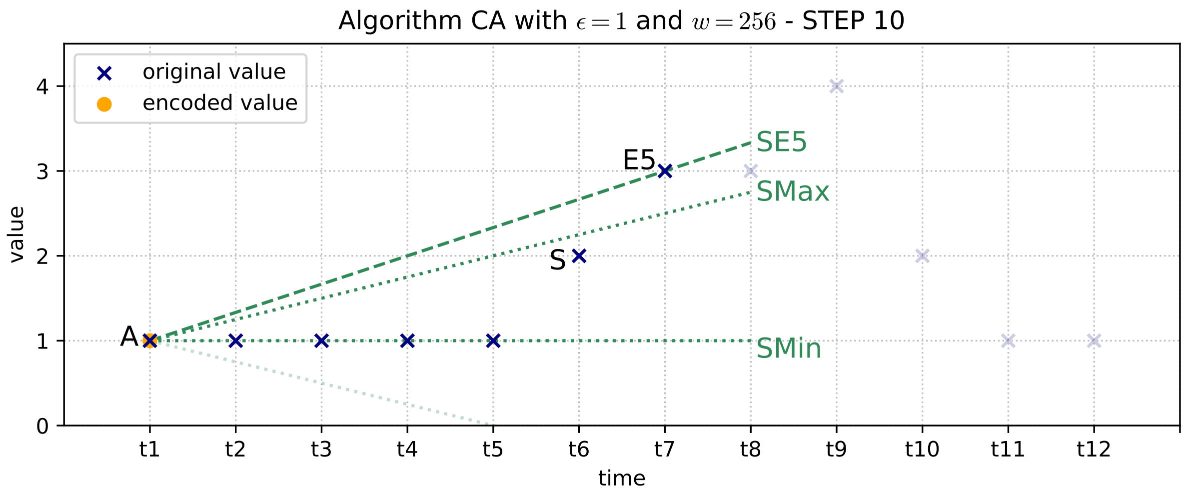

Algorithm CA with $\epsilon = 1$ and $w = 256$ - STEP 2
original value
encoded value
SMax
SMin
A
S
E1
SE1
value
time
t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12
0 1 2 3 4

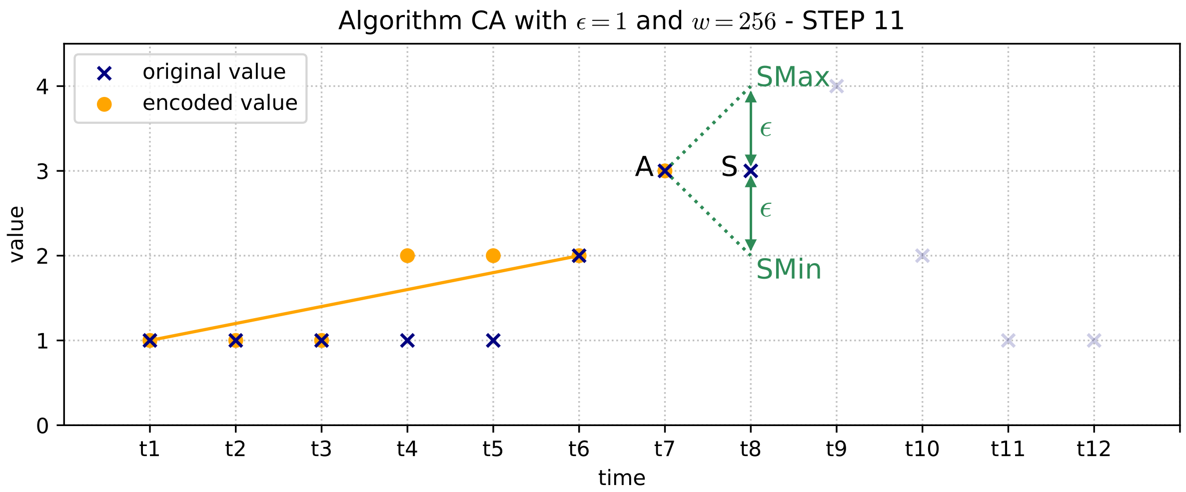

Algorithm CA with $\epsilon = 1$ and $w = 256$ - STEP 3
original value
encoded value
SMaxOld
SMax
SMinOld
SMin
A
S
ϵ
ϵ
value
time
t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12
0 1 2 3 4


Algorithm CA with $\epsilon = 1$ and $w = 256$ - STEP 7
original value
encoded value
SMaxOld
SMax
SMinOld
SMin
A
S
ϵ
ϵ
value
time
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
t12
0
1
2
3
4


Algorithm CA with $\epsilon = 1$ and $w = 256$ - STEP 8
original value
encoded value
SMax
SE4
E4
A
S
SMin
value
time
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
t12
0
1
2
3
4


Algorithm CA with $\epsilon = 1$ and $w = 256$ - STEP 9
original value
encoded value
SMax=SMaxOld
S
ϵ
ϵ
A
SMin
SMinOld
value
time
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
t12
0
1
2
3
4


Algorithm CA with $\epsilon = 1$ and $w = 256$ - STEP 10
original value
encoded value
value
time
A
S
E5
SE5
SMax
SMin
t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12
0 1 2 3 4


Algorithm CA with $\epsilon = 1$ and $w = 256$ - STEP 11
original value
encoded value
value
time
A
S
SMax
SMin
ϵ
ϵ
t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12
0 1 2 3 4

TODO: describe example in Figure 2.14.

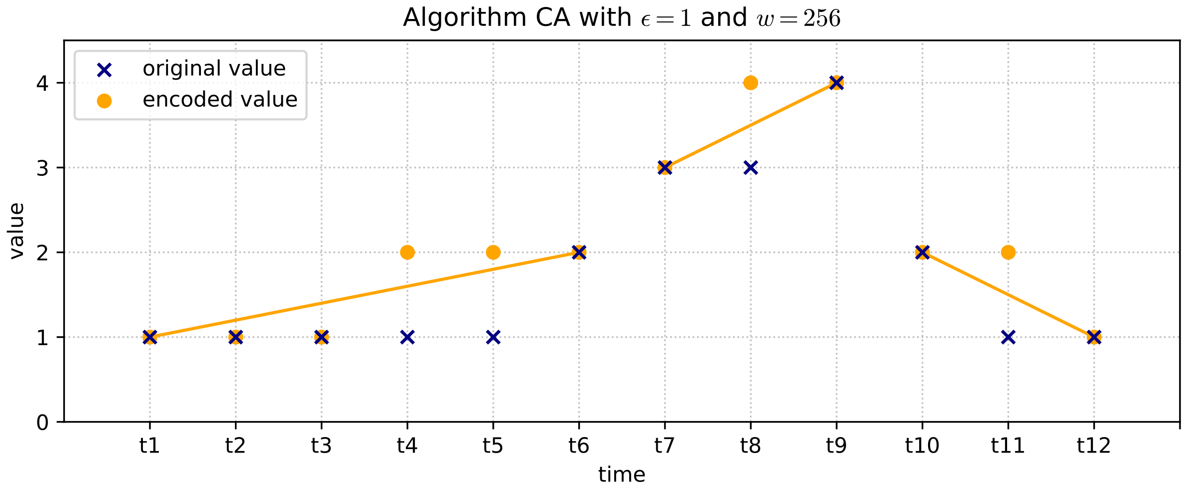


FIGURE 2.14: Example CA

2.7 Algorithms PWLH and PWLHInt

Algorithm PWLH [11], also known as PieceWise Linear Histogram, supports lossless and lossy compression, with both variants (M and NM), and it has a window size parameter (w) that establishes the maximum block size in which the data are processed and encoded. It is a Linear model algorithm, so it encodes signals using linear functions. In particular, the data points in each window are modeled by a line segment that minimizes the maximum distance from these points to the segment. For the operations in the two-dimensional Euclidean space, which involve calculating said segment, and computing the convex hull of the data points by applying Graham's Scan algorithm [22], we reused the code from the framework linked in [6].

Figure 2.15 shows the pseudocode for the code_column_M subroutine for algorithm PWLH. Creating a new window involves creating an associated empty convex hull (lines 1 and 24), and every time a data entry is added or removed from the window, the convex hull must be updated (lines 5, 13, 16 and 26). The window is coded in two scenarios: when it reaches the maximum size allowed (line 9), or when the convex hull violates the threshold condition (line 14). The latter scenario is further explained in the example presented ahead in this section.

```
input  : column: column of the csv data file to be coded
         out: binary file encoded with algorithm PWLH
         ϵ: maximum error threshold
         w: maximum window size
1  win = new_window()
2  foreach entry in column.entries do
3      if win.size == 0 then
4          win.push(entry)
5          win.update_convex_hull()
6          continue
7      end
8      code_window = false
9      if win.size == w then
10         code_window = true
11     else
12         win.push(entry)
13         win.update_convex_hull()
14         if not win.PWLH_condition_holds(ϵ) then
15             entry = win.unpush()
16             win.update_convex_hull()
17             code_window = true
18         end
19     end
20     if code_window then
21         point_A, point_B = win.get_approximation_segment()
22         out.code_float(point_A.y)
23         out.code_float(point_B.y)
24         win = new_window()
25         win.push(entry)
26         win.update_convex_hull()
27     end
28 end
```

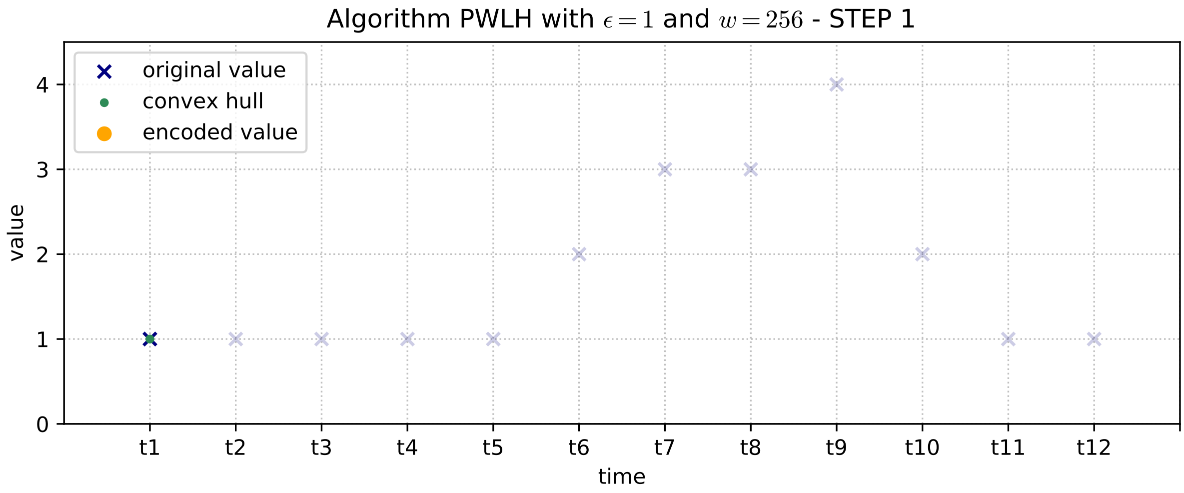
FIGURE 2.15: PWLH.code_column_M pseudocode.

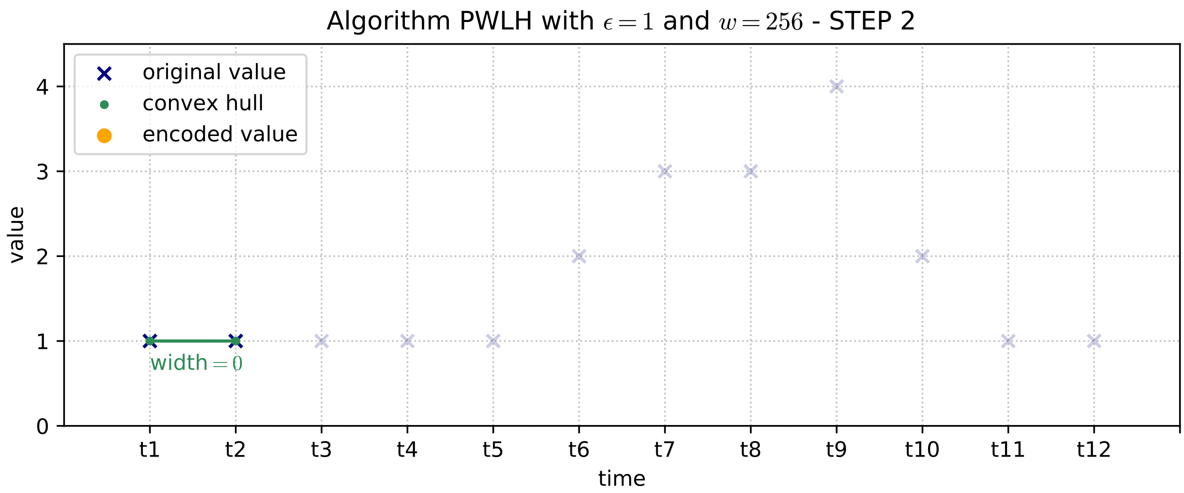
Coding a window involves coding the y-coordinates of the two endpoints of the segment that minimizes the maximum distance from the points in the window to the segment. These coordinates are encoded as float values (lines 22-23), since this is the precision adopted in the method used for calculating said segment (line 21). Notice that the x-coordinates are previously encoded with the timestamp column (recall line 9 in the pseudocode presented in Figure 2.1), so they do not have to be encoded again. However, it is important to note that, since PWLH is a Linear model algorithm, the x-coordinates are considered in the actual routine, but they were omitted in the pseudocode for clarity.

