

Noise Detection Pattern Recognition in a Univariate Time Series Case

Carlos Callejo Peñalba

School of Science

Thesis submitted for examination for the degree of Master of
Science in Technology.

Espoo 13.8.2019

Supervisor

Prof. Juho Kannala

Advisor

Er. Aymen Mouelhi

Copyright © 2019 Carlos Callejo Peñalba



Author Carlos Callejo Peñalba

Title Noise Detection Pattern Recognition in a Univariate Time Series Case

Degree programme Data Science

Major Data Science

Code of major SCI3095

Supervisor Prof. Juho Kannala

Advisor Er. Aymen Mouelhi

Date 13.8.2019

Number of pages 57+1

Language English

Abstract

The goal of this project is to find a specific time series pattern in diverse univariate time series data. The data comes from memory reports of computing servers. These reports show the amount of memory that is in use in the servers, as well as the total memory available.

The specific pattern that was required to find was noise, which consists of periods of time in which there are high fluctuations of in-use memory.

The algorithms that detect these patterns have been manually developed from scratch, due to the diverse nature of the data with which more control was needed. Some of the developed algorithms are a custom peak and valley detector, a noisy segment calculator by two different methodologies (which will be compared), and a daily seasonality detector.

In the results, we get all the systems with noise of the customer, a score of noisiness of each one, and a ranked list of those from the noisiest to the least noisy. An analysis of the two or three noisiest cases is performed, depending on the methodology used.

Additionally, a deployment of part of the work is done on a tool to assess its feasibility. This proof of concept will potentially allow deploying all current and future developments on this tool.

Keywords Time series, Pattern recognition, Segmentation, Noise, Peaks

Preface

I want to thank Professor Juho Kannala and my instructor Er. Aymen Mouelhi for giving me this opportunity to work with them, as well as for their much appreciated guidance and help.

Espoo, 13.8.2019

Carlos Callejo Peñalba

Contents

Abstract	3
Preface	4
Contents	5
1 Introduction	7
2 Background	8
2.1 Overview of SAP	8
2.2 SAP HANA	9
2.2.1 Column-oriented	9
2.2.2 High-level Architecture	10
2.2.3 Memory Management	11
2.3 SAP EarlyWatch Alert	12
2.4 Previous Work	13
2.5 Time Series Pattern Recognition Techniques	15
3 Data Description and Retrieval	17
3.1 Data Description	17
3.2 Data Nature	18
3.3 Averaging Technique	20
3.4 Data Retrieval	21
3.5 Data Preprocessing	23
4 Objectives	25
4.1 Primary Objective: Time Series Pattern Recognition	25
4.2 Secondary Objective: Deployment on SAP Data Hub	26
5 Algorithms	27
5.1 Peak Detector	27
5.2 Noisy Segment Detector	30
5.2.1 Noisy Segment Detector by Peak Closeness	30
5.2.2 Noisy Segment Detector by Peak Closeness and Memory Distance	32
5.3 Noisy Segment Occupancy Scorer	33
5.4 Daily Seasonality Detector	35
6 Results	39
6.1 Daily Seasonal Systems	39
6.2 Peak and Valley Detection	39
6.3 Comparison of Noisy Segment Detection Algorithms	39
6.4 Most Noisy Systems with High Sensitivity Peak Detection	41
6.5 Most Noisy Systems with Low Sensitivity Peak Detection	42
7 Deployment on SAP Data Hub	51

	6
8 Discussion	55
9 Conclusion	56
References	57
A Autocorrelation Plots of the Noisiest Systems	58

1 Introduction

Digital Process Automation is a goal that many companies strive to achieve. It uses digital technology components to automate (as much as possible) any kind of human intervention within workflows. This offers multiple advantages, such as offering consistent and faster results, better tracking of the full workflow and, of course, lowering operating costs, since we might replace a person that was manually working in a workflow by a digital process. Additionally, if the outcome of the process is for the final customer, then we might be improving the user experience as well.

The work in this master thesis is the result of a 6-month internship in a big German IT company, which needs to automate some of the workflows within one of its products. This product consists in a platform that provides information to the customer about the state of their servers, such as memory usage levels, system errors, warnings, etc. Currently, if the customer has a problem with a server, an employee of the company has to manually identify what and where is the problem to help the customer. This is highly costly in terms of resources and time, and for this reason, the company wants to automate this human-labour process.

In this project, the data treated is memory usage reports coming from each customer server. These reports provide the system memory in use and the total memory available. This data is hourly based, and therefore, it is time series data.

The goal of the project is to find a concrete time series pattern in customer data. This pattern is called noise, and it happens when there are high memory usage fluctuations during a short period of time. This unstable scenario is usually unwanted because it indicates the possibility that there is something that is not working properly in the customer server. One of the consequences of having this instability is that, if the fluctuations are very high and the peaks of these are close to the total available memory limit of the server, then the server might have unloaded data tables every time the in-use memory was close the total available memory. If the system memory is almost full, unloading data is required to store new one, but it drastically affects the performance of the server for future data operations. We want to avoid this situation at all costs.

Once these noisy patterns are found, there will be a noisiness ranking process for all the servers of the customer. A score will be given to each one of the servers, which will allow to rank them from the noisiest to the least noisy. This ranking will be the main result of the work.

Although it is out of scope of the project, getting the noisiest cases will help in future developments or research to prioritize the study of these cases, identifying why the systems have noise, and proposing a solution to fix this behaviour automatically. This is extremely important for the company because, as it was exposed before, they want to help the customer without depending on their employees.

Lastly, and in addition to the main project, a deployment of part of the work will be done on one of the tools of the company in order to assess the feasibility of this tool. If feasible, all current and future developments will be deployed on this tool because it has high scalability properties, which are needed by the company when they release the full service for the customer.

2 Background

In this section we will introduce an overview of the company (SAP), one of its main products (HANA) and some of its technical aspects, what it is SAP EarlyWatch Alert and why it matters, and some of the work of the previous intern at the company.

2.1 Overview of SAP

SAP (*S*ysteme, *A*nwendungen und *P*rodukte in der Datenverarbeitung, “Systems, Applications & Products in Data Processing”) is a German multinational software corporation that makes enterprise software to manage business operations and customer relations. It is third largest software and programming company of the world. SAP headquarters are in Walldorf, Germany, with regional offices in 180 countries. The company has over 425,000 customers and is a component of the Euro Stoxx 50 stock market index.

SAP focuses on 25 industries and six industry sectors: process industries, discrete industries, consumer industries, service industries, financial services, and public services. It offers integrated product sets for large enterprises, mid-sized companies, and small businesses. SAP has strong partnerships with other tech giants such as Intel, IBM, Microsoft, and Apple.

SAP ECC (**E**nterprise Resource Planning **C**entral **C**omponent) is the flagship product of the company. This is where transactions of the customer are executed and stored. All the main modules SD (Sales and Distribution), MM (Materials Management), FI (Finance), PS (Project System), CO (Controlling), and many more, are part of ECC. It was called R/3 earlier, then renamed ERP (after some upgrades), and at present is called ECC.

Other surrounding products of the main ECC system are SAP BI (Business Intelligence), CRM (Customer Relationship Management), SCM (Supply Chain Management), SRM (Supplier Relationship Management), PI (Process Integration), and others.

SAP HANA (**H**igh-performance **A**Nalytic **A**ppliance) is one of the latest products of the company and has become one of its greatest commercial success. Around 40% of all the customer transactions are made through HANA. It is the fastest growing product in the history of SAP and it has become a crucial product for them. Basically, it consists in a system in which the entire database is loaded in a high-capacity RAM (Random Access Memory) and where the database architecture is (mostly) column-oriented. This makes the transactions be processed extremely fast. HANA was introduced for BI but now ECC and other systems are also available on this in-memory database architecture. The tremendous success of HANA is making the company to invest high amounts of money and resources in it, with the final goal of having all their products and solutions integrated and running on HANA (and on the cloud). This will eventually make HANA the core product of the company. The current state of the HANA platform, in a business perspective, is shown in Figure 1.

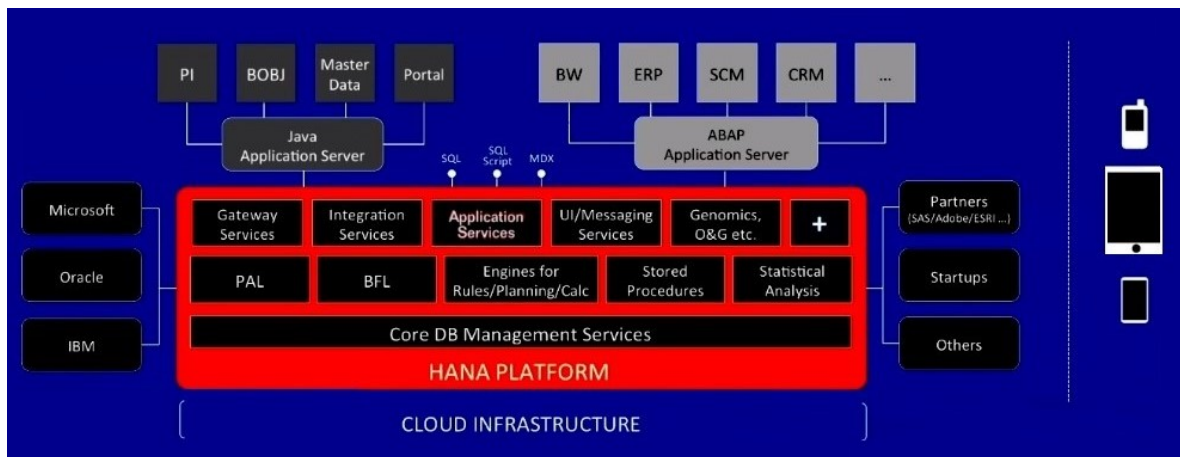


Figure 1: Business overview of the current state of HANA.

2.2 SAP HANA

SAP HANA is an in-memory, column-oriented, relational database and database management system developed and marketed by SAP SE [1]. Its primary function is to store and retrieve data as requested by the applications. In addition, it performs advanced analytics (predictive analytics, spatial data processing, text analytics, text search, streaming analytics, graph data processing, etc.) and includes extract, transform, and load capabilities as well.

The in-memory database technology allows the processing of massive amounts of real-time data in a short period of time. The in-memory computing engine allows HANA to process data stored in RAM as opposed to reading it from a regular disk. This allows the application to provide instantaneous results from customer transactions and data analysis.

Characteristics:

- Distributed system.
- In-memory (RAM-based).
- Column-oriented, but also row-oriented for some data merging tasks.
- It uses its own SQL language, called SQLScript.
- It handles both OLTP (Online Transaction Processing) and OLAP (Online Analytical Processing) capabilities.

2.2.1 Column-oriented

A good way of explaining what a column-oriented database is, is to compare it against the traditional row-oriented ones:

Row-oriented data stores:

- Data is stored and retrieved in one row at a time; hence, even if we only require getting part of the data of the row, the whole row is loaded, which affects the performance.
- Records in row-oriented data stores are easy to read and write.
- Row-oriented data stores are best suited for online transaction processing (OLTP) since the focus is on processing the data, and not on analyzing it.
- They are not efficient in performing operations applicable to entire datasets; therefore, aggregation in row-oriented data stores is an expensive job or operation.
- Compression mechanisms are poor.
- Example: relational databases.

Column-oriented data stores:

- Data are stored and retrieve in columns.
- Read and write operations are slower compared to row-oriented.
- Column-oriented stores are best suited for online analytical processing (OLAP) since the focus is on analysing the data, and not on processing it.
- They are efficient in performing operations applicable to the entire dataset, and thus, it enables efficient aggregation over many rows and columns.
- This type of data stores basically permits high compression rates due to little distinct or unique values in columns.
- Examples: HBase and HANA databases.

The Figure 2 shows a visual and practical example of how data is stored in memory when we use both types of databases.

2.2.2 High-level Architecture

A running HANA system consists of multiple communicating processes (services). Figure 3 shows the main HANA database services in a classical application context.

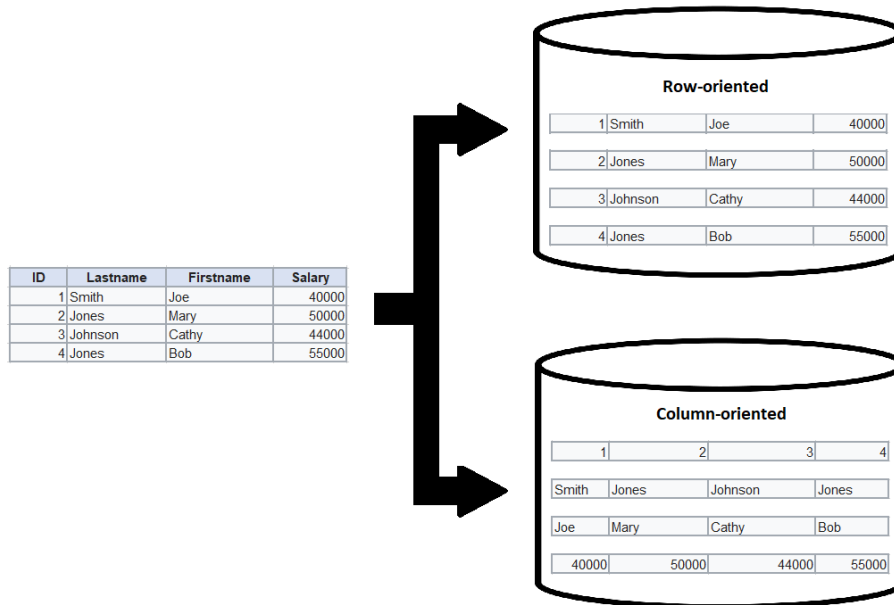


Figure 2: Row-oriented vs column-oriented database storages.

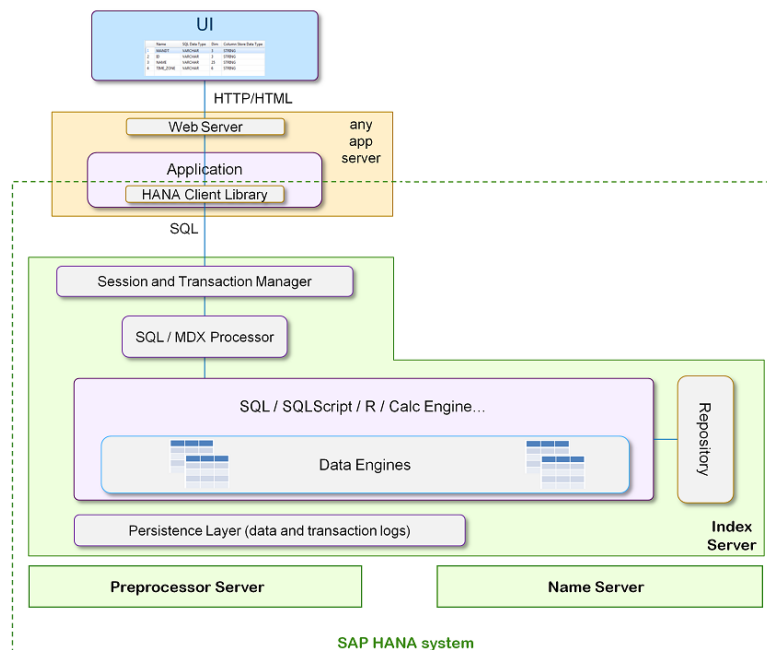


Figure 3: HANA database high-level architecture.

2.2.3 Memory Management

SAP HANA pre-allocates and manages its own memory pool, used for storing in-memory tables, for thread stacks, and for temporary results and other system data structures [2]. This is shown in the Figure 4.

When more memory is required for table growth or temporary computations, the

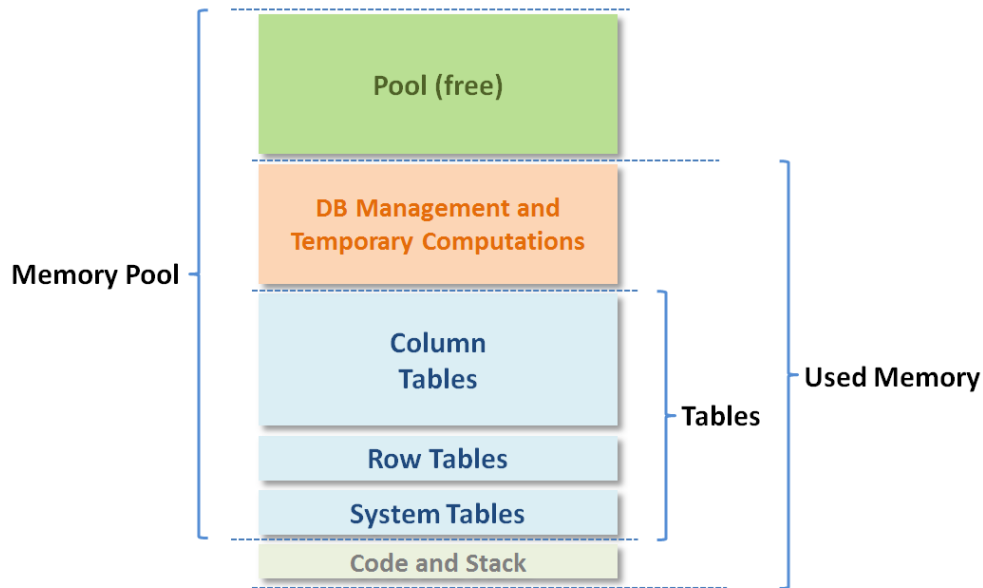


Figure 4: HANA used memory.

HANA memory manager obtains it from the pool. When the pool cannot satisfy the request, the memory manager will increase the pool size by requesting more memory from the operating system, up to a predefined *Allocation Limit*.

