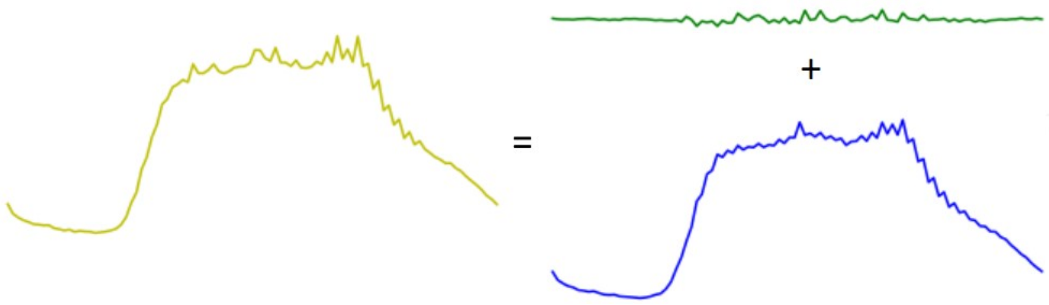
CHALMERS
UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG



Prototype-based compression of time series from telecommunication data

Master's thesis in Computer science and engineering

GABRIEL ALPSTEN
SHARAN SABI

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Master's thesis 2019

Prototype-based compression of time series from telecommunication data

GABRIEL ALPSTEN
SHARAN SABI

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Prototype-based compression of time series from telecommunication data

GABRIEL ALPSTEN
SHARAN SABI

Supervisor: Alexander Schliep, Department of Computer Science and Engineering
Advisor: Ellinor Rånge, Ericsson AB
Examiner: Devdatt Dubhashi, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: The time series in yellow can be modeled as the sum of the prototype in blue and the residual in green.

Typeset in LaTeX
Gothenburg, Sweden 2019

Prototype-based compression of time series from telecommunication data

GABRIEL ALPSTEN
SHARAN SABI
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis explores a technique and use-cases for compressing time series data by the development of prototypes. The methods explored revolve primarily around the idea that a large group of time series can be represented by a much smaller number of prototypes and the calculated residual values between the time series.
We evaluate different clustering techniques to develop prototypes, transform the data by forming residual time series, and explore storage of the transformed dataset to file. This is implemented and compared to two general-purpose compression techniques: Snappy and Zstandard. Our techniques outperform Snappy and Zstandard for non-constant time series, with significant improvements using an error restricted lossy algorithm we present. This thesis further evaluates the use of the compressed format for the prediction of missing data and discusses applications.

Keywords: Compression, Time-Series, Prototypes, Clustering, Prediction

Acknowledgements

We want to express our gratitude towards our supervisors, Alexander Schliep (Chalmers) and Ellinor Rånge (Ericsson), for their continuous, useful feedback and guiding us throughout our thesis work. We are also grateful for advice and ideas from Dan Ståby, Sima Shahsavari, and Mats Bäckström, at Ericsson. In addition, a thank you to Bengt Sjögren and Rosaria D'Alessandro for their help with setting up our environment and making sure that we did not break it. We would also like to acknowledge our examiner Devdatt Dubhashi for his helpful feedback and remarks during the midterm discussion.

Finally, we would like to thank the whole team at Ericsson for keeping our morale high with their constant support, long lunches, and fun afterworks!

Gabriel Alpsten and Sharan Sabi, Gothenburg, June 2019

Contents

List of Figures

List of Tables

1

Introduction

In recent years, the amount of data collected has seen an upward trend. As the use of big-data applications grows, the cost of storage and computation increases. Compression methods have shown promising results to reduce these costs and enable the development of applications that work with big-data. Compression has not only the potential to reduce storage and communication requirements, but also speed up computation by working on more information-dense representations of the data. Our thesis explores compression of time series data generated from radio base stations produced by Ericsson. Finding efficient schemes to compress the data has potentially huge benefits.

In this thesis, we develop a method of *prototype-based compression*. If some time series are similar to many other time series in the dataset, we can form *prototypes* and each series can be defined by the *residuals*, the difference, to one of these prototypes. If a small number of prototypes represent the dataset well, then the prototypes and the residuals together would need smaller storage than the original dataset.

We consider both lossless and lossy storage schemes. Lossy compression has the potential of obtaining much higher compression ratios at the expense of fidelity, but the usefulness depends on the application. Furthermore, the storage format can be decompressed in progressively more accurate steps, enabling applications to read data accurately or read a smaller sized version inaccurately from the same file. We demonstrate this concept with the implementation of a prediction task.

1.1 Presentation of performance metric data

The data used in this study arise from event counters and status measurements tracking the performance of radio access network cells. In the rest of this study, we will refer to these different values as *counters*. These counters are sampled four times per hour, forming time series of 96 samples each day, and we used up to 93 000 such time series. To make this project manageable, we looked at a representative subset of the counters. Figure 1.1 shows how a selection of commonly monitored metric values vary in size. Some of the metrics are almost always zero, or some other constant value, whereas other metrics vary much more. Figure 1.2 show typically occurring types of shapes that the metric time series have. Even if many metrics take on large values that require many bits to be represented, the properties for data collected from the same cell tend to stay very similar over time.

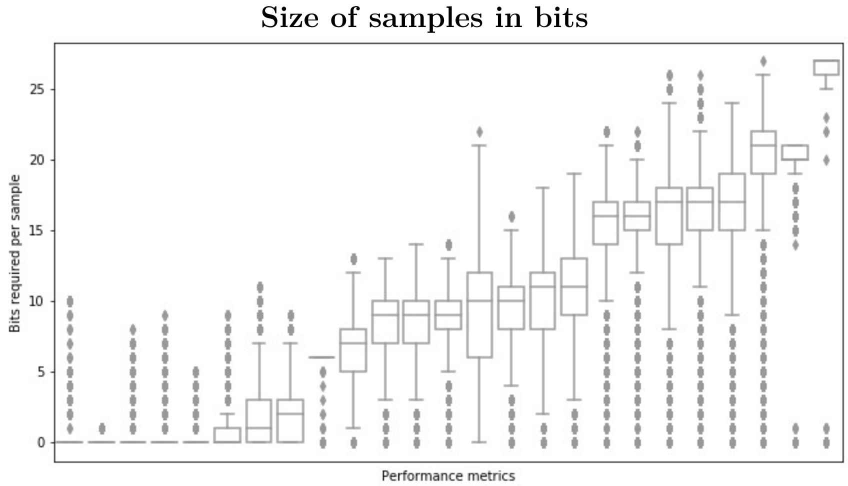
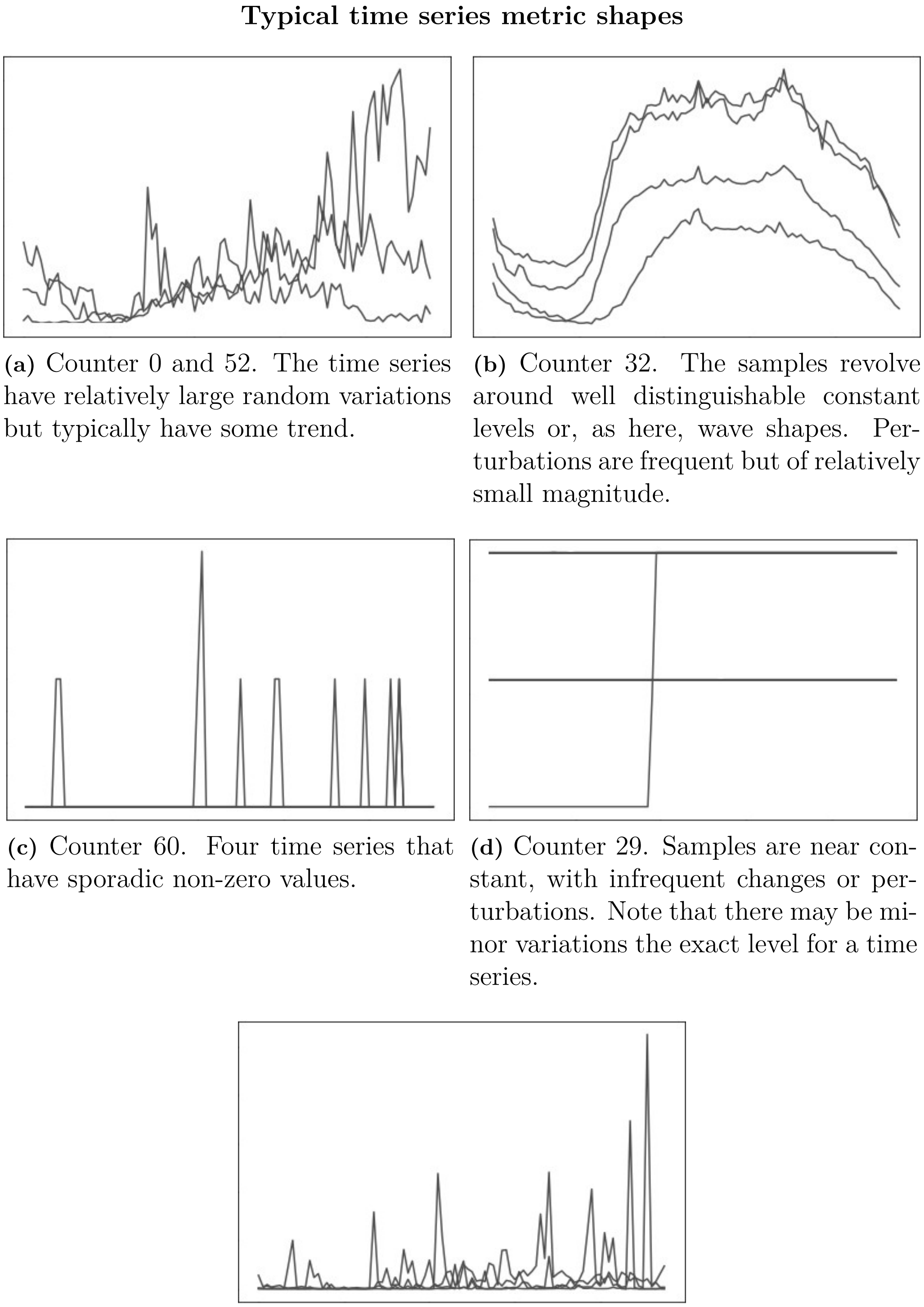


Figure 1.1: Size distribution with outliers for 25 commonly monitored metrics from 7 days of data for 350 random cells. Some counters are generally zero and only infrequently non-zero whereas other metrics vary over large ranges of scales.



Typical time series metric shapes

(a) Counter 0 and 52. The time series have relatively large random variations but typically have some trend.

(b) Counter 32. The samples revolve around well distinguishable constant levels or, as here, wave shapes. Perturbations are frequent but of relatively small magnitude.

(c) Counter 60. Four time series that have sporadic non-zero values.

(d) Counter 29. Samples are near constant, with infrequent changes or perturbations. Note that there may be minor variations the exact level for a time series.

(e) Counter 59. Four time series with random variations and relatively large spikes of perturbations.

Figure 1.2: Four characteristic shapes of metric time series. Counter 0, 52 and 32 are similar as the samples vary with 0 being more random and having more complex trends. Counter 60 and 29 are near constant at either zero or some other level. Counter 59 show relatively large perturbations but neither have sub-sequences of constant values nor distinguishable trends.

1.2 Prototype-based compression

The dataset comprises of 93,969 time series, which are in turn composed of 96 samples over the period of 24 hours. The time series data come from various sources, and vary in magnitude by a large range, depending on the source. The time series plots show that many of them follow similar trajectories over the course of a day.

Clustering can be performed on the dataset to group similar time series, and a prototype can be found from each cluster. The prototype will be a time series, and it may be the cluster center, or the median of the cluster elements. A cluster can now be stored as a prototype along with the *residuals* of time series in the cluster to the prototype. A residual time series is found by calculating the differences of each time series in a cluster to the cluster prototype. The sum of the bits required to store the prototype time series along with the residual time series for each time series in the original dataset, will effectively be less than the original time series. This hypothesis is based on the following assumptions: 1) The same prototype can be used to represent multiple time series 2) The residual time series will have values significantly smaller than the original time series, since we expect the time series to be very similar to the prototype used to find the residuals. This in turn saves storage space since smaller numbers require less no. of bits in a packed storage.

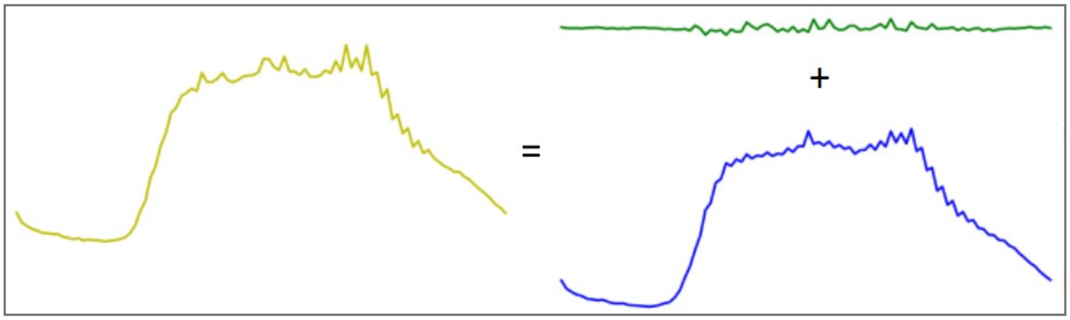

An illustration of a prototype and residual

Figure 1.3: The time series in yellow can be modeled as the sum of the prototype in blue and the residual in green.

The compression strategy proposed above, of storing residual and the prototype instead of the original time series, is a lossless process since the original data can be recreated without any loss. In order to achieve greater savings in storage space, we can also store the residuals in a more compact form at the expense of precision. This idea of lossy compression is also explored further in the project.

In addition to serving as a general purpose compression format for time series, the compressed format is itself useful since pieces of the structure provide a condensed representation of the dataset as a whole. The prototypes, the assignments and other parts of the compressed format may be useful for some computation tasks, answering querries approximately without the need of decompressing and restoring the residuals. For those tasks, in particular, the effective compression will be considerable as the assignment or other stored parameters are much smaller than the 96 values of the whole time series.

1.3 Related works

1.3.1 Principal Component Analaysis

One method used for time series analysis is the statistical method, Principle Component Analysis (PCA). PCA uses an orthogonal transformation to convert a set of observations of possibly correlated variables (entities, each of which takes on various numerical values) into a set of values of linearly uncorrelated variables called principal components [1]. However, this method may fail to model the data without information loss and does not utilize inherent similarities within similar sets of time series.

1.3.2 Neural networks

Neural networks are powerful in many machine learning tasks and can often find dense representations of data by projecting into lower spaces. This may be achieved by training a network to reduce the input data to a few values and then reconstruct the original data (auto encoder). Other techniques involve reusing a network trained for one task by extracting the feed forward state at some layer of the network. However, neural networks are often computationally expensive and are hard to introspect.

1.3.3 Wavelets

Agarwal et al. [2] worked with a time series that stretched over many days. They found that the data from subsequent days and spatially closely located sensors had clear similarities. They utilized these features by arranging the data in a three dimensional matrix with the axes time, day and location, which was then compressed using methods designed for images. They showed that this arrangement could provide higher rates of compression compared to data that was arranged in only two dimensions. A limitation of this approach is, however, that the compression only utilizes the similarities of values that are close to each other in the matrix structure, that are, neighboring values in time, day and location.

1.3.4 Differential encoding

One approach to compressing sequences is *differential encoding.* It is based on predicting values with some simple function and then only storing the residual. If the predictions tend to be close and the residuals are smaller than the original values and can be stored more compactly. One such application is the audio compression algorithm FLAC which calculates predictions as a linear combination of preceding samples and store residuals with the variable length coding scheme Rice [3], typically obtaining a 50-70% compression rate. A recent algorithm Sprintz [4] designed for

IoT sensors do linear predictions and store residuals in small batches of evenly sized numbers. By utilizing run-length encoding of zeros, online parameter learning and a final step Huffman encoding, Sprintz achieved a 50-70 % on a number of tested datasets while maintaining high speed.