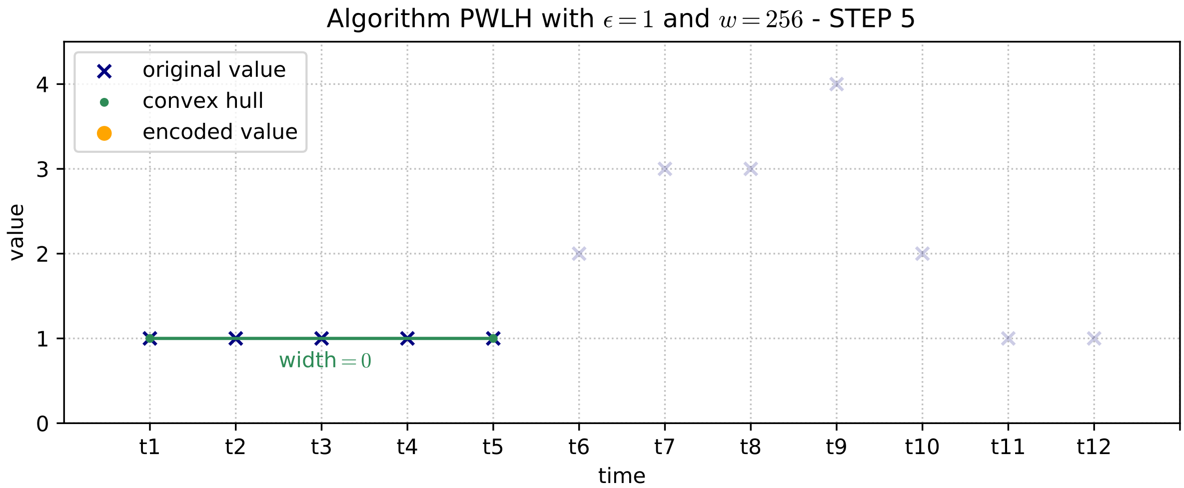
TODO: mencionar PWLHInt.

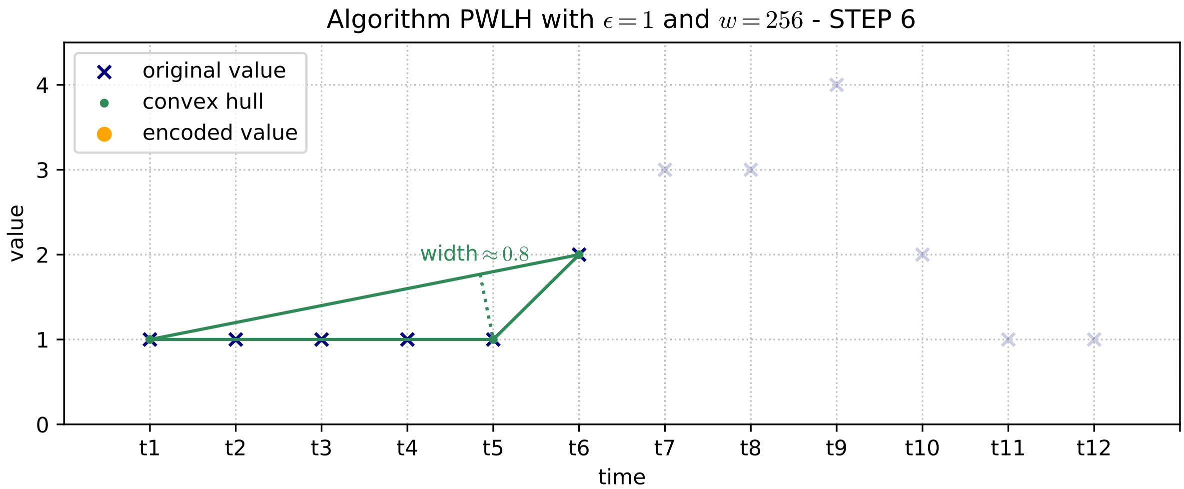
TODO:

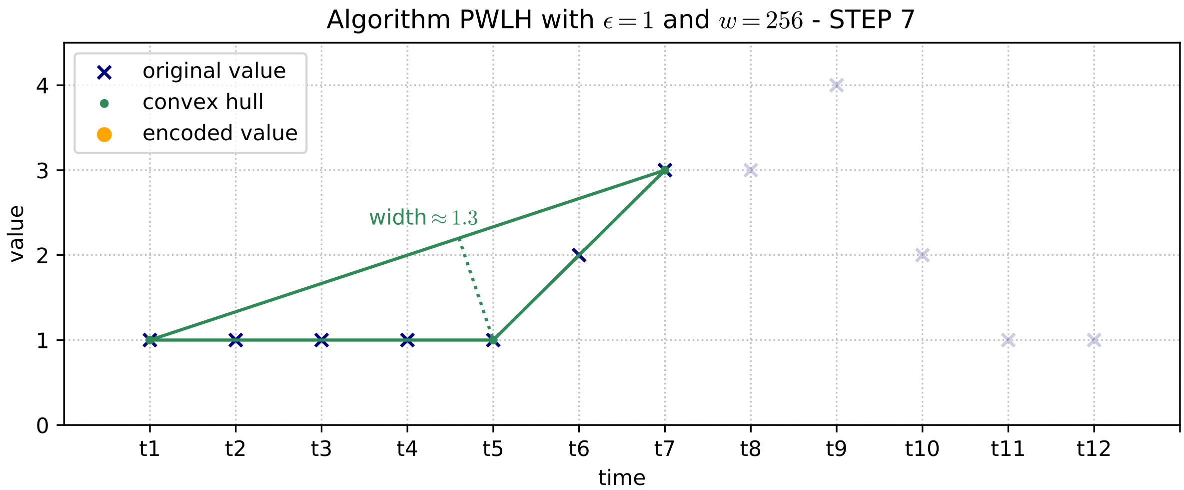
- mencionar PWLHInt antes de mostrar pseudocódigos
- describir el ejemplo de las próximas 9 imágenes.

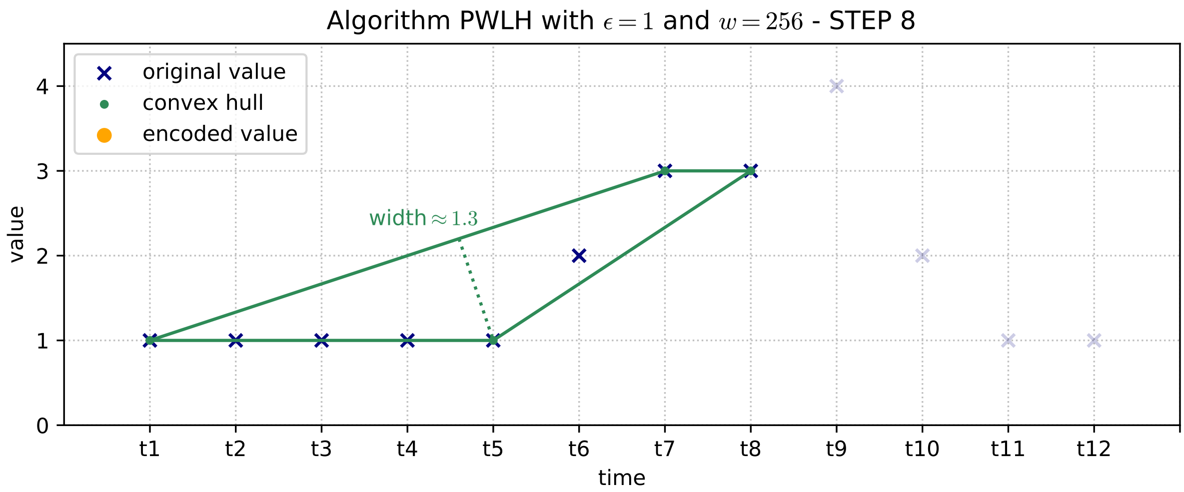




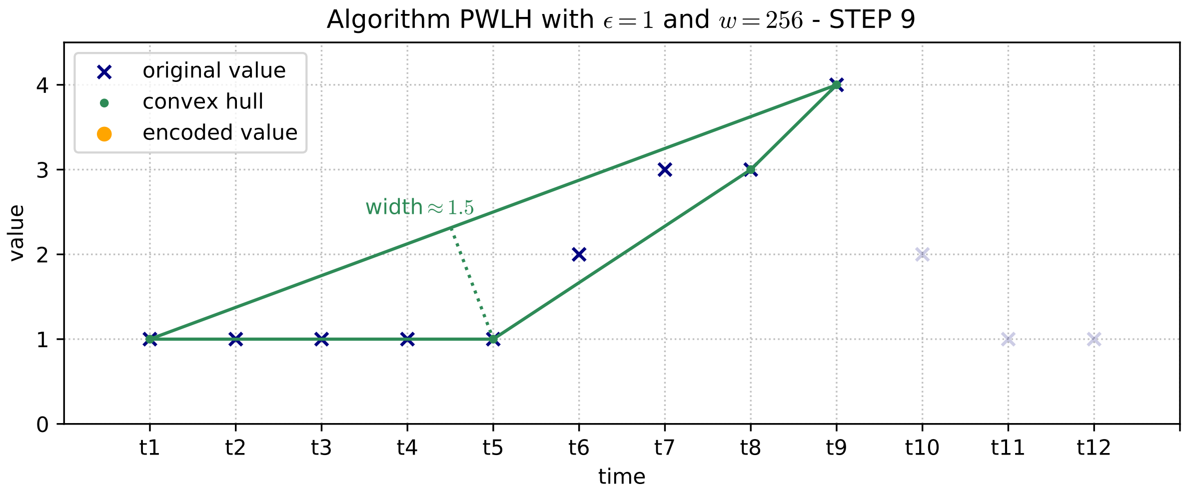

Algorithm PWLH with $\epsilon = 1$ and $w = 256$ - STEP 5
original value
convex hull
encoded value
width = 0
value
time
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
t12


Algorithm PWLH with $\epsilon = 1$ and $w = 256$ - STEP 6
original value
convex hull
encoded value
width ≈ 0.8
value
time
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
t12

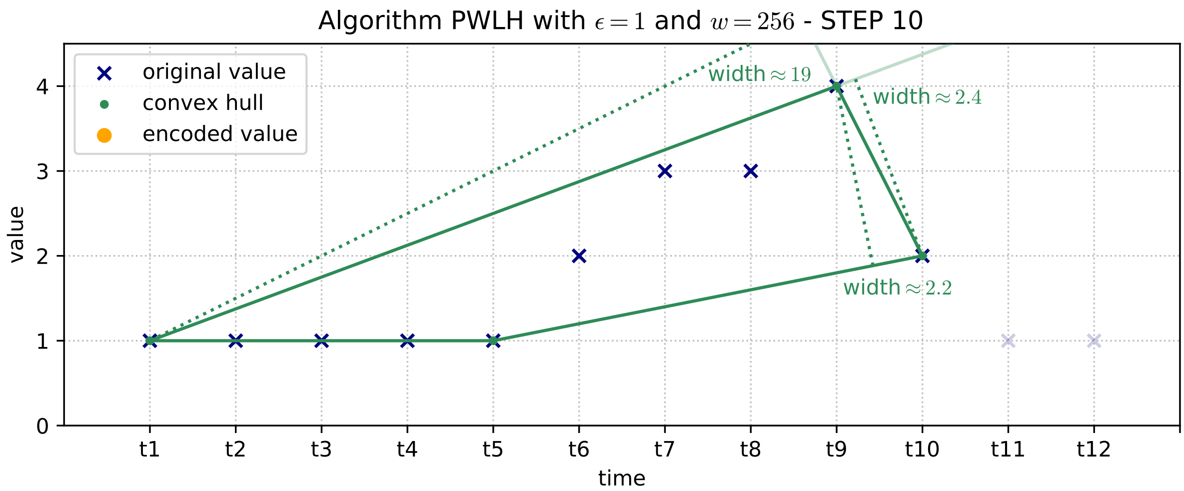

Algorithm PWLH with $\epsilon = 1$ and $w = 256$ - STEP 7
original value
convex hull
encoded value
width ≈ 1.3
value
time
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
t12