Memory is a finite resource. Once the allocation limit has been reached and the pool exhausted, the HANA memory manager will no longer be able to allocate memory for internal operations without first giving up something else. In fact, this is exactly what will happen: buffers and caches will be released, and column store tables will be unloaded, column by column, based on a least-recently-used order. When tables are partitioned over several hosts, this is managed per host; that is, column partitions will be unloaded only on hosts with acute memory shortage.

Despite all these abilities, it is theoretically possible that the memory manager will still face a need for more memory that it cannot satisfy. For instance, when too many concurrent transactions use up all memory, or when a particularly complex query performs a cross-join on very large tables, creating a huge intermediate result that exceed the available memory. Such situations can even lead to an *Out Of Memory* failure.

2.3 SAP EarlyWatch Alert

SAP EarlyWatch Alert is an automatic service that monitors the essential administrative areas of an SAP system. Alerts indicate critical situations and give solutions to improve performance and stability. EarlyWatch Alert Workspace, which is shown in Figure 5, displays these alerts.

At present, there is a need for an SAP employee to manually check the health status of the HANA system (or group of systems) of the customer. SAP wants to automate all these processes as much as they can, with the ultimate goal of not

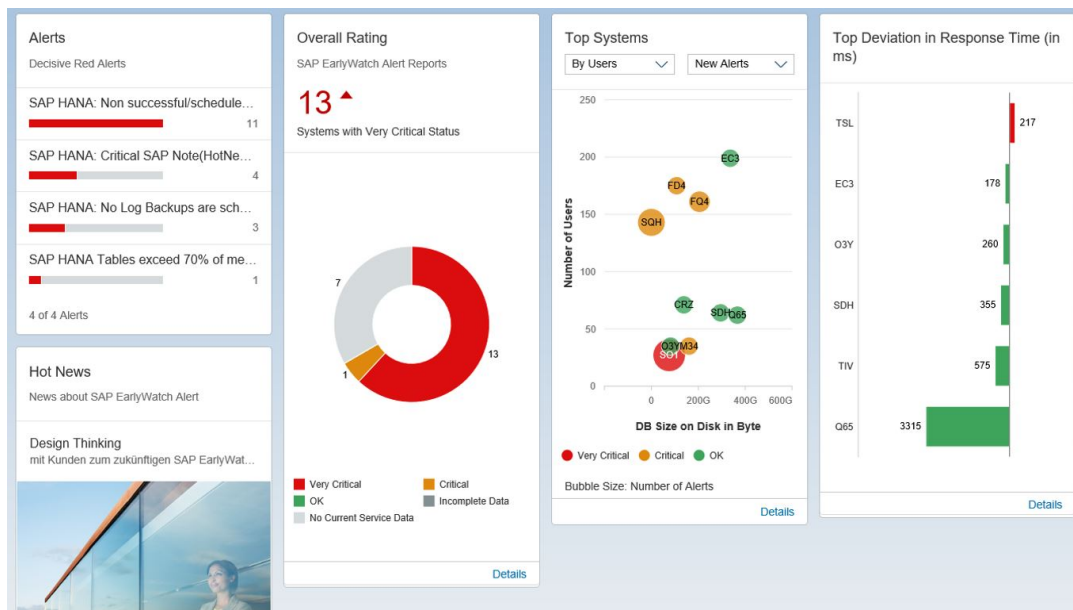


Figure 5: EarlyWatch Alert Workspace.

having any employee doing all these manual checks for the customer. In other words, the customer, by himself, should be able to understand and react to all the possible problems and warnings of the systems, thanks to the output that EarlyWatch Alert is providing.

At the current stage of development of EarlyWatch Alert, the system only provides alerts or warnings of the problems that have already happened, but it is not really able to provide predictive alerts of future problems that can potentially happen.

Having said this, SAP wants to develop algorithms that will analyze and process historical data for past-related outcomes, and other algorithms that will use the historical data for future-related outcomes (i.e., forecasting). The former will help to progress even further on what EarlyWatch Alert already does, and the latter will contribute to the new predictive goals. In any case, both help to achieve the automation that the company is pursuing with EarlyWatch Alert.

In my work, I will focus on getting past-related outcomes; this will be detailed in the *Objectives* chapter.

2.4 Previous Work

Detection d'Anomalies dans des Systemes SAP (“Anomalies Detection in SAP Database”) [3] is the title of the project carried out by the previous intern. He was working with the same HANA memory usage time series data, as well as with HANA *response time* time series data, which is out of scope in this work. As in the case of what this work pursues, he developed new ways to process and analyze data with the same goal of progressing in the development of EarlyWatch Alert.

These are some of the main developments he worked on: *last major change point detector*, and *root-causes analysis*.

Last Major Change Point Detector

This algorithm detects the last moment of the time series in which there is a significative memory change pattern, which in most of the cases is a system restart. This is really useful when we are doing forecasting, because instead of doing a forecast taking the whole time series dataset as training, we just take a subset of it from the moment in which this major change point was detected to the most recent sample. This leads to a better forecast, since in most of the cases after a system restart or other kind of major change point the memory data behaves differently to how was behaving before; thus, it makes sense to just use the latest data and not the whole dataset.

Without going into much detail, in this algorithm different signal processing techniques were applied in the following order:

1. A Savitzky-Golay filter was used.
2. A convolution between the time series and a Heaviside step function was performed.
3. The first order difference of the convolution was calculated twice; therefore, we get the second derivative of the signal.
4. The natural logarithm of the second derivative is calculated.
5. The date and time when the last significative change point of the last processed signal occurs is obtained. If this moment is quite recent, a previous not-so-recent change point is selected. The effective last major change point will be 30 hours after the change point real datetime.

Figure 6 shows the final result of this process on a time series example.

Root-causes Analysis

In this process the objective is to detect why a system has increased its memory consumption. There are two main possibilities for this: the amount of data to process was just higher, or there was a problem with the memory allocators.

First, we obtain when the memory allocators took place and the category of those in a HANA table called 'HN ALLOCA'. Once we have the allocators, a manual tagging process of these allocator categories is done to assign them a *severity score*.

We compare each full time series with an exponential growth time series signal via DTW (Dynamic Time Warping), which is a time series shaped-based comparator. In the cases in which the time series resembles to an exponential growth, an analysis of the allocators will be carried out. The severity score of the allocators of the time series will determine how good or bad these systems are performing. It is also possible to do a ranking of all the allocators by their occurrence, as the Figure 7 shows.

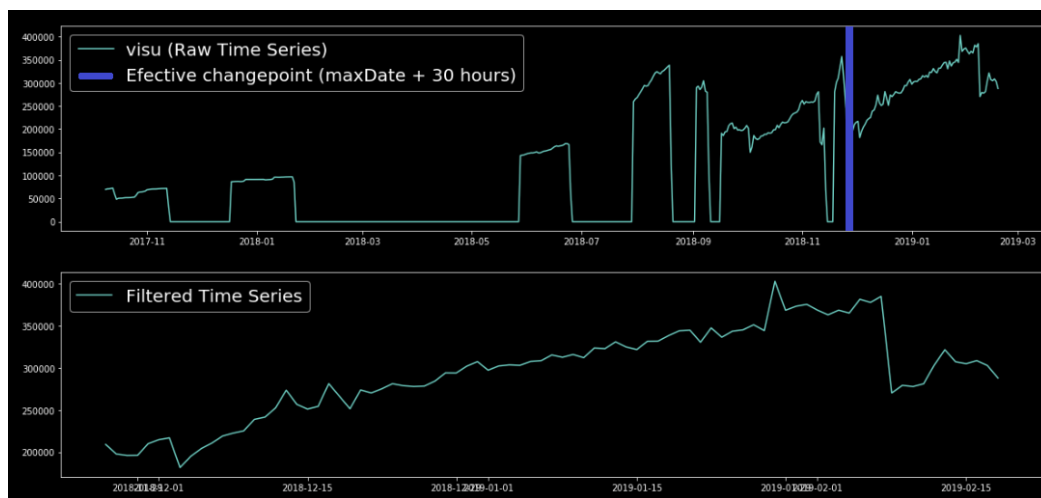


Figure 6: Results of the Last Major Change Point Detector applied on a time series example. The upper part shows the full time series, which contains several change points. The found last major change point is represented as the thick blue line. In the lower part, we see a subset of the original time series that starts 30 hours after the found last major change point.

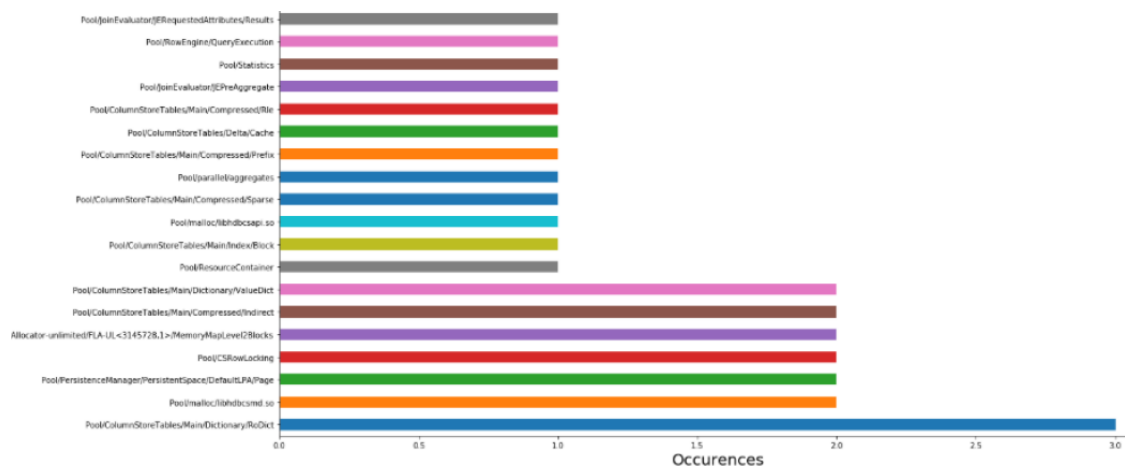


Figure 7: Ranking of the memory allocators based on their occurrence in 10 HANA systems.

2.5 Time Series Pattern Recognition Techniques

Because in this work we perform pattern recognition, a research about the topic was carried out beforehand in order to get an introduction in the topic and to decide what were the best approaches for the goals of the project, which will be exposed in the *Objectives* Chapter.

In time series, if we want to find out if a time series follows a specific type of shape, shift, or other kind of characteristics, we will need to make use of algorithms that evaluate the similarity between that time series and, at least, another one that will act as a reference [4]. These time series similarity characteristics are many and

they are divided in four main groups: *time-based* similarity, *shape-based* similarity, *change-based* similarity, and *structure-based* similarity. For this project, shape-based and structure-based time series similarities are the most interesting for the goals of the project.

In shape-based similarities, we have *Euclidean Distance* [5] and *Dynamic Time Warping* (DTW) [6]. Euclidean distance is strong time dependant, whereas DTW is weak. Both are quite effective to evaluate how similar the shape of two time series are, but they are not effective with long and noisy data (quite many of the time series of our data are unfortunately like this). Then we have other types of shaped-based similarity approaches, such as *Shapelet* [7], which is time independent and phase independent.

Then, in structure-based similarity approaches, we mainly find the *Symbolic Aggregate Approximation* (SAX) algorithm [8] that performs dimensionality reduction and indexing with a lower-bounding distance measure. This one works really well with long time series (and it is efficient), but you lose the time structure of the time series (therefore time is not considered at all) and it only allows comparing time series that have exactly the same length.

Once we get the similarity scores after comparing two or more time series, we can perform time series clustering and/or classification. For example, if we want to build a classifier for SAX, we can use *SAX-VSM* (where VSM means Vector Space Model) [9]. As another example, for DTW we can use the *1NN-DTW* classifier (first Nearest Neighbor-DTW) [10], which is an exceptionally competitive classifier. We can find much more ways to classify these time series, including by using Hidden Markov Models, Convolutional Neural Networks, more traditional approaches, etc.

As we can see, going into detail in all those requires much time and effort, and as we will see later when we understand how the data used in this work is, it is not evident which one of the algorithms to use; and even if we choose one algorithm, that does not mean it will work for all the time series cases of our dataset, which is quite diverse.

For this reason, we will follow a manual approach to detect the target patterns of the time series of this work. For that, we have developed from-scratch algorithms that calculate exactly what we want to get, for all the different types of time series that our dataset has.

3 Data Description and Retrieval

This chapter will first introduce the features of the data, then its nature, the averaging technique that will be used in the data retrieval, the data retrieval, and lastly, the data preprocessing.

3.1 Data Description

A time series is a series of data points indexed in time order. Time series are very frequently plotted via line charts. They are used in statistics, signal processing, pattern recognition, econometrics, mathematical finance, weather forecasting, earthquake prediction, electroencephalography, control engineering, astronomy, communications engineering, and largely in any domain of applied science and engineering which involves temporal measurements.

The features of the time series data that is used in this work are the following:

- *CLIENT*: (Key). This attribute is useless since all its rows always have value 1. We will not use it.
- *SESSNO*: (Key). It is the ‘session number’. A new session is created when the customer sends new data to process.
- *CONINS*: (Key). It is the ‘context instance’. Each system can have one or more instances running.
- *ROW_NUM*: (Key). This attribute counts the number of samples (1, 2, 3, ...) for one system with specific *CLIENT*, *SESSNO* and *CONINS*. It acts like an index. In our case, it will be useless since it is not providing us any extra value (we will use datetime values as index). This attribute seems more like a requirement for storing the data in the database.
- *SYSTNO*: It is the ‘system number’. It is the system identifier in the distributed system, which we usually have. Each system can have a different number of instances running.
- *CALWEEK*: It is a combination of the year and week number of the year when a sample was taken. We will not use this attribute because other features (*ZEW_CMDY* and *ZEW_CMHR*), together, provide more precise datetime information.
- *ZEW_CMDY*: It provides the year, month, and day of a sample.
- *ZEW_CMHR*: It provides the hour of a sample (thus from 0 to 23 values only). There are no more time attributes that indicates a more precise time when a sample was taken, so we will have hourly data at most.
- *ZEW_IUM*: It is the ‘instance memory used’. It shows the in-memory (RAM-type) data amount that is *currently in use* by the HANA system. This value will be in constant change over time.

- *ZEW_GALIMIT*: It is the ‘global allocation limit’. This value indicates the maximum amount of in-memory (RAM-type) memory that the system (and its operating system) can provide to HANA for its usual storage and processing purposes. The customer of the HANA system can freely change this value to provide more or less available memory for HANA. The value usually remains constant over time until the customer decides to change the value and, after the change, the value will remain constant at that new memory amount. It is extremely rare to see systems in which the *ZEW_GALIMIT* dynamically changes over time, but there are some cases. An important remark is that the *ZEW_IUM* can never surpass the *ZEW_GALIMIT*.

As it can be inferred from having the *ZEW_CMHR* feature, the time series is hourly based. This means that every hour a measurement of the data values of the customer systems is performed.

More information about the features such as their type and dimensions can be found in Table 1.

Name	SQL Data Type	Dim	Column Store Data T...	Key	Not Null
CLIENT	NVARCHAR	3	STRING	(X1)	X
SESSNO	NVARCHAR	13	STRING	(X2)	X
CONINS	NVARCHAR	40	STRING	(X3)	X
ROW_NUM	INTEGER		INT	(X4)	X
SYSTNO	NVARCHAR	18	STRING		X
CALWEEK	NVARCHAR	6	STRING		X
ZEW_CMDY	NVARCHAR	8	STRING		X
ZEW_CMHR	NVARCHAR	10	STRING		X
ZEW_IUM	DECIMAL	17,3	FIXED		X
ZEW_GALIMIT	DECIMAL	17,3	FIXED		X

Table 1: Features information provided by the database.

We will usually work with the *ZEW_IUM* and *ZEW_GALIMIT* values of one system over time. In fact, they will not be the values of one system but the average of several systems, as we will explain later. In most of the situations, we will not use the values *ZEW_IUM* and *ZEW_GALIMIT* individually; instead of that, we will use the ratio $ZEW_IUM/ZEW_GALIMIT$ for most of the calculations, leaving the individual values for other tasks like plotting.

3.2 Data Nature

Several examples will be shown in order to explain the nature of the data we will be working with. The concept of time series *noisyness* will be also shown during these examples.

First, in Figure 8 we see how *ZEW_IUM* (the memory in use of the system) naturally changes its value over time. In this case, this value is quite constant over time; thus, we consider this time series as a non-noisy time series. We also see that *ZEW_IUM* has 3–4 important memory drops, most probably due to restarts of the

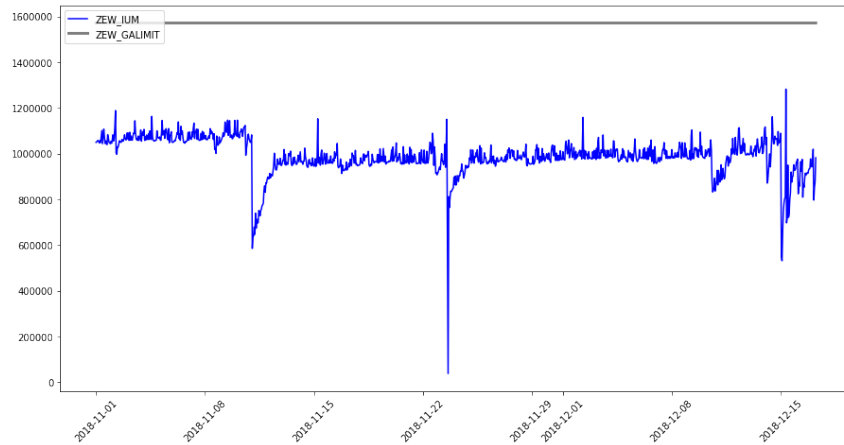


Figure 8: Example of a non-noisy time series.