Differential encoding of linear predictor functions provide a simple and some times a very effective compression strategy. However, these algorithm tend not to perform well on data that is relatively sporadic [4] and fail to make use of redundancies on a data set level. Since the predictions made for each sample is individual for each time series and sample, it is not straight forward how the compressed format can be useful without decompression.

1.3.5 Clustering based approaches

Vector quantization is a lossy compression method for vectors that tend to perform better than compression methods operating on single values [3]. The vectors are clustered with a variant of k-means. Variants of the algorithm likely exist, but the most common one minimizes the overall error but does not provide any guarantee on the precision of each value in the vectors.

Aghabozorgi et al. [5] presents an approach for clustering time series data with k-means by using prototypes. Their prototypes are cluster centers with number of data points reduced, making them more efficient for querying. However, their approach provides no obvious path to data compression.

Basheer and Sha [6] applied compression techniques to reduce the communication required between wireless sensors. They clustered sub sequences of the data and assigned Huffman codes to each of these clusters, providing an error restricted lossy compression. Although they showed the utility of clustering techniques for time sequences, their technique does not utilize the similarity between different time series.

Ding et al. [7] clustered time series with DBSCAN. They presented a parameterless dimensionality reduction based on the algorithm piecewise aggregate approximation (PAA) and argued that a random sub sample is acceptable for clustering. In addition, they used L^1 norm as similarity metric, which is fast to compute and is resilient to outliers.

1.3.6 General purpose algorithms

There exists a number of general purpose loss-less compression algorithms. One common one is GZIP, but recently some new techniques such as Zstd, Brotli and LZ4 have improved computation and compression ratio trade-offs [4]. These may serve as a good baseline for comparison. Since time series tend not to have exact repetitions of sub-strings, it is a hard challenge for dictionary based compressors to find good patterns. Thus, they don't have the potential to compress very well[4].

1.3.7 Summary

In summary, we have identified several different strategies in the literature relevant for compression of time series where some operate on single time series and some on a data set level. The data set level compression strategies are based on either neural networks or clustering of vectors or time series. However, these referenced works tend to only provide lossy compression. Therefore, some similar clustering techniques may be combined with encoding of residuals to provide a lossless or error restricted lossy compression as described in section 3.8.

1.4 Ethical considerations

The use of telecommunication performance data for machine learning tasks and analysis is not new, but having data that is better compressed may help in doing it at a larger scale. The data is used to find problems (anomalies) in the networks and cells. It also helps in finding out where new equipment is most needed based on performance. Indirectly, the data analysis could help the network operators get better performance from their mobile networks and help them utilize the resources allocated for the purpose better. This study does not deal with any personal data of the end users as the data used are mainly performance metrics from the various machines in the mobile networks, so there are hardly any privacy issues concerning this task. Compression techniques are not new and can be used for many other applications as well, those for which, it is hard for us to analyze our impact.

2

Theory

2.1 Similarity measures for time series

An empirical evaluation of similarity measures for time series data was presented by [8]. We use some of these similarity metrics in the clustering algorithm, and evaluate how well each of them perform in finding a prototype that gives us the smallest residuals. They are:

- L^1 Norm — L^1 norm is the sum of absolute differences of components the vectors in a space. It is also sometimes known as Manhattan distance or Taxicab norm. It assumes equal weightage to all components of the vector. Comparing the similarity of two time series using L^1 norm can be done by adding up the L^1 norm values of corresponding points from start to end of the two series assuming they have equal length. Here, value 0 would indicate that the time series are exactly equal to each other.
- Euclidean Distance — Sometimes referred to as L^2 norm, the euclidean distance between two points p and q is the length of the line segment joining the points p and q. It can be calculated using the cartesian coordinates of the two points in an n dimensional space. If p and q are two points in Euclidean n-space, then the distance (d) from p to q, or from q to p is given by the Pythagorean formula:

  $$d(p,q) = \sqrt{((q_1 - p_1)^2 + (q_2 - p_2)^2 + ... + (q_n - p_n)^2} = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2} \quad (2.1)$$

  Comparing the similarity of two time series using Euclidean Distance can be done by adding up the Euclidean Distance values of corresponding points from start to end of the two series assuming they have equal length. Here, value 0 would indicate that the time series are exactly equal to each other.
- Dynamic Time Warping (DTW) distance — Dynamic Time Warping is another classical approach in time series similarity calculation. It is very useful in detecting similarities when one time series might have values added in a more accelerated or decelerated manner at varying interval but behaves similar in terms of magnitude. It is also useful in comparing time series that are similar

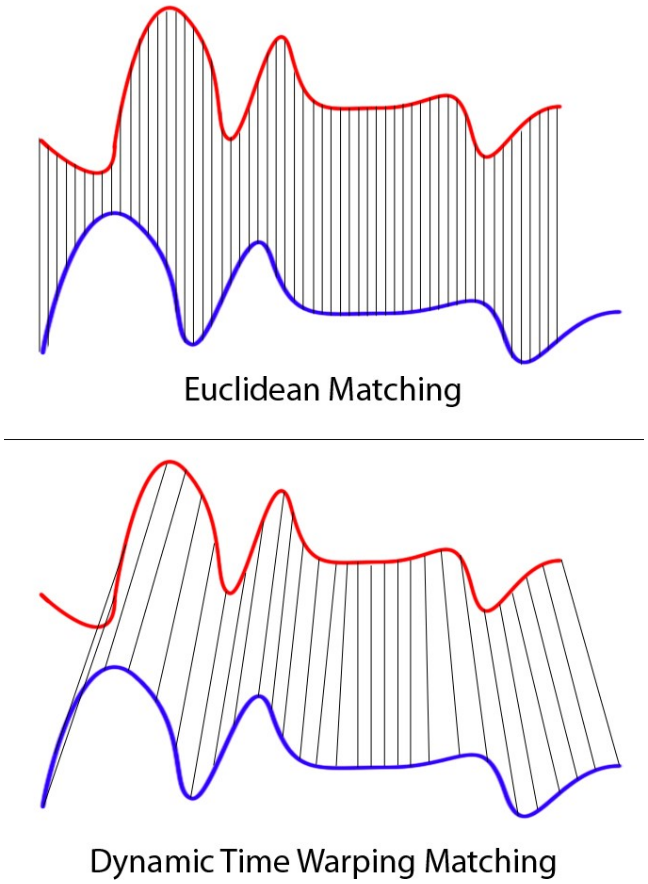


Figure 2.1: Euclidean Distance vs Dynamic Time Warping

but may have varying lengths. In general it is able to compute an optimal match between two time series if 1) Every point in one series is matched with one or more points from the other series 2) The first and the last point of one series must be matched with the first and the last point of the other series respectively 3) The matching of points from the first sequence to the other must be monotonically increasing and vice versa. DTW is often referred to as an elastic dissimilarity measure. It works by finding an optimal alignment between the time series. The cost of this alignment can often be computed by the recursive formula

$$D_{i,j} = f(x_i, y_j) + min D_{i,j-1}, D_{i-1,j}, D_{i-1,j-1} \tag{2.2}$$

for i = 1..M and j = 1..N where M and N are the lengths of the time series.

- Cross Correlation — Cross Correlation is another similarity measure between two time series. It is sometimes known as the sliding dot product. It is very useful in searching one long time series for a shorter time series. It is calculated as a product of the function of the two time series t_1 and t_2, by keeping t_1 series stationary and sliding the other series over it point by point, with the first shift being the first point of t_1 corresponding to the last point of t_2 and the last shift being the last point of t_1 corresponding to the first point of t_2. For two time series t_1 and t_2, the cross-correlation at each shift s can be defined as

$$(t_1 * t_2)(s) = \int_{\infty}^{\infty} t_1(x+s)t_2(x)dx \tag{2.3}$$

where x is the time index ranging from 1..M for t_2 with length M.

When calculating the similarity of two time series, simply comparing sample to sample as is done in L^1 norm and ED may not reveal all intuitive similarities. For example, two time series may have similar features without being perfectly aligned, resulting in a relatively large distance metric. Instead, metrics such as Cross Correlation and DTW match samples more flexibly so that they may discover similarities to a greater set of time series.

2.2 *K*-Means for Time Series

k-means [9] is a clustering method used in Data Analysis. It takes in a dataset of N data-points $(x_1, x_2, ...x_n)$ and forms K clusters $S_1, S_2, ...S_k$ in which each data point is assigned to the cluster with the closest mean which we call the prototype for the cluster. The objective of the algorithm can be described as :

$$\min_{s} \sum_{i=1}^{k} \sum_{x \in S_t} ||x - \mu_i||^2 \tag{2.4}$$

where μ_i is the mean of data points in S_i.

2.2.1 Algorithm

The algorithm assumes an initial set of k means $m_1, m_2, ..., m_k$.

In the assignment step on the t^{th} iteration of the algorithm can be shown as follows, where each data point is assigned to the cluster whose mean has the highest similarity. [10]

$$S_i^{(t)} = \left\{ x_p : \left\| x_p - m_i^{(t)} \right\|^2 \leq \left\| x_p - m_j^{(t)} \right\|^2 \forall j, 1 \leq j \leq k \right\} \tag{2.5}$$

In the update step, new centroids for the $(t+1)_t h$ are calculated for all clusters after the assignment step.

$$m_i^{(t+1)} = \frac{1}{\left| S_i^{(t)} \right|} \sum_{x_j \in S_i^{(t)}} x_j \tag{2.6}$$

The assignment step and the updation step, is repeated a specified number of iterations or until the updation step no longer changes assignments. There is no guarantee that the algorithm will find an optimum [11] and the problem is NP-hard in the general Euclidean Space even in a simple case of 2 clusters.

2.3 *K*-shape clustering for Time Series

K-shape [12] clustering works similarly to *k*-means but it also addresses the problems of scaling and translation variances that may be present in the data. The data is normalized to gain scale invariance. Cross correlation similarity measure is used to find an optimal alignment between two time series to gain translation invariance. By using Fast Fourier Transform the complexity of computing cross correlation go down to O(m log(m)), which is orders of magnitude faster than DTW or cDTW. This works by keeping one signal static and sliding the other one over the former to compute their inner product for each shift s of the latter. After all possible shifts are considered, a cross correlation sequence of distance m is defined. The goal is to compute the position w where the cross correlation is maximized. The cluster center calculation is an iteratively improving method that optimizes the shift distance for the members in the cluster. This algorithm may be interesting for use with Telecommunication performance data since each day in the data is relatively short, and the data has relatively few number of distinct shapes in each time series. However, this technique alone does probably not handle big sample to sample variance efficiently.

2.4 Modelling integers in binary

Integers are typically stored using a well-defined size, commonly 8, 16, 32 or 64 bits. However, when compressing integers, it may be desirable to store with flexible sizes. It is therefore relevant to analyze the number of *significant* bits in a stored integer, that is, the lower bound for how many bits are required. For example, the integer nine can be stored in binary like 1001, using four significant bits. If nine is stored using more bits, it entails padding of zeros which contain no information. The number of significant bits can be calculated with

$$\text{bits}(x) = \begin{cases} \lceil \log_2(x+1) \rceil & \text{if } x \text{ unsigned integer} \\ \lceil \log_2(2\,|x|+1) \rceil & \text{if } x \text{ signed integer} \end{cases} \tag{2.7}$$

where the $+1$ inside the $\log_2$ is needed as 0 take up space on the number line. Signed integers need to model both positive and negative values which lead to an extra bit for the sign. Zeros may not require any bits at all. $2^{32} - 1 = 4\ 294\ 967\ 295$ is the largest value that can be stored in a 32-bit unsigned integer.

Negative integers are commonly stored in 2-complementary form. However, as the sign is stored in the most significant bit, all bits become significant. Another encoding for signed integers is *zigzag* encoding. Zigzag encoding interleave positive and negative values in the sequence $[0, -1, 1, -2, 2...]$. This allows small negative values to come closer to zero and have fewer significant bits.

3

Methods

3.1 Data preparation

The data and the environment for computation are provided by Ericsson. The data is already pre-processed, reducing the effort which was required to isolate a suitable data set for this study. We needed to sort data from different network cells into a format that is simple and flexible as we planned to try different kinds of calculations requiring us to write customized code for filtering and visualization etc. For initial analysis, visualization and tests we will use six small datasets of 9,732 time series, each containing 96 values, consecutively measured every fifteen minutes, over a day. Later, for testing large scale clustering and compression ratios, we use a larger dataset containing 93,969 time series. The time series data comes from different event counters on various radio equipment controlled by Ericsson. Each dataset represents a different counter. We chose the counters based on the shape of the time series formed by their data, so we represent the common types of counters.(see Figure 1.2. Initial analysis showed that some counters contained null values. We replace the null values with -1 for computational purposes.

3.2 Definition of Terms

Definition 3.2.1. (Time series data set). A data set X, X_t for a time series with the identifier t.
Definition 3.2.2. (Prototype). P for the collection of prototypes. P_i for one prototype with identifier i.
Definition 3.2.3. (Prototype assignment). u denote a mapping from time series to prototype identifiers. For example, $P_{u(t)}$ is the prototype assigned to time series t
Definition 3.2.4. (Residual). R is the the transformed data set. P_i must have the same dimensions as X_t. R_t is formed with

$$R_t = X_t - P_{u(t)}$$

and the original data is restored with

$$X_t = R_t + P_{u(t)}$$

Definition 3.2.5. (Residual parameter). The use of a prototype may be accompanied with the application of some parameters ϕ_t. A specific ϕ_t relate to one time series X_t and need to be stored alongside the residuals. ϕ denotes the collection of all parameters.

3.3 Scale invariance with data normalization

Since the time series dataset contains many times series that have a similar shape, it is natural to assume that they would fall into the same clusters while clustering and use the same prototype during compression. However, they sometimes have substantial differences in scales of magnitude. This can cause them to cluster improperly. One way to resolve this issue is by normalizing the time series around the same mean value.

In the motivation for the clustering algorithm k-Shape [12], the authors claim that scale normalization of data yields scale-invariant clusters. The normalization strategy used by k-Shape is *z-normalization*, or zero normalization, that shift time series so that the mean overall samples is zero and then scales the samples so that the variance is equal to one. However, which normalization strategy is chosen should depend on the properties of the data at hand.

For many of the performance metrics we have observed, the low values of activity occur during night time. The measurements during the daytime often vary a lot. If samples are shifted in order to make the average of the day normalized to zero, then the predictable and commonly similar values of the low-activity hours are shifted out of place. Therefore, it may match our data better to only scale the data and not shift values, but this is a topic for comparative evaluation.

When a normalization strategy is in place, a practical approach is to apply the normalization to the whole dataset as a preparation for the clustering algorithm. The clustering later produces a set of prototypes that will be in the normalized domain. In order for the prototypes to be used, they need to be upscaled to match the original scale of the time series. If a normalization process $N(X_t, \alpha)$ normalizes the time series X_t with the argument α, the inverse normalization process $N^{-1}(P_i \alpha)$ applied to a normalized prototype P_i will make it similar in scale to X_t. Therefore, we introduce $\phi_{t,\alpha}$ as the inverse normalization factor for X_t, and the residual to a normalized prototype become:

Definition 3.3.1. (Scaled prototype residual).

$$R_t = X_t - \lceil \phi_{t,\alpha} * P_{u(t)} \rceil$$

and the original data is restored with

$$X_t = R_t + \lceil \phi_{t,\alpha} * P_{u(t)} \rceil,$$

where $P_u(t)$ and $\phi_{t,\alpha}$ are floating point numbers whereas R_t and X_t are integers.