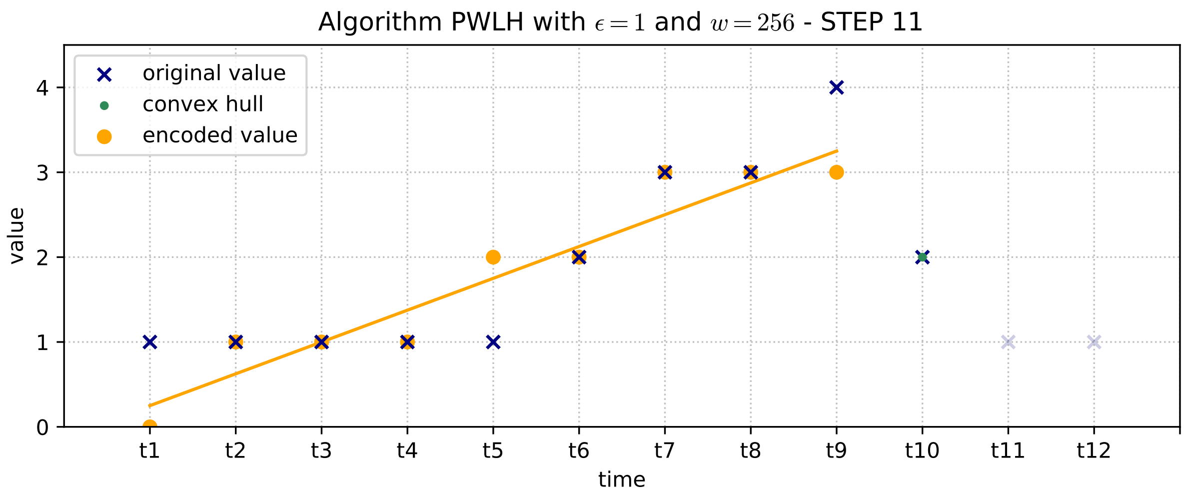
Algorithm PWLH with $\epsilon = 1$ and $w = 256$ - STEP 8
original value
convex hull
encoded value
width ≈ 1.3
value
time
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
t12



Algorithm PWLH with $\epsilon = 1$ and $w = 256$ - STEP 9
original value
convex hull
encoded value
width ≈ 1.5
value
time
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
t12



Algorithm PWLH with $\epsilon = 1$ and $w = 256$ - STEP 10
original value
convex hull
encoded value
width ≈ 19
width ≈ 2.4
width ≈ 2.2
value
time
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
t12


Algorithm PWLH with $\epsilon = 1$ and $w = 256$ - STEP 11
original value
convex hull
encoded value
value
time
t1
t2
t3
t4
t5
t6
t7
t8
t9
t10
t11
t12
0
1
2
3
4

TODO: describe examples in figures 2.16 and 2.17.

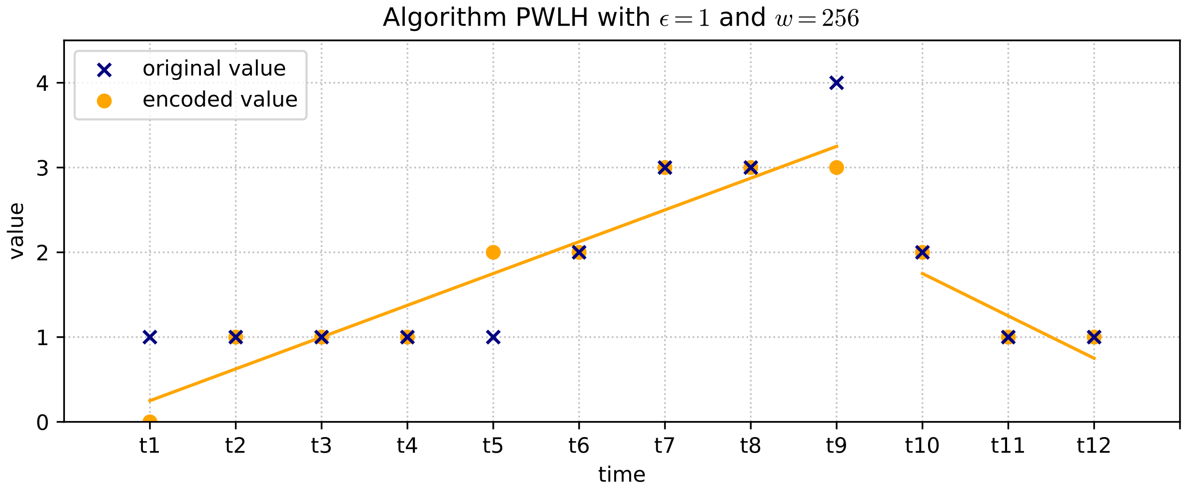


FIGURE 2.16: Example PWLH

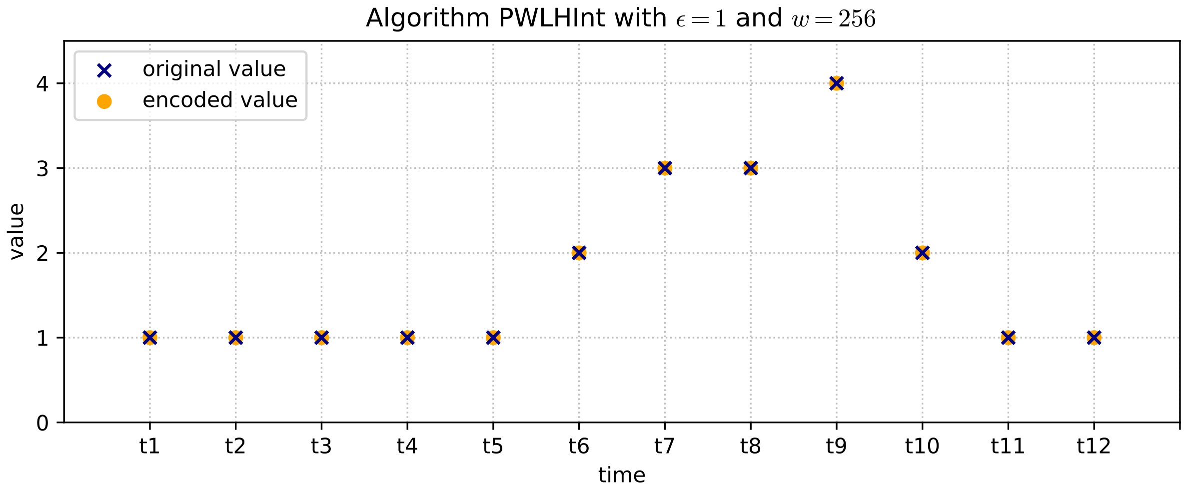


FIGURE 2.17: Example PWLHInt

2.8 Algorithm SF

TODO: describe example in Figure 2.18.

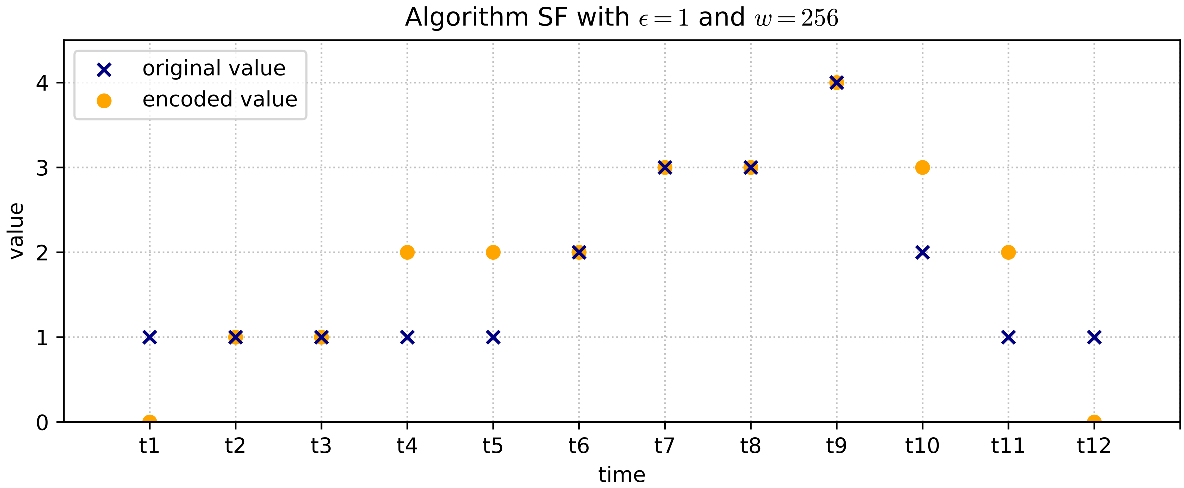


FIGURE 2.18: Example SF

2.9 Algorithm FR

Mencionar que lo utiliza la ESA (European Space Agency)

TODO: describe example in Figure 2.19.

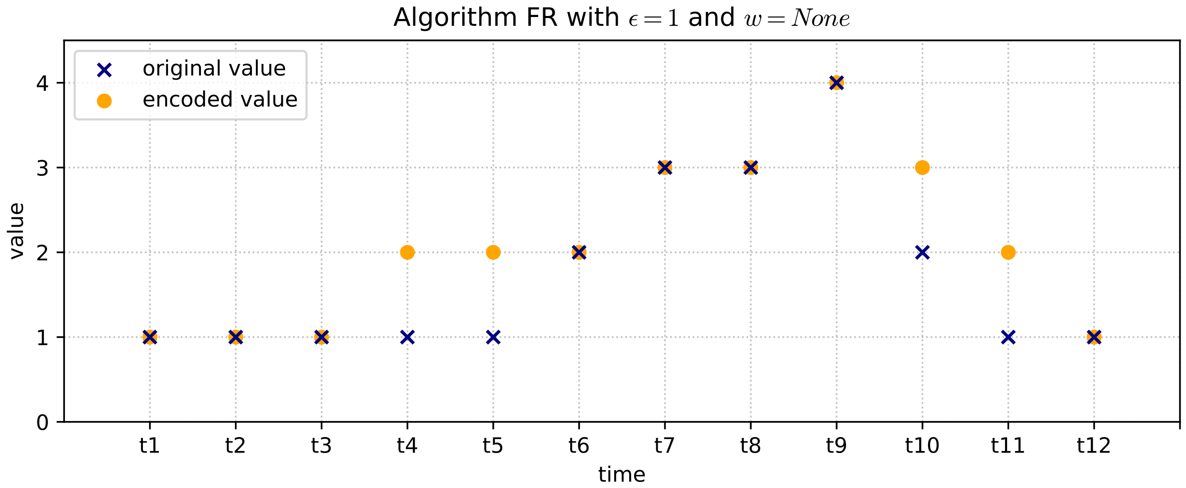


FIGURE 2.19: Example FR

2.10 Algorithms GAMPS and GAMPSLimit

Ver los siguientes documentos:

- [08] AVANCES / DUDAS
- [09] AVANCES / DUDAS
- [10] AVANCES / DUDAS
- [11] AVANCES / DUDAS
- [12] AVANCES / DUDAS

Chapter 3

Experimental Results

In this chapter we present our experimental results. The main goal of our experiments is to analyze the performance of each of the coding algorithms presented in Chapter 2, by encoding the various datasets introduced in Chapter 1.

In Section 3.1 we describe our experimental setting and define the evaluated combinations of algorithms, their variants and parameter values, and the figures of merit used for comparison.

In Section 3.2 we compare the compression performance of the masking and non-masking variants for each coding algorithm. The results show that on datasets with few or no gaps the performance of both variants is roughly the same, while on datasets with many gaps the masking variant always performs better, in some cases with a significative difference. These results suggest that the masking variant is more robust and performs better in general.