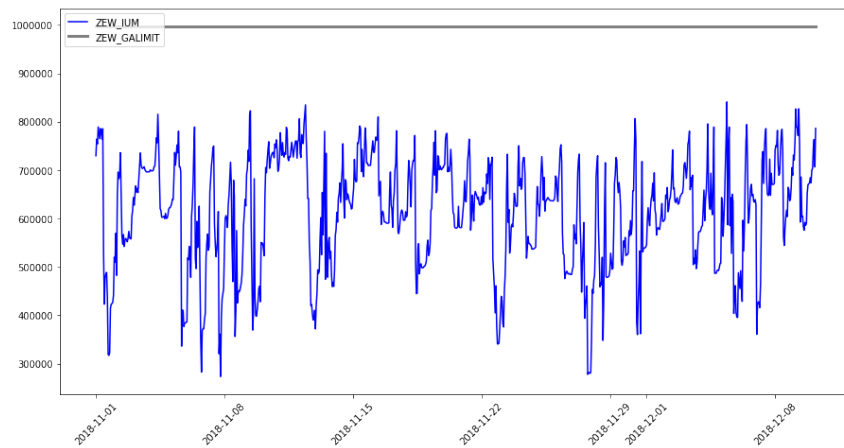


Figure 9: Example of a noisy time series.

system. `ZEW_GALIMIT` (the allocation limit) remains constant since the customer decided not to change this value at any moment.

Figure 9 shows, in contrast with the previous example, a time series in which `ZEW_IUM` changes intensively over time; thus, we consider this time series as a noisy time series. For this reason in this case it is hard to identify if there have been system restarts or not. `ZEW_GALIMIT`, as in the previous example, remains constant since the customer decided so.

In Figure 10, we have data of a system over more than one year and four months. There are many data points missing, specially between the months of February and June in 2018. It does not seem that the system was off when the data was missing, since the `ZEW_IUM` trend goes uphill before and after all those months in a matching manner. Thus, it seems most probably that the system was on, but not sending the time series data to SAP for some reason. At this moment, there is no way for SAP to know if a customer system was off, or if it was on and simply not sending the data to SAP. Regarding the `ZEW_GALIMIT` values, we see that this time the customer decided to change this value several times during the whole time period.

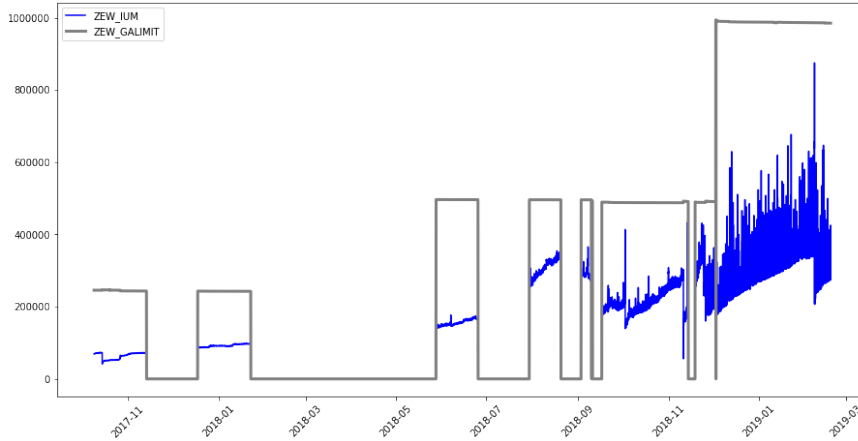


Figure 10: Example of a time series with missing data (1).

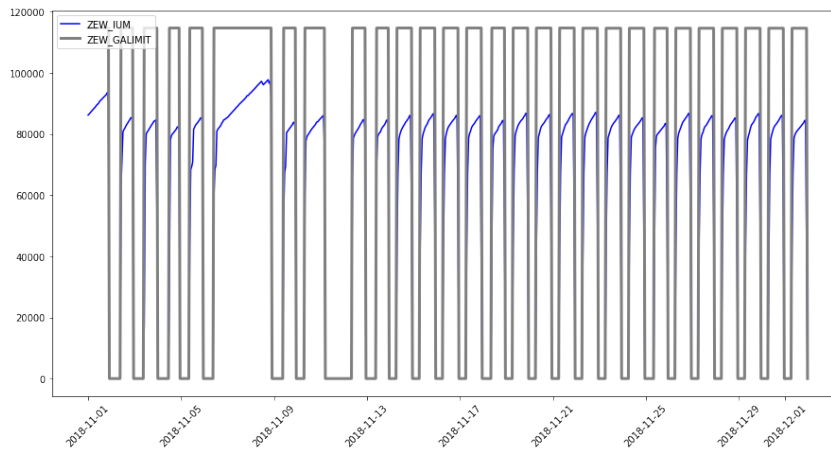


Figure 11: Example of a time series with missing data (2).

Figure 11 shows an example (with length of one month) in which there are many missing data points in a quite consistent manner. Basically, what happens here is that the customer decided to turn off the system almost every night. We see this because the missing data occurs at night time and because the ZEW_IUM values grow every time the system is turned on again (because the system needs to reload and process data).

Lastly, in Figure 12, we have a system that dynamically changes its ZEW_GALIMIT value per hour, which is extremely rare. Additionally, we see that it is a noisy time series, and that there are some missing data points along time.

3.3 Averaging Technique

Since HANA is a distributed system, the data of all the different CONINS (instances) will look highly similar in shape and behavior if we have the same SESSNO and SYSTNO. For this reason, it is a common technique by SAP to perform an averaging process for all these systems/instances. This way, we will end up with a single time

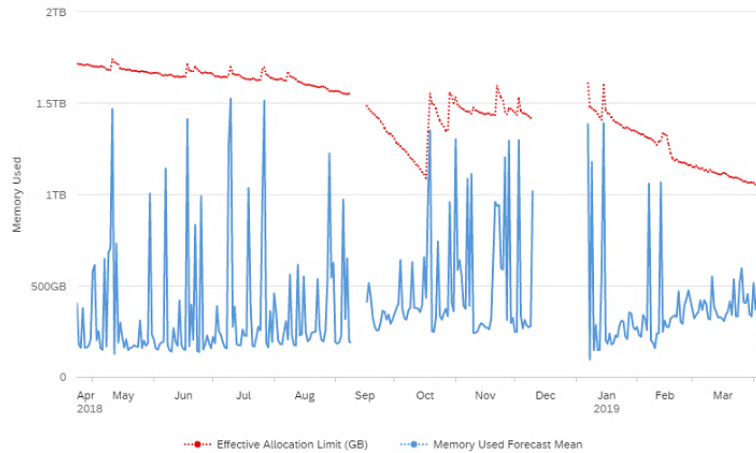


Figure 12: Example of a time series with dynamically changing ZEW_GALIMIT.

series of the whole distributed system that keeps, as a whole, the same meaning than all the individual time series. By doing this averaging process, later calculations will be faster to perform and algorithm results can still be applied for all the individual systems.

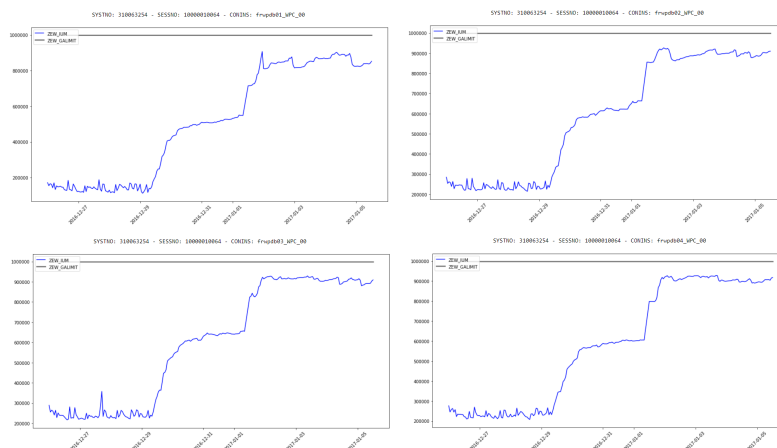


Figure 13: Four systems (out of sixteen) in a distributed system.

As an example, the four first instances/systems of the distributed system with `SYSTNO=310063254` and `SESSNO=10000010064` (the instances with `CONINS` from `frwpdb01_WCP_00` to `frwpdb04_WCP_00`) are plot in the Figure 13. In this distributed system there are in total sixteen instances, with similar shape as the ones showed in Figure 13. If we apply the averaging process to all those instances, we will get the resulting time series shown in Figure 14.

3.4 Data Retrieval

The data is retrieved from a HANA server, which belongs to SAP, that periodically collects the memory usage data from other HANA systems that belong to the

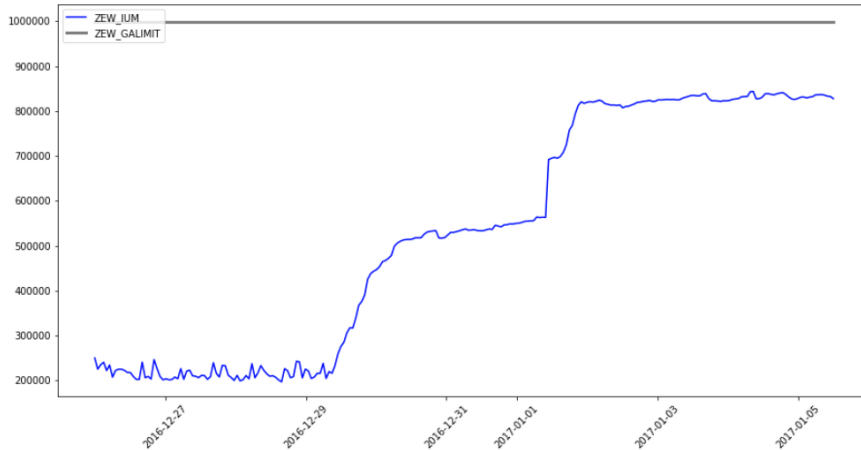


Figure 14: Result after applying the averaging technique over sixteen systems of a distributed system.

customers.

The database schema we will use is called *PHD*, which contains several data tables. In this project, we will only use data from the HANA *MEM1* table. On Friday 2019/5/17, there are around 228,215,000 rows in this table.

We will retrieve and use the first 10,000,000 system-averaged rows of *PHD-MEM1*. The averaging process is directly done in HANA and not locally since HANA computes this operation much faster. The data averaging and data retrieval processes will be performed together in the same query, which is the following:

```
SELECT TOP 10000000 CONINS, SYSTNO, ZEW_CMDY, ZEW_CMHR,
      avg(ZEW_IUM) , avg(ZEW_GALIMIT)
FROM "CBR2PHD" . "CBR2PHD::CBR2PHD_REP.SAPF75_STCSC_HANA_MEM1"
GROUP BY "CONINS" , "SYSTNO" , "ZEW_CMDY" , "ZEW_CMHR"
ORDER BY "CONINS" , "SYSTNO" , "ZEW_CMDY"
```

The size of the output file with the 10,000,000 rows is 667 MB. This amount of data should cover about the 20% of the systems of PHD-MEM1. The reason why the 100% of the data was not retrieved is because it just takes much more time to retrieve and compute. In any case, covering the 100% of the systems of PHD-MEM1 is not really relevant since we can extrapolate the results from the 20% to the full spectrum.

The five first rows of the retrieved dataset are shown in the Table 2. The obtained data shows that the averaging process has been performed and that some meaningless attributes have been suppressed already (CLIENT, SESSNO, ROW_NUM, and CALWEEK). Some preprocessing on the data needs to be performed right after, before running the algorithms. E.g., rows are not currently sort by time.

	CONINS	SYSTNO	ZEW_CMDY	ZEW_CMHR	AVG(ZEW_IUM)	AVG(ZEW_GALIMIT)
0	001hanabp1_HB1_01	312504865.0	20170109	6	195187.0	496669.0
1	001hanabp1_HB1_01	312504865.0	20170109	2	200513.0	496669.0
2	001hanabp1_HB1_01	312504865.0	20170109	5	194962.0	496669.0
3	001hanabp1_HB1_01	312504865.0	20170109	8	195640.0	496669.0
4	001hanabp1_HB1_01	312504865.0	20170109	21	199444.0	496669.0

Table 2: Five first rows of the dataset retrieved from PHD schema and MEM1 table.

3.5 Data Preprocessing

After retrieving the system-averaged data from HANA, it is needed to perform some preprocessing for the data before running the developed algorithms. These steps are done in the following order:

1. We subset the dataset by CONINS and SYSTNO; therefore, we get an unique data frame per CONINS and SYSTNO combination. The next steps will be done for each unique data frame.
2. We combine ZEW_CMDY (date) and ZEW_CMHR (time) into one single attribute, which is called ZEW_CM_FULL (datetime).
3. We drop all possible duplicates of rows based on their datetime (ZEW_CM_FULL), keeping the first row in case of duplicates. Having rows with the same datetime is rare, but when it happens it usually does because there is a change of value in ZEW_GALIMIT.
4. We set ZEW_CM_FULL as the index. This will help to manipulate better the data frames since we are working with time series data.
5. We calculate the ratio between ZEW_IUM and ZEW_GALIMIT (specifically, we perform a $ZEW_IUM / ZEW_GALIMIT * 100$ operation), storing the results in a new column called *IUM_GALIMIT_RATIO*. This feature is really *important* since it allow us to work with the memory values in a scale from 0 to 100, instead of from 0 to undefined. All HANA systems are unique, and they may not have the same ZEW_GALIMIT memory amount. The value of ZEW_GALIMIT can change in one system over time as well. Therefore, with this memory scale from 0 to 100 we can properly work with memory amounts in our algorithms for all the system cases, regardless of the ZEW_GALIMIT value.

A resulting data frame after the preprocessing of the retrieved dataset is shown in the Table 3. Note that the CONINS and SYSTNO to which the data belongs to, will be identified outside of the data frame.

We will also create another data frame for each system, in which we add zero values for ZEW_IUM, ZEW_GALIMIT and IUM_GALIMIT_RATIO when a sample of the time series is missing. This *expanded dataset* is necessary for some

	ZEW_IUM	ZEW_GALIMIT	IUM_GALIMIT_RATIO
ZEW_CM_FULL			
2018-12-10 00:00:00	2296196.0	2846329.0	80.672192
2018-12-10 01:00:00	2296130.0	2846329.0	80.669873
2018-12-10 02:00:00	2296652.0	2846329.0	80.688213
2018-12-10 03:00:00	2297179.0	2846329.0	80.706728
2018-12-10 04:00:00	2297157.0	2846329.0	80.705955
2018-12-10 05:00:00	2317324.0	2846329.0	81.414482

Table 3: First rows of a data frame after the preprocessing.

algorithms such as the daily seasonality classifier, and it is really useful to plot the data frames, since we can easily detect whether the system has had missing values at some point or not. An example of one of this expanded data frames is shown in the Table 4.

	ZEW_IUM	ZEW_GALIMIT	IUM_GALIMIT_RATIO
2018-12-19 08:00:00	2343719.0	2846329.0	82.341816
2018-12-19 09:00:00	2343577.0	2846329.0	82.336828
2018-12-19 10:00:00	0.0	0.0	0.000000
2018-12-19 11:00:00	1298151.0	2846329.0	45.607904
2018-12-19 12:00:00	2280442.0	2846329.0	80.118707

Table 4: Example of a preprocessed and expanded data frame.

4 Objectives

Time series pattern recognition is the main objective of this work, whereas a deployment on a data pipeline engine called *SAP Data Hub* will be the secondary objective. Both will help in the development of SAP EarlyWatch Alert, as explained in the *Background* chapter.

4.1 Primary Objective: Time Series Pattern Recognition

Time series pattern recognition is the main goal in this internship at SAP. I was asked to work on four different cases of pattern recognition:

- Steep decrease.
- Steep increase.
- Constant rise.
- Peaks.

These patterns are clearly shown in the Figure 15.

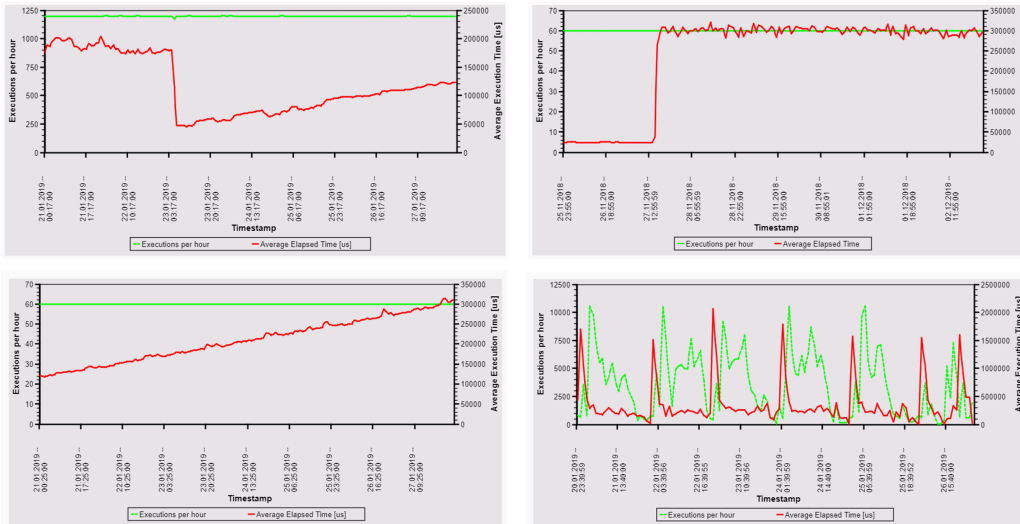


Figure 15: The four time series recognition patterns to work on.

For the *steep decrease* and *steep increase*, we can use the *last major change point detector* algorithm developed by the previous intern, with the goal of detecting when the memory and its trend drastically change. That leaves us only with *constant rise* and *peaks* as target cases.