3.4 Time invariance using similarity metrics for timeseries

Commonly when comparing vector type objects, such as time series, pairwise comparison on samples at corresponding indices is performed. For time series, however, it is natural that features may occur at slightly different timings without there being a change of meaning. If time series are only compared against each other with fixed alignment, slight miss-alignments of similar features may appear as there are no common features at all. Two strategies were considered for this purpose: shifting all samples in the prototype series uniformly in time, similar to cross-correlation, and shifting samples individually similar to how DTW measure similarity, see section 2.1. This way, the prototypes are applied in the residual calculation in a form that further minimize the residuals.

Shifting all samples uniformly and comparing similarity for each of these alignments is efficiently performed by cross-correlation. The optimal alignment of two time series according to the L^2 norm is given by $argmax_s CC_s(a, b)$, where CC is the cross correlation vector and s is the alignment [12]. When cross correlation shift time series, it pad the missing ends with zeros. Similarly, given an alignment, the samples of a prototype can be shifted prior to application of the residual formula.
Definition 3.4.1. (Shifted prototype). Let

$$\text{shift}_s(\vec{x}) = \begin{cases} (|s| \ \ \text{zeros}, x_0, x_1 ... x_{m-s}), & s \geq 0 \\ (x_{1-s} ... x_{m-1}, x_m, |s| \ \ \text{zeros}), & s < 0 \end{cases} [12]$$

where m is the size of $\vec{x}$. Then residuals to a shifted prototype can be calculated with

$$R_t = X_t - \lceil \phi_{t,\alpha} * \text{shift}_{\phi_{t,s}}(P_{u(t)}) \rceil,$$

and the original data is restored with

$$X_t = R_t + \lceil \phi_{t,\alpha} * \text{shift}_{\phi_{t,s}}(P_{u(t)}) \rceil.$$

Instead, shifting samples individually similarly to DTW would be powerful as this would allow for alignment features that have different lengths. In addition, not every sample in the prototype would have to be compared against. However DTW is computationally expensive, so it was largely ignored. In addition, the alignment would need to be stored along with residuals. Since the storage format would have to model any accepted alignment, the total number of possible alignments can provide a lower bound for how many variants the format must be able to enumerate. By filling a dynamic programming matrix over how many paths exist in the DTW alignment matrix, it can be calculated that 2 bits is required per sample. Even when the warping window is restricted, as in cDTW, close to 2 bits is needed even for small warping windows.

3.5 Clustering and Prototype Calculation

The main algorithm we use for clustering and evaluation is K-means. We use euclidean distance (ED) as similarity metric and optimization of the ED cost for cluster center calculation. We further test a variant that uses dynamic time warping (DTW) distance for similarity metrics. It also uses Dynamic Time Warping Barycenter Averaging for center calculation. We also evaluate the faster variant called soft-DTW. Another variant we evaluate, which utilizes cross-correlation as a similarity metric, is called k-shape [12].

The algorithms mentioned above usually require multiple passes over the entire dataset before they converge and this can prove to be expensive for large datasets. We introduce a greedy clustering method which works as a two-pass algorithm. The number of clusters K is not specified. In the first pass, we iterate through every time series exactly once. The first time series is in a cluster by itself and is also the prototype of the cluster. For each new time series, we calculate its distance to existing cluster centers and if no cluster center is below a specified threshold value, tau, the new time series is added as a prototype. In the second pass, each time series is assigned to the closest matching prototype. We run the greedy algorithm for multiple values of tau and using different distance measures and evaluate the results.

Algorithm 1 Greedy Clustering

```
Input: dataset, tau, distance measure
Output: cluster assignments, prototypes
prototypes = []:
Prototype formation :
for for each time series x do
    if there exist no prototype c so that d(c,x) < tau then
        add x to prototypes
    end if
end for
Prototype assignment :
for for each data point x: do
    find a prototype c so that d(c,x) minimal
    assign x to that cluster
end for
return  cluster assignments, prototypes
```

It is noteworthy that even the similarity metrics that utilize time shifting, has a cost very similar to ED. One important part of the ED formulation is that the differences between the samples compared are squared. This gives a high penalty for large deviations as squares grow faster, and holds even for very small numbers such as those encountered when comparing normalized data. This type of cost evaluation is different from the evaluation of the cost of storing residual integers.

Larger residuals do have a larger cost, but the cost only grows by $\lceil \log_2 \rceil$ of the difference in residual, ignoring effects by batch bit packing during file storage.

When the clustering algorithms finish execution, it provides the cluster assignments for the dataset and the prototypes for these formed clusters. We also evaluate if improvements can be achieved by recalculating the prototypes for the clusters obtained by using other center calculations: mean, and median.

Once the prototypes have been developed, they can be used to transform the time series of the dataset into a residual form. For each of the time series, the best matching prototype is picked for this purpose.

Prototypes can be formed on the basis of different kinds of similarity metrics. The distance calculation measure used to calculate the residual should match the method for which the clustering was made and how the cluster centers were calculated. For instance, if the centers were recalculated, the distance measure should match the method used for the recalculation. For example, if a center was formed without time warping, it might be of little benefit to calculate time warped residuals, but this is a question for evaluation.

The application of prototypes also depends on the data normalization strategy as presented in section 3.3. As described, as the prototypes were formed in the normalized domain, they need to be properly inverse normalized to match the magnitude of the time series at hand.

3.6 Limitations of lossless residual compression

Blalock et al. [4] reported that they compressed 16-bit typed data sets worse compared to 8-bit datasets. The reflection they gave was that large numbers are much more precise and least significant bits are often more dominated by noise. Sayood [3] also went into details of differential encoding, but neither of them analyzed when differential encoding is effective or if there are any formal bounds.

An unsigned integer n requires $\lceil \log_2(n) \rceil$ bits to be stored. Compression rate can be described as one minus the ratio of the compressed and uncompressed size; for instance, 100 bytes reduced to 80 bytes has a compression rate of 20 %. If we want to achieve a compression rate c by reducing the size of a number, we can only leave $\lceil \log_2(n) \rceil (1-c)$ number of bits for the prediction residual. The size of such number can maximally be

$$\mathrm{E}_{max}(n) = 2^{\lceil \log_2(n)(1-c) \rceil} \tag{3.1}$$

or simplified

$$\hat{\mathrm{E}}_{max}(n) = 2^{\log_2(n)(1-c)} = n^{1-c}. \tag{3.2}$$

This equation grows very slowly in comparison to n even for small c. It may be of interest to compare the size of the original number n with $\mathrm{E}_{max}(n)$. For instance, if the properties of the data is such that the prediction algorithm can only get 20 % close to the true value, compression is essentially non-existing except for small n.

$$\frac{\hat{E}_{max}(n)}{n} = n^{-c} \tag{3.3}$$

The above equation shows that compression only with residuals is limited to the precision of the predictions and that it furthermore depends on the size of the numbers. Prediction errors must be proportionally smaller when the numbers are bigger in order to maintain the same compression rate.

3.7 Lossy residuals

So far in the thesis, the algorithms presented can be applied without information loss. In order to increase the potential compression ratio, we present a formulation of residuals that produce smaller numbers at the expense of fidelity by adopting the concept *nonuniform companded quantization* [3]. *Quantization* means that the number line is segmented to a specified set of levels that we will refer to as *quantization levels*. A value is mapped onto a quantization level, which is then stored as an integer. This mapping is irreversible as a range of values may be mapped to the same quantization level, hence, some information loss incur. *Companded* refers to the quantization levels not being spaced uniformly. Prior to integer mapping, the values are applied to a *compressor* function that stretch or compress areas on the number line. This way, the compressor function provides quantization levels that have higher precision where they are needed most. In order to restore the compressed quantized values, an *expander* function is applied, which can be thought of as a lossy inverse to the compressor function.

When analysing a lossy compression scheme, some error is allowed, but the error cannot be too large. Common ways of measuring errors include the average absolute error and the mean square error (MSE) [3]. However, a low score on these measures does not guarantee any properties of each sample. For instance, occasional outliers could be ignored without a substantial impact on MSE, whereas it may be outliers that are most crucial to be stored precisely. Due to the difficulty of deciding if some value is more important than another, we define a uniform criterion for how large the error proportional to the value is allowed to be:

Definition 3.7.1. (Compression precision ϵ). $\epsilon > 0$ is a precision parameter so that every true integer x is proximate to its reported quantized value q as defined by $q(1-\epsilon) \leq x \leq q(1+\epsilon)$.

For example, if $\epsilon = 0.1$, then every true integer x is within ±10% of its reported value q. Based on the definition of ϵ, restrictions can be formulated for the compressor and the expander functions. Definition 3.7.1 state that the stored integer is either larger or smaller than the assigned quantization level. Thus, the space between two subsequent quantization levels l and $l+1$ are covered by two regions, one that will be "rounded down" and one that will be "rounded up". The first region can at most be $\epsilon * \text{expander}(l)$ long and the second region at most $\epsilon * \text{expander}(l+1)$ long since that is how far the error restriction reach. This is formalized by the expression

$$\text{expander}(l+1) - \text{expander}(l) \leq \epsilon * \text{expander}(l) + \epsilon * \text{expander}(l+1), \qquad (3.4)$$

of which its equality formulation has the solution $\text{expander}(x) = a^x$ with $a = \frac{1+\epsilon}{1-\epsilon}$. For example, $\epsilon = 0.1$ yield $a \approx 1.222$. Since a^x solves the equation 3.4 for all continuous $x > 0$, an expander based on this function will satisfy definition 3.7.1. However, a^x is not useful for small x. For $x < \epsilon^{-1}$, the accepted error interval does not stretch even to the adjacent integers and it is not desirable to use a greater number of quantization levels than there are integers. For example, consider the number 95 and $\epsilon = 0.01$. The accepted error interval for a quantization level centered at 95 reach to $95(1 + 0.01) = 95.95$ but not all the way to 96. The first time that the accepted error interval includes another integer is at $\epsilon^{-1} = 100$. Therefore, it is more efficient that the first section of the expander function is linear.

Definition 3.7.2. (Expansion and compaction basis a).

$$a = \frac{1+\epsilon}{1-\epsilon}$$

Definition 3.7.3. (Expander).

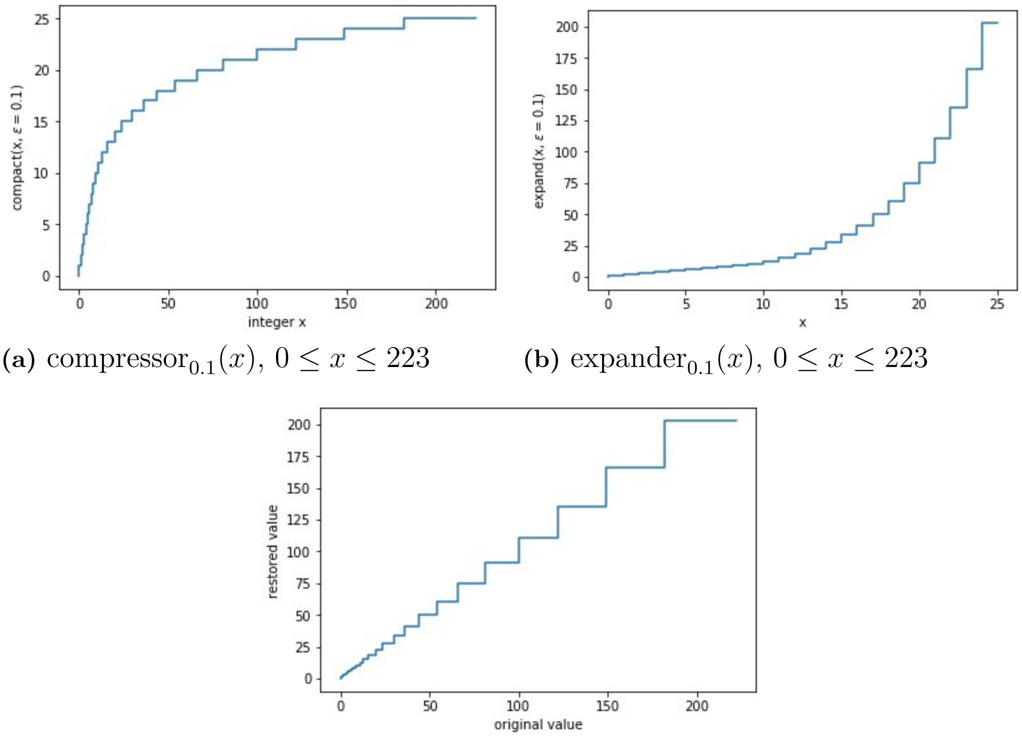
$$\text{expander}_\epsilon(x) = \begin{cases} x & x \leq \epsilon^{-1} \\ \lfloor \epsilon^{-1} \rfloor a^{x - \lfloor \epsilon^{-1} \rfloor} & x > \epsilon^{-1} \end{cases}$$

For continuous variables, it is trivial to construct the compressor as an inverse to the expander. However, the compressor also needs to do the correct integer mapping. Each quantization value q has both a leading and a trailing accepted error interval with the decision boundary being located at $q(1+\epsilon)$. Since $a^x = q * (1+\epsilon) => x = \log_a(q) + \log_a(1+\epsilon)$, one may use $\log_a(1+\epsilon)$ as a constant to make the rounding easier.

Definition 3.7.4. (Compressor).

$$\text{compressor}_\epsilon(x) = \begin{cases} x & x \leq \epsilon^{-1} \\ \lfloor \epsilon^{-1} \rfloor + \lceil \log_a(x\epsilon) - \log_a(1+\epsilon) \rceil & x > \epsilon^{-1} \end{cases}$$

Companded quantization $(\epsilon = 0.1)$



(a) $\text{compressor}_{0.1}(x)$, $0 \leq x \leq 223$

(b) $\text{expander}_{0.1}(x)$, $0 \leq x \leq 223$

(c) Resulting quantizer for $0 \leq x \leq 223$

Figure 3.1: Figure (a) represent the compressor function for $\epsilon = 0.1$ and the values produced are significantly made smaller. Figure (b) represent the corresponding expander functions that restore the compressed values to the precision $\epsilon = 0.1$. Note that both functions are linear and identical for $0 \leq x \leq 10$ as no greater steps can be taken without violating the error constraint in definition 3.7.1, and that 223 is the last integer to obtain the compressed value 25. Figure (c) combine the two functions, often referred to as a quantizer.

Examples of compressor$_\epsilon(x)$

x	$\epsilon = 10^{-1}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-4}$	$\epsilon = 10^{-5}$
10	10	10	10	10	10
100	21	100	100	100	100
1000	33	215	1000	1000	1000
10^6	67	561	4454	33026	215129
10^9	102	906	7908	67565	560517
$2^{32}-1$	109	979	8636	74852	633389

Table 3.1: This table show some examples of compressor$_\epsilon(x)$. Note that no loss of precision occurs for values $\leq \epsilon^{-1}$ and that compressor$_\epsilon(x)$ grows slowly beyond ϵ^{-1}.

Examples of bits(compressor$_\epsilon(x)$)

x	bits(x)	$\epsilon = 10^{-1}$	$\epsilon = 10^{-2}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-4}$	$\epsilon = 10^{-5}$
10	4	4	4	4	4	4
100	7	5	7	7	7	7
1000	10	6	8	10	10	10
10^6	20	7	10	13	16	18
10^9	30	7	10	13	17	20
$2^{32}-1$	32	7	10	14	17	20

Table 3.2: This table show the number of significant bits for the examples of compressor$_\epsilon(x)$ in table 3.1. The compressor does not provide any compression on numbers smaller than ϵ^{-1} but the number of bits required for large numbers plateau.

With this way of compressing values, we can apply it also for residuals. An intuitive way would be to apply the compressor function on the residuals R, in which case compressor$_\epsilon(R)$ represent the compressed distance between X and the used prototypes. This yield improvements, but it is even more efficient if both X and the prototypes are compressed with the residuals measuring the difference in quantization levels.

Definition 3.7.5. (Lossy residuals to a scaled prototype).

$$R_t = \text{compressor}_\epsilon(X_t) - \text{compressor}_\epsilon(\phi_{t,\alpha} P_{u(t)})$$

and the original data is restored with

$$\hat{X}_t = \text{expander}_\epsilon(R_t + \text{compressor}_\epsilon(\phi_{t,\alpha} P_{u(t)}))$$

Note that the reconstructed $\hat{X}_t$ is of type floating point number as the error constraint ϵ is not guaranteed if $\hat{X}_t$ is rounded.