In Section 3.3 we analyze the extent to which the window size parameter **impacts/affects** the compression performance of the coding algorithms. We compress each dataset file, and compare the results obtained when using the optimal window size (i.e. the one that achieves the best compression) for each file, with the results obtained when using the optimal window size for the whole dataset. The results indicate that the **impact/effect** of using the optimal window size for the whole dataset instead of the optimal window size for each file is rather small.

In Section 3.4 we compare the performance of the different coding algorithms among each other and with the general purpose compression algorithm gzip. Among the tested coding algorithms, for larger error thresholds APCA is the best algorithm for compressing every data type in our experimental data set, while for lower thresholds the recommended algorithms are PCA, APCA and FR, depending on the data type. If we also consider algorithm gzip, **there isn't an algorithm that is better for compressing every data type for any threshold value / for no threshold value there exists an algorithm that is better for compresing every data type**. Depending on the data type, the recommended algorithms are APCA and gzip for larger error thresholds, and PCA, APCA, FR and gzip for lower thresholds.

3.1 Experimental Setting

We denote by A the set of all the coding algorithms presented in Chapter 2. For an algorithm $a \in A$, we denote by a_v its variant v, where v can be *M* (masking) or *NM* (non-masking). There exist some $a \in A$ for which either a_M or a_{NM} is invalid (recall this information from Table 2.1). We denote by V the set of variants composed of every valid variant a_v for every algorithm $a \in A$. Also, we denote by A_M the subset of algorithms from A composed of every algorithm for which both variants, a_M and a_{NM}, are valid.

We evaluate the compression performance of every algorithm $a \in A$ on the datasets described in Chapter 1. For each algorithm we test every valid variant a_v. We also test several combinations of algorithm parameters. Specifically, for the algorithms that admit a window size parameter w (every algorithm except *Base* and *SF*), we test all the values of w in the set $W = \{4, 8, 16, 32, 64, 128, 256\}$. For the encoders that admit a lossy compression mode with a threshold parameter e (every encoder except *Base*), we test all the values of e in the set $E = \{1, 3, 5, 10, 15, 20, 30\}$, where each threshold is expressed as a percentage fraction of the standard deviation of the data being encoded. For example, for certain data with a standard deviation of 20, taking $e = 10$ implies that the lossy compression allows for a maximal per-sample distortion of 2 sampling units.

Definition 3.1.1. We refer to a specific combination of a coding algorithm variant and its parameter values as a *coding algorithm instance (CAI)*. We define *CI* as the set of all the CAIs obtained by combining each of the variants $a_v \in V$ with the parameter values (from W and E) that are suitable for algorithm a. We denote by $c_{<a_v,w,e>}$ the CAI obtained by setting a window size parameter equal to w and a threshold parameter equal to e on algorithm variant a_v.

We assess the compression performance of a CAI mainly through the compression ratio, which we define next. For this definition, we regard *Base* as a trivial CAI that serves as a base ground for compression performance comparison (recall the definition of algorithm *Base* from Section 2.3).

Definition 3.1.2. Let f be a file and z a data type of a certain dataset. We define f_z as the subset of data of type z from file f. For example, for the dataset Hail, the data type z may be Latitude, Longitude, or Size.

Definition 3.1.3. Let f be a file and z a data type of a certain dataset. Let $c \in CI$ be a CAI. We define $|c(z, f)|$ as the size in bits of the resulting bit stream obtained by coding f_z with c.

Definition 3.1.4. The *compression ratio (CR)* of a CAI $c \in CI$ for the data type z of a certain file f is the fraction of $|c(z, f)|$ with respect to $|\text{Base}(z, f)|$, i.e.,

$$CR(c, z, f) = \frac{|c(z, f)|}{|\text{Base}(z, f)|}. \tag{3.1}$$

Notice that smaller values of CR correspond to better performance. Our main goals are to analyze which CAIs yield the smallest values in (3.1) for the different data types, and to study how the CR depends on the different algorithms, their variants and the parameter values.

To compare the compression performance between a pair of CAIs we calculate the relative difference, which we define next.

Definition 3.1.5. The *relative difference (RD)* between a pair of CAIs $c_1, c_2 \in CI$ for the data type z of a certain file f is given by

$$RD(c_1, c_2, z, f) = 100 \times \frac{|c_2(z, f)| - |c_1(z, f)|}{|c_2(z, f)|}. \tag{3.2}$$

Notice that c_1 has a better performance than c_2 if (3.2) is positive.

In some of our experiments we consider the performance of algorithms on complete datasets, rather than individual files. With this in mind, we extend the definitions 3.1.3–3.1.5 to datasets, as follows.

Definition 3.1.6. Let z be a data type of a certain dataset d. We define $F(d, z)$ as the set of files f from dataset d for which f_z is not empty.

Definition 3.1.7. Let z be a data type of a certain dataset d. Let $c \in CI$ be a CAI. We define $|c(z, d)|$ as

$$|c(z, d)| = \sum_{f \in F(d,z)} |c(z, f)|. \tag{3.3}$$

Definition 3.1.8. The *compression ratio (CR)* of a CAI $c \in CI$ for the data type z of a certain dataset d is given by

$$CR(c, z, d) = \frac{|c(z, d)|}{|\text{Base}(z, d)|}. \tag{3.4}$$

Definition 3.1.9. The *relative difference (RD)* between a pair of CAIs $c_1, c_2 \in CI$ for the data type z of a certain dataset d is given by

$$RD(c_1, c_2, z, d) = 100 \times \frac{|c_2(z, d)| - |c_1(z, d)|}{|c_2(z, d)|}. \tag{3.5}$$

3.2 Comparison of Masking and Non-Masking Variants

In this section, we compare the compression performance of the masking and non-masking variants of every coding algorithm in A_M. Specifically, we compare:

- PCA_M against PCA_{NM}
- APCA_M against APCA_{NM}
- CA_M against CA_{NM}
- PWLH_M against PWLH_{NM}
- PWLHInt_M against PWLHInt_{NM}
- GAMPSLimit_M against GAMPSLimit_{NM}

For each algorithm $a \in A_M$ and each threshold parameter, we compare the performance of a_M and a_{NM}. For the purpose of this comparison, we choose the most favorable window size for each variant a_v, in the sense of the following definition.

Definition 3.2.1. The *optimal window size (OWS)* of a coding algorithm variant $a_v \in V$, and a threshold parameter $e \in E$, for the data type z of a certain dataset d, is given by

$$OWS(a_v, e, z, d) = \underset{w \,\in\, W}{\arg\min} \left\{ CR(c_{<a_v,w,e>}, z, d) \right\}, \tag{3.6}$$

where we break ties in favor of the smallest window size.

For each data type z of each dataset d, and each coding algorithm $a \in A_M$ and threshold parameter $e \in E$, we calculate the RD between $c_{<a_M,w^*_M,e>}$ and $c_{<a_{NM},w^*_{NM},e>}$, as defined in (3.5), where $w^*_M = \text{OWS}(a_M, e, z, d)$ and $w^*_{NM} = \text{OWS}(a_{NM}, e, z, d)$.

As an example, in figures 3.1 and 3.2 we show the CR and the RD, as a function of the threshold parameter, obtained for two data types of two different datasets. Figure 3.1 shows the results for the data type “SST” of the dataset SST, and Figure 3.2 shows the results for the data type “Longitude” of the dataset Tornado. In Figure 3.1 we observe a large RD favoring the masking variant for all tested algorithms. On the other hand, in Figure 3.2 we observe that the non-masking variant outperforms the masking variant for all algorithms. We notice, however, that the RD is very small in the latter case.

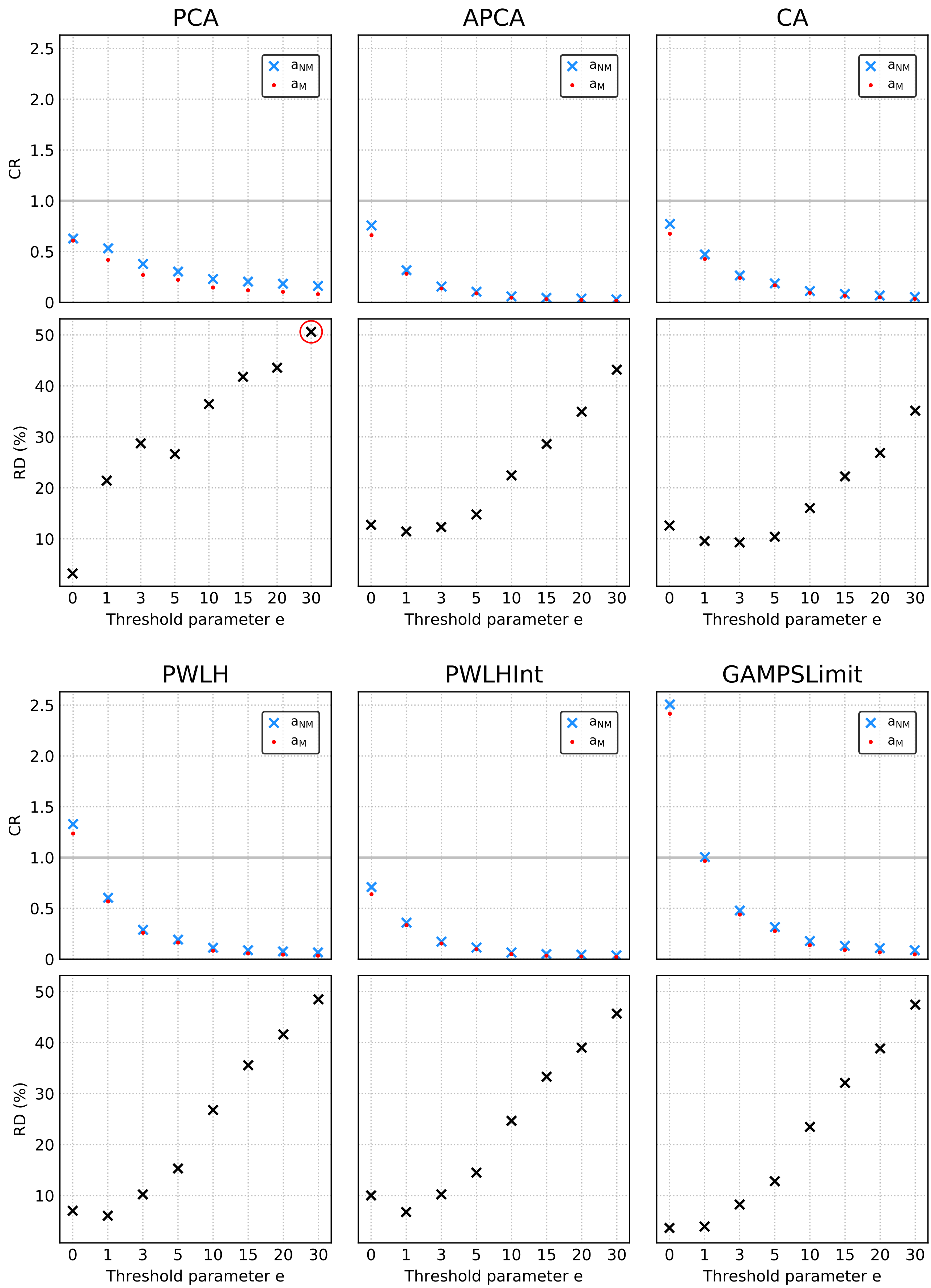


FIGURE 3.1: CR and RD plots for every pair of algorithm variants $a_M, a_{NM} \in A_M$, for the data type "SST" of the dataset SST. In the RD plot for algorithm PCA we highlight with a red circle the marker for the maximum value (50.60%) obtained for all the tested CAIs.