It is important to consider that *constant rise* is a time period, whereas *peaks* are concrete moments in time. Nevertheless, as it will be shown later in the algorithms section of this document, we will not only be limited to detect these peaks individually as datetimes, but also as time segments/frames, which will be called *noisy segments*.

These noisy segments contain several individual peaks (and valleys, optionally) that are close in time. E.g., if we have 3 or 4 peaks in the same day, we might consider the whole day as ‘noisy’. Thus, in the end, as it happens with the *constant rise* pattern, we will be looking for datetime periods.

4.2 Secondary Objective: Deployment on SAP Data Hub

SAP wants to integrate all the EarlyWatch Alert developments into a new SAP platform called *SAP Data Hub*. This tool is able to process batch and stream data in a pipeline execution. More precisely, the goal is to establish a connection to our HANA server (where all the time series are stored), retrieve the data, process it locally, and then store the results back again in HANA. As part of my internship, I was asked to test this new tool and deploy some of my code on it, performing all the previous operations. If everything works correctly, it will serve as a proof of concept for the company so they can use this tool for their goals. One of the reasons why SAP wants to use this tool is because it is the only way for them to scale all the executions of the EarlyWatch Alert developments to a larger number of systems.

5 Algorithms

In this chapter the algorithms that have been developed will be detailed.

5.1 Peak Detector

Although detecting peaks (and valleys) should be an easy task since there are many available Python-based peak detector algorithms and code libraries published, there were none found that specifically did the peak detection for the natural behavior of our data and goals.

Basically, all peak detectors find absolute peaks, some of them filter these peaks discarding the peaks in which a stablished threshold value distance of the peaks with their immediate left and right neighbors is not surpassed, and none of them can expand the previous filter to include more neighbors than the immediate ones. The example in Figure 16 shows this graphically.

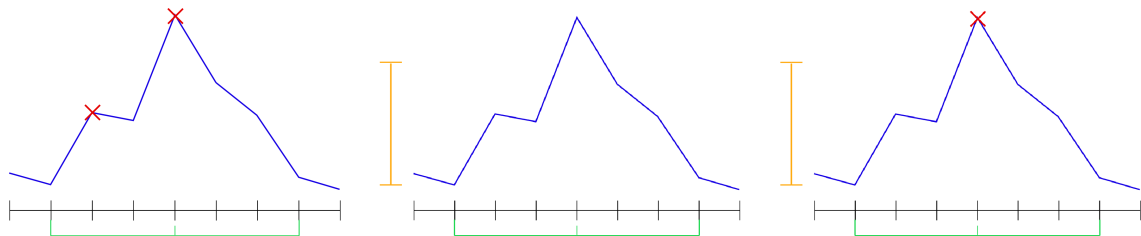


Figure 16: *Red crosses*: final peaks. *Orange bar*: minimum distance with the neighbors that the candidate peaks need to surpass to be considered as final peaks. *Left figure*: absolute peak detector (without any filters). *Middle figure*: peak detector filtering by minimum distance with the immediate left and right neighbors (there are two candidate peaks, but none is final peak because they do not surpass the minimum distance with the immediate neighbors). *Right figure*: the developed custom peak detector filtering by minimum distance with more than the immediate neighbors.

Since there is no algorithm currently developed that works as the right figure of Figure 16, we decided to develop a custom one for our goals. The reason why this is needed for this memory-usage time series scenario is because when we have peaks of memory, the memory does not necessarily increase in one hour, peak, and then decrease in the next hour. This *peaking process* can last more than one hour to increase and more than one hour to decrease. In addition to this, it is also possible that during the increasing phase of this peaking process, at some point in this time interval, the memory remains stable or decreases a bit, like happens in the Figure 16. We think this behavior is valid and we want to allow it, since it does not change the fact that we are reaching the highest, overall peak of the time interval we have set. Therefore, in the Figure 16, we are only interested in getting the second peak, since the first peak is part of the increasing phase of the peaking process of the second peak, which surpasses our stablished memory distance threshold at some point within the time interval/range (in this case, composed by the three neighbors to the left of the peak) that we have stablished as well.

The first implementation of this custom peak detector was purely done in *Python* by the use of dictionaries and lists. Nevertheless, this was quite inefficient when we were trying to get all the peaks (and valleys) of numerous large time series datasets, therefore the whole process was taking a lot of time. Moreover, if we wanted to tweak the input parameter values of the peak detector, we needed to recompute everything again; thus, in total, the amount of time spent was very high.

For this reason, we decided to reimplement the peak detector by using the Python library *NumPy*. NumPy is able to support and process large, multi-dimensional arrays and matrices, providing a large set of high-level mathematical functions to operate on these arrays. On top of that, NumPy is mostly written in *C*, using Python just as a ‘wrapper’. This *C* implementation makes this library highly efficient, since *C* is a low-level programming language. A benchmark was performed to compare the performance between the first and second implementation: the second implementation was more than 50 times faster than the first one.

Going into detail, in the second implementation we decided not to start from scratch and reuse some code of another NumPy-based peak detector (publicly available) as a starting point, and then start to build the custom peak detector from there. I decided to use a peak detector made by a developer named Marcos Duarte (to whom I give credit), which main function is called *detect_peaks* [11], and which implementation is based on the *findpeaks* function of MatLab. Basically, this implementation works the same as the one on MatLab, providing the same functionalities and results. The reason why we chose this peak detector as a starting point was because all the code is contained in just one file source (a Jupyter notebook), which makes it really portable and easy to modify, and because the only dependency was NumPy.

The chosen peak detector will initially detect all the possible peaks of a time series (candidate peaks) and, from there, a developed custom filter will only leave the peaks we are interested in (final peaks).

The input parameters of the algorithm are:

- *Expanded* time series: The algorithm will take and use the `IUM_GALIMIT_RATIO` values from the time series. The expanded time series version is required because we need to set the zero values in case we have missing data. Note: alternatively you can use the normal *non-expanded* time series, but it is not advised because if there is missing data then there is a chance to get false positive peaks when the missing data gaps happen.
- Peak threshold: For a peak to be considered as a candidate final peak, this is the minimum amount of memory percentage that the peak needs to change from itself to at least one of the close neighbors of it that are within the *hours range* time span. This threshold needs to be surpassed with the left *and* right neighbors of the peak.
- Hours range: This value indicates how many hours (i.e., neighbors) before and after the peak are allowed to reach the minimum *peak threshold* percentage. If

the peak threshold is not reached at left *and* right within the specified hours, then the peak is not valid.

- Valley: This is a flag with values *True* or *False*. If *True*, all the calculations of the algorithm will be done for retrieving valleys instead of peaks. Please note that, for clarity, we only talk about calculating peaks here, but actually calculating the valleys is the very same process but with only one difference: we need to invert all the memory values of the time series beforehand (this is actually what this flag does).

This algorithm or custom filter only allows candidate peaks as final peaks if *all* the following conditions are fulfilled:

- The memory distance/difference of the peak in evaluation with at least one of its left neighbors within *hours range* is greater than the *peak threshold*.
- If any of the left neighbors within *hours range* is peak, then those peaks need to be smaller than the one we are evaluating.
- The memory distance/difference of the peak in evaluation with at least one of its right neighbors within *hours range* is greater than the *peak threshold*.
- If any of the right neighbors within *hours range* is peak, then those peaks need to be smaller than the one we are evaluating.

The output after applying the algorithm will be a *list of dictionaries*. The components of these dictionaries are:

- Datetime: This is the date and time of the occurrence of the peak.
- Memory ratio value: This is just the IUM_GALIMIT_RATIO memory ratio value of that peak.
- Distance: The ‘distance’ of a peak is the sum of the memory (IUM_GALIMIT_RATIO) distances from a peak to the minimum value on the left of the peak, *and* to the minimum value on the right of the peak; in both cases just calculating the minimum values within the *hours range* number of neighbors to each side. If we are calculating valleys, it works the same, but we will calculate the distances to the maximum values on left and right instead. This distance measurement is really important because it is used in the *noisy segment detector* that is *based on peak closeness and memory distance*, which will be described later. An example of a time series with a peak and its minimum values, for the distance calculation, is shown in Figure 17.

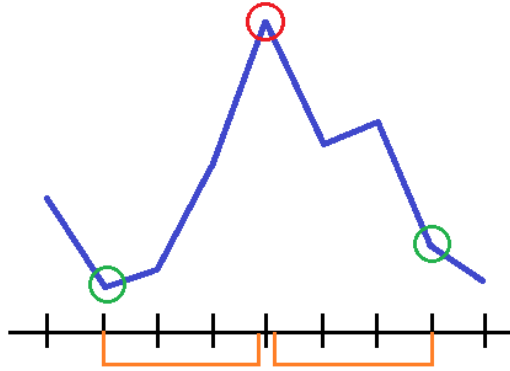


Figure 17: Orange time frame lines: ‘hours_range’ input parameter. Red circle: calculated valid peak. Green circles: minimum values on left and right within ‘hours_range’. If we want to calculate the ‘distance’ value, it is the sum of the memory distances from the peak to both minima.

5.2 Noisy Segment Detector

The noisy segment detector is an algorithm that detects time periods of a time series in which there are high concentration of peaks, which are individual time units. These noisy segments are established by how *close-in-time* the peaks are and, as an additional option, by how much the memory fluctuates from a peak to its closest neighbor data points, i.e., by how *important* each peak is. This peak ‘importance’ measurement is called *distance*, as seen in the output parameters of the peaks after applying the *Peak Detector* algorithm.

Thus, there are *two* noisy segment detectors: the first one, called *noisy segment detector by peak closeness*, which is based on how close-in-time peaks are; and the second one, called *noisy segment detector by peak closeness and memory distance*, which is based on how close-in-time peaks are and their memory distance values. The former is based on a *countdown* approach, and the latter is based on a *sliding window* approach.

For convenience, during this section we will be just referring to apply these algorithms to peaks, although the algorithms can also be applied to peaks and valleys altogether — actually, they can be applied to any data point in a time series.

5.2.1 Noisy Segment Detector by Peak Closeness

This algorithm will only take into consideration how close-in-time the peaks are. It follows a countdown-type algorithm approach.

Some necessary input parameters are required:

- *Expanded* time series: The algorithm will take and use the IUM_GALIMIT_RATIO values from the time series. The expanded time series version is required because we need to set the zero values in case we have missing data.
- Peaks (and valleys): These were previously detected after using the *Peak Detector* algorithm.

- Minimum number of peaks per segment: A noisy segment must be composed by at least two peaks. If we set this parameter, the minimum number of peaks per segment will be this value.
- Maximum distance between peaks: This means that if we want to create a segment, or we already have a segment ‘on-going’ and we want to extend it, the maximum possible distance in time between the last found peak and the next one must be equal or less than the value established by this parameter. In other words, there will *not* be a noisy segment after the last found peak if there is no other peak after the time established by the parameter.

The pseudocode of the algorithm is shown in Algorithm 1 (please note that it slightly differs from the actual implementation). The pseudocode is quite self-explanatory, although it is important to keep in mind that if the backwards counter reaches zero and we have not found a new peak, the current segment (if there was any) will end in the last found peak. To clarify, in the pseudocode *segstart* means the start of a segment, *segend* means the end of a segment, *peaksinseg* means the number of peaks found in the current segment, and *backcount* is the backwards counter.

Algorithm 1 Noisy Segment Detector by Peak Closeness

```

1: procedure NOISYSEGMENTDETECTORBYPEAKCLOSENESS
2:   minpeaks  $\leftarrow$  minimum number of peaks per segment.
3:   maxdist  $\leftarrow$  maximum distance between peaks.
4:   loop (iteration over all rows of the time series):
5:     if row is peak then
6:       peaksinseg  $\leftarrow$  peaksinseg + 1.
7:       backcount  $\leftarrow$  maxdist.
8:       if not segstart then
9:         segstart  $\leftarrow$  peak.
10:      if peaksinseg  $\geq$  minpeaks then
11:        segend  $\leftarrow$  peak.
12:      else
13:        backcount  $\leftarrow$  backcount - 1.
14:        if backcount = 0 and segend then
15:          append segment(segstart,segend,peaksinseg) in seglst.
16:          clear segstart.
17:          clear segend.
18:          peaksinseg  $\leftarrow$  0.
19: return seglst.

```

The output after applying the Algorithm 1 is a *list of segments*, which have a *dictionary* data structure. The components of these dictionaries are:

- Start: datetime when the segment starts.
- End: datetime when the segment ends.

- Number peaks: number of peaks that are contained in the segment.
- Length: length, or total number of rows, of the segment.

5.2.2 Noisy Segment Detector by Peak Closeness and Memory Distance

This algorithm expands the *Noisy Segment Detector by Peak Closeness* algorithm to detect noisy segments in a time series, considering not only the position of the detected peaks in the time series, but also their relevance, which is provided by the ‘distance’ value of the peaks (see output values after applying the *Peak Detector* algorithm). This algorithm follows a *sliding window* approach.

With this algorithm, it could be possible, for example, that even though we have multiple and extremely close peaks in a 24 hours range, we do not detect this time frame as a noisy segment since the relevance of these peaks is not high enough.

Some necessary input parameters are required (this time the original time series is not required as an input parameter because the development of this algorithm has been done differently):

- Peaks (and valleys): These were previously detected after using the *Peak Detector* algorithm.
- Minimum number of peaks per segment: A noisy segment must be composed by at least two peaks. If we set this parameter, the minimum number of peaks per segment will be this value.
- Window size: This is the size of the sliding window, which will iterate over the whole time series vector looking for peaks and their *distance* values.
- Window threshold: Value that the sum of all the *distance* values of the peaks evaluated in a sliding window need to surpass to consider that sliding window iteration as a noisy segment.

It is important to consider that the window size and window threshold need to be manually adjusted until get a good match that satisfies the user requirements. These values depend in a great manner on the *peak threshold* input parameter of the *Peak Detector* algorithm as well. Thus, the process to adjust the window size and window threshold is quite tedious, because after applying a new set up, you have to manually check if the noisy segment results follow the user requirements accordingly. An automatic adjustment of these values would be most probably possible through another algorithm, but for that we would need to have, in advance, labels of what the user considers noisy segments.

The pseudocode of the algorithm is shown in Algorithm 2 (please note that it slightly differs from the actual implementation). In this algorithm, ‘noisy subset’ is equivalent to ‘noisy segment’. A short explanation of the procedures of the pseudocode follows:

1. Vector creation: In this procedure we create a vector array with the ‘distance’ values of the peaks. The length (or number of elements) of this array is the number of hours in between the first detected peak of the time series and the last one. The peak datetimes matches in position with the elements of the array: If an element of the array is a peak, this element will acquire the ‘distance’ value of the peak, and acquire 0 otherwise.
2. Sliding window: A sliding window of size *window size* will be created. This window will iterate over the *array of distances*. Over each iteration, it will sum all the elements (distances) of the array inside that window, and evaluate if the result is greater or equal than the *window threshold* input parameter. If it is greater or equal, a subset is created with the content of the current window and stored in *noisy_array_subsets*.
3. Segment creation: In this procedure, if the index of an element of a subset between its starting point and ending points (inclusive) overlaps with any of the element indexes of another subset, these two subsets will be combined together into one single and bigger subset, with now wider starting and ending points. This merging process can be applied to two or more overlapping subsets. After applying this procedure, any subset stored in *noisy_array_subsets* does not overlap with each other anymore.
4. Segment shortening: In case of having noisy subsets with starting and ending points that are not peaks, we shorten them to start and end in peaks.
5. Segment outputting: We output, in a list, all the starting points, ending points, number of peaks, lengths, and scores of all the noisy subsets/segments found. These ‘scores’ are the sum of all the ‘distance’ values of the peaks inside of each segment.

The output after applying the Algorithm 2 is a *list of segments*, which have a *dictionary* data structure. They components of these dictionaries are:

- Start: datetime when the segment starts.
- End: datetime when the segment ends.
- Number peaks: number of peaks that are contained in the segment.
- Length: length, or total number of rows, of the segment.
- Individual score: sum of all the ‘distance’ values of the peaks inside the segment.

5.3 Noisy Segment Occupancy Scorer

The scorer of the already calculated noisy segments returns a score that states ‘how noisy’ a time series is. It will be used to rank the systems, from noisiest to the least noisy. The higher the output number is, the noisier. A zero value output means

Algorithm 2 Noisy Segment Detector by Peak Closeness and Memory Distance

```

1: procedure VECTORCREATION
2:    $array\_size \leftarrow toHours(Last\ peak\ datetime - First\ peak\ datetime)$ .
3:    $distances\_array \leftarrow \{x_0, x_1, x_2, \dots, x_{array\_size}\}$  where  $x_i$  is:
4:      $peak\_distance$  if  $peak$ 
5:     0 otherwise
6: procedure SLIDINGWINDOW
7:    $number\_windows \leftarrow array\_size - window\_size + 1$ .
8:   for  $i=0$  to  $i=number\_windows-1$  do
9:     if  $(\sum_{n=i}^{i+window\_size} distances\_array[n]) \geq window\_threshold$  then
10:      append  $distances\_array[i : i+window\_size]$  in  $noisy\_array\_subsets$ .
11: procedure SEGMENTCREATION
12:   if 2 or more  $subsets$  partially overlap in time in  $noisy\_array\_subsets$  then
13:     combine into one  $subset$  in  $noisy\_array\_subsets$ .
14: procedure SEGMENTSHORTENING
15:   if  $subset$  in  $noisy\_array\_subsets$  does not start and end at peak then
16:     shorten  $subset$  to start and end at peak.
17: procedure SEGMENTOUTPUTING
18:   for  $subset$  in  $noisy\_array\_subsets$  do
19:      $score \leftarrow \sum_{n=subset\_start}^{subset\_end} distances\_array[n]$ .
20:     append  $subset\_start, subset\_end, number\_peaks, length, score$  in  $output\_list$ .
21:   return  $output\_list$ .