3.8 File storage

The stored residual file needs to contain the prototypes P, assignments u, parameters ϕ, and residuals R. In order to store residuals compactly, using as few bits as possible for each sample, we adopt the technique *batch encoding.* Integers are stored in batches containing a *batch header* and a *batch body.* The batch body contains eight integers and each integer is encoded using the same number of bits, that the batch header specifies using 5 bits. 5 bits is enough to model the number range 0-31. As residuals may take on negative values, the integers are stored in zigzag encoding, see section 2.4. The complete file format is presented in Table 3.3.

File format for batch encoded residuals

Item	Number of values	Data type	Description
$\|P\|$	1	uint32	Number of prototypes
$\|R\|$	1	uint32	Number of time series
P	$\|P\| * 96$	float32	Prototypes
u	$\|R\|$	uint16	Prototype assignments
ϕ_α	$\|R\|$	float32	Scaling parameter
R	$\|R\| * 96$	batches	Residual batch bodies

Table 3.3: File format where each item is laid out in the order they appear in the table. $|P|$ and $|R|$ specify the number of prototypes and time series so that the rest of the file can be read correctly. P is laid out as a multidimensional C-array, where the elements of the first prototype are stored first, then followed by the elements of the second prototype, and so on. u, ϕ and R stored with matching ordering, so that the t^{th} element reference the same time series in each of these entities. Also, if u_t store the number i, this reference the i^{th} prototype in P. In order to keep the format and the evaluations simple, this format does not store any metadata.

This format is compared to a variant that employs *run-length encoding* of zeros. With run-length encoding, an additional value in the header specifies the number of proceeding of 0's. This way, several samples may sometimes be skipped over efficiently. Since our time series are of length 96, we use 7 bits for this purpose.

These two formats are inspired by the compression algorithm Sprintz [4], that employs run-length encoding. The authors of Sprintz argue that a batch size of eight samples is a good tradeoff considering the cost of storing batch headers. If batches are short, the cost of storing the header is proportionally high. If instead, batches are long, then large sporadic values will cause a greater number of samples to be stored inefficiently. However, this tradeoff was not evaluated in this thesis. Furthermore, Sprintz is efficiently implemented using SIMD instructions, CPU vector instructions, which was not done in this thesis. Instead, the python library bitstring [13] was used. It is possible that an implementation using SIMD instructions would benefit from additional padding within batch headers and bodies to make more values byte aligned, but this was ignored for simplicity. Further ideas on how the format may be developed are presented in section 5.5.

Given the file format for batch encoded residuals as shown in table 3.3, we can also formulate a mathematical way of calculating the file size where

$$\text{bits\_batch\_bodies}(\vec{x}) = \sum_{\vec{b} \in \text{batches}(\vec{x})} 8 * \max_{i \in 1..8} \text{bits}(b_i) \tag{3.5}$$

calculates the number of bits in one batch body. The size of the whole file can then be calculated with

$$2 * 32 + |P| * 96 * 32 + |R| * (16 + 32 + 12 * 5) + \sum_t \text{bits\_batch\_bodies}(R_t). \tag{3.6}$$

that also represent the true optimization target.

3.9 Prediction of missing data

In order to illustrate the idea of using a compressed format for computation, we present a simple algorithm for predicting missing data. As the time series has similar properties over the course of days, a good guess for missing data is that it should be similar to known days. For a cell c and the known set of days, History(c) of size N_c, the average time series is calculated with

$$\hat{X}_c = \frac{1}{N_c} \sum_{t \in \text{History}(c)} X_t. \tag{3.7}$$

This can be performed with a single pass over the dataset by incrementing the instances of N_c and the sums of X_t, and doing the division at the end. Similarly, averages can be calculated on the predictions provided with prototypes with

$$\hat{X}_c = \frac{1}{N_c} \sum_{t \in \text{History}(c)} \phi_{t,\alpha} P_{u(t)}. \tag{3.8}$$

The usage of only parameters for this task drastically reduce IO requirements. Reading an uncompressed dataset with 32-bit integers, N time series of 96 samples require reading $32 * 96N = 3072N$ bits, given the storage format provided in section 3.8. Reading the parameters P, u and ϕ_α comparatively require approximately $96 * 32|P| + (16 + 32)N = 3072|P| + 48N$ bits. With $|P| = 0.01N$, the compressed format require approximately $79N$ bits, a reduction of 97.4%.

For the evaluation, it was assumed that the mapping History(c) was known so that considerations revolving metadata didn't need to be addressed. The arithmetic was implemented using the Python library Numpy. Furthermore, the algorithm is not meant to provide competitive predictions as it mainly serves as an illustration. Some more ideas on the problem are provided in section 5.7.

3.10 Evaluation

We evaluate different clustering methods and try different n and k. We also investigate the effects of recalculating the cluster centers, i.e., prototypes, and calculating the residuals on the basis of different distance measures. The effects of using lossy residuals in terms of savings in storage space and accuracy is also evaluated. We evaluate these on different datasets, which consist of time series representing different performance management counters provided by Ericsson.

After the clustering is done, the prototypes and residuals that represent the dataset is stored into file formats as specified in section 3.8. Different file formats are compared to each other, to see how storing them in batches of packed bits evaluate against storing them with a run-length encoding of zeros. We also compare the file sizes to the size of the file which contains the original dataset in bit packed form This evaluations gives us the compression ratio of storing data using our prototype based compression technique to the original dataset. We compare the results from our compression to compression sizes obtained by compressing the dataset in byte form using two standard compression techniques: Snappy and Zstd.

To evaluate how well, the prototype represents the entire dataset, we also use them to predict a new time series for a whole day. We also assess how well it can perform at predicting missing values in a time series. These predictions will be evaluated by comparing them to a test dataset and measuring the accuracy.

4

Results and Discussion

In this chapter we present and discuss different results obtained from the execution of the methods described in the Methods chapter: Implementation and comparison of different clustering algorithms, results for compression ratios, lossy compression and comparison to two standard compression techniques.

4.1 Clustering Algorithms

4.1.1 Dynamic Time Warping

First of all, we found that DTW and cDTW with DTW averaging to calculate clusters, and soft-DTW were very slow to run. For instance, running DTW with $N = 400$ and $k = 40$ took roughly half an hour. This may be due to the implementation of DTW in the library. For instance, cDTW seemed to run consistently slower than DTW. We found that it ran faster when the convergence criterion for the cluster center calculation was changed, but it was still too slow to be practical.

4.1.2 *k*-Shape and *k*-Means clustering

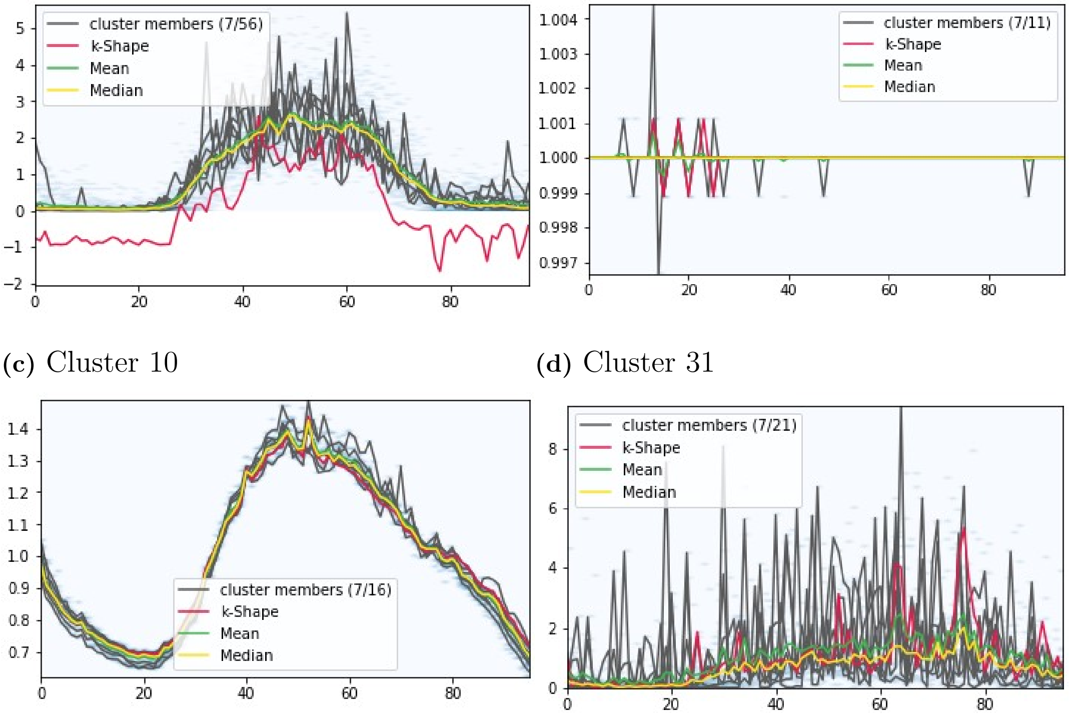
We ran k-Shape and k-Means clustering on a datasets with 9,372 time series using different values of k. The tables comprising of results for different values of k, for different counters can be found in the Appendix. Initially, the cluster centers were recalculated and the difference it causes can also be inferred from the tables. Here we show a summarized table for k=1280 since we saw that k=1280 gave the best results.

The following tables presents the summarized results:

Table 4.1: Table summarizing clustering on N = 9372 with K = 1280, using *K*-means and *k*-Shape for different counters. The results are evaluated by measuring the sum of absolute differences of values from each time series to the center of the cluster. Mean/Sample shows the mean value for samples in each counter and bits show the average amount of bits that would be required to store the samples. The table is divided into three: Original, which shows the mean of the samples in the dataset and the bits/sample required to store them; *k*-Shape and *k*-means, which show the mean of the samples in the residual dataset after clustering and bits/sample required to store them.

	Original		K Shape		*k*-Means	
	Mean/Sample	Bits	Mean/Sample	Bits	Mean/Sample	Bits
Counter 0	206950.2	16.5	235545.2	17.4	68617.3	15.6
Counter 29	76110633.9	27.3	23050.2	0.5	0.0	0.0
Counter 32	821282.5	20.8	9105.6	11.9	6972.0	13.1
Counter 59	76320.9	13.2	92910.2	12.4	56810.2	14.6

Figure 4.1: Normalized data from *k*-Shape clustering on Counters 0, 29, 32, & 59 (N=9372, K=1280).



(a) Cluster 20

(b) Cluster 829

(c) Cluster 10

(d) Cluster 31

The data in the tables, presented above and in the Appendix, show the evaluation results from running *k*-means and *k*-shape clustering on a dataset consisting of 9732 time series vectors for different values of K = 30, 320 and 1280. *k*-Means performed better than *k*-shape clustering in most cases whereas *k*-shape performed slightly better in specific instances. It is hard to infer a pattern of cases where *k*-shape was better since it seems rather sporadic.

Comparing results from the different counters show that some counters are very compressible with a reduction of number of bits required to store the residuals from the original value, for example, from 28 bits/sample to 0 (since all the information was captured by the prototypes) in the case of Counter 29. This can be attributed to the regularity of the time series in terms of magnitude and also the minimal number of deviations from its regular shape. Counter 59, on the other hand, gave poor results due to its stochastic nature with large variations in magnitude, making it harder to cluster and thus forming poorly evaluating prototypes. Counter 32 and to some extent counter 0 gave good evaluation results due to their repetitive and regular nature between different cells and days. It can also be observed from Figrue 4.1 (a) that, due to zero-normalization, *k*-shape sometimes pulls down the normalized prototype values to below zero and this shift can cause calculation of larger residual values.

4.1.3 Comparing parameters of Greedy Clustering

Table 4.2: Table showing results from running the greedy clustering on a dataset of 9372 time series for various parametric settings on different counters. Here D denotes the distance measure used for comparison of time series in the cluster, τ, the threshold value defined for creating a new cluster, and K, the number of clusters that were formed by the clustering. It also shows the number of bits required to store the original dataset in a batch packed form, the bits required after compression using greedy clustering and the compression ratio. Storing the data raw with 32-bits per sample require $2.88 * 10^8$ bits.

Counter	τ	D	K	Batch Packed	Residual file	Reduction (%)
0	1920	bits	433	166 990 380	166 830 388	0.096
0	48	L2	131	166 990 380	171 935 044	-2.961
0	91.2	L1	2045	166 990 380	176 460 676	-5.671
29	744	bits	387	251 965 644	13 481 484	94.65
29	0.00096	L2	1174	251 965 644	15 077 700	94.016
29	0.00096	L1	1174	251 965 644	15 077 700	94.016
32	1488	bits	1520	194 275 932	156 376 324	19.508
32	0.48	L2	1504	194 275 932	159 367 580	17.968
32	4.8	L1	412	194 275 932	159 175 612	18.067
52	840	bits	306	90 328 852	77 735 132	13.942
52	9.6	L2	2172	90 328 852	92 865 148	-2.808
52	57.6	L1	2390	90 328 852	93 657 676	-3.685
59	1728	bits	1234	155 514 028	162 202 876	-4.301
59	48	L2	281	155 514 028	162 559 356	-4.530
59	96	L1	27	155 514 028	160 409 572	-3.148

The results above show that greedy clustering gives a high compression ratio for counter 29, but it performs poorly for Counters 0, 52, and 59. τ was tuned for each counter as a small change often drastically influenced the number of clusters. The tuning process was conducted manually and do not reflect optimal values. Also, using the number of significant bits as similarity measure increased the compression rate compared to using a similar number of prototypes formed using other similarity measures.