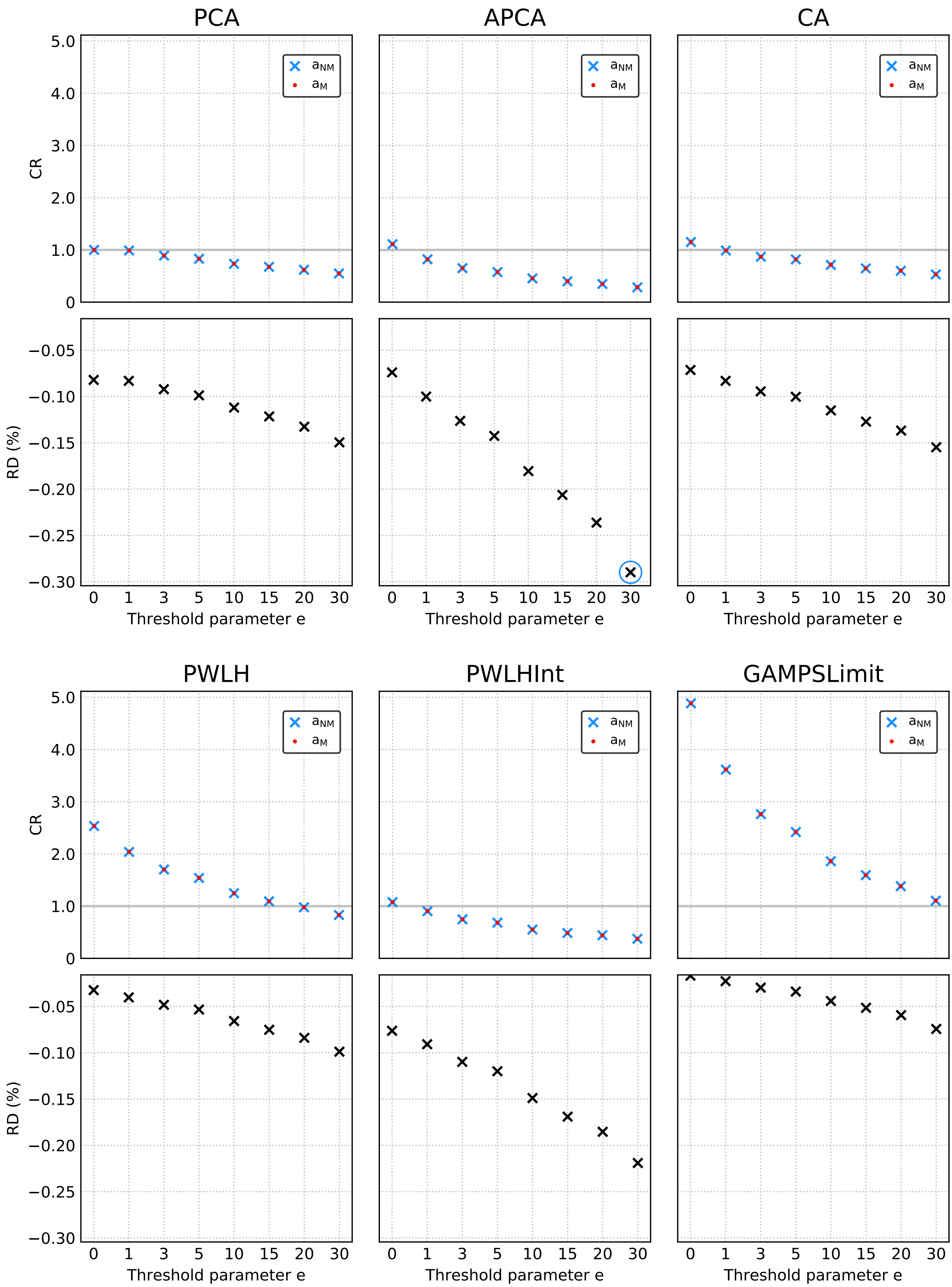


FIGURE 3.2: CR and RD plots for every pair of algorithm variants $a_M, a_{NM} \in A_M$, for the data type "Longitude" of the dataset Tornado. In the RD plot for algorithm APCA we highlight with a blue circle the marker for the minimum value (-0.29%) obtained for all the tested CAIs.

We analyze the experimental results to compare the performance of the masking and non-masking variants of each algorithm. For each data type, we iterate through each algorithm $a \in A_M$, and each threshold parameter $e \in E$, and we calculate the RD between the CAIs $c_{<a_M, w^*_M, e>}$ and $c_{<a_{NM}, w^*_{NM}, e>}$, obtained by setting the OWS for the masking variant a_M and the non-masking variant a_{NM}, respectively. Since we consider 8 threshold parameters and there are 6 algorithms in A_M, for each data type we compare a total of 48 pairs of CAIs. Table 3.1 summarizes the results of these comparisons, aggregated by dataset. The number of pairs of CAIs evaluated for each dataset depends on the number of different data types it contains.

Dataset	**Dataset Characterstic**	**Cases where a_M outperforms a_{NM} (%)**	**RD (%) Range**
IRKIS	Many gaps	48/48 (100%)	(0; 36.88]
SST	Many gaps	48/48 (100%)	(0; 50.60]
ADCP	Many gaps	48/48 (100%)	(0; 17.35]
ElNino	Many gaps	336/336 (100%)	(0; 50.52]
Solar	Few gaps	73/144 (50.7%)	[-0.25; 1.77]
Hail	No gaps	0/144 (0%)	[-0.04; 0)
Tornado	No gaps	0/96 (0%)	[-0.29; 0)
Wind	No gaps	0/144 (0%)	[-0.12; 0)

Table 3.1: Range of values for the RD between the masking and non-masking variants of each algorithm (last column); we highlight the maximum (red) and minimum (blue) values taken by the RD. The results are aggregated by dataset. The second column indicates the characteristic of each dataset, in terms of the amount of gaps. The third column shows the number of cases in which the masking variant outperforms the non-masking variant of a coding algorithm, and its percentage among the total pairs of CAIs compared for a dataset.

Consider, for example, the results for the dataset Wind, in the last row. The second column shows that there are no gaps in any of the data types of the dataset (recall the dataset description from Table 1.1). Since the dataset has three data types, we compare a total of $3 \times 48 = 144$ pairs of CAIs. The third column reveals that in none of these comparisons the masking variant a_M outperforms the non-masking variant a_{NM}, i.e. the RD is always negative. The last column shows the range for the values attained by the RD for those tested CAIs.

Observing the last column of Table 3.1, we notice that in every case in which the non-masking variant performs best, the RD is close to zero. The minimum value it takes is -0.29%, which is obtained for the data type "Longitude" of the dataset Tornado, with algorithm APCA, and error parameter $e = 30$. In Figure 3.2 we highlight the marker associated to this minimum with a blue circle. On the other hand, we also notice that for the datasets in which the masking variant performs best, the RD reaches high absolute values. The maximum (50.60%) is obtained for the data type "VWC" of the dataset SST, with algorithm PCA, and error parameter $e = 30$, which is highlighted in Figure 3.1 with a red circle.

The experimental results presented in this section suggest that if we were interested in compressing a dataset with many gaps, we would benefit from using the masking variant of an algorithm, a_M. However, even if the dataset didn't have any gaps, the performance would not be significantly worse than that obtained by using the non-masking variant of the algorithm, a_{NM}. Therefore, since masking variants are, in general, more robust in this sense, in the sequel we focus on the set of variants V^* that we define next.

Definition 3.2.2. We denote by V^* the set of all the masking algorithm variants a_M for $a \in A$.

Notice that V^* includes a single variant for each algorithm. Therefore, in what follows we sometimes refer to the elements of V^* simply as algorithms.

3.3 Window Size Parameter

In this section, we analyze the extent to which the window size parameter impacts on the performance of the coding algorithms. For these experiments we consider the set of algorithm variants V_W^*, which is obtained from V^* by discarding algorithm SF, which doesn't have a window size parameter (recall this information from Table 2.1). Also, we only consider the four datasets that consist of multiple files, i.e. IRKIS, SST, ADCP and Solar (recall this information from Table 1.1). For each file, we compare the compression performance when using the OWS for the dataset, as defined in (3.6), and the LOWS for the file, defined next.

Definition 3.3.1. The *local optimal window size (LOWS)* of a coding algorithm variant $a_v \in V_W^*$, and a threshold parameter $e \in E$, for the data type z of a certain file f is given by

$$LOWS(a_v, e, z, f) = \underset{w \,\in\, W}{\arg\min} \left\{ CR(c_{<a_v,w,e>}, z, f) \right\}, \tag{3.7}$$

where we break ties in favor of the smallest window size.

For each data type z of each dataset d, and each file $f \in F(d, z)$, coding algorithm variant $a_v \in V_W^*$, and threshold parameter $e \in E$, we calculate the RD between $c_{<a_v,w^*_{global},e>}$ and $c_{<a_v,w^*_{local},e>}$, as defined in (3.2), where $w^*_{global} = \text{OWS}(a_v, e, z, d)$ and $w^*_{local} = \text{LOWS}(a_v, e, z, f)$. In what follows, we denote the OWS and the LOWS as w^*_{global} and w^*_{local}, respectively.

As an example, in figures 3.3 and 3.4 we show w^*_{global}, w^*_{local}, and the RD between $c_{<a_v,w^*_{global},e>}$ and $c_{<a_v,w^*_{local},e>}$, as a function of the threshold parameter e, obtained for the data type z = "VWC", for two different files of the dataset d = IRKIS. Figure 3.3 shows the results for the file f = "vwc_1202.dat.csv", and Figure 3.4 shows the results for f = "vwc_1203.dat.csv". Observe that the values of w^*_{global} are the same for both figures, which is expected, since both are obtained from the same data type of the same dataset.

In Figure 3.3 we notice, for instance, that for algorithm APCA the OWS and LOWS values match for every threshold parameter e, except 3 and 10. The OWS is larger than the LOWS when $e = 3$, but it is smaller when $e = 10$. In these two cases, the RD values are 1.52% and 1.76%, respectively. In Figure 3.4 we observe that in every case the OWS is larger than or equal to the LOWS . We highlight the marker for the maximum RD value (10.68%) obtained for all the tested CAIs, and we further comment on this point in the remaining of the section. Notice that in both figures the RD is non-negative in every plot, which makes sense, since the CR obtained with the OWS can never be lower than the CR obtained with the LOWS .

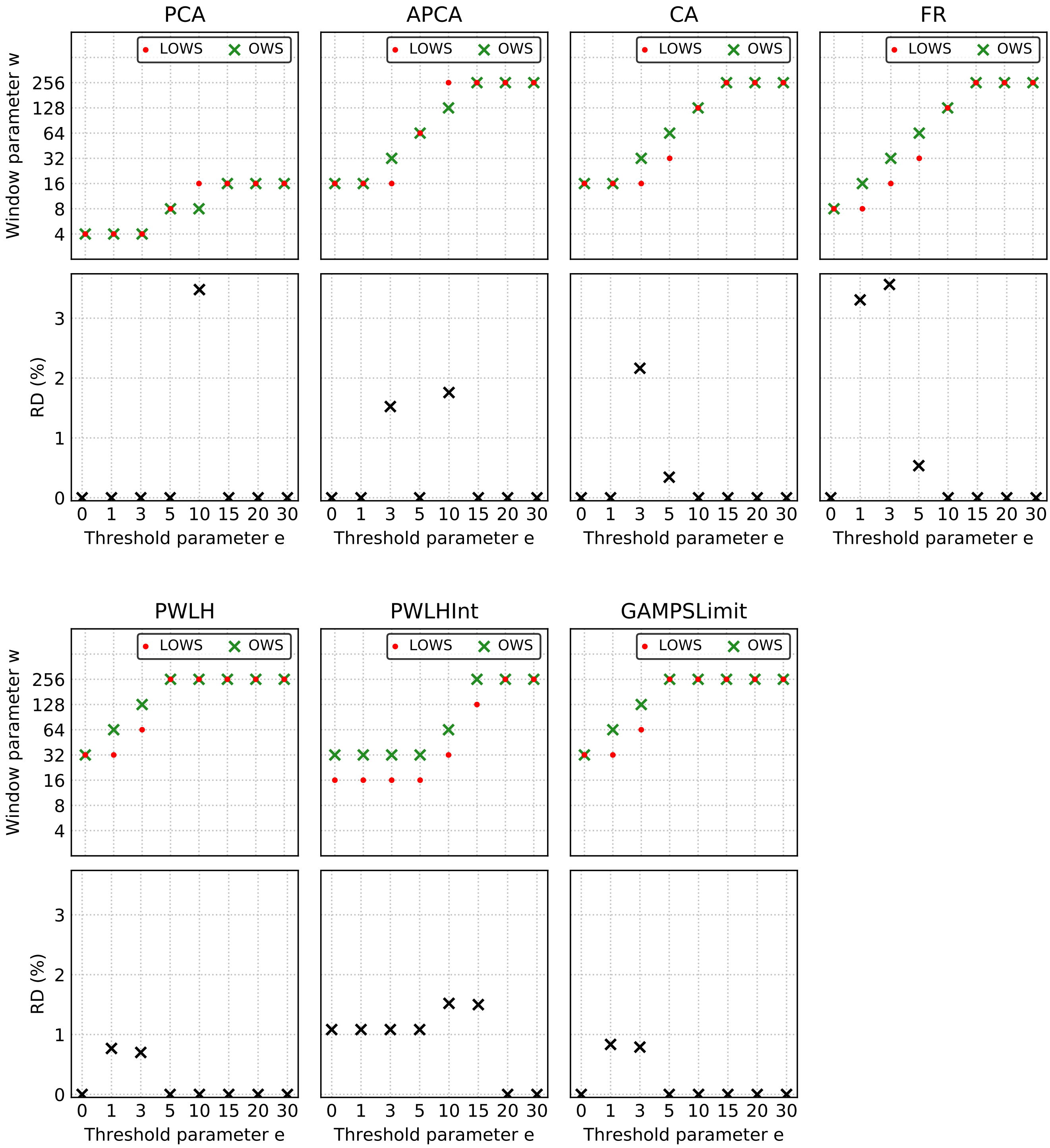


FIGURE 3.3: Plots of w^*_{global}, w^*_{local}, and the RD between $c_{<a_v,w^*_{global},e>}$ and $c_{<a_v,w^*_{local},e>}$, as a function of the threshold parameter e, obtained for the data type "VWC" of the file "vwc_1202.dat.csv" of the dataset IRKIS.