```

that there is no noise, which happens in time series where no noisy segments were detected. The scorer then should be only applied to time series that contain noisy segments.

It calculates the total length of all the noisy segments of a time series, then it calculates the total length of the time series *excluding* the missing samples (if any), and finally it divides the first total by the second one. This will be the ratio of occupation of the noisy segments in the time series.

5.4 Daily Seasonality Detector

This algorithm detects whether a system has daily seasonality or not. To achieve this, we calculate the pearson correlation of the time series with the *same* time series, but lagged; this is called *autocorrelation*. In this case, the lag of the time series will always be 24, because we are dealing with hourly data and we are looking for daily seasonal systems. If the pearson correlation is higher than a threshold that have we set (and we have accomplished another condition, which will be specified later in this algorithm description), then this system will be detected as daily seasonal.

The autocorrelation of a time series is given by

$$\hat{\gamma}(h) = \frac{1}{n} \sum_{t=1}^{n-|h|} (x_{t+h} - \bar{x})(x_t - \bar{x})$$

where h is the lag, and t is a time series *time point* (the x associated to t is the value of the time series *data point* itself).

The input parameters of this algorithm are:

- ZEW_IUM values (*not* the IUM_GALIMIT_RATIO values) of the *expanded* time series: We need the ZEW_IUM values because the IUM_GALIMIT_RATIO memory ratio would partially spoil the daily seasonality detection in case of ZEW_GALIMIT change, i.e., we need absolute data input. We also need the expanded time series because *zero* values (due to missing data points) can occur repeatedly over time, e.g., when a system turns off every day from 00:00 to 06:00.
- Correlation threshold: This is a condition to consider a time series as daily seasonal. The Pearson correlation value of the time series with itself lagged 24 hours should be higher than this parameter.
- Minimum distance with minimum correlation: This is another condition to consider a time series as daily seasonal. This is the minimum value difference possible between the Pearson correlation at lag 24 hours with the minimum Pearson correlation achieved during any of the autocorrelations of the time series at lags 1–23.

The output of this algorithm is simply a boolean: *True* if daily seasonal, *False* if not.

The algorithm is not complex: The only conditions that are needed to be met in order to detect if a system is daily seasonal are to surpass the thresholds indicated by the inputs *correlation threshold* and *minimum distance with minimum correlation*. Figure 18 will show the autocorrelation plot of a positive daily seasonal system, and Figure 19, Figure 20, Figure 21, and Figure 22 will show the ones of negative cases. All the systems in the figures are real examples that come from the dataset of this project. It is important to read the captions of such figures, because there it is explained why a system is (or is not) daily seasonal. In any case, to make it clear visually, the value of the correlation at lag 24 needs to be over the red line and over the black dashed line. Note that at lag 0, we will always have the maximum correlation possible because we are comparing the time series with itself.

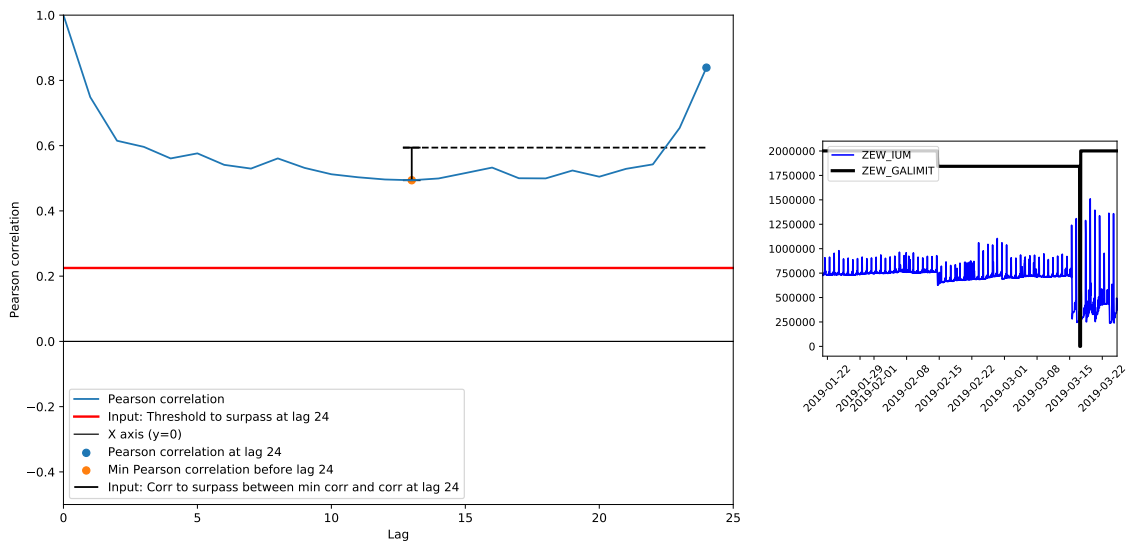


Figure 18: Left: Autocorrelation plot of a daily seasonal system, in which the Pearson correlation at lag 24 is over the target threshold *and* in which there is enough distance between the minimum correlation of lags before 24 and the correlation at lag 24. Right: Associated time series.

The intuition behind the algorithm to know if a system has daily seasonality is that, after lag 0 (where the correlation is always 1), the correlation drops as much as it can to then, at lag 24, increase again to peak there and surpass the two established thresholds. If this happens, that time series will be daily seasonal.

The reason why we want to detect whether a system has daily seasonality is because, if the system is noisy and it has daily seasonality, then we can most probably consider it, actually, as *not noisy*. The logic here is that if we have daily seasonality, then the data points of a system are equidistantly distributed every 24 hours, including the data points of the detected peaks, which means that this system does not really have any problem: The system always behaves in the same way, so these relevant peaks should have been already addressed by the customer/admin of the concrete HANA system.

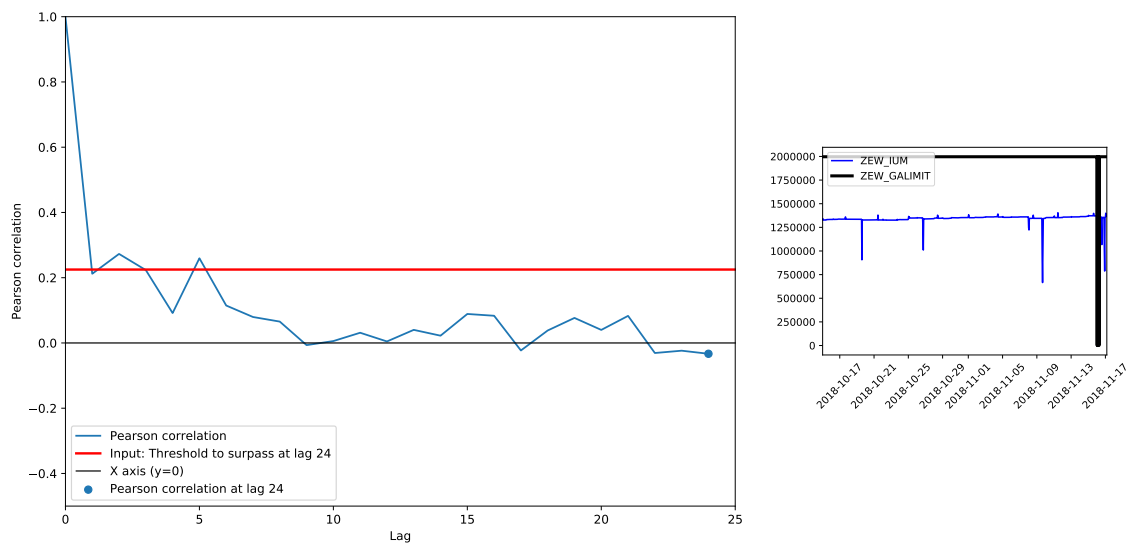


Figure 19: Left: Autocorrelation plot of a non-daily seasonal system, in which the Pearson correlation at lag 24 is under the target threshold. Right: Associated time series.

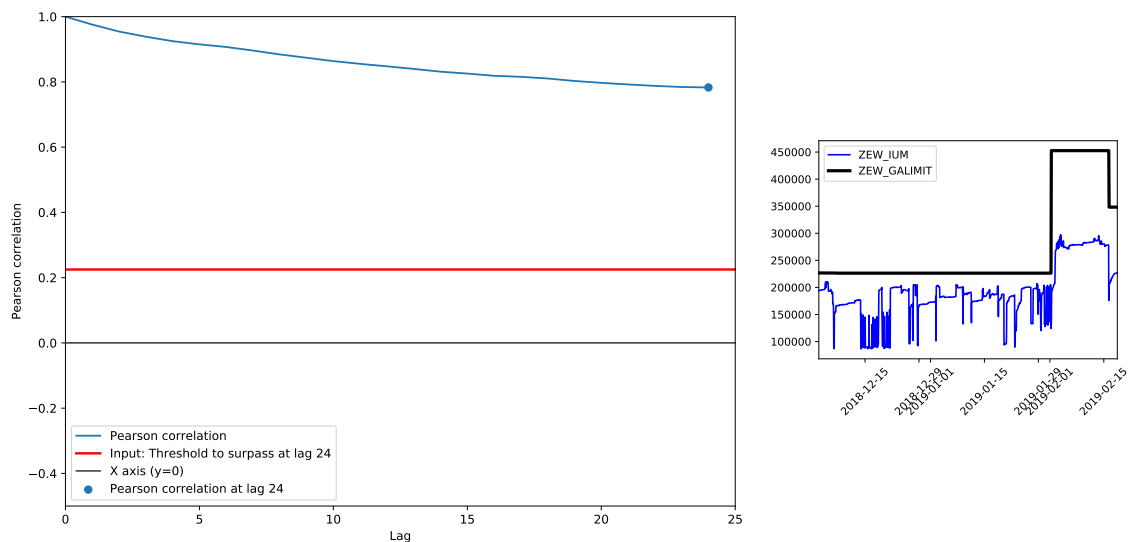


Figure 20: Left: Autocorrelation plot of a non-daily seasonal system, in which the Pearson correlation at lag 24 is over the target threshold *but* there is no minimum correlation before lag 24. Right: Associated time series.

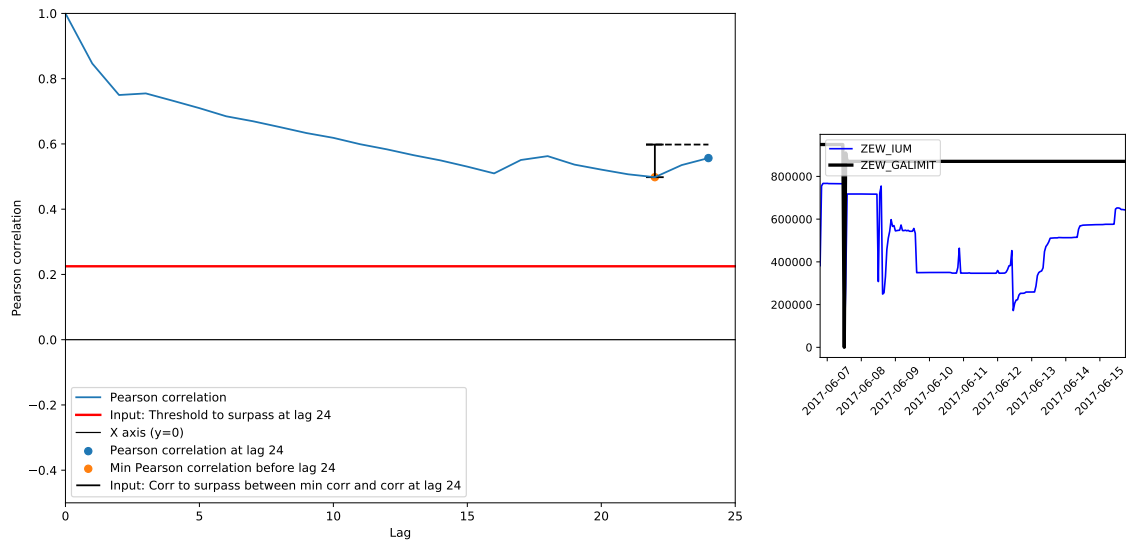


Figure 21: Left: Autocorrelation plot of a non-daily seasonal system, in which the Pearson correlation at lag 24 is over the target threshold but there is not enough distance between the minimum correlation of lags before 24 and the correlation at lag 24. Right: Associated time series.

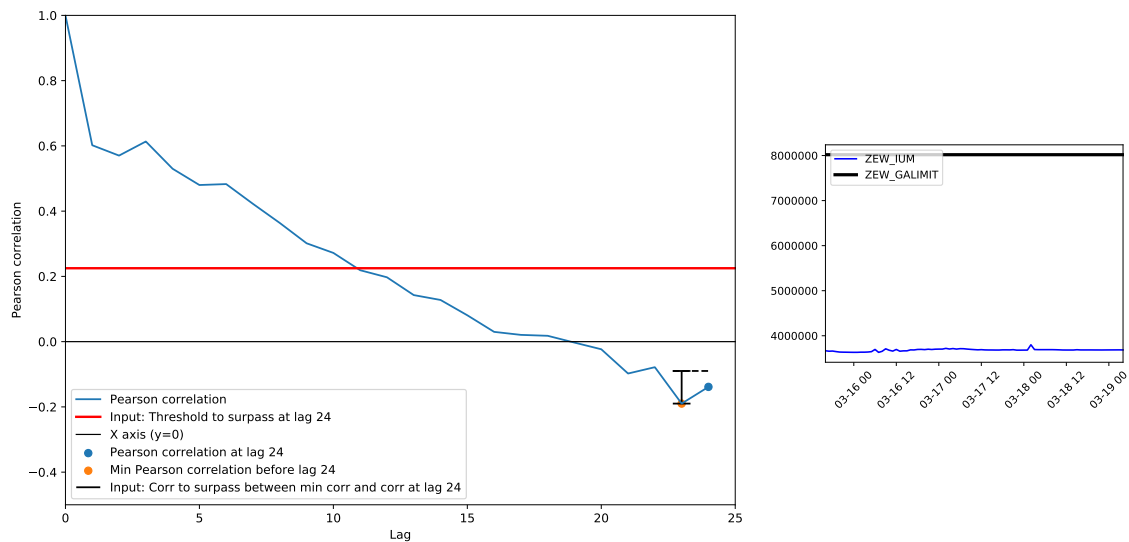


Figure 22: Left: Autocorrelation plot of a non-daily seasonal system, in which the Pearson correlation at lag 24 is under the target threshold *and* in which there is not enough distance between the minimum correlation of lags before 24 and the correlation at lag 24. Right: Associated time series.

6 Results

6.1 Daily Seasonal Systems

The number of daily seasonal systems detected in our subset of the PHD-MEM1 table through the *Daily Seasonality Detector*, with default input parameters ($\text{corr_threshold}=0.225$ and $\text{min_distance_with_min_corr}=0.1$), is *41 out of 1529 systems*.

6.2 Peak and Valley Detection

Some of examples of the outcomes after applying the *Peak Detector* algorithm in different time series are shown in Figure 23, Figure 24, and Figure 25. Figure 24 additionally shows the peaks of a short time series ranked by their relevance (Subfigure 24b), and the three most relevant peaks (Subfigure 24c). In all these cases, the input parameters for the algorithm are 5% for the *peak threshold*, and 3 hours for the *hour range*.

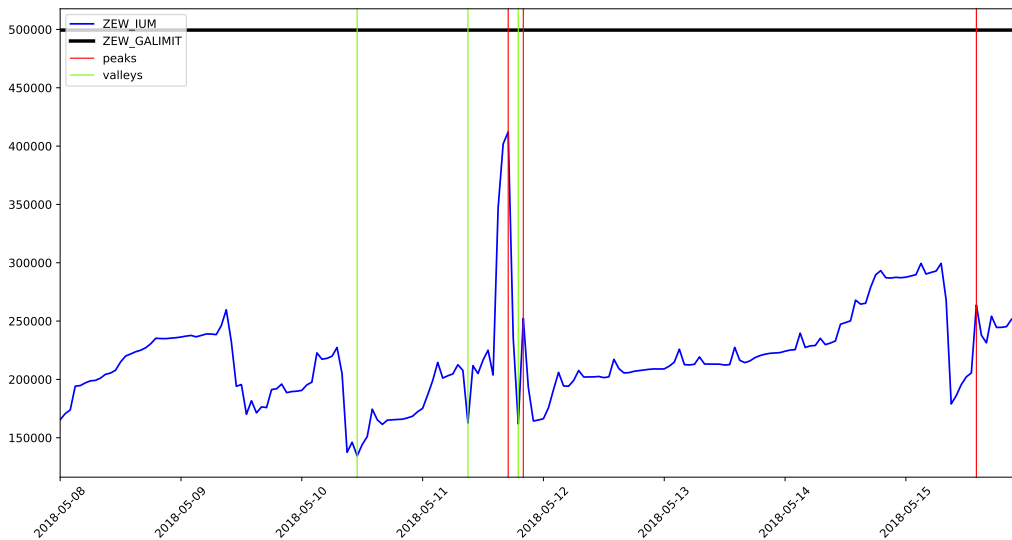


Figure 23: Peak and valley detection in a system.

6.3 Comparison of Noisy Segment Detection Algorithms

The purpose of this section is to see the outcomes of both noisy segmentation algorithms, *Noisy Segment Detector by Peak Closeness* and *Noisy Segment Detector by Peak Closeness and Memory Distance*, when applied to a time series and its detected peaks and valleys, while at the same time perform a comparison of both techniques. In the end, the choice of one algorithm over the other will be due to the preferences and/or requirements of the user.