4.1.4 Greedy clustering vs *k*-Means clustering

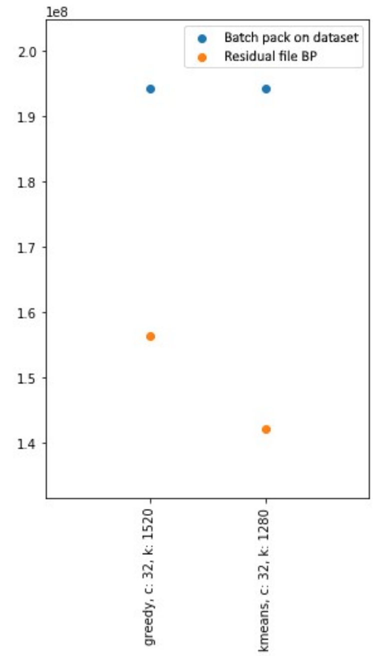
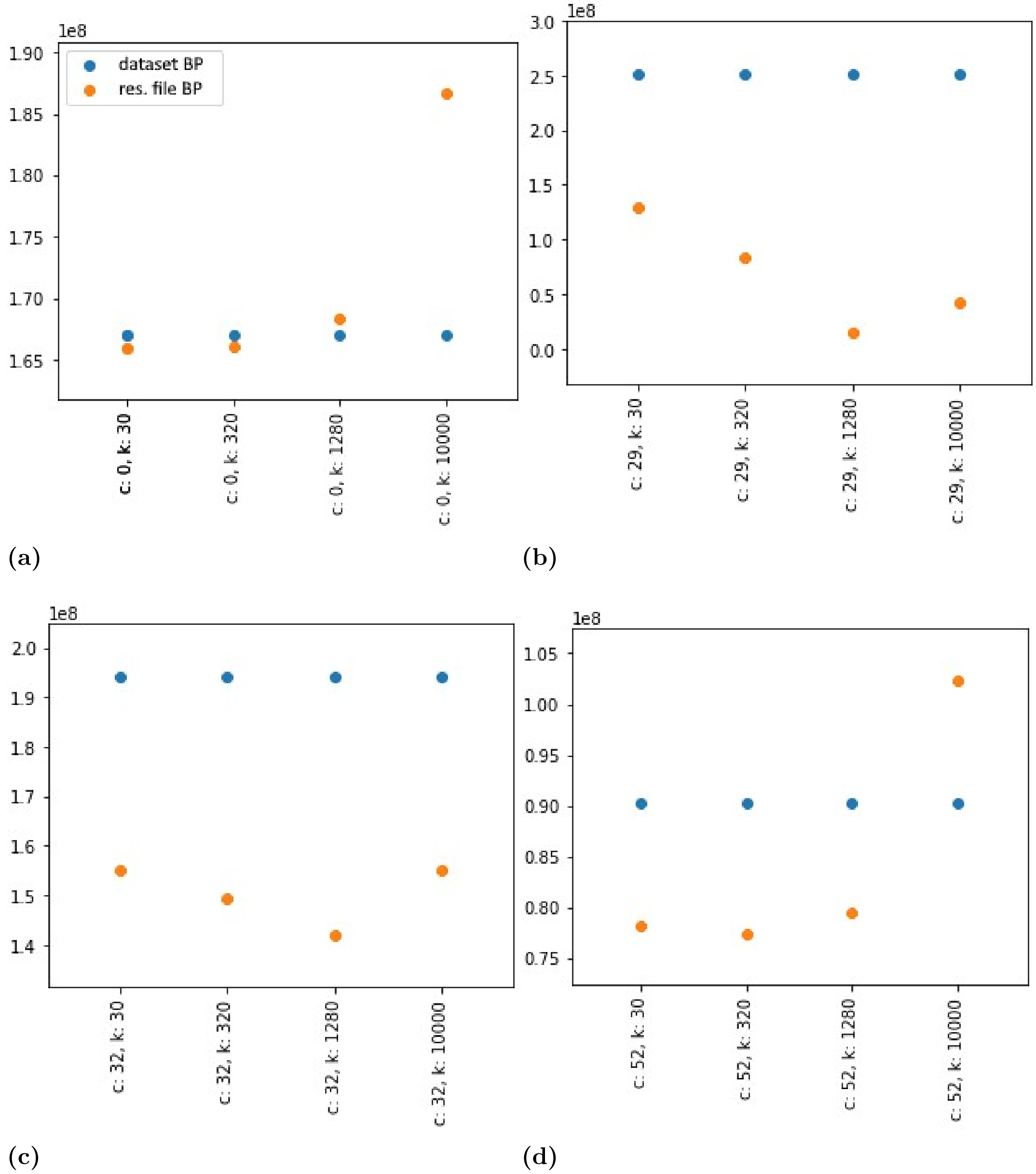


Figure 4.2: Comparison of compression achieved by k-means vs Greedy clustering for the best performed value of k for counter 32. Storing the data raw with 32-bits per sample require $2.88 * 10^8$ bits.

The plot above shows the comparison of the greedy clustering compression that performed best for counter 32 to the k-means clustering compression that performed best for the same counter. We can see that the k-means clustering compressed the file to a smaller size than the greedy algorithm did. This was the case in tests performed for the other counters by varying the k values, and the τ, and and similarity measures for the greedy algorithm. Only k-means clustering is considered in the results and comparisons shown, henceforth, since it outperformed greedy clustering in most cases. Although, k-means performed better than greedy clustering in our dataset of 93969 time series, it might be a viable option for much larger datasets since it passes through the entire dataset only twice and provided competitive results.

4.2 Compression for different counters

Different counters in the dataset form differently shaped time series. Some of them are regular over days and, thus, compresses better than the other counters. The following plot shows the reduction in data size for prototype based compression for four different counters.



(a)

(b)

(c)

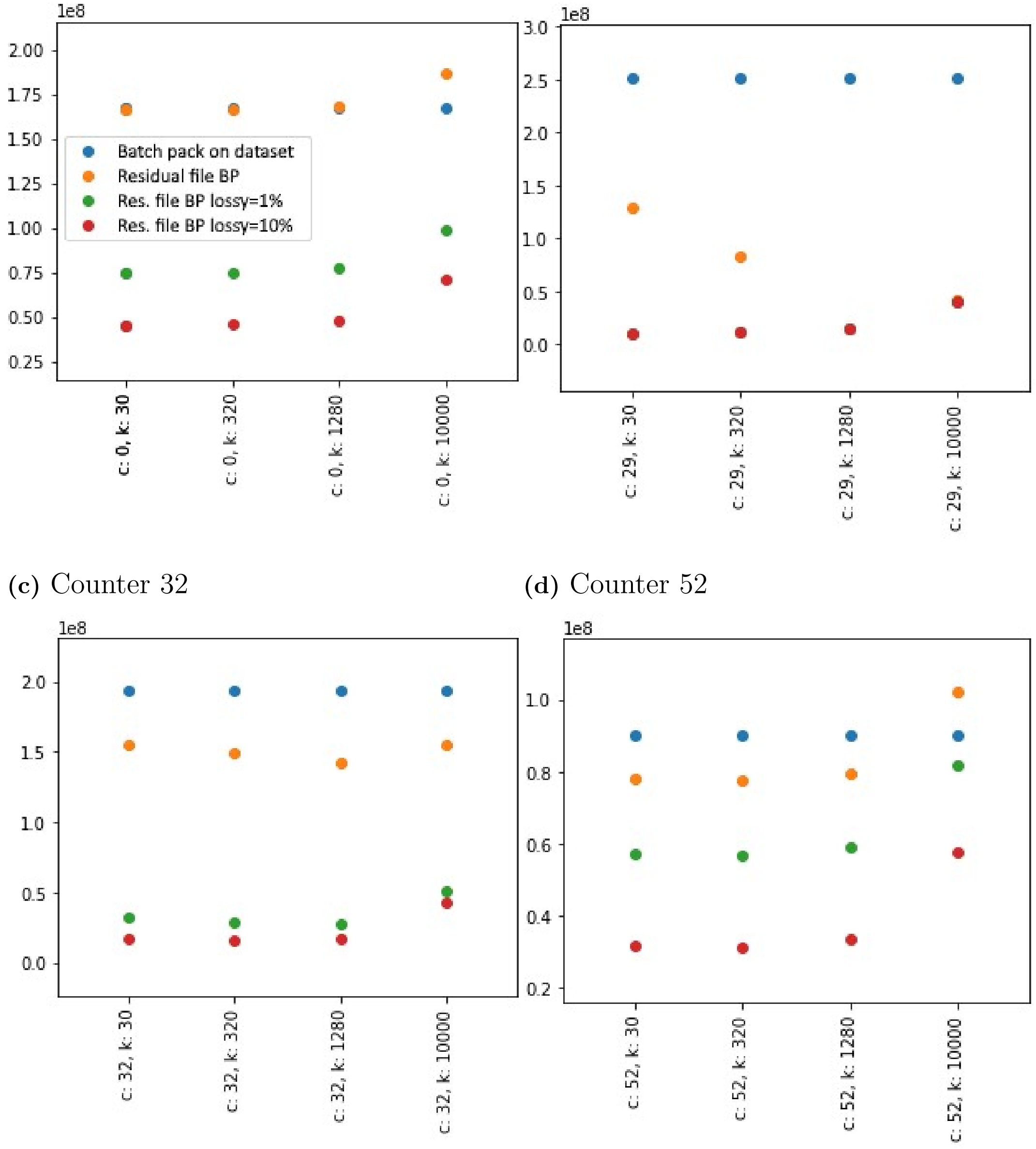
(d)

Figure 4.3: This figure compares the size of storage of the original dataset after batch packing (blue) and the lossless residual file (orange) which includes the prototypes, residuals, scaling factors, and assignment labels. Storing the data raw with 32-bits per sample requires $2.88 * 10^8$ bits.

Here counter 29 compresses close to 95 percent compared to only batch packing the time series. The compression improves for increase in the number of clusters up till 1000 and then worsens for k=10000. We can infer from this that the overhead required to store the prototype crosses the optimum number of prototypes at this point. The counters 32 and 52 also had a reduction in storage size with varying compression ratios for different cluster sizes. For counter 0, the residual file was worse than batch packing for larger number of clusters. This can be attributed to the irregularity of the time series for that counter. Results from this comparison shows that counters that form similar time series over time can be compressed better than irregular counters due to a better performing clustering which in turn forms prototypes that fit the dataset better.

4.3 Lossy Compression

Figure 4.4: The figure shows the comparison of lossy residuals to lossless residuals (orange) when they are both stored as files in a batch packed form, for the acceptable imprecision constant $\epsilon = 1\%$ (green) and $\epsilon = 10\%$ (red). They are also compared to only batch packing (blue) to show relative decrease in file size. Storing the data raw with 32-bits per sample requires $2.88 * 10^8$ bits.


(a) Counter 0

(b) Counter 29

(c) Counter 32

(d) Counter 52

Performing lossy compression, see Section 3.1 and 3.8, provided much higher reduction in the file sizes compared to the lossless compression. In all cases, the lossily compressed files were much smaller in size compared to the only batch packing time series and the lossless prototype-based compression. In the case of counter 29, the prototypes formed from the clustering were able to represent the entire dataset. Thus the residuals formed, when the values of k were 1280 and 10000, were zero values, hence, the lossy compression makes no difference on the residual values and hence the file size remains the same. In addition to this, since the time series retains a similar to the true data with a known maximum error factor, it may still be utilized to approximate Key Performance Indicators or other aggregate values.

Storing a lossy residual file was generally more efficient compared to only using the numerical compressor function, see Definition 3.7.4, together with batch packing. For counter 59 and 60 there was little or no difference, but for the other counters the lossy residual file compressed better or significantly better. It was most noticeable for counter 32 with instances where the lossy residual file was 2.3 and 5.9 times smaller at $\epsilon = 1\%$ and $\epsilon = 10\%$ respectively, but also the other counters showed a trend where larger ϵ lead to larger relative improvements.

Table 4.3: Table showing how lossy residuals, see Definition 3.7.5, reduce the size of residual files.

counter	k	Reduction bits ($\epsilon = 1\%$)	Reduction bits ($\epsilon = 10\%$)
0	320	55%	73%
29	30	92%	92%
32	320	81%	89%
52	30	27%	60%
59	30	47%	64%

4.4 Comparison to Snappy and Zstd

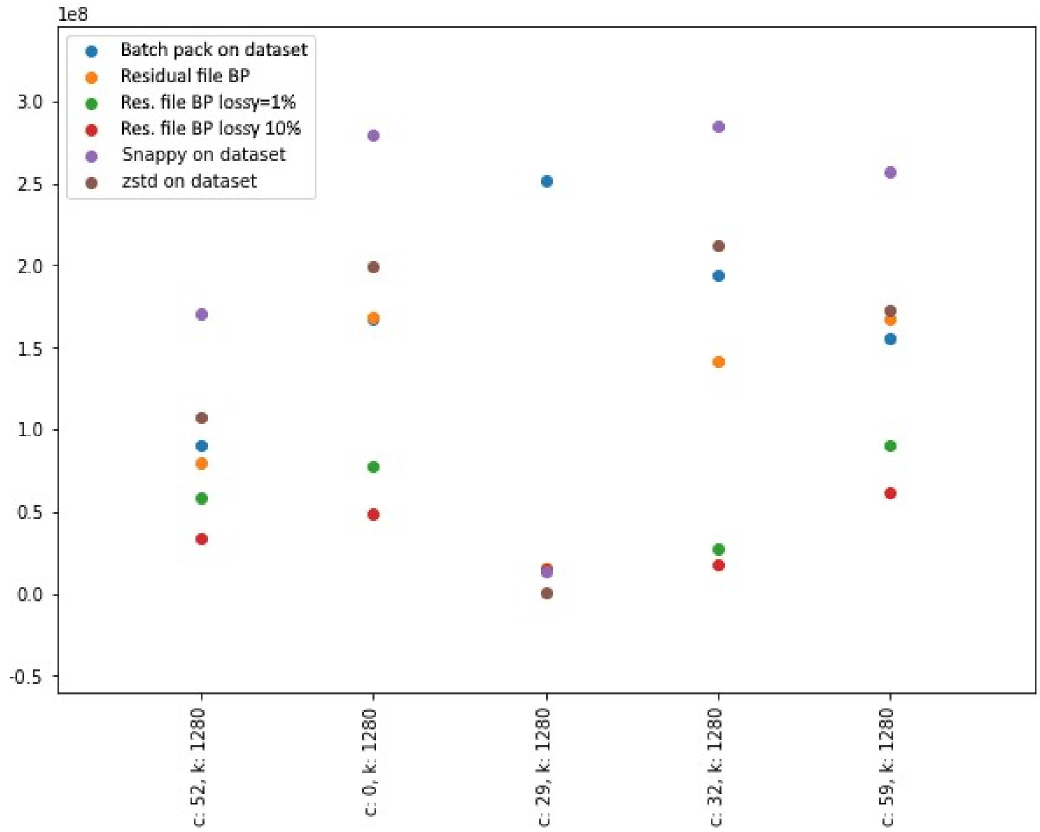


Figure 4.5: Comparison of prototype based compression to two widely used compression algorithms: Snappy and Zstd. The figure shows comparison for 5 different counters with 1280 prototypes in the prototype compression. Here Snappy (purple) and Zstd (brown) shows the file size after using the compression technique on the dataset which was converted to byte form. Storing the data raw with 32-bits per sample require $2.88 * 10^8$ bits.

We used two standard compression techniques called Snappy and Zstd to compress the time series dataset which was converted to byte form for compression and compared the resulting file sizes to that of the file compressed by our technique. The plot above presents the results.

All counters without piece-wise constant sections compressed better with batch packing and our residual file, particularly for lossy compression. Snappy's and Zstd's worse performance can be explained by them requiring exact reoccurring subsequences. Batch packing utilizes similarity of size and the residual file reoccurring patterns that do not have to be exact.

Counter 29, however, with large non-zero constant values, compressed significantly better with Zstd. Snappy was only 10% better than a residual file, but Zstd was

30 times better than that. The good performance by both Snappy and Zstd can be explained by their good handling of repeated values. Unfortunately, we do not present the results for counter 60.

4.5 Secondary compression using Snappy and Zstd

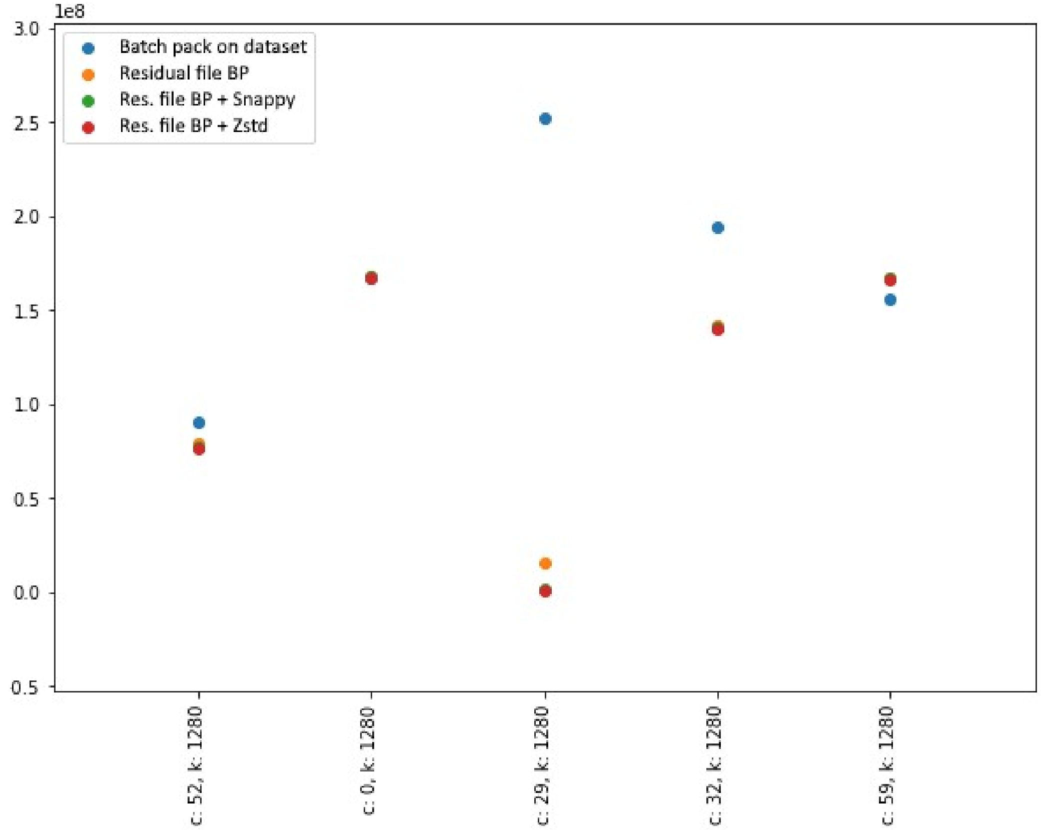


Figure 4.6: Usage of Snappy or Zstd as a secondary compression method, where snappy or zstd was used to compress the file that contains the prototype based compressed data. Storing the data raw with 32-bits per sample require $2.88 * 10^8$ bits.