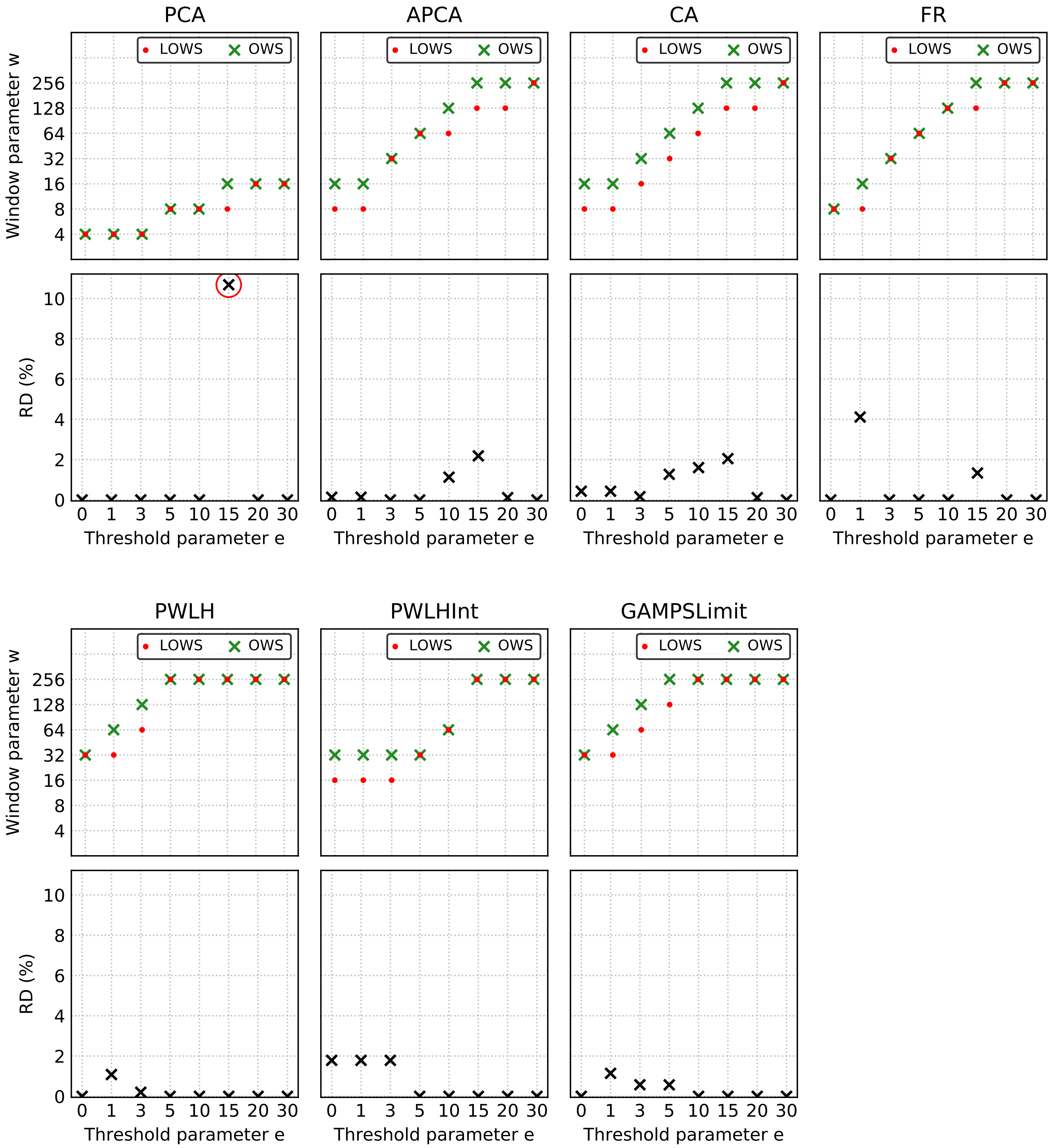


FIGURE 3.4: Plots of w^*_{global}, w^*_{local}, and the RD between $c_{<a_v, w^*_{global}, e>}$ and $c_{<a_v, w^*_{local}, e>}$, as a function of the threshold parameter e, obtained for the data type "VWC" of the file "vwc_1203.dat.csv" of the dataset IRKIS. In the RD plot for algorithm PCA we highlight with a red circle the marker for the maximum value (10.68%) obtained for all the tested CAIs.

We analyze the experimental results to evaluate the impact of using the OWS instead of the LOWS on the compression performance of the tested coding algorithms. For each algorithm, we iterate through each threshold parameter, and each data type of each file, and we calculate the RD between the CAI with the OWS and the CAI with the LOWS . Since we consider 8 threshold parameters and there are 13 files with a single data type and 4 files with 3 different data types each, for each algorithm we compare a total of $8 \times (13 + 4 \times 3) = 200$ pairs of CAIs. Table 3.2 summarizes the results of these comparisons, aggregated by algorithm and the range to which the RD belongs.

	RD (%) Range				
Algorithm	**0**	**(0,1]**	**(1,2]**	**(2,5]**	**(5,11]**
PCA	186 (93%)	4 (2%)	3 (1.5%)	2 (1%)	5 (2.5%)
APCA	174 (87%)	13 (6.5%)	7 (3.5%)	6 (3%)	0
CA	172 (86%)	16 (8%)	6 (3%)	6 (3%)	0
FR	171 (85.5%)	14 (7%)	8 (4%)	7 (3.5%)	0
PWLH	184 (92%)	13 (6.5%)	3 (1.5%)	0	0
PWLHInt	173 (86.5%)	9 (4.5%)	13 (6.5%)	4 (2%)	1 (0.5%)
GAMPSLimit	182 (91%)	16 (8%)	2 (1%)	0	0
Total	1,242 (88.7%)	85 (6.1%)	42 (3%)	25 (1.8%)	6 (0.4%)

Table 3.2: RD between the OWS and LOWS variants of each CAI.
The results are aggregated by algorithm and the range to which the RD belongs.

For example, consider the results for algorithm CA, in the third row. The first column indicates that the RD is equal to 0 for exactly 172 (86%) of the 200 evaluated pairs of CAIs for that algorithm. The second column reveals that for 16 pairs of CAIs (8%), the RD takes values greater than 0 and less than or equal to 1%. The remaining three columns cover other ranges of RD values. Notice that for every row (except the last one), the values add up to a total of 200, since we compare exactly 200 pairs of CAIs for each algorithm.

The last row of Table 3.2 is obtained by adding the values of the previous rows, which combines the results for all algorithms. We notice that in 88.7% of the total number of evaluated pairs of CAIs, the RD is equal to 0. In these cases, in fact, the OWS and the LOWS coincide. In 97.8% of the cases, the RD is less than or equal to 2%. This means that, for the vast majority of CAI pairs, either the OWS and the LOWS match or they yield roughly the same compression performance. This result suggests that we could fix the window size parameter in advance, for example by optimizing over a training set, without compromising the performance of the coding algorithm. This is relevant, since calculating the LOWS for a file is, in general, computationally expensive.

We notice that there are only 6 cases (0.4%) in which the RD falls in the range (5, 11], most of which (5 cases) involve the algorithm PCA. The maximum value taken by RD (10.68%) is obtained for the data type "VWC" of the file "vwc_1203.dat.csv" of the dataset SST, with algorithm PCA, and error parameter $e = 15$. In Figure 3.4 we highlight this maximum value with a red circle. In this case, the OWS is 16 and the LOWS is 8. According to these results, the performance of algorithm PCA seems to be more sensible to the window size parameter than the rest of the algorithms. Except for these few cases, we observe that, in general, the impact of using the OWS instead of the LOWS on the compression performance of coding algorithms is rather small. Therefore, in the following section, in which we compare the algorithms performance, we always use the OWS.

3.4 Algorithms Performance

In this section, we compare the compression performance of the coding algorithms presented in Chapter 2, by encoding the various datasets introduced in Chapter 1. We begin by comparing the algorithms among each other and later we compare them with gzip, a popular lossless compression algorithm. We analyze the performance of the algorithms on complete datasets (not individual files), so we always apply definitions 3.1.6–3.1.9. Following the results obtained in sections 3.2 and 3.3, we only consider the masking variants of the evaluated algorithms (i.e. set V^*), and we always set the window size parameter to the OWS (recall Definition 3.2.1).

For each data type z of each dataset d, and each coding algorithm variant $a_v \in V^*$ and threshold parameter $e \in E$, we calculate the CR of $c_{<a_v, w^*_{global}, e>}$, as defined in (3.4), where $w^*_{global} = \text{OWS}(a_v, e, z, d)$. The following definition is useful for analyzing which CAI obtains the best compression result for a specific data type.

Definition 3.4.1. Let z be a data type of a certain dataset d, and let $e \in E$ be a threshold parameter. We denote by $c^b(z, d, e)$ the *best CAI* for z, d, e, and define it as the CAI that minimizes the CR among all the CAIs in CI, i.e.,

$$c^b(z, d, e) = \underset{c\ \in CI}{\arg\min} \left\{ CR(c_{<a_v, w^*_{global}, e>}, z, d) \right\}. \tag{3.8}$$

When $c^b(z, d, e) = c_{<a^b_v, w^{b*}_{global}, e>}$, we refer to a^b and w^{b*}_{global} as the *best coding algorithm* and the *best window size* for z, d, e, respectively.

Our experiments include a total of 21 data types, in 8 datasets. As an example, in Figure 3.5 we show the CR and the window size parameter w^*_{global}, as a function of the threshold parameter, obtained for each algorithm, for the data type “SST” of the dataset ElNino. For each threshold parameter $e \in E$, we use blue circles to highlight the markers for the minimum CR value and the best window parameter (in the respective plots corresponding to the best algorithm). For instance, for $e = 0$, the best CAI achieves a CR equal to 0.33 using algorithm PCA with a window size of 256. So in this case, algorithm PCA is the best coding algorithm, and 256 is the best window size. For the remaining seven values for the threshold parameter, the blue circles indicate that in every case the best algorithm is APCA, and the best window size ranges from 4 up to 32.