For the outcomes in this section, the following input parameters for the algorithms have been used:

- Peak Detector: *peak threshold 5%, hour range 3 hours.*
- Noisy Segment Detector by Peak Closeness: *minimum number of peaks per segment 3, maximum distance between peaks 8.*
- Noisy Segment Detector by Peak Closeness and Memory Distance: *minimum number of peaks per segment 3, window threshold 20, window size 8.*

There will be two comparisons between both algorithms, which follow:

Comparison 1

In this comparison, which is shown in the Figure 26, we see in the center of the time series two short noisy segments when we are doing *closeness segmentation* (Subfigure 26a), while when we are doing *closeness and memory distance segmentation* (Subfigure 26b) there is only one bigger segment.

In the first case, there is an empty space between these two segments because in 8 hours or more there are no more peaks or valleys.

In the second case, we still have the 8 hours (or more) gap without peaks/valleys, but this time we are doing a sliding window process in which the last peak of the first small segment and the first peak of the second segment are quite significant (they have high ‘distance’ values). Due to this, the gap in between these peaks is ‘covered’ by noise, and thus, in the end we have just one big noisy segment. If these last and first peaks were less significant (if they had significantly smaller ‘distance’ values), we would have had again two small segments split by an empty space, like in the first case.

Comparison 2

In this second comparison, which is shown in Figure 28, it happens the contrary to the first comparison: we have in the center of the time series a really big noisy segment when we are doing the *closeness segmentation* (Subfigure 27a), while we have three smaller segments when we are doing the *closeness and memory distance segmentation* (Subfigure 27b).

In the first case, the segmentation is satisfactory since it is doing what it was supposed to do. We just have one big segment because there are no time gaps equal or longer than 8 hours without peaks or valleys, in other words, we are able to find a new peak or valley within 8 hours since the last peak or valley.

In the second case, the last and first peaks of each one of the three smaller segments have peaks that are not very significant (their ‘distance’ values are small); therefore, the sliding window algorithm does not consider these empty spaces as noise. In each one of these three segments, peaks and valleys are bigger in each central area, but smaller on the sides.

6.4 Most Noisy Systems with High Sensitivity Peak Detection

“High sesibility” in the peak and vally detection means that the *peak threshold* of the *Peak Detector* algorithm has a low value, in this case, 5%. This implies that many peaks and valleys will be detected and, therefore, we can use a noisy segment detector since the peaks and valleys are more probably to be close in time. Valleys are also considered as noise because, as the peaks, they originate in the time series a fluctuation in the memory values, but in this case ‘negative’. Because the noisy segments are built on the detection of peaks and valleys, these segments highly depend on the input parameters of the *Peak Detector*. The input parameters for all the used algorithms follow:

- Peak Detector: *peak threshold* 5%, *hour range* 3 hours.
- Noisy Segment Detector by Peak Closeness: *minimum number of peaks per segment* 3, *maximium distance between peaks* 8.
- Noisy Segment Detector by Peak Closeness and Memory Distance: *minimum number of peaks per segment* 3, *window threshold* 20, *window size* 8.

Two most noisy systems through both approaches

Once the noisy segments have been calculated, we can score the time series with the *Noisy Segment Occupancy Scorer*. Thanks to this score, we will be able to rank the systems from noisiest to least noisy; therefore, we can know which are the noisiest systems.

The two most noisy systems from both segmentation approaches are:

- Segmentation by Noisy Segment Detector by Peak Closeness:
 - Noisiest system: Subfigure 28a. The system has daily seasonality, as it is shown in Figure A1.
 - Second noisiest system: Figure 29. The system does not have daily seasonality, as it is shown in Figure A2.
- Segmentation by Noisy Segment Detector by Peak Closeness and Memory Distance:
 - Noisiest system: Subfigure 28b. The system has daily seasonality, as it is shown in Figure A1.
 - Second noisiest system: Figure 30. The system does not have daily seasonality, as it is shown in Figure A3.

As it can be seen, the most noisy system from both noisy segmentation approaches is the same one. Nevertheless, in the second position of both approaches we have different systems.

6.5 Most Noisy Systems with Low Sensitivity Peak Detection

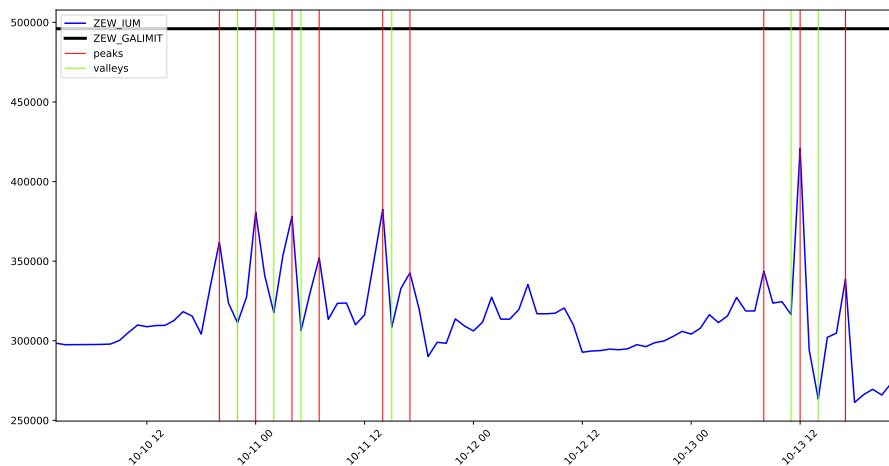
“Low sensibility” in the peak detection means that the *peak threshold* of the *Peak Detector* algorithm has a high value, in this case, 30%. This implies that few peaks will be detected and, therefore, we *cannot* use a noisy segment detector, since the peaks will be probably widely spread over time. Valleys are not considered this time for this same reason. We are not looking for time periods of noise, but for individual relevant peaks, with precise datetime. The input parameters for the *Peak Detector* are: *peak threshold* 30%, *hour range* 3 hours.

Three most noisy systems through both approaches

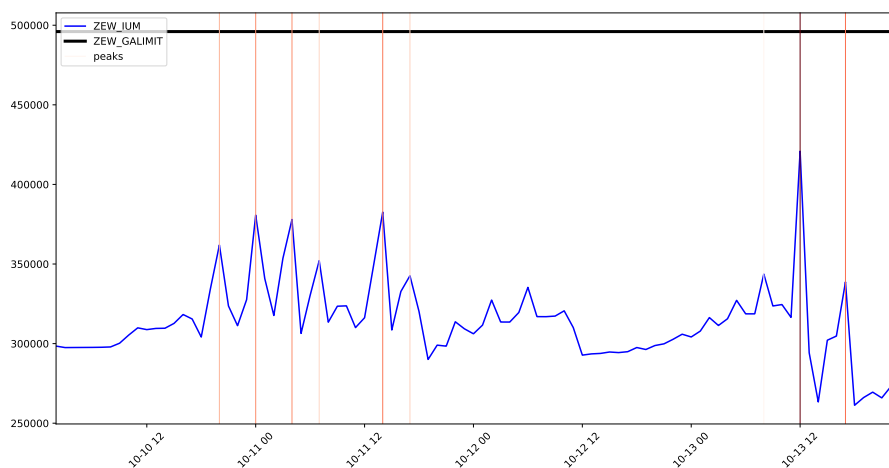
Once the peaks of the systems have been detected, we can rank the systems from the one with the highest number of peaks ratio, to the one with the lowest number of peaks ratio. This ratio is the number of peaks in the time series divided by the length of the time series. The three systems with highest peak ratio are:

- Noisiest system: Figure 31. The system does not have daily seasonality, as it is shown in Figure A4.
- Second noisiest system: Figure 32. The system does not have daily seasonality, as it is shown in Figure A5.
- Third noisiest system: Figure 33. The system has daily seasonality, as it is shown in Figure A6.

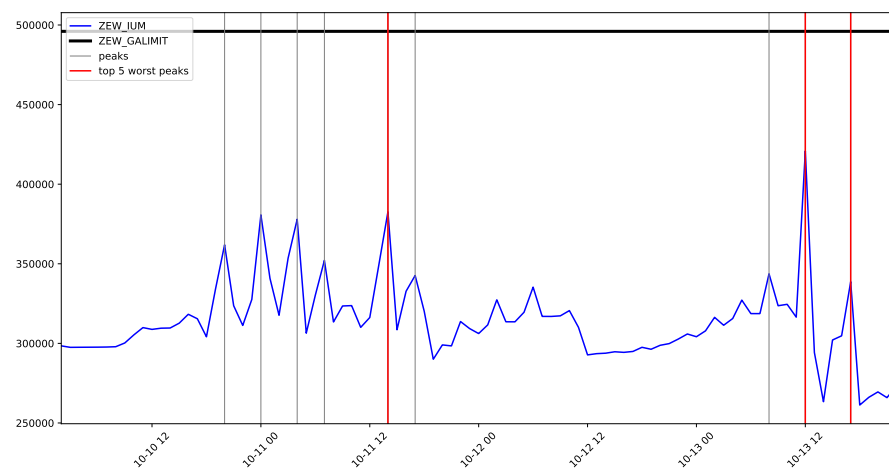
In the first and second positions, we have two time series with just one detected peak, but that is because the time series are not specially long (9–10 days data), therefore their peak ratio score is high. In the third position, we have a time series with several peaks detected, but the peak ratio score is less than the scores of the two previous cases, because the length of the third time series is greater (2 months).



(a) Peak and valley detection.



(b) Ranking of the peaks by their relevance (by their 'distance' value). The more intense the red color, the more relevant the peak.



(c) Worst 3 peaks, i.e., top 3 peaks by their relevance (by their 'distance' value).

Figure 24: Peak and valley detection in another system. This figure is composed by three subfigures.

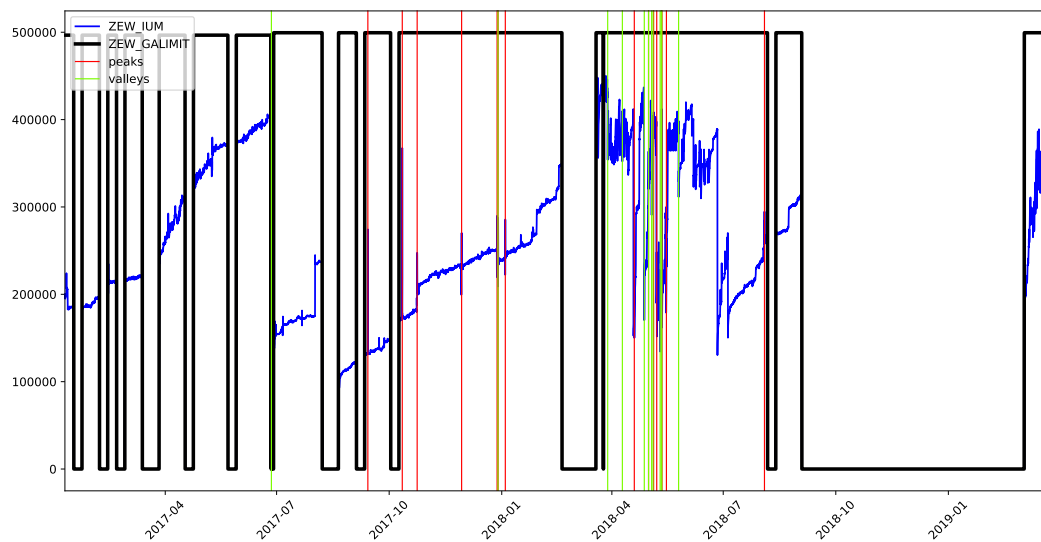
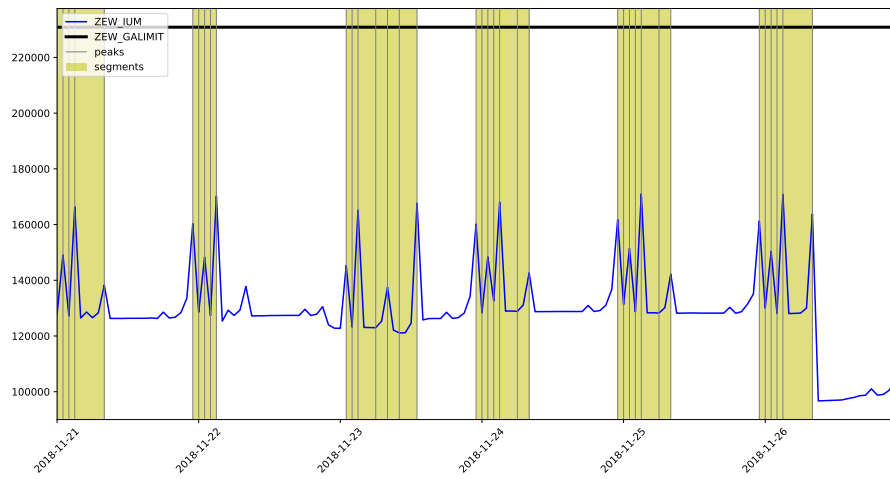
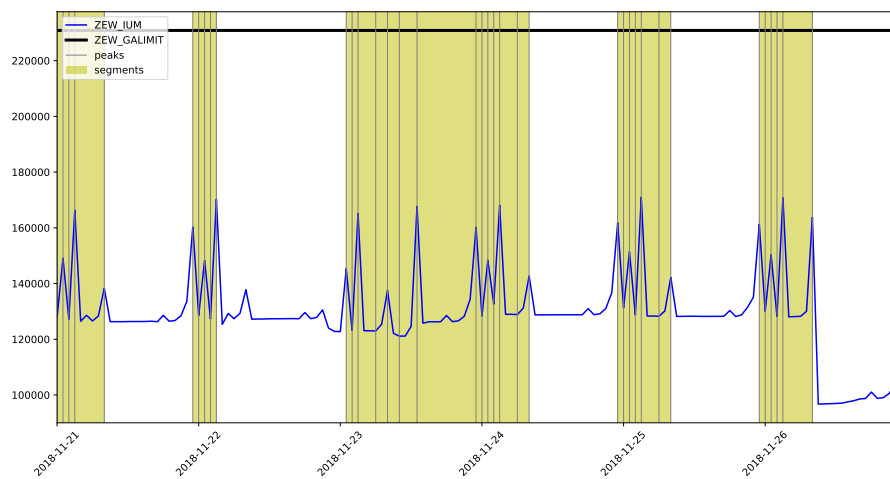


Figure 25: Peak and valley detection in an usual, long time series data.

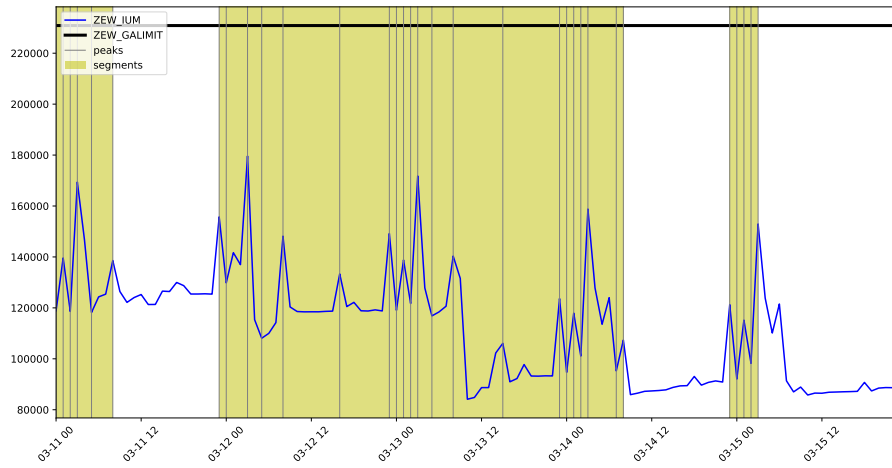


(a) Noisy segment detection by peaks/valleys closeness only.

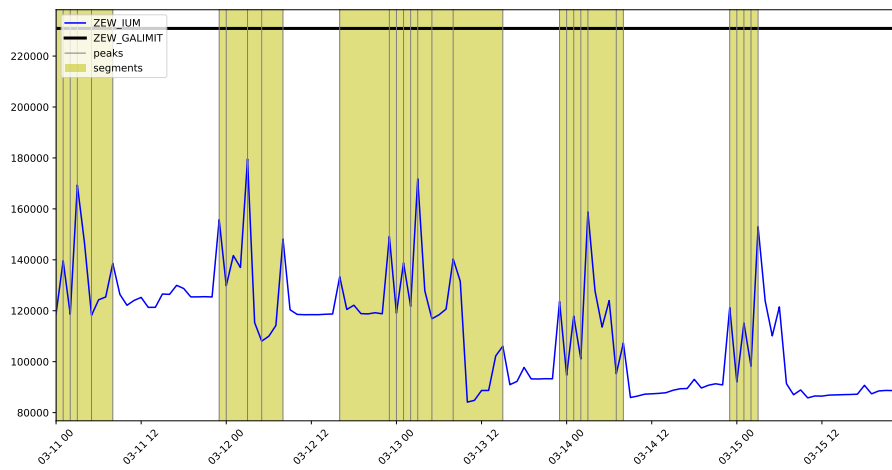


(b) Noisy segment detection by peaks/valleys closeness and memory distance.

Figure 26: Comparison number 1.

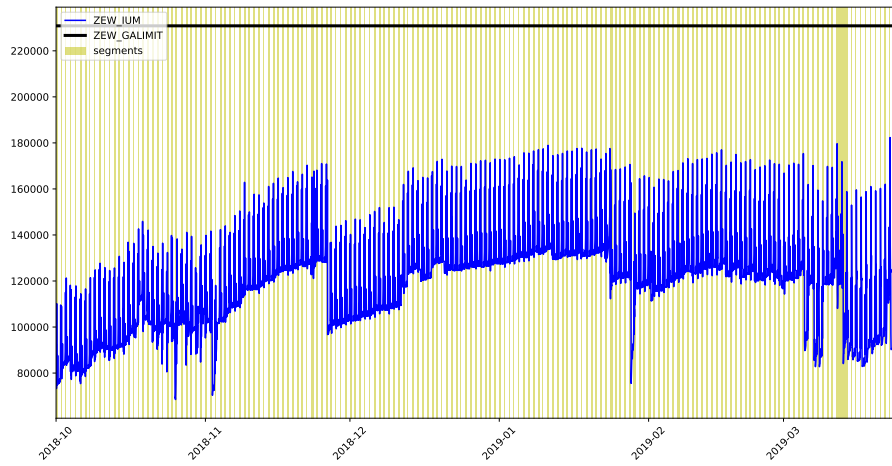


(a) Noisy segment detection by peaks/valleys closeness only.

