University of Piraeus

Department of Digital Systems

Master Program in

«Digital Communications and Networks»

Master's Thesis

Design and Implementation of a Knowledge Generation and Visualization tool over the SAP HANA Platform in the "Big Data" context

Morakos Petros - ME12071

Supervisor:
Professor Panagiotis Demestichas

Piraeus 2014

Preface

This thesis is made as a completion of the master education in "Digital Communications and Networks" master program of the "Department of Digital Systems" of University of Piraeus. This work was executed into the "Telecommunication Networks and integrated Services (TNS)" Laboratory of the University of Piraeus.

Several persons have contributed academically, practically and with support to this master's thesis. Therefore, I would like to thank my head supervisor Professor Panagiotis Demestichas and the co-supervisor Dr. Konstantinos Tsagkaris for their academic and technical guidance during the completion of this dissertation. Furthermore, I would like to thank the "Telecommunication Networks and integrated Services" Laboratory members and, especially, Mr. George Poulios and Mrs. Aimilia Bantouna for their valuable technical input and support throughout the entire dissertation period.

Finally, I would like to thank my family and friends for being supportive during my studies in the master program of the "Department of Digital Systems" of University of Piraeus.

University of Piraeus

October 2014

_____

Petros Morakos

Contents

1 Introduction

Many aspects and research initiatives have shown the importance of information that are produced by the applications and the user devices. Nowadays, we receive information by different sources such as the social networking, the Rich Site Summary (RSS) feeds etc. [1]. As it is understood, these various sources of information provide disparate data. Furthermore, the user devices imposes further increases in the volume, the velocity and the variety of the data. This fact results to a difficult data management [1]. This phenomenon in digital systems is called as the "Big Data" phenomenon.

The "Big Data" phenomenon is about the heavy analysis of the data. This analysis is required in order to retrieve the meaningful insights from the large volumes of data. New architectures has been proposed for (i) handling the volume of data, (ii) aggregating, (iii) exploiting and (iv) building knowledge on them [1]. In the Telecommunication Networks field, the "knowledge" refers to the meaningful information that the network operators needs to retrieve from his network infrastructure and it cannot be monitored. The European Committee for Standardization says that "the knowledge is the combination of data/information with the opinions/skills/experience of experts, which can be humans or computational systems and results in a valuable asset that can be used to aid decision making" [1], [2].

A huge issue for the network operators is that the resource management of their infrastructure is fulfilled through a manual procedure which means that they collect and manage the information of their network in a manual way [1]. This is a big obstacle for their decision making process and an opposite direction from the tendency of the automated resource management and automated decision making that all the research initiatives impose [1]. These automated procedures for dynamic configuration and reconfiguration of the network infrastructure will result to an decrease in Operational Expenditure (OPEX) and Capital Expenditure (CAPEX) of the network enterprises.

In general, the dynamic resource planning of the network is the next step for the network infrastructures as we are in a continuously changing environment with different needs and requirements for the users. So, it is worth to induce prediction loads in the network infrastructures in terms of the near or distant future in order to perform dynamic network planning [1] and proceed to a more sophisticated decision making.

This work proposes a machine learning based tool that exploits the SAP HANA Platform powered by the SAP A.G. which is a software platform that "brings to the light" the fundamentals (i.e. in-memory computing etc.) of the "Big Data" phenomenon. In this context, we suggest a tool for building knowledge on network load and predicting it. In Figure 1, we illustrate the problem statement.

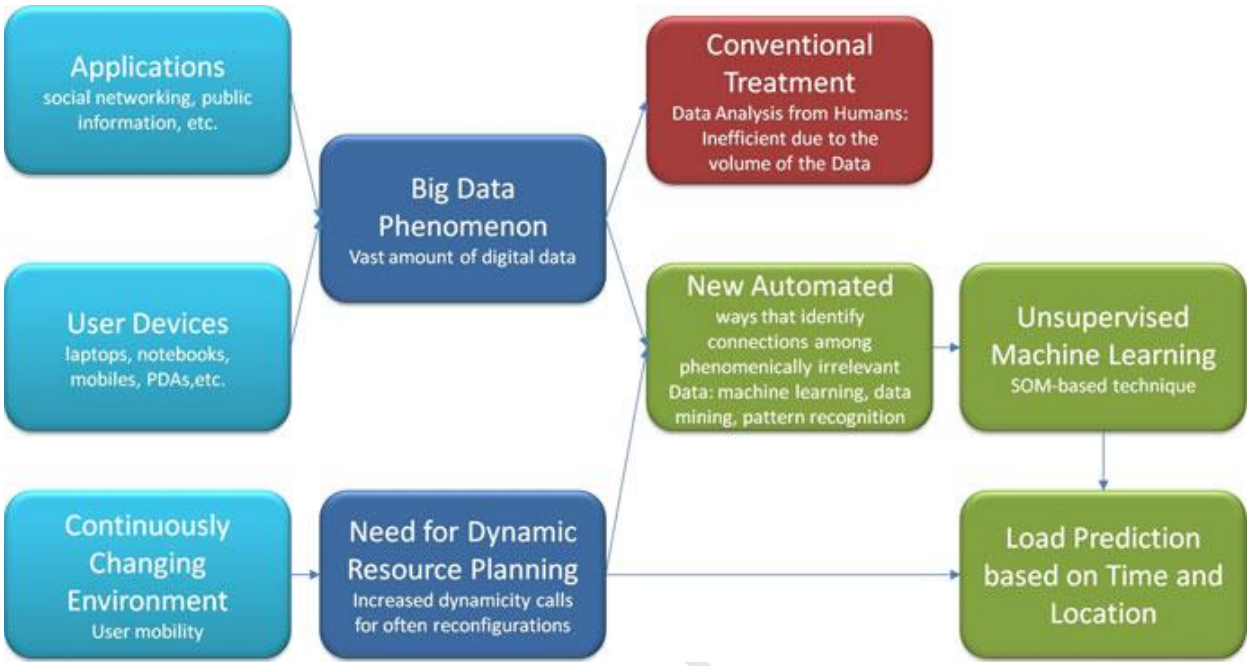

Figure 1: Motivation and high level problem statement [1].

Essentially, the main problem is analyzed as follows [1]:

- The requirement for a mechanism which manipulates the disparate data, thus the "Big Data", and transform them into a meaningful format for the network operator in order to build knowledge and, eventually, store a decreased instance of them.
- The requirement for predictions of network load for the achievement of automated dynamic network resource planning.

The rest of this work is structured as follows:

- Chapter 2 discusses the Self-Organizing Maps algorithm and its formulation.
- Chapter 3 analyzes the theory, the aspects and the capabilities that the SAP HANA Platform provides to the enterprises in order to build software into the "Big Data" context.
- Chapter 4 illustrates the design and the implementation aspects of the proposed machine learning based tool.
- Chapter 5 depicts examples regarding the functionality of the tool and it provides some indicative results.

2 Self-Organizing maps (SOM)

2.1 Introduction

In this chapter, we will describe the main issues regarding the theory and application of the Self-Organizing Maps (SOM) algorithm which is the main machine learning algorithm of the Knowledge Generation and Visualization (KGV) tool. The Self-Organizing Maps will be used as the main training algorithm of the tool in which it will participate in the training process of a specified use-case taken either from the Telecommunication field or another field (i.e. energy). More specifically, the KGV tool uses an extension of the Self-Organizing Maps which is called Parameter-less Growing Self-Organizing Maps (PLGSOM). In our demonstration, we present use-cases taken from the Telecommunication domain.

In the following paragraphs, we will present the main theory of the Self-Organizing Maps, the main theory of the extension that we applied in our tool, the Parameter-less Growing Self-Organizing Maps, the metrics that we used for the estimation of the distances between the units in a Artificial Neural Network (ANN).

2.2 Supervised vs. Unsupervised Learning

An important aspect of an Artificial Neural Network (ANN) model is whether it needs guidance in learning or not. Based on the way they learn, all artificial networks can be divided into three learning categories:

- The supervised learning approach
- The unsupervised learning approach

In supervised learning approach, a desired output result for each input vector is required when the network is trained. An ANN of the supervised learning type, such as the multi-layer perceptron, uses the target result to guide the formation of the neural parameters. It is thus possible to make the neural network learn the behavior of the process under study.

In unsupervised learning, the training of the network is entirely data-driven and no target results for the input data vectors are provided. An ANN of the unsupervised learning type, such as the self-organizing map, can be used for clustering the input data and find features inherent to the problem.

2.3 Self-Organizing Maps Theory

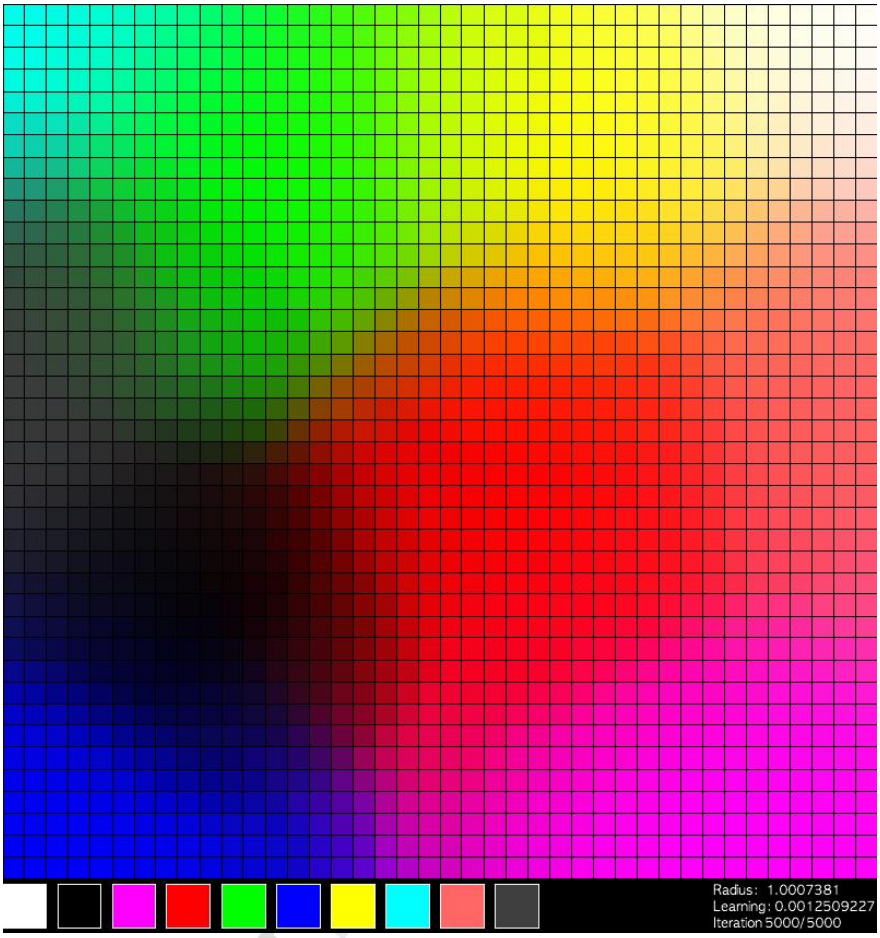

Figure 2: Two-dimensional Self-Organizing Map

The Self-Organizing Maps (SOM) algorithm was proposed by professor Teuvo Kohonen [3]. It is one of the most popular neural network models and it belongs to the category of competitive learning networks. This algorithm is based on unsupervised learning, which means that no human intervention is needed during the learning and that little needs to be known about the characteristics of the input data.

The SOM algorithm provides a topology preserving mapping from the high dimensional space to map units. Map units, also known as neuron, usually form a two-dimensional lattice and thus the mapping is mapping from high dimensional space onto a plane.

The property of topology preserving means that the mapping preserves the relative distance between the point. The points that are near each other in the input space are mapped to

nearby map units in the SOM. Thus, the SOM can serve as a cluster analyzing tool of high-dimensional data.

The SOM algorithm has, also, the capability to generalize. Generalization capability means that the network can recognize or characterize inputs it has never encountered before. A new input is assimilated with the map unit it is mapped to.

2.3.1 The main SOM Training Algorithm

In this section, we study the issues which are related with the SOM algorithm that is applied on the training process of the KGV tool. The SOM is a neural network-based approach, as already mentioned above. Training of SOM is the shaping of a so-called map, which is basically a 2-dimensional representation of high dimensional input data [3] - [5].

In the training phase, input samples (in the form of vectors that describe the state of the system) are mapped onto the 2D grid in a competitive manner. This process is also known as vector quantization. In this traditional version of the SOM algorithm we have fixed-size maps, which means that the initial configured size of the map remains the same during the training process [4].

Each input sample is a vector $x \in R^n$ and it is consisted with features and a label. The units of the SOM map are the neurons of the neural network. Each unit represents a vector of weights $m \in R^n$ which is adjusted during the training process of the Self-Organizing Algorithm (SOM). The length of weights vectors is equal with the number of features of the input samples. Essentially, the weight vectors of the units are initialized either randomly or by taking its values from a set of values [4].

Let's assume now, that we have a dataset of vectors. The general process is the following [4]-[5]:

1. We set the number of the initial units of the map. Then, we initialize (i.e. randomly etc.) the vectors of weights for each unit.
2. The algorithm retrieves a vector from the dataset and it estimates the Best Matching Unit (BMU) for this vector. The Best Matching Unit is the unit of the map which is "closest" to the selected vector each time. Usually, the Best Matching Unit is estimated by applying the Euclidean distance metric between the vector and each unit of the SOM map.

$$\|x - m_c\| = \min_i\{\|x - m_i\|\}, \quad (1)$$

where m is the vector with the weights of a unit,i is the unit of the map in each iteration and c is the BMU regarding the vector c. Essentially, the $\|.\|$ represents the distance measure

3. After we find the BMU, the vectors of all units of the SOM map are updated by applying the following formula:

$$m_i(t+1) = m_i + a(t)h_{ci}(r(t))[x(t) - m_i(t)], \quad (2)$$

where t denotes time, $a(t)$ denotes the learning rate and $h_{ci}(r(t))$ denotes the neighborhood kernel around the winner unit c, with neighborhood radius $r(t)$.
Essentially, the vectors of the BMU are moved closer to the input vector in the input space. Such an adaptation process stretches the weights of the BMU and its neighbors towards the sample vector. This is, perfectly, depicted in the Figure 3.

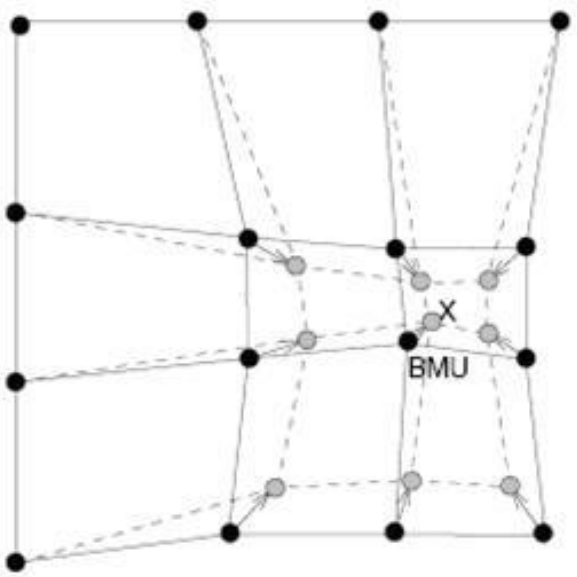

Figure 3: Updating of the SOM map

The neighborhood function $h_{ci}(r(t))$ determines how strong are the connections between the neurons each other. Essentially, the neighborhood function and the number of neurons determine the granularity of the resulting mapping. The larger the area where neighborhood

function has high values, the more rigid is the map. The larger the map, the more flexible it can become. This interplay determines the accuracy and generalization capability of the SOM. The neighborhood function is the following [4]-[5]:

$$h_{ci} = a(t)exp\left(-\frac{\|r_c - r_i\|^2}{2\sigma^2(t)}\right), \quad (3)$$

where $a(t) \in (0,1)$ is the learning rate, r_c , $r_i \in R^2$ are the vectors of the units of the SOM map and the $\sigma(t)$ is the magnitude of the neighborhood function.

The learning rate $a(t)$ and the neighborhood function $\sigma(t)$ decreases monotonically as the training process progresses.

Generally, the three steps of the SOM algorithm that we described above is repeated for every vector. After we passed all the vectors this means that an epoch has been finalized. The purpose is to execute as many epochs as it is needed in order to have a well-adapted SOM map corresponded to the input dataset sample vectors.

2.3.2 The extension of Parameter-less Self-Organizing Maps (PLGSOM) algorithm

In the Knowledge Generation and Visualization tool, we apply an extended version of the Self-Organizing Maps. This extension is called Parameter-less Growing Self-Organizing Maps [6] - [8].

This approach aims to satisfy the two main issues of the main SOM algorithm. The first issue is related with the fixed size of the SOM map that the training algorithm encounters. This means that the user has to be experienced enough to set appropriately the stable dimensions of the SOM grid. The issue of the fixed SOM map imposes the under/over-fitting during the training process of the SOM algorithm. The second issue is related with the configuration of the learning rate $a(t)$ and the magnitude of the neighborhood function $\sigma(t)$. Again, an experienced user is needed in order to set their initial and final values.

For the purposes of overcoming the above issues, we decided to apply the Parameter-less Growing Self-Organizing Maps so as to increase appropriately the dimensions of the map during the training process and for the dynamic configuration of the learning rate $a(t)$ and the magnitude of the neighborhood function $\sigma(t)$.

Dynamic Increase of SOM map dimensions

For the purposes of the dynamic increase of the dimensions of the Self-Organizing Map 2D grid, we focus on the specification of some concepts or configuration parameters [6]:

- **The quality of the SOM map**: Conversely, we can mention this parameters as the error of the units.
- **The training cycle /or the cycle duration**

Essentially, the cycle duration is the number of iterations during the training process. After this set of iterations are fulfilled, the algorithm checks if the Self-Organizing Maps grid will be increased or not. As we can understand, this concept aims to control the dimensions of the SOM 2D map. Generally, these number of iterations we could call them as an *epoch*. In addition, the number of iterations are initialized, usually, as the number of the input dataset sample vectors that we insert to the training algorithm PLGSOM.

On the other hand, the quality of the SOM map is measured by applying some error measurements techniques such as:

- The quantization error
- The classification error
- The mean quantization error
- The mean quantization-classification error

We apply the above error estimation techniques for each unit i.
In particular, the quantization error is declared as [6]:

$$qe_i = \sum_{x_j \in C_i} \|m_i - x_j\|, \quad (4)$$

where m_i is the weights of a unit of the SOM 2D map and the $x_j \in C_i$ are the vectors of the input dataset that have been mapped onto the unit i.

The mean quantization error is declared as [6]:

$$mqe_i = \frac{1}{n_c} qe_i, \tag{5}$$

where n_c is the number of the vectors of the input dataset that have been mapped onto the unit i.

The classification error is declared as:

$$cle_i(t) = \frac{k}{k-1}\left(1 - \frac{\max_{l \in L}\{f_{li}(t)\}}{n_i(t)}\right), \quad n_i(t) \neq 0$$
$$cle_i(t) = 1, \ n_i(t) = 0 \tag{6}$$

where:

- $cle_i(t)$ is the classification error of unit i.
- L is the set of the labels/ classes.
- $k = |L|$
- $f_{li}(t)$ is the amount of vectors which have label l and they have been mapped onto the unit i of the SOM map.
- $n_i(t) = \sum_{l \in L} f_{li}(t)$

Then, the mean quantization-classification error is declared as:

$$qce_i = cle_i * mqe_i \tag{7}$$

Generally, we described above the estimation error techniques in order to measure the error of each unit and , thus, the user could pick one of them.

Essentially, after each epoch is finalized, the algorithm performs the error estimation for each unit. The algorithm selects the unit *e* with the greatest error in the SOM map and it starts the dynamic increase of map process. This is performed by adding a new row or a new column near the unit *e* with the greatest error in the SOM map [6].

The algorithm selects the nearest neighbors of the unit e. Afterwards, from these neighbors, we select the unit d which has the greatest Euclidean distance from the unit e:

$$d = arg\max_{i \in N_e}\{\|m_e - m_i\|^2\}, \quad (8)$$

where N_e is the set of the four nearest neighbors (i.e. up, down, left, right) of the unit e with the greatest error in the SOM map.

After the algorithm have selected the unit e and the unit d , we proceed to the insert of a new row or a new column into the SOM 2D map. In case that the unit e and the unit d are placed on the same row of the SOM 2D map then, the algorithm inserts a new column. Otherwise, the algorithm will append a new row.

After the algorithm inserts the new units into the SOM map, it estimates the weight vectors of each of them. In particular, if the algorithm has inserted a new column, the weights of the newly inserted units will be estimated as the mean of the units that are placed on the right and left side of each of them. Similarly, if the algorithm has inserted a new row, the weights of the newly inserted units will be estimated as the mean of the units that are placed at up and down side of each of them. The Figure 4 illustrates clearly this procedure.

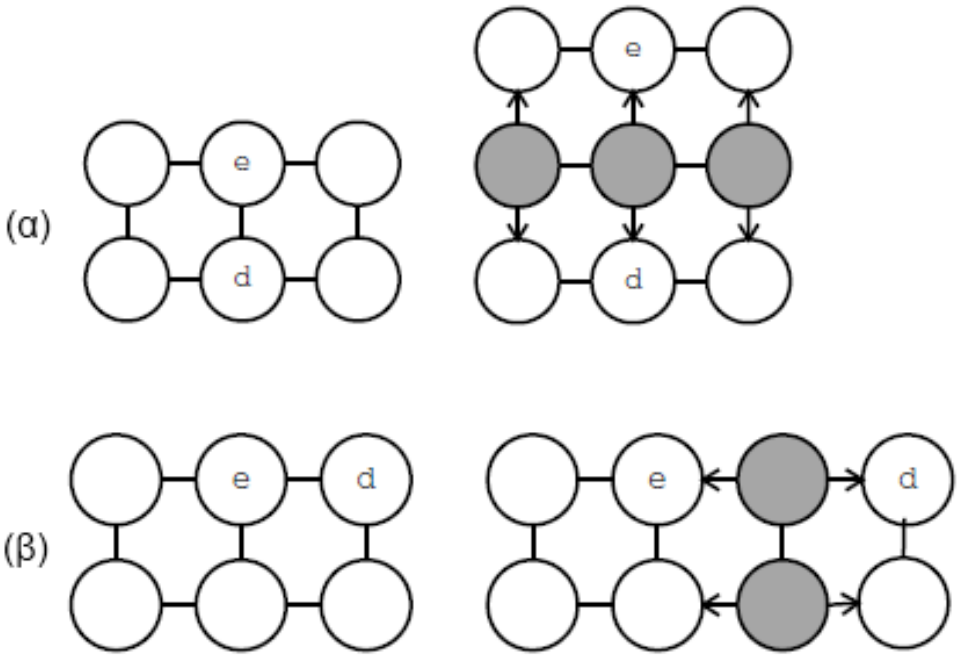

Figure 4: Adding units to the SOM 2D map [6]: a)Adding a new row, b) Adding a new column.

Dynamic Parameters Configuration

In this section, we provide the details about the dynamic configuration of the parameters, the learning rate $a(t)$ and the magnitude of the neighborhood function $\sigma(t)$. For this process, we applied the method proposed by the works in [7] and [8]. More specifically, the learning rate is estimated at each iteration during the training process. The learning rate estimation takes into account the distance of the sample vector from the BMU and the greatest sample vector distance that is present on the SOM map:

$$a(t) = \frac{\|x(t) - m_c(t)\|^2}{r(t)}, \tag{9}$$

where $r(t)$ is the greatest sample vector distance that is present on the SOM map until the step t.

The $r(t)$ is calculated as follows:

$$r(t) = \max\left(\|x(t) - m_c(t)\|^2, r(t-1)\right), \tag{10}$$

and:

$$r(0) = \|x(0) - m_c(0)\|^2, \tag{11}$$

As far as the magnitude of neighborhood function $\sigma(t)$, similarly, it is estimated at each iteration during the training process in every epoch. It is estimated by taking into account the learning rate $a(t)$:

$$\sigma(a(t)) = \max\left(\sigma_{max} * a(t),\ \sigma_{min}\right), \tag{12}$$

where :

$$\sigma_{min} = 1, \tag{13}$$

$$\sigma_{max} = \sqrt{2} * \lambda * d, \tag{14}$$

In (14) we assume that :

$$0 \leq \lambda \leq 1$$

which is configured by the user. In addition, the d is the diameter of the biggest fully inscribed circle on the SOM map. Essentially, the neighborhood function is defined as follows:

$$h_{ci} = a(t) exp\left(-\frac{\|r_c - r_i\|^2}{2\sigma^2(t)}\right), \tag{15}$$

Generally, the adaptation of the units could be fulfilled by applying the typical formula (2).

Essentially, the result of the training process of Parameter-less Growing Self-Organizing Maps algorithm is the steep adaptation of the SOM map in data that are unprecedented. In addition, the SOM map can be adapted in a smooth way for data that are near to the already observed data.

Termination Criteria

As the learning rate $a(t)$ doesn't always decrease, we should define new termination criteria of the PLGSOM algorithm. In particular, we should set rules of stopping the training process and, especially, the growing process that imposes the PLGSOM algorithm. Thus, we have defined three rules as the termination criteria:

- The quality of the map
- A time interval
- SOM map area

As far as the quality of the map, we define a percentage (i.e. 0 - 100%) which represents the required quality that the SOM map should be satisfy. This parameter is defined as :

$$\frac{1}{error}, \tag{16}$$

where $error$ represents the error that was defined before in (4) - (7).

As far as the time interval as the termination criteria, the user just defines the maximum time period that the training process will be executed. Finally, as far as the SOM map are as a potential termination criteria, the user just defines the maximum area that he wants the SOM map to grow, e.g. < 900 units.

2.4 Classification methods using the SOM map

In this chapter, we introduce the details about the classification techniques that we apply in the Knowledge Generation and Visualization (KGV) tool in order to map the input samples either in validation process or the prediction process of the tool.

2.4.1 Best Matching Winners

The Best Matching Winners method selects the n nearest winner units as far as the dataset y. The method classifies each sample vector in the unit which is the nearest between the n-winners. If we face equality between the n-winners, the method chooses one randomly.

2.4.2 k-Nearest Winners

The k-Nearest Winners method selects the k nearest units as far as the dataset y. In this method, we take into account all the units of the SOM map, not only the winners as we did in the Best Matching Winners. From each of the k-nearest winners, the method finds the nearest unit as far as the position in the map and the vector sample is classified at the greatest label in that unit. If we face equality, the method chooses one randomly.

2.4.3 Nearest Winner's Average

The Nearest Winner's Average method selects the nearest unit to the vector sample and it estimates the mean average of the labels in order to attach it to the vector sample. This method is used in datasets with real values as labels. Initially, the method selects the winner unit which is the nearest to the vector sample. Afterwards, it estimates the mean average of its input label that have been mapped in this unit during the training process. This mean average is the estimated label of the given vector sample.

3 SAP HANA Platform

3.1 Introduction

In Big data era that we going through, the applications are needed to be built in a way for fast provisioning of information and data. Thus, the systems that are hosting the applications and the data should adapt this notion. In general, there are a number of software packages that provide data warehousing and fast retrieval of data through their technology. In this context, many of them are resided in distributed environment and they have adopted the in-memory computing technology for fast interactions with data and accelerated provisioning of them.

In this chapter, we analyze the role of the SAP HANA Platform in the design of the Knowledge Generation and Visualization tool. Generally, SAP HANA supports the in-memory computing technology. It is deployable as an on-premise appliance, or in cloud environments. In this context, we can say that it is a flexible data source agnostic in-memory data platform that allows customers to analyze large volumes of data in real-time. In addition, it is presented as a development platform, providing an infrastructure and tools for building high performance software based on SAP HANA Application Services (SAP HANA XS) [10]. So, as we can understand the SAP HANA is not just a core database but, also, it is a full application server for building and executing applications.

So, in the following sections, we present the architecture and the main capabilities that the SAP HANA provides to users and applications.

3.2 SAP HANA Architecture

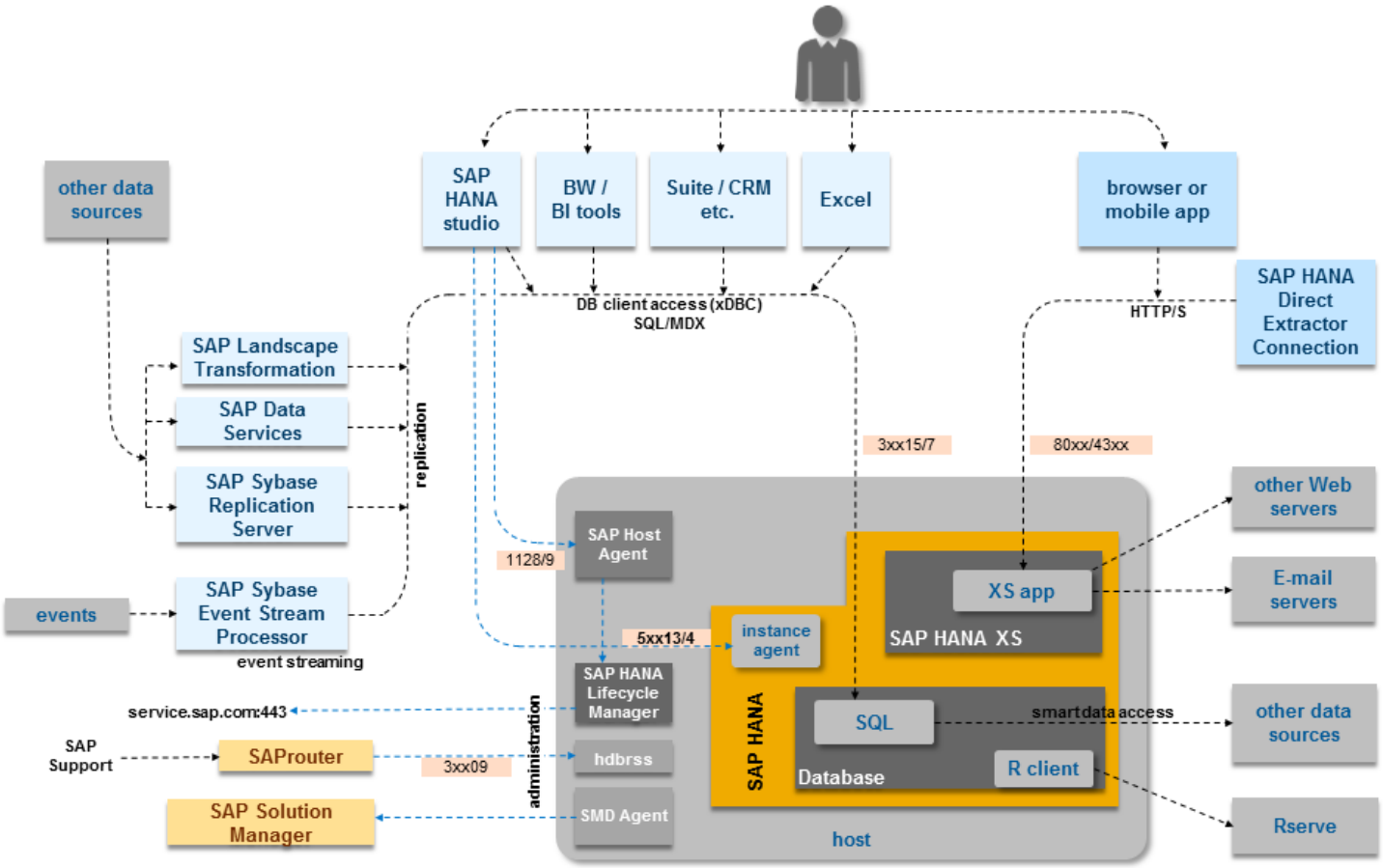

Figure 5: SAP HANA Platform architecture [10]

In this section, we describe, in further details, the general architecture of the SAP HANA Platform. Figure 5 illustrates the main components of the SAP HANA and the entities that interacts. Mainly, in KGV Tool, we exploit:

- The SAP HANA Database
- The SAP HANA XS

Also, in our development process, we used the SAP HANA Studio as our main Integrated Development Environment (IDE). It's a software which gives the capability to the user to design the database and build applications based on the SAP HANA Extended Application Services (SAP HANA XS).