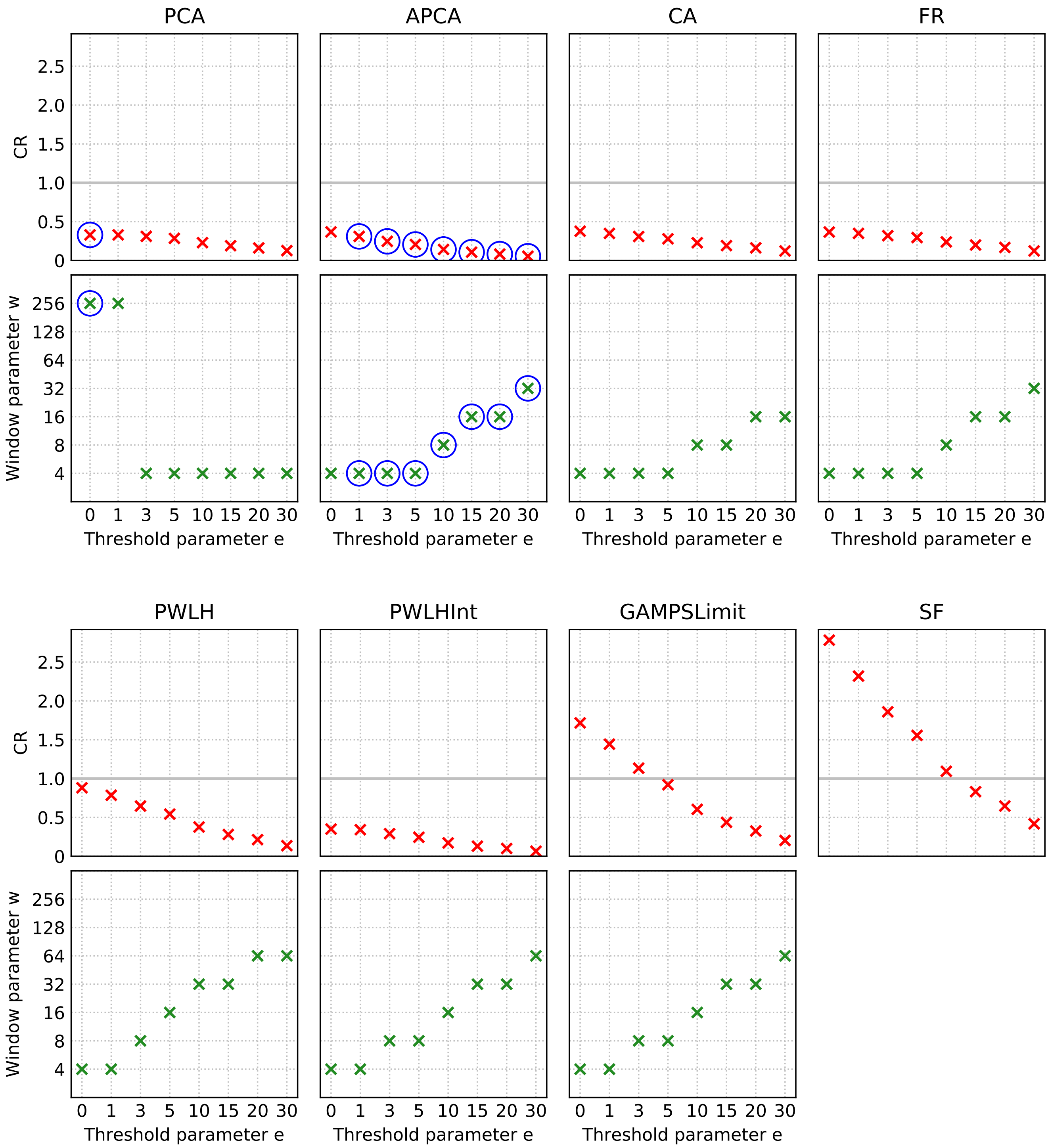


FIGURE 3.5: CR and window size parameter plots for every algorithm, for the data type "SST" of the dataset ElNino. For each threshold parameter $e \in E$, we use blue circles to highlight the markers for the minimum CR value and the best window size parameter (in the respective plots corresponding to the best algorithm)

Table 3.3 summarizes the compression performance results obtained by the evaluated coding algorithms, for each data type of each dataset. Each row contains information relative to certain data type. For example, the 13th row shows summarized results for the data type "SST" of the dataset ElNino, which are presented in more detail in Figure 3.5. For each threshold, the first column shows the CR obtained by the best CAI, the second column shows the base-2 logarithm of its window size parameter, and the cell color identifies the best algorithm.

PCA	APCA	FR

		e = 0		e = 1		e = 3		e = 5		e = 10		e = 15		e = 20		e = 30	
Dataset	Data Type	CR	w	CR	w	CR	w	CR	w	CR	w	CR	w	CR	w	CR	w
IRKIS	VWC	0.20	4	0.18	4	0.12	5	0.07	6	0.03	7	0.02	8	0.02	8	0.01	8
SST	SST	0.61	8	0.28	3	0.14	5	0.09	6	0.05	7	0.03	8	0.02	8	0.02	8
ADCP	Vel	0.68	8	0.68	8	0.67	2	0.61	2	0.48	2	0.41	2	0.35	3	0.26	3
Solar	GHI	0.78	2	0.76	3	0.71	4	0.67	4	0.59	4	0.52	4	0.47	4	0.38	4
	DNI	0.76	2	0.72	4	0.66	4	0.61	4	0.54	4	0.49	4	0.43	4	0.36	4
	DHI	0.78	2	0.77	2	0.72	4	0.68	4	0.60	4	0.54	4	0.48	4	0.39	4
ElNino	Lat	0.16	4	0.16	4	0.16	4	0.15	4	0.12	4	0.10	5	0.09	5	0.06	6
	Long	0.17	3	0.17	4	0.13	4	0.12	5	0.09	6	0.07	6	0.05	7	0.02	8
	Z. Wind	0.31	8	0.31	8	0.31	8	0.31	8	0.27	2	0.24	2	0.21	2	0.16	3
	M. Wind	0.31	8	0.31	8	0.31	8	0.31	8	0.29	2	0.26	2	0.23	2	0.19	2
	Humidity	0.23	8	0.23	8	0.23	8	0.23	8	0.21	2	0.18	2	0.16	2	0.13	2
	AirTemp	0.33	8	0.33	8	0.30	2	0.27	2	0.22	2	0.19	3	0.17	3	0.13	4
	SST	0.33	8	0.31	2	0.25	2	0.21	2	0.14	3	0.11	4	0.08	4	0.05	5
Hail	Lat	1.00	8	1.00	8	0.90	2	0.83	2	0.71	2	0.65	3	0.57	3	0.47	3
	Long	1.00	8	1.00	8	0.86	2	0.78	2	0.65	2	0.55	3	0.49	3	0.39	4
	Size	0.81	2	0.81	2	0.81	2	0.81	2	0.81	2	0.81	2	0.81	2	0.64	3
Tornado	Lat	1.00	8	0.85	2	0.71	2	0.65	2	0.54	3	0.47	3	0.42	4	0.33	4
	Long	1.00	8	0.82	2	0.65	2	0.58	3	0.46	3	0.40	4	0.35	4	0.28	4
Wind	Lat	1.00	8	1.00	8	0.89	2	0.81	2	0.70	2	0.62	3	0.56	3	0.47	3
	Long	1.00	8	0.95	2	0.80	2	0.73	2	0.62	3	0.54	3	0.49	3	0.40	4
	Speed	0.65	4	0.44	3	0.26	6	0.17	7	0.16	5	0.12	6	0.10	6	0.08	6

TABLE 3.3: Compression performance of the best evaluated coding algorithm, for various error thresholds on each data type of each dataset. Each row contains information relative to certain data type. For each threshold, the first column shows the minimum CR, and the second column shows the base-2 logarithm of the window size parameter for the best algorithm (the one that achieves the minimum CR), which is identified by a certain cell color described in the legend above the table.

We observe that there are only three algorithms (PCA, APCA, and FR) which are used by the best CAI for at least one of the 168 possible data type and threshold parameter combinations. Algorithm APCA is used in exactly 134 combinations (80%), including every case in which $e \geq 10$, and most of the cases in which $e \in [1, 3, 5]$. PCA is used in 31 combinations (18%), including most of the lossless cases, while FR is the best algorithm in only 3 combinations (2%), all of them for data type "Speed" of the dataset Wind.

Since there is not a single algorithm that obtains the best compression performance for every data type, it is useful to analyze how much is the RD between the best algorithm and the rest, for every experimental combination. With that in mind, next we define a pair of metrics.

Definition 3.4.2. The *maximum RD* (*maxRD*) of a coding algorithm $a \in A$ for certain threshold parameter $e \in E$ is given by

$$maxRD(a, e) = \max_{z,d} \left\{ RD(c^b(z, d, e), c_{<a_v, w^*_{global}, e>}) \right\}, \tag{3.9}$$

where the maximum is taken over all the combinations of data type z and dataset d, and we recall that $c^b(z, d, e)$ is the best CAI for z, d, e.

The maxRD metric is useful for assessing the compression performance of a coding algorithm a on the set of data types as a whole. Notice that maxRD is always non-negative. A satisfactory result (i.e. close to zero) can only be obtained when a achieves a good compression performance *for every data type*. In other words, bad compression performance *on a single data type* yields a poor result for the maxRD metric altogether. When maxRD is equal to zero, a achieves the best compression performance for every combination. Analyzing the results in Table 3.3, we observe that maxRD(APCA, e) $= 0$ for every $e \geq 10$. Since the best algorithm is unique for every combination (i.e. exactly one algorithm obtains the minimum CR in every case), it is also true that, when $a \neq$ APCA, maxRD$(a, e) > 0$ for every $e \geq 10$.

Definition 3.4.3. The *minmax RD* (*minmaxRD*) for certain threshold parameter $e \in E$ is given by

$$minmaxRD(e) = \min_{a \in A} \left\{ maxRD(a, e) \right\}, \tag{3.10}$$

and we refer to arg $\min_{a \in A}$ as the *minmax coding algorithm* for e.

Again, minmaxRD is always always non-negative. Notice that minmaxRD$(e) = 0$ for certain e, if and only if there exists a minmax coding algorithm a such that maxRD$(a, e) = 0$. Continuing the analysis from the previous paragraph, it should be clear that APCA is the minmax coding algorithm for every $e \geq 10$, since maxRD(APCA, e) $= 0$ for every $e \geq 10$.

Table 3.4 shows the maxRD(a, e) obtained for every pair of coding algorithm variant $a_v \in V^*$ and threshold parameter $e \in E$. For each e, the cell corresponding to the minmaxRD(e) value (i.e. the minimum value in the column) is highlighted.

	maxRD (%)							
Algorithm	e = 0	e = 1	e = 3	e = 5	e = 10	e = 15	e = 20	e = 30
PCA	40.52	42.28	53.11	62.01	71.73	75.33	77.21	80.28
APCA	33.25	15.64	9.00	29.96	0	0	0	0
CA	38.28	38.28	54.68	63.12	65.44	72.94	77.21	81.84
PWLH	73.46	72.93	72.52	82.14	83.24	86.86	88.94	91.19
PWLHInt	29.72	34.00	49.94	68.95	76.68	69.96	74.72	79.89
FR	48.75	49.85	52.21	52.70	54.82	55.35	54.48	64.72
SF	92.46	92.23	92.16	92.11	91.68	91.24	90.95	91.33
GAMPSLimit	85.84	85.73	84.37	83.92	83.02	83.00	82.88	82.22

TABLE 3.4: maxRD(a, e) obtained for every pair of coding algorithm variant $a_v \in V^*$ and threshold parameter $e \in E$. For each e, the cell corresponding to the minmaxRD(a) value is highlighted.

In the lossless case, PWLHInt is the minmax coding algorithm, with minmaxRD being equal to 29.72%. This value is rather high, which means that none of the considered algorithms achieves a CR that is close to the minimum simultaneously *for every data type*. Recalling the results from Table 3.3 we notice that $e = 0$ is the only threshold parameter value for which the minmax coding algorithm doesn't obtain the minimum CR in any combination. In other words, when

$e = 0$, PWLHInt is the algorithm that minimizes the RD with the best algorithm among every data type, even though it itself is not the best algorithm for any data type.

When $e \in [1, 3, 5]$, the minmax coding algorithm is always APCA, and the minmaxRD values are 15.64%, 9.00% and 29.96%, respectively. Again, these values are fairly high, so we would select the most convenient algorithm depending on the data type we want to compress. Notice that in the closest case (algorithm FR for $e = 5$), the second best maxRD (52.70%) is about 75% larger than the minmaxRD, which is a much bigger difference than in the lossless case.

When $e \geq 10$, the minmax coding algorithm is also always APCA, but in these cases the minmaxRD values are always 0. In the closest case (algorithm FR for $e = 20$) the second best maxRD is 54.48%. If we wanted to compress any data type with any of these threshold parameter values, we would pick algorithm APCA, since according to our experimental results, it always obtains the best compression results with a significant difference over the remaining algorithms.

3.4.1 Comparison with algorithm gzip

In this subsection we consider the results obtained by the general purpose compression algorithm gzip [23]. This algorithm only operates in lossless mode (i.e. the threshold parameter can only be $e = 0$), and it doesn't have a window size parameter w. Therefore, for each data type z of each dataset d, we have a unique CAI (and obtain a unique CR value) for gzip.

In all our experiments with gzip we perform a column-wise compression of the dataset files, which, in general, yields a much better performance than a row-wise compression. This is due to the fact that in most of our datasets, there is a greater degree of temporal than spatial correlation between the signals. All the reported results are obtained with the "`--best`" option of gzip, which targets compression performance optimization [24].