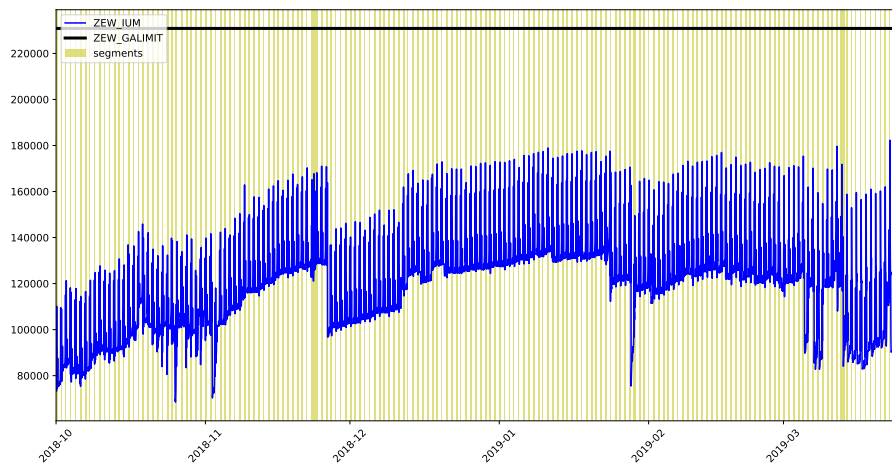


(b) Noisy segment detection by peaks/valleys closeness and memory distance.

Figure 27: Comparison number 2.



(a) Noisy segmentation performed by peak/valley closeness only.



(b) Noisy segmentation performed by peak/valley closeness and memory distance.

Figure 28: System with the most occupancy score, i.e., the noisiest system.

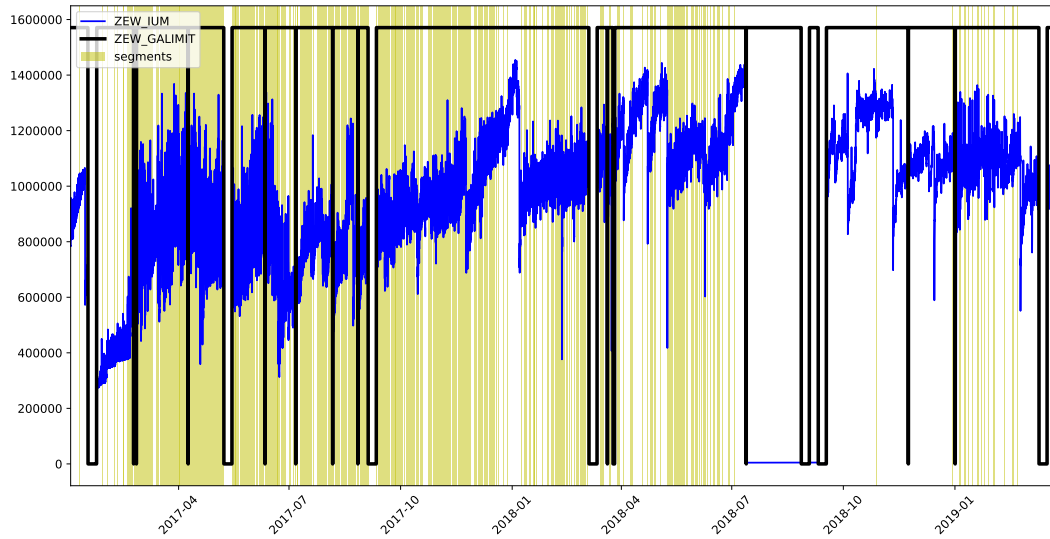


Figure 29: Second noisiest system by noisy segments by peak/valley closeness only.

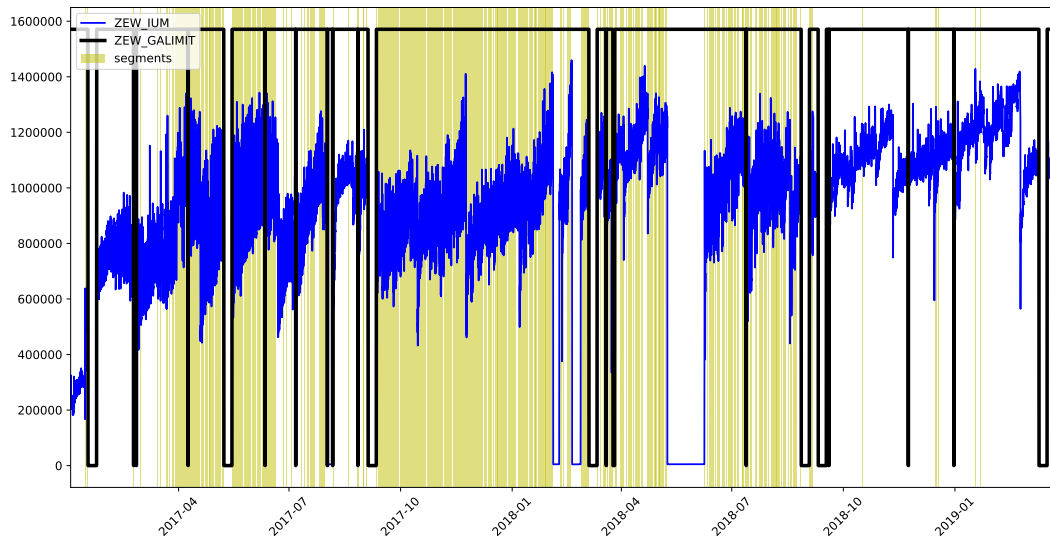


Figure 30: Second noisiest system by noisy segments by peak/valley closeness and memory distance.

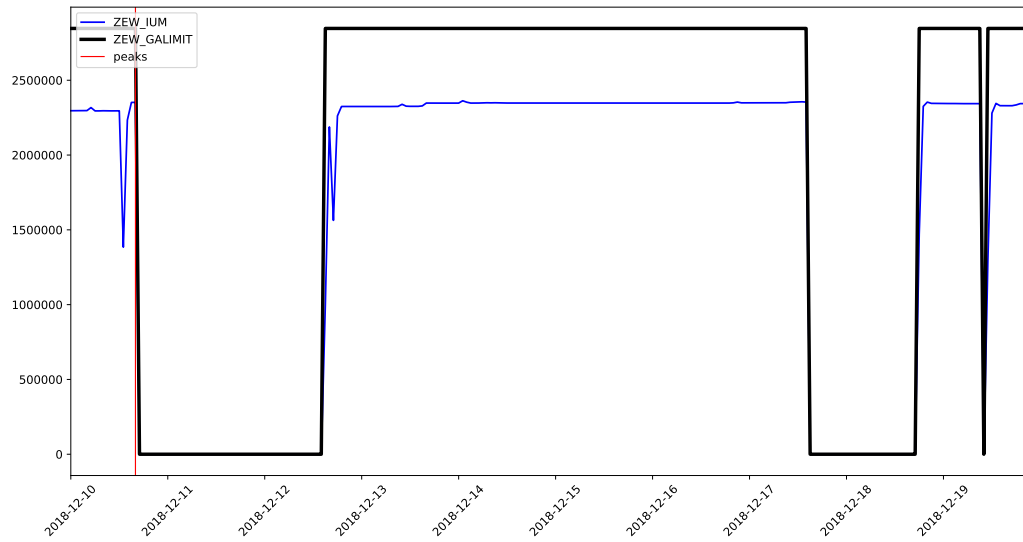


Figure 31: Noisiest system with 30% peak threshold detection.

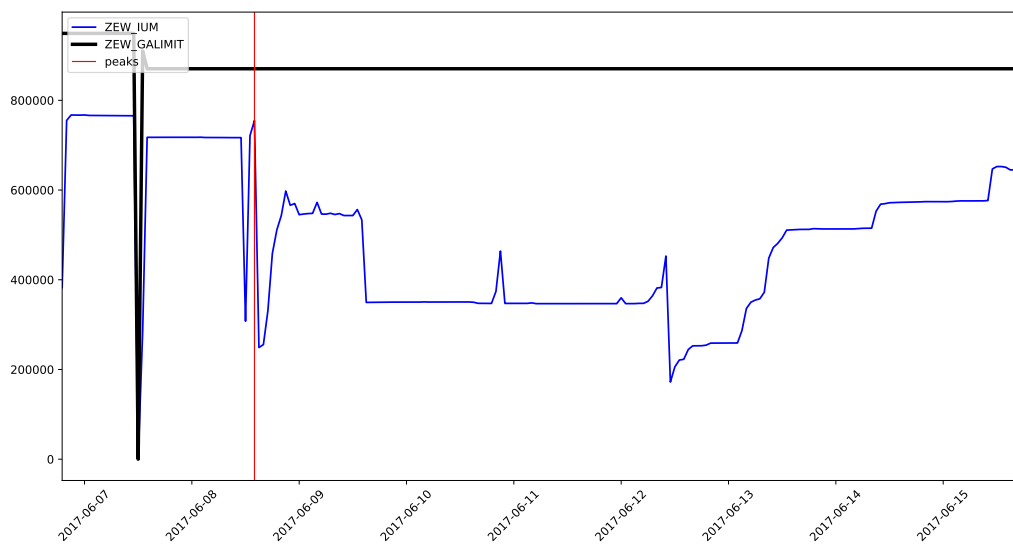


Figure 32: Second noisiest system with 30% peak threshold detection.

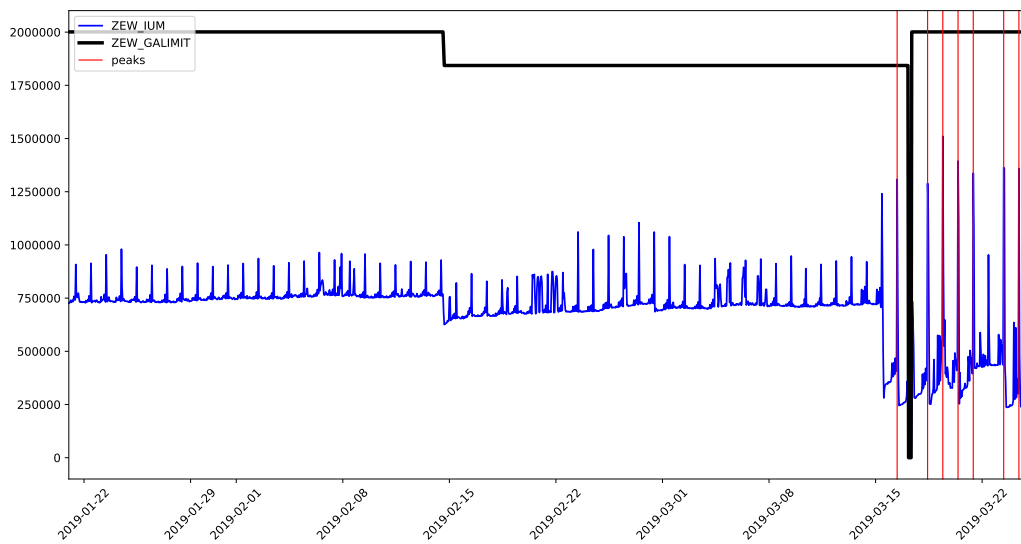


Figure 33: Third noisiest system with 30% peak threshold detection.

7 Deployment on SAP Data Hub

SAP needs to run all the developed algorithms (and the ones to be done) without performance issues, and that is where SAP Data Hub takes action: it runs on a cluster of 4 nodes (or more, if required), with incredible scaling properties. This will ensure a positive and consistent experience for the customer.

Going into detail about SAP Data Hub, it is a tool that lets the user integrate data, orchestrate data processing, and manage metadata across enterprise data sources and data lakes. It also allows the user to build powerful pipelines, as well as manage, share, and distribute data. The Figure 34 shows the whole picture of the tool but, in essence, what this tool does is to get, process, and output data in a *pipeline sequence*.

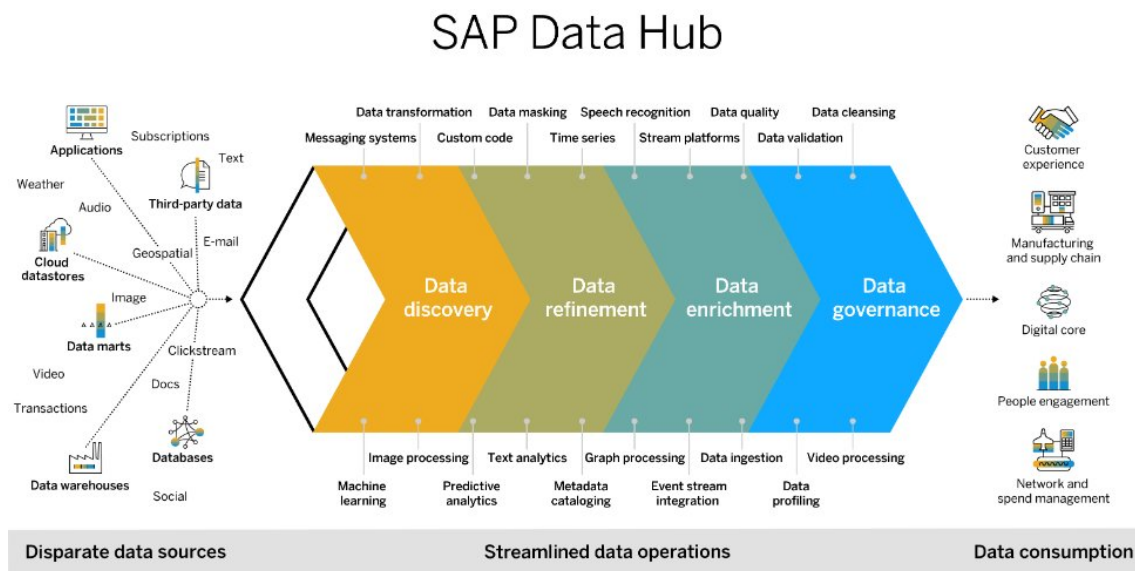


Figure 34: SAP Data Hub.

Some of the features of Data Hub are:

- Deployable anywhere: on premise, private, and managed cloud services.
- Automated, reusable data pipelines for scale and efficiency.
- Unified metadata catalog for visibility across all data assets.
- End to end data landscape management for effective use of all data.
- Central graphical user interface to monitor and distribute data.
- Batch and stream processing.

As said, this tool operates in a pipeline sequence. This pipeline is composed by operators, that takes data, process it, and passes it to the next operator. Each operator is *logically* a single Docker container, which is a self-contained runtime

environment: this provides isolation and fault tolerancy, in the sense that if one operator fails, it does not necessarily mean that it will affect to the whole pipeline. These operators do not interact with each other, unless you connect them, e.g., the output of one operator can be the input of the next one.

There are many predefined operators provided by default, but you can create your own operator in different programming languages such as Python and JavaScript, thanks to an included API. Thus, for example, it is possible to create an operator that takes data, runs your Python script, and outputs the data. To achieve this, you need the Python code and add in it two calls to this API: one for retrieving the data, and the other one for outputting it.

A Proof of Concept

As said in the *Objectives* chapter, I was asked to do a proof of concept of SAP Data Hub to test its feasibility; if feasible, then this tool will be used in production.

I created the pipeline which is shown in Figure 35. Basically, what it does is to take the first specified number of rows of the MEM1 table from the PHD schema from HANA, and *locally* run the *Daily Seasonality Detector* algorithm on this data. Usually (that is, if the specified number of rows is high enough) it will include data from more than one system; therefore, we will get results of daily seasonality for multiple systems. When these results are calculated, they will be written back into HANA, although in this case it will be in a different schema and table.

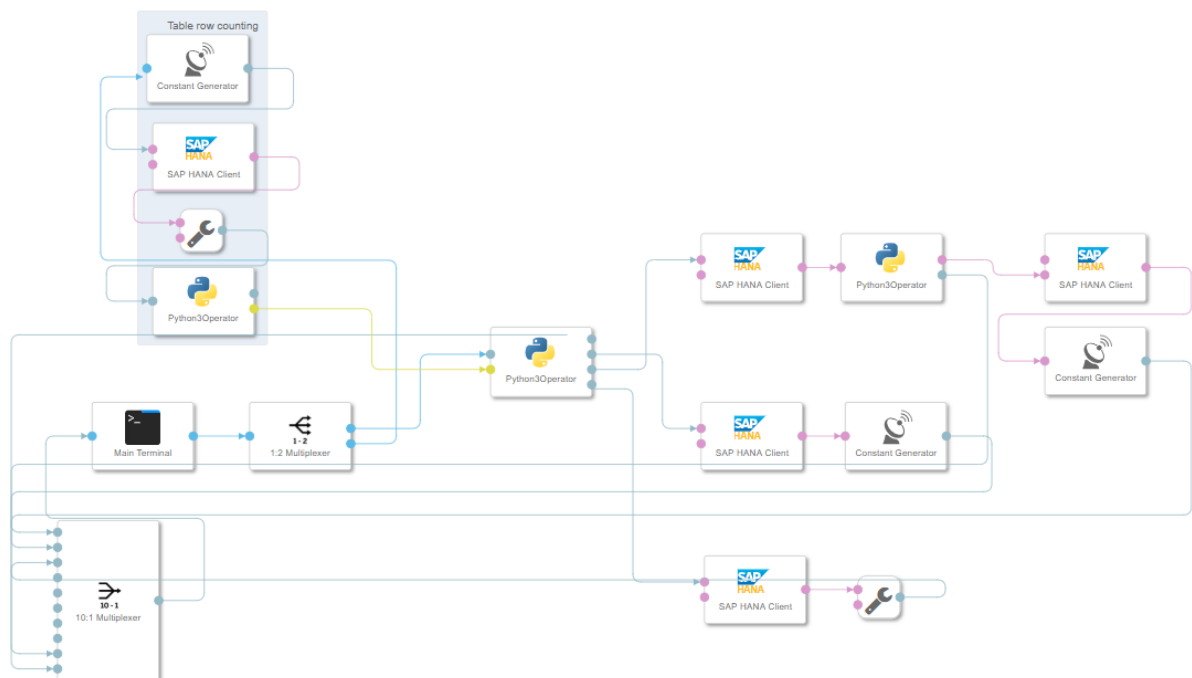


Figure 35: A Data Hub pipeline that takes time series data from several systems to check whether these systems have daily seasonality or not.

