Carlos Rijo Bate

Computer Science Degree

Ubiquitous Hierarchical Self-Organizing Map inspired by Financial Data Taxonomy - A reactive approach to HSOM based on web services

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: Nuno Cavalheiro Marques, Assistant Professor,
NOVA University of Lisbon - FCT

Co-adviser: Carmen Morgado, Assistant Professor,
NOVA University of Lisbon - FCT

Examination Committee

Chairperson: Maria Cecília Gomes
Raporteurs: Luís Cavique
Nuno Cavalheiro Marques

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

June, 2017

Ubiquitous Hierarchical Self-Organizing Map inspired by Financial Data Taxonomy - A reactive approach to HSOM based on web services

This document was created unsing the (pdf)LaTeX processor, based in the "unlthesis" template[1], developed at the Dep. Informática of FCT-NOVA [2]. [1] https://github.com/joaomlourenco/unlthesis [2] http://www.di.fct.unl.pt

Abstract

Nowadays we have access to an almost immeasurable amount of data that needs to be processed and shown in a friendly way so people can easily read it and take their own conclusions, moreover, there is a significant amount of data that is streamed in real-time. Even though there are situation where we want to get insights from the past data, there are other scenarios where we would like to analyze it in real-time like the stock market, which is an unpredictable and constantly-changing field.

In this thesis a solution is proposed to achieve an advanced data exploration tool that uses machine learning and distributed computation techniques in order to find relations in the hierarchical data in a flexible, adaptable, decentralized and available manner. A Self-Organizing Map variant (Ubiquitous SOM) shall be used as building blocks of the Hierarchical SOM for such goal resulting in a Ubiquitous Hierarchical SOM model, the UbiHSOM. This thesis aims to develop a set of microservices that analysts, data scientists and developers can use for their own needs, following their own policies to achieve their demands and customizing taxonomy accordingly. For instance, the stock market products can be divided into various sectors, hence following a certain hierarchical taxonomy. A similar taxonomy can also be identified in other fields, therefore the same solution could also be used to extract knowledge from those fields.

Keywords: UbiSOM, HSOM, UbiHSOM, data stream correlations, microservices

Resumo

Hoje em dia temos acesso a uma quantidade imensurável de dados que precisam de ser processados e mostrados ao utilizador de forma simplificada, para que este consiga analisá-los com facilidade e tirar as suas próprias conclusões. Embora existem situações onde pretendemos extrair conhecimento de dados passados, existem outros cenários que apelam à extração de conhecimento através de dados que chegam em tempo real, por exemplo, a bolsa de valores, que é um sector imprevisível e em constante mudança. Nesta tese é proposta uma ferramenta de exploração de dados que utiliza técnicas de aprendizagem automática e sistemas distribuídos de modo a encontrar relações em dados hierárquicos de uma maneira flexível, adaptável, descentralizada e disponível. Para encontrar tais relações uma variante dos Mapas Auto-Organizados (Ubiquitous SOM) irá ser utilizada como base de um modelo hierárquico de mapas auto-organizados (Hierarchical SOM) para alcançar o modelo Ubiquitous Hierarchical SOM (UbiHSOM). Esta tese tem também como objecto desenvolver um conjunto de microserviços que utilizadores como, e.g., analistas,data scientists e programadores podem utilizar para satisfazer as suas necessidades e personalizar as correlações como desejarem de forma a atingirem os objectivos pretendidos aplicando as suas políticas e customizando a taxonomia consoante o problema em mão. Um exemplo disso será a bolsa de valores onde os produtos podem ser divididos em vários sectores dando assim origem a uma taxonomia hierárquica, que para além da bolsa de valores, também pode ser identificada uma taxonomia semelhante noutras áreas.

Palavras-chave: UbiSOM, SOM, correlações em streams de dados, microserviços

Contents

List of Figures

Acronyms

ANN Artifican Neural Network.

API Application Programming Interface.

AWS Amazon Web Services.

BMU best matching unit.

BSON Binary JSON.

DBMS Database Management System.

HSOM Hierarchical Self-Organizing Map.

HTTP Hypertext Transfer Protocol.

JSON JavaScript Object Notation.

JVM Java Virtual Machine.

REST Representational State Transfer.

SaaS Software as a Service.

SOM Self-Organizing Map.

SQL Structured Query Language.

UbiHSOM Ubiquitous Hierarchical Self-Organizing Map.

UbiSOM Ubiquitous Self-Organizing Map.

URI Unified Resources Identifiers.

XML eXtensible Markup Language.

Chapter 1

Introduction

1.1 Motivation

Nowadays we have access to an almost immeasurable amount of data that needs to be processed and shown in a friendly manner so people can easily read it and take their own conclusions about it. The financial world is one of the biggest data providers. It is an example of a real-world scenario that requires a large tool set to analyze high volumes of data, thus, time people (e.g., brokers, economists, investors, analysts, data scientists) waste on reaching conclusions should be minimized in order to improve performance. The faster the analysts find a correlation between financial data the better, because it increases their decision-making time leading to a more solid and supported decision. Hence it would be interesting if they have a tool that could find those correlations. For instance, machine learning techniques were used by the European Central Bank [54] in order to map the state of financial stability, visualize the sources of systemic risks and predict systemic financial crises.

Typically large volumes of data are unstructured, which makes the data hard to process and hard to analyze. Moreover, the data itself can be n-dimensional, making it even harder for humans to visualize and analyze it. Even though some companies have put some effort into organizing information in a more "easy-to-cope-with" way, it is still hard to analyze it and identify more complex correlations. This problem complexity increases if the analysis and visualization has to be treated in a real-time context, pointing towards the need of a tool that enables real-time large scale data analysis and visualization.

Cognitive computing[35] means enabling machines to interact with humans in a more natural way. It is a problem-solving approach that uses hardware or software to approximate the form or function of natural cognitive processes: learning, perception

and motivation. The motivation is the purpose that guides both learning and perception. Learning is what the system must do, i.e., the system improves based on the experience rather than programming. Perception is how the data is acquired, i.e., where the data that is used in the learning process in order to increase experience comes from. Cognitive computing enables the creation of computing systems that can solve complex problems without constant human oversight. Machine learning is an artificial intelligence discipline geared toward the technological development of human knowledge. Machine learning allows computers to handle new situations via analysis, self-training, observation and experience. Typically machine learning algorithms can be classified as being supervised or unsupervised, where supervised algorithms learn how to predict the output from the input data, whereas unsupervised algorithms will try to find the structure or relationships between different inputs. Among many others, Self-Organizing Map (SOM) is an unsupervised-learning clustering algorithm with advanced visualization capabilities used to project and visualize high-dimensional data on a two dimensional map. SOMs fall under the cognitive computing category.

The volume of unstructured information is growing at an enormous rate at various fields, including the stock market. The stock market is a place where securities are bought and sold. Securities are commonly stocks, bonds, warrants or options, where a stock shows that his holder owns a percentage of a certain company.

Usually people that invest in the stock market tend to have a selected portfolio. A portfolio is a collection of financial products that should not have high related products because the breakdown of one might influence the others, one with a wide variety of products is desired. So other products will compensate for the ones that collapse. SOM can be used with financial data to identify clusters and correlations between financial products. Even though the information is typically unstructured, in sectors like the stock market, the information follows a certain taxonomy, which can be advantageous because the tools used to analyze it can be more specific in order to improve the quality of the conclusions.

The financial field is also a very motivating example, since it is one of the most active data providers, across its various sectors, where the stock market is a big data producer and a sector that plays an important role on the field. Moreover, it highly benefits from computational aid to analyze and visualize data and find correlations between it as well. The correlation between financial market prices are relevant because when we invest in a set of related products, if one collapses, probably all collapse. If somehow those correlations could be extracted from data and presented (in a friendly way) for further analysis it would save analysts time, thereby potentially increasing their performance.

1.2 Problem Description and Objectives

A SOM variant called Ubiquitous Self-Organizing Map (UbiSOM) (see section 2.5) can cope with this high volume of data in real-time[58], furthermore it was already used

to analyze financial data[58]. The current UbiSOM is a stand-alone solution, however, nowadays developers and data scientists are relying more and more on web-services and Software as a Service (SaaS) cloud-based solutions [17]. Moreover, SOM variants with a static architecture (e.g. UbiSOM) translates into limited capabilities for the representation of hierarchical relations of the data. In order to analyze the hierarchical relations of the data there are SOM variants that follow a hierarchic architecture instead of a static one, namely, Hierarchical Self-Organizing Map (HSOM). A hierarchical architecture allied with a solution that explores web-services and UbiSOM capabilities, leads to a more decentralized and available solution that hopes to open the boundaries of developers and data scientists analytic capabilities by widening their tool set. In this thesis the Ubiquitous Hierarchic Self-Organizing Map (Ubiquitous Hierarchical Self-Organizing Map (UbiHSOM)) is proposed as being a HSOM variant that combines HSOM, UbiSOM and web-services capabilities. Hence, this thesis aims to:

- adapt a SOM variant called UbiSOM (see section 2.5) to be used in a web-service approach;
- build a flexible and adaptable system that offers a web service solution for UbiSOM and UbiHSOM to cope with real time data analysis;
- study a way to use UbiHSOM to find correlations in the data;
- test the usability of this model (UbiHSOM) to analyze a financial data set taxonomy (section 2.7).

Previous results in SOM ([39], [48]), UbiSOM [60] and HSOM [29] and *GoBusiness* research database [65] will be taken into account.

1.3 Proposed Solution

This thesis proposes a set of services that enable developers and data scientists to build UbiSOM based solutions in order to extract knowledge from unstructured data. Such solutions includes an HSOM variant - the UbiHSOM - that uses UbiSOM as its building-blocks. This variant aims to take advantage of the underlying data taxonomy in order to create a similar internal structure and analyze the data to find relations in real-time. In order to achieve a flexible, adaptable, decentralized and available solution, distributed systems and machine learning techniques will be explored in order to develop web-services capable of coping with such challenges.

The solution (UbiHSOM) is inspired by the taxonomy of the stock market (as we will later see in section 2.8), however a similar taxonomy can also be identified in other fields of interested, thus, even though the solution is inspired by the stock market, it also aims for a more generic one that can be used in various fields that present a similar taxonomy. Making it possible for professionals to adapt to other use case scenarios besides the stock market.

1.4 Document Organization

The following paragraphs will explain the structure of this document by giving a brief description of what each chapter is about.

In the current chapter the motivation, problem description and problem solution are introduced. Also a brief introduction to the stock market is given to facilitate the understanding of this thesis and its application.

Chapter 2 is focused on exploring state of the art and concepts (like clustering and K-means), required to the development of this thesis. Where various papers and other references are discussed to explain the frameworks and techniques that will be used on the development of the proposed solution, namely:

- Pearson's *r* correlation is explored as a technique to detected correlations between two data points;
- SOM as a unsupervised learning clustering technique that detects correlation between data points;
- UbiSOM algorithm as a variant of SOM for non-stationary data;
- HSOM as a method that takes advantage of merging techniques to merge various SOMs;
- the client-server architecture;
- the representational state transfer application programming interface architectural style;
- the microservices architectural style;

Chapter 3 is where the proposed solution (the UbiHSOM) to this thesis problem is addressed. It also explains how the frameworks and techniques discussed on Chapter 2 will be combined to achieve such solution.

Chapter 4 addresses the specification of the services that result from the proposed solution.

Chapter 5 takes into account the specification given in chapter 4 and explains how the solution was implemented in order to achieve such specification.

Chapter 6 presents a case study where the specified and defined services are used in order to extract knowledge from a collection of data sets, namely, geometric shapes data set (that has geometric figures e.g. square, triangle, diamond, pentagon, hexagon), the cube, iris and financial data set.

Finally, chapter 7 discuss how the initial objectives where met, the pros and cons of the proposed solution and it also briefly discusses the future of the solution and what could possibly be made with it.

Chapter 2

Related Work and Technologies

Cognitive computing was introduced as a way of improving machine-human interaction. This can be achieve through various machine learning techniques that already exist. Machine learning can be supervised or unsupervised. The latter is a class of machine learning problems that copes with finding meaning in unlabelled data. Unlike supervised learning, the goal of unsupervised learning is to find structure in unlabelled data instead of trying to predict some value.

This chapter begins by presenting Pearson's r correlation as the classical correlation measure technique then moving to a brief introduction to clustering and K-means which are fundamental machine learning concept and technique, respectively. Then SOMs are explained along with UbiSOM (a SOM variant) and HSOM. Those are important concepts for the development of the thesis, since, SOM is better than K-means due to its ease of data visualization, as discussed later. The idea will be to use UbiSOM as the basis SOM variant for UbiHSOM.

Formerly, the stock market motivation raised some challenges to professionals that want to analyse its behavior. The financial data set is provided by *GoBusiness* Finance research project, specifically concerning S&P500 and EuroStoxx600, that is divided into market sectors. The data will lately be used to extract knowledge and results, the data is described in section 2.7. The data has a specific structure – it is divided into market sectors – that a long with the previous introduced concepts can lead to a new solution (section 2.8). The structure of the data will have a huge impact in how the solution will be design and implemented, thus some distributed systems background is given. Section 2.10 introduces the client-server model that is a fundamental concept to understand the sections that precede: section 2.11 and 2.12. Section 2.11 presents an architectural style that is used by the microservices style explained in section 2.12.

Section 2.13 lists various existing frameworks, services and toolkits that are candidates to be used as the implementation tool. At the end of the chapter a discussion (section 2.13.5) takes place where the various technologies are compared in order to find which suits the solution better.

2.1 Pearson's *r* Correlations

Pearson's *r* correlation measures the strength of the linear relationship between two variables which is represented by *r*. Equation (2.1) is used to calculate the correlation, where $-1 \leq r \leq 1$. Being $r = 1$ and $r = -1$ the values that express high correlation, has the *r* value tends to 0 the correlation decreases to none, i.e., $r = 0$. Figure 2.1 shows examples of positive, negative and no correlation.

$$r = \frac{\Sigma(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\Sigma(x_i - \bar{x})^2 \Sigma(y_i - \bar{y})^2}} \tag{2.1}$$

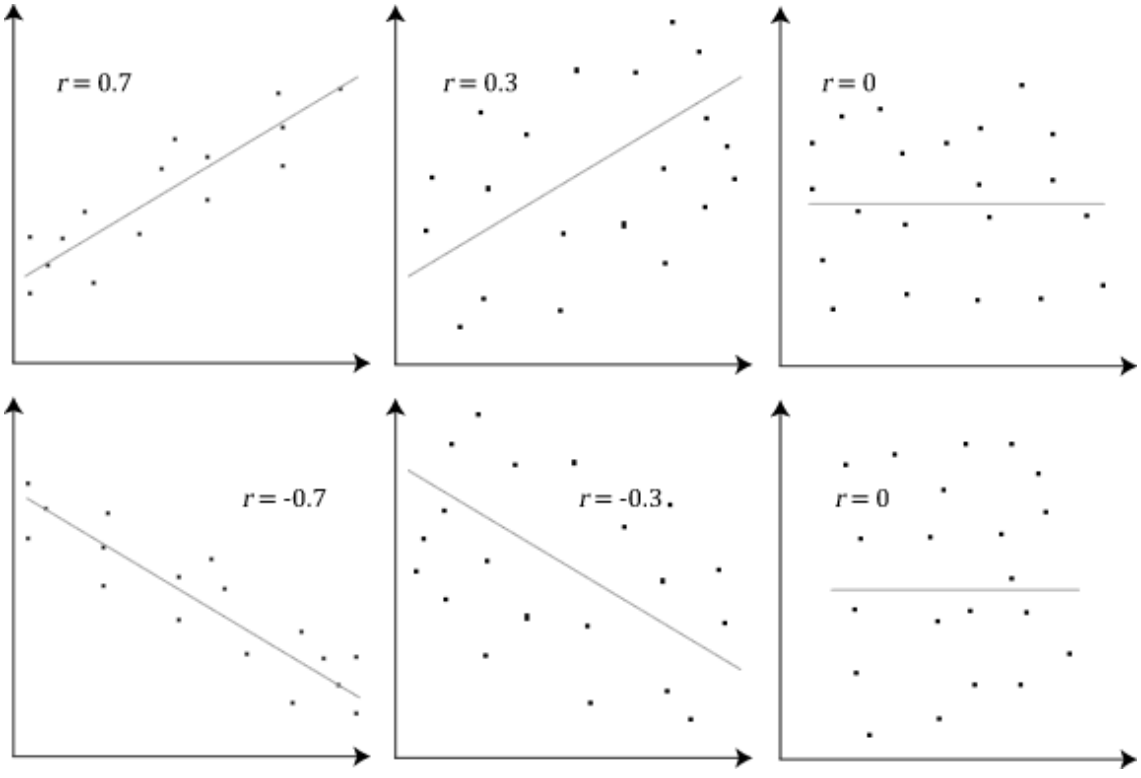


Figure 2.1: Pearon's r Correlation [25]

The problem with this technique is that it only works for linear relationships and outliers will taint the *r* value. Moreover the study of correlations for data partitions is not directly supported by this statistical tool.

2.2 Clustering

Clustering is an unsupervised learning technique with the goal of grouping similar examples and separate the different ones. Thus examples with similar characteristics are aggregated in the same group (called cluster) whereas the other examples are assign to a different cluster. Clustering is very useful because it help us understand the structure of the data and the relationship between different examples and features. It may also show us some important properties of the data that we were not aware of, like correlation between the data. The next sections explore different clustering algorithms, namely section 2.3, 2.4, 2.5 and section 2.6.

2.3 K-means

The K-means algorithm is a classic and widely used clustering algorithm that consists in dividing the data set into k clusters by the prototype of each cluster, i.e., each data point belongs to the cluster represented by the closest prototype. There are various ways of initializing those prototypes [11], lets assume those are randomly initialized. The main idea behind the K-means algorithm is to divide the data set into k clusters, where k is an initialization parameter. For each k there is a prototype and a data point is assigned to the closest one which recomputes its value based on the mean point of all data points assigned to it. The algorithm goes on until it converges or some stopping criteria is met[27]. Figure 2.2 shows an example of the algorithm results for different values of k given an image. We can see that in this case the value of k is related to the number of colors shown.

Figure 2.2: K-means example [11]

2.4 Self-Organizing Maps

The Self-Organizing Map SOM is an unsupervised-learning clustering algorithm invented by Professor Teuvo Kohonen in 1981-82. SOM itself is an Artifican Neural Network (ANN) that has various applications [36]. It is used to project and visualize high-dimensional

data on lower dimensions (mainly 2D and 3D), because SOM preserves the relationships among the input data, even after transforming the input dimensions.

An analogy will be described in the next paragraph to ease understanding of the intuition behind SOM.

Imagine a table full of objects (those can be anything). If you throw an elastic mantle on the table, you'll be able to visualize and identify (through relief) clusters of objects with different heights. Lets say that the data are the objects and the mantle is the SOM. Basically the idea is: we have a set of objects where we throw an elastic mantle at, that wraps the set and enables the visualization of existing clusters. The mantle will be stretched on areas where there are no objects and normal where there are clusters.

Let m be a d-dimensional prototype vector $m_i = [m_{i1}, ..., m_{id}]$, x a sample data vector, ω the training set, δ the best matching unit (BMU), λ the set of all prototype vectors and ω the data set. Each neuron i has a m. In short, the algorithm goes like this for each training step:

1. Choose a random x from ω
2. Calculate the distance (2.2) between x and λ
3. Update δ and respective neighbors (2.3)

$$\|x - m_c\| = \min_i \|x - m_i\| \tag{2.2}$$

$$m_i(t+1) = m_i(t) + \alpha(t) h_{ci}(t)[x - m_i(t)], \tag{2.3}$$

where t denotes time, $\alpha(t)$ the learning rate and h_{bi} is a neighborhood kernel centered on the BMU. $\alpha(t)$ and h_{bi} radius decreases monotonically with time.

SOM is a mesh of neurons [36] (so as expected each neuron has its own weights, i.e., the m_i vector), topologically organized (like a grid) as shown in the Figure 2.3. This grid is usually two or tri-dimensional, higher dimensional grids are possible but they are not often used because it is hard to visualize such dimensions. The input data is presented to the SOM and at the neuron level, each neuron drags itself towards the nearest data point, lets call the closest neuron to the data point, best matching unit BMU. This action has side effects on the neighbor neurons, it causes them to get dragged along with the BMU, as shown on Figure 2.3. By doing this the grid will assume the data set shape after some iterations. Figure 2.4 illustrates the reaction of a SOM when submitted to a data set-shaped cactus. It is easy to see how SOM wraps the data set. The cactus-like shape of the neurons is not perfect due to the connection between the neurons.

When analyzing real-life data, care must be taken because if erroneous data is fed to the SOM, the result is also erroneous or of bad quality. Thus variable normalization must be used since SOM uses Euclidean distance. This subject is discussed with more detail in sub section 2.7.1.

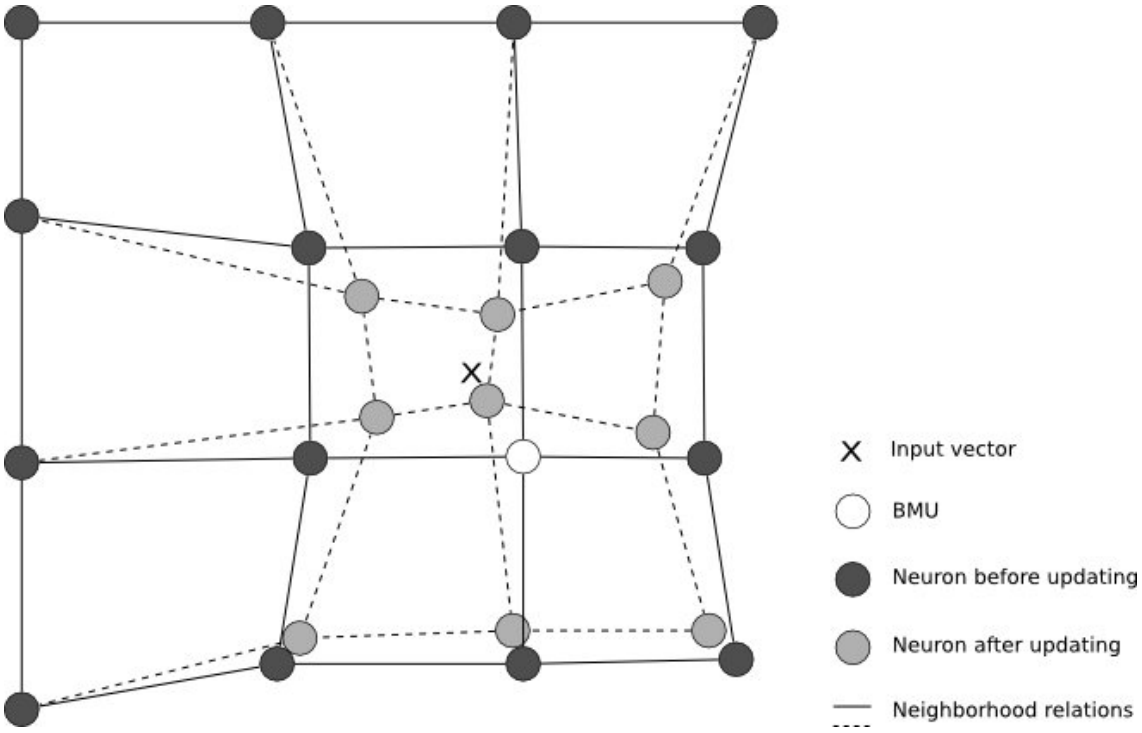


Figure 2.3: SOM process [57]

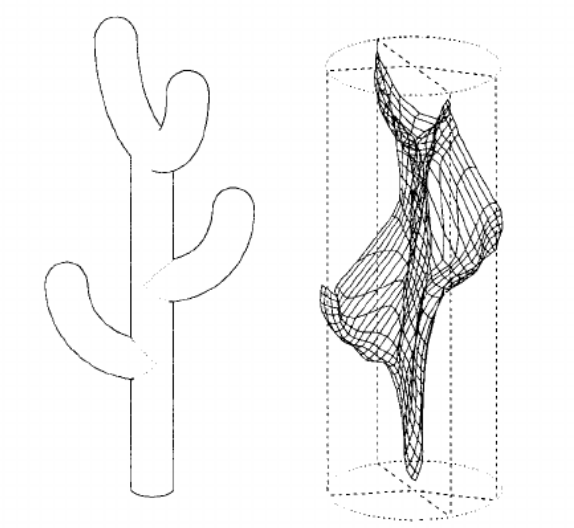


Figure 2.4: SOM process illustration [36]

Since it is easy to project data to two-dimensions, there are several visualization techniques [47] that can be used to visualize the SOM:

- **Hit Histograms** – show the distribution of the best matching units for a given data set.
- **U-Matrix** – shows the distance between each two adjacent prototypes.
- **Component Planes** – shows the relative component distributions of the input data

2.5 UbiSom

Among many variations of SOM, UbiSOM [59] is the one that fits the needs of this thesis solution, it is underlying model will be described in this section as well the reasons behind its preference.

UbiSOM is a variant of the SOM algorithm that was developed to cope with streaming environments to address the problems of the classical algorithms, which is the lack of support to process the endless incoming data from streams.

The UbiSOM algorithm implements a finite state-machine of two states, as shown in Figure 2.5:

- **Ordering State** – this is the initial and the revert state of the algorithm, where it allows the map to unfold over the underlying distribution and to recover from abrupt changes in the that distribution.
- **Learning State** – this state is the main state of the algorithm, its responsible for updating the algorithm control variables every time new data arrives.

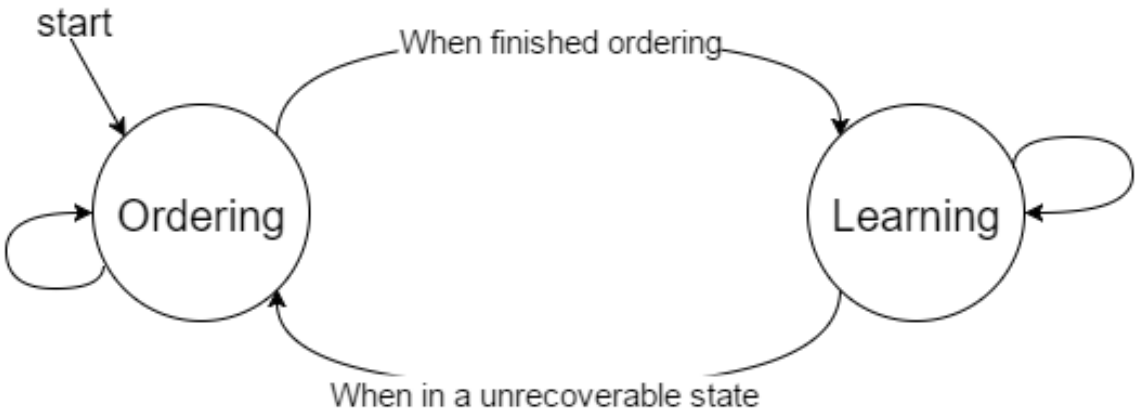


Figure 2.5: SOM process

UbiSOM relies on the following learning assessment metrics:

- **average quantization error** – this metric gives an indication of how well the map is currently quantifying the underlying distribution, i.e, how well the model fits the underlying distribution. If the underlying data stream is stationary the average quantization error is expected to stabilize thus the map is converging. However if the distribution changes, this metric is expected to increase.

- **average neuron utility** – measures the proportion of neurons that are actively being updated. Its an additional measure that helps detecting underused neurons that are detected by the average quantization error.

Basically the metrics above are used to decide when the algorithm should change from the *Learning State* to the *Ordering State*, since there are cases of abrupt changes that unables the map to resume convergence (more detail about the UbiSOM parameters is given in section 4.3.2). Therefore if the metrics values were at their peak during the last T iterations, it means there were abrupt changes on the underlying distribution so the UbiSOM transitions back to the ordering state.

Other variants of SOM try to address the non-stationary data streams problems but as discussed in [60] they do not guarantee a compact model due to parametrization issues. Another great feature of the UbiSOM is the fact that it can recover from an unordered state. This feature as a great impact in a stock market scenario because an abrupt change of the distribution can mean that a product disappeared, the UbiSOM is able to detect it and a notification can be sent to the responsible asset for further analysis.