Table 3.5 summarizes the compression performance results obtained by gzip and the other evaluated coding algorithms, for each data type of each dataset. Similarly to Table 3.3, each row contains information relative to a certain data type, and for each threshold, the first column shows the CR obtained by the best CAI, the second column shows the base-2 logarithm of its window size parameter (when applicable), and the cell color identifies the best algorithm. Notice that, for $e > 0$ we compare the gzip lossless result with the results obtained by lossy algorithms.

We observe that algorithm gzip obtains the best compression results in 36 (21%) of the 168 possible data type and threshold parameter combinations. Algorithms APCA, PCA, and FR now obtain the best results in exactly 106 (63%), 23 (14%), and 3 (2%) of the total combinations, respectively. Algorithm APCA is still the best algorithm for most of the cases in which $e \geq 3$. However, now there is no value of e for which APCA outperforms the rest of the algorithms for every data type, since gzip is the best algorithm for at least one data type in every case. In particular, gzip obtains the best compression results for the data type "Size" of the dataset Hail for every e. We also observe that gzip obtains the best relative results against the other algorithms for smaller values of e, which is expected, since lossy algorithms performance improves for larger values of e. However, even for $e = 0$, gzip only outperforms the rest of the algorithms in about a half (10 out of 21) of the data types.

GZIP	PCA	APCA	FR

		e = 0		e = 1		e = 3		e = 5		e = 10		e = 15		e = 20		e = 30	
Dataset	Data Type	CR	w	CR	w	CR	w	CR	w	CR	w	CR	w	CR	w	CR	w
IRKIS	VWC	0.13		0.13		0.12	5	0.07	6	0.03	7	0.02	8	0.02	8	0.01	8
SST	SST	0.52		0.28	3	0.14	5	0.09	6	0.05	7	0.03	8	0.02	8	0.02	8
ADCP	Vel	0.61		0.61		0.61		0.61	2	0.48	2	0.41	2	0.35	3	0.26	3
Solar	GHI	0.69		0.69		0.69		0.67	4	0.59	4	0.52	4	0.47	4	0.38	4
	DNI	0.67		0.67		0.66	4	0.61	4	0.54	4	0.49	4	0.43	4	0.36	4
	DHI	0.61		0.61		0.61		0.61		0.60	4	0.54	4	0.48	4	0.39	4
ElNino	Lat	0.08		0.08		0.08		0.08		0.08		0.08		0.08		0.06	6
	Long	0.07		0.07		0.07		0.07		0.07		0.07	6	0.05	7	0.02	8
	Z. Wind	0.31	8	0.31	8	0.31	8	0.31	8	0.27	2	0.24	2	0.21	2	0.16	3
	M. Wind	0.31	8	0.31	8	0.31	8	0.31	8	0.29	2	0.26	2	0.23	2	0.19	2
	Humidity	0.23	8	0.23	8	0.23	8	0.23	8	0.21	2	0.18	2	0.16	2	0.13	2
	AirTemp	0.33	8	0.33	8	0.30	2	0.27	2	0.22	2	0.19	3	0.17	3	0.13	4
	SST	0.32		0.31	2	0.25	2	0.21	2	0.14	3	0.11	4	0.08	4	0.05	5
Hail	Lat	1.00	8	1.00	8	0.90	2	0.83	2	0.71	2	0.65	3	0.57	3	0.47	3
	Long	1.00	8	1.00	8	0.86	2	0.78	2	0.65	2	0.55	3	0.49	3	0.39	4
	Size	0.37		0.37		0.37		0.37		0.37		0.37		0.37		0.37	
Tornado	Lat	1.00	8	0.85	2	0.71	2	0.65	2	0.54	3	0.47	3	0.42	4	0.33	4
	Long	1.00	8	0.82	2	0.65	2	0.58	3	0.46	3	0.40	4	0.35	4	0.28	4
Wind	Lat	1.00	8	1.00	8	0.89	2	0.81	2	0.70	2	0.62	3	0.56	3	0.47	3
	Long	1.00	8	0.95	2	0.80	2	0.73	2	0.62	3	0.54	3	0.49	3	0.40	4
	Speed	0.65	4	0.44	3	0.26	6	0.17	7	0.16	5	0.12	6	0.10	6	0.08	6

TABLE 3.5: Compression performance of the best evaluated coding algorithm, for various error thresholds on each data type of each dataset, including the results obtained by gzip. Each row contains information relative to certain data type. For each threshold, the first column shows the minimum CR, and the second column shows the base-2 logarithm of the window size parameter for the best algorithm (the one that achieves the minimum CR), which is identified by a certain cell color described in the legend above the table. Algorithm gzip doesn't have a window size parameter, so the cell is left blank in these cases.

Similarly to Table 3.4, Table 3.6 shows the maxRD(a, e) obtained for every pair of coding algorithm variant $a_v \in V^* \cup \{\text{gzip}\}$ and threshold parameter $e \in E$. For each e, the cell correspondent to the minmaxRD(e) value is highlighted.

We observe that, for every e, the minmaxRD values are rather high, the minimum being 26.58% (algorithm gzip for $e = 0$). We conclude that none of the considered algorithms achieves a competitive CR *for every data type*, and the selection of the most convenient algorithm depends on the specific data type we are interested in compressing.

gzip is the minmax coding algorithm when $e \in [0, 1]$, and in both cases the minmaxRD values are rather high, i.e. 26.58% and 47.98%, respectively. APCA remains the minmax coding algorithm for every $e \geq 3$, but its minmaxRD values are now not only always greater than zero, but also quite high, ranging from 42.92% ($e = 30$) up to 54.42% ($e = 3$). This implies that there exist some data types for which the RD between the APCA and gzip CAIs is considerable, which means that, if we had the possibility of selecting gzip as a compression algorithm, APCA would no longer be the obvious choice for compressing any data type when $e \geq 10$, as we had concluded in the previous section.

	maxRD (%)							
Algorithm	e = 0	e = 1	e = 3	e = 5	e = 10	e = 15	e = 20	e = 30
GZIP	26.58	47.98	73.79	82.94	91.10	93.94	95.40	96.69
PCA	73.94	73.55	68.28	67.23	71.73	75.33	77.21	80.28
APCA	59.11	58.39	54.42	54.41	54.40	54.38	54.38	42.92
CA	73.83	73.46	69.31	68.30	65.44	72.94	77.21	81.84
PWLH	87.53	87.46	87.37	87.27	87.02	86.86	88.94	91.19
PWLHInt	71.26	71.01	70.71	68.95	76.68	69.96	74.72	79.89
FR	76.46	76.12	72.78	71.97	67.37	64.35	64.05	64.72
SF	96.31	96.13	96.09	95.96	95.87	95.74	95.64	95.05
GAMPSLimit	91.51	91.51	91.51	91.51	91.50	91.50	91.50	88.85

TABLE 3.6: maxRD(a, e) obtained for every pair of coding algorithm variant $a_v \in V^* \cup \{\text{gzip}\}$ and threshold parameter $e \in E$. For each e, the cell corresponding to the minmaxRD(a) value is highlighted.

3.5 Conclusions

In conclusion, our experimental results indicate that none of the implemented coding algorithms obtains a satisfactory compression performance in every scenario. This means that selection of the best algorithm is heavily dependent on the data type to be compressed and the error threshold that is allowed. In addition, we have shown that, in some cases, even a general compression algorithm such as gzip can outperform our implemented algorithms. In general, according to our results, algorithms APCA and gzip achieve better compression results for larger error thresholds, while PCA, APCA, FR and gzip are preferred for lower thresholds. Therefore, if one wishes to compress certain data type, our recommended way for choosing the appropriate algorithm is to select the best algorithm for said data type according to Table 3.6.

In our research we have also compared the compression performance of the coding algorithms' masking and non-masking variants. The experimental results show that on datasets with few or no gaps both variants have a similar performance, while on datasets with many gaps the masking variant always performs better, sometimes achieving a significative difference. We concluded that the masking variant of a coding algorithm is preferred, since it is more robust and performs better in general.

In addition, we have studied the extent to which the window size parameter **impacts/affects** the compression performance of the coding algorithms. We analyzed the compression results obtained when using optimal global and local window sizes. The experimental results reveal that the **impact/effect** of using the optimal global window size instead of the optimal local window size for each file is rather small.

3.6 Future Work (TODO)

Some ideas:

- Consider non-linear models (e.g. Chebyshev Approximation)
- Consider new datasets
- Consider other metrics (e.g. RMSE)
- Investigate why certain algorithms perform better on certain data types

- Create universal coder, with every algorithm as a subroutine

Bibliography

[1] EnviDat - IRKIS Soil moisture measurements Davos. https://www.envidat.ch/dataset/soil-moisture-measurements-davos, 2019. [Accessed 3 September 2020].

[2] NOAA - TAO Data Download. https://tao.ndbc.noaa.gov/tao/data_download/search_map.shtml, 2016. [Accessed 3 September 2020].

[3] SolarAnywhere - Data. https://data.solaranywhere.com/Data, 2020. [Accessed 3 September 2020].

[4] El Nino Data Set. https://archive.ics.uci.edu/ml/datasets/El+Nino, 1999. [Accessed 3 September 2020].

[5] Kaggle - Storm Prediction Center. https://www.kaggle.com/jtennis/spctornado, 2016. [Accessed 3 September 2020].

[6] N.Q.V. Hung, H. Jeung, and K. Aberer. An Evaluation of Model-Based Approaches to Sensor Data Compression. *IEEE Transactions on Knowledge and Data Engineering*, 25(11):2434–2447, 2013.

[7] T. Bose, S. Bandyopadhyay, S. Kumar, A. Bhattacharyya, and A. Pal. Signal Characteristics on Sensor Data Compression in IoT - An Investigation. *2016 13th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, pages 1–6, 2016.

[8] I. Lazaridis and S. Mehrotra. Capturing Sensor-Generated Time Series with Quality Guarantees. *Proc. ICDE*, pages 429–440, 2003.

[9] E. Keogh, K. Chakrabarti, S. Mehrotra, and M. Pazzani. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. *ACM Transactions on Database Systems*, 27(2):188–228, 2002.

[10] G.E. Williams. Critical Aperture Convergence Filtering and Systems and Methods Thereof. *US Patent 7,076,402*, Jul. 11, 2006.

[11] C. Buragohain, N. Shrivastava, and S. Suri. Space Efficient Streaming Algorithms for the Maximum Error Histogram. *Proc. ICDE*, pages 1026–1035, 2007.

[12] H. Elmeleegy, A.K. Elmagarmid, E. Cecchet, W.G. Aref, and W. Zwaenepoel. Online Piecewise Linear Approximation of Numerical Streams with Precision Guarantees. *Proc. VLDB Endowment*, 2(1):145–156, 2009.

[13] J.A.M. Heras and A. Donati. Fractal Resampling: time series archive lossy compression with guarantees. *PV 2013 Conference*, 2013.

[14] S. Gandhi, S. Nath, S. Suri, and J. Liu. GAMPS: Compressing Multi Sensor Data by Grouping and Amplitude Scaling. *Proc. ACM SIGMOD International Conference on Management of Data*, pages 771–784, 2009.

[15] J.J. Rissanen. Generalized Kraft Inequality and Arithmetic Coding. *IBM Journal of Research and Development*, 20(3):198–203, 1976.

[16] Arithmetic Coding + Statistical Modeling = Data Compression. `https://marknelson.us/posts/1991/02/01/arithmetic-coding-statistical-modeling-data-compression.html`, 1991. [Accessed 3 September 2020].

[17] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 2nd edition, 2006.

[18] R.E. Krichevsky and V.K. Trofimov. The performance of universal encoding. *IEEE Transactions on Information Theory*, 27(2):199–207, 1981.

[19] I.H. Witten, R.M. Neal, and J.G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30(6):520–540, 1987.

[20] Data Compression With Arithmetic Coding. `https://marknelson.us/posts/2014/10/19/data-compression-with-arithmetic-coding.html`, 2014. [Accessed 3 September 2020].

[21] Solomon W. Golomb. Run-length encodings (Corresp.). *IEEE Transactions on Information Theory*, 12(3), 1966.

[22] R. Graham. An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. *Information Processing Letters*, 1:132–133, 1972.

[23] GNU Gzip. `https://www.gnu.org/software/gzip/`, 2018. [Accessed 3 September 2020].

[24] GNU Gzip Manual - Invoking gzip. `https://www.gnu.org/software/gzip/manual/html_node/Invoking-gzip.html`, 2018. [Accessed 3 September 2020].