In general, the SAP HANA Platform is an in-memory data platform whose core functionality is an innovative in-memory relational database management system. In this context, it can increase application performance and the designer/ developer can build applications that integrate the business control logic and the database layer with unprecedented performance. Thus, the only thing that the developer should take into account how he can minimize the data movements. The essential answer to this is that the more he can do directly on the data in memory next to the CPUs, the better the application will perform. Generally, this is the important aspect of the SAP HANA platform [11].

3.2.1 SAP HANA Database

The SAP HANA is based on multi-core CPUs with fast communication buses between the processor cores. In addition, it is assumed that it is consisted of terabytes of main memory. This is done, for the purposes of containing the data into the main memory, thus, minimizing the performance cost of disk I/O. However, the platform needs the safety of the disks for permanent persistency in the event of a power failure or another catastrophe. In addition, back up tasks are performed periodically into the system. These tasks doesn't slow down the performance of the system because they are performed asynchronously as background tasks [11].

Figure 6 depicts the entities that the database layer is consisted. One of core functionalities of the database layer is the database/relational engines. It contains two engines that form the representation of the data [9]:

- The Column-based store
- The Row-based store

Essentially, the SAP HANA Platform is optimized based on the Column-based store. This doesn't mean that it doesn't efficiently support the Row-based store. SAP HANA supports both, but it is particularly optimized for the Column-based store [11].

Column vs Row Store

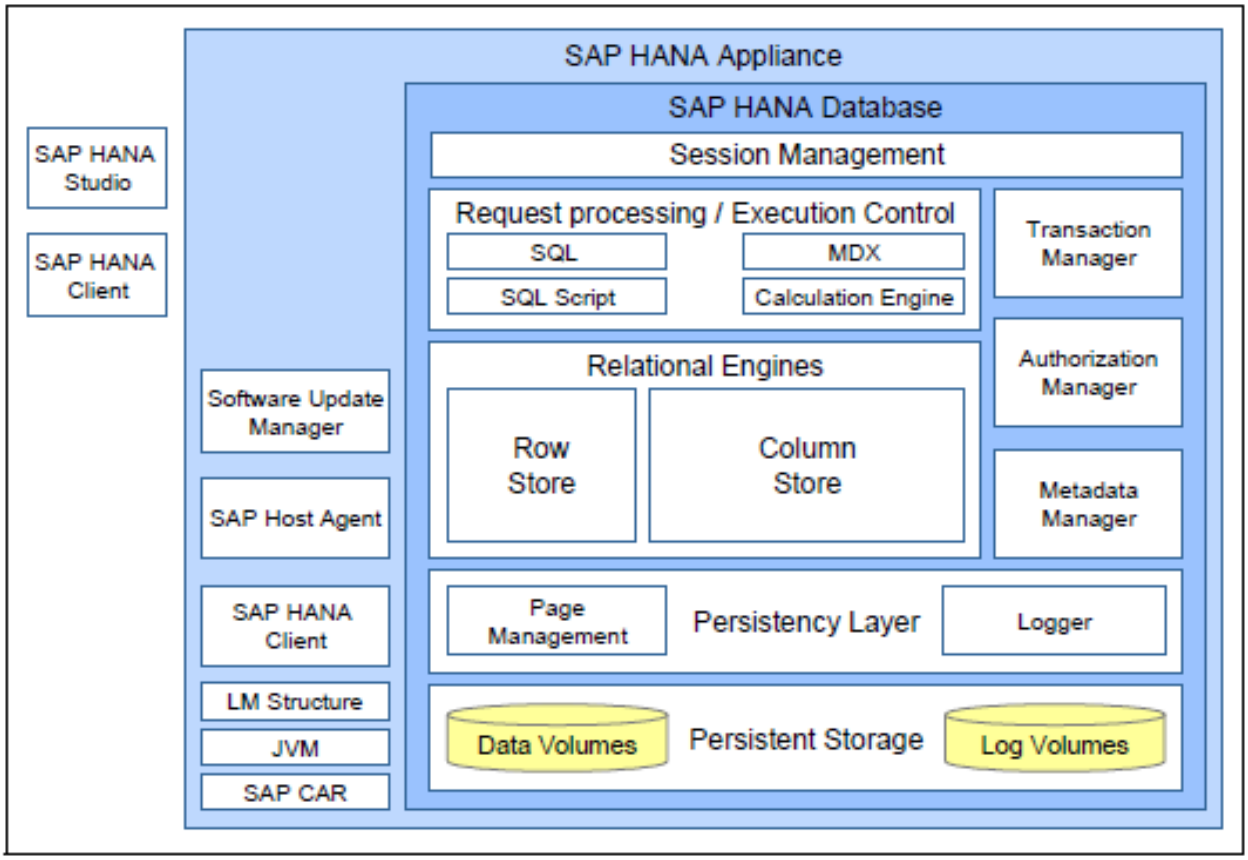

Figure 6: SAP HANA Architecture Overview [9]

Generally, a database table is a two dimensional data structure consisted of rows and columns while, on the other hand, the computer memory is represented as a linear structure [11]. As we can observe in Figure 7, the Row-based store organizes a table as a sequence of records in the main memory. In particular, this row store is optimized for write operations and it has a lower compression rate. In addition, its query performance is much lower compared to the Column-based store [9].

On the other hand, the Column-based store is for storing relational data in columns, optimized for holding data mart tables with large amounts of data, which are aggregated and used in analytical operations (i.e. Analytic Views, Calculation Views etc.) [9]. As we can observe in Figure 7, in column storage the entries of a column are stored in contiguous memory locations. This approach, opposite to the Row-based store, provides highly efficient compression. In case of sorted column, potentially, there are repeated adjacent values. SAP HANA employs highly efficient compression methods, e.g. run-length encoding, cluster coding and dictionary coding [11].

The Column-based store eliminates the need for additional index structures. In particular, this approach provides a similar method with the built-in index for each column. It allows read operations with very high performance. This is resulted from the speed of the in-memory technique and through the compression mechanisms - especially the dictionary compression. So, the fact that the Column-based store eliminates the additional indexes, reduces complexity [11].

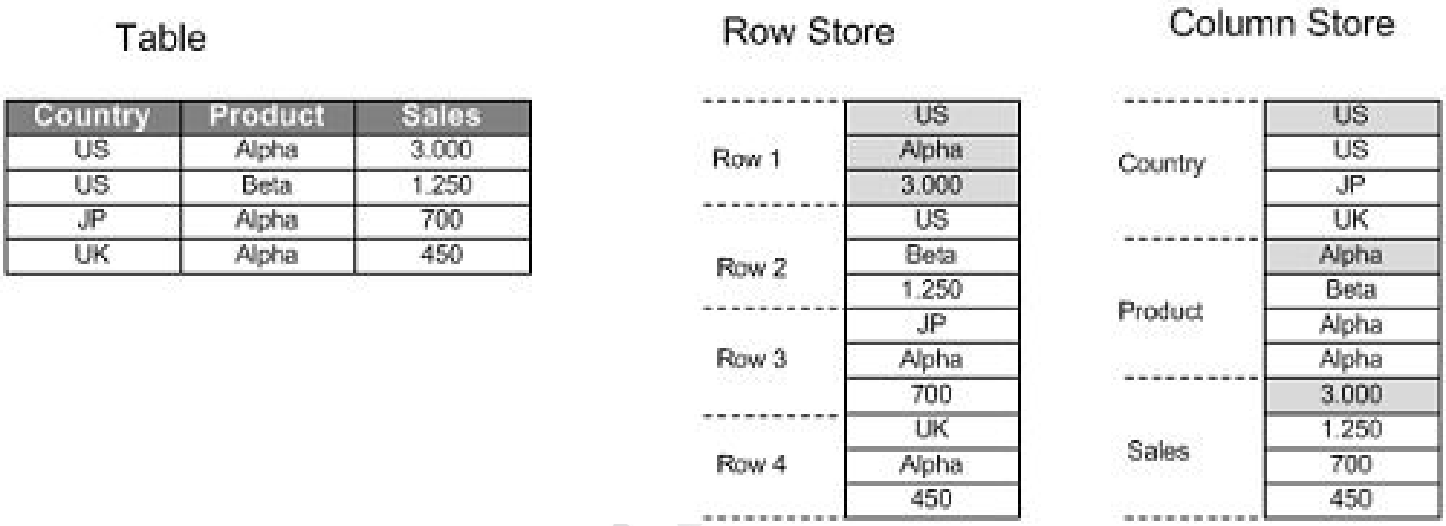

Figure 7: SAP HANA DB engines - Column vs Row store [11]

Essentially, the tables which apply the Row-based store approach are loaded into memory at start-up time, whereas the tables that adopt the Column-based store approach can be either loaded at start-up or on demand, during normal operation of the SAP HANA database [9].

The Column-based approach gives the opportunity for processing some tasks, such as joins, aggregations and scans / searching, in parallel. The multiple cores, that the SAP HANA lies up on, are used at the same time, fully utilizing the available computing resources of the distributed environment. In particular, the operations on single columns can be represented as loops over an array stored in contiguous memory locations. These operations can be executed at the CPU cache [11]. On the other hand, the Row-based store executes the same operations slower because of the distribution of the same column data across the main memory. So, in this approach the CPU is, also, slowed down too because of the misses of the cache [11].

The efficient compression of data, that the Column-based store offers, accelerates the CPU cache executions. This is done because we face limited data transport between the memory

and the CPU cache, thus, the performance and the response times are greatly better. Generally, the performance gain exceeds the additional computing time needed for decompression [3].

As we have seen in Figure 7, the data has been vertically partitioned in the Column-based approach. In this context, the data can be easily processed in parallel. A number of columns can be searched or aggregated by different processor cores. Moreover, a column can be partitioned and, then, processed in parallel by the different cores (Figure 8) [11].

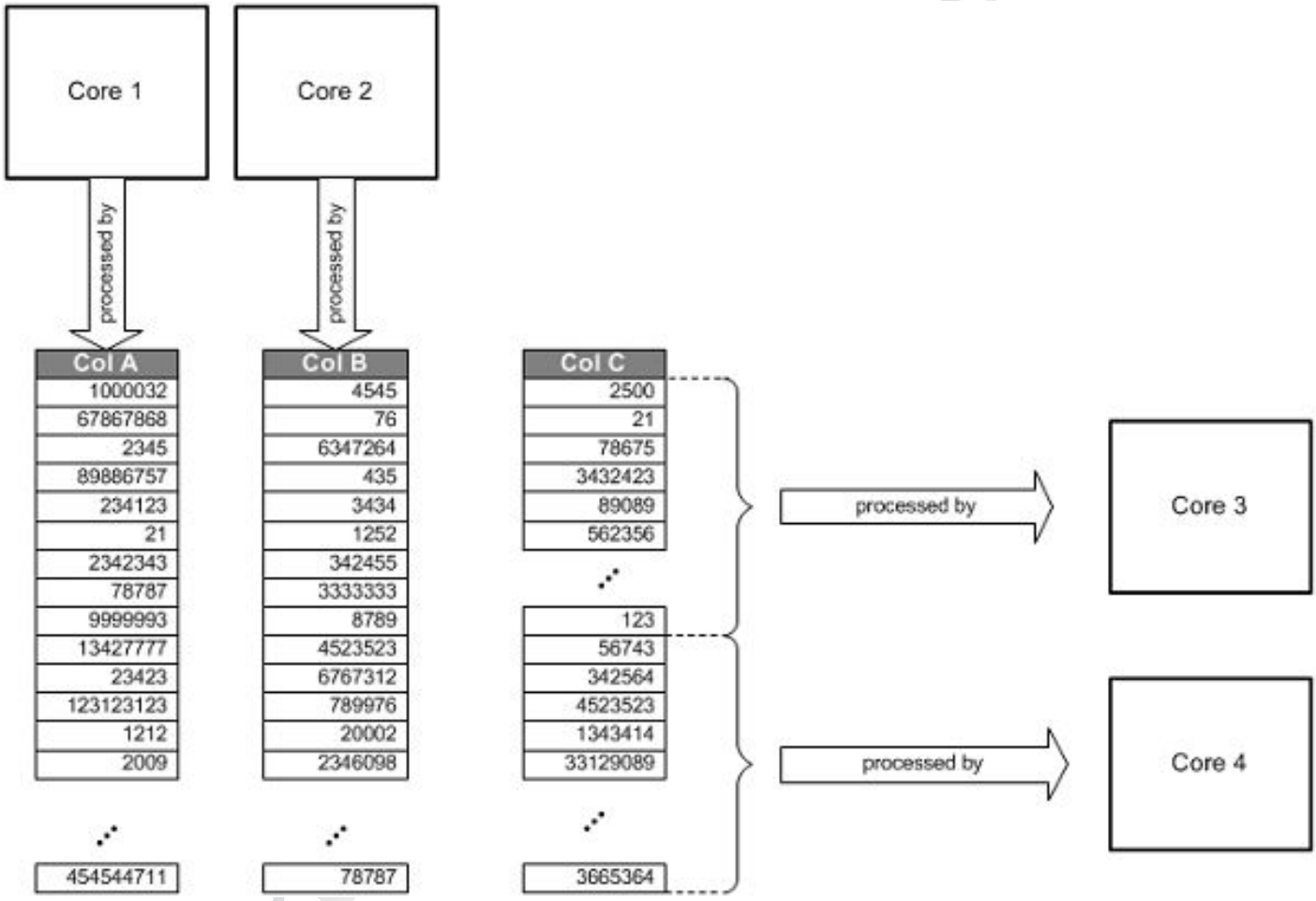

Figure 8: Parallel Processing in SAP HANA [11]

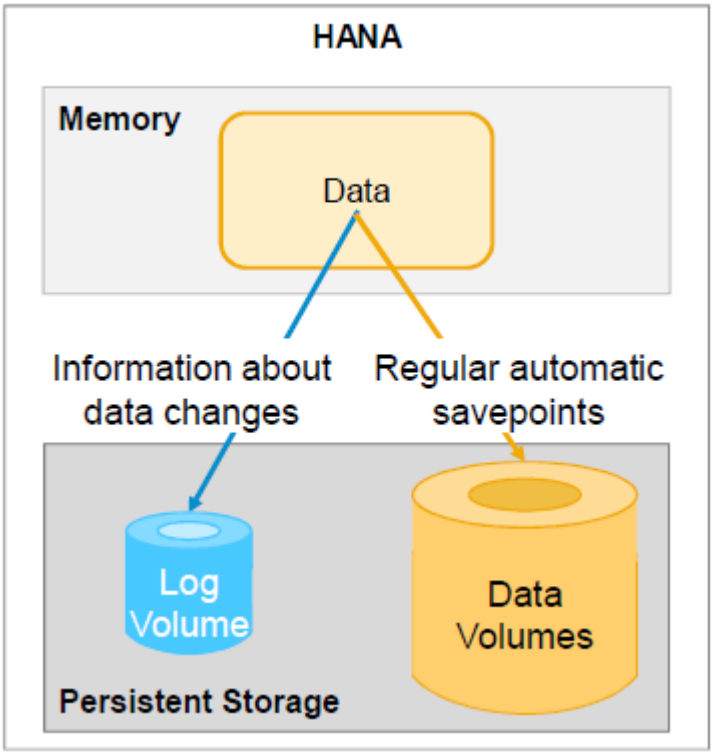

Figure 9: SAP HANA Persistency layer [9].

Back-up and Recovery in SAP HANA Database

In SAP HANA architecture, under the relational / database engines, there is the persistency layer. It provides data persistency across the engines. It is responsible for the durability and the atomicity of transactions. Thus, it assures that the database will be restored to the last committed state after a restart. In this context, it ensures that the transactions are either completely executed or completely undone.

This layer offers the savepoints functionality which writes the data volumes to permanent storage (i.e. on disks). When a transaction commits in the SAP HANA database, the logger of the persistency layer persists it in a log entry to the log volumes on persistent storage (Figure 9) [9].

Essentially, the SAP HANA maintains in the format of savepoints two types of data in storage (Figure 10) [9]:

- The transaction logs
- The data changes

At a savepoint the changed data are stored in storage. When the system starts-up, the logs can be processed from the last saved savepoint. The savepoints assure the consistency across all

the transactions of the SAP HANA database. The default configuration of triggering the savepoints is an time interval of five minutes [9].

In general, the SAP HANA Platform supports two back-up types:

- Backing up the savepoints in the form of full data back-ups. These can be used to restore a database to a specific point in time.
- Smaller periodic log backups to recover from fatal storage faults to minimize the loss of data.

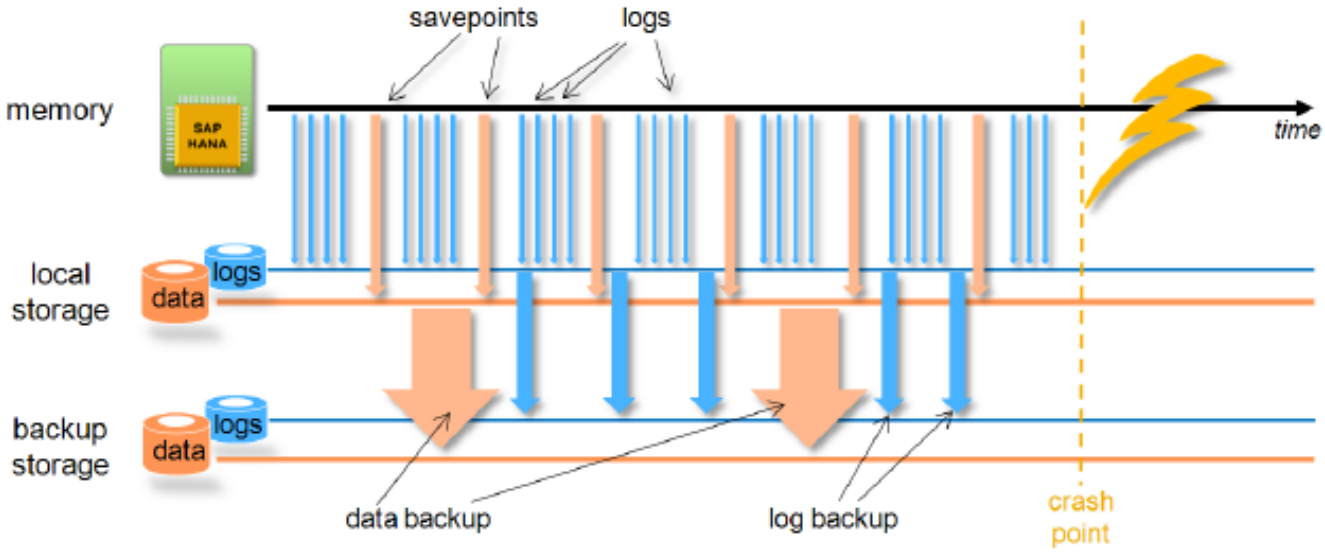

Figure 10: Back-up and Recovery in SAP HANA [9].

The recovery is fulfilled through the data back-ups, the log back-ups and the log area. When the recovery process begins, all the data and log back-ups must be accessible in the file system. Once the data back-up finishes successfully, the entries of the log back-ups are re-attached. In general, for the purposes of recovering the database to a previous point in time, the data back-up and the log back-ups are mandatory to be attached successfully [9].

The aforementioned process of back-up and recovery satisfies the necessity for the high availability of the data in a multi-tenant environment.

Application Services

In previous sections, we have described the database/relational engines of the SAP HANA system and general the main issues regarding the platform. Figure 11 illustrates, in general, the SAP HANA system. The main SAP HANA management component that is shown there is the index server. The index server contains all the components that we described in the previous sections (i.e. relational engines, persistence layer etc.).

The programming language that it permits the index server is the SQL/ or MDX. This requirement is satisfied by the "SQL / MDX processor" in Figure 11.

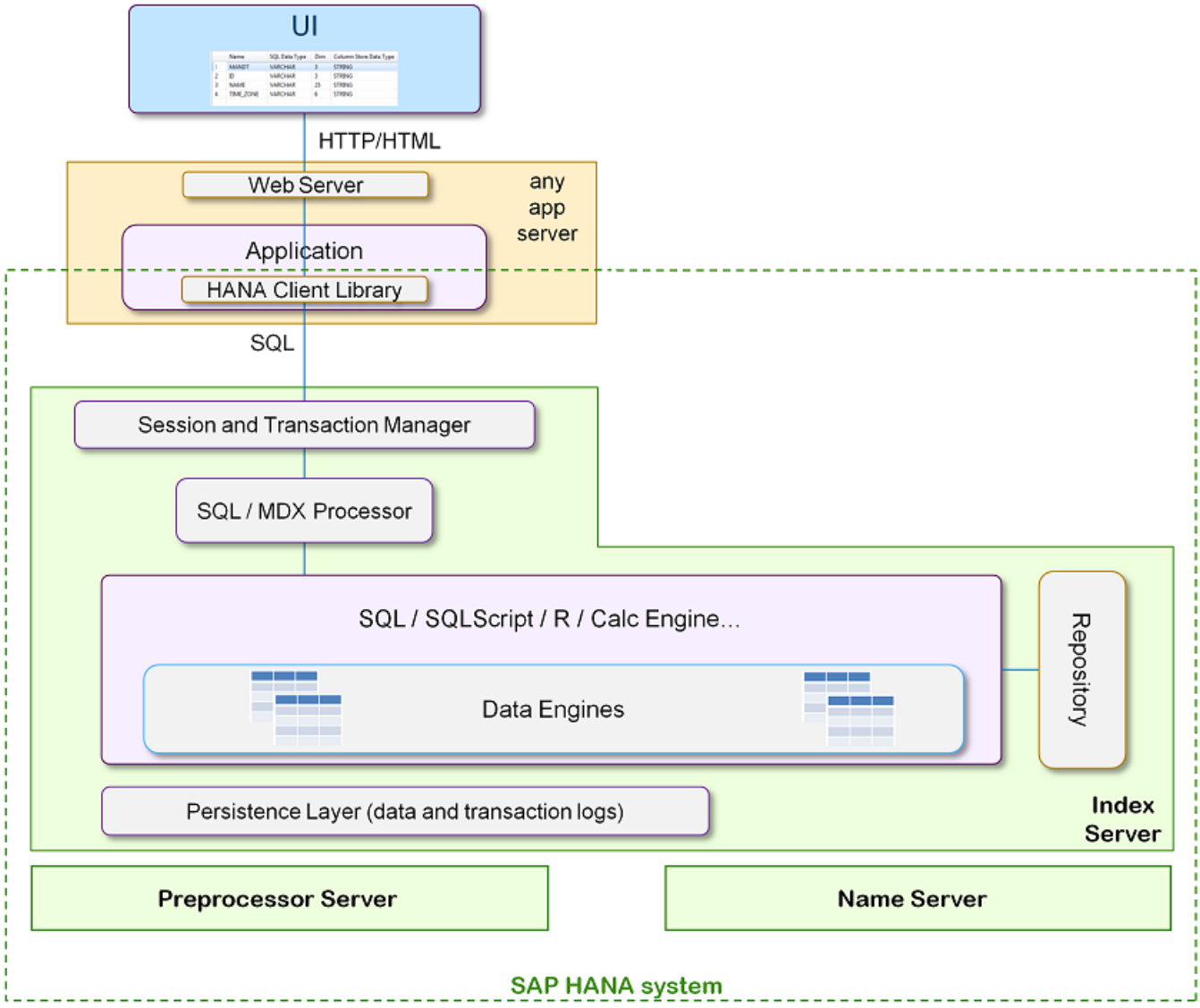

Figure 11: SAP HANA Database - High level architecture [11].

For the purposes of exploiting the aforementioned processor and in order to satisfy the statements written in SQL programming language, the SAP HANA database introduces the scripting language named SQLScript [11].

As we already know, there is the typical problem of the classical applications which tend to keep the calculation logic far away from the database. Thus, it is needed to request and receive big amount of data from the database. The programs tends to slowly iterate over these amount of data and thus, they are hard to optimize and parallelize. The SQLScript integrates the data-intensive application logic into the database. In addition, it is based on side-effect free functions that performed on tables using SQL queried for set processing. Therefore, the SQLScript is assumed parallelizable over multiple processors [11].

Moreover, as we will see in the following sections, the index server supports the R programming language for its calculation logic.

3.2.2 SAP HANA Extended Application Services (SAP HANA XS)

The SAP HANA Platform offers a framework for web application developing and exploiting the database model. The framework is based up on the Model-View-Controller (MVC) software architecture paradigm. As we can see in Figure 12 the "Model" is represented by the database, the "Controller" is the main business logic of an application and the "View" is the front-end, the User Interface, of the application.

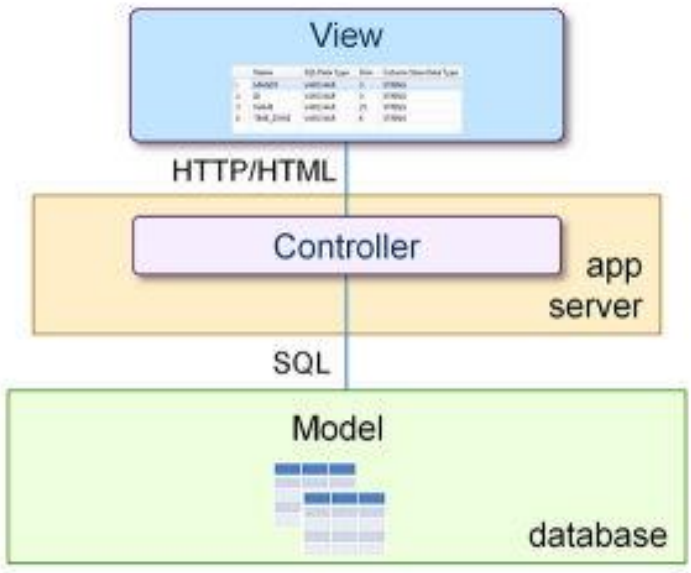

Figure 12: Model-View-Controller(MVC) Architecture [11].

The SAP HANA Platform extends the traditional role of the database. In particular, the platform is used as a development and execution environment for native data-intensive applications. As the Figure 13 illustrates the "Controller" and the "Model" are entirely accommodated on SAP HANA Platform. By this way, the developer can exploit, for his application logic, the parallelization capabilities and the in-memory technology of the SAP HANA that we described in previous sections [11]. The logic of the application comes closer to the data model. This fact results to the performance gain due to this integration with the data source [11].

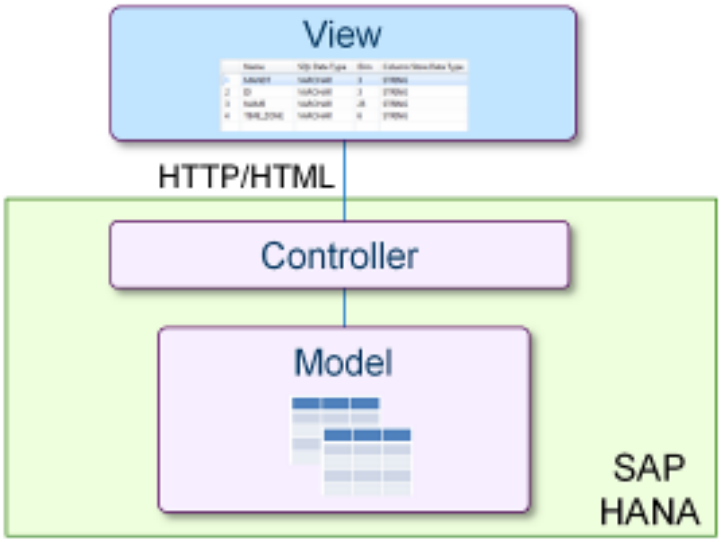

Figure 13: MVC on SAP HANA Platform [11].

The SAP HANA Platform introduces the above advantages through the SAP HANA Extended Application Services which provide a comprehensive set of embedded services for the end-to-end support for Web applications. In particular, the platform offers a lightweight web server, OData support, a server-side Javascript execution environment and, as we have seen in the previous section (i.e. section 3.2.1) an environment for the SQL and SQLScript languages.

The SAP HANA Extended Application Services are provided from the SAP HANA XS server (see Figure 5). The XS server is accessed by the clients through the HTTP protocol. As we can understand, by using the SAP HANA Platform the developer doesn't need an application server to host his logic of the application.

The tables, views, procedures of the SAP HANA database can be exposed by the SAP HANA XS server either through the OData service or by building native application-specific code that runs

on the SAP HANA XS. In the KGV tool, we used both of the offered capabilities that SAP HANA Platform provides.

In a native application-specific code scenario, we can assume a thin layer between the clients on one side and the database objects (i.e. views, tables and procedures) in the index server on the other side [11]. Amongst the others, the important point in this approach is that the communication between the SAP HANA XS server and the index server is optimized for high performance. In addition, the integration of SAP HANA XS server into the entire SAP HANA Platform in the form of application server, it simplifies the administration and offers a better development environment in the whole [10],[11].

Generally, the SAP HANA Platform considers two types of applications [11]:

- Native Applications
- Non-Native Applications

As we have, already, mentioned the native applications and ,especially, the logic of them, run on the SAP HANA Platform. In Figure 14, we can see a representation of how native applications should be considered and designed.

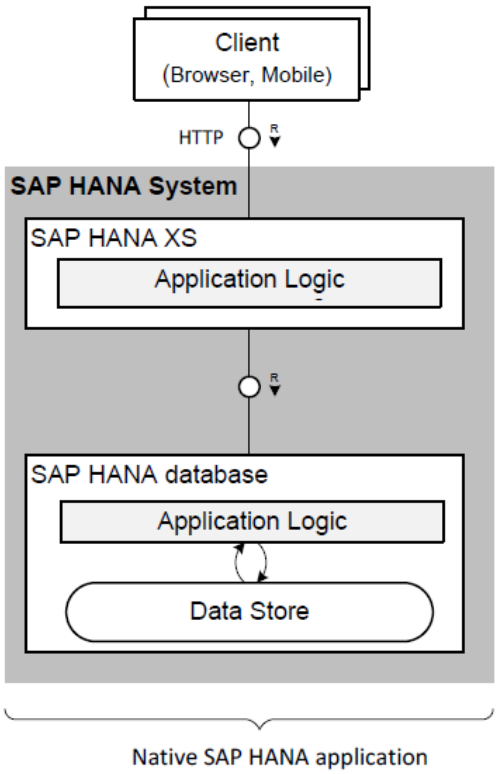

Figure 14: Native Application development [11].

On the other hand, the SAP HANA Platform gives the capability to apply the traditional MVC architecture. In particular, the non-native applications are built in a different external environment (i.e. Java). These applications are connected to the SAP HANA by applying typical communication interfaces, such as the JDBC, ODBC, ADBC or ODBO (Figure 15).

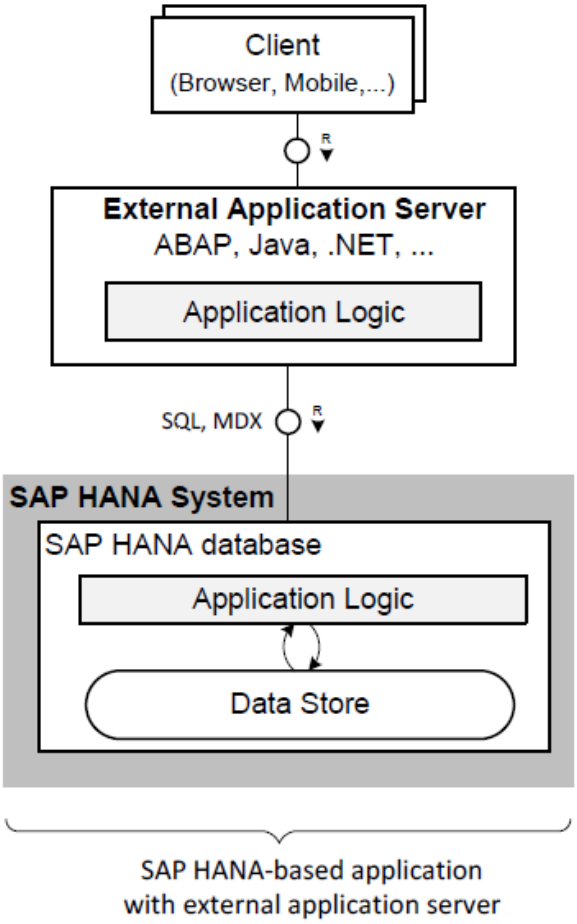

Figure 15: Non-native Application Development [3].