It was also shown that the converging time after the identification of an unordered state is fast [60], which meets another non-functional requirement. At this time, it also outperforms current SOM algorithms in stationary and non-stationary data streams [58].

UbiSOM is the selected SOM variant for the development of this thesis solution because it was built on the basis of non-stationary data streams and network computing to increase performance [59], UbiSOM fulfill most of the non-functional requirements of this thesis.

2.6 Hierarchical SOM

This section explores SOM merging techniques, since HSOM needs merging techniques to merge maps. Ttwo merging techniques are discussed in sub section 2.6.1. The HSOM process is explored in sub section 2.6.2 and HSOM taxonomy in 2.6.3.

2.6.1 Merging SOM's

To take advantage of the data taxonomy (please refer to 2.7) two SOMs could be merged together. The HSOM hierarchic structure has many tiers, i.e., node layers, that are achieved by merging SOMs through merging techniques. Here two of them will be discussed, one

aims at prototype exchanging [59], while the other is about exchanging BMU coordinates between SOMs [29].

It was shown in [29] that the merging of the maps can be achieve with the simple exchange of the BMU coordinates. For instance, let A and B be two first tier SOMs and C be the resulting SOM of merging A and B (being C a second tier SOM). To achieve C, A and B must be merged by feeding C with their BMUs coordinates. Resulting on C having as input patterns A and B outputs, i.e., their BMUs coordinates. While in [59] the merging of SOMs is done through prototype exchange instead of BMU coordinates.

2.6.2 Hierarchical SOM Process

This description is based on Professor Roberto Henriques PhD thesis [29] which extensively detailed the HSOM subject.

HSOM [38] is a generic multilayer type SOM. The idea behind HSOM is to use SOMs as building blocks of a higher layer SOM. In this thesis HSOM is preferred over the others because the outputs of one SOM are used to actively train another SOM unlike the other methods. Furthermore the financial data set taxonomy is similar with HSOM taxonomy as we shall see later on section 2.6.3.

HSOM works as multiple layers connected in a feed-forward way, i.e, the interaction between layers works as the following, let α be a first tier SOM and β a second tier one: the outputs of α are used to train β which maps the original data patterns using the outputs from α, and so on for the other existing higher tier SOMs, as shown in Figure 2.6.

The essence of HSOM is that to feed the next layer SOM the original data set is not used to train the map but instead the prototypes are the ones used for that purpose. So, for instance, the next layer SOM could have as input the coordinates of the BMU, its quantization error and its distance to each unit. Hypothetically, this method also reduces computational effort because it reduces the dimensionality of the inputs to each SOM, the distance functions used for training will also be simpler and thus faster to compute.

In short, HSOM operating principle [38] is:

1. For each input vector x, the best matching unit is chosen from the first layer map and its index b is input to the second layer;

2. The best matching unit for b is chosen from the second layer map and its index in the output of the network.

As discussed in [29] the HSOM is less sensitive to outliers, allows the use of a higher number of variables without degradation in performance and allows hence allowing the use of larger SOMs.

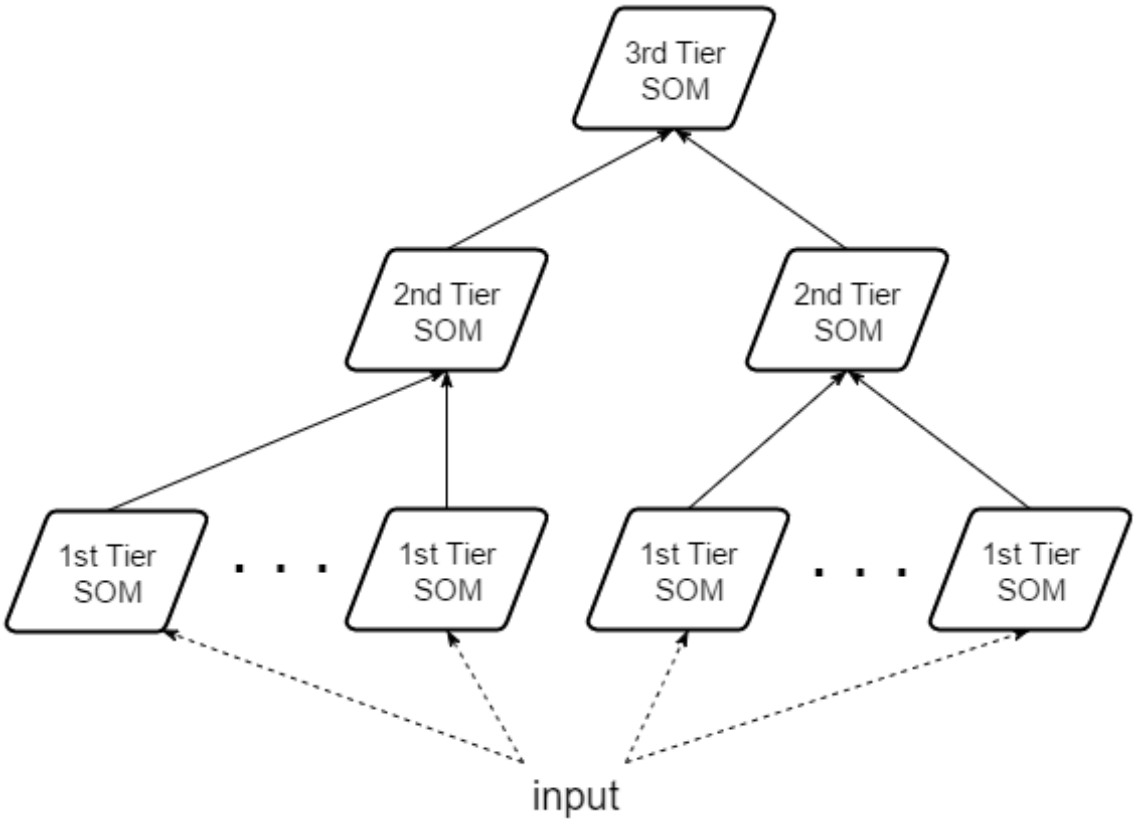
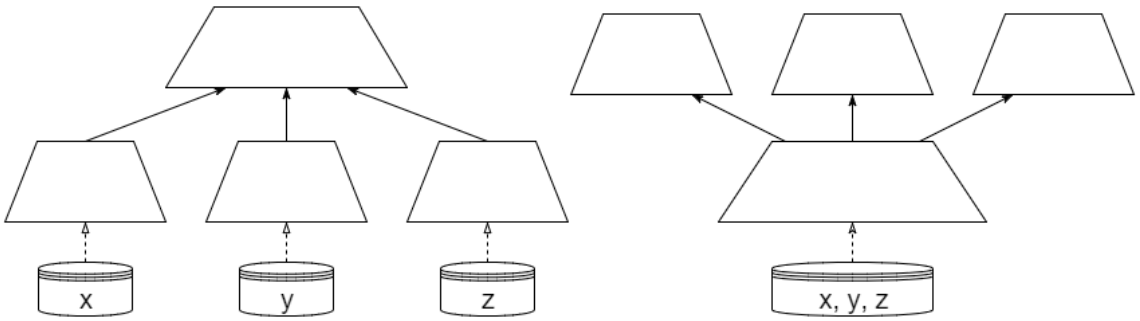


Figure 2.6: SOM process

2.6.3 Hierarchical SOM taxonomy

There are various HSOM methods based on their objective and type of structure used, e.g., agglomerative and divisive. For the agglomerative method we have two classes: thematic and cluster based. For the divisive method we have other two classes: static and dynamic. Here only the thematic (agglomerative) class will be discussed.



(a) Agglomerative Hierarchical SOM (b) Divisive Hierarchical SOM

Figure 2.7: Types of Hierarchical SOMs

The agglomerative HSOM lays its ground on having the data being fed to the first tiers SOMs and then fusing their outputs to the next tier SOM. Whereas the divisive HSOM

has a single first tier SOM having then several SOMs on the second tier. The main difference between the two methods is that with the agglomerative HSOM the abstraction level increases as we go up in the hierarchy. While the agglomerative HSOM is the opposite, the first tier SOM is less accurate and as we ascend the hierarchy it the SOMs become more detailed and accurate.

The thematic agglomerative HSOM, as the name suggests, is based on themes, i.e., the input space is viewed as a collection of subspaces, each of them belonging to a theme. Each theme serves as the input of one SOM and its outputs are used to train a higher tier SOM and so forth.

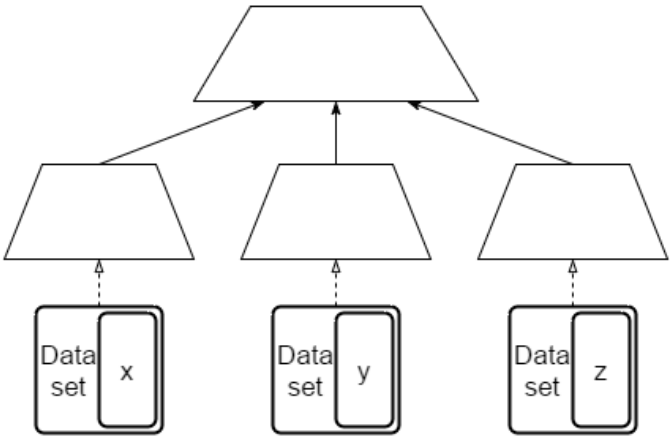


Figure 2.8: SOM process

This structure presents several advantages for multidimensional clustering:

- reduction of computation, due to input space partitioning
- the creation of thematic clusters may be interesting to the analyst
- the diversity of clusters can give the analyst different perspectives to analyze thus allowing a better understanding and exploration of emerging patterns.

2.7 The Financial Data

Bloomberg[1] is one of the main world wide financial market information providers, known also for the TV channel, *Bloomberg Television*, where they stream stock market related information, they also sells information terminals to, e.g., banks, brokers and insurance companies.

Information regarding *S&P500* and *EuroStoxx600*, will be used since those are classic and stable markets. *S&P500* is a stock market index based on the market capitalization's

[1] www.bloomberg.com

of 500 large american companies. *EuroStoxx600* is the european stock market based on the market capitalization of 600 large companies.

The information is stored in the *GoBusiness MySQL* database that is fed with *Bloomberg* data regarding the last 20 years. The data regarding the several financial products is divided into industry groups, e.g., Retailing, Banks, Automobiles & Components, Telecommunication Services and Health Care Equipment & Service. This taxonomy is similar to the HSOM taxonomy discussed in sub section 2.6.3, which makes this data set good for analyses.

The general database schema was defined in [65].

The information is stored in the *GoBusiness MySQL* database that is fed with market data extracted from a *Bloomberg* terminal regarding the last 20 years.

2.7.1 Pre-processing

SOM requires input data to meet a vector-like format, i.e., each value of the vector must be in the numerical/decimal format and it needs to be normalized to fall between the $[0,1]$ domain interval, since UbiSOM considers that the data arrives already normalized. The data is stored in a *MySQL* database errors must be removed due to the possible existence of *null* values. There are different solutions to address this problem:

- replace *null* values with *don't care* values;
- remove vectors with *null* values;
- use input vectors with missing values [63].

2.8 A similar taxonomy

It is easy to find correlation between two variables with Pearson's r correlation. The problem with this technique is that it only works for linear relationships and outliers will taint the r value. Moreover the study of correlations for data partitions is not directly supported by that method. Outliers tend to happen a lot making Pearson's r correlation unreliable (in this case), therefore, making SOM Euclidean distance per cluster provides a better approach.

Among the various SOM variants, UbiSOM is preferred over K-means because the analyses is focused on non-stationary data, i.e., it processes in real-time data and also because of UbiSOM visualization power. Moreover the main advantage of UbiSOM relative to K-means is the adaptive distance measure [10], since UbiSOM offers a distance measure that takes into account all the points in the cluster. When it comes to the HSOM taxonomy, the thematic agglomerative class is the one that best copes with hierarchic datasets, e.g. the financial data set, since the data is divided into industry groups and those groups are made up of financial products. Therefore both the dataset and the HSOM model seem to

follow a very similar taxonomy.

Two merging SOM techniques were discussed in sub section 2.6.1, one uses the BMU coordinates as input patterns for the next level SOM while the other uses the prototypes. The technique that will be used is the BMU coordinates exchange approach. The BMUs coordinates are defined as two dimensional Cartesian coordinates and it is represented by a tuple (x, y). The coordinates correspond to the BMU position on the prototypes grid. This approach implies that each HSOM node of tier two and above will have dimensionality equal to two in order to match the input from the lower tier nodes which is the BMU coordinates (that also has dimensionality of two). There is no related work that evaluates which of them is better and its not the goal of this thesis to do it either.

2.9 MongoDB

Nowadays there is a great collection of Database Management System (DBMS), most of them are based and use the Structured Query Language (SQL) declarative language. Those DBMSs are commonly concerned with relational databases, however, with the growth of the big data and machine learning fields, new systems started to emerge, e.g., NoSQL DBMSs. As the name suggests, NoSQL DBMSs are not based on SQL, the underlying technology they use varies, depending on the requirements they aim to fulfill.

MongoDB falls under the NoSQL DBMSs category. It is implemented in C, C++ and JavaScript and uses collections instead if databases and documents instead of tuples, being a collection a set of documents and a document a Binary JSON (BSON) file. Unlike the relational databases that are concerned about the data integrity and maintaining (by following the *ACID*[56]), MongoDB only cares about inserting, updating, removing and reading documents. Besides the previously mentioned capabilities it also offers some extra features, for instance, the *MapReduce* operation[42] that can be used to process past data (as it is later discussed in section 7.2).

The MongoDB is the go-to DBMS for the proposed solution because the solution uses JavaScript Object Notation (JSON) as the exchange data model. JSON was chosen over XML because JSON maps directly with JavaScript objects, it is easier to parse, lighter and since the chosen DBMS[16] is MongoDB (which uses BSON data model[13]) it perfectly matches the one-to-one mapping. Moreover, the solution follows the Representational State Transfer (REST) style which typically uses JSON as the exchange data model (please refer to section 2.11).

2.10 Client-Server Architecture

The client-server architecture style [26] is frequently used on applications. In this kind of architecture the client sends requests to the server, the server receives the requests, computes and sends the results to the client. This kind of pattern is used in a lot of

services, mainly web-based services, e.g: online gaming, financial trading, web servers, mobile applications. The model is depicted in Figure 2.9.

Figure 2.9: Client-Server Model

2.11 Representational State Transfer API

REST [23] is an architectural style widely used to develop web-based services [73]. A web-based service, is a service that uses the Web to communicate with its clients (being them machines or humans). "A web service is a software system designed to support interoperable machine-to-machine interaction over a network."[71]. REST uses the World Wide Web existing defined protocols and operations such as Unified Resources Identifiers (URI)/URL and Hypertext Transfer Protocol (HTTP) operations (GET, PUT, DELETE, POST, among others [31]), to communicate with clients. Thus clients can manipulate the exposed resources identified with its URI through HTTP operations. Each resource has its own URL where it can be accessed and manipulated by the client. The communication is stateless and its over HTTP and the data is often exchanged with eXtensible Markup Language (XML) or JSON files.

This kind of style follows the client-server architecture pattern since each request to the server equals to a HTTP operation on an URL. The client is always supplied with the entire state of the resource instead of calling an operation to get some part of it.

Normally this is the typical behavior of the HTTP operations:

- **GET** - returns the state of a resource
- **PUT** - updates a resource
- **DELETE** - deletes a resource
- **POST** - creates a new resource

The main feature that distinguishes REST from other architectural styles is that REST applies the principle of generality to the component interface making it uniform between components. Thus improving the overall system architecture and decoupling implementations from the services they provide.

In short, a REST web-service (Application Programming Interface (API)) is a service with a set of well defined URLs, exposed via web that can be manipulated with HTTP operations, commonly, GET, PUT, DELETE and POST.

2.12 Microservices Architectural Style

This style has gained popularity [66], [1],[52] due to monolithic applications frustrations, namely: keep it modular, horizontal scaling requires the entire application to be scaled, normally the whole application is written in one unique language, as the application grows it becomes harder to test it, among others [24].

This kind of style is very similar to SOA[50] style however the community opinions are divided [24].

According to [45] microservices are "small autonomous services that work together, modelled around a business domain" . One of the goals of this architectural style, is to build independent and decoupled services that are orchestrated to complete a task. These services communicate with each other through lightweight mechanisms (e.g. an HTTP resource API). In a monolithic application all the logic is bundled into a single process, hence it can be horizontally scaled by running many instances in different nodes with the help of a load-balancer for payload balancing, however, it is not efficient because it is only advantageous for the features that have a fair amount of requests, whereas the others are just uselessly using resources that are more than enough for their payload. On the other hand, microservices can also scale vertically because the system is divided into small independent functional services with non-shared state. Thus making it easier to scale.

The microservices architectural style promotes:

- **Maintainability** – each microservice is a small component only concerned about one functionality, this originates small codebase;
- **Scalability** – microservices are easily scaled because they are decoupled from the rest of the system;
- **Resilience** – in case of failure only a few features of the application will go down, the application itself – as a whole – will continue executing;
- **Polyglot** – decoupling makes it possible to implement the various integrating microservices in different programming languages, leading to a more diverse environment, thus allowing the developers to pick the the right tool and data storage technologies to better cope with the problem at hand;
- **Evolutionary Design** – if any microservice needs to be improved there is no need to rewrite the entire application since it is decoupled, only the desired microservice needs to be updated because it is (ideally) functionally independent.

This architectural style has been gaining a lot of support and companies like Netflix and Amazon are migrating their monolithic application to microservices. Spotify has also adopted the microservices architectural style. In fact, Amazon Web Services (AWS) has a service called AWS Lambda (more on this later), that allows clients to create a

microservice like computation unit that reacts to events, thus promoting the same type of architecture.

This style has its pitfalls, one of them being the heavy decoupling. For instance, while designing the solution architecture the developer decides to turn everything into a microservice thus decoupling everything from everything. This is really important because when one decides to break down everything into a microservice the application structure becomes incredibly complex and hard to maintain. A balanced decoupling methodology is necessary for an application that follows this kind of style to benefit from its advantages.

2.13 Distributed and Data Processing Frameworks, Services and Toolkits

This section analyzes various existing technologies that could be used for the proposed solution, namely: Apache Spark[6] and Storm[7], AWS Kinesis[4] and Lambda[8], Akka toolkit[2] and Vert.x toolkit[20]. At the end of the section one of the a discussion takes place where the technologies are compared to finally choose one for the proposed solution.

2.13.1 Apache Spark and Storm

Spark is an Apache Foundation open-source project. It is a flexible in-memory framework that allows it to handle batch and real-time analytic and data processing workloads. It is a fast and general-purpose cluster computing system for large scale data processing. It is an alternative to MapReduce but it is not tied to the two-stage MapReduce paradigm.

Spark has a lot of core components, but to have a grasp of the big picture we just need to know the following. Resilient Distributed Dataset (RDD) is immutable data that is distributed across the cluster prime to be executed on. RDD's allows us to do a Directed Acyclic Graph (DAG), in analogy with MapReduce it is similar with its chain mechanism[19]. A DAG in Spark represents the steps needed to get from A to B. For instance, A joins two data sets and B counts a specific property. Thus one defines a job in Spark, by defining the DAG along with its actions and transformations to be applied to the RDD collection.

Apache Storm is an Apache Foundation open-source project. Storm is the Hadoop of real-time stream processing. Figure 2.10 shows an example of a Storm based application topology. Spouts units are responsible for receiving data and pass it to Bolts. Bolts are the computation unit that process input and either persist it or passes it to some other Bolt.

2.13.2 Amazon Web Services Kinesis and Lambda

Kinesis and Lambda are AWS services. Kinesis is for streams, it takes data from data producers and transmits it. Lambda is for processing. Lambda reacts to asynchronous or

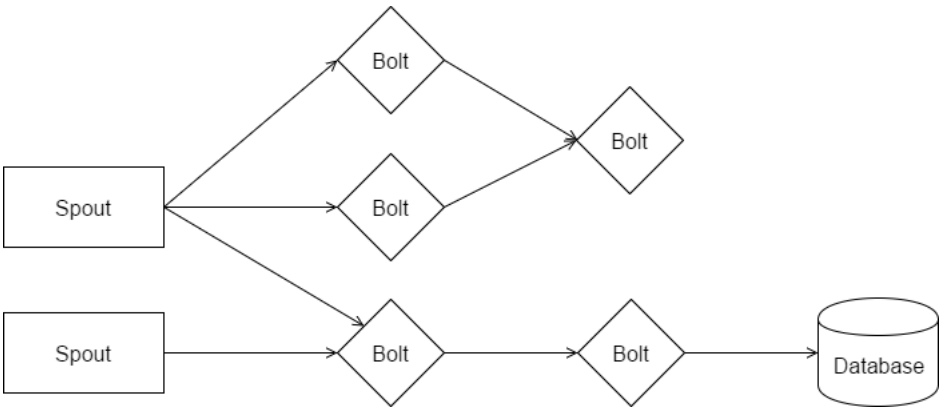


Figure 2.10: Apache Storm - Topology

synchronous events, in AWS a lot of services can trigger events: Amazon S3, Alexa skill, Amazon DynamoDB, among others[5].

Both services can be combined to build real-time data processing applications. Kinesis is responsible to build the data stream and fire events that triggers Lambda for processing.

AWS is paid and their services are limited [18],[9].

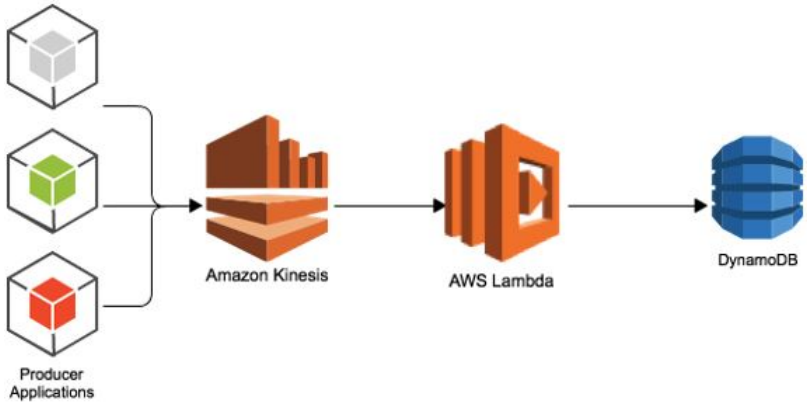


Figure 2.11: AWS - Kinesis + Lambda [55]

Figure 2.11 illustrates a solution where a kinesis stream receives data from producer applications (e.g., IoT devices, Amazon DynamoDB Update Stream) and fires events on a Lambda functions, that processes the input and stores the results in a AWS DynamoDB database. Lambda functions can be chained with kinesis streams.

None of this services are local, they run in the AWS cloud and Internet connection is needed to interact and use them.

2.13.3 Akka toolkit

According to the publisher this is their definition: "Akka is a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM." [2].

Akka is a toolkit from Lightbend and it offers a single unified programming model for simple concurrency, distribution and fault-tolerance (those are given out of the box by Akka). Akka uses the Actor Model which is an abstraction layer for the developer so he does not need to worry about distributed computing challenges (e.g. shared state, threads).

Common Akka use cases are:

- Transaction processing
- Service backend
- Concurrency/parallelism
- Batch processing

The Actor Model was firstly implemented in Erlang. An actor is an unit of computation that receives messages and responses to it, thus an actor embodies processing (behaviour), storage (state) and communication. One interesting characteristics about actors is that they can change behaviour at run-time. Each actor has a mailbox where it receives its messages to process. An actor can create new actors, receive messages and in response: make local decisions (e.g. alter local state), perform arbitrarily, side-effecting action, send messages and respond to the sender zero or more times. It only processes exactly one message at a time.

2.13.4 Vert.x toolkit

At 2011 Tim Fox started a project named Node.x, inspired in Node.js, where the "x" stands for a polyglot project unlike Node.js (that just supports JavaScript). Later in 2013 the project was moved to the Eclipse Foundation under the name of Vert.x, winning the "Most Innovative Java Technology" award at the JAX Innovation awards in 2014. Julien Viet was appointed lead of the Vert.x project at the beginnings of 2016.

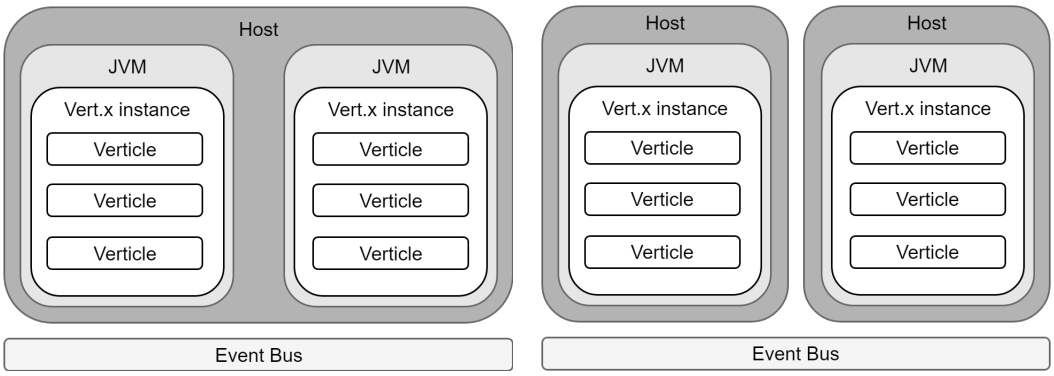
Vert.x is an open-source tool-kit for building reactive application that runs on the Java Virtual Machine (JVM). Since it is a tool-kit and not framework, it is unopiniated, i.e., there are no conventions on how the code should be written in order to take advantages of the features it offers. Vert.x is also reactive, which means it follows the properties defined in the reactive manifesto [12]. In short reactive applications should be: responsive, resilient, elastic and message-driven.

- **Responsive** – they respond to the user in a timely manner;
- **Resilient** – they stay responsive to the user in case of failure;
- **Elastic** – they stay responsive in a varying workload by scaling;

- **Message-driven** – they use asynchronous message passing to establish boundaries between components in the system.

Vert.x primitives are verticles. A verticle is the executable unit of a Vert.x application. Verticles run inside a Vert.x instance that runs inside a JVM and the JVM runs on a host as shown in Figure 2.12. It is possible to run multiple hosts with Vert.x where all the verticles in a vert.x application connect and communicate with each other through a distributed event bus fired by Vert.x.

The distributed event bus connect verticles within one JVM or on multiple JVMs within multiple hosts. The distribution/clustering part is done internally by Vert.x, it uses Hazelcast[28] to accomplish it. Basically the event bus is a message bus that supports both the request/response model and the publish/subscribe model. Hence verticles interact with each other using messages.



(a) Vert.x - Host (b) Vert.x - Multiple Host

Figure 2.12: Types of Hierarchical SOMs

Vert.x is event driven, everything in Vert.x is an event (e.g. HTTP request, messages between verticles and timers). To handle this events Vert.x uses an event loop, i.e., a loop that is constantly checking if whether or not there are more events to process. Unlike Node.js (that also has an event-loop) Vert.x distinguishes itself because it fires one event loop per CPU core (multi-reactor pattern), whereas, Node.js only works with one event-loop. The event loop process is very simple, Figure 2.13, for each new event, Vert.x fetches it and delivers it to a handler.

2.13.5 Discussion

Apache Spark and Storm are really powerful frameworks that for both batch and real-time data processing. Even though they offer a lot of advantages they're still a framework which means conventions must be followed in order to achieve the desired solution. The use of this frameworks would result in a more heavy, less agile and less ubiquitous solution. Heavy because they are both frameworks, this means it has more size, it is more

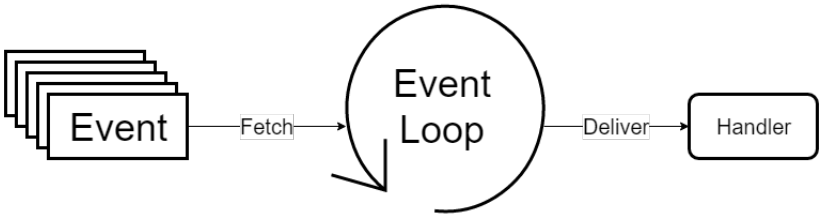


Figure 2.13: Vert.x - Event Loop

complex and this can compromise low-end computing hardware performance, since they have bigger hardware requirements.