It was also checked if using Snappy or Zstd as a secondary compression would provide additional data savings. The plot above shows the comparison of batch packing (blue) to residual files with batch packing (orange). It is also compared to an additionally compressed file formed by using the Snappy (green) or Zstd (red) compression on the residual file. The results show that there are no considerable improvement or setback for most counters. For counter 29, however, Snappy provides an additional 9.5 times improvement and Zstd 19.7 times improvement, only 35% worse than using Zstd as primary compression. Unfortunately, we do not present the results for counter 60.

4.6 Prediction of missing data

Speed: Prediction of missing data

	Averaging (ms)	**Prototype averaging (ms)**
Read file	242	14
Prediction single cell	0,15	0,25
Total	242	15
Prediction of 3928 cells	1481	2306
Total	1723	2321

Table 4.4: Execution times from running the averaging of 10 days of data, see Equation 3.7, and averaging of used prototypes, see Equation 3.8. Each time series dataset was stored as uncompressed 32 bit integers at 36MB whereas the residual file header was stored uncompressed at 1MB and it contained 1280 prototypes. The experiment was implemented in Python providing computational speed far below expected values but the results still illustrate potential speedups. The running times were averaged over 6 runs.

Prediction Accuracy Averaging

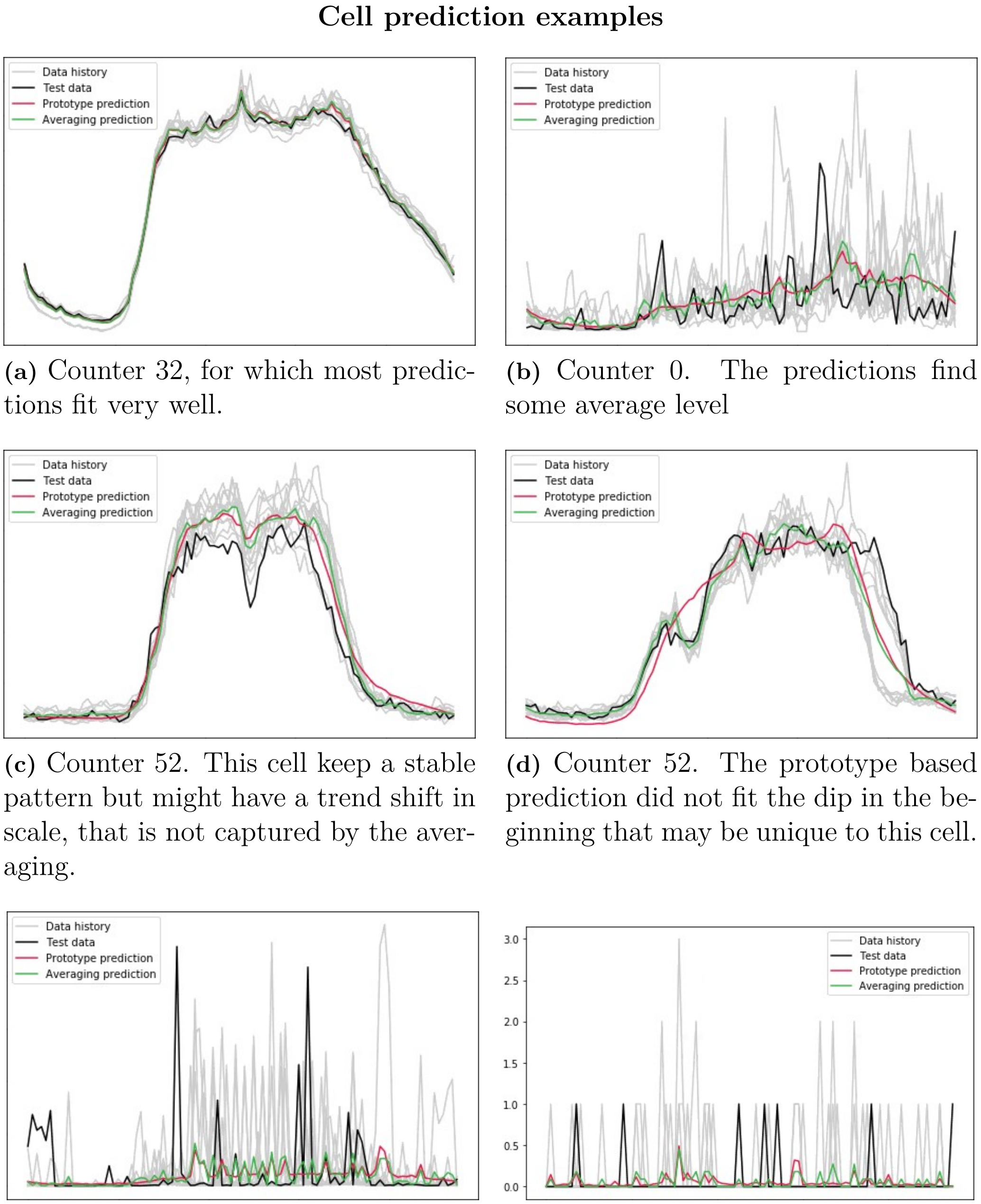
Counter	**MSE**	**Avg. Error**	**Avg. Error/Value (*)**	**Sample=0**
0	54 068.368	2.142	3.910	12 906
29	0.983	0.001	0.001	5 774
32	1.063	0.034	0.034	5 774
52	15.470	0.265	0.285	13 960
59	3 898 867.011	17.096	115.964	30 154
60	8.496	1.647	0.025	366 661

Table 4.5: Comparison of prediction errors for different counters. The 3928 test data time series is normalized using 1-mean normalization and the same normalization factor is used for the predictions. In total, the test data for each counters contains 377088 samples.
(*) Only for non-zero samples.

Prediction Accuracy Prototype Averaging

Counter	**MSE**	**Avg. Error**	**Avg. Error/Value (*)**	**Sample=0**
0	41 920.701	2.145	6.041	12 906
29	0.983	0.001	0.001	5 774
32	1.062	0.035	0.034	5 774
52	14.107	0.271	0.325	13 960
59	2 954 982.289	17.059	148.048	30 154
60	6.086	1.646	0.025	366 661

Table 4.6: Comparison of prediction errors for different counters. The 3928 test data time series is normalized using 1-mean normalization and the same normalization factor is used for the predictions. The prototypes come from k-means clustering with $k = 1280$.
(*) Only for non-zero samples.



(a) Counter 32, for which most predictions fit very well.

(b) Counter 0. The predictions find some average level

(c) Counter 52. This cell keep a stable pattern but might have a trend shift in scale, that is not captured by the averaging.

(d) Counter 52. The prototype based prediction did not fit the dip in the beginning that may be unique to this cell.

(e) Counter 59. This prediction fit neither the low levels nor the peaks.

(f) Counter 60. Similar problems as counter 59.

Figure 4.7: These plots show examples of predictions for various counters. The prototype averaging tend to be very similar to the averaging of the true time series, but sometimes miss features that are recurring for a particular cell. Example (b), (e) and (f) show that averaging is problematic with data that has many sporadic outliers. In these cases it is impossible to predict the exact timing of peaks, but it may investigated if it is possible to predict low levels more accurately together with aspects such as outlier frequency and peak levels.

4.7 Conclusion

In this thesis, we have explored techniques and use cases for compressing time series data provided by Ericsson. The methods explored revolve primarily around the compression idea that a large group of time series can be represented by a much smaller number of prototypes and the calculated residual values between the time series and their corresponding assigned prototypes.

Initial analysis and visualization of the dataset revealed that some counters formed regular time series curves, whereas others were more noisy and unpredictable. From the provided dataset, we used six performance measurement counters which formed time series, which we thought was representative of the larger dataset. It was expected that the more regular time series would be easier to represent with a smaller number of prototypes, and hence, more compressible.

Prototypes for the dataset were formed using the clustering technique k-means. Different clustering techniques such as k-means, k-shape, and greedy clustering were used on the dataset and evaluated before deciding on k-means. We observed that k-means outperformed k-shape and greedy clustering in most cases. However, greedy clustering still provided competitive results and might be a viable option for larger datasets when multiple iterations through the entire dataset become expensive and impractical. Parametric optimization may be further required to find the optimal τ, distance function, and the number of clusters. Dynamic Time Warping as a distance measure while clustering proved to be too slow to be practical for large datasets and a large number of clusters with running time increasing linearly to the number of clusters.

When a prototype is formed, it is desirable that the prototype is similar to the time series it is applied. However, when the prototypes come from averaging of cluster members, a single large outlier can make every member deviate slightly. If the value ranges are large, this can cost an addition of several bits, as can be suggested from k-means clustering on counter 29 as can be seen in Table 6.4. Instead selecting the median sample made a huge improvement for small k. However, the median is challenging to calculate in large scale applications. Therefore we used the mean time series of the cluster as the prototype.

As expected initially, the clustering performed very well on counters 29 and 32 which formed regular time series over multiple days, whereas counters 0, 59 performed poorly due to their noisy nature. The residual file for counter 29 performed very well with greedy clustering and was 94.6 percent smaller than only batch packing. However, for counter 59 and some clustering cases of counter 0, due to their noisy nature, clustering was unable to provide meaningful prototypes and the residual values were high. This causes higher file sizes due to storing residuals for every time series along with the prototypes and the resulting files were not compressed.

We observed that Lossy compression provided significantly higher compression rates even for noisy counters were compression increased up to 92% compared to batch packing. Decompression of lossily compressed time series still retains similar structure to that of the original time series and thus might prove useful for tasks like calculation of Key Performance Indicators or other aggregates.

Our algorithm also outperformed two standard general-purpose compression algorithms Snappy and Zstandard for 4 out of 5 tested counters. This result shows that the techniques used in this thesis are suitable for time series that lack constant sub-sequences, particularly when common trends exist in the dataset. On the other hand, for time series that are piecewise constant, Snappy and Zstandard perform better. However, by using either Snappy or Zstandard as a secondary compression, the gap is closed with no reduction in compression ratio for the other time series.

Another major use case for the prototypes besides forming the basis of the compression technique is that it could be used to analyze trends of data over time and to predict missing data in a dataset. For regular counters 29, 32 and 52 we were able to predict a time series for a whole day for 3928 test data with an error range of 0.001 to 0.325 per sample on a 1-mean normalized dataset.

Our investigations conclude that our technique of prototype-based compression proves very effective for time series datasets which contain data which are regular and repetitive and is efficient in detecting patterns to form smaller representative datasets. This also benefits increase in computational capability and reduction in storage sizes. We believe that our methods and results open up opportunities for larger investigations into the topic of time series clustering and compression, while providing huge benefits to Ericsson.

5

Future work

In order to solve many analytical tasks in telecommunication or IoT, there exist a need for efficient large scale storage and algorithms. Reducing the data into simpler forms may prove crucial in this development. This thesis presents a compression format that can be decompressed to various degrees. Prototypes, parameters, and residuals may all be used, but some may be ignored in order to reduce IO usage.

The idea of selectively using only aspects of the data can be taken further. If the compression format contains more statistics and summarizing values about the prototypes and time series such as variance and maximum levels, more kinds of tasks would be supported on the compressed format. For instance, if it is known that the residual variance for a particular prototype is low, that assignment label may in itself be enough to validate the conditions of a query. For tasks where similar optimizations are possible, the compression may be seen as an intermediate step in the computation. As long as only a few values for each time series is stored for this purpose, the storage size is insignificant in comparison to the original data, but significant IO and computational reductions may be achieved. This is particularly the case when information loss is acceptable such as when residuals are stored imprecisely or ignored completely. Information loss prevents the original data to be restored accurately, but can also be helpful in improving algorithm and prediction generality.

As a way forward, it would be relevant to survey what kinds of analytical tasks are relevant in the domain and what kind of information is needed to answer such queries. If some dense and well descriptive values can be easily computed and then reused for multiple analytical tasks, then there is a good chance it is worth developing this thesis further. The rest of this chapter further discuss the potential impact of applying compression techniques and provide some suggestions for tasks that may be considered for the compressed format.

5.1 Compression to enable simple architectures

If the size of the dataset is large, it can only be stored and processed in compute clusters. That is costly, add performance penalties, and may make development harder. If instead, compression is applied to such a degree that the dataset fits in a single machine, many new kinds of system architectures can be enabled, at least for some use cases. One strategy may be to store a highly compressed version of the data set in addition to the accurate data set. The compressed dataset can then be duplicated into development environments, be used to train and evaluate machine learning models, or be used to run proximate queries. An approximate query may be sufficient but can also be used to prune and optimize computation on the larger dataset.

5.2 Effects of compression on performance

In this thesis, only little focus was directed towards evaluating computational speed. The performance results presented for the prediction task in section 4.6 is underwhelming and not what would be expected from a well-optimized implementation. In general, it is helpful to think of tasks that are either bound by IO or by computation.

When the computation is light, it is expected that the execution time is bounded by IO, at least for well-optimized applications. Examples of light tasks include data selection queries, simple statistical calculations, and some machine learning tasks. When IO bottlenecks the execution, a reduction of IO will cause speedup. Therefore, it is desirable to select and read a smaller amount of data from disc. Similar speedup will also be obtained from efficiently decompressing highly compressed data. Lossy representation may, therefore, provide execution speedup.

When the computation is substantial, the execution time is bound by computation and IO speed is of lesser significance. Examples include processes dominated by overhead costs, algorithms that frequently access random memory and machine learning such as deep learning. For these cases, optimizing IO have comparably lesser significance. As long as the decompression in itself is fast, no significant impact is inflicted on execution time, but the savings in storage remain.

5.3 Large scale prototype formation

In this study a relatively small amounts of data was used. Scaling up the amount of data used has has prospects of contributing to higher compression rate as more redundancies is expected from a larger dataset. However, in order to use prototype based compression at such scale, the algorithms involved need to be developed. One core challenge is to prepare and cluster a time series data on a large scale. Once the collection of prototypes is developed, it can be distributed and used in parallel.

5.3.1 Data preparation

For the applications presented in this thesis, the collected data is interpreted as time series. The data that originate from the same cell is considered one unit, which poses challenges if data is collected continuously. A common way of handling data is to store it as rows in a distributed table. However, it is crucial for performance that data from the same cell is stored in the same machine. When data is processed as time series, all individual samples need to be grouped together. If the data is not stored in close proximity, it risks causing expensive reorganization and transmissions over network.

5.3.2 Increasing number of prototypes

By increasing the number of prototypes, more aspects of the dataset may be modeled and more redundancies utilized for compression. In this regard, the compression problem differs from many other clustering applications. When the intent is for a human to interpret the clusters, there cannot be too many of them. However, when the usage is purely for machine use, in particular for compression, the suitable number of clusters is likely much higher. As long at it improves the total storage by adding more prototypes, there might be a good reason to do so. For instance, if the number of prototypes $|P|$ equals 1% of the total number of time series, $|P| = 0.1N$, the storage of those are still likely insignificant.