3.2.3 SAP HANA Integration with R

In the previous section, we observed in Figure 5 and Figure 11 that the SAP HANA interacts with R programming language. In particular, it enables the embedding of R code in the SAP HANA database context. Thus, it allows the R code to be processed as part of the overall query execution plan. As we can understand, a developer can integrates into his design the R programming language environment [12].

For the purposes of the R integration fulfillment, the SAP HANA offers a data exchange mechanism. This mechanism supports the transfer of database tables into the vector-oriented

data structures of the R language. If we compare this mechanism with the standard SQL interfaces, we observe that we gain in performance. This happens because the standard SQL interfaces are tuple-based and , so, they demand an additional data copy on the R side [12].

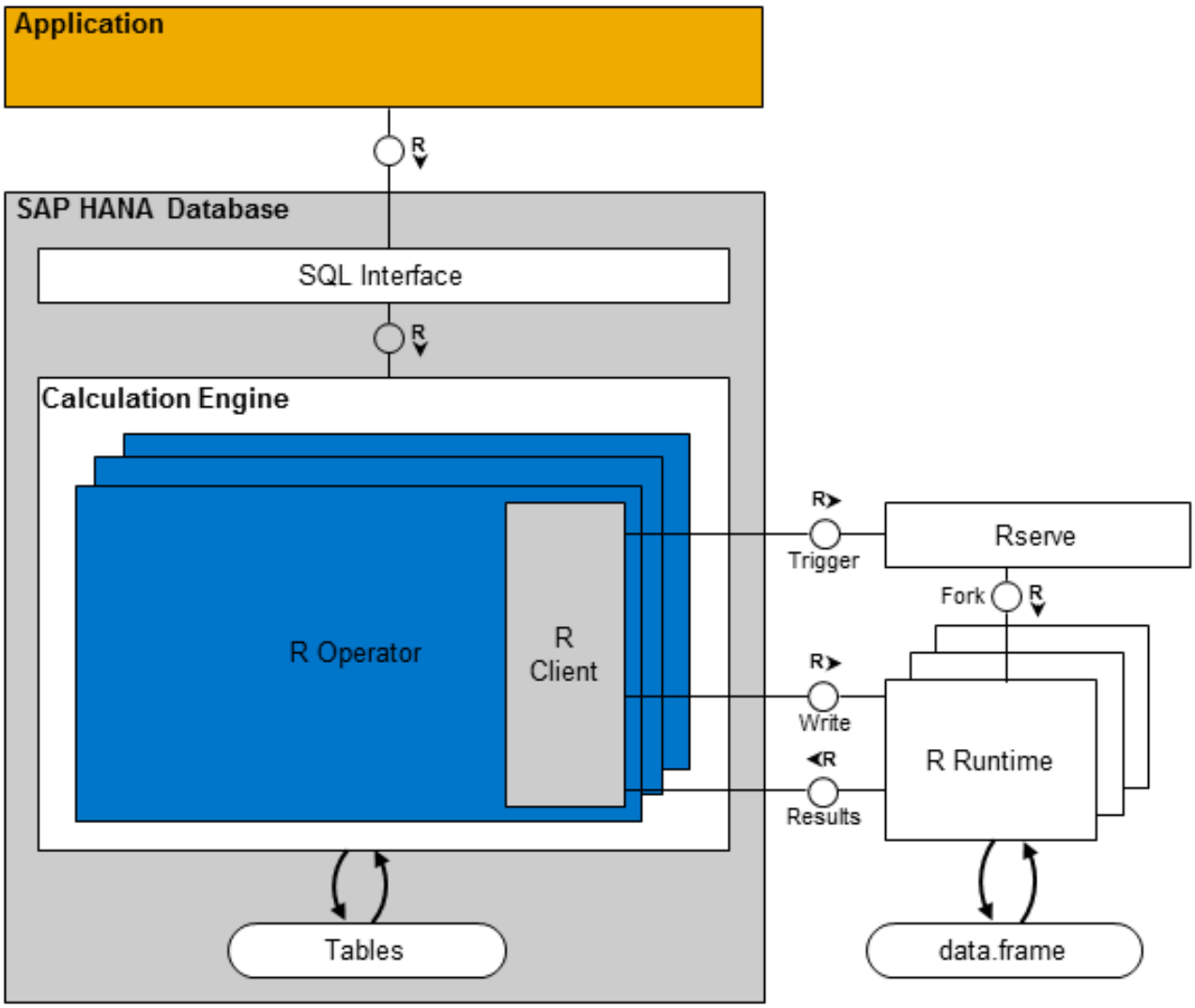

Figure 16: SAP HANA integration with R [12].

Essentially, the SAP HANA Database provides to developers two types of procedures:

- SQLScript procedures
- RLANG procedures.

The type of the SQLScript procedure is the typical procedure in which the developer can write code in SQL. On the other hand, the type of the RLANG procedure is provided to the developer

by the SAP HANA Platform in order to embed R code. The SAP HANA database interacts with the external R environment for the execution of the R code. As we can understand, there is an elegant way for the development in R and calling R functions through SQLScript procedures. For the purposes of this kind of integration, the calculation engine of SAP HANA was extended [12].

Essentially, the calculation engine supports data flow graphs which describe the logical database execution plans. The data flow graphs contain native database operators. One of the operators that the SAP HANA can support, after its extension, is the R operator. The R operator can use a number of input objects, such as row or column tables from the row-based or column-based stored, correspondingly, other data sources etc. In the end, the R operator returns a result table. So, we can assume that the R function code is passed as a string argument to the operator [12].

As we can observe in Figure 16, we assume three entities the application, the SAP HANA database and the R environment hosted on a separate host. So, the scenario is the following [12]:

1. The calculation model triggers an R operator,
2. Then, the R Client creates a request to the Rserve mechanism of the R host in order to initialize a session for the R process.
3. Afterwards, the R Client transfers the R function code and the dedicated database tables that the RLANG procedure has as input parameters to the R host. After that, the R Client triggers for execution.
4. When the R environment finishes the execution of the R function, it returns to the calculation engine an R "data.frame" with the results. The "data.frame" is a matrix-like structure that the R programming language supports.
5. Once the calculation engine receives the "data.frame", it converts it to meaningful format. In particular, it converts it to the column-oriented structure that the SAP HANA database supports. Essentially, the "data.frame" and the column-oriented structures are very similar, thus, such a conversion is managed efficiently by the calculation engine.

In the above process, we exploit the fact that the SAP HANA database offers parallelization in these processes, so, we can assume parallel R processes (thus, RLANG procedures) execution.

4 Design and Development of the Knowledge Generation and Visualization (KGV) tool

4.1 Introduction

The Knowledge Generation and Visualization (KGV) tool is designed as a re-usable and cross-domain tool, i.e., expanding in multiple areas, for building knowledge on the past experience of an eco-system, visualizing it and predicting future states of the eco-system. However, for prototyping purposes, the application that is targeted comes from the telecommunication area. Therefore, the first application to be created refers to building knowledge, visualizing it and predicting the future load of a network based on past monitoring data of the network and other information coming from the Big Data pool, e.g., calendar and environmental information.

In the next sections, we present the design and the development techniques that we used in order to realize the tool in the context of the SAP HANA Platform. We present the different parts that consists the KGV tool. As the Figure 17 depicts, the parts of the tool are the following:

- **the User-Interface (UI):** This part is related with the front-end of the tool which a user could exploit in order to proceed to a number of tasks, such as the creation of a use-case, the load of a use-case, the upload of a (validation) dataset, the training process, the prediction process, the validation process and the visualization of the results.
- **the Data model (SAP HANA Platform):** This part offers the data model that we have designed for the accommodation and management of the data. The SAP HANA Platform provides an environment for accelerated calculations and executions into the database context which is important in the Big Data era. Also, it offers an environment for developing software modules and applications near the fast database in order to bring the calculations as near as possible to the data.
- **the Engine / the algorithm (R Host):** This part is responsible for the main calculation tasks of the algorithm, Parameterless Growing Self-Organizing Maps. In addition, this part provides the tools for the prediction and the validation processes and, generally, for the main tasks which are related with the use-cases.

4.2 Design of tool

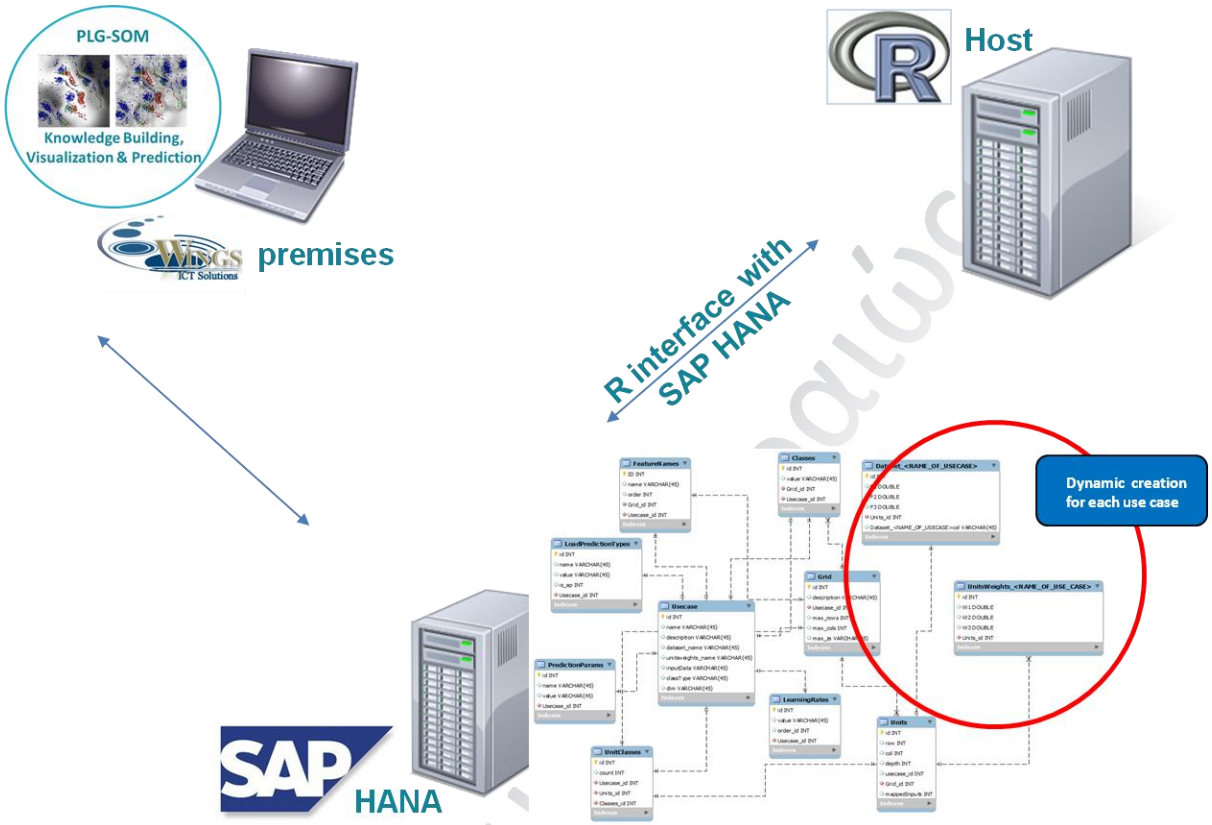

Figure 17: General architecture

The tool was designed considering a multi-tenant architecture in mind. This means that multiple tenants can use, extend and configure the application regarding their own needs. The data model that the tool uses should support features such as, extensibility and high availability. The SAP HANA Platform supports these features, amongst the others.

4.2.1 SAP HANA Database design

Figure 17 depicts the architecture that the Knowledge Generation and Visualization tool is built up on. The tool uses various technologies and programming languages, such as, Java, R, C and XSJS, in order to fully exploit the powerful engine of SAP HANA Platform and accelerate the computations.

At first, we have created the database schema on which each tenant could create and apply each own use-case and proceed at training and prediction processes. Figure 18 demonstrates the database schema that SAP HANA Platform accommodates. In the following, we explain the usage of each table.

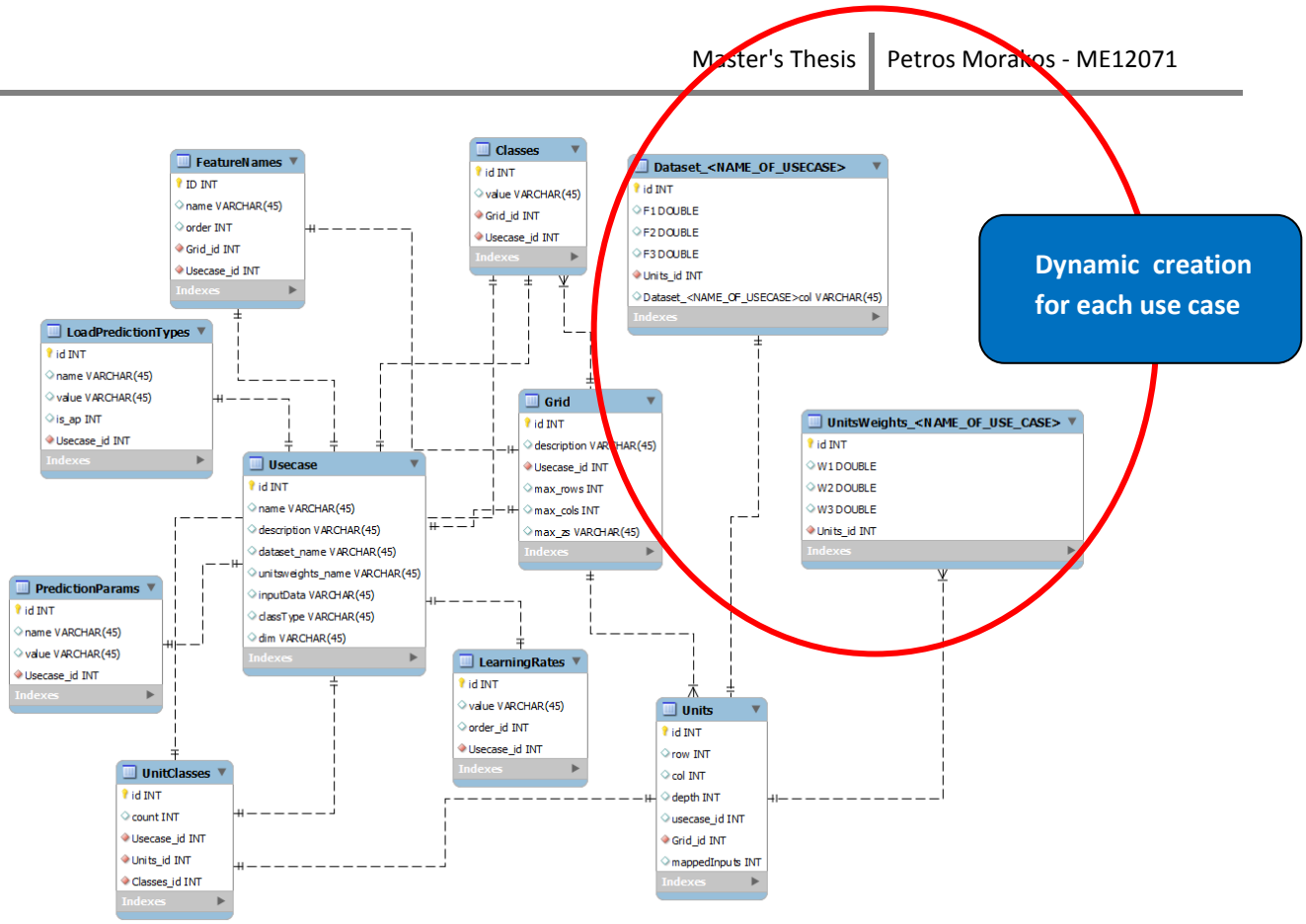

Figure 18: Database schema of the KGV tool.

Dynamic creation of Tables

When a tenant creates a use-case in the KGV tool, he creates, also, automatically the following tables:

- **A table 'Dataset_<NAME_OF_USECASE>':** This table holds the dataset of a usecase. It corresponds all the features as column names (F1, F2, F3,....) and the related values for each vector as rows. In addition, the table has a column 'Unit_id' which keeps an ID for the SOM cell that the vector belongs.
- **A table 'UnitsWeights_<NAME_OF_USECASE>':** This table holds the weights of the units for a specific use case. So, columns (W1, W2, W3,....) will be as many as the number of features in the dataset of the usecase. In addition, there is a column 'Unit_id' which similarly to the 'Dataset_<NAME_OF_USECASE>' table holds the ID of the SOM cell. This column declares to which unit belongs the row of this table.
- **A table 'Validation_<NAME_OF_USECASE>':** This table holds the validation dataset of a usecase. It corresponds all the features as column names (F1, F2, F3,....) and the related values for each vector as rows. In addition, it consists of the column 'Estimation' which holds the prediction of each vector. Also, the table has a column 'Unit_id' which connects a mapped input to the specified unit.

Dynamic creation of Attribute Views

Moreover, when a tenant creates a use-case, he creates the following Attribute Views:

- **An Attribute View 'AT_DU_< NAME_OF_USECASE >':** This view provides the join operation between the table 'Dataset_<NAME_OF_USECASE' and the table 'Units' (e.g. Dataset_<NAME_OF_USECASE>.Unit_id = Units.id). This view will provide us the vectors that belong to the cell of the SOM grid.
- **An Attribute View 'AT_WU_< NAME_OF_USECASE >':** This view provides the join operation between table 'UnitsWeights_<NAME_OF_USECASE>' and table 'Units' (e.g. UnitsWeights_<NAME_OF_USECASE>.Unit_id = Units.id). This view will provide us which weight vector belongs to which unit.

Standard tables

The database schema keeps standard tables and SAP HANA views which are used by all tenants:

- **Usecase (id, name, description, dataset_name, unitsweights_name, inputData, classType, dim, currentQuality, currentCycle, currentIteration, aggrLearningRate, targetQuality, distanceMetric, errorMetric, cycleDuration, neighborhoodSize, gridInitializator, trainingMode, qualityMethod, maxTrainithough, maxMapArea, dataSelectorType, stopOnQDecrease, existTables, net_traffic)**
  - **id (INT):** Use case identifier.
  - **name (VARCHAR):** Name of the use case.
  - **description (VARCHAR):** Short description of use case.
  - **dataset_name(VARCHAR):** This is a meta-data column that holds the name of the table where the vectors/dataset can be found.
  - **unitsweights_name(VARCHAR):** This is a meta-data column that holds the name of the table where the weights of grid can be found.
  - **inputData(VARCHAR):** This columns identifies if the usecase holds 'real' or 'class' input data.
  - **classType(VARCHAR):** This field holds the label of the dataset.
  - **dim(VARCHAR):** This field holds how many features appear in the dataset.
  - **currentQuality(VARCHAR):** This field holds the current quality of the grid which is generated by the training process.
  - **currentCycle(VARCHAR):** This field holds the current cycle of the training process.
  - **currentIteration(VARCHAR):** This field holds the current iteration of the training process.
  - **aggrLearningRate(VARCHAR):** This field holds the total aggregated learning rates.
  - **targetQuality(VARCHAR):** This field holds the target quality of the training process.

- **distanceMetric(VARCHAR):** This field holds the distance metric that is used by the training and the prediction processes.
- **errorMetric(VARCHAR):** This field holds the error metric that the training process use.
- **cycleDuration(VARCHAR):** This field holds the cycle duration of the training process.
- **neighborhoodSize(VARCHAR):** This field holds the neighborhood size that the training process use.
- **gridInitializator(VARCHAR):** This field holds the grid initializator that the training process use.
- **trainingMode(VARCHAR):** This field holds the training mode that the training process use. Values = {Supervised, Unsupervised}
- **qualityMethod(VARCHAR):** This field holds the quality method that the training process use.Values = {Mean, Minimmum}
- **maxTrainingTime(VARCHAR):** This field holds the max training time that the training process will long. This is optional.
- **maxMapArea(VARCHAR):** This field holds the max map are that the grid will reach on the training process. This is optional.
- **dataSelectorType(VARCHAR):** This field holds the data selector type that the training process use.
- **stopOnQDecrease (VARCHAR):** This field specifies if the stop of the training process is related with the quality of the grid.
- **existTables (INT):** This field specifies if the "Dataset" and "UnitWeights" column Tables has already created for the usecase.
- **net_traffic (VARCHAR):** This field specifies if the usecase corresponds to a network traffic usecase or not.

➢ **Grid (id,description, usecase_id, max_rows, max_cols, max_zs)**

- **id (INT):** Identifier of the SOM grid
- **description (VARCHAR):** Short description of the SOM grid
- **usecase_id (INT):** ID of the use case to which this SOM grid belongs
- **max_rows (INT):** The rows of the SOM grid. This is modified during the training of the PLGSOM and the KGV knowledge building.
- **max_cols(INT):** The columns of the grid. This is modified during the training of the PLGSOM and the KGV knowledge building.
- **max_zs(INT):** This field is for future use.

➢ **Units (id, row, col, depth, usecase_id, grid_id, mappedInputs)**

- **id (INT):** Identifier of the SOM cell
- **row (INT):** index of the row

- **col (INT):** Index of the column
- **depth(INT):** This is for future use
- **usecase_id (INT):** The id of the corresponded usecase.
- **grid_id (INT):** ID of the SOM grid to which the SOM cell belongs.
- **mappedInputs (INT):** This specifies the total amount of inputs which are corresponded to the unit.

- **Classes(id ,value, grid_id, usecase_id)**
  - **id(INT):** Identifier
  - **value(VARCHAR):** Value of the class.
  - **grid_id(INT):** ID of the SOM grid to which the class belongs.
  - **usecase_id(INT):** The id of the corresponded usecase.

- **FeatureNames(id, name, usecase_id, grid_id, order)**
  - **id(INT):** Identifier
  - **name(VARCHAR):** Name of the feature.
  - **usecase_id(INT):** The id of the corresponded usecase.
  - **grid_id(INT):** This is the id of the corresponded grid.
  - **order(INT):** This is used for order purposes.

- **LearningRates(id,value,usecase_id,order_id)**
  - **id(INT):** Identifier
  - **value(VARCHAR):** Value of the learning rate.
  - **usecase_id(INT):** The id of the corresponded usecase.
  - **order_id(INT):** This is used for order purposes - an 'order_id' corresponds to the number of iteration.
- **MeanLearningRates(id,value,usecase_id,order_id)**
  - **id(INT):** Identifier
  - **value(VARCHAR):** Value of the mean learning rate.
  - **usecase_id(INT):** The id of the corresponded usecase.
  - **order_id(INT):** This is used for order purposes - an 'order_id' corresponds to the number of epoch.

- **LoadPredictionTypes(id,name,value,usecase_id,is_ap)**
  - **id(INT):** Identifier
  - **name(VARCHAR):** This is the name of feature which takes part in the prediction process.
  - **value(VARCHAR):** This is the value of the corresponded feature.

- **usecase_id(INTEGER):** The id of the corresponded usecase.
- **is_ap(INT):** This is an identifier which identifies if the feature is an Access Point or not. Possible values = {0,1}

- **PredictionParams(id,name,value,usecase_id)**
  - **id(INT):** Identifier
  - **name(VARCHAR):** This is the name of the parameter which takes part in the prediction process.
  - **value(VARCHAR):** This is the value of the corresponded parameter.
  - **usecase_id(INT):** The id of the corresponded usecase.

- **UnitClasses(id,unit_id,classes_id,count,usecase_id)**
  - **id(INT):** Identifier
  - **unit_id(INT):** The ID of the unit.
  - **classes_id(INT):** The ID of the class.
  - **count(INT):** This field holds how many times appear the specified class into the specified unit.
  - **usecase_id(INT):** The id of the corresponded usecase.
- **UnitInfo(id, mappedValuesAverage, stdDev, absDev, mqe, inputs, unit_id, usecase_id)**
  - **id(INT):** Identifier
  - **mappedValuesAverage(VARCHAR):** This is the average (label) value of mapped inputs for a unit.
  - **stdDev(VARCHAR):** This is the standard deviation value of mapped inputs for a unit.
  - **absDev(VARCHAR):** This is the absolute deviation value of mapped inputs for a unit.
  - **mqe(VARCHAR):** This is the mean quantization error of mapped inputs for a unit.
  - **inputs(INT):** This is the number of inputs which are mapped upon a unit.
  - **unit_id(INT):** The ID of the unit.
  - **usecase_id(INT):** The id of the corresponded usecase.
- **DatasetInfo(id, name, value, usecase_id, order_id)**
  - **id(INT):** Identifier
  - **name(VARCHAR):** This is the name of the dataset parameter
  - **value(VARCHAR):** This is the value of the corresponded dataset parameter.
  - **usecase_id(INT):** The id of the corresponded usecase.
  - **order_id(INT):** This is used for order purposes.

Standard Procedures

The database schema design includes some procedures which are written either on SQLScript or on R language. The SAP HANA supports procedures written on R language by exploiting a special interface which connects the SAP HANA database with a host which accommodates an R server. In this context, the SAP HANA Platform transforms the potential input tables into an R data structure (i.e.' data.frame') and sends the code to R server for execution. Finally, the R server responds with a 'data.frame' to SAP HANA Platform which transforms it to a related SQL column table. In the following, we describe the standard procedures that a use-case will have:

- **INSERT_UNITS:** This procedure is a standard SQLScript procedure which insert new Units into the the 'UNITS' column table.
- **GET_UNITS:** This procedures is a standard RLANG procedure. It calls an R function in order to retrieve the units of SOM map created by the last training process.
- **GET_LEARNINGRATES:** This procedure is a RLANG procedure. It calls an R function in order to retrieve the learning rates matrix of the last training process.
- **INSERT_LEARNINGRATES:** This procedure is a standard SQLScript procedure which inserts the new learning rates of the last training process into the "LEARNINGRATES" Column Table.
- **GET_MEANLEARNINGRATES:** This procedure is a standard RLANG procedure. It calls an R function in order to retrieve the mean learning rates matrix created by the last training process.
- **INSERT_MEANLEARNINGRATES:** This procedure is a standard SQLScript procedure. It inserts the mean learning rates, which was retrieved by the procedure 'GET_MEANLEARNINGRATES', into the column table 'MEANLEARNINGRATES'.
- **GET_GRID:** This procedure is a RLANG procedure. It calls an R function in order to retrieve general information of the grid of the last training process.
- **INSERT_GRID:** This procedure is a standard SQLScript procedure which inserts general information of the grid of the last training process into the "GRID" Column Table.
- **GET_TRAINPARAMS:** This procedure is a RLANG procedure. It calls an R function in order to retrieve general information about the last training process of a specific usecase.
- **INSERT_TRAINPARAMS:** This procedure is a standard SQLScript procedure which inserts general information about the last training process of a specific usecase into the "USECASE" Column Table.
- **GET_UNITCLASSES:** This procedure is a RLANG procedure. It calls an R function in order to retrieve information about the amount of vectors that corresponded to each unit, categorized by each class. This kind of information is generated after the last training process of a usecase.

- **INSERT_UNITCLASSES:** This procedure is a standard SQLScript procedure which inserts the information that was retrieved by the " GET_UNITCLASSES " into the "MAPPEDINPUTS" Column Table.
- **CREATE_MODEL_FROM_TEMPLATES:** This procedure is a SQLScript procedure. It calls the procedure 'CREATE_MODEL_FROM_TEMPLATES_PROCESS' in order to create the appropriate column tables, SAP HANA views, procedures and types for a new use-case.
- **CREATE_MODEL_FROM_TEMPLATES_PROCESS:** This procedure is a RLANG procedure. It calls an R function in order to create dynamically the appropriate column tables, SAP HANA views, procedures and types. In particular, the R function calls a Java process which resides on the same host and performs the above action.
- **CLEAR_DB:** This is a SQLScript procedure. It deletes the content of column tables 'CLASSES', 'FEATURENAMES', 'GRID', 'LEARNINGRATES', 'MEANLEARNINGRATES', 'UNITCLASSES', 'UNITS', 'USECASE', 'PREDICTIONPARAMS', 'LOADPREDICTIONTYPES', 'UNITINFO'.
- **CLEAR_DB_PER_USECASE:** This is a SQLScript procedure. It deletes the related content of a a specified use-case from column tables: 'CLASSES', 'FEATURENAMES', 'GRID', 'LEARNINGRATES', 'MEANLEARNINGRATES', 'UNITCLASSES', 'UNITS', 'USECASE', 'PREDICTIONPARAMS', 'LOADPREDICTIONTYPES', 'UNITINFO'.
- **INSERT_ONE_UNITINFO:** This is a SQLScript procedure. It inserts to column table 'UnitInfo' a row with info (i.e. id, mappedValuesAverage, stdDev, absDev, mqe, inputs, unit_id, usecase_id) about a unit of a SOM map.
- **INSERT_DATASETINFO:** This is a SQLScript procedure. It inserts the dataset info related to a specified use-case, which was retrieved by the procedure '<NAME_OF_USECASE>_CALC_DATASET_INFO', into the column table 'DATASETINFO'.

Dynamic creation of procedures

Similarly with dynamic creation of column tables and SAP HANA views, the KGV tool creates dynamically some procedures when a use-case is created by a tenant. In the following, we describe these procedures:

- **< NAME_OF_USECASE >_TRAIN:** This procedure is a standard SQLScript procedure. This procedure performs the training process. In particular it calls a module that runs on the R host and performs the training process. Afterwards, it calls the following procedures in order to store into Hana DB all the appropriate statistics and information of the training process that was performed by the R module: '<NAME_OF_USECASE>_TRAINING_PROCESS', 'GET_UNITS', 'INSERT_UNITS', '<NAME_OF_USECASE>_GET_WEIGHTS', '<NAME_OF_USECASE>_INSERT_WEIGHTS', '<NAME_OF_USECASE>_GET_DATASET', '<NAME_OF_USECASE>_INSERT_DATASET',

'GET_LEARNINGRATES', 'INSERT_LEARNINGRATES', 'GET_MEANLEARNINGRATES', 'INSERT_MEANLEARNINGRATES', 'GET_GRID', 'INSERT_GRID', 'GET_TRAINPARAMS', 'INSERT_TRAINPARAMS'

- **< NAME_OF_USECASE >_TRAINING_PROCESS:** This procedure is a RLANG procedure. It is the core procedure on the Hana DB side which calls and initializes the R module on R host in order to perform the training process.
- **< NAME_OF_USECASE >_GET_WEIGHTS:** This procedure is a RLANG procedure. It calls an R function and retrieves the weights of the grid that the last training process of a usecase has already created.
- **< NAME_OF_USECASE >_INSERT_WEIGHTS.** This procedure is a standard SQLScript procedure. This procedure is called after the '**< NAME_OF_USECASE >_GET_WEIGHTS**' procedure and it inserts the weights at the related **'UnitsWeights_<NAME_OF_USECASE>'** Column Table.
- **< NAME_OF_USECASE >_GET_DATASET:** This procedure is a RLANG procedure. It calls an R function and retrieves the dataset of the usecase that the last training process of a usecase has already modified. In particular, the training process modifies the 'UNIT_ID' column.
- **< NAME_OF_USECASE >_INSERT_DATASET:** procedure is a standard SQLScript procedure. This procedure is called after the '**< NAME_OF_USECASE >_GET_DATASET**' procedure and it inserts the dataset at the related **'Dataset_<NAME_OF_USECASE>'** Column Table.
- **< NAME_OF_USECASE >_LOAD_PREDICTION:** This procedure is a standard SQLScript procedure. This procedure retrieves all the appropriate information from the Column Tables in order to initialize the network load prediction process.
- **< NAME_OF_USECASE >_LOADPREDICTION_PROCESS:** This procedure is a RLANG procedure. It initializes the load prediction process which resides on the R module on the R host.
- **< NAME_OF_USECASE >_VALIDATION:** This procedure is a SQLScript procedure. It calls the '<NAME_OF_USECASE>_VALIDATION_PROCESS' procedure in order to perform the validation process for a specified usecase. Finally, it returns the results of the validation process.
- **< NAME_OF_USECASE >_VALIDATION_PROCESS:** This procedure is a RLANG procedure. It calls an R function in order to perform the validation process.
- **< NAME_OF_USECASE >_CALC_UNITINFO:** This procedure is a SQLScript procedure. It iterates through all units of a SOM map. At each unit, it calls the procedure '<NAME_OF_USECASE>_CREATE_UNITINFO' for the calculation of unit info (i.e. standard deviation, mean quantization error etc.) and the procedure 'INSERT_ONE_UNITINFO' in order to insert the corresponding unit info into 'UNITINFO' column table.