On the other hand there is AWS Kinesis and Lambda that also offers out of the box cloud based real-time data processing with a better learning curve but the services are paid and have limitations as shown in 2.13.2. This makes this option expendable because it requires additional – difficult to predict – costs.

This leaves us with Akka and Vert.x toolkit. Both use the Actor Model and they are both asynchronous. Akka has its advantages and pitfalls, it is meant to be distributed – and it is – but a lot of configuration lifting is required to setup certain things that Vert.x eases, namely, distributed messaging. In order to do it with Akka one must correctly configure actors the same can be achieved with Vert.x without such burden. Even though I found it easier to model actor message response behaviour in Akka, I think that versus the configuration burden it is not worth it. Moreover Vert.x is more polyglot than Akka.

With Vert.x there is no need to worry about concurrency and synchronization because Vert.x ensures that there is only one thing at a time accessing one handler that is being served by (one and the same) event loop. This provides you with actor-like concurrency. So there is no need to worry with locking or synchronization.

Caution is required (in Vert.x) when trying to do a blocking operation (e.g. access a disk file or a database), because if and handler performs such kind of operation the event loop will block. This means that it will wait until the operation terminates and only then will fetch another event thus blocking the whole system. The solution to this problem however is simple. If a blocking operation is required, that task must be delivered to a worker. A worker is simply a verticle that will carry on the task (in background) and when it is completed will return the result, thus not blocking the event loop.

Vert.x was chosen as the main developing tool-kit because of being reactive, concurrent, asynchronous, elastic, responsive, message-driven and the ease of communication between components (verticles). Hence is easy to follow a microservice[22] approach.

Chapter 3

Proposed Solution

In section 2.5, UbiSOM was presented as the go-to SOM variant to work with real-time data and to deal with the challenges it raises. However, there are real-life situations that follow some kind of structure, sometimes hierarchic, requiring the use of models that follow it, e.g., HSOM. There are HSOM variants for specific situations (e.g. geographic related problems[29]) that rely on non-stationary data, nonetheless if the use case implies working with data in real-time, those solutions are fruitless. Thus some kind of HSOM variant that deals with real-time data is needed.

Section 2.6 explored HSOM general concepts, how to merge them, the associated process and an underlying taxonomy. The similarities between the data set taxonomy and HSOM taxonomy are evident (as discussed in section 2.8) making it possible to take advantage of them.

This chapter tries to propose a solution that mixes (both) UbiSOM and HSOM models in order to build a HSOM variant that supports real-time data analysis, called UbiHSOM by having UbiSOM as its building-block.

3.1 An UbiSOM taxonomy

As we have seen in chapter 2 section 2.6.3, HSOM is a set of SOMs that work together to analyze data in a hierarchical fashion. The HSOM highly depends on the underlying SOM variant capabilities, hence, in order to have a HSOM variant - the UbiHSOM - able to handle real-time data analysis, the underlying SOMs should also have the same capabilities. Therefore the UbiSOM was chosen because it is ubiquitous and real-time data ready, making it the perfect SOM variant for this case.
Let's first look at a single UbiSOM instance as illustrated in Figure 3.1. The model at this level of abstraction is described in [58]. As any other machine learning model it

feeds on data. The model processes the data (learns), updates its prototypes (neural-network) and it is ready to receive more data, the difference is that the data is (ideally) non-stationary data (real-time data). As pointed in section 2.5, whenever the UbiSOM node starts receiving data that deviates a lot from what it has been receiving, it restarts and tries to adapt to the new data. This is a crucial requirement when working with real-time data, because it is really hard to predict if the structure will not change in the future[58].

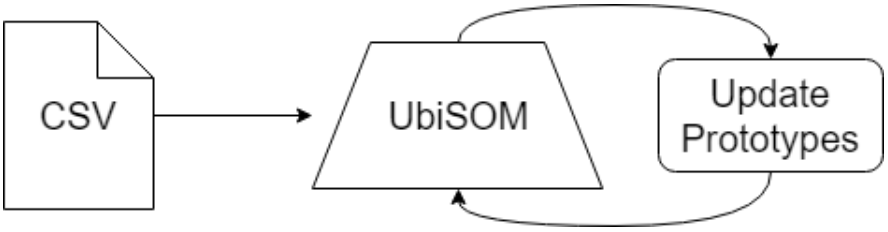


Figure 3.1: UbiSOM Instance

From now on the term UbiSOM instance and UbiHSOM node (or just node) will be used interchangeably across this chapter.

Figure 3.2 proposes an UbiHSOM representation. As illustrated in the figure, it is easy to observe that each UbiHSOM node is connected with some other node by a link and the first tier nodes have data sources (e.g. database, CSV, datastream, file) connected to them. The first tier nodes feeds on data provided by data sources , process it and then propagates the processed data in a feed-forward fashion to the higher tiers nodes, i.e., when a first tier receives data and learns, it updates its prototypes and sends the results to the higher tier nodes. This taxonomy follows the previously discussed thematic agglomerative HSOM taxonomy (discussed in section 2.6.3), where the first tier nodes receive data from a data source (each regarding a different topic), e.g. database, and forward the results to the higher tier nodes. Being the highest tier node the global representation of all the topics combined. It is possible to identify three components by analyzing the figure: UbiHSOM nodes, links and data sources.

In short, UbiHSOM nodes are simply UbiSOM models that take data as input and processes it. Moreover as described in section 4.3 the original UbiSOM was extended and some features added, as we will later see in section 4.3. The UbiHSOM nodes are linked to each other and each of them forwards its results to the next tier nodes, therefore we have achieved the UbiHSOM model. In detail, what actually happens is that every time a new data sample arrives at an UbiHSOM node the BMU of that data sample is calculated, the prototypes updated and its the BMU coordinates are forwarded. Note that in the figure its possible to see that the links are pointing to some node, this means the data propagates from one node to the other. E.g. we have node A directly connected to node B, with the arrow pointing to B, thus the data comes from A to B.
In sum there are:

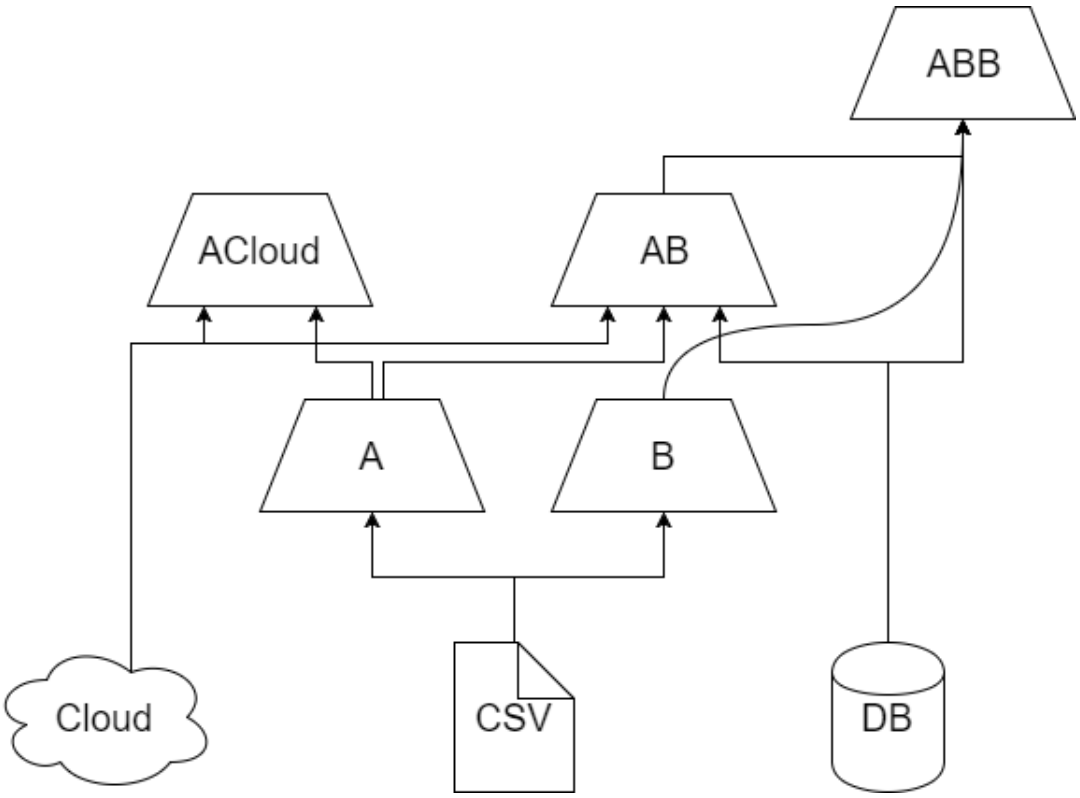


Figure 3.2: UbiHSOM - Ideal Model

- **UbiHSOM nodes** – that are an extension of the UbiSOM models with some new features;
- **Data sources** – that serves as the data supplier to the UbiHSOM;
- **Links** – that connects UbiHSOM nodes and propagates data from one node to another.

What can a data source be? – This is an important question, since SOM needs data to process, it is important to know where can that data come from and what can it be. Let us first look at a few non-stationary SOM model examples. In this scenario the data comes from a database, where the data is pulled through a query and fed to the model. There is another case where the data is stored in a CSV (comma separated value) file and fed to the model. In this settings the data comes from two different sources but they have a similar impact. Since the solution expects the data to be in real-time, hence, ideally, the data can come from anywhere, because UbiSOM is ready for it. So its possible to have various types of data sources, for instance, one being a device (e.g. thermal sensor), another being a video stream and the other some WebSocket API stream channel. Careful must be taken because each model has a dimensionality that should be respected (like in UbiSOM), so if the data source is not trustable on that matter, the data should be filtered

before being fed to the model (more on this on section 4.3.1).

What about the links? – The links are really interesting, even though they look very simple in the figure and their functionality is pretty straight forward – send X to B – they open opportunities. Let us say we have two models A and B, and a link L. In a normal case, A receives data, processes it and sends the results to B through L. It could be useful to have some kind of intermediate process (a middleware) that performs a computation on the data before it reaches B. One could, for instance, check if the new processed sample deviates from the previous ones and if it does check if there is a better one on a convenient database, thus ensuring (somehow) the quality of B. So now we would have A, B, L and M, the middleware that computes the data from L before it reaches its target.

The next section aggregates all this ideas and tries to define the UbiHSOM model.

3.2 The UbiHSOM Model

In this section the motivation and the concepts from the presented scenarios are used to formally define the UbiHSOM model. The previous section shown that an UbiHSOM is essentially composed by three main components: UbiHSOM nodes, data sources and links. Where data sources feed data to first tier nodes and those process and propagate the data to higher tier nodes through links.

There are various data structures that could be used to represent the UbiHSOM state, but the most dynamic and similar, is a simple directed graph. One of the reasons being that links are one-way links, i.e., they only send data from one point to another and the reverse does not apply, hence this is like a simplex channel in computer networking: A only sends data to B, B never replies. Another reason is that tiers, i.e. node layers, should be dynamic, i.e., there are nodes that can be connected with other higher level tiers as shown in Figure 3.3. Where A is directly connected with a third tier model and that model is also connected to a second tier model. Here the tiers are merely abstract to ease understanding, because they can be blurred in certain situations where particular connections regarding nodes of different layers are done as depicted in the Figure. Even though some node input is other node output, to keep things simple those concerns should be decoupled, thus creating a more dynamic structure that allows nodes to connect with whatever is desired. Therefore a node should only be concerned about learning from the data it receives, while links should be concerned about delivering data from one point to another.

Assuming the UbiHSOM is represented by a directed graph the mapping is the following:

- **Vertices** – a vertex in the graph represents an UbiHSOM node or a data source.

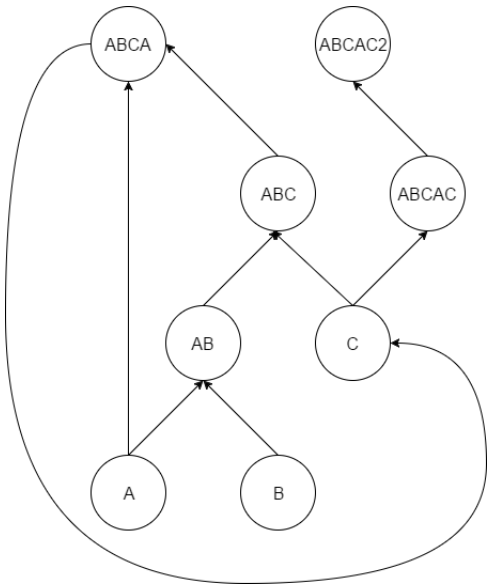


Figure 3.3: UbiHSOM – Dynamic Tier

- **Edges** – an edge in the graph represents the connection (links) between one UbiHSOM node to other.

The graph abstracts everything, the main goal is to connect a vertex/node with another using an edge/link. The link represents a one-way (simplex) channel and nodes receive data from the channel, process it and send it to another link/channel. This is the reason why data sources are also represented as nodes (even tho they are a different kind of node).

Done the decoupling it is now easy to see how to assemble everything. If we want a data source, we create a data source vertex α, if we want an UbiHSOM node we create another vertex β, if we want the data source to communicate with β we create a link $\Rightarrow$ between α and β. Every time a new data sample arrives on α, it uses the link $\Rightarrow$ and sends it to β. Thus we have $\alpha \Rightarrow \beta$. For instance, let's suppose we want a new UbiHSOM node ψ, that receives both data from the α and from β:

$$\alpha \Rightarrow \beta, \alpha \Rightarrow \psi, \beta \Rightarrow \psi \tag{3.1}$$

The same example is illustrated in Figure 3.4. Where α is blue to represent a data source vertex.

In addition to creating vertices and edges, one might want to remove a vertex. However, removing a vertex can lead to an incorrect graph representation. Thus whenever a vertex is removed, the edges that targeted it must be managed. The removal method is called cascade removal, because it removes all the vertices that depend on the vertex that is to be removed.

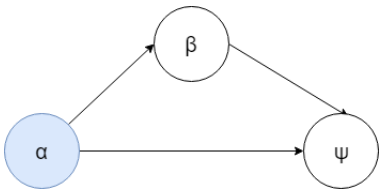


Figure 3.4: UbiHSOM - Model Example

Figure 3.5 illustrates cascade removal. Let C be the vertex to be removed. The vertices that depend on C, i.e. those that are directly or indirectly connect to C, are the vertices D and E. As the figure shows after removal only the vertices A, B and F remain with their respective edges.

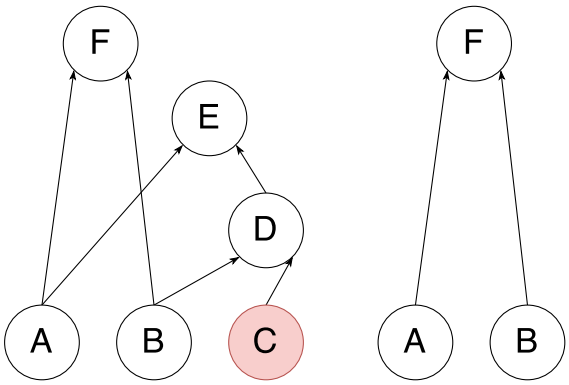


Figure 3.5: Cascade removal of a vertex

Since the UbiHSOM is represented using a directed graph, the model is very flexible therefore it can adapt to many situations. However it poses a concern that is related with the UbiSOM implementation, which is the fact that UbiSOM has fixed dimensionality. Let δ be a data stream that conveys weight vectors α with dimension of four. In order for the UbiSOM model to correctly cope with the data stream data, it must be initialized with a dimension of four, hence matching α dimension. Since δ is a stream of unstructured data, it should not be possible to change the dimensionality of α. Therefore the data that is fed to the first tier nodes should have a similar format.

3.2.1 UbiHSOM concatenation process

There is one peculiar detail regarding the input that a higher tier UbiHSOM node receives. Let A, B and C be nodes. Let ψ and η be data streamers. As represented in equation 3.2.1.

$$\psi \Rightarrow A, \eta \Rightarrow B, A \Rightarrow C, B \Rightarrow C (3.2)$$

Node C will always receive its input whenever A or B process a data point, thus there is no guarantee that node A and B will send their output at the same time, hence (with the current model), C receives inputs asynchronously. According to[29], a HSOM variant is only considered a HSOM if the input of a higher tier node is the concatenation of the outputs of the lower tier nodes. Therefore, the current model does not fulfill this requirement because C receives its inputs asynchronously whenever A or B send an input. In order to fulfill the concatenation requirement, the outputs of the lower tier UbiHSOM nodes, i.e. node A and B, must be concatenated before they are submitted to node C. We will assume that every inputs that arrive at node C within a time-frame T are considered equivalent and valid to concatenate. The order in which the inputs are concatenated also matters.

Let us say that ψ transmits x,y values every 50 milliseconds and η transmits y,z values every 60 milliseconds. Node A receives x,y data points and outputs the BMU coordinates Ax, Ay (the map coordinates of the most similar prototype) while node b receives y,z data points and outputs the corresponding BMU coordinates Bx, By. Concatenating Ax, Ay and Bx, By will result in a different data point than Bx, By, Ax, Ay, thereby, the ordering of the inputs before the concatenation process is important.

Let us use Figure 3.6 to illustrate an example of this mechanism behavior, where A and B are data streams that emit data points. A emits data every 50 milliseconds whereas B emits data every 60 milliseconds. Considering a 20 milliseconds window and an order of A, B, we can the 20 milliseconds intervals being represented at the top of the figured and labeled W1, W2...W7. For each W the concatenation will be:

- W1 – A1, B1;
- W2 – A2, B2;
- W3 – None, because A2 was received at W2 hence there is no matching data point for B3 in the W3 interval;
- W4 – A3, B4;
- W5 – None, because B2 was received at W4 hence there is no matching data point for A4 in the W5 interval;
- W6 – None, because A4 was received at W5 hence there is no matching data point for B5 in the W3 interval;
- W7 – A5, B6

Concluding, in order to fulfill the concatenation requirement a node that receives input from lower tier nodes, the UbiHSOM must ensure:

- it only considers that the inputs are equivalent within a certain time-frame;
- the order in which the inputs are to be concatenated.

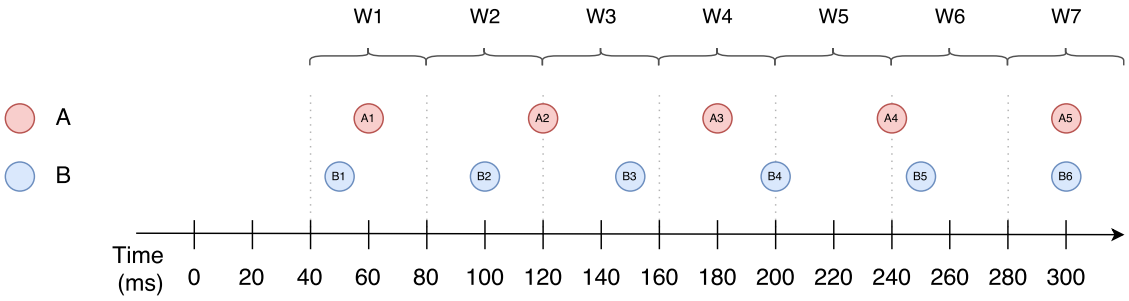


Figure 3.6: Example of the UbiHSOM concatenation mechanism

3.3 Proposed Architecture

In the previous section it was specified how the UbiHSOM is represented and what are its core components. To keep things dynamic and generic all the concepts were decoupled in order to achieve a simple UbiHSOM representation, a directed graph, that represents one UbiHSOM. Decoupling originates functionality decomposition that must be addressed in another way, the goal of decoupling it is to simplify the solution, decomposed functionalities (concerning nodes and links) raise the need for two new services to cope with them. Hence here on this section those functionalities are identified and the services specified.

The graph itself does not has any other functionalities besides adding/removing vertices (nodes) and adding/removing edges (links). However the nodes need to be able to receive data and process it, and the links need to be able to receive the data and send it. Hence this maps directly to two functionalities: create UbiHSOM node and create link. Those two simple functionalities and the auxiliary graph that represents the UbiHSOM are enough to create a service that offers the pretended HSOM generic solution, the UbiHSOM.

At section 2.12 we have the advantages that using the microservices architectural can have when it is appliead to a solutions. Since we have already decoupled the functionalities that the UbiHSOM service should have, it is possible to see that if the microservices architectural style is applied it will highly benefit from its advantages. The style can be applied because the service was already decoupled in independent functionalities (that as we will later see, can be mapped to independent services). These functionalities can be achieve with the help of two auxiliary services. One of them creates UbiHSOM nodes (do not forget that each UbiHSOM node is an extended UbiSOM model), so every time we want to create a vertex, i.e., an UbiHSOM node, we create the vertex in the graph and the UbiHSOM node in the service. This service is called UbiFactory. There will be another service responsible for the links, every time an edge is created in the graph (and thus two vertices are connected), a link would be created in the service. This service is called DataStreamers. The DataStreamers service can also be used to create data source vertices. The UbiHSOM service will then be responsible to make sure the graph and the existing UbiHSOM nodes, data sources and links are synchronized. Each of this service

will expose an API to be consumed, which are defined in sections 4.2 and 4.3.

Figure 3.7 depicts the overall system proposed architecture. The notation used has each service/microservice represented by a hexagon[3]. By observing the figure we can identify four services: UbiFactory, DataStreamers, UbiHSOM and Mongo. All of them communicate through a distributed bus. The UbiFactory service is responsible for creating UbiHSOM nodes (extended UbiSOM instances) on demand. DataStreamers service enables the creation of links and data sources. Either UbiFactory and DataStreamers are independent services that are used by the UbiHSOM service to create UbiHSOMs. The Mongo service is responsible to persist data from the UbiFactory, DataStreamers and UbiHSOM services services.

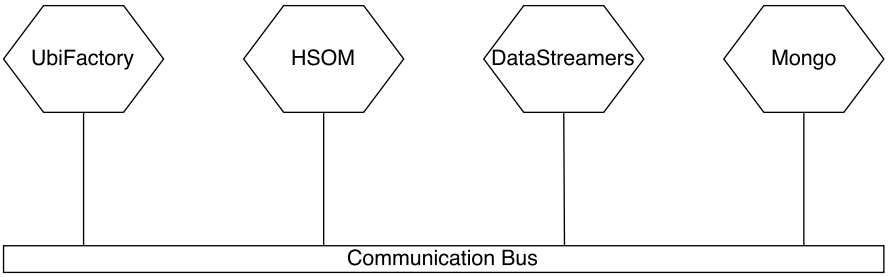


Figure 3.7: Overall Architecture

This kind of architecture follows the microservice style, by having decoupled autonomous services that communicate with one another to achieve a certain result. Here we can see that the UbiHSOM service interacts with UbiFactory and DataStreamers services in order to achieve what it is pretended: an UbiHSOM structure. Whenever a vertex is created a new UbiHSOM node is also created, the same goes for the edges, whenever one is created it originates a new DataStreamer object.
In short each service is responsible for:

- **UbiFactory** – creates and deploy UbiHSOM nodes instances, i.e., each UbiHSOM node is an extended version of an UbiSOM instance. Hence whenever an UbiHSOM node is deployed in the system an UbiSOM instance with a few additional functionalities is deployed in the system. Each UbiSOM instance is executing the UbiSOM algorithm;
- **DataStreamers** – responsible for creating DataStreamers instances of various types as seen in section 4.2.1. The basic role of a DataStreamer is to receive data from A and forward it to B.
- **UbiHSOM** – by working together with UbiFactory and DataStreamers enables the creation of UbiHSOMs.
- **Mongo** – this is an auxiliary service that is used by the others to persist data in the NoSQL MongoDB introduced in section 2.9.

The services are detailed in chapter 4.

Such decentralized architecture offers high maintainability because new features can be added to both UbiFactory and DataStreamers services without compromising the UbiHSOM service.

Besides that it also offers high scalability. If the service demand is high and the performance starts to degrade, either service can be horizontally or vertically to cope with such workload.

By having a single autonomous service that is responsible for deploying UbiHSOM nodes, one is not bound to the UbiHSOM solution. It is also possible to create single UbiHSOM nodes that cope with a specific problem that is not in a HSOM context. Another advantage is that new solutions that rely on UbiSOM capabilities can be easily implement using this service. Not only SOM solutions but also other HSOM solutions by conjugating DataStreamers with UbiFactory and/or a new service that deals with a specific problem.

The architecture offers a wide range of possibilities when it comes to creating UbiSOM based solutions.

Chapter 4

Application Programming Interface

Typically, An application is composed by various components that communicate with each other to fulfill a certain requirement, thus for the communication to take place they need to have some type of protocol or specification because it may want to expose some features to enable other application components to use it. It is common that companies like Microsoft, Apple, Amazon, Facebook, Netflix, Spotify, Instagram and many others, offer some features so developers around the world can communicate and use them in their application (e.g. Google Maps). Developers benefit a lot from this services because they make it easier for developers to take the most out of the features that the top tier tech companies offer, for instance, many websites and mobile apps use Facebook or Google login API.

API or Application Programming Interface is the interface exposed by an application to be used by developers. Normally the API's lie between two software components, offering developers a clearly defined set of methods to interact with the API components. There are various types of APIs: routines specifications, data-structures, objects, databases, operating systems, web-services (REST), SOAP services, among others.

On the previous chapter three core services (DataStreamers, UbiFactory and UbiHSOM) were identified in order to achieve the desired solution. It is not in the scope of this chapter to explain implementation details (as specification and implementation are completely different things), therefore those are explained in chapter 5. This chapter aims at specifying and describing the APIs as well as present usage examples of how they can be used to achieve real-world data analytics solutions.

4.1 Introduction

This thesis exposes APIs for developers to interact with and use them in their applications so they can take advantage of self-organizing maps data analysis and visualization power. An API can have many forms, depending on the business domain functional and non-functional requirements. Three distinct REST APIs were developed and each of them has its own purpose and aims to target the challenges presented on the previous chapter. The following defined APIs follow the REST architectural style previously described in section 2.11, one can call them REST APIs.

The chapter starts with the specification of the DataStreamers and UbiFactory APIs that are lately used by the UbiHSOM API. The latter orchestrates the formers in order to achieve the functionalities previously presented in chapter 3. Each section is dedicated to describe the respective APIs, it starts with a brief introduction and then delves into its underlying models and exposed resources, ending with usage examples. The examples are given by using three tools, namely: *curl*, *wget* and *Postman*. They are all tools that can send HTTP requests and receive the respective responses. *curl* and *wget* are Unix-based command line tools whereas *Postman* is a *Google Chrome* browser extension with the same purpose as the others but it is offers a graphical user interface. Python scripts that programmatically use the API will also be presented. All of the *JSON* models needed to interact with the APIs are specified in the appendix A.

4.2 Data Streamers API

On the previous chapters we identified the need of having links between nodes that transmit data from one node to another, hence, this section will be focused on specifying an API for a service that it is able to create streams that can channel data from one point to another, i.e., data streamers.

Join with the previous paragraph this thesis context a data streamer represents a channel that streams data asynchronously and continuously. The system offers three types of streams:

- **DB** – a stream that pulls stationary data from a chosen data set and simulates a real-time stream;
- **Proxy** – this stream receives data from a given system address and sends it to a chosen address;
- **Zip** – it merges various streams inputs from different data streams and sends the concatenation of those inputs.

The terms data streamer and streamer will be used interchangeably through out this section.

4.2.1 Data Streamers Models

In this sub-section the models of the different data streamers will be defined (and represented with an UML object diagram).

4.2.1.1 DB

In chapter 3 we saw that there were nodes (called data sources) that were responsible to stream input to the UbiHSOM nodes. This section will describe the DB data streamer model which embodies the data source nodes previously mentioned. The DB data streamer has the purpose of simulating a real-time data stream by pulling data from a desired data set (according to its configuration) and stream the data to an output channel. Most of the data sets are stored in a SQLite3[46] database and the ones available are:

- Iris
- Cube
- Financial Data

The Financial Data[64], Cube[67] and Iris[51] data sets are stored in a *SQLite* database[46], whereas the Geometric Shapes (e.g., square, triangle, diamond, pentagon, hexagon) data set is stored in a MongoDB database. Two types of DB data streamers were implemented, the random DB data streamer that randomly pulls data points and the sequential DB data streamer, which sequentially pulls data points from the data set. The basic model of the DB data streamer has the following fields:

- id: the streamer identifier given by the system;
- type: the streamer type, in this case its always DB;
- db: the name of the data set to pull data from;
- timer: the time in milliseconds from which the streamer pulls data from the database, i.e., let γ be DB.timer and η be DB.randomness (which is a field of the random DB data streamer as we will later see). Each γ milliseconds the streamer pulls η tuples from the given data set DB.db and sends it to the output address;
- selectors: the features (or columns) of the data set we wish to get from the database, that if left empty will return tuples with all features from the data set (e.g. for a data set that has the columns A, B and C, if the selector is equal to A and C, the data streamer will only output the values regarding those two columns (A and C) for each tuple it sends);
- out: the address given by the system to where the streamer will output data to;

- pull-type: the type of pulling mechanism wanted, either random or sequential;

The random DB data streamer extends the model above and adds the field randomness, which is the number of tuples to pull from the data set at each cycle. Whereas the sequential data streamer uses the raw base model with no additional fields.