A drawback with using a large number of prototypes is that it would require a large number of similarity comparisons. If every time series is compared to every prototype, a total of $N|P|$ comparisons are needed. It would, therefore, be crucial to employ some indexing technique so that the number of comparisons can be brought down to $O(N \log(|P|))$. Interesting algorithms to analyze includes K-d trees, iSax 2.0 [14], and the use of hierarchical clusters as in [15]. K-d trees provide efficient lookups for vectors but work only well for few dimensions. iSax provides a tree search based on comparing progressively more accurate representatives. Tan et al. [15] clusters time series in two levels so that the top level narrow down the search of cluster representatives to be compared.

A significant increase in the number of prototypes may also increase the challenges for analytical tasks operating on prototype labels. As more clusters form, many will be similar to each other. Distinguishing classes and behaviors from the data may be challenging as the set of prototypes in itself may be large and assignments more varied. On the other hand, since a broader set of prototypes can represent the dataset more accurately, there exists potential of achieving better results. As the prototype assignments provide a condensed version of the dataset. Over time, the data from a particular cell form a sequence of parameters and discrete labels. It may enable the use of algorithms that would be unsuitable for the raw time series of integers.

5.3.3 Large scale clustering

Due to the large scale of the data, it is likely not feasible to employ iterative algorithms such as k-means, where the algorithm iterate over the data several times. Instead, the greedy clustering, Algorithm 1, provide an idea for how clustering may be employed more efficiently. Hopefully, single pass over the dataset is enough form a collection of prototypes, which would entail profound performance gains compared to other clustering techniques. Furthermore, Ding et al. [7] argue that clustering can be employed safely on a subset of time series.

It is interesting to analyze how the collection of prototypes would change over time. According to experience from this project, the characteristics of time series remain similar over days and weeks. Experts in the field confirm this with the extension that changes commonly happen throughout seasons in the year and events such as concerts and holidays. That suggests that the prototypes formed at one point in time can also be reused for data that arrive later, but this would require more investigation. If it is the case that prototypes are relatively stable over time, only limited additions or removals would be required regularly. A distributed prototype update may be executed with the following steps

1. Distribute collection of prototypes.
2. Scan a sub-sample of new data for new potential prototypes.
3. Possibly attempt de-duplication.

with only step three being single thread bounded. Some special handling may be needed so that comparative analysis can be done between rare events that occur infrequently or even only yearly.

5.3.4 Lossy prototypes

In this thesis, the prototypes were produced normalized time series and stored accurately. If the goal of the clustering is to produce prototypes that will be used for lossy compression, it might also be considered if the prototypes can be transformed into a lossy form. One way may be to utilize the compressor function presented in Definition 3.7.4. For instance, if all time series are compressed before clustering, some different time series might end up more similar, making the prototypes more efficient.

5.3.5 Time warping

As reported in results, there was some utility in doing k-shape clustering that compare shifted time series. Even if it did not show to improve compression much at the tested scale, it might be of interest to find time-invariant features for analytical purposes. Time-invariant similarity measures may also be evaluated for Greedy clustering or similar cluster algorithms, and new approaches may be considered for similarity measurements.

For time series that are mostly zero with a few single non-zero features as in counter 60, the prototype would not need to be a full-length time series but only the length of the single feature. For other types of time series, one may consider how the padding at the inserted ends is performed. In cross-correlation, the ends are padded with zeros, but it may also be considered to pad with the edge-most value and to roll in values removed from the other side. According to the results of the approach tried in this thesis, well compressible counters tended not to shift at all, but it may be because of the limitations of the tested technique. Also, it may make little sense in measuring large shifts. If only small shifts are compared, the computational expense should be significantly reduced compared to cross-correlation.

5.4 Time series shorter than a day

In this thesis, the data for one whole day was considered to be one time series and a prototype span the whole day, but it may be considered to split the data into even smaller segments. It is more likely to find matching short sequences than long ones, and there might be comparatively little meaning in only matching complete days. Since the same prototype shapes may be shared for each segment of the day, the total size of prototypes may also be reduced. Shorter time series may make it simpler to mine features that only span parts of a day. For instance, some analytical use cases only focus on the worst parts or moments of a day and ignore the rest. It does come with its challenges though.

The number of assignments and parameters would increase, but it may also be desirable to increase the resolution of the lossy components of the storage format. A greater number of similarity searches would be required, but on the other hand, each similarity calculation is faster as each segment is shorter.

Another challenge is that the number of split locations increase. Since important features may span over the arbitrarily located splits, precautions may be needed to not lose information. One attempt to address it is to employ some time warping technique. For instance, the exact location of each segment is slightly adjustable. Furthermore, the prototypes could be slightly longer than the sections they are applied to so that only a subsequence of a prototype is applied.

It may be useful for real-time applications to use shorter time series. When the prototypes are a whole day long, they can only be used for analysis when a complete day of data is accessible. If the prototypes are used for inference when only a portion of the data is available, the best matching prototypes may say little about the remaining part of the day. Shorter time series can this way provide faster feedback. It may, for instance, be possible to analyze if the day progress in an expected way.

Since a shorter time series have the potential of making the prototype based storage fit better, it may be possible to ignore exact residuals in applications that accept some level of information loss. That may make the lossy components of the prototype compression usable for a greater number of applications.

5.5 Compression format extensions

One important motivation for the compression format presented in this thesis is the ability to use parts of the format without decompressing or restoring the original data accurately. Therefore, it might be relevant to add additional values to enable a broader range of tasks. Examples may include

- The number of times a prototype has been used.
- The variance or other distribution for residuals, either for the whole time series or for sections of the data.
- Maximum values in time series or segments of time series.
- Indicative statistics for each batch, such as if residuals tend to be positive or negative and if the variation among residuals is large.

5.5.1 Missing values

In this thesis, missing values in the data were stored as the number -1. For missing values, it was therefore required to store large residuals. It may instead be beneficial to adjust the zigzag encoding so that one code can model null. For instance, a zigzag encoding may instead be defined by the sequence $[0, null, 1, -1, 2, -2...]$

Another strategy is to construct a dense index of where nulls occur. If one considers the storing codes that state that all values in a time series exist, all are missing or that some exist, only two bits are required. This represents a compression of approximately 0.06% compared to reading 32-bit integers uncompressed. Furthermore, for data that only has some existing values, an accurate record of which samples are missing can be stored using at most 96 bits, which is about 3% of uncompressed integers. Doing any analysis of missing values has, therefore, the potential to obtain high-speed.

5.5.2 Improved batch storage

In this thesis, five bits were used to store the number size. Five bits is sufficient in a logical sense as it can model the value range 0-31. However, five bits does not store well so it would be preferable to store the batch number sizes as a whole byte, improving algorithm simplicity and speed. If the batch sizes are stored separately from the batch bodies, they may be used in computational tasks without using the residuals in the batch bodies.

As each batch body contain eight samples, the beginning of each batch body will always be byte aligned. Despite how many bits are used, the total length sum to a multiple of 8. Therefore, keeping an accumulative sum of the batch number sizes provide an index for where batch bodies are laid in memory.

The batch number size bounds the deviation to the used prototype, as can be seen in Table 5.1. The additional three bits may also be used to store $2^3 = 8$ times increased fidelity. This way, the batch number size more accurately describes the maximum size of residuals, and not only how many significant bits that are used. The deviation interval may, for instance, be used to infer if values exceed or fall below a threshold, or if there is a change that it does. Even if the precision is limited, it might accelerate tasks such as trend analysis, when precision may have limited necessity.

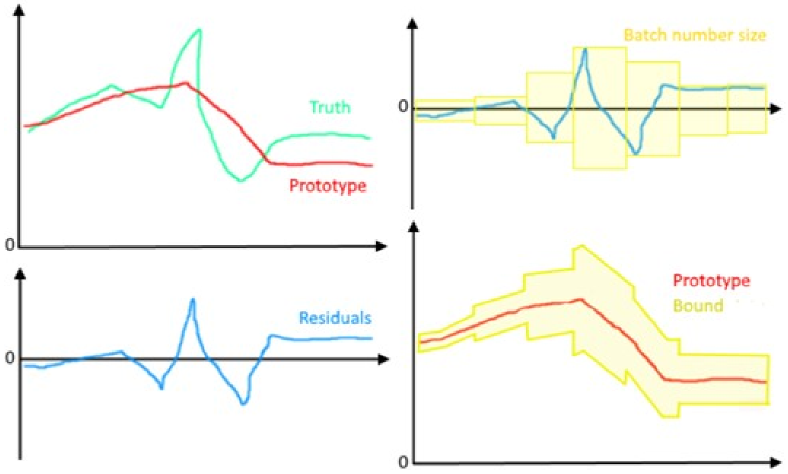

Batch number size illustration

Figure 5.1: In the top left, the true time series and a prototype are shown. The bottom left show the resulting residual. As the batches are stored, the size describes a bounding interval for the residual values. The same bounding interval can then be applied to the scaled prototype instead of decompressing the residuals

5.5.3 Large scale data considerations

Since the prototypes are intended to be used on very large data sets, they should likely be stored separately from the time series in the dataset. This way, the same collection of prototypes can be used for many other compressed files. Also, it is adjustable how many time series are put in a single file. Even in the same file, the content may be organized with repeating header and body sections to enable easy parallel reads.

With prototype compression, the data from the same cell and day is stored in the same file, compared to being spread out in multiple rows in a big table. This may enable the development of efficient dataset scale indexes.

5.5.4 High speed decompression

In order to obtain high performance of compression, decompression, and many analytical tasks, it would be required to implement algorithms with efficient programming languages. For instance, CPU vector instructions, SIMD, may be relevant for residual calculations and bit packing procedures. The similar algorithm Sprintz [4] reported single threaded decompression speed over 3GB/s. This show that there exist promising possibilities in the ability to implement a compression algorithm that can operate on very high speeds.

5.6 Anomaly detection

Prototype-based compression provide added structure to the data, of which several aspects can be used for anomaly detection. For a start, if some time series does not compress well, it suggests that something is unusual. Furthermore, assignments, stored statistics, and parameters can be analyzed to find trends. For instance, the assignments over time form a series of discrete labels, which may be a suitable subject for Markov models. Another strategy may be to keep statistics of each prototype. If that statistics can be used accurately enough, decisions may be conducted solely on the assignments of prototypes. One such example may be if the prototype shape satisfies some condition and stored statistics reveal that no large residuals exist for that prototype.

Even if the algorithms using lossy forms of the data are not entirely accurate, they may be used to find a reduced number of candidates. More precise forms of the data can then be processed selectively, potentially yielding a total speedup and a reduced use computation resources.

5.7 Predictions of future data

As was discussed in 4.6, there is a significant variation in how regular the counters are. Trying to predict exact values at all points in time is not only impossible, but it might also be irrelevant. For instance, instead of guessing the exact magnitude at particular times, it might be more relevant to estimate aspects such expected maximum of amplitude and frequency of such peaks. If aspects such as these are stored, it may easier develop algorithms to distinguish trends and train models on a more extensive history of data.

5.8 Multivariate analysis

So far, we've only discussed the application of prototype compression on a single counter at the same time. However, sophisticated analysis likely requires the consideration of many counters simultaneously. There exists many strategies to take this forward. For example, counters may be clustered separately, which require care of the clustering algorithm as there exist many different counters. Another strategy is to run all counters in the same instance of the clustering. This way, the prototypes are shared for all counters, and some comparative analysis may be performed based on cluster assignments. A third strategy is to cluster in a multivariate setting so that each prototype contains values for each counter. This may reveal patterns that span over multiple counters but may not be suitable for compression as it should make it harder to find well-fitting prototypes for many time series.

5.9 Summary of future work

Key topics:

- Compress datasets and store lossy versions of large datasets, either by using lossy residuals by ignoring them entirely.
- Utilize or develop new algorithms designed for large-scale time series similarity search and clustering.
- Develop time-invariant similarity measures and residual calculation suitable for large scale compression.
- Evaluate clustering and compression of shorter time series than a day.
- Improve the storage format by encoding *null* values in zigzag encoding and separate the batch headers so they are accessible without reading batch bodies.
- Extend the storage of prototypes and time series with additional statistics such as maximum values and variance.
- Implement high-speed compression and decompression
- Employ the storage format for tasks such as threshold breach searches, anomaly detection, trend analysis, and predictions.
- Consider multivariate analysis.

Bibliography

[1] Arnak V Poghosyan, Ashot N Harutyunyan, and Naira M Grigoryan. Compression for time series databases using independent and principal component analysis. In *Autonomic Computing (ICAC), 2017 IEEE International Conference on*, pages 279–284. IEEE, 2017.

[2] Shaurya Agarwal, Emma E Regentova, Pushkin Kachroo, and Himanshu Verma. Multidimensional compression of its data using wavelet-based compression techniques. *IEEE Transactions on Intelligent Transportation Systems*, 18(7):1907–1917, 2017.

[3] Khalid Sayood. *Introduction to data compression.* Morgan Kaufmann, 2012.

[4] Davis Blalock, Samuel Madden, and John Guttag. Sprintz: Time series compression for the internet of things. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(3):93, 2018.

[5] Saeed R Aghabozorgi, Teh Y Wah, Amineh Amini, and Mahmud R Saybani. A new approach to present prototypes in clustering of time series. 2011.

[6] Aseel Basheer and Kewei Sha. Cluster-based quality-aware adaptive data compression for streaming data. *Journal of Data and Information Quality (JDIQ)*, 9(1):2, 2017.

[7] Rui Ding, Qiang Wang, Yingnong Dang, Qiang Fu, Haidong Zhang, and Dongmei Zhang. Yading: fast clustering of large-scale time series data. *Proceedings of the VLDB Endowment*, 8(5):473–484, 2015.

[8] Joan Serra and Josep Ll Arcos. An empirical evaluation of similarity measures for time series classification. *Knowledge-Based Systems*, 67:305–314, 2014.

[9] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, Calif., 1967. University of California Press. URL `https://projecteuclid.org/euclid.bsmsp/1200512992`.

[10] David J. C. MacKay. *Information Theory, Inference & Learning Algorithms.* Cambridge University Press, New York, NY, USA, 2002. ISBN 0521642981.

[11] J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979. ISSN 00359254, 14679876. URL `http://www.jstor.org/stable/2346830`.

[12] John Paparrizos and Luis Gravano. k-shape: Efficient and accurate clustering of time series. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1855–1870. ACM, 2015.

[13] bitstring - a python library for packing and analysis of bit strings. `https://github.com/scott-griffiths/bitstring`. Accessed: 2019-05-21.

[14] Alessandro Camerra, Themis Palpanas, Jin Shieh, and Eamonn Keogh. isax 2.0: Indexing and mining one billion time series. In *2010 IEEE International Conference on Data Mining*, pages 58–67. IEEE, 2010.

[15] Chang Wei Tan, Geoffrey I Webb, and François Petitjean. Indexing and classifying gigabytes of time series under time warping. In *Proceedings of the 2017 SIAM International Conference on Data Mining*, pages 282–290. SIAM, 2017.