- **< NAME_OF_USECASE >_CREATE_UNITINFO:** This procedure is a RLANG procedure. It calls a R function in order to estimate unit information for a unit, such as standard deviation, mean quantization error, average value, mean average deviation and the number of mapped inputs.
- **< NAME_OF_USECASE >_COMPONENT_LINE_PLOT_MAP:** This is a SQLScript procedure. It calls the '<NAME_OF_USECASE>_COMPONENT_LINE_PLOT_MAP_PROCESS' procedure in order to produce results for the graph diagram "parameter VS label", where 'parameter' is a given parameter taken from the features' set and 'label' is the estimated label of the dataset.
- **< NAME_OF_USECASE >_COMPONENT_LINE_PLOT_MAP_PROCESS:** This is a RLANG procedure. It calls a R function in order to to produce results for the graph diagram "parameter VS label", where 'parameter' is a given parameter taken from the features' set and 'label' is the estimated label of the dataset. It returns results to '<NAME_OF_USECASE>_COMPONENT_LINE_PLOT_MAP' procedure.
- **< NAME_OF_USECASE >_CALC_DATASET_INFO:** This is a SQLScript procedure. It calls the '<NAME_OF_USECASE>_CALC_DATASET_INFO_PROCESS' for the estimation of statistics about the whole dataset of a given use-case, such as minimum values, maximum values, mean values, quantization error etc. When the '<NAME_OF_USECASE>_CALC_DATASET_INFO_PROCESS' procedure returns the results, it calls the 'INSERT_DATASETINFO' procedure to insert them into the 'DATASETINFO' column table.
- **< NAME_OF_USECASE >_CALC_DATASET_INFO_PROCESS:** This is a RLANG procedure. It calls a R function for the estimation of statistics about the whole dataset of a given use-case, such as minimum values, maximum values, mean values, quantization error etc. Finally, it returns the results to '<NAME_OF_USECASE>_CALC_DATASET_INFO'.

Standard table types

The database schema keeps standard table types which are used by all tenants. These types are commonly used in procedures. In the following, we provide the description of the standard table types:

- **T_UNITS (id,row,col depth, usecase_id, grid_id, mappedinputs)**: This is a table type which is corresponded to column table 'UNITS' and it is used either as input type or output type in procedures '<NAME_OF_USECASE>_TRAINING_PROCESS', 'GET_UNITS', and 'INSERT_UNITS'.
- **T_GRID (id, description, maxrows, maxcols, maxzs, usecase_id)**: This is a table type which is corresponded to column table 'GRID' and it is used either as input type or output type in procedures 'GET_GRID' and 'INSERT_GRID'

- **T_LEARNINGRATES (id, value, usecase_id)**: This is a table type which is corresponded to column table 'LEARNINGRATES' and it is used either as input type or output type in procedures 'GET_LEARNINGRATES' and 'INSERT_LEARNINGRATES'.
- **T_MEANLEARNINGRATES (id, value, usecase_id, order_id)** : This is a table type which is corresponded to column table 'MEANLEARNINGRATES' and it is used either as input type or output type in procedures 'GET_MEANLEARNINGRATES' and 'INSERT_MEANLEARNINGRATES'.
- **T_UNITCLASSES (id, unit_id, classes_id, count, usecase_id)** : This is a table type which is corresponded to column table 'UNITCLASSES' and it is used either as input type or output type in procedures 'GET_UNITCLASSES' and 'INSERT_UNITCLASSES'.
- **T_USECASE (id, name, description, dataset_name, unitsweights_name, inputdata, classtype, dim, currentquality, currentcycle, currentiteration, aggrlearningrate, targetquality, distancemetric, errormetric, cycleduration, neighborhoodsize, gridinitialization, trainingmode, qualitymethod, maxtrainingtime, maxmaparea, dataselectortype, stoponqdecrease, existtables, nettraffic)** : This is a table type which is corresponded to column table 'USECASE' and it is used either as input type or output type in procedures 'GET_TRAINPARAMS' and 'INSERT_TRAINPARAMS'.
- **T_RESULTPREDICTION (id, name, value, usecase_id):** This is a table type which is used either as input type or output type in procedures 'GET_RESULTPREDICTION' and 'GET_RESULTPREDICTION_PROCESS'.
- **T_STARTID (startid):** This is a table type which is used in order to pass integer parameters into , basically, RLANG procedures. It is used either as input type or output type in procedures '<NAME_OF_USECASE>_TRAINING_PROCESS', '<NAME_OF_USECASE>_CREATE_UNITINFO', '<NAME_OF_USECASE>_GET_UNITINFO_PROCESS' and '<NAME_OF_USECASE>_CALC_DATASET_INFO_PROCESS'.
- **T_CLP_MAP_ONLY (component, estimation) :** This is a table type which is used either as input type or output type in procedures '<NAME_OF_USECASE>_COMPONENT_LINE_PLOT_MAP' and '<NAME_OF_USECASE>_ COMPONENT_LINE_PLOT_MAP_PROCESS'.
- **T_DATASETINFO (id, name, value, usecase_id, order_id):** This is a table type which is corresponded to column table 'DATASETINFO' and it is used either as input type or output type in procedures '<NAME_OF_USECASE>_CALC_DATASET_INFO_PROCESS' and 'INSERT_DATASETINFO'.

Dynamic creation of table types

When a tenant creates a use-case in the KGV tool, he creates, also, dynamically the following table types which are used by the dynamically created procedures:

- **T_<NAME_OF_USECASE>_DATASET:** This is a table type which is corresponded to column table 'DATASET _<NAME_OF_USECASE>'. It is used either as input type or output type in procedures '<NAME_OF_USECASE>_GET_DATASET', '<NAME_OF_USECASE>_INSERT_DATASET', '<NAME_OF_USECASE>_LOAD_PREDICTION' and '<NAME_OF_USECASE>_LOADPREDICTION_PROCESS' .
- **T_<NAME_OF_USECASE>_DU:** This is a table type which is used either as input type or output type in procedures '<NAME_OF_USECASE>_CREATE_UNITINFO' and '<NAME_OF_USECASE>_GET_UNITINFO_PROCESS'.
- **T_<NAME_OF_USECASE>_WEIGHTS:** This is a table type which is corresponded to column table 'UNITSWEIGHTS_<NAME_OF_USECASE>'. It is used either as input type or output type in procedures '<NAME_OF_USECASE>_GET_WEIGHTS' and '<NAME_OF_USECASE>_INSERT_WEIGHTS'.
- **T_<NAME_OF_USECASE>_WU:** This is a table type which is used either as input type or output type in procedures '<NAME_OF_USECASE>_VALIDATION_PROCESS', '<NAME_OF_USECASE>_CREATE_UNITINFO', '<NAME_OF_USECASE>_GET_UNITINFO_PROCESS' and '<NAME_OF_USECASE>_COMPONENT_LINE_PLOT_MAP_PROCESS'.

4.2.2 R server interface with SAP HANA

Figure 17 depicts an interface between the SAP HANA Platform and a R host. As it is, already, understood from the description of the main SAP HANA database schema on section 4.2.1, there is an exclusive communication between the HANA Platform and the R host. In particular, the developed "SAP HANA interface to R" provides to developers the capability to exploit the R language which offers an environment for statistical computing tasks. As a result, we have developed an R module which resides in an R host in the same network with the SAP HANA Platform host. This R module is responsible for the main execution tasks of the KGV tool which are related to the training algorithm of SOM and the prediction techniques, amongst the others. Once a RLANG procedure on SAP HANA Platform instantiates the R module, a function is being called to execute a task.

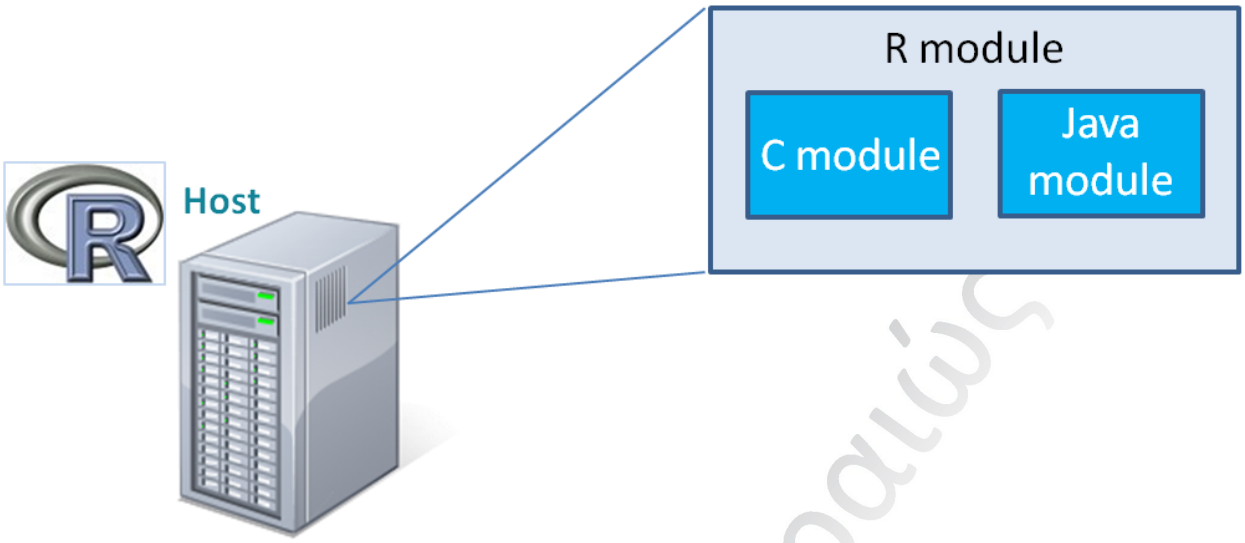

Figure 19: R module of KGV tool

Figure 19 demonstrates a diagram about the R module and its sub-parts, C module and Java module. More specifically, the R module consists of a module written in C which provides a number of functions for the training and prediction tasks of the KGV tool. The R language provides an interface in order to load and execute C compiled code [13], [14]. Essentially, the C part provides the functionality of the training and prediction and the R part offers the functionality of estimating the quality of SOM map while training is in progress. Moreover, the R part provides the growing process of SOM map and it includes some utility functions to satisfy the needs of SOM algorithm and the requests of SAP HANA procedures. The Java part of the module is responsible to create into the SAP HANA all the dynamic column tables, HANA views, procedures and table types. In the following paragraphs, we describe the functions of R module (i.e. and R,C and Java parts).

R module

As we mentioned above, the main functionality of the R module is the estimation of SOM map quality, the growing of the SOM map during the training process and some utility functions. It is called by the HANA procedures written in RLANG and , most of the times, it receives the appropriate data in format of "data.frame" data structure [15].

In addition, the R module, and thus the R code, is like a proxy between the SAP HANA database and the C/Java modules. In particular, the R module receives the data from the SAP HANA Platform, it instantiates the appropriate variables and finally it forwards the data to the responsible module for the main execution. This process happens, for example, on training and

prediction tasks which are implemented by the C module. Subsequently, we describe the core functions of the R module which deploy the aforementioned tasks:

- **train (gridParams, stopCriteriaParams, vecs, trainParams, classesDF, lids, mlrids, uids, cids):** This function manages the main training process of PLGSOM.
  - gridParams(Data.frame): It contains parameters about the grid.
  - stopCriteriaParams(Data.frame): It contains parameters about the grid.
  - ves(Data.frame): It contains the dataset of the training process.
  - trainParams(Data.frame): It contains some initial parameters of the training process.
  - classesDF(Data.frame): It contains the classes of the dataset label if the dataset has type 'Class'.
  - lids (Integer): It contains the initial id of learning rates matrix that the training process will produce.
  - mlrids (Integer): It contains the initial id of mean learning rates matrix that the training process will produce.
  - uids (Integer): It contains the initial id of units matrix that the training process will produce.
  - cids (Integer): It contains the initial id of classes matrix that the training process will produce if the dataset has type 'Class'.
- **quality():** This function initializes the SOM map quality measurement. It takes into account some global variables which are initialized by the train() function.
- **meanQmethod():** This function executes the SOM map quality measurement. It applies the mean technique which means that it sums the errors of all units and it divides the sum-error with the amount of SOM map units. It takes into account some global variables which are initialized by the train() function.
- **errormetric(unitWeights, uVectors):** It initializes the error metric.
  - unitWeights (matrix): The weights of a unit.
  - uVectors (matrix): The dataset of the training process.
- **qe(arrayA, matrixB, demtric, qe0):** This function implements the quantization error.
  - arrayA (Array/vector): It is an 1xM array of weights of the unit.
  - matrixB (matrix): It is a NxM matrix which holds the vectors that corresponds to the current unit.
  - dmetric (string): It takes values of "Euclidean", "SquaredEuclidean" etc.
  - qe0 (double): It is the quantization error of the training dataset.
- **mqe(arrayA, matrixB, demtric, lengthDataset, qe0)**: This function implements the mean quantization error.
  - arrayA (Array/vector): It is an 1xM array of weights of the unit.

- matrixB (matrix): It is a NxM matrix which holds the vectors that corresponds to the current unit.
- dmetric (string): It takes values of "Euclidean", "SquaredEuclidean" etc.
- lengthDataset (integer): It is the amount of dataset vectors.
- qe0 (double): It is the quantization error of the training dataset.

- **clse (arrayA, matrixB, numClasses):** This function implements the classification error metric.
  - arrayA(Array/vector): It is an 1xM array of weights of the unit.
  - matrixB (matrix): It is a NxM matrix which holds the vectors that corresponds to the current unit.
  - numClasses(integer): It is the amount of classes of the label of dataset.
- **mqclse(arrayA, matrixB, dmetric, matrixDataset,numClasses):** This function implements the mean quantization classification error metric.
  - arrayA(Array/vector): It is an 1xM array of weights of the unit.
  - matrixB (matrix): It is a NxM matrix which holds the vectors that corresponds to the current unit.
  - dmetric (string): It takes values of "Euclidean", "SquaredEuclidean" etc.
  - lengthDataset (integer): It is the amount of dataset vectors.
  - numClasses(integer): It is the amount of classes of the label of dataset.
- **insertBetween (errorUnitIndex, neighborToError, gridID):** This function inserts either a row or a column between the error unit and its neighbor (i.e. most dissimilar).
  - errorUnitIndex (integer): The matrix index of the error unit.
  - neighborToError (Array/vector): It is an 1xM array of weights of the neighbor (i.e. most dissimilar) of error unit.
  - gridID(integer): It's the id of the SOM map.
- **insertRow (rowIndex, gridID)**:This function inserts a row into the SOM map. It, also, inserts the corresponding vectors into matrices 'weightsGrid' and 'unitsGird', which are global variables.
  - rowIndex(integer): This is the index of the row (i.e. in the SOM map) from which the new row will be inserted.
  - gridID(integer): It's the id of the SOM map.
- **insertColumn (colIndex, gridID):** This function inserts a column between two columns in grid. It, also, inserts the corresponding vectors into matrices 'weightsGrid' and 'unitsGrid', which are global variables.
  - colIndex(integer): This is the index of the column (i.e. in the SOM map) from which the new column will be inserted.
  - gridID (integer): It's the id of the SOM map.

- **getDirectNeighbors (errorUnitIndex, gridType):** This function returns the neighbors of the specified unit. In particular, it returns the top, right, bottom and left neighbor units of the specified unit.
  - errorUnitIndex(integer): The matrix index of the error unit.
  - gridType (string): It contains values, "Grid2d" or "Grid3d".
- **getMostDissimilarNeighbor (errorUnitIndex, dmetric):** This function finds the most dissimilar neighbor of the error unit.
  - errorUnitIndex(integer): The matrix index of the error unit.
  - dmetric (string): It takes values of "Euclidean", "SquaredEuclidean" etc.
- **gridInitializator (initializator,mode, startid):** This function initializes the grid, thus, it sets the initial units and the initial weights of the SOM map.
  - initializator (Data.frame): It keeps the initial rows, cols ,depth, dim, (Array)ranges, (Array)minimums and (Array)maximums.
  - mode (string): It takes values "Random" or "Gradient".
  - startid (integer): This is the initial id of the 'weigthsGrid' and 'unitsGrid' matrices.
- **gradientGInitializator (rows, cols, depth, gridID, dim, ranges, minimums, tmode, dataType, startid):** This function initializes gradiently the SOM map.
  - rows (integer): The number of initial rows of the SOM map.
  - cols (integer): The number of initial columns of the SOM map.
  - depth (integer): The number of units in third dimension of SOM map.
  - gridID (integer): It's the id of the SOM map.
  - dim (integer): It's the number of features of dataset.
  - ranges (Array/vector): It's the ranges of values for each feature of dataset.
  - minimums (Array/vector): It's the minimum values of each feature of dataset.
  - tmode (string): It takes values, "Supervised" or "Unsupservised".
  - dataType (string): It takes values, "Real" or "Class".
  - startid (integer): This is the initial id of the 'weigthsGrid' and 'unitsGrid' matrices.
- **randomGInitializator (rows, cols, depth, gridID, dim, minimums, maximums, tmode, dataType, numClasses, startid):** This function initializes randomly the SOM map.
  - rows (integer): The number of initial rows of the SOM map.
  - cols (integer): The number of initial columns of the SOM map.
  - depth (integer): The number of units in third dimension of SOM map.
  - gridID (integer): It's the id of the SOM map.
  - dim (integer): It's the number of features of dataset.
  - minimums (Array/vector): It's the minimum values of each feature of dataset.
  - maximums (Array/vector): It's the maximum values of each feature of dataset.
  - tmode (string): It takes values, "Supervised" or "Unsupservised".
  - dataType (string): It takes values, "Real" or "Class".

- numClasses (integer): It is the amount of classes of the label of dataset (i.e. for 'Real' dataset cases).
- startid (integer): This is the initial id of the 'weigthsGrid' and 'unitsGrid' matrices.

- **createOneUnitInfo (usecase_id, vectorUnits, weightsUnits, dmetric, startid):** This function creates information for a unit. In particular, it estimates mapped values average, the standard deviation, the absolute deviation, the mean quantization error etc. for a unit. The calculation process of unit information is fulfilled by the C module.
  - usecase_id (integer): The identifier of the specified use-case.
  - vectorsUnits (matrix): The mapped inputs of the current unit retrieved from the attribute view 'AT_DU_<NAME_OF_USECASE>'.
  - weightsUnits (matrix): The weights of the current unit retrieved from the attribute view 'AT_WU_<NAME_OF_USECASE>'.
  - dmetric(string): It takes values of "Euclidean", "SquaredEuclidean" etc.
  - startid (integer): This is the start id of the generated/returned matrix.
- **distancemetric(arrayA, arrayB, met, mode):** This function initializes the distance metric method.
  - arrayA(Array/Vector): This the first vector.
  - arrayB (Array/Vector): This is the second vector.
  - met (String): This contains values, "Euclidean" or "SquaredEuclidean" etc.
  - mode(String): This contains values, "neighborhood" or "distance".
- **euclidean (arrayA, arrayB,mode):** This function implements the Euclidean distance metric.
  - arrayA(Array/Vector): This the first vector.
  - arrayB(Array/Vector): This is the second vector.
  - mode(String): This contains values, "neighborhood" or "distance".
- **sqreuclidean(arrayA,arrayB,mode)**: This function implements the squared Euclidean distance metric.
  - arrayA sqreuclidean(arrayA,arrayB,mode):
  - arrayB(Array/Vector): This is the second vector.
  - mode(String): This contains values, "neighborhood" or "distance".
- **getMaximumValues(dim):** This function estimates the maximum values of each feature of a dataset. Also, it uses global variables initialized previously by the 'train()' function.
  - dim (Integer): It's the number of features of dataset.
- **getMinimimValues(dim):** This function estimates the minimum values of each feature of a dataset. Also, it uses global variables initialized previously by the 'train()' function.
  - dim (Integer): It's the number of features of dataset.
- **getRanges(dim):** This function estimates the ranges values of each feature of a dataset.
  - dim (Integer): It's the number of features of dataset.

- **qe0(dataset,dmetric,dataType):** This function initializes the estimation of the quantization error of the use-case dataset.
  - dataset(matrix): It contains the dataset.
  - dmetric(String): It takes values of "Euclidean", "SquaredEuclidean" etc.
  - dataType (String): It contains values, "Real" or "Class".
- **qe0Real(matrixA,dmetric):** This function estimates the quantization error of the dataset regarding a 'Real' use-case.
  - matrixA(matrix): It contains the dataset.
  - dmetric(String): It takes values of "Euclidean", "SquaredEuclidean" etc.
- **qe0Class(matrixA,dmetric):** This function estimates the quantization error of the dataset regarding a 'Class' use-case.
  - matrixA(matrix): It contains the dataset.
  - dmetric(String): It takes values of "Euclidean", "SquaredEuclidean" etc.
- **getStdDeviation(dataset, dimension, dataType):** This function initializes the estimation of the standard deviation of a dataset of a use-case.
  - dataset (matrix): It contains the dataset.
  - dimension (integer): It contains the number of features for which it will estimate the standard deviation.
  - dataType (String): It contains values, "Real" or "Class".
- **getStdDeviationReal (dataset, dimension):** This function estimates the standard deviation of the dataset regarding a 'Real' use-case.
  - dataset (matrix): It contains the dataset.
  - dimension (integer): It contains the number of features for which it will estimate the standard deviation.
- **getStdDeviationClass (dataset, dimension):** This function estimates the standard deviation of the dataset regarding a 'Class' use-case.
  - dataset (matrix): It contains the dataset.
  - dimension (integer): It contains the number of features for which it will estimate the standard deviation.
- **prediction (validationVecs,vecs, weightsUnits, unit_classes, gridParams, validationMethod, predictionParameters, cls, dataType, name, id):** This function proceeds to produce validation results about a trained use-case. Initially, it is called by the HANA procedure '<NAME_OF_USECASE>_VALIDATION_PROCESS'. Then, the R function defines and initializes the appropriate variables and, then, it calls the C module to manage the validation process.
  - validationVecs (data.frame): This data.frame contains the validation dataset.
  - vecs (data.frame): This data.frame contains the dataset of use-case.

- weightsUnits (data.frame): This data.frame contains the attribute view AT_WU_<NAME_OF_USECASE>.
- unit_classes (data.frame): This data.frame contains the classes of each unit of the SOM map of a specified use-case.
- gridParams(data.frame): This data.frame contains parameters about the grid.
- validationMethod (String): I contains the selected validation method. Values:'NearestWinnerAverage', 'BestMatchingUnits' etc.
- predictionParameters(data.frame): This data.frame contains parameters about the validation process.
- cls(data.frame): This data.frame contains the classes of the dataset, if the use-case is a 'Class' one.
- dataType (String): It contains values, "Real" or "Class".
- name (String): The name of use-case.
- id (integer): The id of use-case.

➢ **loadPrediction (dateTime, features, selectedAPfeatures, vec, weightsUnits, gridParams, predictionParameters):** This function proceeds to network load prediction process for a trained use-case. Initially, it is called by the HANA procedure '<NAME_OF_USECASE>_LOADPREDICTION_PROCESS'. Then, the R function defines and initializes the appropriate variables and, then, it calls the C module to execute the load prediction process.

- dateTime(data.frame): This data.frame contains the the time period in which the load prediction process will take place.
- features (data.frame): This data.frame contains the features of the dataset of the specified use-case.
- selectedAPFeatures (data.frame): This data.frame contains the selected Access Points (APs) for which the process will produce prediction results on the specified time period.
- vecs (data.frame): The dataset of the use-case.
- weightsUnits (data.frame) : This data.frame contains the attribute view AT_WU_<NAME_OF_USECASE>.
- gridParams(data.frame): This data.frame contains parameters about the grid.
- predictionParameters(data.frame): This data.frame contains parameters about the validation process.

➢ **component_line_plot (vecs, weightsUnits, unit_classes, gridParams, validationMethod, predictionParameters, cls, dataType, name, id):** This function produces results on how a specified feature of the dataset progress versus the estimation results on each vector of dataset. It is called by the HANA procedure

'<NAME_OF_USECASE>_COMPONENT_LINE_PLOT_MAP_PROCESS'. This R function delegates the C module for the execution and the production of the results.

- vecs (matrix): The dataset of the use-case.
- weightsUnits (data.frame) : This data.frame contains the attribute view AT_WU_<NAME_OF_USECASE>.
- unit_classes (data.frame): This data.frame contains the classes of each unit of the SOM map of a specified use-case.
- gridParams(data.frame): This data.frame contains parameters about the grid.
- validationMethod (String): I contains the selected validation method. Values:'NearestWinnerAverage', 'BestMatchingUnits' etc.
- predictionParameters(data.frame): This data.frame contains parameters about the validation process.
- dataType (String): It contains values, "Real" or "Class".
- name (String): The name of use-case.
- id (integer): The id of use-case.

- **saveFinalDataset(xvectors,headers,name,id):** This function saves temporarily the dataset in order to be retrieved later for storage purposes on HANA.
  - xvectors (matrix): The dataset
  - headers (Array/vector): The name of columns of column table 'DATASET_<NAME_OF_USECASE>'.
  - name(String) : The name of use-case.
  - id (integer): The id of use-case.
- **saveFinalUnits(uFrame,name,id):** This function saves temporarily the units in order to be retrieved later for storage purposes on HANA.
  - uFrame(data.frame): The units of SOM map.
  - name(String) : The name of use-case.
  - id (integer): The id of use-case.
- **saveFinalWeights(weightsGrid,headersWeights, name,id):** This function saves temporarily the weights of units in order to be retrieved later for storage purposes on HANA.
  - weightsGrid(matrix): The weights of all units of SOM map.
  - headersWeights(Array/vector): The name of columns of column table 'UNITSWEIGHTS_<NAME_OF_USECASE>'.
  - name(String) : The name of use-case.
  - id (integer): The id of use-case.
- **saveFinalGrid(maxRows,maxCols,name,id,gridID):** This function saves temporarily the grid information in order to be retrieved later for storage purposes on HANA.
  - maxRows (integer): The rows of the SOM map.

- maxCols (integer): The columns of the SOM map.
- name(String) : The name of use-case.
- id (integer): The id of use-case.
- gridID(integer): It's the id of the SOM map.

- **saveFinalLearningRates(learningRates,name,id,startid):** This function saves temporarily the learning rates of the training process in order to be retrieved later for storage purposes on HANA.
  - learningRates (matrix): The learning Rates.
  - name(String) : The name of use-case.
  - id (integer): The id of use-case.
  - startid (integer): This is the initial id of the of the learning rates matrix.
- **saveFinalMeanLearningRates(mlr,name,id,startid):** This function saves temporarily the mean learning rates of the training process in order to be retrieved later for storage purposes on HANA.
  - mlr (Array/vector): The learning Rates.
  - name(String) : The name of use-case.
  - id (integer): The id of use-case.
  - startid (integer): This is the initial id of the of the mean learning rates matrix.
- **saveFinalTrainParams(finalParamsFrame,name,id):** This function saves temporarily the parameters of the training process in order to be retrieved later for storage purposes on HANA.
  - finalParamsFrame (data.frame): The training parameters.
  - name(String) : The name of use-case.
  - id (integer): The id of use-case.
- **saveFinalqe0(qe0, name, id):** This function saves temporarily the quantization error of the training dataset of a specified use-case in order to be retrieved later for storage purposes on HANA.
  - qe0 (data.frame): The quantization error.
  - name(String) : The name of use-case.
  - id (integer): The id of use-case.
- **saveResultPrediction(nameResults,valueResults, name, id):** This function saves temporarily the prediction results a specified use-case in order to be retrieved later for storage purposes on HANA.
  - nameResults (Array/vector): The name of results.
  - valueResults (Array/vector): The results.
  - name(String) : The name of use-case.
  - id (integer): The id of use-case.

- **getFinalDataset(usecaseDF):** This function retrieves and returns the final dataset data.frame to SAP HANA Platform for storage purposes. This R function is called by the HANA procedure '<NAME_OF_USECASE>_GET_DATASET'.
    - usecaseDF (data.frame): This data.frame contains information about the specified use-case.
- **getFinalUnits(usecaseDF):** This function retrieves and returns the final units data.frame to SAP HANA Platform for storage purposes. This R function is called by the HANA procedure 'GET_UNITS'.
    - usecaseDF (data.frame): This data.frame contains information about the specified use-case.
- **getFinalWeights(usecaseDF):** This function retrieves and returns the final weights data.frame of SOM map units to SAP HANA Platform for storage purposes. This R function is called by the HANA procedure '<NAME_OF_USECASE>_GET_WEIGHTS'.
    - usecaseDF (data.frame): This data.frame contains information about the specified use-case.
- **getFinalGrid(usecaseDF):** This function retrieves and returns the final grid information data.frame to SAP HANA Platform for storage purposes. This R function is called by the HANA procedure 'GET_GRID'.
    - usecaseDF (data.frame): This data.frame contains information about the specified use-case.
- **getFinalLearningRates(usecaseDF):**This function retrieves and returns the final learning rates data.frame to SAP HANA Platform for storage purposes. This R function is called by the HANA procedure 'GET_LEARNINGRATES'.
    - usecaseDF (data.frame): This data.frame contains information about the specified use-case.
- **getFinalMeanLearningRates(usecaseDF):** This function retrieves and returns the final mean learning rates data.frame to SAP HANA Platform for storage purposes. This R function is called by the HANA procedure 'GET_MEANLEARNINGRATES'.
    - usecaseDF (data.frame): This data.frame contains information about the specified use-case.
- **getFinalTrainParams(usecaseDF):** This function retrieves and returns the final parameters of the training process to SAP HANA Platform for storage purposes. This R function is called by the HANA procedure 'GET_TRAINPARAMS'.
    - usecaseDF (data.frame): This data.frame contains information about the specified use-case.
- **getFinalqe0(name, id):** This function retrieves and returns the final quantization error of the training dataset to SAP HANA Platform for storage purposes.
    - name(String) : The name of use-case.

- id (integer): The id of use-case.

- **getResultPrediction(usecaseDF):** This function retrieves and returns the prediction results to SAP HANA Platform for storage purposes.
  - usecaseDF (data.frame): This data.frame contains information about the specified use-case.
- **estimateDatasetInfo(startid,id,dataset,dimension,dataType):** This function estimates the information regarding the training dataset of a specified usecase, such as, standard deviation, minimum values, maximum values, ranges, variance, mean quantization error etc.
  - startid (integer): This is the initial id of the of the generated/returned data.frame.
  - id (integer): This is the identifier of the current use-case.
  - dataset (matrix): This is the training dataset of the specified use-case.
  - dimension (integer): This is the number of features/columns of dataset.
  - dataType (String): It takes values, "Real" or "Class".