The DB data streamer will later be used in the Chapter 6 case studies to send data to UbiHSOM nodes in order to train them.

4.2.1.2 Proxy

The Proxy streamer receives data from the user given input address and sends it to the user given output address just like a proxy, thus creating a direct channel from "in" to "out". It is represented by the following model:

- id: the streamer id given by the system
- type: the stream type, in this case its always PROXY
- in: the streamer channel input system address, given by the user
- out: the streamer channel output system address, given by the user

The proxy data streamer will later be used to connect the DB data streamers with UbiHSOM nodes, thereby representing the links that connected nodes that were introduced in chapter 3.

4.2.1.3 Zip

Chapter 3, section 3.2.1 identified the problem of having a high level UbiHSOM node receiving input asynchronously (that led to an erroneous representation of the underlying data), which required a mechanism that concatenated the various inputs the UbiHSOM node received before submitting them to the learning process. The Zip data streamer aims to concatenate various inputs from various data streams (an example is given below), it is represented by:

- id: the streamer identifier given by the system
- type: the streamer type, in this case its always ZIP
- order: the order from which the streamer will concatenate the input
- timer: time frame in which the messages are considered to be equivalent

As an example let us assume that we have a data set named foo with the columns A, B and C (this example uses the DB data streamer model specified in subsection 4.2.1.1). Let γ be a sequential DB data streamer that pulls data from foo with the selector equals to A and B. Let η be another sequential DB data streamer that pulls data from foo with

Method	URL	Description
GET	/datastreamers	Returns all DataStreamers
GET	/datastreamers/:id	Returns the DataStreamer with the id = :id
POST	/datastreamers	Creates a new DataStreamer
DELETE	/datastreamers/:id	Deletes the DataStreamer with the id = :id
PATCH	/datastreamers/:id	Sends data to the DataStreamer with the id = :id

Table 4.1: Data Streams REST API

its selector equals to B and C. Both γ and η have the same timer value, which is 100 milliseconds. If we want to concatenate both outputs - A,B and B,C - we create a new Zip data streamer with its order value equal to a list that has γ id and η id (note that if we desired the concatenation of B,C and A,B instead of A,B and B,C, the order should be η and γ), and a timer equal to 50 milliseconds. Every 50 milliseconds the newly created Zip data streamer would concatenate the inputs it has received in that time frame and send the resulting concatenation to the output channel. A more detailed explanation with implementation details is give in section 5.3.

This data streamer will later be used by the UbiHSOM service to concatenate the outputs of two first tier UbiHSOM nodes and send the concatenation as the input to a second tier UbiHSOM node.

4.2.2 The API

This section will define and describe the Data Streamers service API, i.e., the resources (endpoints) will be listed and instructions on how to manipulate them will be given as illustrated in Figure 4.1. It is important before moving on to this chapter to be familiar with client-server architecture (section 2.10), REST style (section 2.11) and the Data Streamers service models (section 4.2.1). Some JSON or XML background is also recommended.

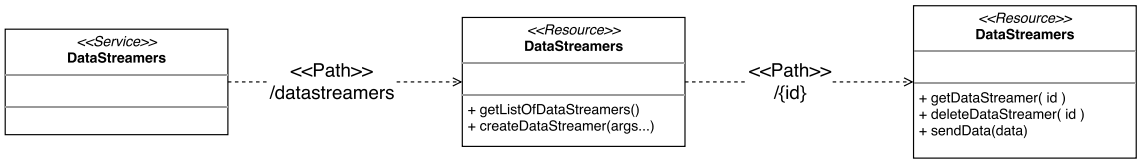


Figure 4.1: UML – DataStreamers service REST API

Table 4.1 presents the available endpoints. Each table row has three columns (method, URL and description) that describe what HTTP method is used to access the resource, then the path/URI/URL in which the resource is available and lastly a description of what the resource is, respectively. It gives a nice overview of the API endpoints and for those who are used to work with them it might be enough, but for a better understanding let us detail the API resources a bit more.

The first resource, GET /datastreamers, can be accessed via HTTP method GET and (as it says), it returns all DataStreamers. DataStreamers are the data streams objects that

were previously created, such objects are defined in subsection 4.2.1. In analogy with *JAVA* objects, this endpoint would be similar to a method *List<DataStreamers> getAllDataStreamers()*, i.e., a method that returns a list of *DataStreamers* objects that can be manipulated. In this case, since it is a REST API, instead of returning a list of *JAVA* objects it returns a list of *JSON* objects that have exactly the same representation and can be further manipulated using the others endpoints.

For GET /datastreamers/:id we have a resource that returns the data streamer object with the given identifier which is passed in the request URL. When we send a request to the server, we should pass the desired data streamer identifier in the request path, if the request is accepted it returns a *JSON* object that internally represents the object. Usage examples are given in section 4.2.3.

The resource POST /datastreamers is used to create a data streamer object. This request differs from the others because it uses the HTTP method POST, i.e., GET requests are just requests to ask the server to send us information, POST requests expect us to send a body attached to it, being the body (in this case) a *JSON* object. This *JSON* object has the information needed to create the desired data stream as shown in section 4.2.3. Using the *JAVA* analogy, this kind of request would translate to a method similar to this *DataStreamer createDataStreamer(Args ...)*, a method that returns a DataStreamer and has a list of arguments. If the request is succeeded an object is created and returned to the client with the status code 200 – Ok (please refer to the HTTP status code list[32]).

When a data streamer is created, an entry in the database is added regarding the newly created data streamer object, moreover, a data streamer instance is also deployed in the system. The deployed instance will then be running as long as the respective entry exists in the database.

The resource DELETE /datastreamers/:id concerns the deletion (and undeployment) of a data streamer. Similar to the GET method but with side effects, i.e., it does not return the data streamer object with the given identifier, instead it deletes it. If it succeeds on deleting and undeploying the data stream from the system, the client receives an HTTP response with the status code 204 – No content.

Finally, the PATCH /datastreamers/:id resource regards the the *sendData(data)* method depicted in Figure 4.1. The service is supposed to receive a *JSON* object with data and it sends that data to the data streamer with the given identifier.

Now that this service API is described the next sub-section provides examples of how to use the API.

4.2.3 Usage Examples

This sub-section provides examples on how to use the Data Streamers API. The examples will be given using three basic tools: *wget*, *curl* and Postman. There is no requirement in what tool should be used to interact with the API, any tool can be used as long as it follows the HTTP protocol. *wget* and *curl* are classic command line tools whereas *Postman* is a *Google Chrome* (the browser) extension and it is a GUI-based tool. It is not in the scope of these examples to teach how to use the tools extensively so for each resource only one of the referenced tools will be used as an example, since it can be easily adapted to the others.

GET /datastreamers – curl
To get all the data streamers currently created in the system using *curl*, simply write this command to the command line/terminal/prompt:

curl http://localhost:8585/datastreamers/

The output should return a JSON array with all the data streamers in the system.

GET datastreamers/id – curl
It is possible to view a single data streamer in detail. The command bellow outputs the data streamer with the identifier *58b7545d21621f10281ab859* JSON representation.

curl -s http://localhost:8585/datastreamers/58b7545d21621f10281ab859

If a data streamer with the given identifier exists, a JSON object representing it is returned to the user.

POST /datastreamers – Postman
Here Postman is used to consume the API. An example is presented for each type of data streamer and each of them have a figure illustrating the example in Postman, please refer to appendix section A.1. Essentially whenever one posts something, the request must have a body. In this case the body must be a JSON object that follows the models presented on sub-section 4.2.1.

The figures in the appendix A section A.1 depicts the API consumption using Postman. All the essential information to understand the examples are on the Figures.

DELETE /datastreamers/id – wget
Here, *wget* is used to delete a previously created echo data streamer. The data streamer identifier must be known in order to delete it, in this case the identifier is *58b72d5021621f10281ab83f*. The identifier is passed in the request path as shown below:

wget −−method=DELETE http://localhost:8585/datastreamers/58b72d5021621f10281ab83f

If the object was successfully deleted, the response message will have the *204 – No content* HTTP status.

4.3 UbiFactory API

UbiFactory API exposes a set of resources to manipulate UbiHSOM nodes (extended UbiSOMs) giving the user the ability to create and deploy UbiHSOM nodes that can thus learn from data. The first sub-section explores the normalization feature. The second defines the models used by the API, then the API is specified in sub-section 4.3.3 and lastly some usage examples are given.
Let's recall what an UbiSOM is. In short, an UbiSOM is a mesh of prototypes/neurons, (let's call this, the underlying map) with weights that adapts to the incoming data. UbiSOM was missing some features, thus, each UbiHSOM node has an UbiSOM associated with the following new features:

- **normalization** – UbiHSOM node offers built-in normalization, whenever a new UbiHSOM node is created it is possible to choose the type of normalization to be applied to the incoming data, more on this on section 4.3.1;
- **hit-counting** – one of the useful things to know when one is working with any type of SOM, is to know how many times one prototype was considered BMU, this statistical value is called hit-count and each prototype has a hit-count value associated with it;
- **prototype distance calculation** – also known as unified distance matrix, it represents the distance between one prototype and its neighbours;
- **weight labels** – even though UbiSOM falls in the unsupervised learning category, the features still have names/labels, thus this functionality binds the prototypes weights with their labels.

4.3.1 UbiFactory Normalization

UbiSOM receives input without caring if the data was normalized or not, ideally the data should already be normalized, but this does not always happens. The UbiSOM domain is $[0, 1]$, however the incoming data can assume many forms and domains which are impossible to predict, making it mandatory to have normalization at the node level.

Why is it at the node level? – One could expect that before sending the data, the data source should be concerned about normalizing it, but that's not ideal, because the data source would have one more burden which would degrade performance and such requirement is hard to introduce. For instance, let us assume the data source is a simple thermal sensor that collects heat temperatures. In this case it does not make sense to expect the

device to normalize the data because there is more than one thermal sensor available in the market from different manufacturers. The same is true for a RSS feed. Since UbiSOM already does computation whenever a new data sample arrives, it does not strain it to normalize the sample before the learning from it.

There are two types of built-in normalization: none and feature scaling.

None – No computation is applied, the raw data sample is fed to the learning model. Caution must be taken because in order to get healthy results the incoming data values must be between $[0, 1]$.

Bounded Feature Scaling – simple min/max normalization based on min/max values given by the user.

4.3.2 UbiFactory Models

Since the UbiHSOM node is an extension of UbiSOM, its model follows the specification given by B. Silva [58] The UbiHSOM node model is as follows:

- **name** – the name of the instance given by the user
- **weight-labels** – the names of the underlying map prototype features
- **width** – the width of the underlying map
- **height** – the height of the underlying map
- **dim** – the prototype weight vector length
- **eta-i** – initial learning rate for ordering state
- **eta-f** – final learning rate for ordering state
- **sigma-i** – initial neighborhood rate for ordering state
- **sigma-f** – final neighborhood rate for ordering state
- **beta-value** – drift function weight factor
- **normalization** – this field contains a normalization model, there are four types of normalization: none and feature scaling.
- **window** – size of sliding window / length of ordering state
- **in** – the address from where the node receives input from, this field is generated by the system

Method	URL	Description
GET	/ubis	Returns all UbiHSOM node
GET	/ubis/:id	Returns the UbiHSOM node with the id = :id
POST	/ubis	Creates a new UbiHSOM node
DELETE	/ubis/:id	Deletes the UbiHSOM node with the id = :id
PATCH	/ubis/:id	Sends data to the UbiHSOM node
GET	/ubis/:id/data	Returns UbiHSOM node data
GET	/ubis/:id/data/hitcounts	Returns UbiHSOM node prototypes hit-counts
GET	/ubis/:id/data/umat	Returns UbiHSOM node U-Mat distance matrix
GET	/ubis/:id/data/weights	Returns UbiHSOM node prototypes weights

Table 4.2: UbiFactory REST API

- **out** – the address from where the node sends output to, this field is generated by the system

The values differ from case to case therefore the parameters must be fine tuned in order to better suit the case [58].

The normalization item of the list above regards the various normalization methods that the solution offers which are represented by the following models:

- **None**
- **Bounded Feature Scaling**

  min – the minimal value that it accepts, given by the user

  max – the maximum value that it accepts, given by the user

4.3.3 The actual API

Before trying to understand this sub-section one must first be familiar with client-server architecture (section 2.10), REST style (section 2.11) and UbiFactory models 4.3.2.

Figure 4.2 shows all the resources this API exposes. A more detailed view is presented in Table 4.2. The API offers nine resources that can be used to interact with it.

If the reader is not familiar with this type of analysis please refer to sub-section 4.2.2 where a more detailed analysis is given for a similar API.

GET /ubis – Returns a list with all the currently existing and running UbiHSOM node instances in the system.

GET /ubis/:id – Returns an UbiHSOM node with the given identifier in *JSON* format.

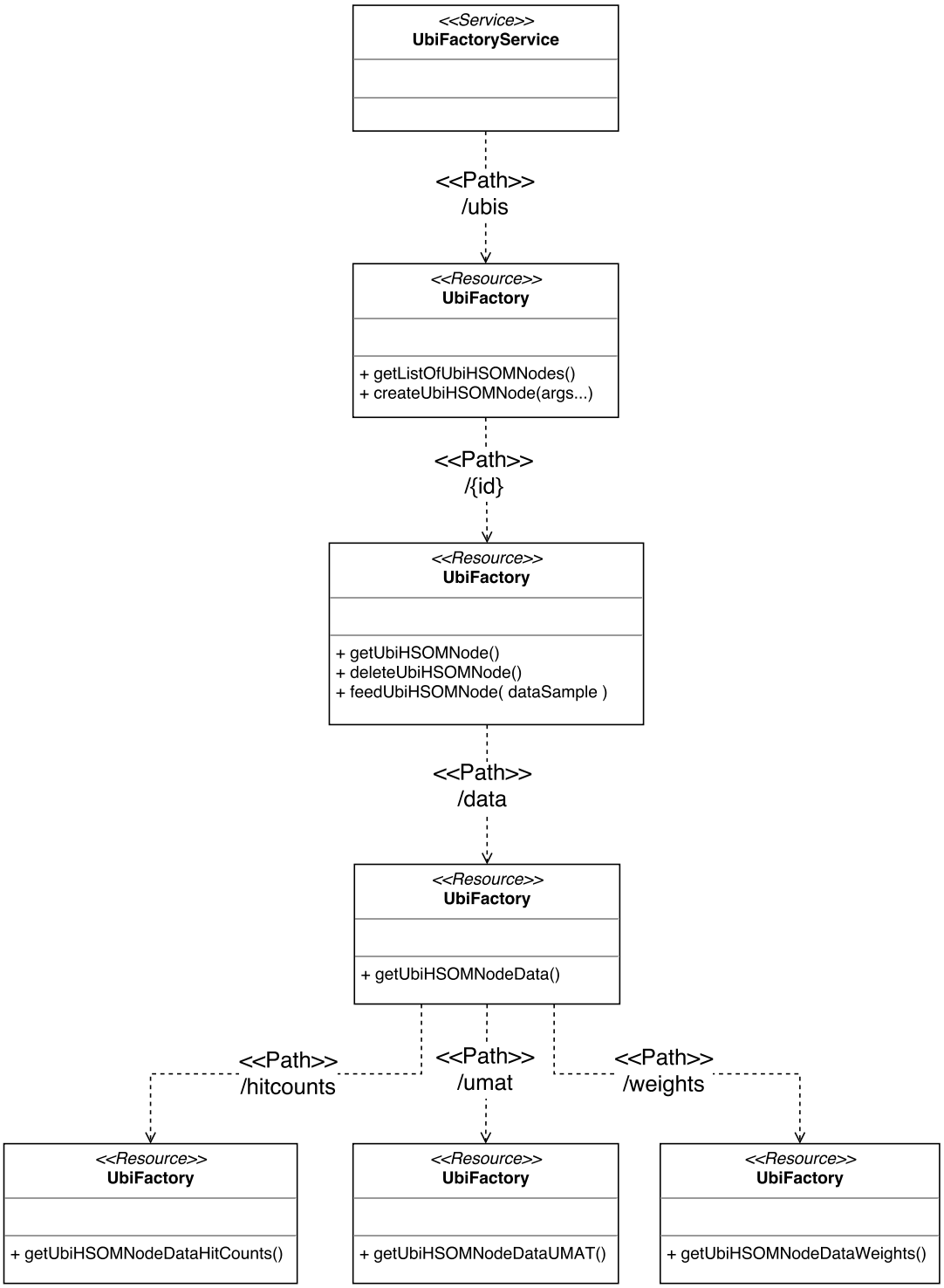


Figure 4.2: UML – UbiFactory service REST API

POST /ubis – The service expects the user to send a JSON object attached to the request that follows the model defined in 4.3.2. The service replies with JSON object that represents the newly created UbiHSOM node instance.

DELETE /ubis/:id – The service deletes and undeploys from the system the running UbiHSOM node instance with the given identifier.

PATCH /ubis/:id – This resource serves the purpose to send data to serve as input to the UbiHSOM node instance with the given identifier. It expects the user to send that data in a JSON object with the following format:

- **data** – [[sample-1], [sample-2], [sample-3], ..., [sample-N]], where each sample-i es an array of numeric values (e.g. integers, floats or doubles)

Remember that the data samples must have the same dimensionality as the UbiHSOM node. If succeeded the reply will have the HTTP status 202 – Accepted.

GET /ubis/:id/data – Returns a list of all JSON representations of the UbiHSOM node instance prototypes. Each prototype is a JSON object with the following information:

- **x** – integer
- **y** – integer
- **label** – string
- **hit_count** – integer
- **distance** – double
- **weights** – JSON array with the respective prototypes weights. An array of doubles.

GET /ubis/:id/data/hitcounts – Returns a JSON array containing the hit-count of each prototype. The array follows a column|row lattice.

GET /ubis/:id/data/umat – Returns a JSON array containing the distance of each prototype to its neighbours. The array follows a column|row lattice.

GET /ubis/:id/data/weights – Returns a JSON array containing the weight vectors of each prototype. The array follows a column|row lattice.

4.3.4 Usage Examples

The examples of the usage of this service do not differ much from the ones previously presented in section 4.2.3 or the ones in A.1. For a real life example of the usage of

the UbiFactory service, please refer to section 6.1. However, a simple JAVA program (please see section 5.6) was developed to illustrate that the UbiFactory service can be used programmatically by developers. The program uses the Unirest[68] library to make the HTTP requests.

The figures in the appendix A section A.2 depicts the API consumption using Postman. All the essential information to understand the examples are on the Figures.

4.4 Ubiquitous Hierarchical Self-Organizing Map API

Section 3.2 of chapter 3 shown how the UbiHSOM model is represented and how it is composed. Later on, at section 3.3, it was explained how the desired UbiHSOM solution could be achieved by the already defined services (in section 4.2 and 4.3) and section 3.2 described the mapping between vertices and UbiHSOM nodes as well as edges and DataStreamers. This section aims to detail the representation by specifying the service models that represent the formerly introduced concepts. It first starts by exposing a use case diagram that gathers some requirements of this service, identifies internal and external influential factors as well as the interaction between requirements and actors. Then the models used by the service are described, the API is specified and lastly some usage examples are given.

The use case diagram illustrated in Figure 4.3, shows actors interaction with the service requirements and it is possible to identify four actors:

- **Mongo** – this actor represents the storage system, it plays an important role because many requirements use it to store information;
- **DataStreamers** – it embodies the previously specified DataStreamer service (please refer to 4.2);
- **UbiFactory** – it represents the UbiFactory service that was early describe in section 4.3;
- **User** – in this context, the user is a generalization, it was not generalized in the diagram in order to keep the figure simple and legible. Nonetheless, the user can be: a developer, an analyst, a data scientist, a mobile app, a backend service, basically the user can be any kind of application (or human) that uses the services to an end. For instance, section 4.3.4 presented a JAVA program that interacts with the UbiFactory API, in this context, this JAVA program can be considered an user.

The actors interact with the following requirements:

- **Create new UbiHSOM & Delete UbiHSOM** – the user can create and delete various UbiHSOM models, whenever either of the actions occurs the *Mongo service* is always requested.

- **Get UbiHSOM** – the user requests information regarding one UbiHSOM model, the system replies after consulting the *Mongo service*.
- **Add Vertex** – the UbiHSOM model where the vertex is to be added is requested from the *Mongo service* – through the *Get UbiHSOM* case – after that an UbiHSOM node is deployed via the *UbiFactory service* (*Deploy UbiHSOM node* case), and then the new vertex is inserted to the UbiHSOM model and it is stored in the database.
- **Deploy UbiHSOM node** – auxiliar case to deploy an UbiHSOM node by using the *UbiFactory service*.
- **Remove Vertex & Undeploy UbiHSOM node** – similar to the *Add Vertex* requirement but instead of inserting a vertex, it removes it and undeploys the respective UbiHSOM node (*Undeploy UbiHSOM node* requirement).
- **Add Edge & Deploy DataStreamer** – identical to the *Add vertex* and *Deploy UbiHSOM node* requirements, but instead of vertices and UbiHSOM nodes this case deals with edges and DataStreamers.
- **Remove Edge & Undeploy DataStreamer** – the behavior is the same as the *Remove Vertex* and *Undeploy UbiHSOM node* cases however this case concerns edges and DataStreamers over vertices and UbiHSOM nodes.

The diagram shows how the user is supposed to interact with the system and how the system responds to such interaction by communicating with the other services. It also helps understanding the next sections that aim to described the associated models (vertices, edges and UbiHSOM) as well as relate the resources specified in the API with the use case requirements.

4.4.1 UbiHSOM Models

This sections focuses on defining the models used in this service. Recall that the UbiHSOM service operates in conjunction with both, the UbiFactory and the DataStreamers services, i.e., as expected, one must be familiar with their associated models that were formerly specified in section 4.3 and 4.2, respectively.

As early mentioned, this service has two core concepts, vertices and edges since it is represented by a directed graph (see section 3.2), moreover, a vertex maps directly to a UbiHSOM node and an edge to a DataStreamer. Therefore the underlying models of the UbiHSOM service for vertices and edges are no different from the already defined models for the UbiHSOM nodes and the DataStreamers (see section 4.3.2 and 4.2.1). However, the UbiHSOM general model is different, it is composed by the following fields:

- **id** – it is the identifier of the UbiHSOM instance;
- **name** – a representative name of the UbiHSOM instance;

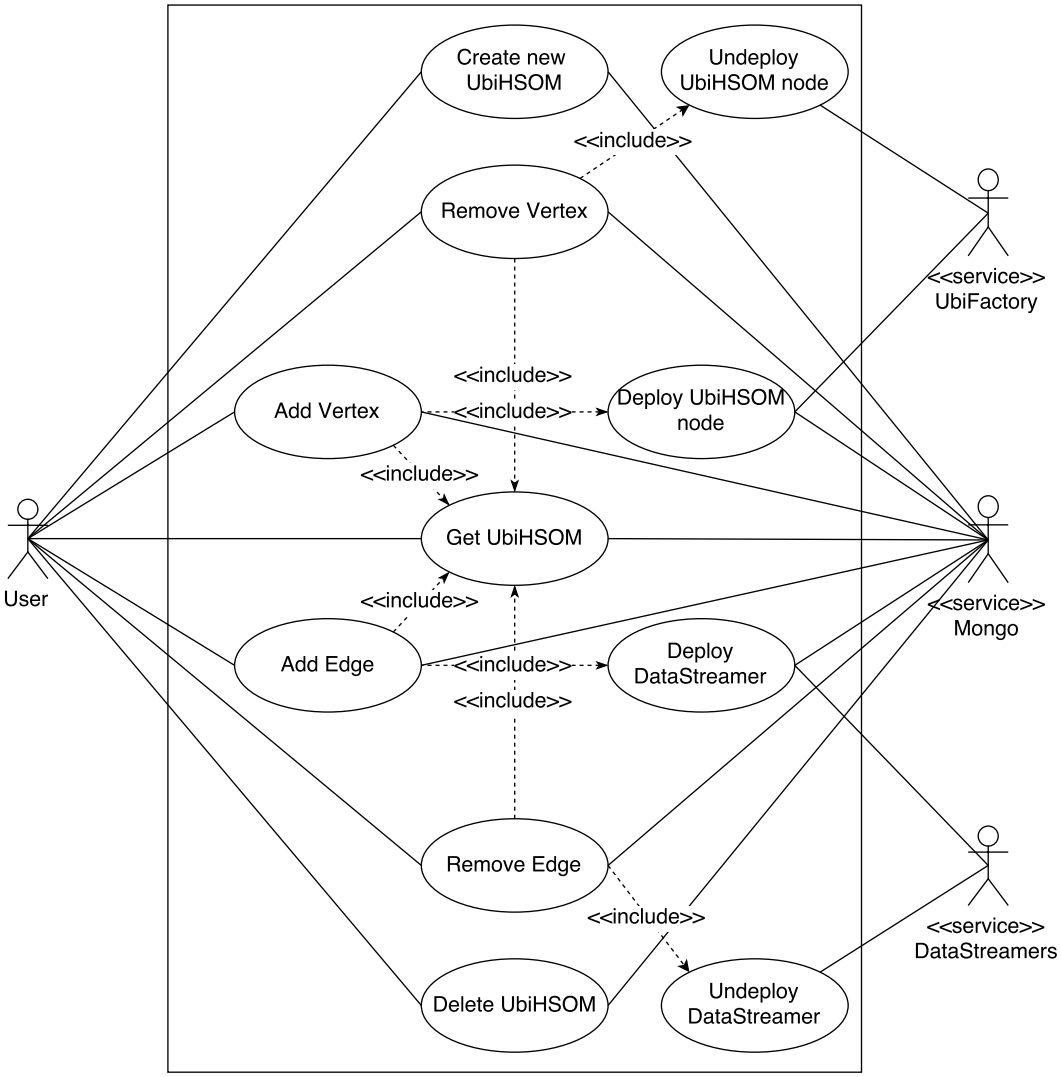


Figure 4.3: UbiHSOM – Use Case Diagram

- **nodes** – a list of all the vertices/UbiHSOM nodes in the graph. Each vertices/UbiHSOM nodes follows the model of the UbiHSOM node described below.

- **graph** – has two lists, one list (nodes) that contains the identifiers of all the vertices currently in the graph and another list (edges), that contains all the edges in the graph. The nodes list is a list composed only by the nodes identifiers whereas the edges list is composed by edges objects, which is described below.

- **nodes** – a list with the vertices of the UbiHSOM instance, each vertex object on the list has the following field: identifier and a nested list with the identifiers of the vertex outgoing vertices;

- **graphs** – a list with the edges of the UbiHSOM instance, each edge has a *source* and *target* field with the identifier of the source vertex and the target vertex.

In Chapter 3 a simple directed graph was proposed as the data structure that represents an UbiHSOM. A graph is composed by nodes and edges between nodes. We have seen that the nodes in the graph represent an UbiHSOM node which, under the hood, extends and uses the UbiSOM algorithm, whereas the edges (or links) connect nodes and have the role of transmitting data between them. We have also seen that to merge UbiHSOM nodes we need to have a mechanism that concatenates both UbiHSOM nodes outputs, the Zip data streamer was developed and previously explained (see section 4.2.1.3) to achieve that purpose. Hence the UbiHSOM node is represented by the following model:

- **id** – the identifier of the node
- **zipper** – the identifier of the node zip data streamer
- **order** – the order of the node zip data streamer
- **timer** – the timer used by the node zip data streamer
- **consumers** – a list of the vertex outgoing vertices
- **steamers** – a list of all the data streamers connected to this node
- **model** – the model of the UbiHSOM node model (please refer to section 4.3.2)

In section 3.1 we have seen the roles of the links and the need for them. They serve as the communication mean between UbiHSOM nodes. Under the hood, a link is simply a Proxy data streamer (it is possible to see the similarities when comparing both of the models) that connects the source UbiHSOM node output channel to the target UbiHSOM node zipper. The links are represented by edges in the graph and the edges have the following model:

- **source** – the identifier of the source vertex
- **target** – the identifier of the target vertex

4.4.2 The actual API

Figure 4.4 gives an overview of the API resources and its functionality. The /ubihsoms path offers the *getListOfUbiHSOMs()* and *createUbiHSOM()* functionalities. The former returns all of the existing UbiHSOM models that are currently in the system. The latter creates a new UbiHSOM instance.