6

Appendix

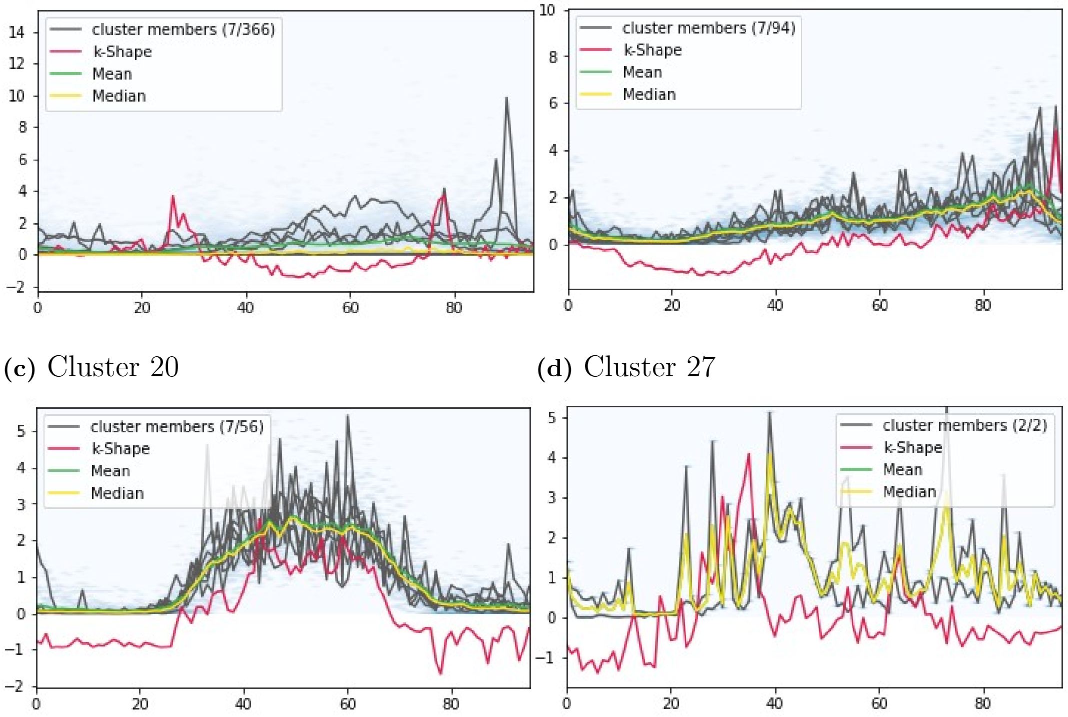
Table 6.1: k Shape clustering on Counter 0

counter 0	N = 9372 K = 30		N = 9372 K = 320		N = 9372 K = 1280	
size(clusters) $\geq$ 1	30		255		798	
size(clusters) = 1	0		18		188	
	Mean/Sample	Bits	Mean/Sample	Bits	Mean/Sample	Bits
Original value	205154.1	16.5	205377.1	16.5	206950.2	16.5
Residual to center averages:						
Kshape	256740.9	17.3	235643.3	17.4	235545.2	17.4
ED	88718.8	15.8	84386.4	15.8	79742.8	15.6
Mean	88718.8	15.8	84386.4	15.8	79742.8	15.6
Median	85711.8	15.5	82210.8	15.2	76574.9	14.7
Shift residual to center averages:						
shift kshape	189800.1	17.1	213003.6	17.3	216047.8	17.3
shift ED	85672.0	15.8	80899.0	15.7	77059.5	15.6
shift Mean	85672.0	15.8	80899.0	15.7	77059.5	15.6
shift Median	82510.7	15.5	79159.6	15.3	74382.9	14.9

Table 6.2: k-means clustering on Counter 0

Counter 0	N = 9372 K = 30		N = 9372 K = 320		N = 9372 K = 1280	
	Mean/Sample	Bits	Mean/Sample	Bits	Mean/Sample	Bits
size(clusters) $\geq$ 1	29.0		197.0		456.0	
size(clusters) = 1	1.0		123.0		824.0	
Original value	205175.9	16.5	207821.7	16.7	222119.8	17.1
Residual to center averages:						
KMeans	79699.7	15.6	70038.6	15.6	68617.3	15.6
Median	77068.2	15.4	67832.4	15.2	66102.5	15.0

Figure 6.1: Normalized data from *k*-Shape clustering (N=9372, K=1280, Counter 0).



(a) Cluster 0

(b) Cluster 3

(c) Cluster 20

(d) Cluster 27

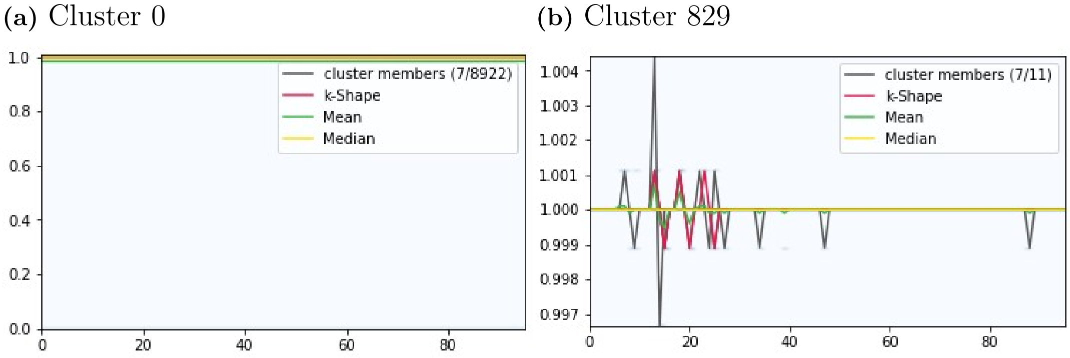
Table 6.3: *k*-Shape clustering on Counter 29

counter 29	N = 9372 K = 30		N = 9372 K = 320		N = 9372 K = 1280	
	Mean/Sample	Bits	Mean/Sample	Bits	Mean/Sample	Bits
size(clusters) ≥ 1	1.0		12.0		26.0	
size(clusters) $= 1$	0.0		1.0		7.0	
Original value	76101761.4	27.3	76100278.4	27.3	76110633.9	27.3
Residual to center averages:						
Kshape	26517.9	0.7	26317.8	0.6	23050.2	0.5
ED	1162540.7	21.4	1165074.0	21.0	1163433.3	21.0
Mean	1162540.7	21.4	1165074.0	21.0	1163433.3	21.0
Median	26517.9	0.7	26285.0	0.5	23191.6	0.5
Shift residual to center averages:						
shift kshape	26517.9	0.7	26316.2	0.6	22492.1	0.5
shift ED	1162541.1	21.4	1165056.0	21.0	1163644.6	21.0
shift Mean	1162541.1	21.4	1165056.0	21.0	1163644.6	21.0
shift Median	26517.9	0.7	26283.2	0.5	23820.9	0.5

Table 6.4: *k*-Means clustering on Counter 29

Counter 29	N = 9372 K = 30		N = 9372 K = 320		N = 9372 K = 1280	
	Mean/Sample	Bits	Mean/Sample	Bits	Mean/Sample	Bits
size(clusters) ≥ 1	7.0		45.0		45.0	
size(clusters) $= 1$	23.0		127.0		127.0	
Original value	76113892.1	27.3	76102955.7	27.3	76102955.7	27.3
Residual to center averages:						
KMeans	300.6	7.4	0.0	0.0	0.0	0.0
Median	205.8	0.5	0.0	0.0	0.0	0.0

Figure 6.2: Normalized data from *k*-Shape clustering (N=9372, K=1280, Counter 29). 8922 of the time series can be described as a constant line or 0, see (a). The heatmap show a small fraction of data at y=0, dragging down the mean center slightly. The rest of the clusters with contains series with perturbations such as the one in (b)


(a) Cluster 0

(b) Cluster 829

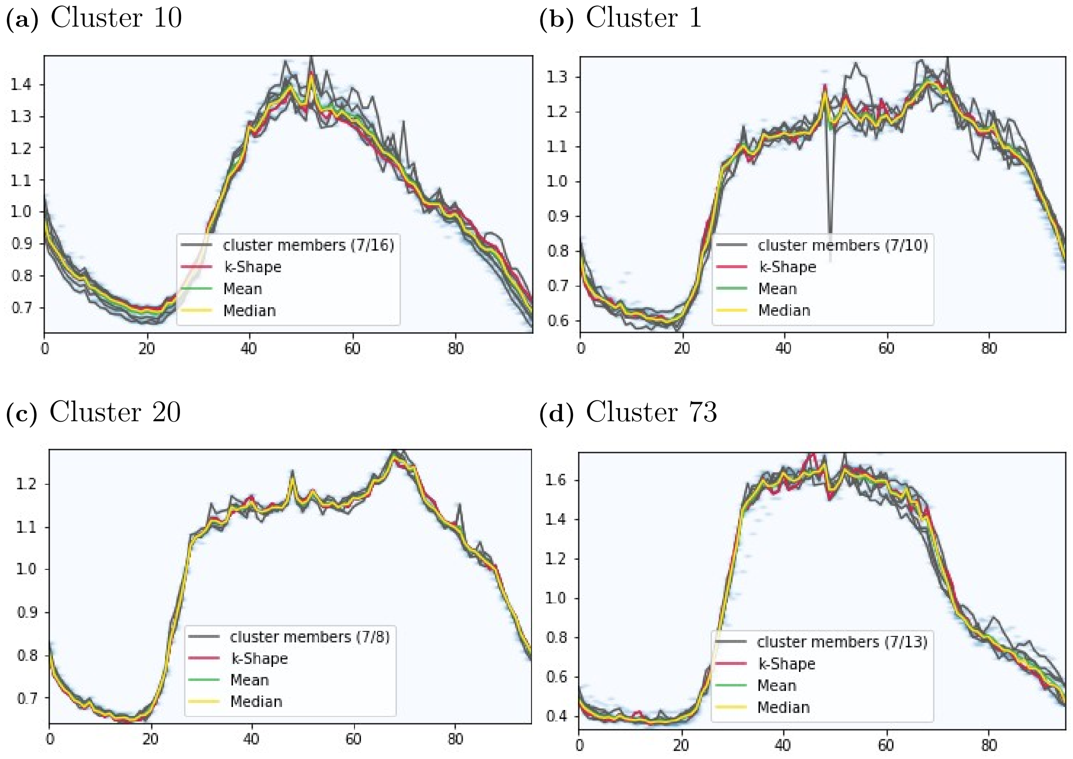
Table 6.5: *k*-shape clustering on Counter 32

Counter 32	N = 9372 K = 30		N = 9372 K = 320		N = 9372 K = 1280	
	Mean/Sample	Bits	Mean/Sample	Bits	Mean/Sample	Bits
size(clusters) ≥ 1	29.0		302.0		1056.0	
size(clusters) $= 1$	0.0		12.0		123.0	
Original value	821056.5	20.8	821303.1	20.8	821282.5	20.8
Residual to center averages:						
Kshape	21245.5	14.9	13363.5	13.7	9105.6	11.9
ED	20206.0	14.8	13395.5	14.1	9066.2	13.3
Mean	20206.0	14.8	13395.5	14.1	9066.2	13.3
Median	20453.2	14.7	13211.2	13.6	8595.7	12.2
Shift residual to center averages:						
shift kshape	21198.0	14.9	13293.0	13.7	9052.0	11.9
shift ED	20156.5	14.8	13332.6	14.1	9056.0	13.3
shift Mean	20156.5	14.8	13332.6	14.1	9056.0	13.3
shift Median	20404.8	14.7	13136.5	13.6	8568.3	12.2

Table 6.6: *k*-means clustering on Counter 32

Counter 32	N = 9372 K = 30		N = 9372 K = 320		N = 9372 K = 1280	
	Mean/Sample	Bits	Mean/Sample	Bits	Mean/Sample	Bits
size(clusters) ≥ 1	25.0		302.0		889.0	
size(clusters) $= 1$	5.0		18.0		391.0	
Original value	821285.7	20.8	821607.2	20.8	830411.9	20.8
Residual to center averages:						
KMeans	16925.1	14.7	10036.4	13.8	6972.0	13.1
Median	16756.6	14.6	9754.2	13.4	6566.4	12.1

Figure 6.3: Normalized data from *k*-Shape clustering (N=9372, K=1280, Counter 32).



(a) Cluster 10

(b) Cluster 1

(c) Cluster 20

(d) Cluster 73

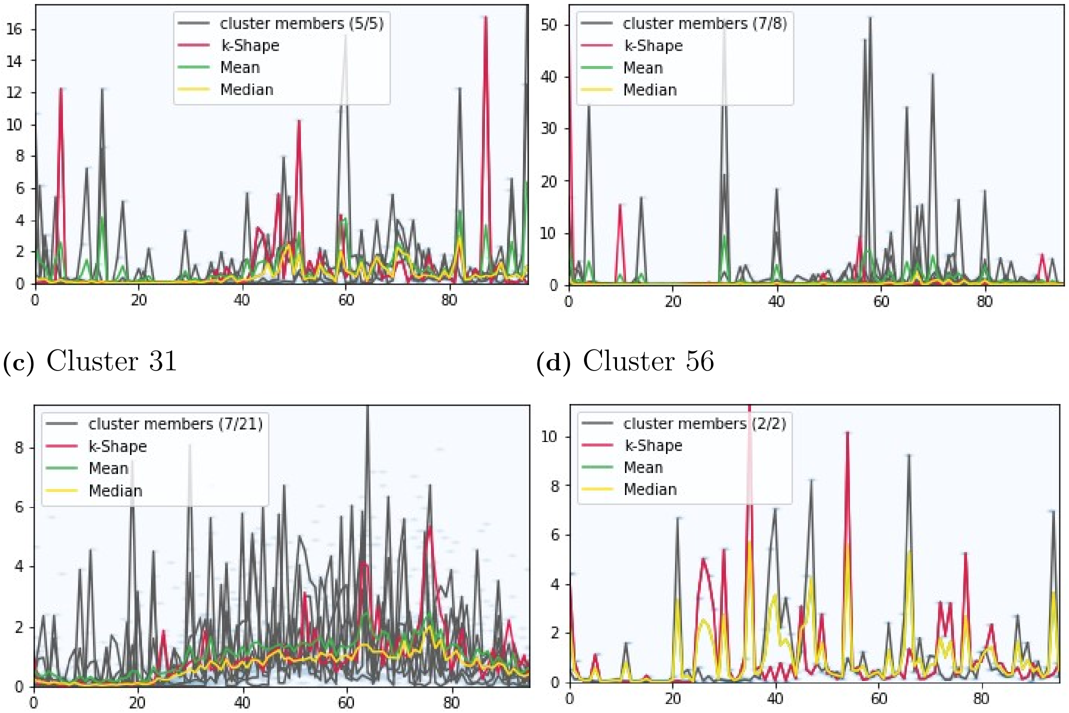
Table 6.7: *k*-shape clustering on Counter 59

Counter 59	N = 9372 K = 30		N = 9372 K = 320		N = 9372 K = 1280	
	Mean/Sample	Bits	Mean/Sample	Bits	Mean/Sample	Bits
size(clusters) ≥ 1	30.0		300.0		1017.0	
size(clusters) $= 1$	0.0		6.0		152.0	
Original value	76476.7	13.2	76470.6	13.2	76320.9	13.2
Residual to center averages:						
Kshape	95860.2	15.2	102879.1	13.6	92910.2	12.4
ED	82639.9	15.6	81839.1	15.4	78153.7	14.9
Mean	82639.9	15.6	81839.1	15.4	78153.7	14.9
Median	65693.2	13.7	63065.4	13.3	60342.7	12.7
Shift residual to center averages:						
shift kshape	75158.9	14.9	75231.6	13.6	59255.5	12.4
shift ED	80515.9	15.6	79565.8	15.4	77184.4	15.0
shift Mean	80515.9	15.6	79565.8	15.4	77184.4	15.0
shift Median	65363.2	13.9	63275.8	13.7	61554.8	13.5

Table 6.8: *k*-means clustering on Counter 59

Counter 59	N = 9372 K = 30		N = 9372 K = 320		N = 9372 K = 1280	
	Mean/Sample	Bits	Mean/Sample	Bits	Mean/Sample	Bits
size(clusters) ≥ 1	30.0		308.0		862.0	
size(clusters) $= 1$	0.0		12.0		418.0	
Original value	76476.7	13.2	76559.5	13.2	78172.2	13.4
Residual to center averages:						
KMeans	75479.0	15.4	61783.0	14.9	56810.2	14.6
Median	62469.2	13.6	51785.3	13.2	48397.3	12.9

Figure 6.4: Normalized data from *k*-Shape clustering (N=9372, K=1280, Counter 59).



(a) Cluster 1

(b) Cluster 10

(c) Cluster 31

(d) Cluster 56