C module

As we mentioned previously, the C module is responsible for the training process and the prediction tasks. The R module delegates the C module to manage this tasks through the "R to C" interface [13], [14]. In the following we describe the functions that take place into the main execution tasks of the C module:

- **update_weights (double *xvectors, int *dimVectors, double *units, int *dimUnits, double *weights, int *dimWeights, double *neighborhoodSize, int *dimGrid, int *dmetricMethod, char **dataType, double *maxDistance,double *learningRates, int *dimLearningRates, int *flagStatus):** This function is responsible for the training process of the PLGSOM algorithm.
  - double *xvectors: The dataset.
  - int *dimVectors: An array which keeps the dimension of the dataset. Mainly, it contains two elements. The rows and the columns of the dataset matrix passed by the R function.
  - double *units: The units of the PLGSOM map.
  - int *dimUnits: An array which keeps the dimension of the units matrix. Mainly, it contains two elements. The rows and the columns of the units matrix passed by the R function.
  - double *weights: The weights of all units of the PLGSOM map.

- int *dimWeights: An array which keeps the dimension of the weights matrix. Mainly, it contains two elements. The rows and the columns of the weights matrix passed by the R function.
- double *neighborhoodSize: The size of the neighborhood.
- int *dimGrid: An array which keeps the dimensions of the PLGSOM map. Mainly, it contains two elements. The rows and the columns of the PLGSOM map.
- int *dmetricMethod: It keeps the identifier about the distance metric to use. Values: 0=Euclidean, 1=Squared Euclidean etc.
- char **dataType: It keeps the dataType of the dataset. Values: 'Real' or 'Class'.
- double *maxDistance: It keeps the maximum distance on PLGSOM map. It is used on the training process.
- double * learningRates: It keeps the learning rates of each iteration (meaning, per vector of dataset).
- int * dimLearningRates: It keeps the dimension (i.e. the length) of the '*learningRates'.
- int *flagStatus: It keeps a flag in order to realize if we are on the first epoch of training process or not. Values: 2='On first epoch', 1='Later epoch'

- **distanceMetric(double *xvector, int lengthXvector, double *weight, int lengthWeight, int metric, int featureIndex):** This function implements the distance metric.
  - double *xvector: A vector of the dataset.
  - int lengthXvector: The length of vector of dataset.
  - double *weight: The weights of a unit of the PLGSOM map.
  - int lengthWeight: The length of the weights of a unit.
  - int metric: The identifier on a distance metric that the user has chosen. Values 0='Euclidean', 1='Squared Euclidean', 2=Squared Euclidean (Missing values) etc.
  - int featureIndex: This variable is used by the "Squared Euclidean(Missing values)" metric and it represents the index of a feature which will not participate on the distance metric process.
- **euclidean(double *xvector, int lengthXvector, double *weight, int lengthWeight):** This function implements the Euclidean distance metric.
  - double *xvector: A vector of the dataset.
  - int lengthXvector: The length of vector of dataset.
  - double *weight: The weights of a unit of the PLGSOM map.
  - int lengthWeight: The length of the weights of a unit.
- **sqrteuclidean(double *xvector, int lengthXvector, double *weight, int lengthWeight) :** This function implements the Squared Euclidean distance metric.
  - double *xvector: A vector of the dataset.
  - int lengthXvector: The length of vector of dataset.

- double *weight: The weights of a unit of the PLGSOM map.
- int lengthWeight: The length of the weights of a unit.

- **bmu(double *xvector, int lengthXvector,double *units, int *dimUnits, double *weights, int *dimWeights, int dmetricMethod,double *bmuArray,int featureIndex):** This function implements the Best Matching Unit (BMU) process of the PLGSOM algorithm.
  - double *xvector: A vector of the dataset.
  - int lengthXvector: The length of vector of dataset.
  - double *units: The units of the PLGSOM map.
  - int *dimUnits: An array which keeps the dimension of the units matrix. Mainly, it contains two elements. The rows and the columns of the units matrix passed by the R function.
  - double *weights: The weights of all units of the PLGSOM map.
  - int *dimWeights: An array which keeps the dimension of the weights matrix. Mainly, it contains two elements. The rows and the columns of the weights matrix passed by the R function.
  - int dmetricMethod: It keeps the identifier about the distance metric to use. Values: 0=Euclidean, 1=Squared Euclidean etc.
  - double *bmuArray: It keeps the results of the BMU process. Element 0= 'id of winner weight',1='row of winner weight', 2='col of winner weight', ...,4='this is the distance of the best matching unit, thus the minimum distance', 5='the id of the best matching unit', 6='the index of the best matching unit in the matrix of "units". '.
  - int featureIndex: It is forwarded on 'distanceMetric' function.
- **getNBestMatchingUnits(double *xvector, int lengthXvector, double *weights, int *dimWeights, int *dmetricMethod, LUNITS *ptr_un, int *parameters,int featureIndex):** This function implements the N-BMU process.
  - double *xvector: A vector of the dataset.
  - int lengthXvector: The length of vector of dataset.
  - double *weights: The weights of all units of the PLGSOM map.
  - int *dimWeights: An array which keeps the dimension of the weights matrix. Mainly, it contains two elements. The rows and the columns of the weights matrix passed by the R function.
  - int *dmetricMethod: It keeps the identifier about the distance metric to use. Values: 0=Euclidean, 1=Squared Euclidean etc.
  - LUNITS *ptr_un: This struct represents the returned N-BMUs.
  - int *parameters: It represents the N. In particular, it contains an integer with the amount of selected BMUs.

- int featureIndex: It is forwarded on 'distanceMetric' function.

- **getBestMatchingWinner(double *xvector, int lengthXvector,double *units, int *dimUnits, double *weights, int *dimWeights, int *dmetricMethod,double *bmuArray,int count):** This function implements the Best Matching Winner(BMW) process.
  - double *xvector: A vector of the dataset.
  - int lengthXvector: The length of vector of dataset.
  - double *units: The units of the PLGSOM map which contain mapped inputs from the training dataset.
  - int *dimUnits: An array which keeps the dimension of the units matrix. Mainly, it contains two elements. The rows and the columns of the units matrix passed by the R function.
  - double *weights: The weights of the units which contain mapped inputs from the training dataset.
  - int *dimWeights: An array which keeps the dimension of the weights matrix. Mainly, it contains two elements. The rows and the columns of the weights matrix passed by the R function.
  - int *dmetricMethod: It keeps the identifier about the distance metric to use. Values: 0=Euclidean, 1=Squared Euclidean etc.
  - double *bmuArray: It keeps the results of the BMW process. Element 0= 'id of winner weight',1='row of winner weight', 2='col of winner weight', ...,4='this is the distance of the best matching winner, thus the minimum distance', 5='the id of the best matching winner', 6='the index of the best matching winner in the matrix of "units". '.
  - int count: For debugging reasons.
- **getMappedValuesAverageHandler(double *datasetVectors, int *dimDatasetVectors,double *winnerUnits, int *dimWinnerUnits, int *unitRowIndex, int *unit_id, double *mappedValuesAverage):** This function is a handler on getting the average value of a unit.
  - double *datasetVectors: The dataset.
  - int *dimDatasetVectors: An array which keeps the dimension of the dataset. Mainly, it contains two elements. The rows and the columns of the dataset matrix passed by the R function.
  - double *winnerUnits: The units of PLGSOM map which contains mapped inputs.
  - int *dimWinnerUnits: The dimensions of 'winnerUnits' matrix
  - int *unitRowIndex: The row index of selected unit.
  - int *unit_id: The id of the selected unit.
  - double *mappedValuesAverage: The average value of the selected unit.

- **getMappedValuesAverage(double *datasetVectors, int *dimDatasetVectors,double *winnerUnits, int *dimWinnerUnits, int unitRowIndex, int unit_id):** This function implements the estimation of average value of a unit.
  - double *datasetVectors: The dataset.
  - int *dimDatasetVectors: An array which keeps the dimension of the dataset. Mainly, it contains two elements. The rows and the columns of the dataset matrix passed by the R function.
  - double *winnerUnits: The units of PLGSOM map which contains mapped inputs.
  - int *dimWinnerUnits: The dimensions of 'winnerUnits' matrix
  - int unitRowIndex: The row index of selected unit.
  - int unit_id: The id of the selected unit
- **getValuesStdDevHandler(double *mappedValuesAverage, double *datasetVectors, int *dimDatasetVectors, double *winnerUnits, int *dimWinnerUnits, int *unitRowIndex, int *unit_id, double *valuesStdDev):** This function is a handler on getting the standard deviation of a unit.
  - double *mappedValuesAverage: the average value of the current unit.
  - double *datasetVectors: The dataset.
  - int *dimDatasetVectors: An array which keeps the dimension of the dataset. Mainly, it contains two elements. The rows and the columns of the dataset matrix passed by the R function.
  - double *winnerUnits: The units of PLGSOM map which contains mapped inputs.
  - int *dimWinnerUnits: The dimensions of 'winnerUnits' matrix
  - int *unitRowIndex: The row index of selected unit.
  - int *unit_id: The id of the selected unit
  - double *valuesStdDev: The standard deviation of the selected unit.
- **getValuesStdDev(double *mappedValuesAverage, double *datasetVectors, int *dimDatasetVectors,double *winnerUnits, int *dimWinnerUnits, int unitRowIndex, int unit_id):** This function implements the estimation of the standard deviation of a unit.
  - double *mappedValuesAverage: the average value of the current unit.
  - double *datasetVectors: The dataset.
  - int *dimDatasetVectors: An array which keeps the dimension of the dataset. Mainly, it contains two elements. The rows and the columns of the dataset matrix passed by the R function.
  - double *winnerUnits: The units of PLGSOM map which contains mapped inputs.
  - int *dimWinnerUnits: The dimensions of 'winnerUnits' matrix
  - int unitRowIndex: The row index of selected unit.
  - int unit_id: The id of the selected unit.

- **getValuesAbsDevHandler(double *mappedValuesAverage, double *datasetVectors, int *dimDatasetVectors,double *winnerUnits, int *dimWinnerUnits, int *unitRowIndex, int *unit_id, double *valuesAbsDev):** This function is a handler on getting the absolute deviation of a unit.
  - double *mappedValuesAverage: the average value of the current unit.
  - double *datasetVectors: The dataset.
  - int *dimDatasetVectors: An array which keeps the dimension of the dataset. Mainly, it contains two elements. The rows and the columns of the dataset matrix passed by the R function.
  - double *winnerUnits: The units of PLGSOM map which contains mapped inputs.
  - int *dimWinnerUnits: The dimensions of 'winnerUnits' matrix
  - int *unitRowIndex: The row index of selected unit.
  - int *unit_id: The id of the selected unit
  - double *valuesAbsDev: The absolute deviation of the selected unit.
- **getValuesAbsDev(double *mappedValuesAverage, double *datasetVectors, int *dimDatasetVectors,double *winnerUnits, int *dimWinnerUnits, int unitRowIndex, int unit_id) :** This function implements the estimation of the absolute deviation of a unit.
  - double *mappedValuesAverage: the average value of the current unit.
  - double *datasetVectors: The dataset.
  - int *dimDatasetVectors: An array which keeps the dimension of the dataset. Mainly, it contains two elements. The rows and the columns of the dataset matrix passed by the R function.
  - double *winnerUnits: The units of PLGSOM map which contains mapped inputs.
  - int *dimWinnerUnits: The dimensions of 'winnerUnits' matrix
  - int unitRowIndex: The row index of selected unit.
  - int unit_id: The id of the selected unit.
- **meanQuantizationErrorHandler(double *xvectors, int *dimXvectors, double *winnerUnits, int *dimWinnerUnits, int *unitRowIndex, int *unit_id, double *cWeight, int *dimWeights, int *dmetricMethod, int *totalNumVectors,double *qe0, double *mqe):** This function is a handler on getting the mean quantization error of a unit.
  - double *xvectors: The dataset.
  - int *dimXvectors: An array which keeps the dimension of the dataset. Mainly, it contains two elements. The rows and the columns of the dataset matrix passed by the R function.
  - double *winnerUnits: The units of PLGSOM map which contain mapped inputs.
  - int *dimWinnerUnits: The dimensions of 'winnerUnits' matrix
  - int *unitRowIndex: The row index of selected unit.

- int *unit_id: The id of the selected unit
- double *cWeight: The weights of current unit.
- double *dimWeights: The length of the 'cWeight' vector.
- int *dmetricMethod: It keeps the identifier about the distance metric to use. Values: 0=Euclidean, 1=Squared Euclidean etc.
- int *totalNumVectors: The amount of vectors of the training dataset.
- double *qe0: The quantization error of the training dataset.
- double *mqe: The mean quantization error of the selected unit.

- **meanQuantizationError(double *xvectors, int *dimXvectors, double *winnerUnits, int *dimWinnerUnits, int *unitRowIndex, int *unit_id, double *cWeight, int *dimWeights, int *dmetricMethod, int *totalNumVectors, double *qe0):** This function implements the estimation of the mean quantization error of a unit.
  - double *xvectors: The dataset.
  - int *dimXvectors: An array which keeps the dimension of the dataset. Mainly, it contains two elements. The rows and the columns of the dataset matrix passed by the R function.
  - double *winnerUnits: The units of PLGSOM map which contains mapped inputs.
  - int *dimWinnerUnits: The dimensions of 'winnerUnits' matrix
  - int *unitRowIndex: The row index of selected unit.
  - int *unit_id: The id of the selected unit
  - double *cWeight: The weights of current unit.
  - double *dimWeights: The length of the 'cWeight' vector.
  - int *dmetricMethod: It keeps the identifier about the distance metric to use. Values: 0=Euclidean, 1=Squared Euclidean etc.
  - int *totalNumVectors: The amount of vectors of the training dataset.
  - double *qe0: The quantization error of the training dataset.
- **bestMatchingUnitsMethod (double *xvector, int lengthXvector, double *units, int *dimUnits,int *dimGrid, double *weights, int *dimWeights, int *dmetricMethod, int *parameters):** This function implements the Best Matching Units method (for prediction purposes).
  - double *xvector: A vector of the dataset.
  - int lengthXvector: The length of vector of dataset.
  - double *units: The units of the PLGSOM map which contain mapped inputs from the training dataset.
  - int *dimUnits: An array which keeps the dimension of the units matrix. Mainly, it contains two elements. The rows and the columns of the units matrix passed by the R function.

- int *dimGrid: An array which keeps the dimensions of the PLGSOM map. Mainly, it contains two elements. The rows and the columns of the PLGSOM map.
- double *weights: The weights of all units of the PLGSOM map.
- int *dimWeights: An array which keeps the dimension of the weights matrix. Mainly, it contains two elements. The rows and the columns of the weights matrix passed by the R function.
- int *dmetricMethod: It keeps the identifier about the distance metric to use. Values: 0=Euclidean, 1=Squared Euclidean etc.
- int *parameters: It contains an integer with the amount of selected BMUs.

- **bestMatchingUnitsWeightedMethod(double *xvector, int lengthXvector, double *units, int *dimUnits,int *dimGrid, double *weights, int *dimWeights, int *dmetricMethod, int *parameters)):** This function implements the Best Matching Units(Weighted) method (for prediction purposes).
  - double *xvector: A vector of the dataset.
  - int lengthXvector: The length of vector of dataset.
  - double *units: The units of the PLGSOM map which contain mapped inputs from the training dataset.
  - int *dimUnits: An array which keeps the dimension of the units matrix. Mainly, it contains two elements. The rows and the columns of the units matrix passed by the R function.
  - int *dimGrid: An array which keeps the dimensions of the PLGSOM map. Mainly, it contains two elements. The rows and the columns of the PLGSOM map.
  - double *weights: The weights of all units of the PLGSOM map.
  - int *dimWeights: An array which keeps the dimension of the weights matrix. Mainly, it contains two elements. The rows and the columns of the weights matrix passed by the R function.
  - int *dmetricMethod: It keeps the identifier about the distance metric to use. Values: 0=Euclidean, 1=Squared Euclidean etc.
  - int *parameters: It contains an integer with the amount of selected BMUs.
- **getNBestMatchingWinners(double *xvector,int lengthXvector, double *winnerWeights, int *dimWinnerWeights, double *winnerUnits, int *dimWinnerUnits ,int *dmetricMethod, LUNITS *ptr_un,int *parameters)** This function implements the N-Best Matching Winners(BMW) process.
  - double *xvector: A vector of the dataset.
  - int lengthXvector: The length of vector of dataset.
  - double *winnerWeights: It's the weights of units which contain mapped inputs.
  - int *dimWinnerWeights: It contains the dimensions of the 'winnerWeights' matrix.

- double *winnerUnits : It 's the units which contain mapped inputs.
- int *dimWinnerUnits: It contains the dimensions of 'winnerUnits' matrix.
- int *dmetricMethod: It keeps the identifier about the distance metric to use. Values: 0=Euclidean, 1=Squared Euclidean etc.
- LUNITS *ptr_un: This struct represents the returned N-BMWs.
- int *parameters: It represents the N. In particular, it contains an integer with the amount of selected BMWs.

➢ **getNearestWinner(UNIT *bmu, double *winnerUnits, int *dimWinnerUnits, double *winnerWeights, int *dimWinnerWeights,int *dmetricMethod, UNIT *ptr_winner):** This function retrieves the nearest winner unit.

- UNIT *bmu: The best-matching-unit.
- double *winnerUnits : It 's the units which contain mapped inputs.
- int *dimWinnerUnits: It contains the dimensions of 'winnerUnits' matrix.
- double *winnerWeights: It's the weights of units which contain mapped inputs.
- int *dimWinnerWeights: It contains the dimensions of the 'winnerWeights' matrix.
- int *dmetricMethod: It keeps the identifier about the distance metric to use. Values: 0=Euclidean, 1=Squared Euclidean etc.
- UNIT *ptr_winner: The returned nearest winner.

➢ **kNearestWinners(double *xvector, int lengthXvector, double *winnerUnits, int *dimWinnerUnits, double *winnerWeights, int *dimWinnerWeights, int *dmetricMethod, int *dimClasses, int *idClasses, int *unitClasses, int *lengthUnitClasses, int *parameters):** This function estimates the K-nearest winner units.

- double *xvector: A vector of the dataset.
- int lengthXvector: The length of vector of dataset.
- double *winnerUnits : It 's the units which contain mapped inputs.
- int *dimWinnerUnits: It contains the dimensions of 'winnerUnits' matrix.
- double *winnerWeights: It's the weights of units which contain mapped inputs.
- int *dimWinnerWeights: It contains the dimensions of the 'winnerWeights' matrix.
- int *dmetricMethod: It keeps the identifier about the distance metric to use. Values: 0=Euclidean, 1=Squared Euclidean etc.
- int *dimClasses: The number of classes, if the training dataset is in 'Class' data-type.
- int *idClasses: The IDs of the classes.
- int *unitClasses: This variable contains the number of inputs labeled by the classes per unit.

- int *lengthUnitClasses: The dimensions of 'unitClasses' matrix.
- int *parameters: It represents the K. In particular, it contains an integer with the amount of selected nearest winners.

- **getNNearestNeighbors(double *dataset, int *dimDataset, double *xvector, int lengthXvector, int *dmetricMethod,int n, NEIGHBOR **ptr_neighbors,NEIGHBOR *ptr_neighbor):** This function returns the N-nearest neighbors.
  - double *dataset: The dataset.
  - int *dimDataset: The dimensions of the 'dataset' matrix.
  - double *xvector: A vector of the dataset.
  - int lengthXvector: The length of vector of dataset.
  - int *dmetricMethod: It keeps the identifier about the distance metric to use. Values: 0=Euclidean, 1=Squared Euclidean etc.
  - NEIGHBOR **ptr_neighbors: An array containing N 'NEIGHBOR'.
  - NEIGHBOR *ptr_neighbor: The selected neighbor.
- **kNN(double *dataset, char **datasetClasses, int *dimDataset, double *xvector, int lengthXvector, int *dmetricMethod, char **classes, int *dimClasses, int *parameters):** This function estimates the K-nearest neighbors.
  - double *dataset: The dataset.
  - char **datasetClasses: The classes of each vector in dataset, if it's a 'Class' one.
  - int *dimDataset: The dimensions of the 'dataset' matrix.
  - double *xvector: A vector of the dataset.
  - int lengthXvector: The length of vector of dataset.
  - int *dmetricMethod: It keeps the identifier about the distance metric to use. Values: 0=Euclidean, 1=Squared Euclidean etc.
  - char ** classes: The classes of the training dataset, if it's a 'Class' one.
  - int *dimClasses: The number of classes, if the training dataset is in 'Class' data-type.
  - int *parameters: It represents the K. In particular, it contains an integer with the amount of selected nearest winners.
- **classificationError(int numClasses,int *unitClasses, int *lengthUnitClasses, int winningClassCount, int numMappedInputs, double cleZero):** This function estimates the Classification Error.
  - int numClasses: the number of classes in a training dataset.
  - int *unitClasses: This variable contains the number of inputs labeled by the classes per unit.
  - int *lengthUnitClasses: The dimensions of 'unitClasses' matrix.
  - int winningClassCount: The count of a specified class into a unit.
  - int numMappedInputs: The number of the mapped inputs into a unit.

- double cleZero: The classfication error of the dataset.

- **quantizationError(double *xvectors, int *dimXvectors, double *winnerUnits, int *dimWinnerUnits, int unitRowIndex, int unit_id, double *cWeight, int *dimWeights, int * dmetricMethod, int totalNumVectors,double *qe0, double *numMappedInputs):** This function estimates the quantization error for a specified unit.
  - double *xvectors: The dataset.
  - int *dimXvectors: An array which keeps the dimension of the dataset. Mainly, it contains two elements. The rows and the columns of the dataset matrix passed by the R function.
  - double *winnerUnits: The units of PLGSOM map which contain mapped inputs.
  - int *dimWinnerUnits: The dimensions of 'winnerUnits' matrix
  - int *unitRowIndex: The row index of selected unit.
  - int unit_id: The id of the selected unit
  - double *cWeight: The weights of current unit.
  - double *dimWeights: The length of the 'cWeight' vector.
  - int *dmetricMethod: It keeps the identifier about the distance metric to use. Values: 0=Euclidean, 1=Squared Euclidean etc.
  - int totalNumVectors: The amount of vectors of the training dataset.
  - double *qe0: The quantization error of the training dataset.
  - double *numMappedInputs: The number of mapped inputs for the specified unit.
- **bestMatchingWinners(double *xvector, int lengthXvector, double *winnerUnits, int *dimWinnerUnits, double *winnerWeights, int *dimWinnerWeights, int *dmetricMethod, int *dimClasses, int *idClasses, int *unitClasses, int *lengthUnitClasses, int *parameters):** This function implements the Best-Matching-Winner process.
  - double *xvector: A vector of the dataset.
  - int lengthXvector: The length of vector of dataset.
  - double *winnerUnits: The units of PLGSOM map which contains mapped inputs.
  - int *dimWinnerUnits: The dimensions of 'winnerUnits' matrix
  - double *winnerWeights: It's the weights of units which contain mapped inputs.
  - int *dimWinnerWeights: It contains the dimensions of the 'winnerWeights' matrix.
  - int *dmetricMethod: It keeps the identifier about the distance metric to use. Values: 0=Euclidean, 1=Squared Euclidean etc.
  - int *dimClasses: The number of classes, if the training dataset is in 'Class' data-type.
  - int *idClasses: The IDs of the classes.

- int *unitClasses: This variable contains the number of inputs labeled by the classes per unit.
- int *lengthUnitClasses: The dimensions of 'unitClasses' matrix.
- int *parameters: It represents the K. In particular, it contains an integer with the amount of selected nearest winners.

- **predictionClassification(double *xvectors, char **xvectorsClasses, int *dimXvectors, double *weights, int *dimWeights, double *winnerWeights, int *dimWinnerWeights, double *units, int *dimUnits, double *winnerUnits, int *dimWinnerUnits, int *dimGrid, double *validationVectors, char **validationClasses, int *dimValidationVectors, int *parameters, int *validation_method, int *dmetricMethod, char **classes, int *idClasses, int *dimClasses, int *unitClasses, int *lengthUnitClasses,double *cleZero, double *successRate):** This function produces validation results for a 'Class' dataset. It is called by the 'prediction' R function.
  - double *xvectors: The dataset.
  - char **xvectorsClasses: The classes of each vector in the training dataset.
  - int *dimXvectors: An array which keeps the dimension of the dataset. Mainly, it contains two elements. The rows and the columns of the dataset matrix passed by the R function.
  - double *weights: The weights of all units of the PLGSOM map.
  - int *dimWeights: An array which keeps the dimension of the weights matrix. Mainly, it contains two elements. The rows and the columns of the weights matrix passed by the R function.
  - double *winnerWeights: It's the weights of units which contain mapped inputs.
  - int *dimWinnerWeights: It contains the dimensions of the 'winnerWeights' matrix.
  - double *units: The units of the PLGSOM map.
  - int *dimUnits: An array which keeps the dimension of the units matrix. Mainly, it contains two elements. The rows and the columns of the units matrix passed by the R function.
  - double *winnerUnits: The units of PLGSOM map which contains mapped inputs.
  - int *dimWinnerUnits: The dimensions of 'winnerUnits' matrix
  - int *dimGrid: An array which keeps the dimensions of the PLGSOM map. Mainly, it contains two elements. The rows and the columns of the PLGSOM map.
  - double *validationVectors: The validation dataset.
  - char **validationClasses: The classes of each vector in the validation dataset.
  - int *dimValidationVectors: The dimensions of 'validationVectors' matrix.
  - int *parameters: Some parameters for the validation process.

- int *validation_method: The validation method which will perform the validation. Values: 1=LayerCertaintyBased, 2=LayerCertaintyBasedEx, 3=BestMatchingWinners, 4=BestMatchingWinnersWeighted, 5=kNearestWinners, 6=kNearestWinnersWeighted, 7=kNN
- int *dmetricMethod: It keeps the identifier about the distance metric to use. Values: 0=Euclidean, 1=Squared Euclidean etc.
- char ** classes: The classes of the training dataset.
- int *idClasses: The IDs of the classes.
- int *dimClasses: The number of classes, if the training dataset is in 'Class' data-type.
- int *unitClasses: This variable contains the number of inputs labeled by the classes per unit.
- int *lengthUnitClasses: The dimensions of 'unitClasses' matrix.
- double *cleZero: The classfication error of the dataset.
- double *successRate: The total estimated successRate of the validation process.

➢ **componentLinePlot(double *xvectors, int *dimXvectors,double *winnerWeights, int *dimWinnerWeights, double *winnerUnits, int *dimWinnerUnits, int *dmetricMethod, double *estimationArray):** This function produces results for the R function ' component_line_plot'.

- double *xvectors: The dataset.
- int *dimXvectors: An array which keeps the dimension of the dataset. Mainly, it contains two elements. The rows and the columns of the dataset matrix passed by the R function.
- double *winnerWeights: It's the weights of units which contain mapped inputs.
- int *dimWinnerWeights: It contains the dimensions of the 'winnerWeights' matrix.
- double *winnerUnits: The units of PLGSOM map which contains mapped inputs.
- int *dimWinnerUnits: The dimensions of 'winnerUnits' matrix
- int *dmetricMethod: It keeps the identifier about the distance metric to use. Values: 0=Euclidean, 1=Squared Euclidean etc.
- double *estimationArray: The returned results: component vs estimation results.

➢ **predictionEstimation(double *xvectors, int *dimXvectors, double *weights, int *dimWeights, double *units, int *dimUnits,double *winnerWeights, int *dimWinnerWeights, double *winnerUnits, int *dimWinnerUnits, int *dimGrid, double *validationVectors, int *dimValidationVectors, int *parameters, int *validation_method, int *dmetricMethod, int *numberOfUnits, double *MAE, double *MAPE, double *MSE, double *RMSE, double *MAAPE, double *estimationArray,**

double *selectedUnitId): This function produces validation results for a 'Real' dataset. It is called by the 'prediction' R function.

- double *xvectors: The dataset.
- int *dimXvectors: An array which keeps the dimension of the dataset. Mainly, it contains two elements. The rows and the columns of the dataset matrix passed by the R function.
- double *weights: The weights of all units of the PLGSOM map.
- int *dimWeights: An array which keeps the dimension of the weights matrix. Mainly, it contains two elements. The rows and the columns of the weights matrix passed by the R function.
- double *units: The units of the PLGSOM map.
- int *dimUnits: An array which keeps the dimension of the units matrix. Mainly, it contains two elements. The rows and the columns of the units matrix passed by the R function.
- double *winnerWeights: It's the weights of units which contain mapped inputs.
- int *dimWinnerWeights: It contains the dimensions of the 'winnerWeights' matrix.
- double *winnerUnits: The units of PLGSOM map which contains mapped inputs.
- int *dimWinnerUnits: The dimensions of 'winnerUnits' matrix
- int *dimGrid: An array which keeps the dimensions of the PLGSOM map. Mainly, it contains two elements. The rows and the columns of the PLGSOM map.
- double *validationVectors: The validation dataset.
- int *dimValidationVectors: The dimensions of 'validationVectors' matrix.
- int *parameters: Some parameters for the validation process.
- int *validation_method: The validation method which will perform the validation. Values: 1=BestMatchingUnits, 2= BestMatchingUnits(Weighted), 3=NearestWinnerAverage.
- int *dmetricMethod: It keeps the identifier about the distance metric to use. Values: 0=Euclidean, 1=Squared Euclidean etc.
- int *numberOfUnits: The number of unit which will take into account the validation method.
- double *MAE: The estimated MAE.
- double *MAPE: The estimated MAPE.
- double *MSE: The estimated MSE.
- double *RMSE: The estimated RMSE.
- double *MAAPE: The estimated MAAPE.
- double *estimationArray: This array keeps all validation/estimation results.
- double *selectedUnitId: Debugging purposes.

- **loadPrediction(double *xvectors, int *dimXvectors, double *weights, int *dimWeights, double *units, int *dimUnits,double *winnerWeights, int *dimWinnerWeights, double *winnerUnits, int *dimWinnerUnits, int *dimGrid, double *validationVectors, int *dimValidationVectors, int *parameters, int *validation_method, int *dmetricMethod, int *numberOfUnits, double *estimationArray, int *dimEstimationArray, double *MAE, double *MAPE, double *MSE, double *RMSE, double *MAAPE)):** This function produces load prediction results for a 'Real' dataset. It is called by the 'loadPrediction' R function.
  - double *xvectors: The dataset.
  - int *dimXvectors: An array which keeps the dimension of the dataset. Mainly, it contains two elements. The rows and the columns of the dataset matrix passed by the R function.
  - double *weights: The weights of all units of the PLGSOM map.
  - int *dimWeights: An array which keeps the dimension of the weights matrix. Mainly, it contains two elements. The rows and the columns of the weights matrix passed by the R function.
  - double *units: The units of the PLGSOM map.
  - int *dimUnits: An array which keeps the dimension of the units matrix. Mainly, it contains two elements. The rows and the columns of the units matrix passed by the R function.
  - double *winnerWeights: It's the weights of units which contain mapped inputs.
  - int *dimWinnerWeights: It contains the dimensions of the 'winnerWeights' matrix.
  - double *winnerUnits: The units of PLGSOM map which contains mapped inputs.
  - int *dimWinnerUnits: The dimensions of 'winnerUnits' matrix
  - int *dimGrid: An array which keeps the dimensions of the PLGSOM map. Mainly, it contains two elements. The rows and the columns of the PLGSOM map.
  - double *validationVectors: The validation dataset.
  - int *dimValidationVectors: The dimensions of 'validationVectors' matrix.
  - int *parameters: Some parameters for the validation process.
  - int *validation_method: The validation method which will perform the validation. Values: 1=BestMatchingUnits, 2= BestMatchingUnits(Weighted), 3=NearestWinnerAverage.
  - int *dmetricMethod: It keeps the identifier about the distance metric to use. Values: 0=Euclidean, 1=Squared Euclidean etc.
  - int *numberOfUnits: The number of unit which will take into account the validation method.
  - double *estimationArray: This array keeps all validation/estimation results.