The /id path refers to /ubihsoms/id and has two functionalities: *getUbiHSOM()* and *deleteUbiHSOM()*. The name of the functionalities are pretty self explanatory. *getUbiHSOM()* returns an UbiHSOM instance with the given identifier. As for the *deleteUbiHSOM()* functionality, it is expected to delete the UbiHSOM instance with the given identifier.

For the /nodes path there is only one operation that returns all the nodes (regarding the UbiHSOM with the given identifier), which is *getAllVertices()*. A level down in the path, exists the /nodeid that offers the *getVertex()* and *deleteVertex()* functionalities, the former aims to return the vertex with the identifier *nodeid* and the latter, deletes a vertex with such identifier (all regarding the UbiHSOM with the given identifier).

As for the /edges path, it offers the *createEdge()* and *deleteEdge()* functionality, the request body expected on both functionalities is a JSON object with the fields:

- **source** – the identifier of the source vertex
- **target** – the identifier of the target vertex

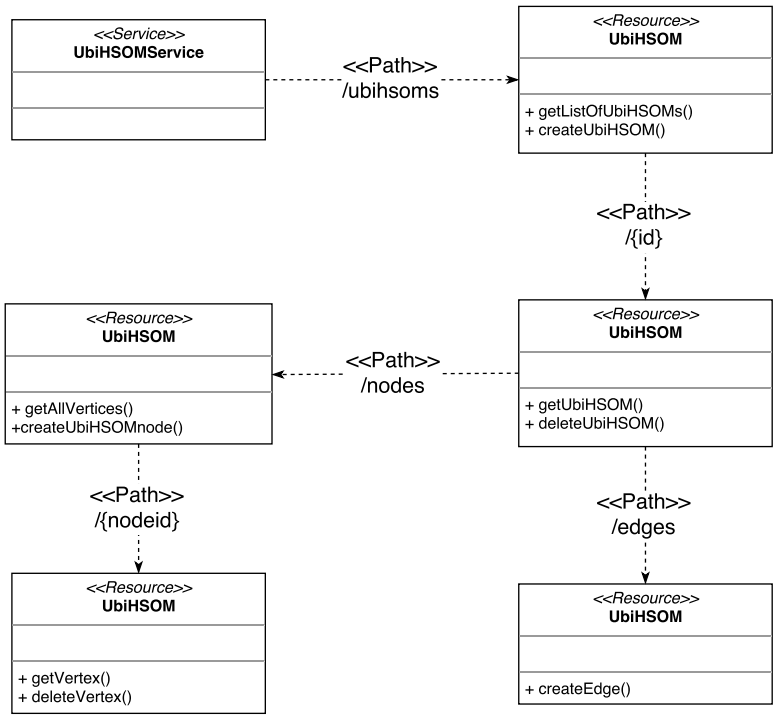


Figure 4.4: UML – UbiHSOM Service REST API

All of the previous discussion about this API specification is given in Table 4.3.

Under the hood, every time a node is created the UbiHSOM service uses the UbiFactory service to create a new UbiHSOM node instance, moreover, every time an edge is created, the UbiHSOM service uses the DataStreamers service to create a Proxy data streamer. The next chapter gives a more detailed perspective on this matter.

4.4.3 Usage Examples

The section started by presenting in Figure 4.3 a use case diagram that illustrated the user interaction with the UbiHSOM service. The example presented in this sub-section relies on the use case diagram use cases to describe the example solution.

Method	URL	Description
GET	/ubihsoms	Returns all UbiHSOMs
POST	/ubihsoms	Creates a new UbiHSOM
GET	/ubihsoms/:id	Returns an UbiHSOM
DELETE	/ubihsoms/:id	Deletes an UbiHSOM
POST	/ubihsoms/:id/nodes	Creates a new UbiHSOM node
GET	/ubihsoms/:id/nodes/:nodeid	Returns an UbiHSOM node
DELETE	/ubihsoms/:id/nodes/:nodeid	Deletes an UbiHSOM node
POST	/ubihsoms/:id/edges	Creates an edge between two UbiHSOM nodes

Table 4.3: UbiHSOM REST API

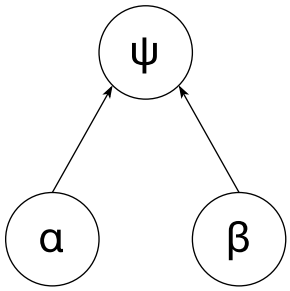


Figure 4.5: UbiHSOM – Example

Figure 4.5 exposes a simple UbiHSOM representation example where one wants to have two first tier maps and join them into a second tier one, the question is, how can the early specified API be used to achieve the former representation? To describe the example solution, the notation previously used in section 3.2 is applied. In order to meet the representation illustrated on Figure 4.5 the user must create three vertices, α, β and ψ, two links and connect them, for instance:

$$\alpha \Rightarrow \psi, \beta \Rightarrow \psi \tag{4.1}$$

This model can be achieve with the presented use cases in Figure 4.3, with the following steps:

1. Create new UbiHSOM
2. Add Vertex (α)
3. Add Vertex (β)
4. Add Vertex (ψ)
5. Add Edge ($\alpha \Rightarrow \psi$)
6. Add Edge ($\beta \Rightarrow \psi$)

Even though this model alone meets the desired UbiHSOM representation, it accomplishes nothing because there is no data source linked to any of the first tiers nodes. For instance, the early introduced DB data streamer (see section 4.2.1.1), can be used as a data source, since it pulls data from one of the available data sets. Let θ be a data source. The final model would look like:

$$\theta \Rightarrow \alpha, \theta \Rightarrow \beta, \alpha \Rightarrow \psi, \beta \Rightarrow \psi \tag{4.2}$$

Please note that every time a vertex is added, one UbiHSOM node is deployed through the UbiFactory service. The same happens to the edges, every time one edge is created a data streamer is deployed.

Although the example is rather simple, it clearly illustrates how the service can be used to build more complex examples. It is just a matter of adding vertices, connect them and feed data to the nodes through the most suited data source for the scenario.

A few Python scripts that consume the UbiHSOM service API were written and they are publicly available (please refer to section 5.6. The scripts use the API to handle the case studies later described in section 6.2.

The figures in the appendix A section A.3 depicts the API consumption using Postman. All the essential information to understand the examples are on the Figures.

Chapter 5

Implementation

The nature of the problem led to a decentralized solution which influenced design choices. Previously it was discussed how it led to the use of microservices architectural style thus impacting the technology to be used. Various technologies were picked and discussed in order to find out which of them better tackles this specific problem, hence the Vert.x toolkit was the result from that discussion because it meets all the previously stated requirements in chapter 2.

Chapter 4 specified the REST APIs and their associated models of the services that were previously identified in chapter 3, moreover, usage examples were also given in order to illustrate how they should be used.

Since the services are already specified, this chapter focuses on their implementation. It starts with a brief recap over the Vert.x toolkit and its concepts which are central to the solution, it then delves into each of the solution components giving a more detailed view over them.

5.1 Vert.x Recap

Vert.x toolkit offers a lot of concurrent and distributed functionalities out of the box. It lets developers to create high-availability clusters, deploy computation units that communicate with each other across the cluster, among others.

It is important to recall some of the concepts early introduced in section 2.13.4. Vert.x runs an event loop that is responsible to handle incoming events. When it comes to Vert.x everything is an event (e.g. a HTTP request, an event bus message, a file read) but some events can block the event loop if not treated in the right way. One of Vert.x rules of thumb is: "Do not block the event loop!"[69].

Vert.x computation units are called verticles. They are an actor-like deployment

scalable and concurrent model that are able to receive and send messages to each other. Those messages are sent through the Vert.x event bus, the nervous system of Vert.x. It allows different parts of the application to communicate with each other and because Vert.x is polyglot it is language independent. This messaging mechanism supports various types of message exchanging patterns: publish/subscribe, point to point and request-response. Since everything in Vert.x is an event, every time one listens to a topic or an address, a handler is registered in the event bus and is then executed when triggered. If the verticle that registered such handlers undeploys (i.e. cease to exist) all of them are automatically unregistered from the event bus.

One Vert.x instance runs in one JVM instance however one Vert.x instance is able to run multiple Vert.x verticles. It is possible to run various JVM instances in the same host.

For a more detailed explanation please refer to section 2.13.4.

5.2 Underlying Components

In chapter 3 three services were identified, one that enables the creation of UbiHSOM nodes, one that concerns the channeling between the nodes and the pulling of data from a collection of data sets to serve as input to the UbiHSOM nodes and finally, the the UbiHSOM service that combines the other two. All of the services were implemented in JAVA using the Vert.x tool-kit because of the reasons mentioned in section 2.13.5. Figure 5.1 illustrates the overall component diagram of the solution. The Vert.x component provides the others with various important features, namely:

- **Vert.x Deployment System** – is responsible for deploying/undeploying Vert.x verticles in the currently running JVM Vert.x instance.
- **Vert.x Event Bus** – serves as the message bus for the entire application. It delivers messages between verticles across the cluster.
- **Hazelcast Cluster Manager** – manages the various Vert.x instances and associated verticles running on the cluster (or locally).

We can see that the UbiFactory and the DataStreamers service components depend only on the Vert.x component. They both offer a Remote Procedure Call API (i.e. an interface that enables developers to remotely call objects methods as they were in a local environment), that other components can use. External components must interact with them using their REST APIs specified in sections 4.2,4.3 and 4.4. It is possible for external components to use the RPC API but this requires that they become bounded to Vert.x.

The UbiHSOM service component depends on both UbiFactory and DataStreamers service components. It communicates with them through their RPC API to achieve its purpose. It uses the UbiFactory component to dynamically create and deploy UbiHSOM nodes and connects them using the DataStreamers service component.

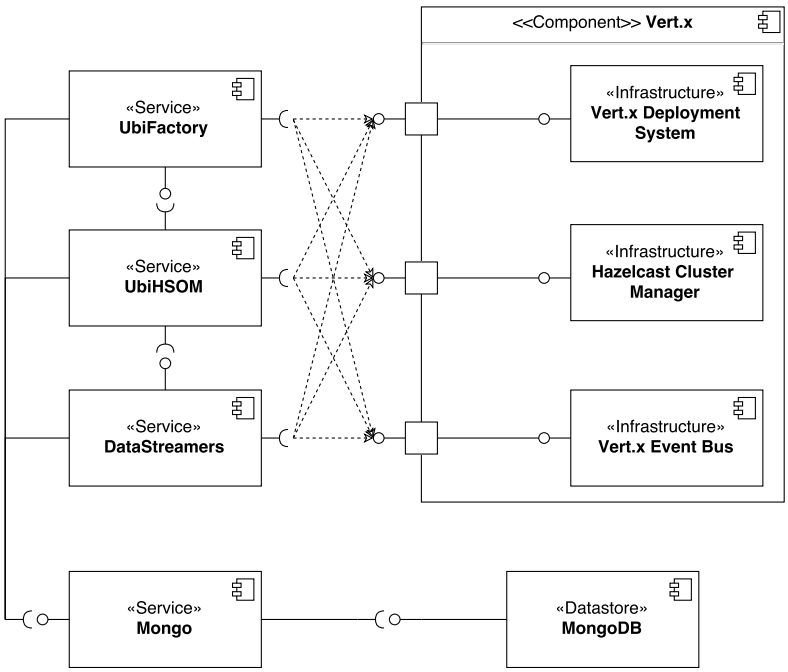


Figure 5.1: Component Diagram

UbiFactory, DataStreamers and UbiHSOM components communicate with the Mongo Service component – which communicates with the MongoDB Datastore – to persist their data. Even though all of these components rely on the Mongo Service to store their data. Each of them can be configured to use its own Mongo Service/MongoDB Datastore as long as this exposes the correct API [70].

This sub-section only gives an overall view over the components. Either UbiFactory, DataStreamers and UbiHSOM service components are better detailed in the following sections.

5.3 DataStreamers

Section 3.3 explained the origin and the need for this component as a fundamental part of the whole system, that is, to create channels to transmit data and create streams of data that pull data from data sets. This section goes deeper on the internals of the component so the reader can have a more technical view about how the system behind the offered API (see section 4.2) was implemented.

Figure 5.2 depicts the component diagram of the implementation. It is composed by four services (two internal, two external), namely, *DataStreamerFrontend*, *DataStreamer-RPC*, *Mongo* and *Vert.x*, respectively, as well as two internal components: *DataStreamer-Factory* and *DBPullerFactory*. Let's now describe each service/component in more detail:

- **Mongo** – as illustrated in the general component diagram on Figure 5.1, Mongo service communicates directly with the MongoDB DBSM. It is used to persist data,

where in this specific case the persisted data regards the data streamers that are created through *DataStreamerRPC* service.

- **Vert.x** – this service was early presented with more detail in 5.2. It is used by the *DataStreamerRPC* service to deploy DataStreamers and interact with them.
- **DataStreamerFactory** – since most of the interactions between external components and the DataStreamers API is done using JSON objects, this component converts the received JSON object to JAVA objects by following the abstract factory pattern [53] to ease object manipulation.
- **DBPullerFactory** – as pointed in section 4.2 of the chapter 4, the DataStreamers service offers the possibility to create streamers that pulls data from a set of internally stored data sets, hence simulating a real-time stream of data. The data sets can have many forms thus they can be stored in various types of DBSM (e.g. SQLite, MySQL, MongoDB). *DBPullerFactory* aims to abstract the DBSM where the data sets are stored by also following the abstract factory pattern[53]. The objects that are created are later used by the DB DataStreamers verticles to simulate the real-time stream of data.
- **DataStreamerRPC** – this component is the core of the DataStreamers system. It implements the whole functionality specified in section 4.2 to manipulate data streamers. It exposes a RPC API used by the DataStreamerFrontend service and interacts with the Vert.x and Mongo services to deploy the various types of data streamers and persist the associated data. Before deploying a data streamers verticle into the cluster, it uses the DataStreamerFactory to first generate the correct data streamers representation that is saved on the database. If the data streamers to be deployed is of type DB, it then interacts with the DBPullerFactory (after creating the correct representation) and then the deployment takes place.
- **DataStreamerFrontend** – serves as the interface between the external user and the underlying DataStreamer service, i.e., exposes the REST API that users use to manipulate resources and interacts with the DataStreamerRPC service to perform the desired operations.

Among the many features the DataStreamers service offers listed in section 4.2 of chapter 4, one that is particularly interesting and deserves a closer look, is the create DataStreamer functionality. The sequence diagram that represents this functionality (regardless of the DataStreamer type) is illustrated in Figure 5.3. In this context an User can assume the following forms: human user (e.g. developer) or an application that is consuming the service (e.g. the UbiHSOM service falls in this category, because it consumes both, the DataStreamer and the UbiFactory services). The user first sends a request *createDataStreamer(datastreamerJson)* to the Frontend service, where *datastreamerJson* is a JSON object that represents the desired data streamer (please refer to section 4.2.1).

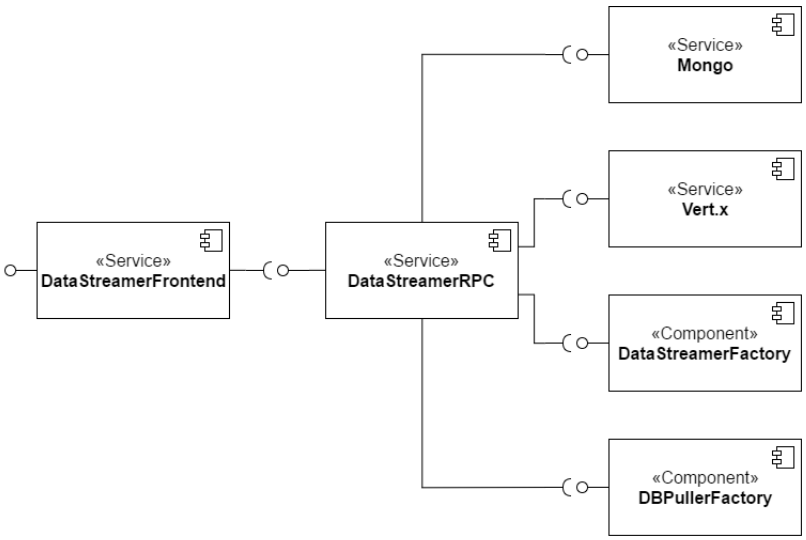


Figure 5.2: DataStreamers – Component Diagram

The Frontend (a short term for DataStreamerFrontend) service is just an API gateway[49] between the user and the system that forwards the message to the RPC service. The RPC service then builds the JAVA object – *dsModel* – based on the JSON object *datastreamerJson* provided by the user. When the JAVA object is generated the RPC service starts two parallel tasks, the database insertion and the verticle deployment. When the representation of the DataStreamer and the verticle deployment are finished if none of them is equal to *null*, the RPC service returns a result to the Frontend service that wraps it in a HTTP response and sends it to the user. However, if any of the *documentId* or *deploymentId* is equal to *null*, the previously inserted representation of the DataStreamer in the database is removed.

We have seen in section 3.2.1 that for a model to be considered a HSOM variant, the outputs of lower tier nodes must be concatenated before being served as inputs of a higher tier node. In section 4.2.1.3 the Zip data streamer was presented as the solution regarding that issue. Let us now formally describe the algorithm behind the Zip data streamer, as shown in the listing bellow (5.1).

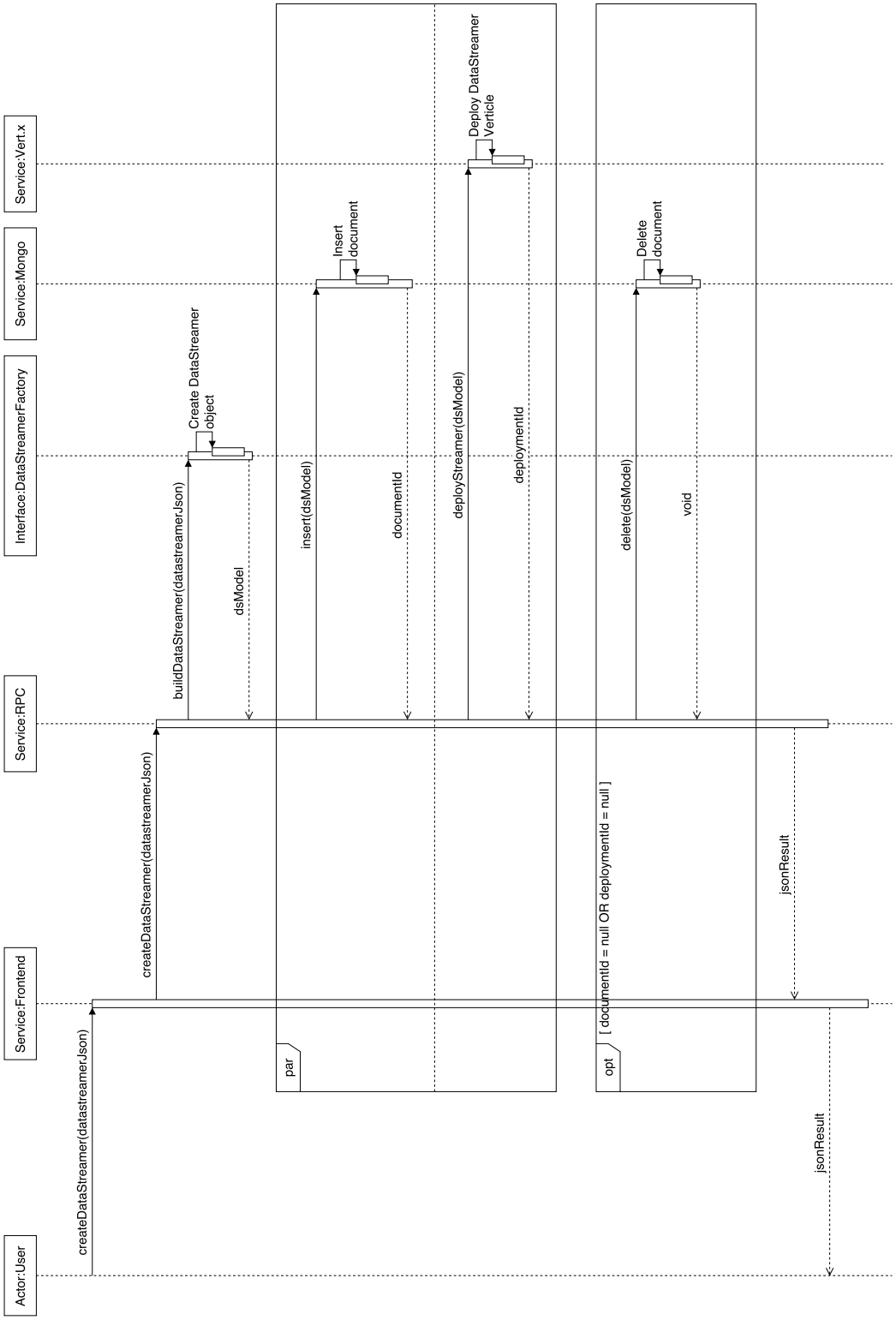


Figure 5.3: DataStreamers – Create DataStreamer Sequence Diagram

Listing 5.1: Zip data streamer algorithm

```
1  Let
2  IN be the input channel
3  OUT be the output channel
4  O be the list of the inputs for the concatenation order
5  I be an input received
6  I.id be the identifier of the input sender
7  I.data be the message payload
8  T be the time frame that the inputs are considered
9  C be the concatenation output
10 M be a map of type I.id, I.data
11
12 Event 1:
13 For each received I in IN
14   if O contains I.id AND M does not contains I.id
15     Put I on M
16
17 Event 2:
18 For every T
19   For each I in M
20     Add I.data to C
21   Send C to OUT
22   Reset M
```

This data streamer has two asynchronous functions, a clock C, which is triggered every T time units (e.g. milliseconds) and a message handler that handles every incoming message that arrives at IN. For each message that arrives it checks if the source identifier (I.id) belongs to the ones defined in the order list, O, it also checks if the message origin is new (by checking if M already contains a message from I.id) and if it is, it stores the received message in M. M stores every object by insertion order. When C is triggered it checks every messages that have arrived and are stored in M and if it messages the number of messages that were defined in O (i.e., it compares if M size and O size are equal) it adds every I in M to the concatenation output, sends to the OUT channel and clears M.

The associated UML class diagrams are available in the appendices, please refer to the appendix section B.1.

5.4 UbiFactory

Previously the UbiFactory API (see section 4.3) was described along with the models needed to interact with it. This section focuses on explaining with more detail the underlying system that supports all the API functionality. It starts with the system component diagram and then shows an example of an operation, namely create UbiSOM operation, with a sequence diagram. For a more in depth view the respective UML class diagrams are shown in the appendix section B.2.

The UbiFactory service purpose is to create and run UbiHSOM nodes, its component diagram that is illustrated in Figure 5.4 is composed by the following services/components:

- **Mongo** – used to persist the data that represents each UbiHSOM node that is created in the system.
- **Vert.x** – it is responsible for deploying the UbiHSOM node verticles.
- **NormalizationFilterFactory** – in section 4.3.1 the normalization types were listed and described. This component is responsible for building the normalization filter that normalize the data before it is fed to the UbiHSOM node. It follows the abstract factory pattern [53].
- **UbiSOM** – this component regards the UbiSOM library developed by Bruno Silva[58] that was formerly explained in section 2.5.
- **UbiFactoryRPC** – this is the orchestrator of the system, the resources described early in section 4.3 are implemented by this service. Whenever an operation takes place, it coordinates the operation with the other services in order to fulfill it. For instance, if one needs to create a new UbiHSOM node, it first inserts some data on

the database and then deploys a verticle that runs an extended version of UbiSOM, an UbiHSOM node (more on this at the end of this section).

- **UbiFactoryFrontend** – this serves as the interface between the users and this system. It exposes the REST API described in section 4.3.

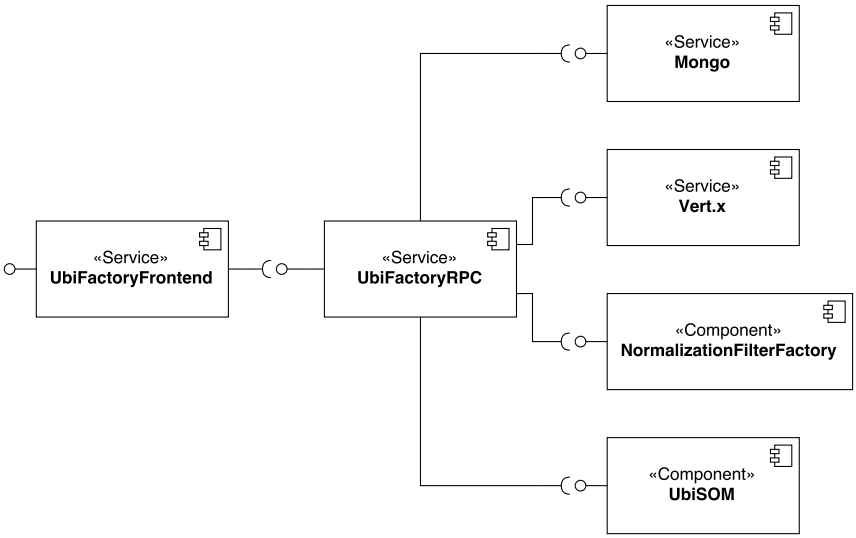


Figure 5.4: UbiFactory - Component Diagram

Figure 5.5 shows the sequential diagram of the operation that creates an UbiHSOM node. The actor user here can be either a client application (e.g. the JavaScript client that is lately use in section 6.1), or a human user. The user starts by requesting the service to create a new UbiHSOM node with a JSON model, that matches the model defined in 4.3.2, to the Frontend service. The Frontend service is nothing but an interface that lies between the user and the RPC service, to which it forwards the requests to. When the request arrives on the RPC service it extracts the embedded JSON object (please see section 4.3.1) that represents the normalization that the user wishes to use. The NormalizationFilterFactory service follows the abstract factory pattern[53] and generates the respective JAVA object based on the received JSON model. Then the RPC service converts the rest of the JSON model to a JAVA object and tries to insert the object in the database as well as deploy the respective verticle. If one of the operations fails it rollbacks the changes. Lastly the service returns a JSON object with the operation result back to the user.

5.5 UbiHSOM

The previous two sections shown the internals of both DataStreamers and UbiFactory services, respectively. It started with the component diagram of each system and then a sequence diagram illustrating an operation was given in order to represent the interaction between the underlying system components when the operation takes place.