The terminal operator is used to interact with the pipeline, having the possibility

to input parameters and display content in an interactive way. In Figure 36, it is shown how to operate in the terminal to input parameters and to see the results, which are also stored (or applied, depending of the case) in HANA. In this figure of the terminal, we have two parts: the upper one, which is the display of the outputs, and the lower part, which is where we input commands and parameters. In the terminal, we can see the different performed operations to achieve the daily seasonality detections of the systems, in chronological order:

```

SAP Data Hub Terminal Clear Output Clear Input Screen Refresh: ON
[ 1 ] Options:
[ 2 ] 1- "do for N", where N are the first (ordered) rows of the table to which we want to calculate the column HAS_DAILY_SEASONALITY.
[ 3 ] Please, do not choose a N number bigger than the total number of rows of the table, which is 227823547
[ 4 ] 2- "check content", which returns the data in the output table.
[ 5 ] 3- "restart", which clears all rows in the output table.
[ 6 ] null
[ 7 ] Getting data from PHD...
[ 8 ] Data retrieved from PHD.
[ 9 ] Dataframe imported.
[10 ] Groups formed.
[11 ] Dataframe preprocesses performed.
[12 ] Check for duplicates done.
[13 ] Daily seasonalities calculated.
[14 ] Writing results in HANA...
[15 ] All done.
[16 ] [{"CONINS": "rh5885_A4H_10", "HAS_DAILY_SEASONALITY": 0, "SESSNO": "0010000000030"}, {"CONINS": "acrnbwihanadb_AH1_00", "HAS_DAILY_SEASONALITY": 1, "SESSNO": "0010000000030"}]
[17 ] All rows have been cleared in the output table.
[18 ] null

Welcome to the SAP Data Hub Terminal
SDH > a
SDH > check content
SDH > do for 1000
SDH > check content
SDH > restart
SDH > check content
SDH >

```

Figure 36: Work process for the developed pipeline in its terminal operator.

- Lines 1–5 (result after inputting “a” or any other string but the reserved strings: “do for N”, “check content”, or “restart”): This displays the input options for the user.
- Line 6 (result after inputting “check content”): The destination table (i.e, where the final results will be stored) does not have any content at this time, and thus, the terminal displays ‘null’ when we want to check the content of this table.
- Lines 7–15 (result after inputting “do for 1000”): This is the main part of the whole process. The program takes the first 1000 rows of the PHD-MEM1 table from HANA, and processes this data to detect if the systems have daily seasonality or not. When the results are calculated, those are stored in the destination table in HANA.
- Line 16 (result after inputting “check content”): This time, because the program has written the results of the daily seasonalities in the destination table, it can show these results. The daily seasonality is indicated in the feature ‘HAS_DAILY_SEASONALITY’: 0 means negative, 1 means positive.

- Line 17 (result after inputting “restart”): It clears the destination table and displays a message on screen when it is done.
- Line 18 (result after inputting “check content”): This time, because we have cleared the destination table, the program is not able to show any content of the table, and therefore it displays ‘null’ again.

As it can be seen, the whole process of retrieving data from HANA, locally processing it, and writing it back to HANA, works correctly; therefore, we have proven the feasibility of this tool for the purposes and goals of the company, which were proposed in the *Objectives* chapter of this document.

8 Discussion

For the main objective, pattern recognition, we have seen the noisiest systems in two different scenarios thanks to the usage of the developed algorithms. They work as intended and provide correct results.

When we are trying to find the noisiest systems by using a high sensitivity peak detection (second scenario), we see that the two first noisiest systems (Figures 31 and 32) are not really interesting because we only detect two peaks that are over 30% of memory change. Nevertheless, in the third noisiest system (Figure 33), we get several of those peaks, which occur consistently in the last part of the time series. In this case, because we were in this high sensitivity scenario, we did not perform a noisy segment detection, but actually in this time series it makes sense to do so because we clearly see that there is a change point in the time series in which there is a shift in the intensity of the peaks. Therefore, in this time series it would be really interesting to go deeper to find out why after this change point the peaks are much more intense than before, e.g., by analysing the HANA memory allocators, by performing time series clustering, etc. Remember that at any time we want such intensive peaks, because the memory of the servers is limited, and we do not want to be close to the memory allocation limit frequently.

After we find out why we have time series with such intense peaks or high memory fluctuations, the next logical step is to figure out how to solve these situations, either on the company side or on the user side. Another possibility is predicting when this kind of situations can potentially happen in the future, in order to act beforehand.

Regarding the implementation on SAP Data Hub, the daily seasonality detection algorithm worked well on it and we got a proof of concept; thus, the next steps in this matter would be to implement and deploy the rest of the algorithms on Data Hub and set everything up for a final deployment to give service to SAP EarlyWatch Alert users.

9 Conclusion

We have seen that all the developed algorithms work well as they were intended to. The results of these algorithms provide insights for the users and for the system administrators that can help them to understand better how their systems are behaving. Nevertheless, at this stage we cannot propose them a fix or solution to the cases of the systems that have intense peaks or high fluctuations of in-use memory, because we still need to identify the reasons why this is happening. This is material for future research and development.

For the algorithms, a manual from-scratch approach was performed, not using or following any of the current general time series pattern recognition techniques. The diversity of our data forced us to be quite specific about what wanted to get. We needed to have full control over the algorithms and their input parameters, in order to have precise control over their outcomes.

References

- [1] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, Wolfgang Lehner. *SAP HANA Database - Data Management for Modern Business Applications*. December 2011.
- [2] Chaim Bendelac, Panos Kokkalis. *SAP HANA Memory Usage Explained*. January 2017.
- [3] Sofiane Lounici. *Anomalies Detection in SAP*. 10.12.2018.
- [4] Stephan Spiegel. *Time series distance measures*. January 2015.
- [5] Ivan Dokmanic, Reza Parhizkar, Juri Ranieri and Martin Vetterli. *Euclidean Distance Matrices: Essential Theory, Algorithms and Applications*. 15.8.2015.
- [6] Stan Salvador, Philip Chan. *FastDTW: Toward Accurate Dynamic Time Warping in Linear Time and Space*. 2014.
- [7] Laura Beggel, Bernhard X. Kausler, Martin Schiegg, Michael Pfeiffer, Bernd Bischl. *Time series anomaly detection based on shapelet learning*. September 2019.
- [8] Jessica Lin, Eamonn Keogh Li Wei, Stefano Lonardi. *Experiencing SAX: a Novel Symbolic Representation of Time Series*. 2007.
- [9] Pavel Senin, Sergey Malinchik. *SAX-VSM: Interpretable Time Series Classification Using SAX and Vector Space Model*. December 2013.
- [10] Rohit J. Kate. *Using Dynamic Time Warping Distances as Features for Improved Time Series Classification*. 12.9.2014.
- [11] Marcos Duarte. *detect_peaks.py* function. URL: https://github.com/demotu/BMC/blob/master/functions/detect_peaks.py. Last update: 3.7.2018.

A Autocorrelation Plots of the Noisiest Systems

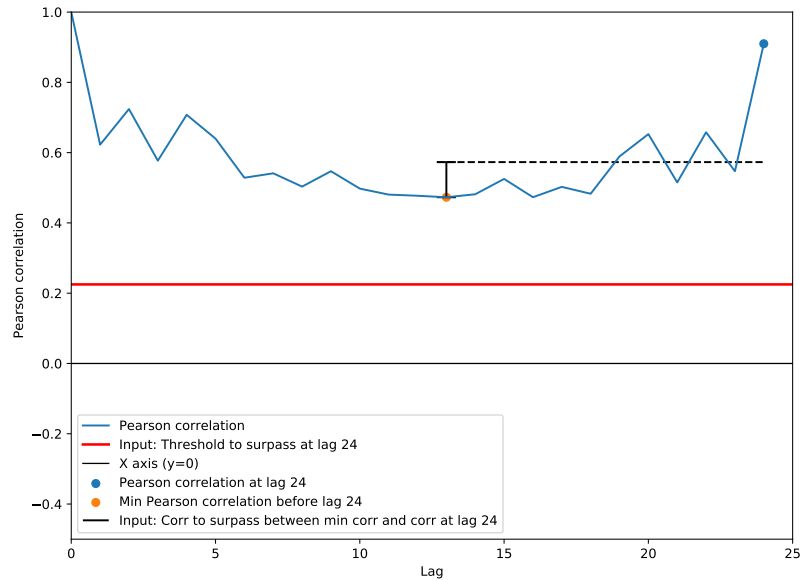


Figure A1: Autocorrelation plot of the noisiest systems by noise segmentation by peak/valley closeness *and* by peak/valley closeness and memory distance. The system has daily seasonality.

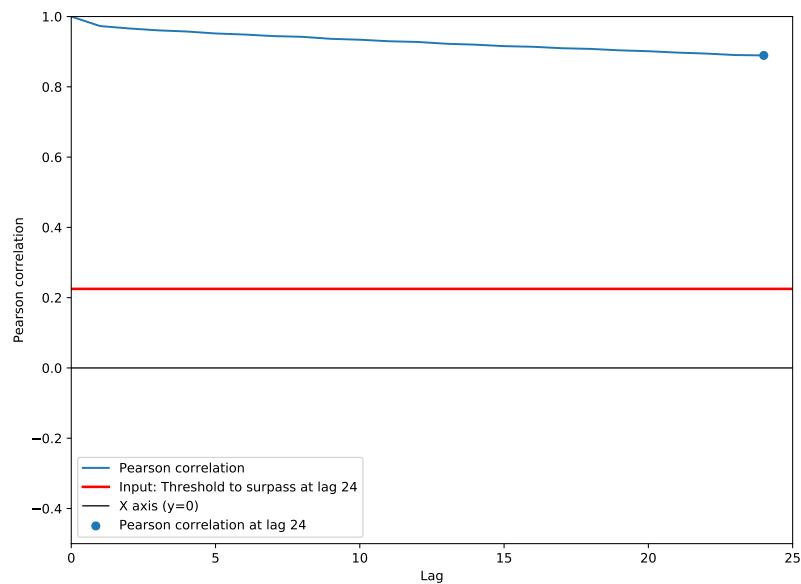


Figure A2: Autocorrelation plot of the second noisiest system by noise segmentation by peak/valley closeness only. The system does not have daily seasonality.

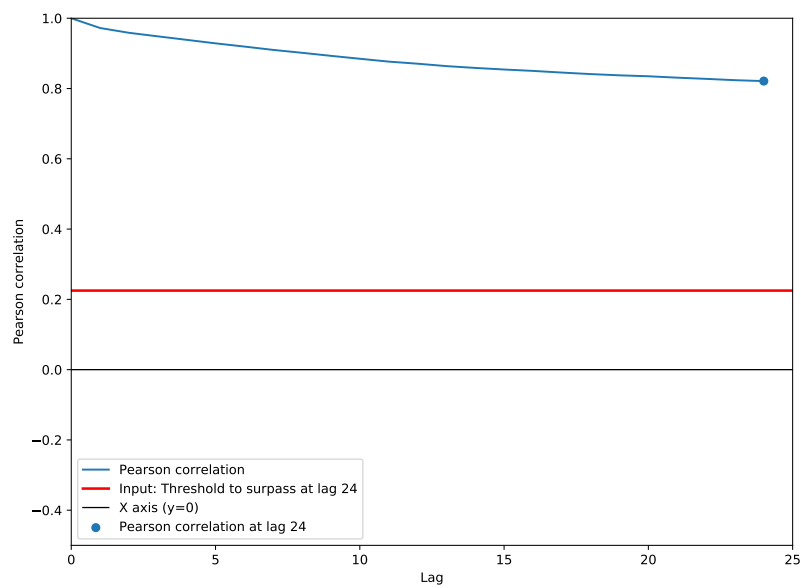


Figure A3: Autocorrelation plot of the second noisiest system by noise segmentation by peak/valley closeness and memory distance. The system does not have daily seasonality.

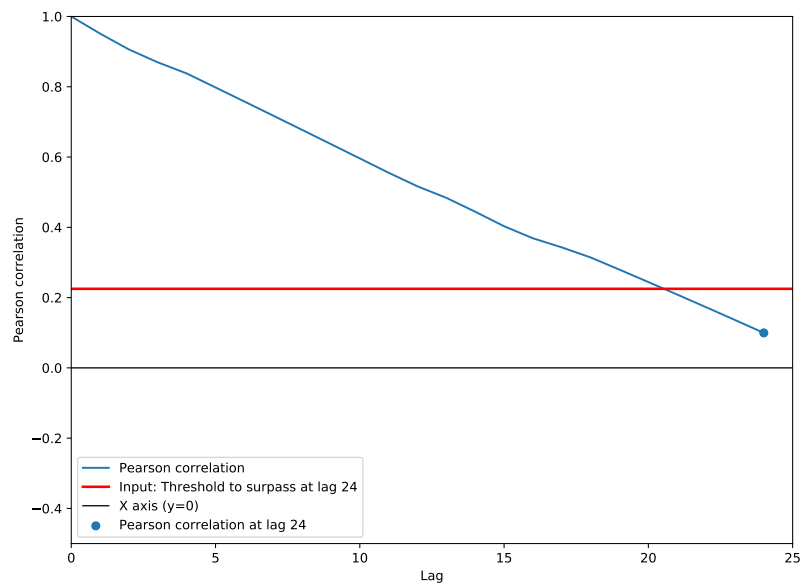


Figure A4: Autocorrelation plot of the noisiest system by 30% peak threshold detection. The system does not have daily seasonality.

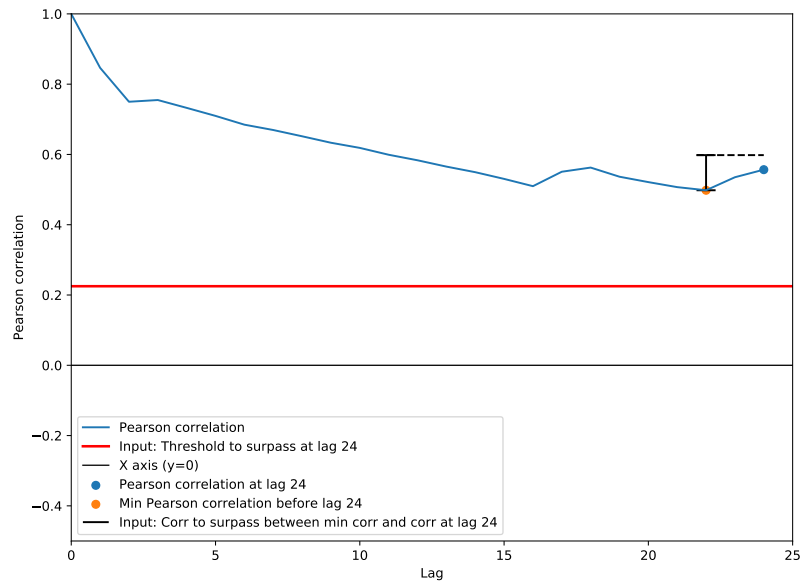


Figure A5: Autocorrelation plot of the second noisiest system by 30% peak threshold detection. The system does not system have daily seasonality.

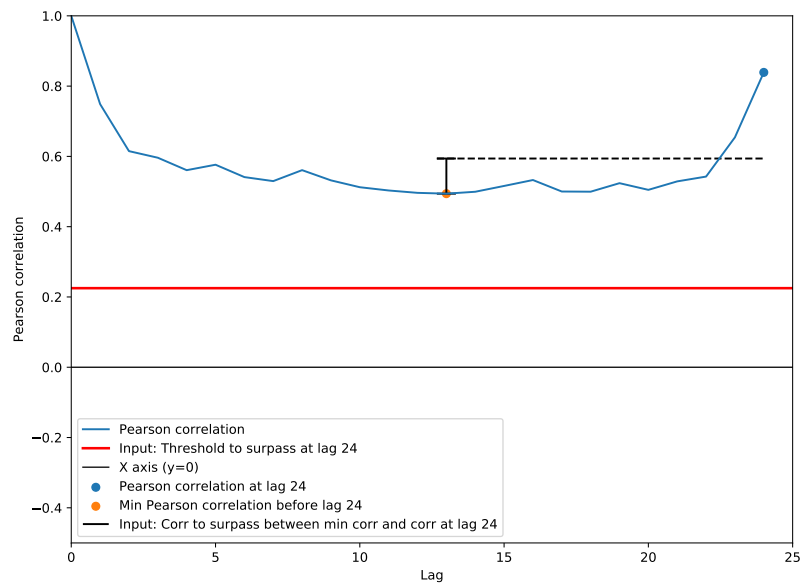


Figure A6: Autocorrelation plot of the third noisiest system by 30% peak threshold detection. The system has daily seasonality.