- double *MAE: The estimated MAE.
- double *MAPE: The estimated MAPE.
- double *MSE: The estimated MSE.
- double *RMSE: The estimated RMSE.
- double *MAAPE: The estimated MAAPE.

Java module

On a previous paragraph, we mentioned that the Java part of the module is responsible to create into the SAP HANA all the dynamic column tables, HANA views, procedures and table types that the section 4.2.1 have already described. The Java module exploits the Java Database Connectivity (JDBC) in order to interact with the SAP HANA Platform. By this way, the R module uses pure SQL in order to create dynamically the objects we have already mentioned above. The process elaborates a number of templates on which it applies regular expression methods for the construction of the corresponding objects for each use-case.

4.2.3 Front-end User Interface (UI)

The front-end user interface is built by using a special technology of SAP HANA Platform, the SAP HANA Extended Application Services (SAP HANA XS) [11]. This technology provides to developers the capability to write server-side code in order to rapidly and easily access the HANA objects that we described in section 4.2.1 (i.e. column tables, HANA views, procedures etc.). In particular, the HANA Platform offers the so called XS server, which supports server-side application programming in Javascript. The developer can exploit the Javascript API to expose authorized data to client requests coming, for example, from a Web browser [11].

Generally, the Javascript APIs provided by the XS server can use the Data Manipulation Language (DML) capabilities in order to select, insert, update and delete data on HANA Platform. Essentially, the Javascript APIs perform the following actions [11]:

- Interact with the SAP HANA XS runtime environment.
- Directly access SAP HANA database capabilities.
- Interact with services on defined HTTP destinations.

The HANA XS engine is built on the Mozilla SpiderMonkey Javascript engine [16] in order to interpret the server-side Javascript code. Generally, the developer can create ".xsjs" source files which are pure Javascript source files and they are interpreted by the HANA XS engine as they are the main controllers of a HANA XS application. In addition, the developer can create ".xsjslib" source files which are custom libraries that a developer may needs to create and

import into his ".xsjs" and ".xsjslib" files in order to confer a re-usable and modular nature to the XS application.

In the following paragraphs, we provide information about the development of the front-end UI of the KGV tool which exploits the SAP HANA Extended Application Services (SAP HANA XS). In particular, we'll describe the main controllers and libraries of the XS Application that was built in order to exploit the designed HANA data model and the calculation R module.

Furthermore, we'll describe the client-side programming modules which are, also, written in Javascript programming language and they are responsible for the graphical design of the user interface and the communication with the SAP HANA Platform Extended Application Services.

Services - Server-side Programming in Javascript

- **createUsecase.xsjs:** This service is responsible for the creation of the initial parameters of a new use-case requested by a user. It receives the name of the new use-case and it initializes the appropriate columns into the column table 'USECASE'.
- **trainUsecase.xsjs:** This service is responsible for the training process of the KGV tool. It receives the initial parameters of the PLGSOM Algorithm such as, the distance metric, the error metric, the neighborhood size, the target quality, the maximum PLGSOM map area etc. After it receives the above parameters, it calls the '<NAME_OF_USECASE>_TRAIN' HANA procedure to proceed to training process. Once the training process has finished, the service calls the '<NAME_OF_USECASE>_CALC_UNITINFO' for the estimation of additional information of units, such as the standard deviation, the absolute deviation, the mean quantization error etc.
- **uploadUsecase.xsjs:** Mainly, this service is responsible for the upload of a dataset on a specified use-case. It receives as parameters the name of filename, the name of use-case and the data type (i.e. either 'Class' or 'Real'). At first, it checks if the appropriate HANA column tables, views and procedures exists. If not, it calls the HANA procedure 'CREATE_MODEL_FROM_TEMPLATES' by passing the id of the use-case and it creates dynamically the appropriate HANA column tables, views and procedures. Afterwards, it uploads the dataset to the column table 'DATASET_<NAME_OF_USECASE>'.
- **loadUsecase.xsjs:** This service is responsible for the load of a specified use-case. It receives as a parameter the name of use-case and, finally, it returns the PLGSOM map and all the appropriate information regarding the already trained use-case.
- **predictUsecase.xsjs:** This service is responsible for the prediction process of the KGV tool. It receives as parameters the (time) interval in order to initialize the prediction

process. Then, it calls the '<NAME_OF_USECASE>_LOAD_PREDICTION' and, finally, it returns the prediction results to the client of the KGV tool.

- **calcDatasetInfo.xsjs:** This service is responsible for the estimation of some statistics regarding the dataset of use-case, such as the maximum values, the minimum values, the quantization error, the deviation etc.
- **getLearningRates.xsjs:** This service is responsible for the retrieval of the learning rates of the last epoch of a specified use-case. This is done by interacting with the column table "LEARNINGRATES". Finally, it returns an array of the learning rates of the use-case.
- **getMeanLearningRates.xsjs:** Similarly with service "getLearningRates.xsjs", this service is responsible for the retrieval of the mean learning rates of the training process of a specified use-case. This is done by interacting with the column table "MEANLEARNINGRATES". Finally, it returns an array of the mean learning rates of the use-case.
- **getDatasetInfo.xsjs:** This service is responsible for retrieving the training dataset statistics. This is done by interacting with the column table "DATASETINFO" which collects all these kind statistics for every use-case. In particular, this service retrieves the minimum values, the maximum values, the quantization error, the mean quantization error, the variance, the deviation, the mean values etc.
- **getFeatureNames.xsjs:** This service is responsible for the retrieval of the feature names which participate into the training and prediction process of the KGV tool. It receives as a parameter the name of the use-case.
- **uploadValidateUsecase.xsjs:** Mainly, this service is responsible for the validation process of an already trained use-case. It receives as parameters the name of use-case, some initial parameters regarding the validation process (i.e. validation method etc.) and the validation dataset. At first, the service uploads the validation dataset on the column table 'VALIDATION_<NAME_OF_USECASE>' and, then, it calls the HANA procedure '<NAME_OF_USECASE>_VALIDATION' which performs the validation process. Finally, it returns the results to the clients of the KGV tool. In particular, the results are two arrays, the known values of the label of dataset and the estimated values of the label of dataset.
- **hanaDML.xsjslib:** This is a library which contains useful functions to the above XS services. In particular, it contains Data Manipulation Language (DML) functions for selection, insertion, update and delete of data into the SAP HANA database. All services has access to this library by importing it appropriately.
- **HANAconnection.xsjslib:** This is a library which contains important functions for the profiling of users. In particular, it identifies the schema and general information about the users in order to be used for the retrieve of data from the appropriate tables and schemas of SAP HANA database.

- **util.xsjslib:** This is a library which contains useful functions to the above XS services. More specifically, it contains general functions regarding usual requested tasks applied on the SAP HANA database. It is about for the fulfillment of more specific and re-usable tasks, such as uploading dataset, retrieving the feature names, retrieving statistics data regarding the units of the PLGSOM map of a specified use-case etc.

Client-side Javascript

- **controller.js:** This Javascript file works like the main controller of the user interface regarding the actions of the elements like buttons, dropdown boxes etc. Each button has been attached a controller function in order to communicate with the SAP HANA Platform, retrieve data and, finally, visualize them appropriately. The most functions exploit the jQuery library of Javascript programming language.
- **prediction.js:** This Javascript file is responsible for the prediction process of the user interface of KGV tool. Mainly, it communicates with the SAP HANA XS services for the management of prediction tasks, it constructs the data to send, it manipulates the data that the user-interface receives, it creates the diagram in order to demonstrate the results of prediction process. It is written in object-oriented logic in order to be initialized and controlled by a function of the 'controller.js' file.
- **grid.js:** This Javascript file is responsible for the design and management of the grid on the user-interface of the KGV tool. Essentially, this grid depicts the results of the training process of a use-case. It contains functions for the creation and update of grid. In addition, it contains controls for the interaction with the grid, such as clicks, zoom in, zoom out etc. It is written in object-oriented logic in order to be initialized and controlled by a function of the 'controller.js' file.
- **unit.js:** This Javascript file is responsible for the management of grid's unit data on the user-interface of the KGV tool. In particular, each unit in a grid is represented by an instance of the object 'Unit' which is declared in this Javascript file. An instance of 'Unit' object holds unit information, such as the weights of a unit, index of row in the grid, index of column the grid and some important statistics of the unit like deviation, mean quantization error, amount of mapped inputs etc. It is written in object-oriented logic in order to be initialized and controlled by a function of the 'controller.js' file.
- **tabs.js:** This Javascript file is responsible for the management of the tabs which are contained into the user-interface of the KGV tool. Each tab in the user-interface addresses a specific job in the KGV tool like training of the PLGSOM algorithm, prediction by using a trained PLGSOM map and a selected prediction method etc.
- **FileUploader.view.js:** This Javascript file is responsible for the design of the elements regarding the upload of dataset for a specific use-case. It uses a Javascript library of SAP HANA in order to design the elements and attach the even handlers on them.

- **FileUploader.controller.js:** This Javascript file takes the role of the controller regarding the actions of the upload elements designed by the 'FileUpload.view.js' Javascript file. It uploads the dataset to the service 'uploadUsecase.xsjs' in order to be imported to the column table 'DATASET_<NAME_OF_USECASE>'. In addition, this controller provides a function for the event handling of the response of the aforementioned service.
- **ButtonValidation.view.js:** This Javascript file is responsible for the design of the button regarding the upload of validation dataset for a specific use-case in the validation tab. It uses a Javascript library of SAP HANA in order to design the elements and attach the even handlers on them.
- **ButtonValidation.controller.js:** This Javascript file takes the role of the controller regarding the actions of the upload elements (i.e. the upload button) designed by the 'ButtonValidation.view.js'. It uploads the validation dataset to the service 'uploadValidateUsecase.xsjs' in order to import it to the column table 'VALIDATION_<NAME_OF_USECASE>' and ,then, to perform the validation process. Moreover, the controller it attaches to the HTTP message to the service a number of parameters, such as the name of use-case, the validation method, the distance metric etc.
- **FileUploaderValidation.view.js:** This Javascript file is responsible for the design of the browser element which is related with the upload of the validation dataset for a specific use-case in the validation tab. It uses a Javascript library of SAP HANA in order to design the elements and attach the even handlers on them.
- **FileUploaderValidation.controller.js:** This Javascript file takes the role of the controller regarding the actions of the upload elements (i.e. the browser element) designed by the 'FileUploaderValidation.view.js'. It contains the event handling function of the browser element which manipulates the responses of the validation service 'uploadValidateUsecase.xsjs'. This means that it receives the results of the validation process and it designs the graph 'Data VS SOM' which compares the given label of the validation dataset and the estimated label of validation process.

In addition, the KGV tool uses the library 'RGraph v3' for the graph diagrams and the visualization of results produced by the core calculation engine of KGV tool. This library is written in Javascript programming language and it exploits the benefits of the HTML5 element 'canvas' in order to easily design graphs for desktop and mobile applications.

5 Scenario Demonstration, Results and Conclusions of the Knowledge Generation and Visualization tool

5.1 Introduction

In this chapter, we provide a demonstration scenario of the Knowledge Generation and Visualization tool in order to highlight the functionalities of the tool and the interactions between the elements of the software that we described in the Chapter 4. In particular, we demonstrate the following capabilities of the KGV tool:

- Creation process of a use-case
- Training Process
- Prediction Process on a Network Load Scenario
- Validation Process in the format of SOM vs Data
- Visualization and provision of statistics related with the specified use-case and its dataset
- Visualization of the SOM map and the statistics that are imposed by it.

In the following sections, we provide further details about the above processes in the form of descriptions and figures.

5.2 Demonstration Scenario

5.2.1 The Creation Process of a use-case - Upload of dataset

The KGV tool provides a well-designed functionality in order to create a use-case and afterwards to upload a dataset of vectors to this use-case. In the "SAP HANA" tab, the user can write the name of the use-case and, then, he presses the button "Create". The KGV tool sends an HTTP request to the application server of the SAP HANA platform. This request is manipulated by the "createUsecase.xsjs" service (see section 4.2.3).

Once the use-case has been created, the user can upload a dataset on the SAP HANA Platform regarding that use-case. The dataset has to be in ".mdt" format. This means that in each vector there will be the ";" as the delimiter between the features. The Figure 20 and Figure 21 illustrate that the user specifies the use-case and ,then, by clicking the button "Browse...", he browses his PC's file system in order to select an appropriate dataset for the use-case.

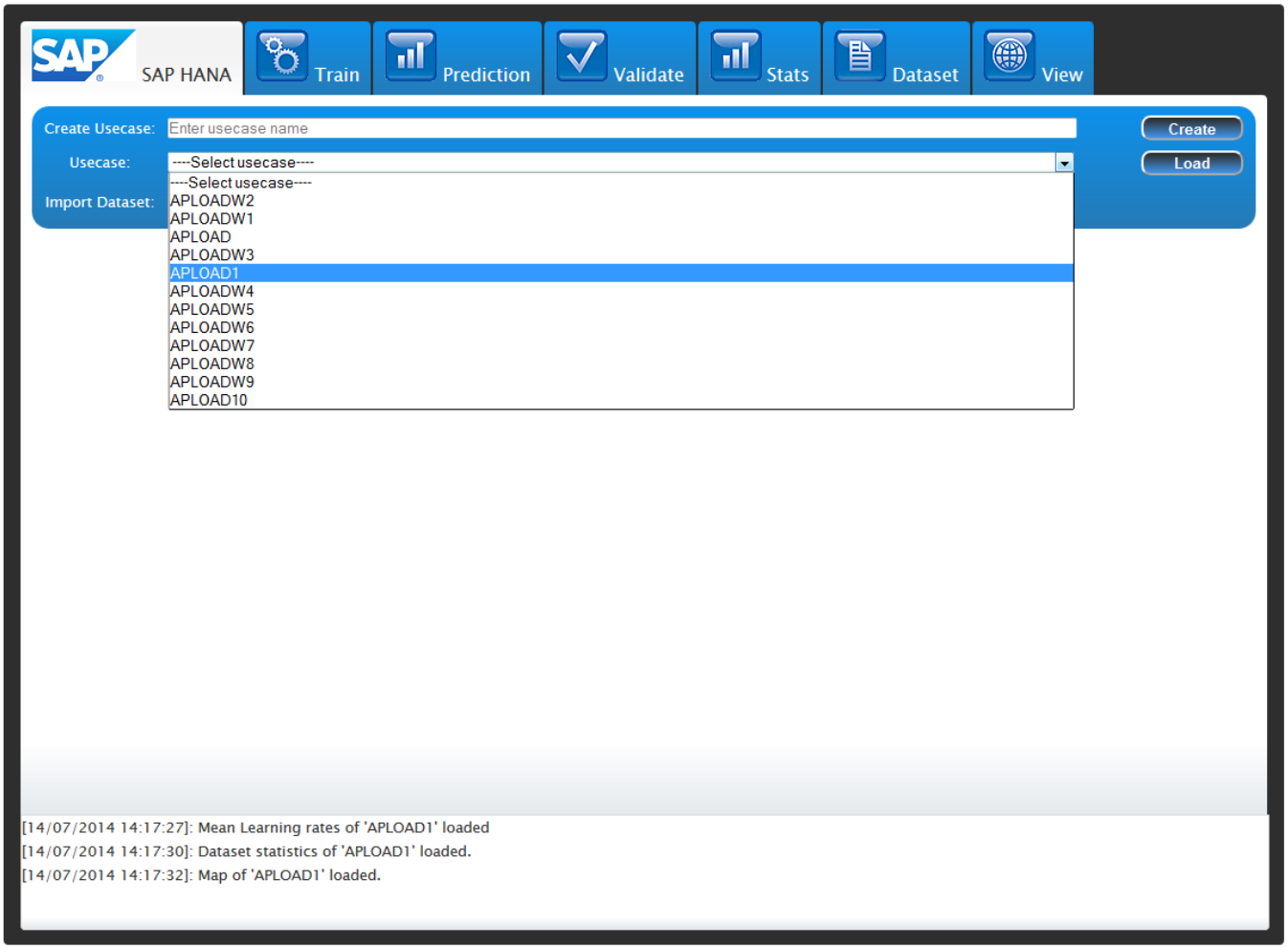

Figure 20: Upload Dataset to use-case - part 1

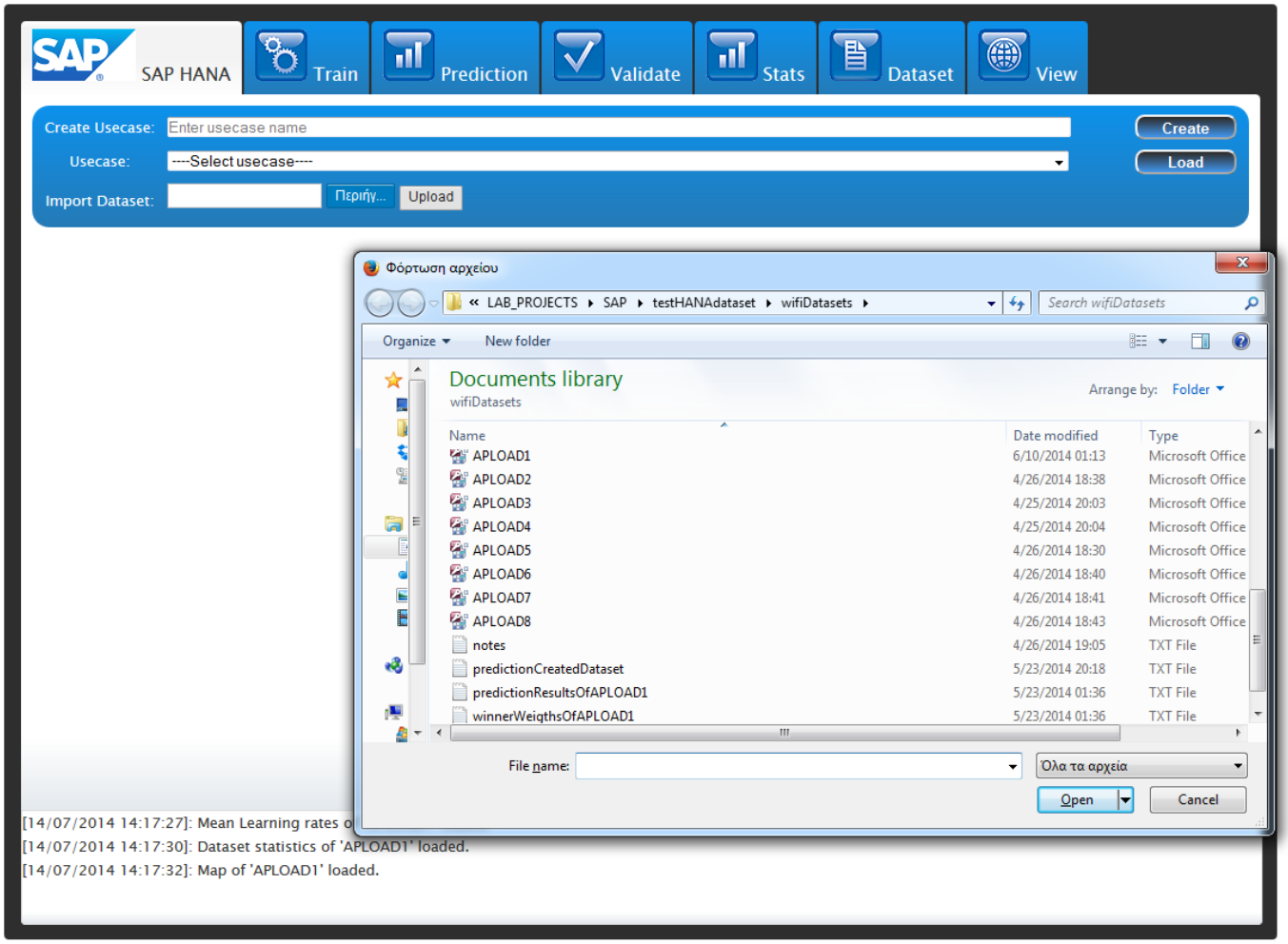

Figure 21: Upload Dataset to use-case - part 2

Once the user presses the button "Upload", the KGV tool sends an HTTP request to the SAP HANA application server. The request is manipulated by the "uploadUsecase.xsjs" service (see section 4.2.3). After the upload of dataset has been finished, the KGV tool informs the user about it (Figure 21).

5.2.2 The Training Process

The training process is related with the training of the Parameter-less Growing Self-Organizing Maps algorithm that we described in Chapter 2. The KGV tool provides an appropriate Web Interface in order to initialize and configure the training process.

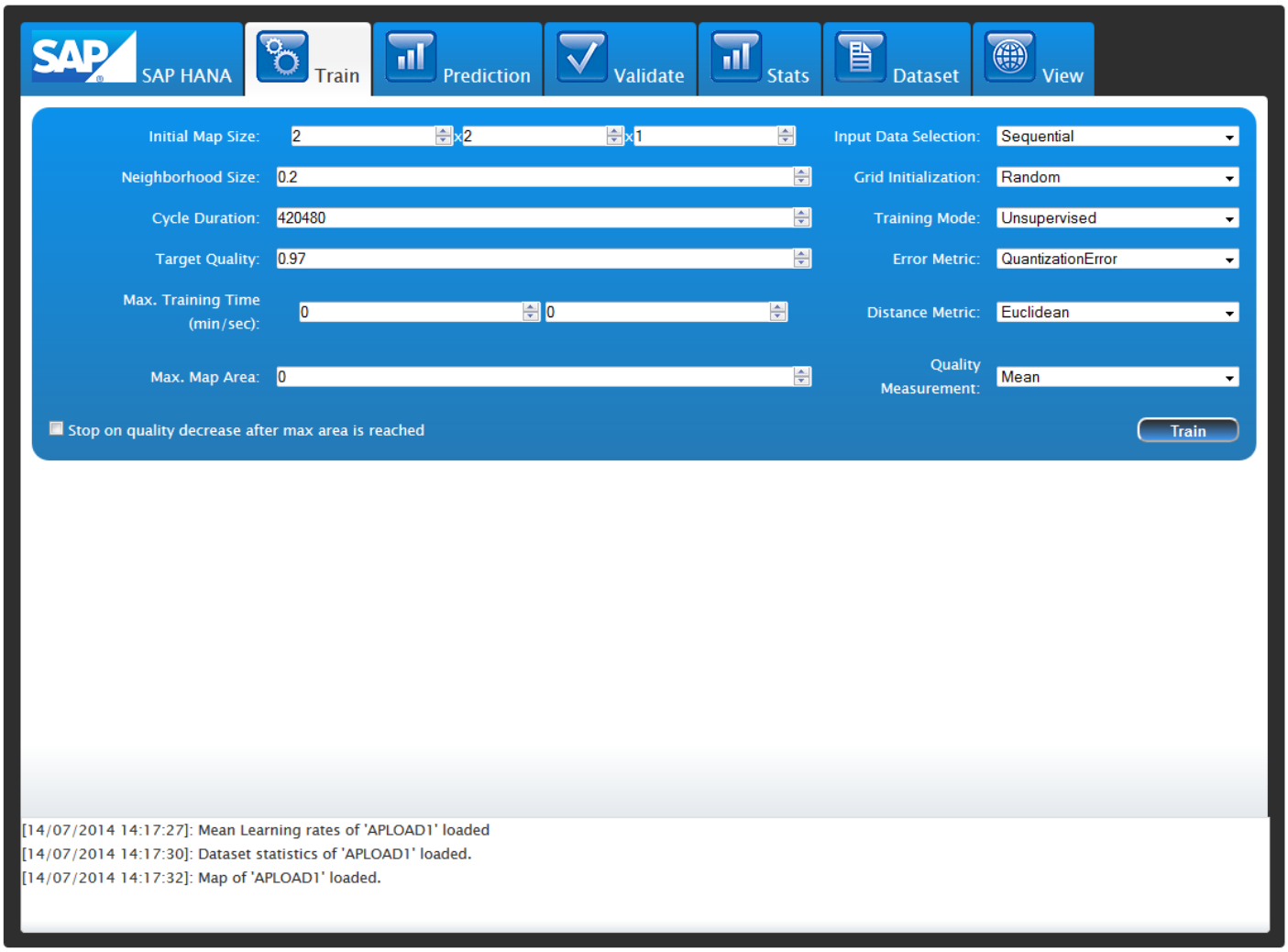

Figure 22: Training Process in KGV Tool

After the creation of the use-case and once he has uploaded a dataset to the SAP HANA database on the related use-case, he selects the use-case in the "SAP HANA" tab and, then, he proceeds to the "Train" tab of the KGV tool User Interface. The Figure 22 illustrates the main configuration parameters which are needed to be defined in order to begin the training process through the KGV tool. In the following table (see Table 1), we provide details about each parameter/option of the tool.

Parameter	Description
Initial Map Size	This parameter represents the initial dimensions of the SOM map.
Neighborhood Size	This parameter represents the size of the neighborhood that we want the PLGSOM

	algorithm to take into account in the estimation of the applying of the neighborhood function that we described in Chapter 2. Essentially, this parameter contributes to the size of the BMU neighborhood (see section 2.3.2).
Cycle Duration	This parameter represents the duration of one epoch. In particular, it represents the number of the iterations that are performed before measuring the quality of the SOM map. Essentially, this parameter is initialized with the number of the vectors in the given dataset of the specified use-case.
Target Quality	This parameter represents the quality that the SOM map is desired to satisfy. As we mentioned in Chapter 2, the target quality is the reverse of the SOM map error.
Maximum Training Time	This parameter represents one of the termination criteria that the user could set to the PLGSOM algorithm in order to finalize the training process. More specifically, this parameter sets the time interval in which the PLGSOM algorithm will perform the training process.
Maximum Map Area	This parameter represents one of the termination criteria that the user could set to the PLGSOM algorithm in order to finalize the training process. In particular, the user could set this parameter in order to finalize the training process of the PLGSOM algorithm after the specified map area is exceeded.
Input Data Selection	This parameter represents the way that the PLGSOM algorithm selects the vectors from the dataset of the use-case. The tool provides three ways of vector selection: • Random: This option guides the algorithm for random vector selection. • Sequential: This option guides the algorithm for selecting the vectors of the dataset sequentially. • Mixed: This option guides the algorithm to select the vectors in a mixed way. This means that the

	algorithm will scramble the dataset and ,then, it will select the vectors in a sequentially way.
Grid Initialization	This parameter represents the way that the initial units of the SOM map will initialize their vector of weights. The tool offers two options of grid initialization: • Random: This option guides the algorithm to initialize randomly the units' weights. In particular, the units' weights will take values from the set of values that the dataset of the use-case provides. • Gradient: This option guides the algorithm to initialize the units' weights gradiently. The algorithm takes into account the up-left corner of the SOM map with the minimum values (i.e. the dataset values) and, it ends up to the right-down corner with the greatest values (i.e. the dataset values).
Training Mode	This parameter represents the training mode that the PLGSOM algorithm will follow in order to train the specified use-case. The tool offers two options regarding the training mode: • Unsupervised learning: This option guides the algorithm to ignore the labels (i.e. the last feature in the dataset) of the vectors of the dataset of the specified use-case during the training process. • Supervised learning: This option guides the algorithm to take into account the values of the label in the dataset during the training process of the specified use-case.
Error Metric	This parameter represents the error measurement method that the algorithm will apply to the quality measurement of the training process. The tool provides three options regarding the error measurement: • Quantization Error: We provide

	detailed description in section 2.3.2 . • Mean Quantization Error: We provide detailed description in section 2.3.2 . • Classification Error: We provide detailed description in section 2.3.2
Distance Metric	This parameter represents the distance measurement method that the algorithm will apply in procedures such as, the Best-Matching-Unit procedure, the quality measurement of the SOM map etc. The tool offers the following options for distance measurement: • Euclidean distance: We provide detailed description in section 2.3.1 . • Squared Euclidean distance: This method represents the squared Euclidean distance.
Quality Measurement	This parameter represents the way that the algorithm will follow in the quality measurement of the SOM map. The tool provides the following options for this process: • Mean: This option guides the algorithm to assume and estimates the mean error of all the units of the SOM map during the training process of the specified use-case. • Minimum: This option guides the algorithm to assume and estimates the greatest error of all the units of the SOM map during the training process of the specified use-case.

Table 1: Configuration parameters of the PLGSOM algorithm.

So, after the user sets the above configuration parameters of the PLGSOM algorithm, he presses the "Train" button. An HTTP request is sent to the application server that is hosted on the SAP HANA Platform. This HTTP request is manipulated by the 'trainUsecase.xsjs' service. This service interacts with the SAP HANA database in order to call and to initialize the R module in the R host side (see Figure 17 in section 4.2) which contains the PLGSOM algorithm.

Once the execution of the training process is finished, the Web User Interface of the KGV tool presents a message in the console at the bottom of the screen: "Training process finished

successfully!". Thus, the user is capable to load the use-case, to proceed to prediction and validation processes and generally view visualized results about the training process, such as the mean learning rate, the learning rate etc.

5.2.3 The Prediction Process of a Network Load Scenario

After the user has trained a use-case such as a network load use-case, he is capable of performing the prediction process that the KGV tool offers. More specifically, in "Prediction" tab, the user can would first select a use-case, e.g. "APLOAD1". Then, the available Access Points are shown at the top of the main panel, from where the user may choose the ones whose load is to be predicted. The next step would be to specify the (time) interval in question. So, for example, we prompt the KGV tool for a load prediction during the week starting from Tuesday up to Wednesday, with a time interval of 15 minutes in-between consecutive predictions for access points 7 and 0. The results are depicted in a chart of Load over Time for the specified access points (Figure 23 & Figure 24).

As we would expect, the load during the small hours is a lot less than the load during the day and the working hours. We can see that the system has learnt this pattern, by experience, and without any human assistance. For AP0 on July 15th at 10:45, we see that the predicted load is low enough and, so the network provider can release some resources. On the other hand, for AP7, the load is predicted to be higher, thus, the network provider should allocate more resources there.