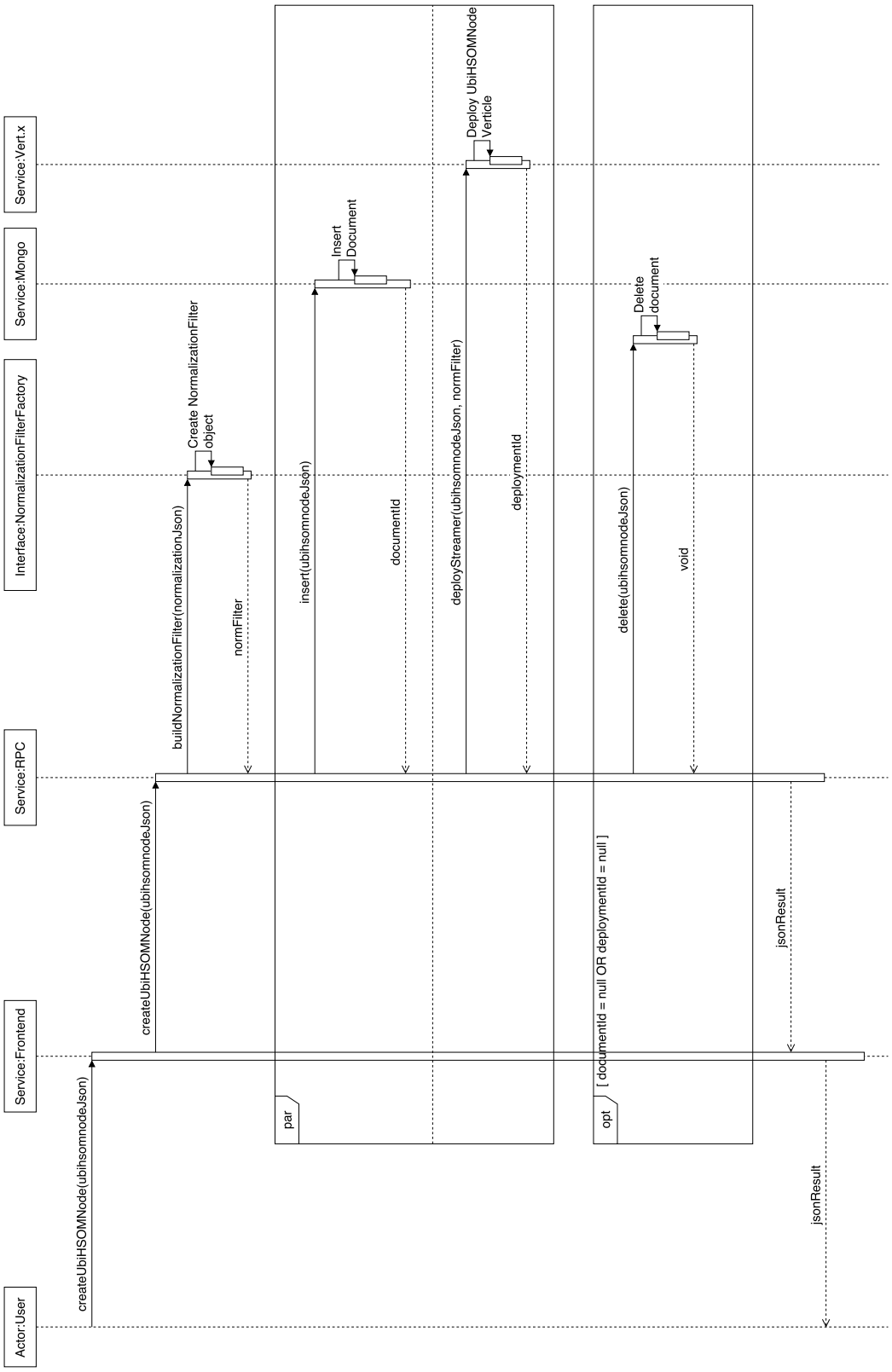


Figure 5.5: UbiFactory - Sequence Diagram for the *create UbiHSOM node* operation

This section focuses on describing how the UbiHSOM service works internally. The definition was presented in section 3.3 and the specification can be found in section 4.4. As we recall the UbiHSOM service relies on the DataStreamers and UbiFactory services to create UbiHSOM nodes, DB data streamers (that stream data to nodes), Zip data streamers (that ensure the integrity of the UbiHSOM nodes input) and Proxy data streamers (which connect UbiHSOM nodes). Figure 5.6 depicts the service component diagram that is composed by the following components/services:

- **Mongo** – it is used to store the data of the created UbiHSOMs
- **Vert.x** – in contrast with its usage in the other systems (DataStreamers and UbiFactory), here, this service main purpose regards communications between the different services. It is used by the *UbiHSOMRPC* service to communicate with *DataStreamersRPC* and *UbiFactoryRPC* via the Vert.x event bus.
- **Graph** – this is an external component (which uses the JGraphT library [43]) that offers the functionalities to represent an UbiHSOM (which follows the model presented in section 3.2). This library was used in order to ease the UbiHSOM service development because it is a JAVA library that implements a directed graph data structure that offers features [43] to create a directed graph and interact with it, by adding/removing nodes and connecting nodes using edges, i.e., it is an implementation of the directed graph data structure.
- **UbiHSOMFactory** – since the input arrives in a JSON format, it must be converted to JAVA objects to ease UbiHSOM manipulation by the *UbiHSOMRPC* service. It uses the *Graph* component to correctly represent an UbiHSOM node after the conversion takes place. The abstract factory pattern[53] is applied here.
- **DataStreamersRPC** – this service was discussed in detail in section 4.2 of this chapter. Its origin and goal are described in section 3.3 of chapter 3.
- **UbiFactoryRPC** – the previous section detailed this service and it is define in section 4.3. The *UbiHSOMRPC* service uses it to creates the models that embody the graph nodes.

Figure 5.7 and 5.8 illustrate the create UbiHSOM node and delete UbiHSOM node operations, respectively. Let us first describe the *create UbiHSOM node* operation. When creating a new UbiHSOM node the user must give an input that follows the model defined in 4.4.1, if the input is correct, the *ubiHSOMRPC* service communicates with *UbiFactoryRPC* service to launch a new UbiHSOM node instance (refer to section 5.4), that if succeeded triggers the creation of a node in the *Graph* component and returns the result to the user.

In contrast with the previous operation that was illustrated through a sequence diagram, the *delete UbiHSOM node* operation is a bit more complex, hence having a sequence

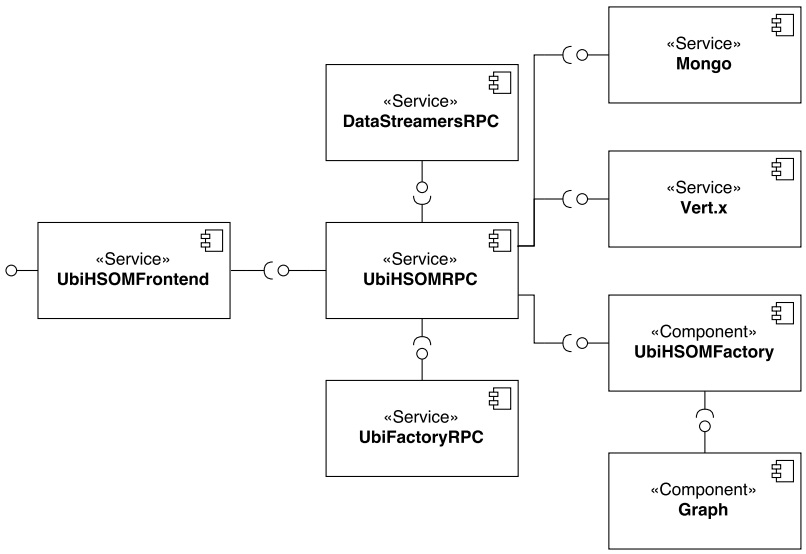


Figure 5.6: UbiHSOM - Component Diagram

diagram describing it would be confusing. Therefore an activity diagram is used instead because it offers more abstraction thus, one can understand how the operation works regardless of the fine details that only add complexity to the explanation. As it is depicted in Figure 5.8, whenever a node is to be remove by this method all the dependent nodes and their links are also remove. This means that for each node that must be removed the associated UbiHSOM node must be undeployed and deleted as well. As for each link that must be removed, the associated DataStreamer needs to be undeployed and deleted. Therefore first a list of the dependent nodes and links is calculated, this list is then iterated and for each node a request to delete the associated UbiHSOM node is sent to the *UbiFactoryRPC* service, a similar process is done for the links and the respective DataStreamers instances used by the node. When all nodes and links are successfully undeployed and deleted, the graph nodes and edges are removed therefore, converging to a valid state.

This service is originally implemented in *JAVA*, even though both RPC and frontend services offer the same API's, it uses the internal RPC's services rather than their respective frontend services. Thereby the system could be completely replicated using a different programming language and use the other services REST APIs to communicate with them. This is possible to achieve due to the system flexibility as we will later see in chapter 6 that presents various case studies using the frontend services. Also note that the *UbiHSOMFactory* component requires the *Graph* component interface. Therefore a different representation that meets the interface specification can be used in order to represent UbiHSOM.

The associated UML class diagrams are available in the appendices, please refer to

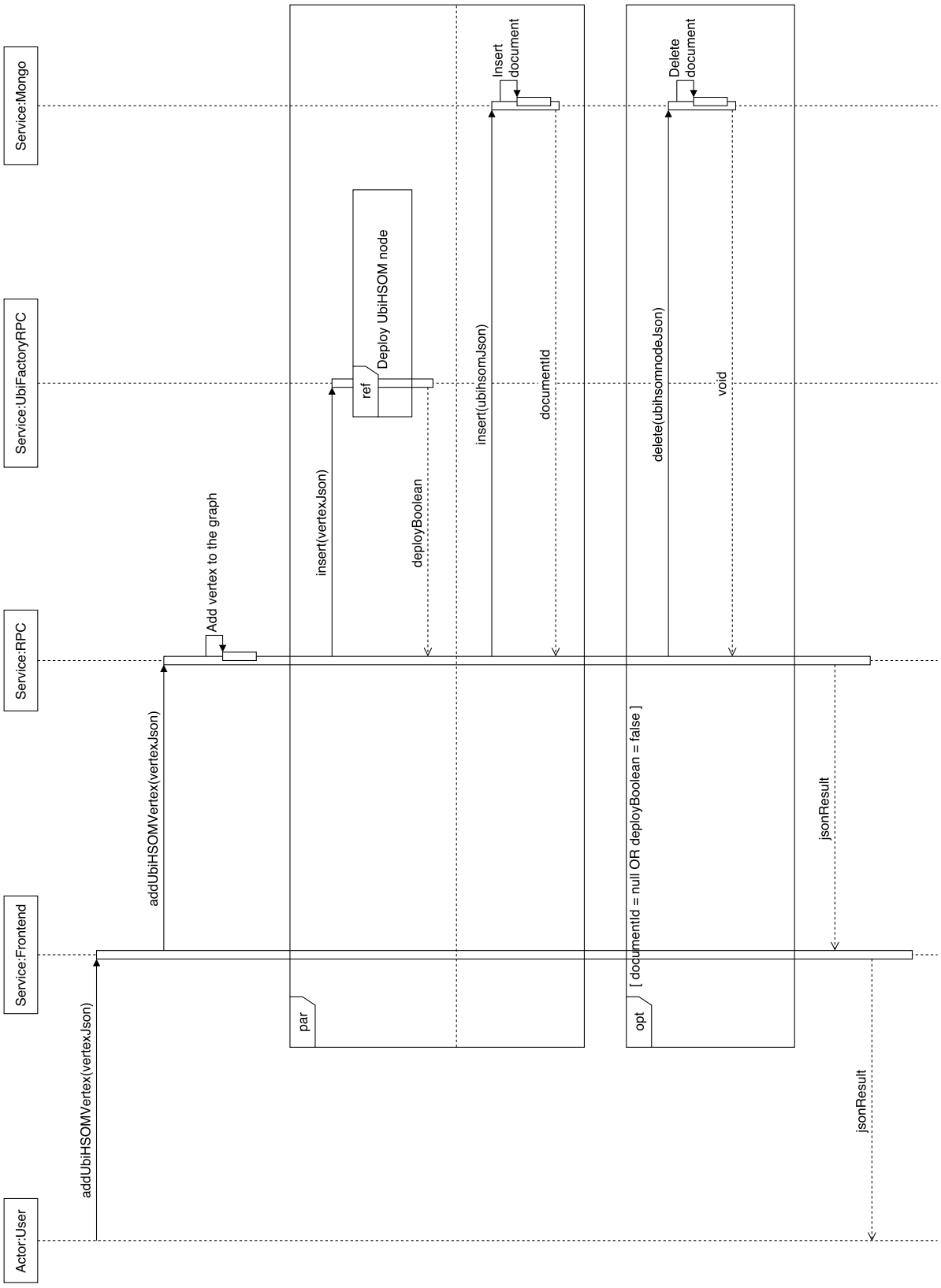


Figure 5.7: UbiHSOM - Create UbiHSOM Sequence Diagram

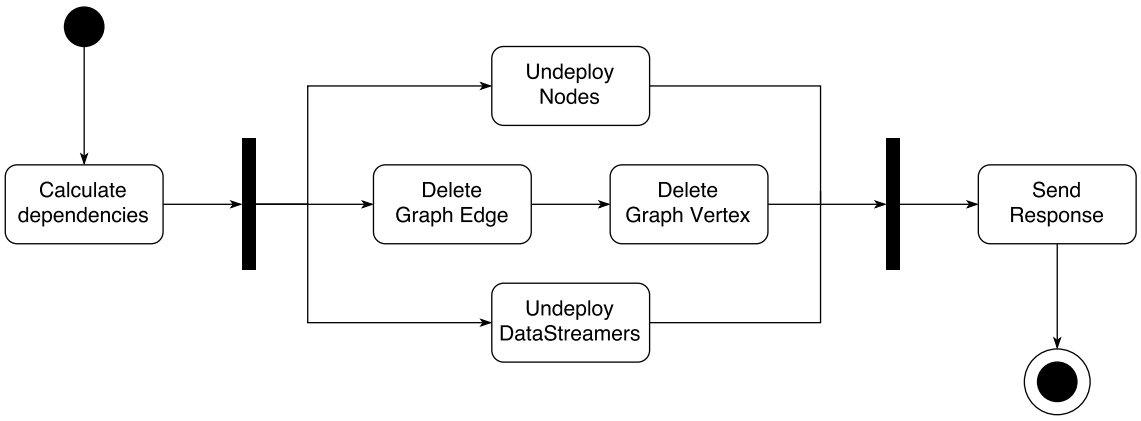


Figure 5.8: UbiHSOM - Delete UbiHSOM Activity Diagram

the appendix section B.3.

5.6 Source Code

This section aims to present the various repositories that host the source code regarding the usage examples scrips given in chapter 4 and the services previously described and specified (in chapter 4 and chapter 5). All of the the source-code is hosted on *GitHub*[15]. *GitHub* is based on *Git*, which is an open source version control system that manages and stores revisions of project. While *Git* is a command line tool, *GitHub* is a *Git* repository hosting service that provides a web-based graphical interface, access control and several collaboration features.

The various programs/scripts were developed using various programming languages, namely, JavaScript, Python and JAVA. There are four repositories:

- ubifactory-javascript-example – this repository hosts a web-client written in HTML, CSS and JavaScript, which interacts with the UbiFactory service in order to create, delete a visualize UbiHSOM nodes;

- ubihsom-javascript-example – similar to the previous repository but this one hosts a web-client that interacts with the UbiHSOM and the DataStreamers services instead of the UbiFactory service;

- rest-client-api – this repository hosts a HTTP REST client written in JAVA. It is another example of interaction between a client (this time written in JAVA) and the UbiFactory service;

- python-ubihsom-scripts – this repository hosts a few scripts written in Python on how to use the UbiHSOM and DataStreamers REST API using Python;

- thesis-ubihsom – this repository hosts all the services that were described as being part of this thesis solution, namely, the DataStreamers service, the UbiFactory service, the UbiHSOM service and the Mongo service. The solution was written in JAVA and it uses the Vert.x tool-kit (as already mentioned). Moreover, it is a complete and functional version of the implementation of the proposed services that was used in chapter 6 experiments (and all the presented results were extracted by using this package), along with the scripts of the repositories above mentioned.

Each of the repositories mentioned above have documentation concerning the solutions they host.

Chapter 6

Case Studies

Having identified (chapter 3), specified (chapter 4) and explained (chapter 5) the solution services it is now possible to demonstrate how these services can be used. This chapter aims to present various case studies where the services are used in various scenarios to achieve different goals. It first starts with a case study where a web application uses the thesis services that were hosted in the AWS public cloud, in order to analyze geometric shapes data by using an UbiHSOM node. The last case study presents various data sets that were analyzed using the UbiHSOM service. The chapter ends with a discussion about the various case studies.

6.1 SOM

As already mentioned, the chosen SOM variant is the UbiSOM. This section focuses on a very particular use case where a web client uses the *UbiFactory* REST API (refer to section 4.3), in order to take advantage of the UbiHSOM node capabilities to process various data sets, namely the geometry shapes. This case study uses a web interface to interact with the user and the UbiFactory service, through which, one can create/delete UbiHSOM nodes. The web interface was written in *HTML*, *CSS* and *JavaScript* and its source code is publicly available[14]. Since the web browsers are becoming more powerful, smarter and ubiquitous the web page makes a good example because it is widely used either on desktop, smartphones or tablets.

This case study aims to depict the use case where a user needs UbiHSOM nodes data processing and visualization features for a collection of geometry shapes.

The geometry shapes data set has many shapes (e.g. square, right triangle, isosceles triangle, diamond, pentagon and hexagon, see the results in appendix C) thus given the UbiSOM visualization capabilities it is possible to see various clusters according to the

shape vertices cardinality as shown in section 6.1.1. It is also possible to observe the shape shifting whenever an UbiHSOM node is fed with different shapes data set because UbiSOM deals with real-time data, this matter is discussed in section 6.1.2.

Since the service is running on AWS, section 6.1.3 focuses on explaining how can this type of solution benefit from the public cloud advantages.

The web page interface (Figure 6.1) has four main functions: create, feed, refresh and delete. It is a very simple interface where one can create an UbiHSOM node instance, feed data to it, delete it and view the results in the form of component planes heatmaps, hit-histogram and U-matrix.

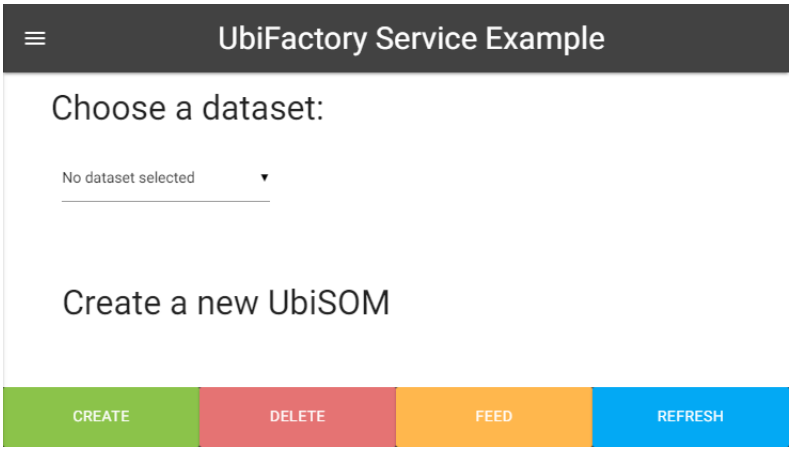


Figure 6.1: Case Study – Web Interface

6.1.1 Square

A square can be easily represented using Cartesian coordinates as shown in Figure 6.2. Four points are identified: $(0,0)$, $(0,1)$, $(1,0)$ and $(1,1)$. These are the points that are fed to the UbiHSOM node.

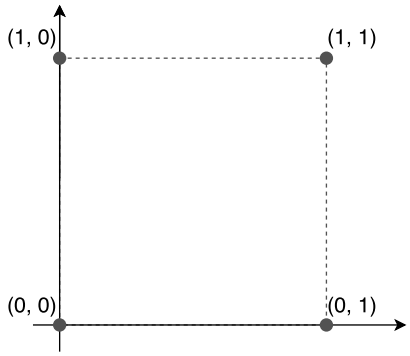


Figure 6.2: Case Study – Cartesian Square

Firstly an UbiHSOM node instance is created, Figure 6.3 presents the resulting visualization of the newly created instance. As expected everything is random since the

UbiHSOM node is based in the UbiSOM, that when it is initialized for the first time it randomizes all its prototypes weights.

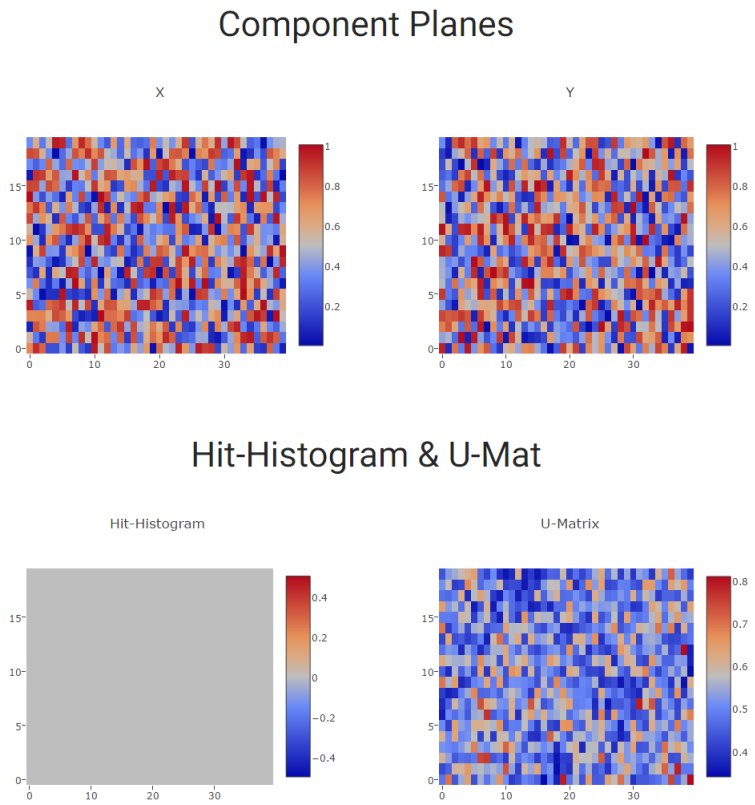


Figure 6.3: Case Study – Initial State

The model is fed with three thousand data points each time the *Feed* button is pressed, and the corresponding component planes, hit-map and U-matrix are returned as a result, like the ones illustrated in Figure 6.4. Considering the data points that were fed correspond to a square – that is represented by four Cartesian points – it is expected that four different clusters are identified. By looking at the U-matrix of the same figure, it is possible to detect four big blue areas, also know as lakes [39]. This means that all the UbiSOM prototypes in those areas, are close to each other (please refer to the scale at the right of the graphic), because the prototypes try to adapt to the input. We can see that the number of lakes correspond correspond to the number of the Cartesian points a square has, which is four. Thereby it is safe to assume that the result is what was early expected, considering that, in the end, four clusters were identified which correspond to the four Cartesian points.

6.1.2 Geometry Shape Shifting

The UbiHSOM nodes work with real-time data, so in this example data regarding a Cartesian representation of a right triangle is firstly fed to an UbiHSOM node, and along

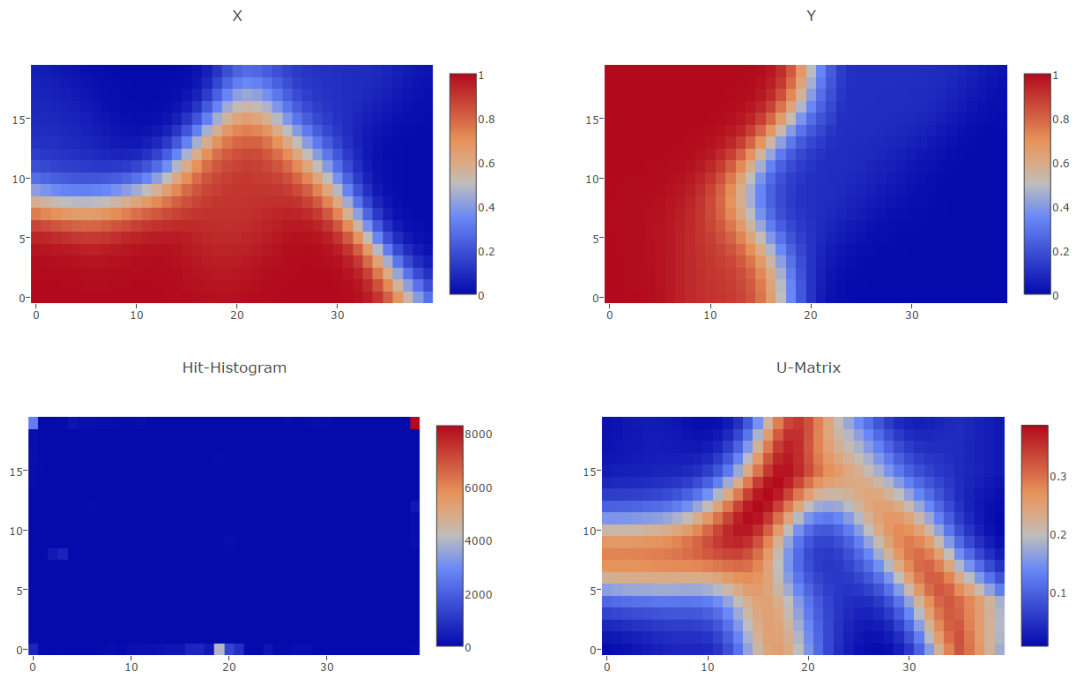


Figure 6.4: Square Case Study – Component Planes, Hit-Histogram and U-Matrix

the way, we feed the model with the square data set and we will see that the shape switches from a triangle to a square. Figure 6.5 shows the two U-matrix, at the left side there is the U-matrix before the square shape and at the right side the U-matrix after a few examples of the square shape were fed. It is interesting to see that a new cluster was starting to form, thereby indicating the model adaptation to the new data. This reinforces the idea and the goal concerning real-time data adaptation.

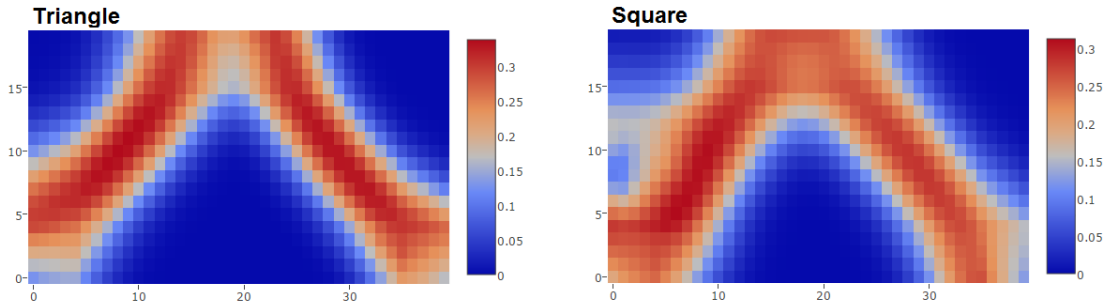


Figure 6.5: Shape Shifting Case Study – U-Matrices

6.1.3 Amazon Web Services

For this case study a very specific test environment was set using AWS. In order to meet the test requirements an AWS EC2 t2.micro instance was launched and MongoDB and JAVA were installed in it. After that, the UbiFactory service was correctly configured in order to communicate effectively with Mongo, and a really simple ad hoc web server, that was developed for serving the previously referred web-page assets, was installed.

The service was up for, roughly, two weeks which added to a total of 2.89$ cost. Figure 6.6 shows a pie chart that conveys the information regarding the decomposition of the total amount. According to the invoice, AWS follows the criteria listed bellow, for all of the accounted services:

- **Elastic Compute Cloud** 0.013$ per On Demand Linux t2.micro Instance Hour;
- **Elastic Compute Cloud** – 0.11$ per GB-month of General Purpose SSD (gp2) provisioned storage - EU (Ireland);
- **Data Transfer** – 0.010$ per GB - regional data transfer - in/out/between EC2 AZs or using elastic IPs or ELB;
- **Simple Storage Service** – 0.023$ per GB - first 50 TB / month of storage used

The biggest slice of the amount depicted in Figure 6.6, concerns the *Elastic Compute Cloud* service that aggregates the computation time and provisioned storage criteria. One surprising fact is that the slice associated with data transfer is the smallest, which could be good because this type of application that tries to extract knowledge from a stream of data, is, indeed, a data bounded application, however further tests must be conducted. On the other hand, having the solution following a microservice approach that promotes scalability could have an impact in the price because it would fall under the green slice. The VAT percentage is high, it totals to a fifth of the whole amount, being, in this case, greater than both, *Data Transfer* and *Simple Storage Service*.