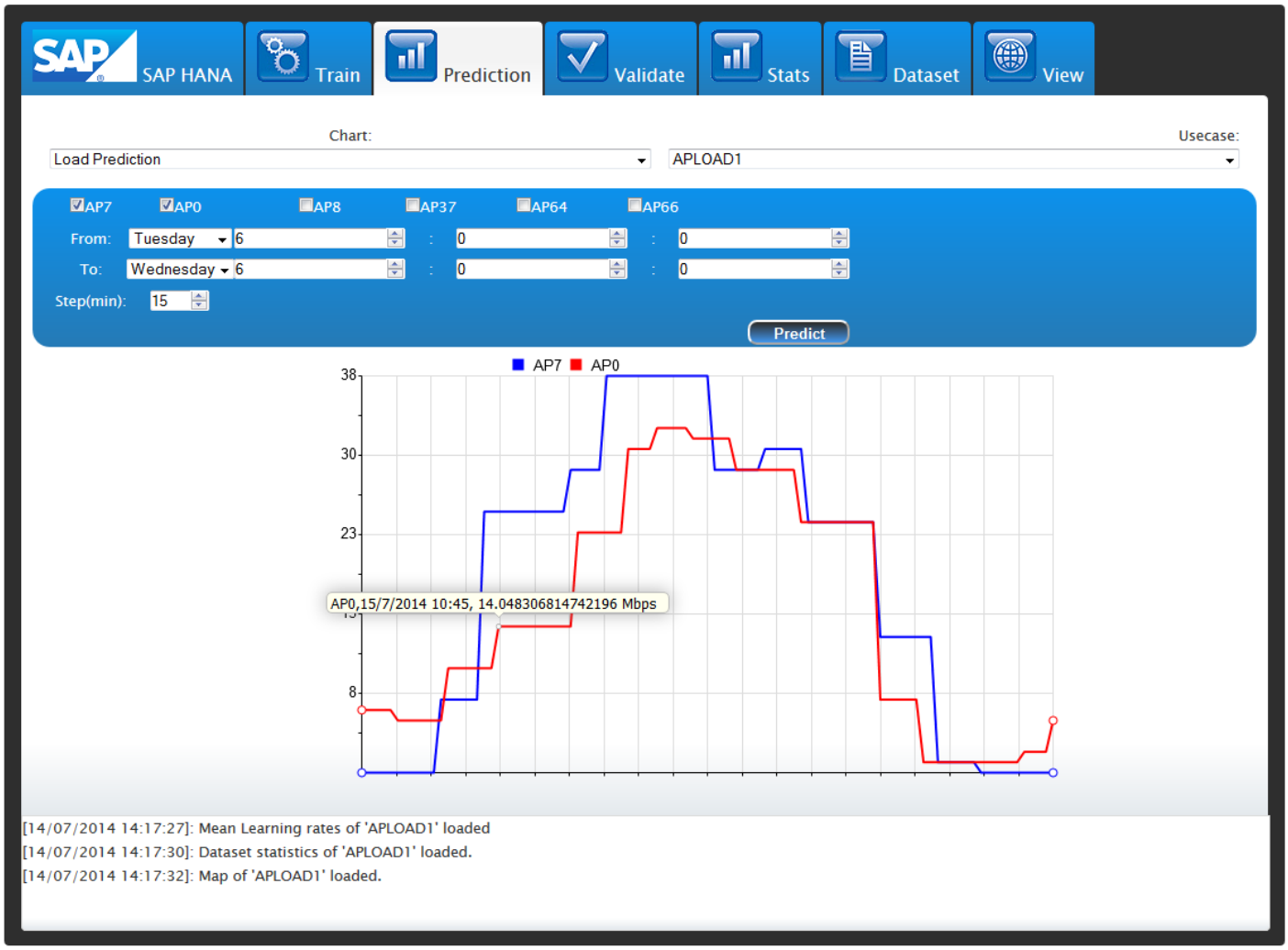

Figure 23: Prediction process of the KGV tool - part 1

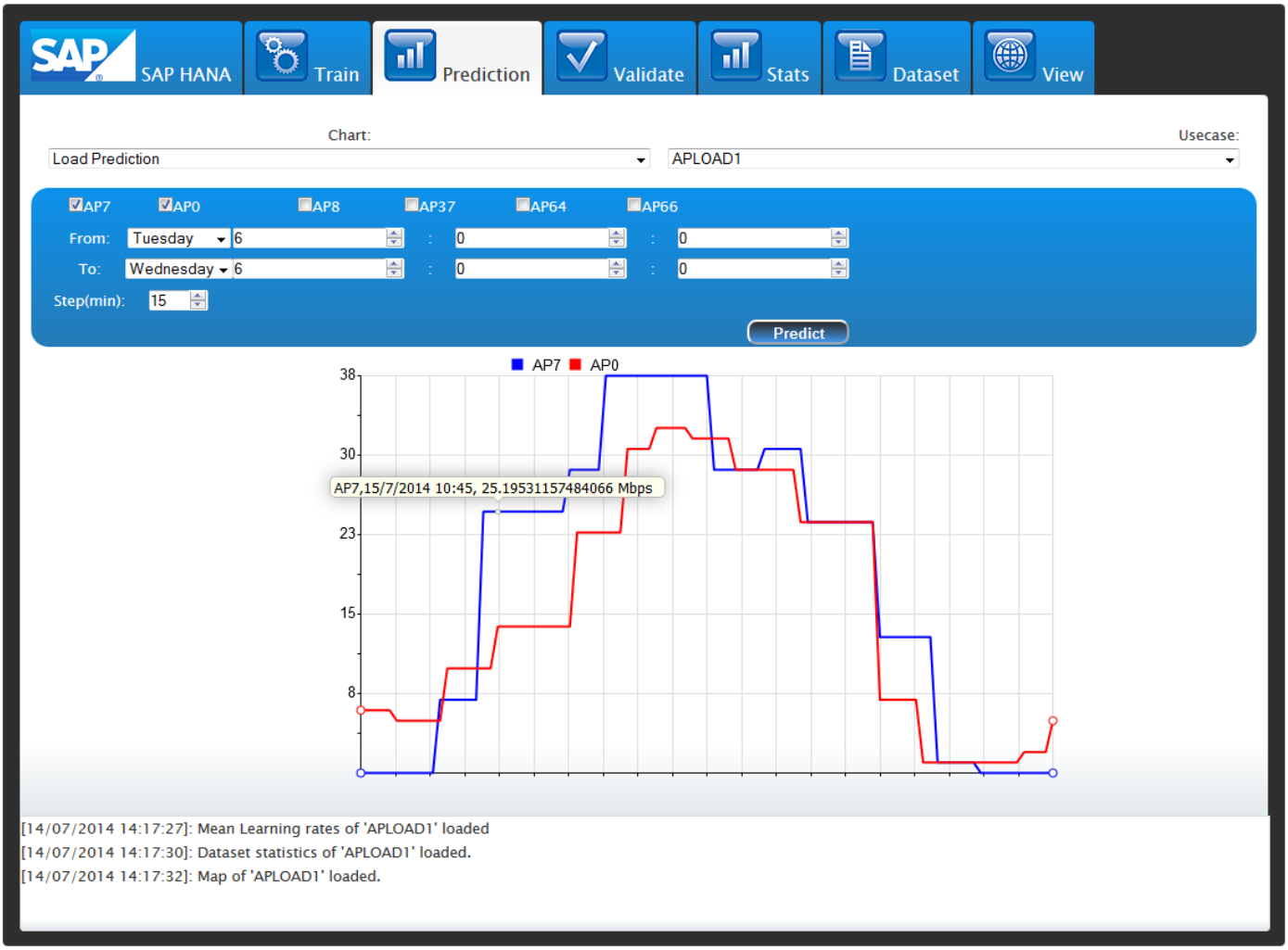

Figure 24: Prediction process of the KGV tool - part 2

Now , let's choose a different example, for example, APLOADW9. As we can see, in the main panel we have the same parameters to set plus temperature and precipitation. This means that the selected use-case has been trained with temperature and precipitation in mind. So, similarly, we fill in the form by requesting a load prediction for AP7 and 0 from Thursday 20:00:00 PM to Friday 20:00:00 PM, a time interval of 15 minutes, a temperature of 15 degrees of Celsius and a precipitation of 10 litres per square meter.

In this case, at "AP7" on Wednesday between 9:15 AM and 10:00 PM we expect a load of approximately 10 Mbps . Accordingly, we can see the prediction results for AP0. (Figure 25 & Figure 26).

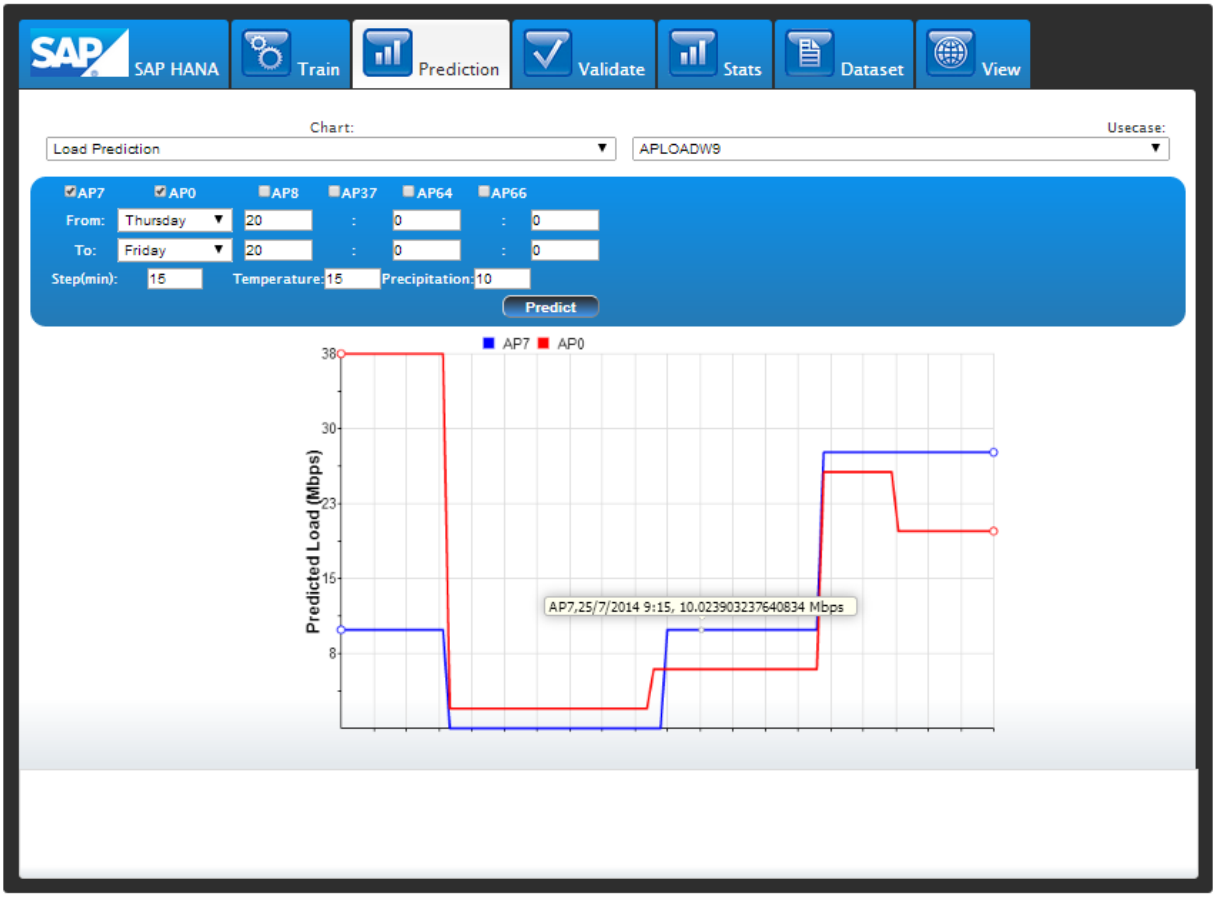

Figure 25: Prediction process of the KGV tool including weather parameters - part 1

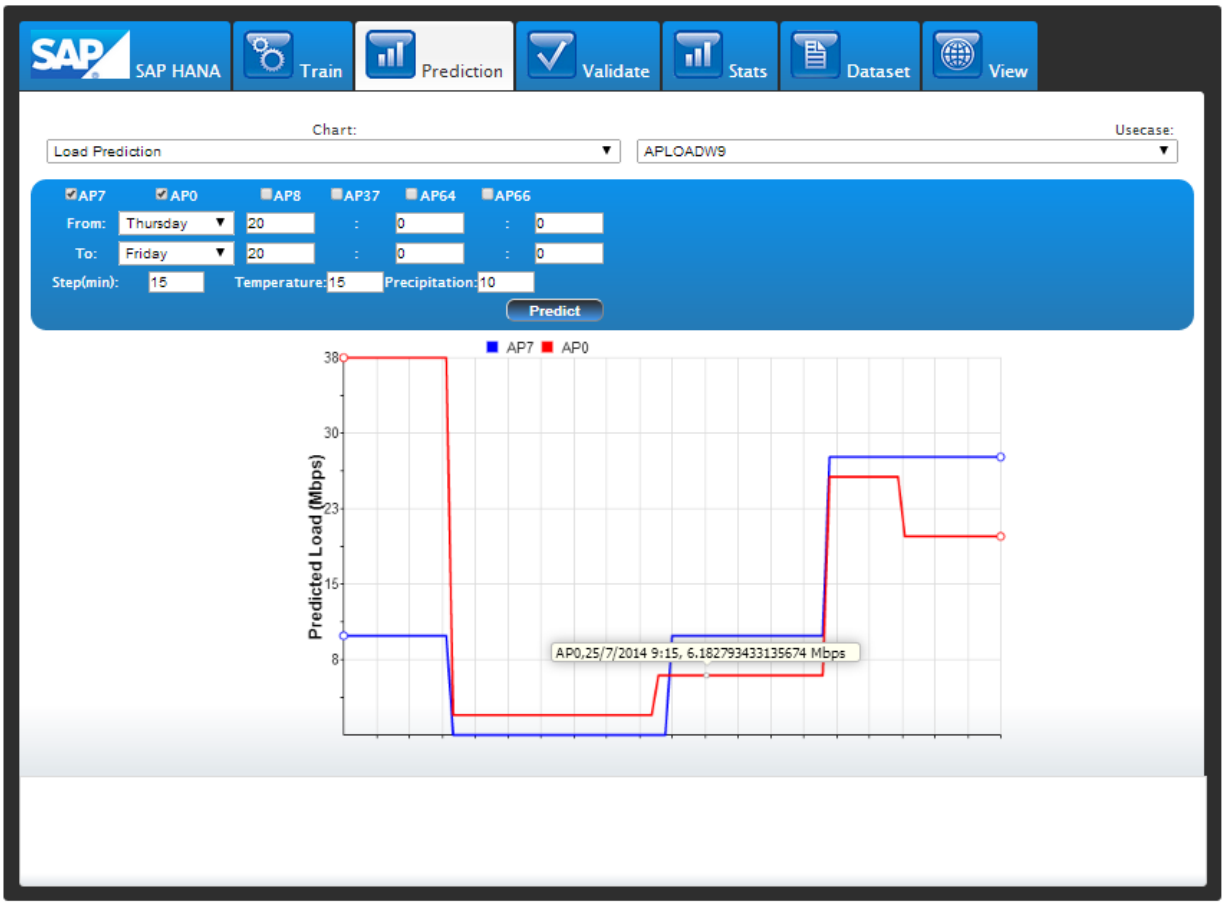

Figure 26: Prediction process of the KGV tool including weather parameters - part 2

5.2.4 The Validation Process

The KGV tool provides the functionality of validating the trained use-case. Essentially, the user provides a validation dataset regarding a specified use-case and the results of validation are illustrated in a graph of SOM vs Data. This graph compares the label of the provided validation dataset with the estimated results of the SOM map. This process is managed at the "Validate" tab.

At first, the user selects a use-case for validation (Figure 27) and, then, he specifies the validation dataset (Figure 28). After that, he selects the method with which the validation process will proceed to validate the SOM map. For now, the KGV tool provide three validation methods:

- The Best Matching Units
- The Best Matching Units (Weighted)

- Nearest Winner's Average

The description of these validation methods are provided at the section 2.4 . In our case, we selected the Nearest Winner's Average (Figure 29). As you can see in Figure 30, when we selected the Nearest Winner's Average method, the KGV tool shown in section "Options" of the Web Interface the element "Distance Metric". This is a configuration parameter of the method which is used by the validation method in order to be prompted for the distance measurement method to use. In particular, the distance metric is used by the validation method in order to find the BMU with the minimum distance in the SOM map for each vector of the validation dataset. The options of distance metric are:

- Euclidean distance
- Squared Euclidean distance
- Manhattan

Further details of the distance metrics are provided at Table 1 on section 5.2.2 .

In Figure 31: Validation Process - part 5 we provide an example of the validation process. This example results to a Mean Squared Error (MSE) of 0.000009.

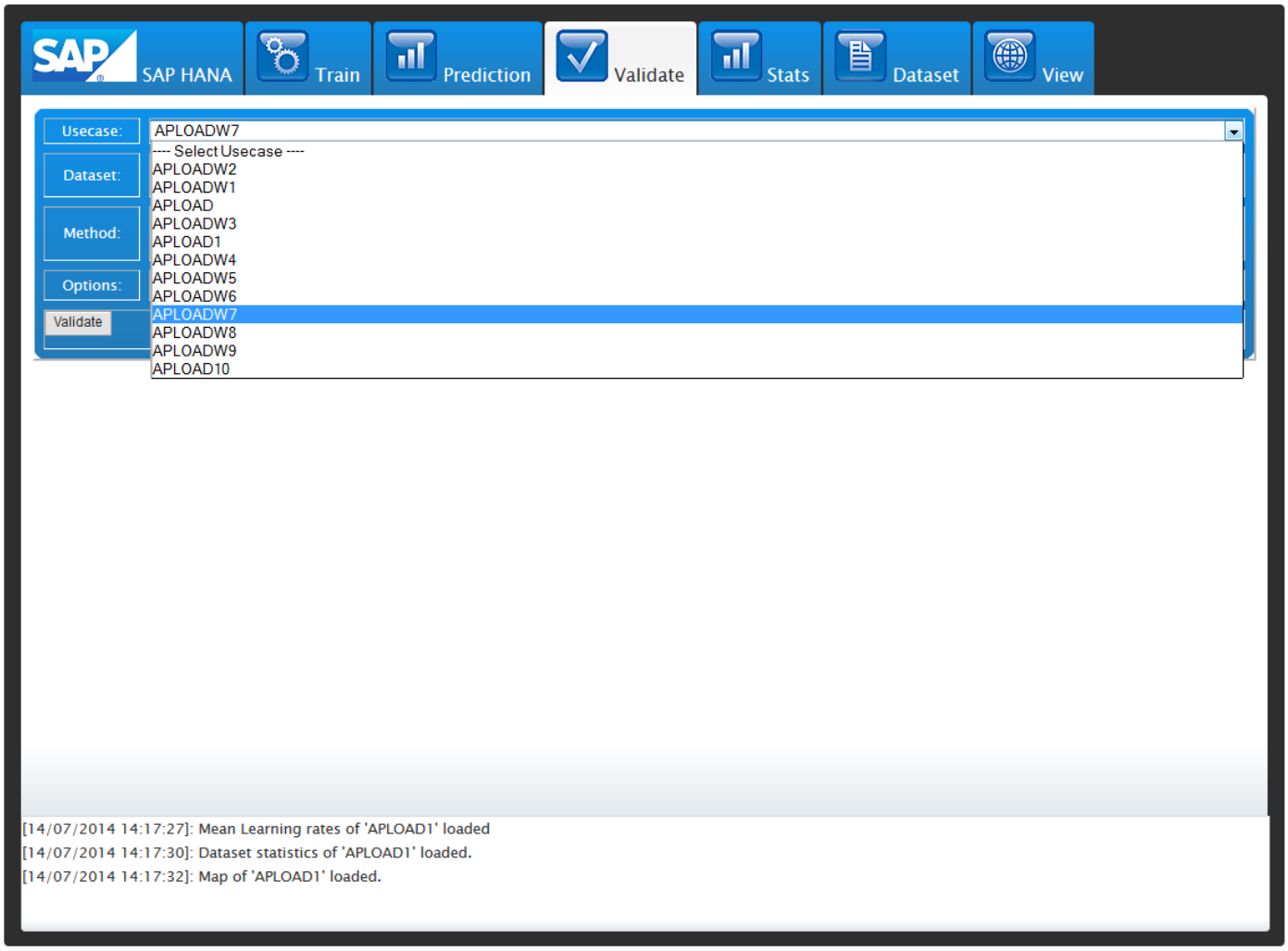

Figure 27: Validation Process - part 1

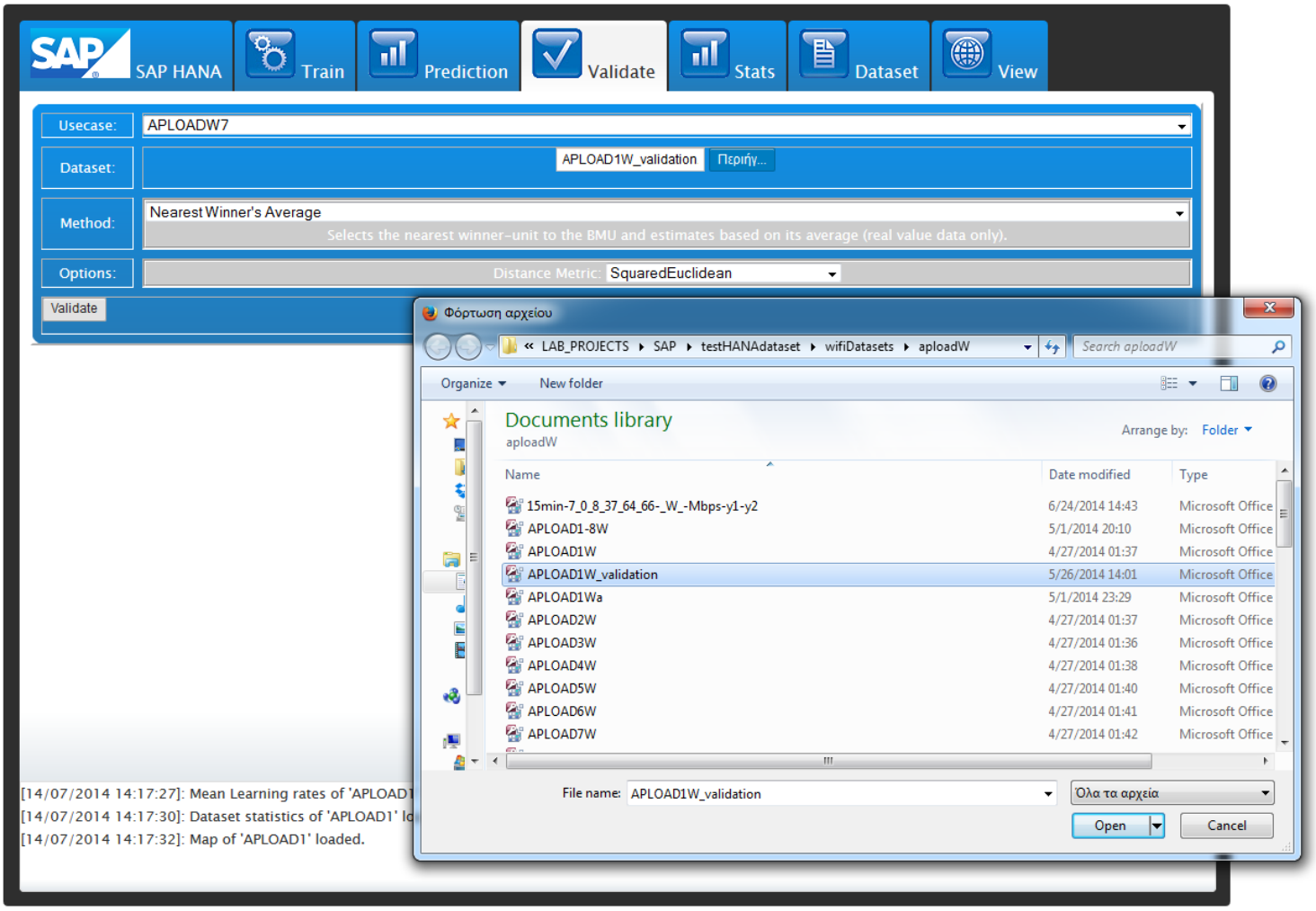

Figure 28: Validation Process - part 2

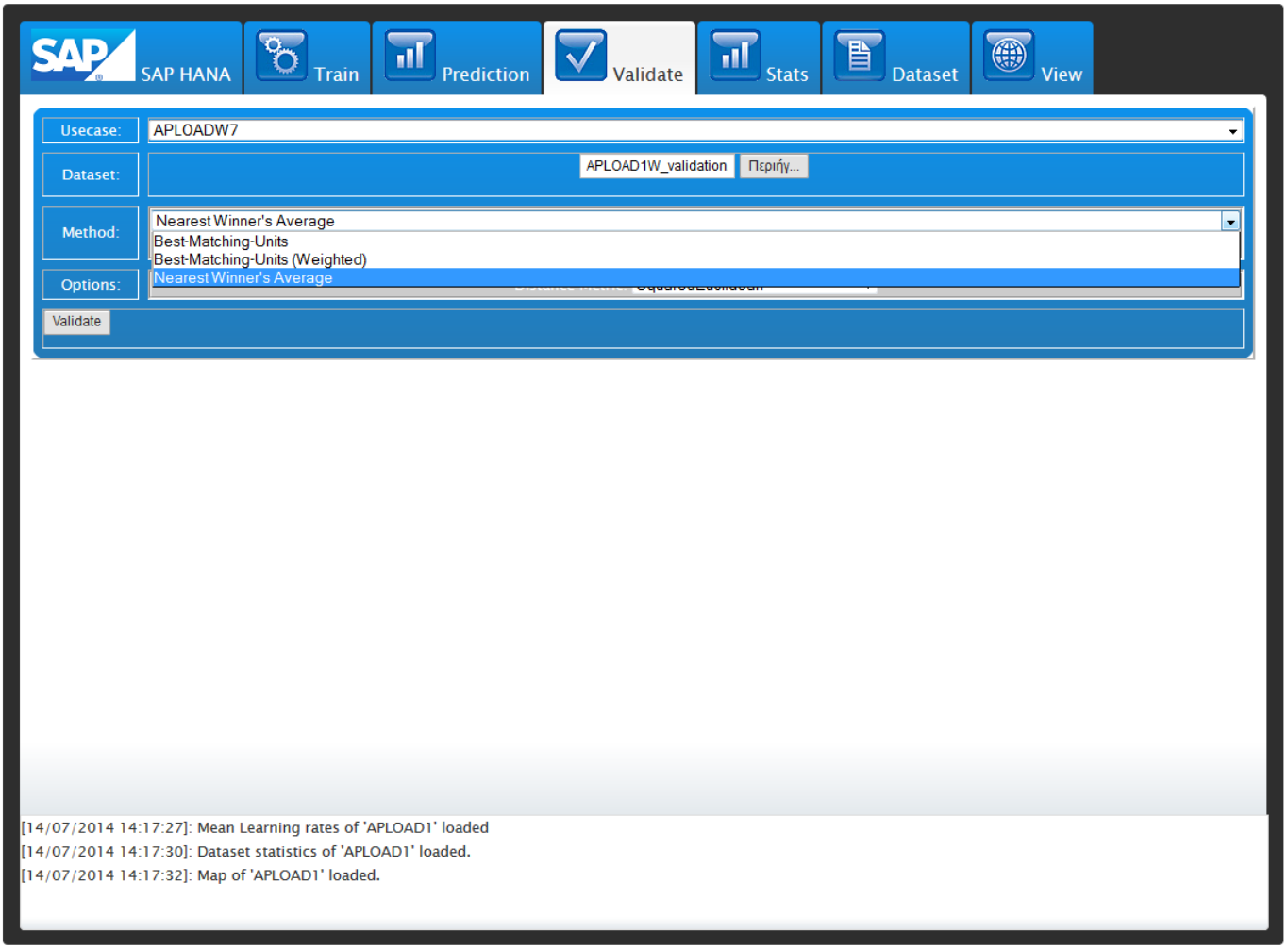

Figure 29: Validation Process - part 3

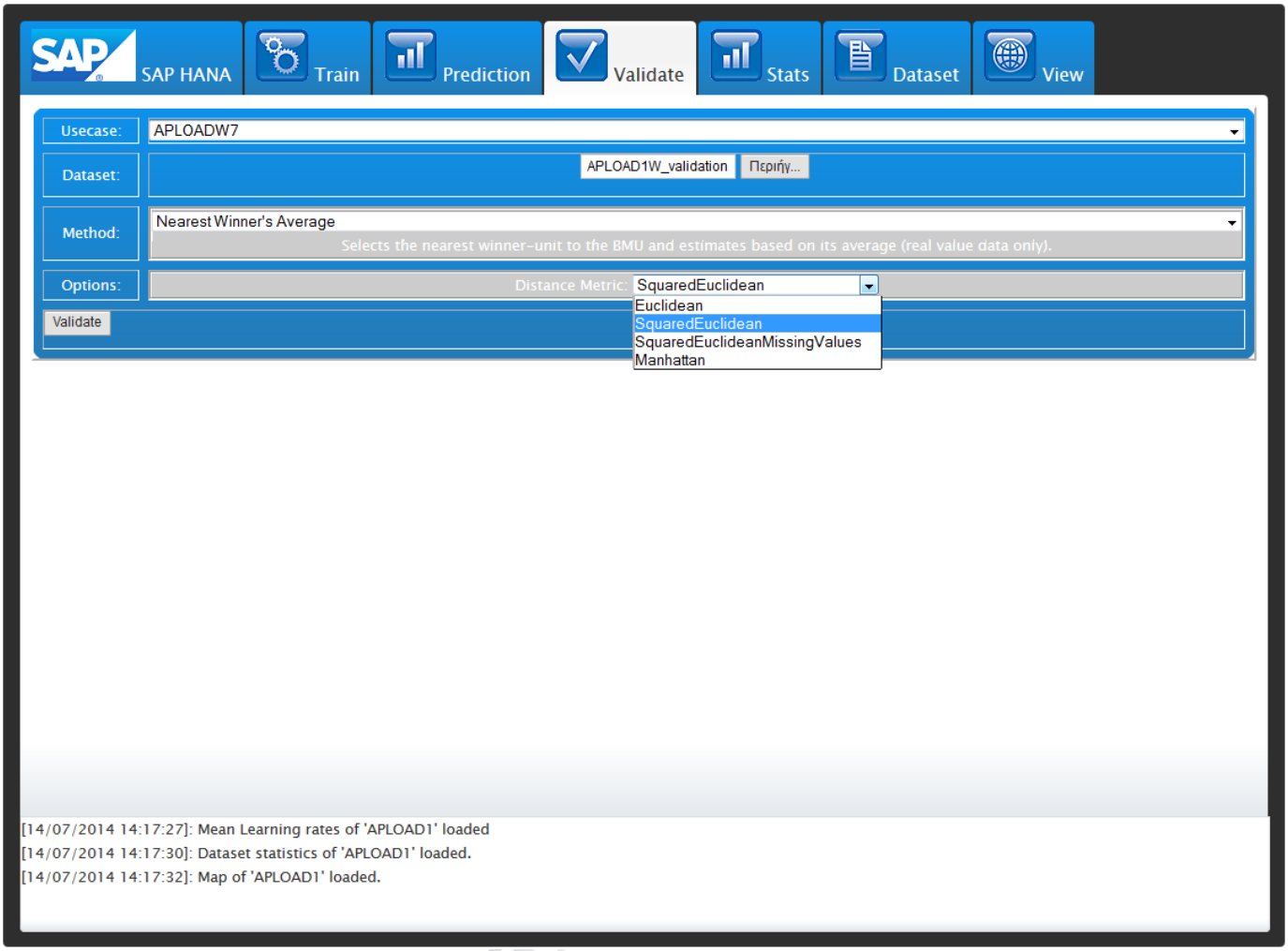

Figure 30: Validation Process - part 4

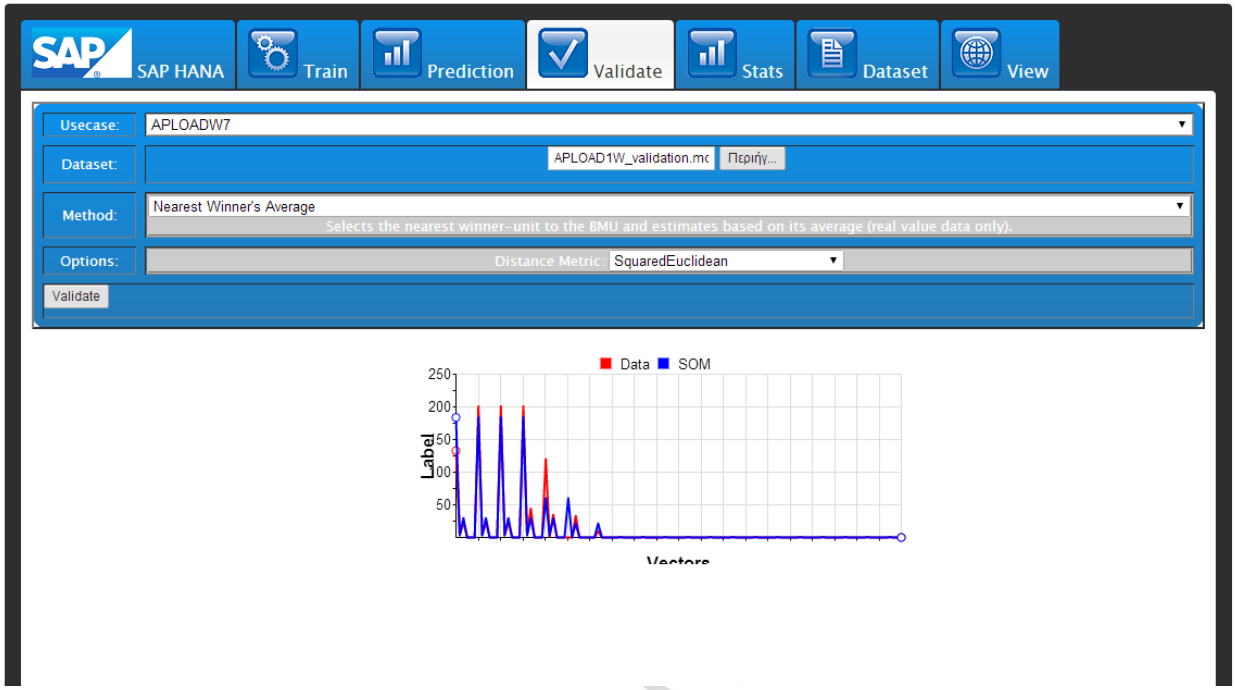

Figure 31: Validation Process - part 5

5.2.5 Visualization and Provision of use-case statistics

The KGV tool gives the opportunity to the user for monitoring the training process statistics through graph diagrams. In particular, in the "Stats" tab, the user can observe some statistics results regarding the learning rate, the mean learning rate and component vs the label.

For the purposes of demonstrating statistics and results of the training process, the user has to select a use-case and , afterwards, press the button "Load" at the "SAP HANA" tab of the KGV tool. Once the user presses the button "Load" an amount of HTTP requests are sent to the application server of the SAP HANA platform. The requests are related with:

- The statistics of the training process of the specified use-case.
- The statistics of the training dataset of the specified use-case.
- The statistics of the SOM map that has been produced by the training process of the specified use-case (see section 5.2.6).

As far as the first bullet, the KGV tool sends asynchronously three HTTP requests which are related with the learning rate, the mean learning rate and component vs the label. The first HTTP request is manipulated by the "getLearningRates.xsjs" service of the SAP HANA application server. This service interacts with the SAP HANA db in order to return the learning rates of the last epoch of the training process of the specified use-case (see section 4.2.3). In addition, as far as the second HTTP request, it is manipulated by the

"getMeanLearningRates.xsjs" service of the SAP HANA application server. Similarly, this service interacts with the SAP HANA db in order to get the mean learning rates of the training process regarding the specified use-case (Figure 32) (see section 4.2.3).

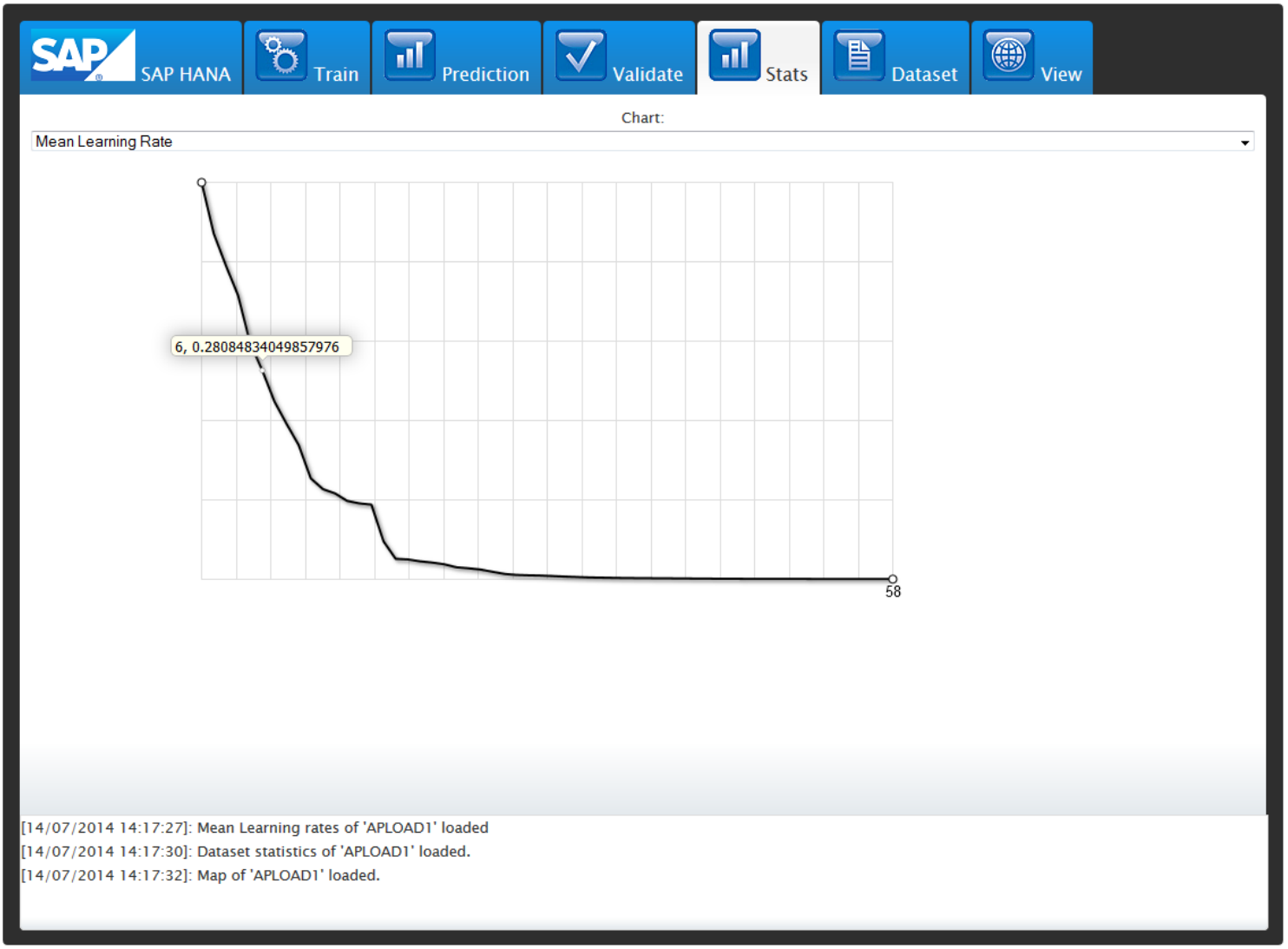

Figure 32: Statistics tab of the KGV tool - part 2

As far as the second bullet, the KGV tool sends an HTTP request which is manipulated by the "getDatasetInfo.xsjs" service of the SAP HANA application server. This service interacts with the column table "DATASETINFO" in order to retrieve the statistics which are related with the training dataset of the specified use-case. In particular, we talk about the following statistics, the minimum values, the maximum values, the quantization error of the dataset, the mean

quantization error of the dataset, the mean values, the deviation, the variance etc (see section 4.2.3). In Figure 33, we illustrate how the "Dataset" tab demonstrates all the above statistics which are related with the training dataset of the specified use-case.

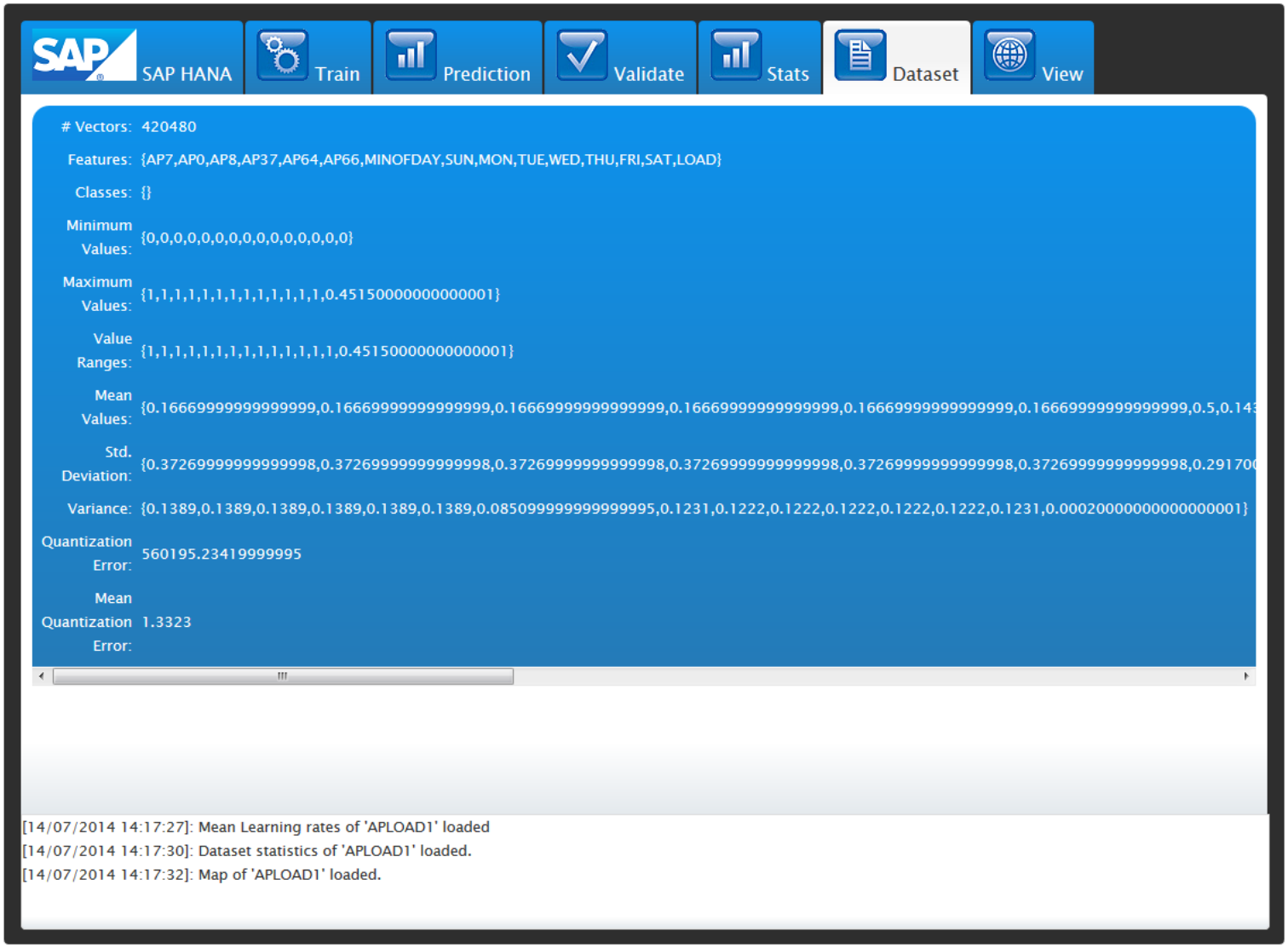

Figure 33: Dataset tab of the KGV tool.

As far as the third bullet, further details are provided in section 5.2.6.

5.2.6 Visualization of the SOM map

As we have observed from the previous section, when the "Load" button of the "SAP HANA" tab is pressed, a number of HTTP requests are sent to the application server of the SAP HANA Platform where a number of services are triggered from them in order to retrieve data asynchronously and to be demonstrated to the user of the KGV tool. In this section, we'll describe the third bullet of the section 5.2.5 which is related with the statistics of the SOM map that is constructed by the training process .

In this case, the KGV tool sends a HTTP request to the application server. This request is manipulated by the "loadUsecase.xsjs" service. This service is responsible for the retrieval of the SOM map of the specified use-case. In particular, it returns the units of the map, its weights, the names of the features of the use-case and the statistics for each unit of the SOM map, such as the quantization error, the deviation, the average of the label values of mapped inputs etc. In Figure 34 & Figure 35, the KGV tool demonstrates all the above information in the form of a 2D map for the use-case "APLOAD1".

In Figure 34, we have selected the unit on row 11 and column 8. On the left side of the screen, the user can observe the statistics of the specified unit, such as the average value, the standard deviation, the mean absolute deviation, the mean quantization error, the amount of mapped inputs and the vector with the weights of the unit. Similarly, in Figure 35, we can see the corresponding statistics for the unit on row 0 and column 14. As we can see in both cases, there is the menu option of the "Legend". This option illustrates the values of the minimum and the maximum labels of the SOM map and their corresponding colors.

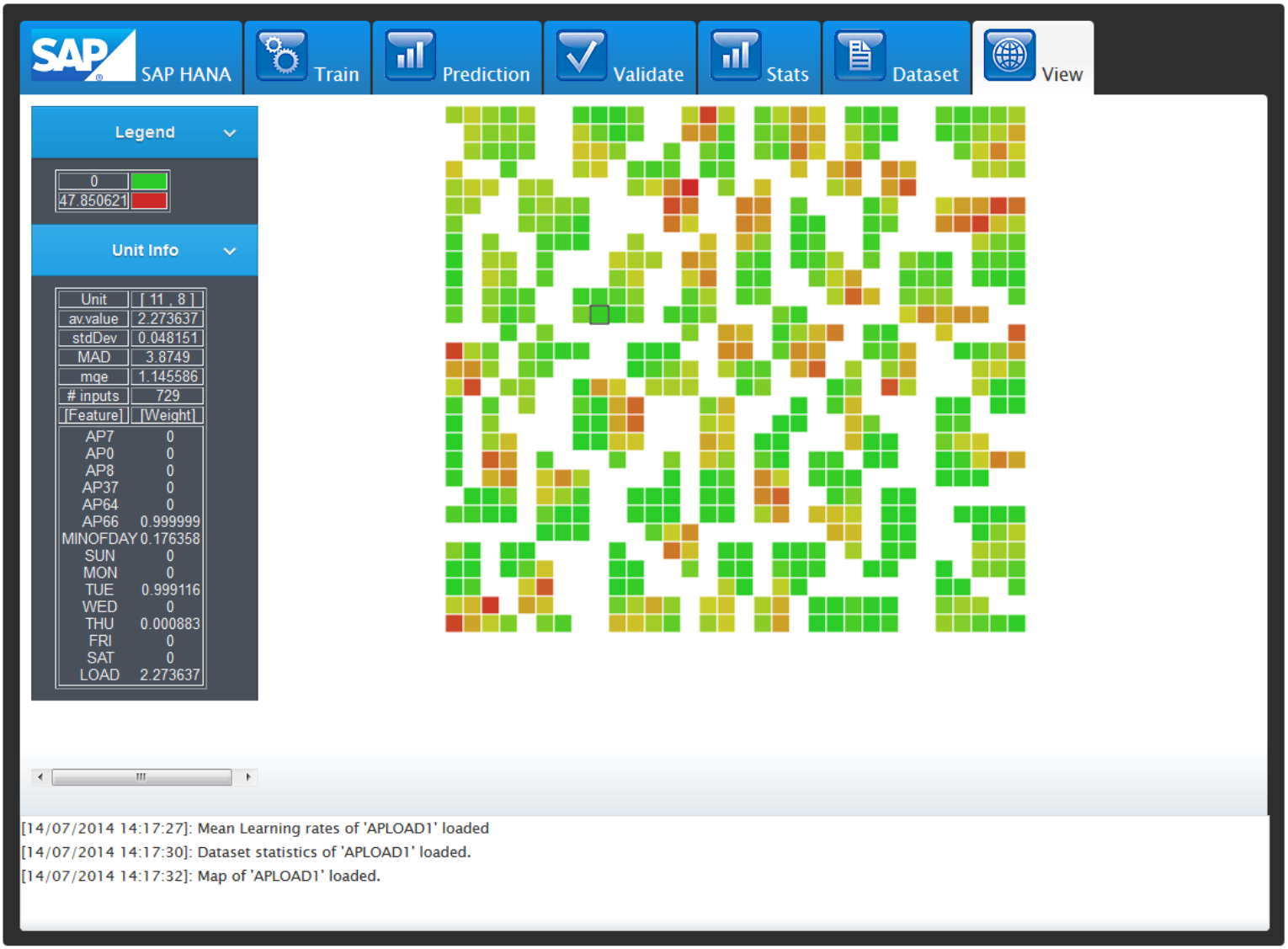

Figure 34: The View tab in the KGV tool - part 1

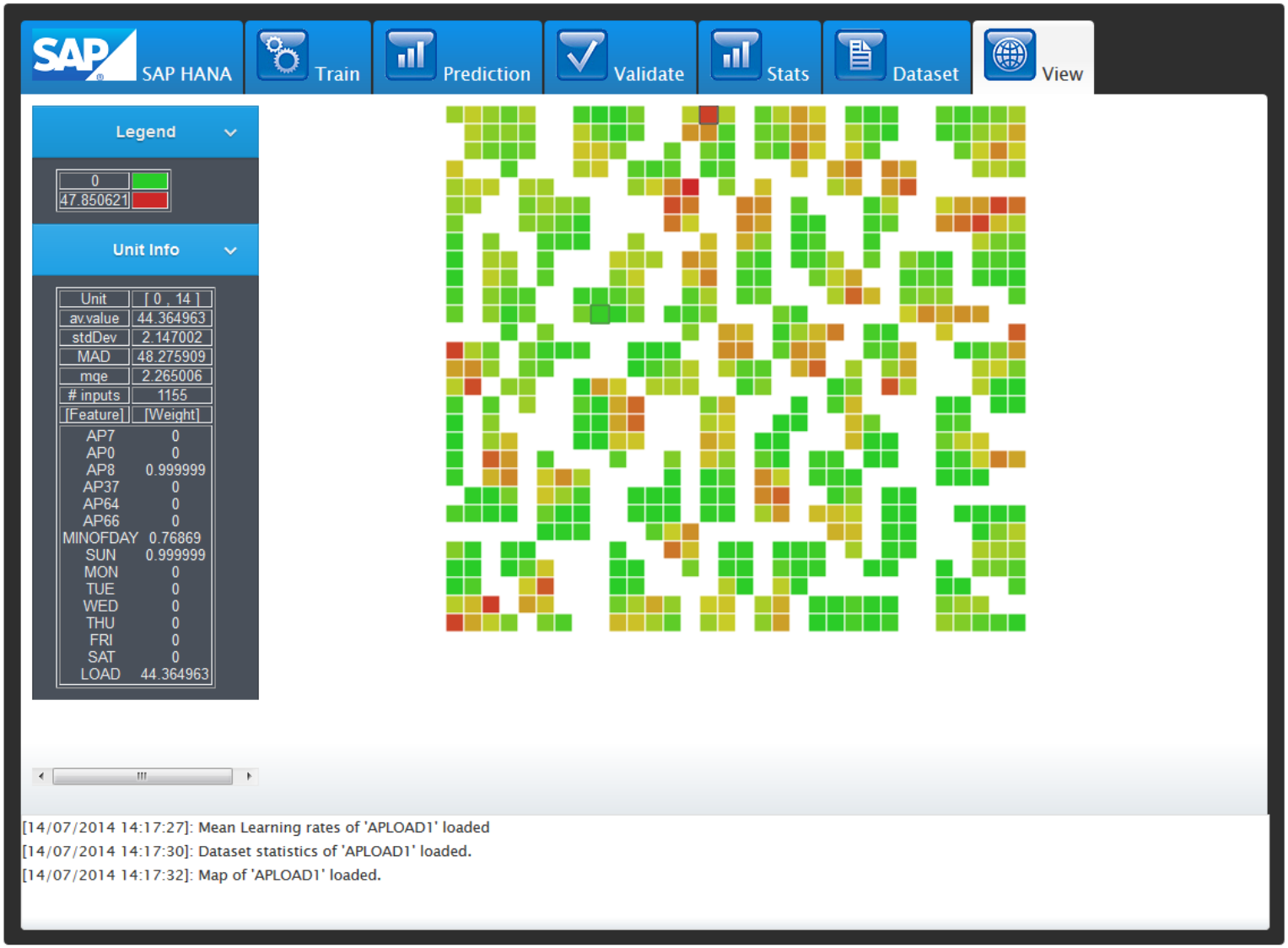

Figure 35: The View tab of the KGV tool - part 2

5.3 Indicative Results from the training process of PLGSOM algorithm

In the previous section, we discussed a scenario related with the usage of the tool and we illustrated results regarding the prediction of future states of the network load. In this section, we provide some indicative results regarding the training process of the PLGSOM algorithm provided by the Knowledge Generation and Visualization (KGV) tool.

Generally, all the scenarios took into account as feature parameters six Access Points (APs) with values (0,1). The value '1' represents that the AP is loaded, while the value '0' that the AP hasn't network load. Moreover, each scenario had as features the days of the week (i.e. Sunday - Monday) with values (0,1). Also, the minute of day, on which the measurement that the a vector represents, had values (0 - 0.95). The values of the minute of day were normalized. Finally, the load of each vector is represented by the last feature of the dataset. This feature is, also, the label of the dataset and it is normalized.

Some scenarios took into account the weather parameters as features in the training process. In particular, these scenarios has as feature parameters the temperature and the precipitation. Their values are normalized.

The Table 2 illustrates the results of three indicative scenarios. The first scenario has a SOM map of 928 cells. In addition, it is configured with a Neighborhood size of 0.2, the distance metric was chosen to be the 'Squared Euclidean Metric' and the error metric was chosen to be the 'Quantization Error'. As we can see, the Mean Squared Error (MSE) for this scenario is 0.000000586. So, the RMSE is 0.00076 (i.e. 2.51 Mbps).

Similarly, in the second scenario we provide the aforementioned information. The cells of the SOM map are 930. In addition, it is configured with a Neighborhood size of 0.2, the distance metric was chosen to be the 'Euclidean Metric' and the error metric was chosen to be the 'Quantization Error'. This scenario takes into account the weather parameters. Essentially, the Mean Squared Error (MSE) is 0.00001325. So, the RMSE is 0.00364 (i.e. 11.95 Mbps).

As far as the third scenario, the cells of the produced SOM map are 930. In addition, it is configured with a Neighborhood size of 0.05, the distance metric was chosen to be the 'Squared Euclidean Metric' and the error metric was chosen to be the 'Mean Quantization Error'. This scenario takes into account the weather parameters. Finally, the Mean Squared Error (MSE) is 0.007125.

The fourth scenario includes a SOM map with 5,040 cells. As configuration parameters this scenario includes a Neighborhood size of 0.2, a distance metric of 'Squared Euclidean Metric' and an error metric of 'Mean Quantization Error'. As before, this scenario contains as features the weather parameters. Essentially, the Mean Squared Error (MSE) is 0.00000127. So, the RMSE is 0.0011 (i.e. 3.7 Mbps).

ID	Unseen Validation Data	#Cells	Weather	#Vectors	Neighborhood Size	Distance Metric	Error Metric	Mean Squared Error (MSE)
1	No	928 (i.e. 29x32)	No	420,480	0.2	Squared Euclidean Metric	Quantization Error	0.000000586
2	No	930 (i.e. 30x31)	Yes	420,480	0.2	Euclidean Metric	Quantization Error	0.00001325
3	No	930 (i.e. 30x31)	Yes	129,582	0.05	Squared Euclidean Metric	Mean Quantization Error	0.007125
4	No	5,040 (i.e. 70x72)	Yes	4,032	0.2	Euclidean Metric	Quantization Error	0.00000127
5	No	2200 (i.e. 40x55)	Yes	28,800	0.2	Squared Euclidean Metric	Mean Quantization Error	0.000004
6	Yes	2200 (i.e. 40x55)	Yes	28,800	0.2	Squared Euclidean Metric	Mean Quantization Error	0.000024
7	No	567 (i.e. 27x21)	No	32,280	0.2	Squared Euclidean Metric	Mean Quantization Error	0.000019
8	Yes	567 (i.e. 27x21)	No	32,280	0.2	Squared Euclidean Metric	Mean Quantization Error	0.0000057
9	No	615 (i.e. 15x41)	No	210,240	0.2	Squared Euclidean Metric	Mean Quantization Error	0.00021
10	Yes	615 (i.e. 15x41)	No	210,240	0.2	Squared Euclidean Metric	Mean Quantization Error	0.0002
11	Yes	780 (i.e. 39x20)	No	420,480	0.2	Squared Euclidean Metric	Quantization Error	0.0005
12	Yes	4526 (i.e.73x62)	Yes	420,480	0.2	Euclidean Metric	Quantization Error	0.0006

Table 2: PLGSOM training algorithm results

In the case of the scenario 3, we notice that it probably suffers from under-fitting which means that we should wait maybe for 4/or 5 (or more) training cycles/epochs to be executed before stopping the training process in order to fulfill a better MSE. Similarly, the knowledge engineer should be careful about the 'over-train' phenomenon while he performs the training process. Moreover, every scenario should be generalized enough after the training process in order to provide sufficient prediction results. All the aforementioned notes should be considered on every scenario that the KGV tool will perform.

For the purposes of providing a well-generalized SOM map, we validated some scenarios with unseen validation dataset. This kind of validation is indicated by the column "Unseen Validation Data" in Table 2. For the scenarios (i.e. 6, 8, 10, 11, 12), that the column "Unseen Validation data" indicates "Yes", we have validated them with datasets different from the training dataset. In particular, we performed these tests in different cases with different kind of configuration and amount of data in order to observe the efficiency of the PLGSOM training algorithm in unseen data.

For example, in scenario 10, we used as training dataset the network load measurements of the first year and as validation dataset the network load measurements of the second year (i.e. MSE = 0.0002). Similarly, in scenario 11, we used as training dataset the network load measurements of the first and the second year and as validation dataset the measurements of the third year (i.e. MSE = 0.0005). The scenario 12, is a similar case with scenario 11 plus it includes, in its design, the weather parameters (i.e. MSE = 0.0006).

As we can observe in these scenarios (i.e. 6, 8, 10, 11, 12) the Mean Squared Error (MSE) is low too. Therefore, we can conclude that the SOM maps of these scenarios are generalized enough and, thus, they are capable to provide efficient network load predictions.

5.4 KGV tool advancements provided by the SAP HANA Platform

As far as, the load of the training process results (i.e. the SOM map), the choice of the SAP HANA Platform with the advanced techniques in storage technology is an ideal option for such real-time applications. In particular, when the KGV tool requests the SOM map (i.e. it contains the statistics of each cell of the map), the SAP HANA Platform responds fast by retrieving the appropriate data from the database, packaging them into a JSON format and, finally, sending them back to the client (i.e. the KGV tool). As we can understand, in this case, there is a classical overhead which resides in the server-side.

We know that the tasks of searching and retrieving data from the database are obstacles with a bad time complexity (i.e. increased time retrieval as the amount of data get bigger) for every system. On the other hand, the SAP HANA Platform manipulates this kind of complexity and it behaves scalable as the data increased. This is fulfilled by exploiting the storage technologies that applies, such as compression of data etc. (i.e. section 3.2) and the other capabilities of the fast Calculation Engine of the database.

The initialization of the training process is performed by calling the SQL procedure "<NAME_OF USECASE>_TRAIN " (see section 4.2.1). Afterwards, as we have described in previous chapters, the SAP HANA Platform exploits the Calculation Engine and the "R Interface" in order to retrieve the data and send them to the "R module" which performs the training process. As we can understand, the obstacle of retrieving the dataset and migrating it to the component which performs the calculation logic is surpassed. In particular, these tasks are accelerated because of the usage of the Calculation Engine and the sophisticated "R Interface". So, the acceleration of the above processes is fulfilled because of:

- the fast data searching/retrieval performed by the Calculation Engine and,
- the fast mapping of a SQL Column table to a data structure understandable from the component which performs the calculation logic (i.e. R module). In our case, the data structure is called, "data.frame" (see section 3.2.3).

It's worth to mention here, that the conversion of a SQL Column Table into a "data.frame" is convenient because their representation is similar. Traditionally, in desktop/web applications we used connectors such as JDBC, ODBC etc. to communicate with databases and retrieve data. These connectors add further processing overhead in order the data to be ready to be used by the applications through APIs. So, we gain from the "R Interface", especially, as the amount of data get larger.

In addition, the SAP HANA Platform accelerates the retrieval, searching, manipulation and demonstration of data because it applies a unified solution (i.e. unified software) regarding the server-side. As we have seen in section 3.2 the SAP HANA Platform provides an application server close to the database (i.e. these consist a unified software) (see Figure 5). So, the server is close to the data. Traditionally, the web applications applied solutions such as LAMP(Linux, Apache, MySQL, PHP), WAMP etc. which weren't unified as one as the SAP HANA Platform is. So, usually, we had a larger processing gap between the storage and the processing logic.

5.5 Conclusions

The "Big Data" phenomenon demands the design of technologies and mechanisms that can store, manipulate, process and retrieve the disparate data in order to, generally, provide meaningful information to users and stakeholders in real-time responses. As far as the telecommunication networks field, the operators collect huge amount of data for long periods and for multiple network nodes.

The network operators should get meaningful information from the large amount of the collected historical data. In particular, they need to get information, most of the times in real-

time, about the future network load of their infrastructure in order to proceed to the crucial resource management and planning. This work proposed a mechanism which is capable of exploiting disparate data in order to build knowledge and it is built upon the SAP HANA software platform which highlights all the "Big Data" aspects. As a result, the developed mechanism provide real-time predictions of the network load which, as we mentioned above, are important about the decision making of the telecommunication network enterprises. In general, the developed tool was evaluated enough and, thus, we revealed that it is capable of learning the traffic pattern with deviations up to 0.0002 in terms of MSE. Finally, the heavy-load tasks of storing, retrieving, manipulating, searching and processing the huge amount of data is supported by the SAP HANA Platform which, essentially, decreases the response times.

6 References

[1] A. Bantouna,G.Poulios, K.Tsagkaris, P. Demestichas, "Network Load Predictions Based on Big Data and the Utilization of Self-Organizing Maps", Springer, September 2013

[2] European Committee for Standardization Website. http://www.cen.eu/cen/News/Newsletter/Pages/default.aspx.

[3] Teuvo Kohonen, "The Self-Organizing Map" ,Proceedings of the IEEE, vol. 78, no. 9, September 1990

[4] T. Kohonen, The Self-Organizing map, Elsevier, Neurocomputing 21 1-6, 1998

[5] J. Vesanto, J. Himberg, E. Alhoniemi, J. Parhankangas, SOM Toolbox for Matlab 5. Technical Report A57, Neural Networks Research Centre, Helsinki University of Technology, Helsinki, Finland, 2000

[6] A.Rauber, D.Merkl, M.Dittenbach. The Growing Hierarchical Self-Organizing Map: Exploratory Analysis of High-Dimensional Data. *IEEE Transactions on Neural Networks.* Νοέμβριος 2002, Τόμ. 13, 6, σσ. 1331-1341.

[7] T.Kuremoto, T.Komoto, K.Kobayashi, M.Obayashi. Parameterless-Growing-SOM and Its Application to a Voice Instruction Learning System. *Journal of Robotics.* Τόμ. 2010.

[8] E.Berglund, J.Sitte. The parameterless Self-Organizing Map Algorithm. *IEEE Transactions on Neural Networks.* 2006, Τόμ. 17, 2, σσ. 305-316.

[9] "Analysis of SAP HANA: High Availability Capabilities", An Oracle White Paper, June 2013

[10] "SAP HANA Master Guide", SAP HANA Platform SPS06, v1.1-17-07-2013, SAP AG, 2013

[11] "SAP HANA Developer Guide", SAP HANA Platform SPS07, v1.0-27-11-2013, SAP AG, 2013

[12] " SAP HANA R Integration Guide", SAP HANA Platform SPS07, v1.0-27-11-2013, SAP AG 2013

[13] R, D. Peng, Jan de Leeuw, "An Introduction to the .C Interface to R" , 2002

[14] The R Development Core Team, "Writing R Extensions", v2.6.0 (2007)

[15] W. N. Venables, D. M. Smith and the R Core Team, "An Introduction to R - Notes on R: A Programming Environment for Data Analysis and Graphics", v3.0.2 (2013)

[16] SpiderMonkey(software), link: http://en.wikipedia.org/wiki/SpiderMonkey_%28software%29

7 Table of Figures