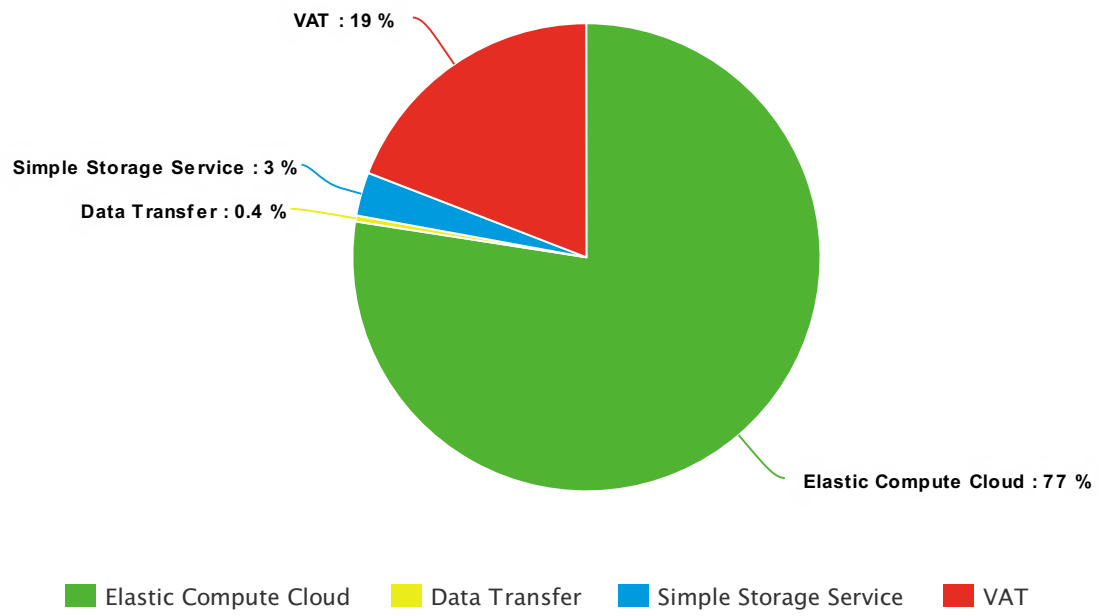


Figure 6.6: AWS Billing Information – Pie Chart

6.2 UbiHSOM

This section presents three case studies where the UbiHSOM service is used to analyze three data sets: cube, iris and financial. Some of the case studies will be performed using a web-interface (illustrated in Figure 6.7), that was developed using HTML, CSS and JavaScript, which is publicly available (see section 5.6). The interface allows the user to create UbiHSOMs, add and delete UbiHSOM nodes as well as connect them, by creating an edge between them. Moreover it lets the user create DB data streamers that are represented on the interface as being Data Sources, which the user can also connect to a node. The interface interacts with the UbiHSOM REST API to support the given interface functionalities. To visualize the nodes representations (i.e. component planes, hit-histograms and u-matrix), the web interface presented in the previous section will be used.

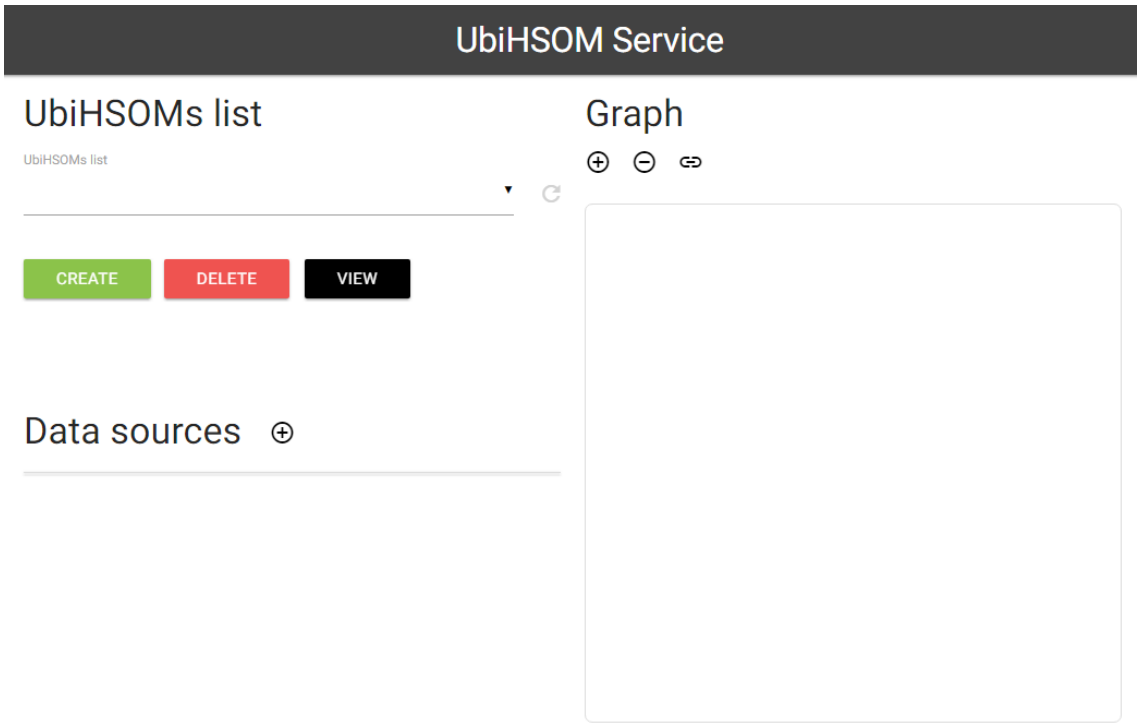


Figure 6.7: UbiHSOM - Web Interface

All of the UbiHSOM nodes used in the case studies below, have the same parameters:

- **name** – varies accordingly to the case study
- **weight-labels** – varies accordingly to the case study
- **width** – 40
- **height** – 20
- **dim** – varies as we will later see

- **eta-i** – 0.1
- **eta-f** – 0.08
- **sigma-i** – 0.6
- **sigma-f** – 0.2
- **beta-value** – 0.7
- **normalization** – none

The parameters were identified [58] as being the ones that shown better results when using UbiSOM.

6.2.1 Cube

This subsection regards a case study very similar to the one presented in subsection 6.1.2, where an UbiHSOM node is fed with data of a square. Instead of a square, here, we will feed an UbiHSOM with data of a cube represented in a three dimensional Cartesian coordinate plane. To represent a cube in a three dimensional Cartesian coordinate plane, eight Cartesian coordinates are needed: (0,0,0); (0,1,0); (1,0,0); (1,1,0); (0,0,1); (0,1,1); (1,0,1); (1,1,1). The data set that will be used has 80 000 data points, which includes the ones previously mentioned and some noise, i.e., it also has other points close to those mentioned above.

In this case study we are trying to validate if the UbiHSOM is able to identify the clusters represented by the cube vertices when the data set is split in (x,y) and (y,z) coordinates, that are then fed to two nodes and merged in a third one. Let us describe how the UbiHSOM structure should look like before using the web-interface, hence, the notation presented in section 3.2 will be used. The goal of this case study is to have, two first tiers nodes, one that receives the coordinates x,y of a point and another that receives the coordinates y,z, that are then linked to a second tier node. Let XY be a data source that sends x,y points. Let YZ be a data source that sends y,z points. Let A,B and C be nodes. The resulting UbiHSOM structure is defined as:

$$XY \Rightarrow A, YZ \Rightarrow B, A \Rightarrow C, B \Rightarrow C \tag{6.1}$$

Which when mapped to a graph, looks as illustrated in Figure 6.8.

There are a few details that must be referred:

- Node A and B have a dimensionality of two because one receives x,y and the other y,z

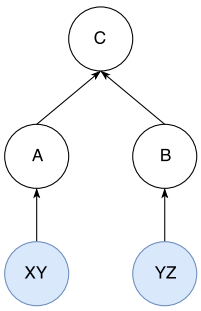


Figure 6.8: Case Study – Cube, graph

- Node C has a dimensionality of four because it receives the outputs of A and B (which are the BMU coordinates of the most recent learned observation) that, together, summing up for a total of four (dimensionality). Thus the input of C should look like: Ax, Ay, Bx, By. Where Ax, Ay are the BMU coordinates of the node A after learning from an observation, and Bx, By are the BMU coordinates of the node B after learning from an observation.

Now that the case study is formally defined, let us replicate it using the web interface. We first start by creating a new UbiHSOM, then we create two data sources XY and YZ, followed by three nodes (A, B and C) and lastly we link everything (as described above). The resulting appearance of the interface after following these steps is illustrated in Figure 6.9.

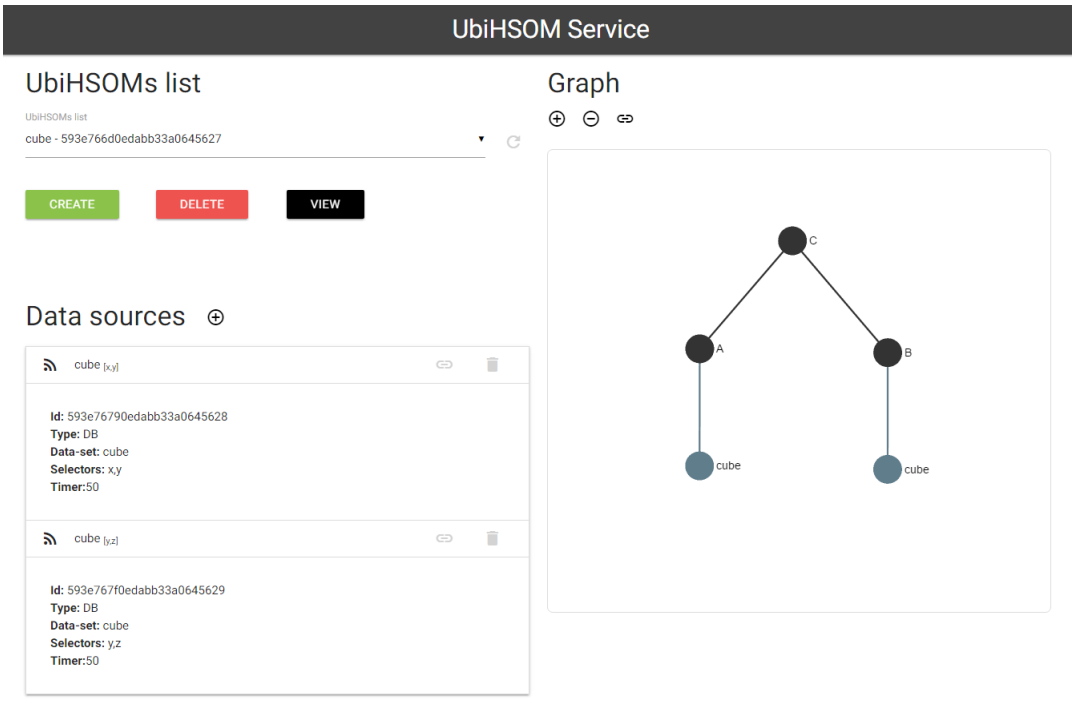
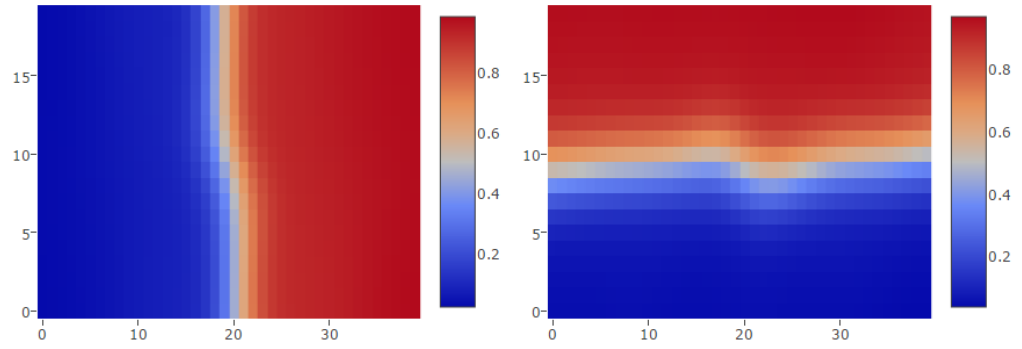


Figure 6.9: Case Study – Cube, web interface

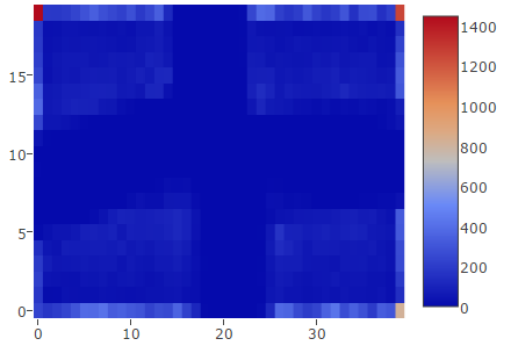
The newly created UbiHSOM structure has now three node and each of them can be analyzed through their component planes, hit-histogram and U-matrix. A detailed and deep analysis of those graphics are out of the scope of this thesis, however, we will briefly analyze it in order to validate if the proposed HSOM variant, the UbiHSOM, is able to identify the clusters from the underlying data when merging two nodes, that, in this case, should be the clusters represented by the cube vertices.

Figure 6.10 represents the component planes, hit-histogram and U-matrix of the node A (which receives the coordinates x,y). In this case the analysis is very similar to the one done on section 6.1.1, as we can see from the U-matrix (that highly resembles the one from the SOM Square case study). Since the x,y coordinates represent a face of the cube (which is a square), we can see four clusters that map to the square vertices. Figure 6.11 represents the node B (that receives the coordinates y,z) and the analysis applied before can be applied here as well, as shown by the U-matrix.

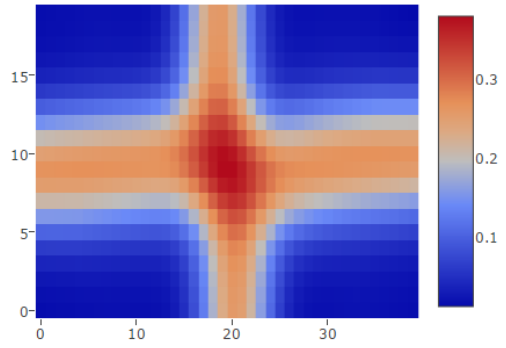


(a) Component Plane of feature X

(b) Component Plane of feature Y



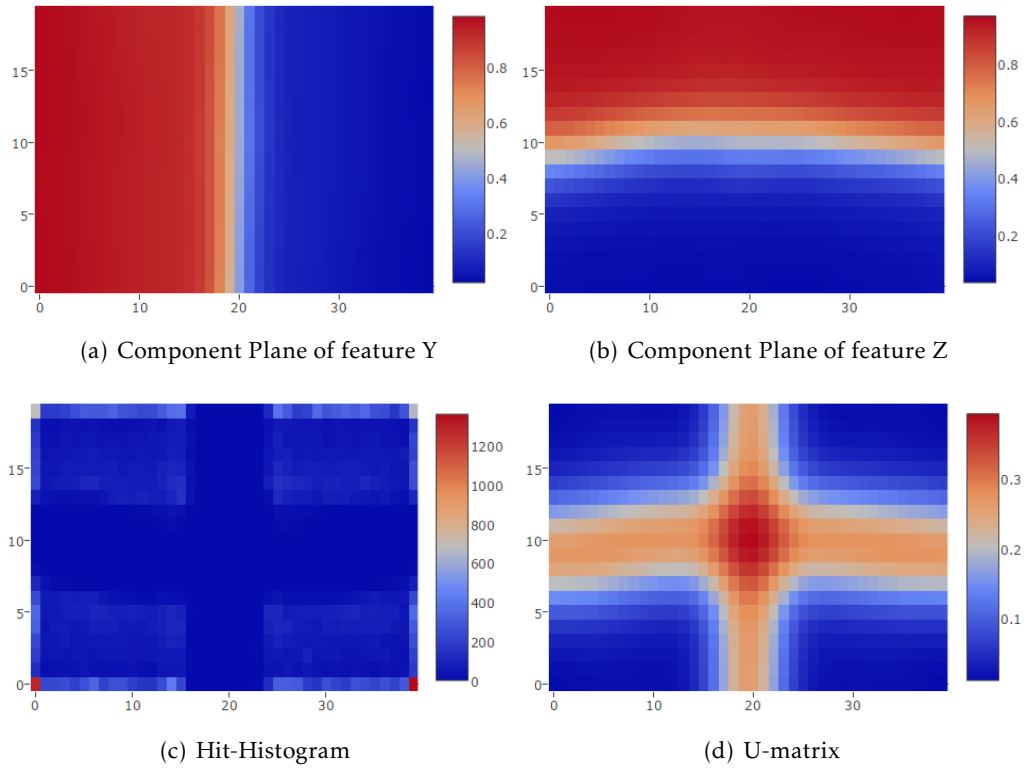
(c) Hit-Histogram



(d) U-matrix

Figure 6.10: Case study – Cube, graphical representation of node A

We will now analyze node C (Figure 6.12) but before, we must recall the input of node C. It receives the outputs from both, A and B, which are the BMU coordinates of the respective maps of the nodes (whenever they learn from a data sample) that when merged become: [Ax, Ay, By, Bz], hence here we are not analyzing where the points were distributed along the map, instead, we are analyzing how they were spatially mapped



(a) Component Plane of feature Y

(b) Component Plane of feature Z

(c) Hit-Histogram

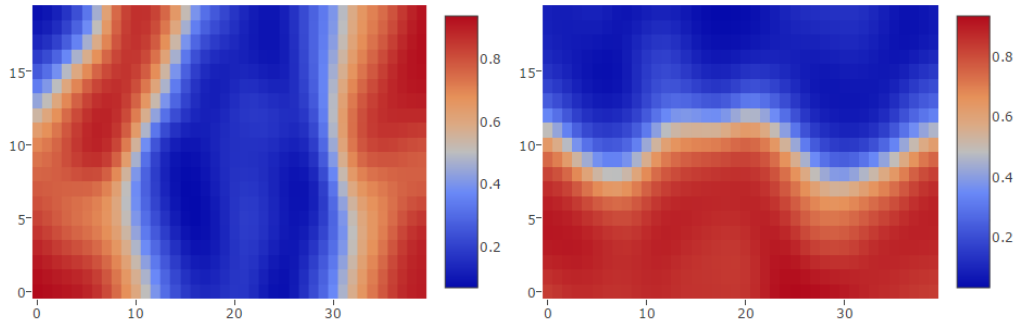
(d) U-matrix

Figure 6.11: Case study – Cube, graphical representation of node B

from node A and B to node C. To accomplish that mapping let us look at node C graphical representations. We first need to identify a cluster in the U-matrix with the help of the hit-histogram, which can be done by looking for a prototype with a high hit count and identify the corresponding cluster on the U-matrix. Once identified the cluster we should now verify which of the prototypes of the cluster has the lowest value in the U-Matrix (i.e. the lowest distance from their neighbours) and then check the value of the chosen prototype (let P be this prototype), in each component plane. Node C component planes are Ax, Ay, Bx, By which means that they are the resulting computation of the concatenation of the BMU coordinates that were outputted from A and B, so we need to map P values to the component planes of node A and B and verify if node C result is correct. We will proportionally convert the values of P[Ax, Ay, Bx, By] to map coordinates. After repeating the process for various clusters in node C, we should now verify if the values of Ay and By match as we can see on table 6.1.

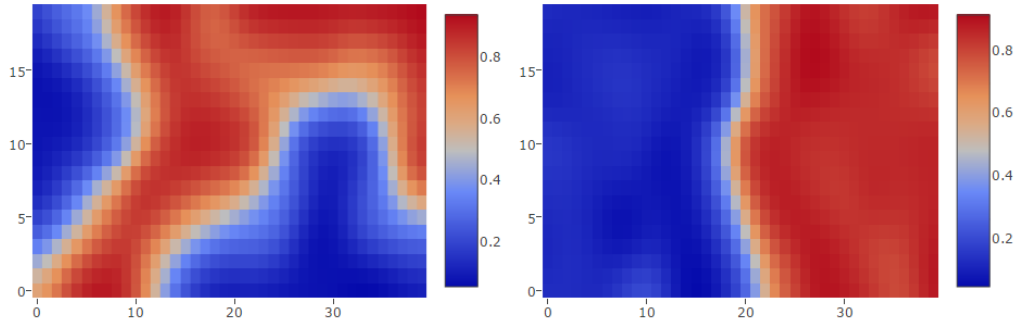
Table 6.1 analyzed twelve clusters (the ones on Figure 6.13) where the previous explained process was applied and as a result we can see that the highlighted prototypes: P40,0; P40,20; P28,0; P26,20; P17,20; P11,20; P17,0 and P0,6 have fulfilled the requirement of A - y and B - y being equivalent. Moreover, we can see that all of the cube vertices can be identified from the Ax, Ay, By, Bz features:

- **P40,0** – 1,1,1



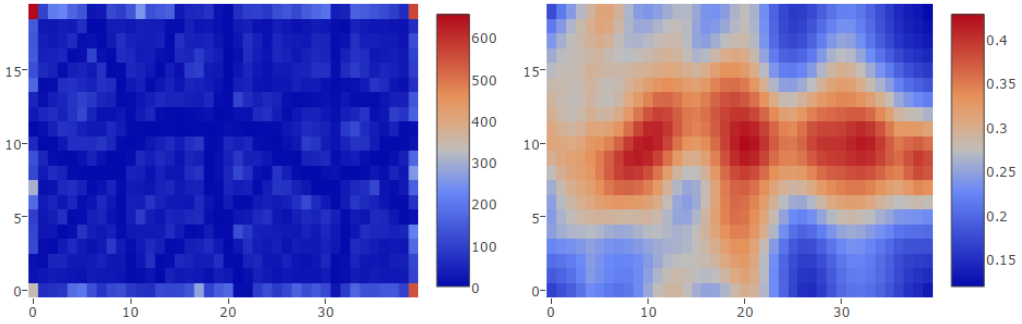
(a) Component Plane of feature A x

(b) Component Plane of feature A y



(c) Component Plane of feature B x

(d) Component Plane of feature B y



(e) Hit-Histogram

(f) U-matrix

Figure 6.12: Case study – Cube, graphical representation of node C

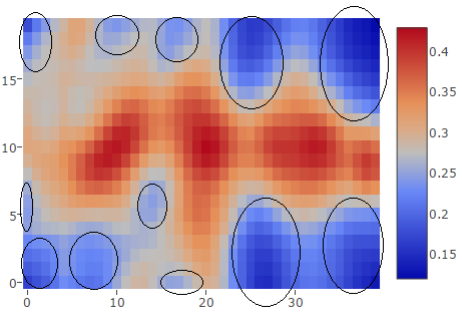


Figure 6.13: Case Study – Cube, Node C U-matrix clusters

Prototypes Coordinates (i, j)	Node C								Node A		Node B	
	A - x	A - y	B - x	B - y	A - i	A - j	B - i	B - j	A - x	A - y	B - y	B - z
0,0	0.93	0.83	0.6	0.12	37.2	16.6	24	2.4	0.97	0.93	0.12	0.04
0,20	0.08	0.08	0.18	0.11	3.2	1.6	7.2	2.2	0.02	0.04	0.94	0.05
40,0	0.9	0.83	0.1	0.88	36	16.6	4	17.6	0.97	0.93	0.96	0.95
40,20	0.91	0.07	0.93	0.88	36.4	1.4	37.2	17.6	0.96	0.04	0.03	0.95
28,0	0.16	0.93	0.07	0.88	6.4	18.6	2.8	17.6	0.05	0.95	0.97	0.95
26,20	0.11	0.1	0.9	0.86	4.4	2	36	17.2	0.03	0.05	0.04	0.95
8,0	0.8	0.85	0.87	0.14	32	17	34.8	2.8	0.95	0.95	0.04	0.05
17,20	0.4	0.03	0.8	0.12	16	0.6	32	2.4	0.09	0.04	0.05	0.05
11,20	0.86	0.07	0.81	0.09	34.4	1.4	32.4	1.8	0.95	0.04	0.05	0.04
15,7	0.09	0.87	0.78	0.06	3.6	17.4	31.2	1.2	0.03	0.94	0.05	0.04
17,0	0.15	0.84	0.17	0.04	6	16.8	6.8	0.8	0.04	0.93	0.95	0.04
0,6	0.82	0.9	0.18	0.11	32.8	18	7.2	2.2	0.95	0.95	0.95	0.04

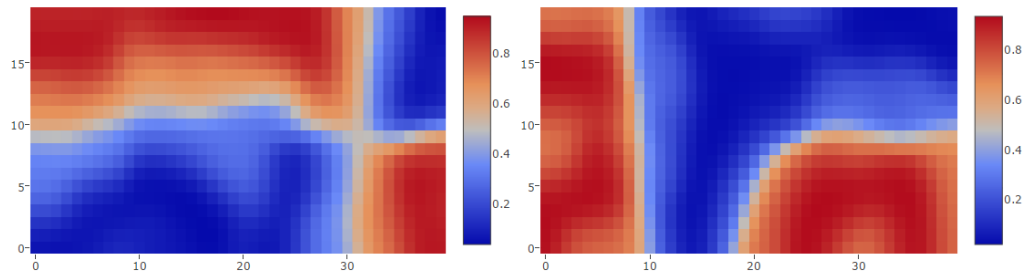
Table 6.1: Case study - Cube data set, coordinates mapping

- **P40,20** – 1,0,1
- **P28,0** – 0,1,1
- **P26,20** – 0,0,1
- **P17,20** – 0,0,0
- **P11,20** – 1,0,0
- **P17,0** – 0,1,0
- **P0,6** – 1,1,0

The other vertices (P0,0; P0,20; P8,0; P15,7) have not satisfied the requirement that Ay and By must be equivalent, one of the reasons being a poor behaviour of the streams, because they started to deviate from each due to the data sources and zip timers that were set (more on this later), thus, leading to inconsistent results, i.e., a mismatch of the data points. Nonetheless, the UbiHSOM was still able to adapt and identify the cube vertices.

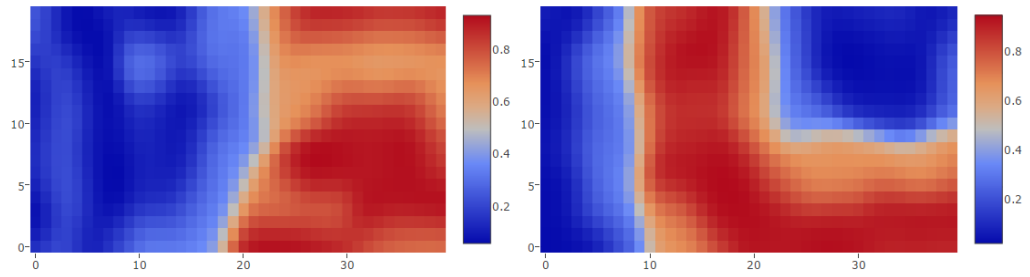
To deal with the previously identified data point mismatching problem the timers values were changed. Node A and B timers went from 70 milliseconds to 25 milliseconds, whereas, node C timer, switched from 90 milliseconds to 50 milliseconds. Figure 6.14 shows the appearance of node C after running an experiment where the streams behaved as expected and did not deviate from one another because the UbiHSOM nodes timers were adjusted. As we can see by analyzing the U-matrix, the clusters are more clear and easier to identify, moreover, they are eight clusters (the same number of vertices of a cube).

All of the following case studies timers were correctly configured to insure the streams did not deviate.



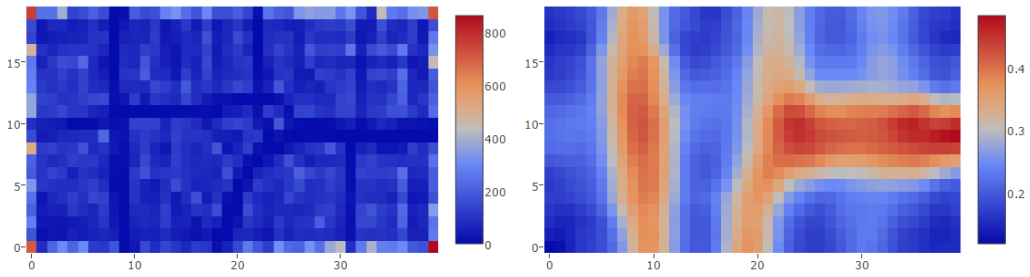
(a) Component Plane of feature A x

(b) Component Plane of feature A y



(c) Component Plane of feature B x

(d) Component Plane of feature B y



(e) Hit-Histogram

(f) U-matrix

Figure 6.14: Case study – Cube, graphical representation of node C configured correctly

6.2.2 Iris

This subsection aim to briefly analyze the Iris flower data set using the proposed UbiHSOM solution. Unlikely the previous section where a web-interface (Figure 6.1) was used to create the UbiHSOM regarding the cube data set, in this case study a Python script was developed and used to create the UbiHSOM structure to study the Iris flower data set. The script is publicly available (refer to section 5.6).

The Iris flower data set was introduced by R. Fisher in 1936 and consist of 50 samples from each of the three species of Iris flowers: Iris Setosa, Iris Versicolor and Iris Virginica. Each sample has four features, all of them measured in cm: Sepal Length, Sepal Width, Petal Length and Petal Width. This data set is a classic data set widely used in the machine learning community in order to test machine learning algorithms. P. Hoey statistically analyzed the data set [30] and concluded that the Iris Virginica is characterized by having a long sepal, long petals and wide petals, whereas the Iris Setosa is characterized by having a short sepal, short petals and very narrow petals and finally the Iris Versicolor is an Iris flower that falls in between the other two.

The analysis of this data set using UbiSOM was already done[58], where the author shown that the UbiSOM could successfully separate the Iris Versicolor and Virginica flowers from the Iris Setosa. Therefore, this subsection will only focus on analyzing it from an UbiHSOM perspective.

This case study aims to study if a higher tier UbiHSOM node C, that receives the outputs of two lower tier nodes A and B, is able to map and identify the various classes in the data set. Let us first formally define the UbiHSOM structure that will be used by following the notation introduced in section 3.2. We will start having two data sources: one that outputs tuples with the format sepal length, sepal width (Sepals); and another that outputs petal length, petal width tuples (Petals). Let A, B and C be nodes. Now we connect Sepals with A and Petals with B and finally A with C and B with C, as shown in 6.2.2. Note that node C will receive a concatenation of the BMU coordinates that result from the learning process of both node A and B, i.e., a tuple with the following format: Sepals x, Sepals y, Petals x, Petals y.

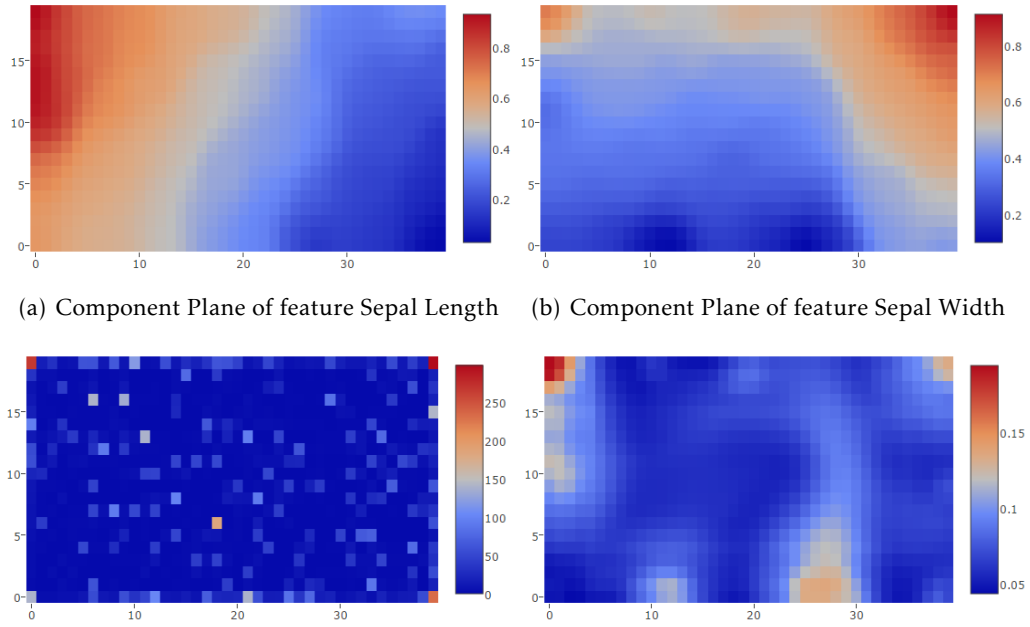
$$Sepals \Rightarrow A, Petals \Rightarrow B, A \Rightarrow C, B \Rightarrow C \tag{6.2}$$

Figure 6.15, Figure 6.16 and Figure 6.17 show the results of node A, B and C, respectively. The analysis method applied here is the same applied in the previous use case (please refer to subsection 6.2.1), where we will try to map the node C coordinates to node A and B values to see if the cluster correspond to any of the Iris flowers class. By analyzing Table 6.2 it is possible to see that the UbiHSOM was able to map the three flowers classes on distinct areas of the map, the clusters regarding the prototypes with the coordinates 40, 0 and 40, 20 follow under the values that resemble the Iris Setosa as

we can see from their low values on sepal length, petal length and petal width features. Whereas the cluster regarding prototype with the coordinates 0, 0 mapped to data points with high sepal length, petal length and petal width, which matches the characteristics of the Iris Virginica flower. Finally we can see that the clusters regarding the prototypes with the coordinates 0, 20 and 0, 27 have balanced features values like the Iris Versicolor flowers.

Prototypes Coordinates (i,j)	Global								A		B	
	A X	A Y	B X	B Y	A i	A j	B i	B j	Sepal Length	Sepal Width	Petal Length	Petal Width
39,0	0.95	0.09	0.91	0.16	37.05	1.71	35.49	3.04	0.06	0.44	0.07	0.04
39,19	0.87	0.88	0.89	0.11	33.93	16.72	34.71	2.09	0.27	0.67	0.08	0.04
0,19	0.42	0.93	0.06	0.92	16.38	17.67	2.34	17.48	0.57	0.5	0.79	0.93
0,0	0.05	0.66	0.04	0.19	1.95	12.54	1.56	3.61	0.89	0.35	0.86	0.71
0,27	0.49	0.14	0.58	0.06	19.11	2.66	22.62	1.14	0.37	0.22	0.53	0.43

Table 6.2: My caption



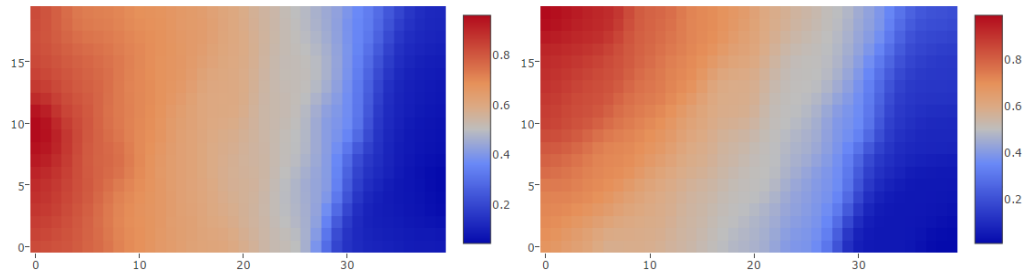
(a) Component Plane of feature Sepal Length

(b) Component Plane of feature Sepal Width

(c) Hit-Histogram

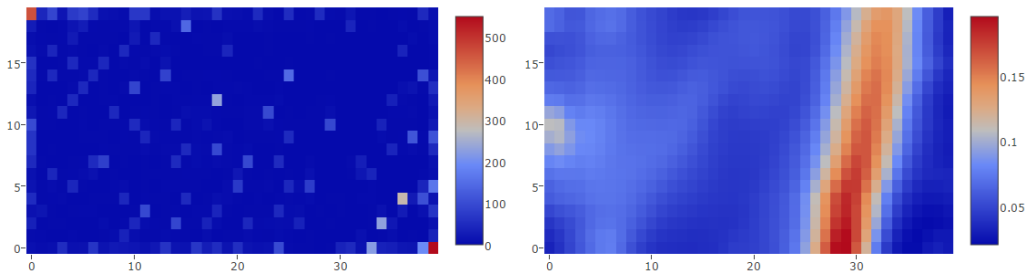
(d) U-matrix

Figure 6.15: Case study – Iris, graphical representation of node A



(a) Component Plane of feature Petal Length

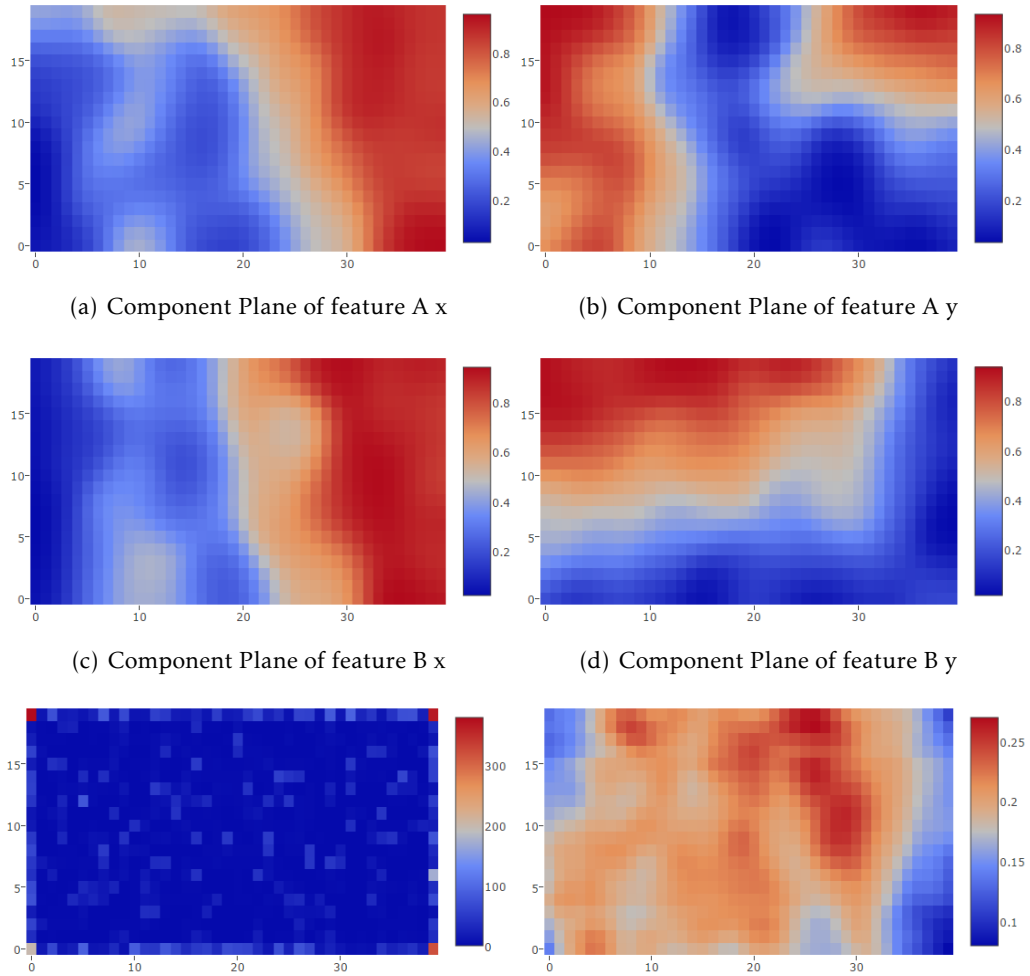
(b) Component Plane of feature Petal Width



(c) Hit-Histogram

(d) U-matrix

Figure 6.16: Case study – Iris, graphical representation of node B



(a) Component Plane of feature A x

(b) Component Plane of feature A y

(c) Component Plane of feature B x

(d) Component Plane of feature B y

(e) Hit-Histogram

(f) U-matrix

Figure 6.17: Case study – Iris, graphical representation of node C

6.2.3 Stock Market

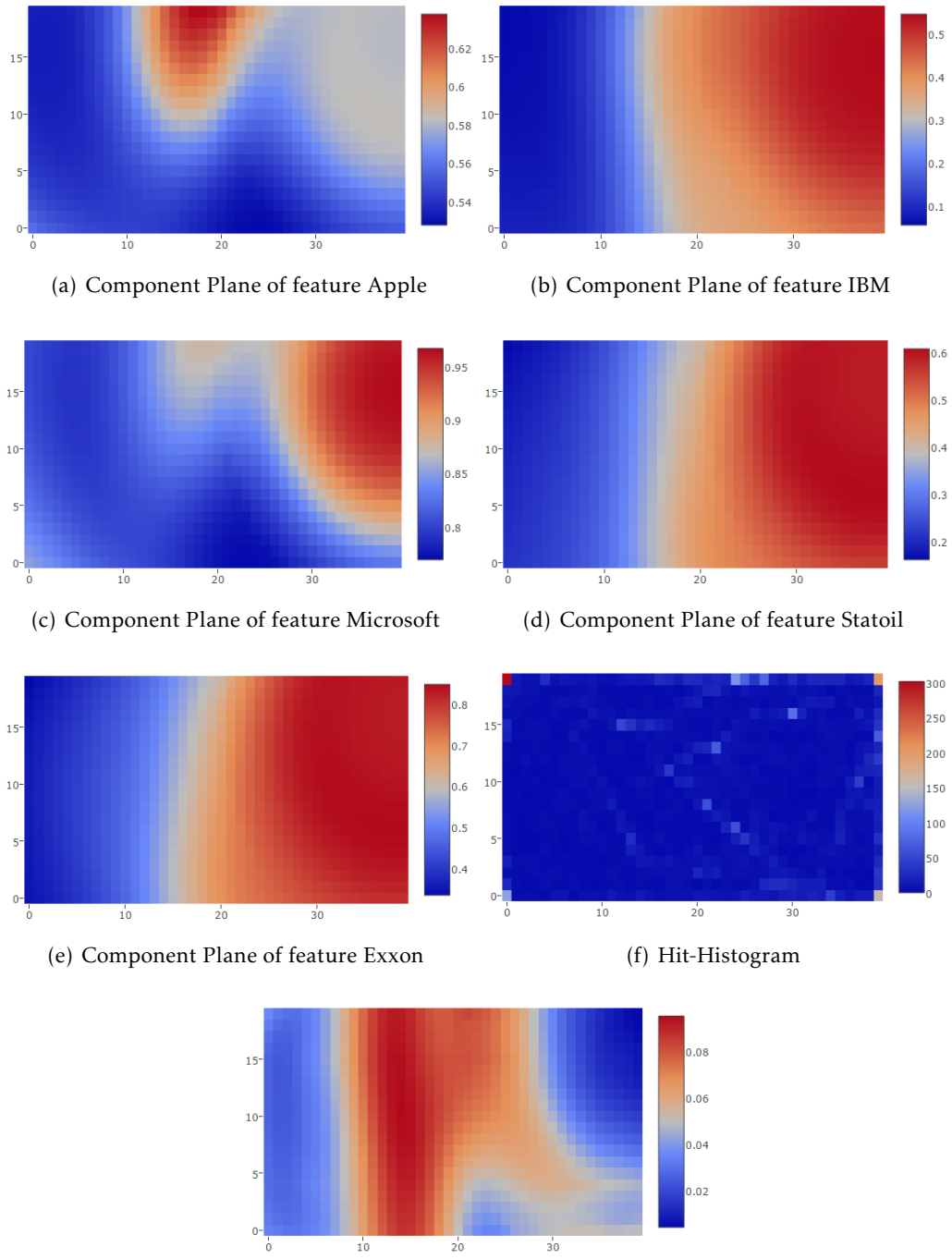
This subsection case study uses a sub set of equities of some selected products, this sub set is based on an updated version of the financial data set that was already used[58] and proposed under the framework of the project *GBFinance* (which is a research project being supported by a protocol between *NOVA-FCT* and *GoBusiness Lda*). The data set has nine columns, being the first one the timestamp column and the other eight the equity prices[58] of the companies: Apple, Statoil, Exxon, IBM, Microsoft, Amazon, Google and Facebook, respectively. The data set values have max/min normalized values (the normalization formula is the as the one in section 4.3.1), by using the global maximum and minimum price for the full range of values.

In this scenario we will use the notation introduced in section 3.2. Let TechOilDS be a data source that emits the values of Apple, IBM, Microsoft, Statoil and Exxon. Let WebDS be a data source that emits the values of Facebook, Google and Amazon. Let TechOil, Web and Global be nodes. Equation 6.2.3 defines this case study UbiHSOM structure.

$$TechOilDS \Rightarrow TechOil, WebDS \Rightarrow Web, TechOil \Rightarrow Global, Web \Rightarrow Global \tag{6.3}$$

Instead of using the web interface a Python script was developed and used to interact with the UbiHSOM REST API to accomplish the UbiHSOM structure previously defined. The script is publicly available (see section 5.6).

Figures 6.18, 6.19 and 6.20 present the graphical representation of nodes TechOil, Web and Global, respectively. Node TechOil (Figure 6.18) U-matrix shows two clear clusters, one at the left side at the map and another at the right side of the map. Node TechOil component planes also show that Apple and Microsoft are somewhat similar while Statoil and Exxon are similar, whereas IBM resembles more the oil companies than the computer related ones.



(a) Component Plane of feature Apple

(b) Component Plane of feature IBM

(c) Component Plane of feature Microsoft

(d) Component Plane of feature Statoil

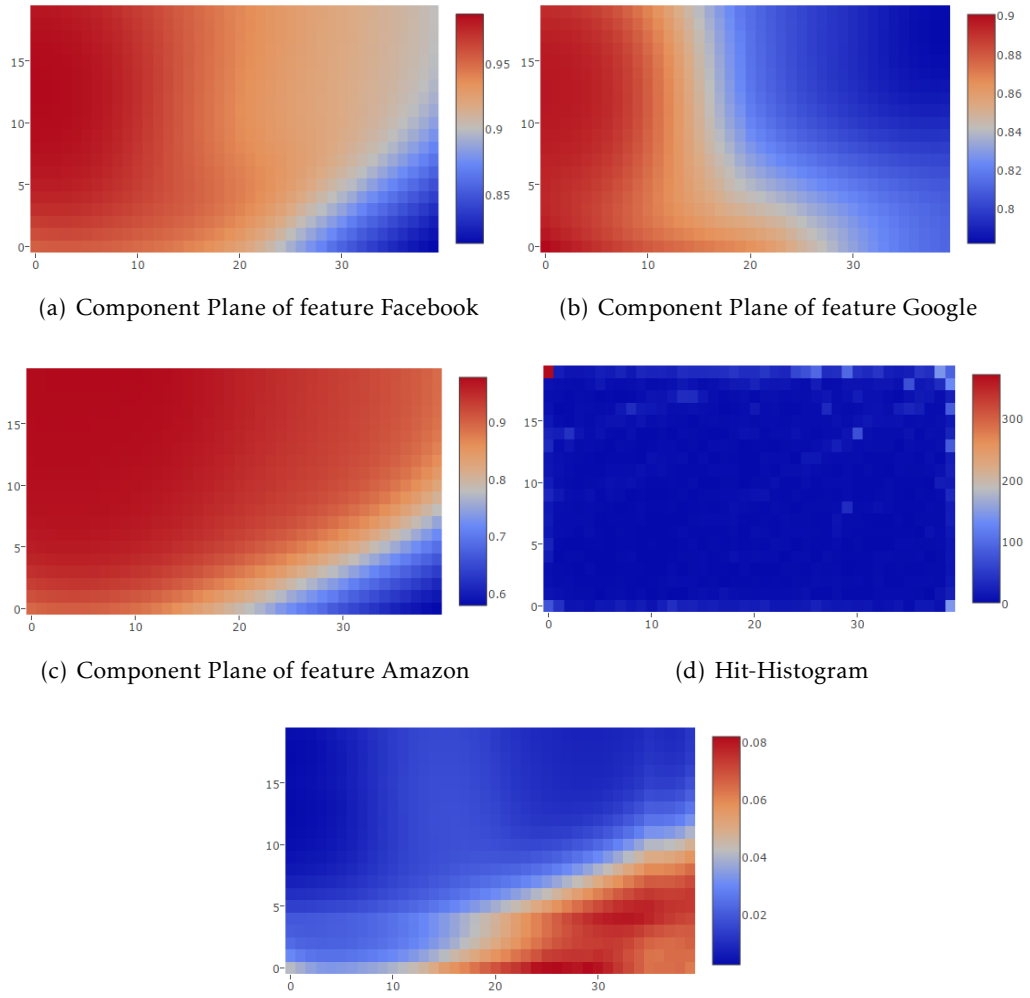
(e) Component Plane of feature Exxon

(f) Hit-Histogram

(g) U-matrix

Figure 6.18: Case study – Stock Market, graphical representation of node TechOil

The U-matrix of Figure 6.19 shows that there is not a real separation between Amazon, Google and Facebook that can possibly mean that they have similar price variations.



(a) Component Plane of feature Facebook

(b) Component Plane of feature Google

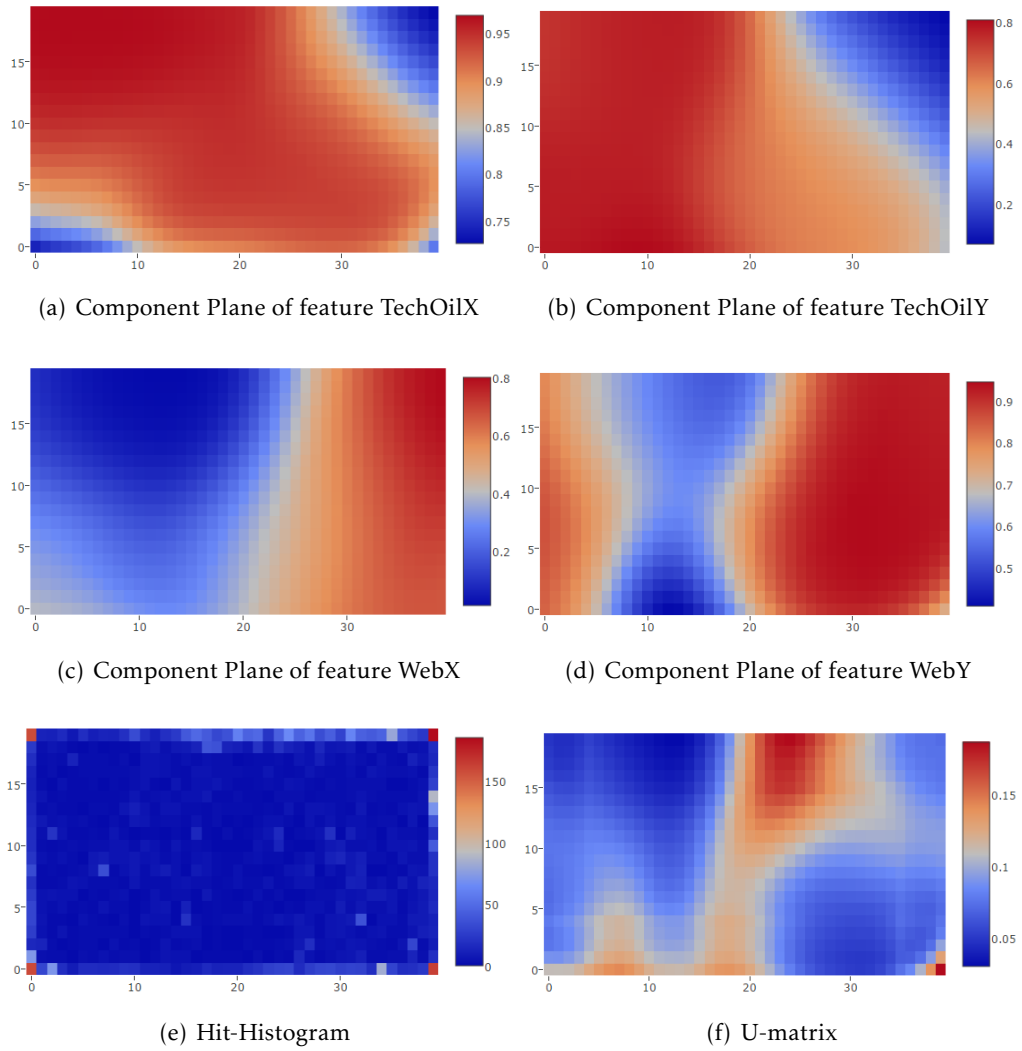
(c) Component Plane of feature Amazon

(d) Hit-Histogram

(e) U-matrix

Figure 6.19: Case study – Stock Market, graphical representation of node Web

Figure 6.20 shows two main separate areas on the U-matrix and it is possible to see that the characteristics of the node Web are: low values on TechOil Y, high values on Web X and Web Y (what characterizes TechOil node is the opposite of the Web node). This could point to different perceptions in the way investors look at IBM, that is seen as a more traditional equity. Also Google component plane slightly differs from those of Facebook and Amazon. Nevertheless, a further and more careful analysis of economical and financial reasons for these results is required, but was considered outside the scope of this thesis.



(a) Component Plane of feature TechOilX

(b) Component Plane of feature TechOilY

(c) Component Plane of feature WebX

(d) Component Plane of feature WebY

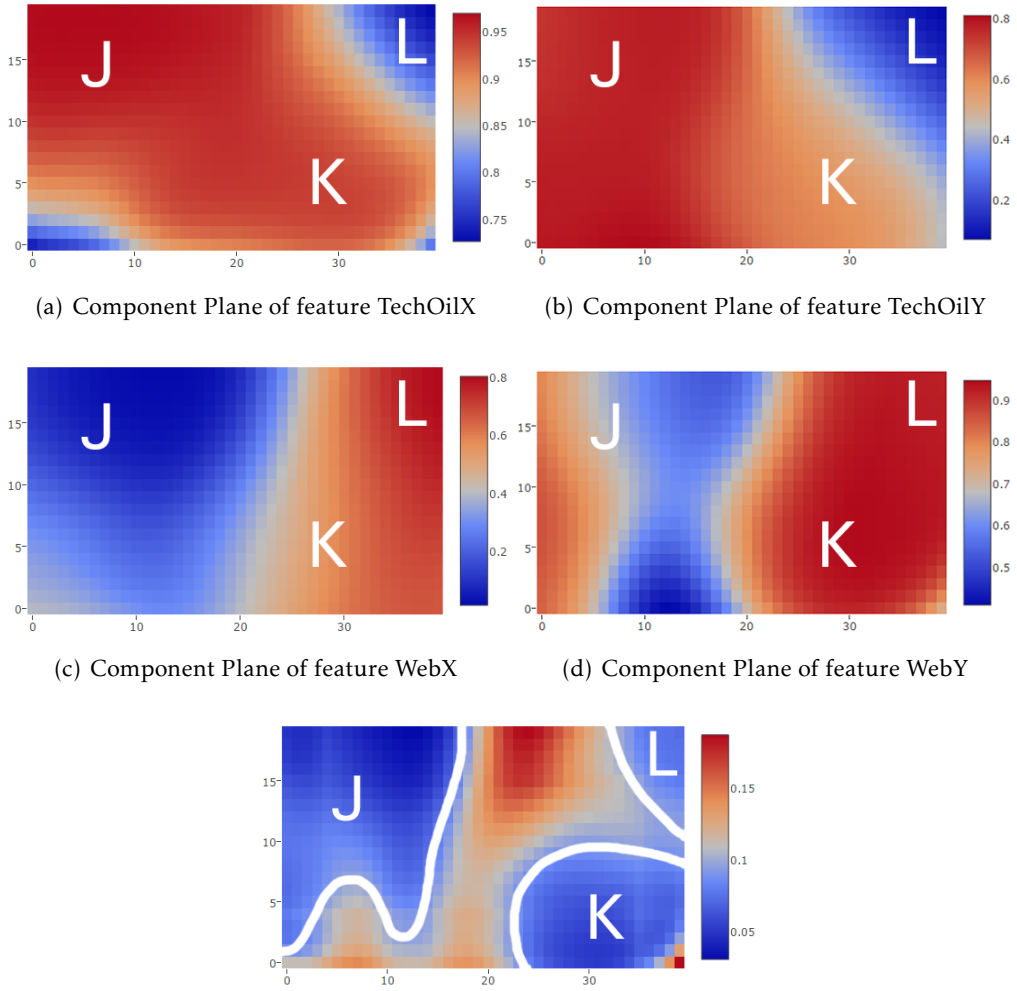
(e) Hit-Histogram

(f) U-matrix

Figure 6.20: Case study – Stock Market, graphical representation of node Global

There are three clusters labeled in Figure 6.21, cluster J, K and L. We can see the mapping of the clusters in each of the component planes of the Global node and each of the clusters is characterized by:

- cluster J - low TechOilX values, high TechOilY, intermediate WebX values and intermediate/high WebY values:
- cluster K - high TechOilX values, intermediate TechOilY, WebX and WebY values;
- cluster L - low TechOilX and TechOilY values, and high WebX and WebY value.



(a) Component Plane of feature TechOilX

(b) Component Plane of feature TechOilY

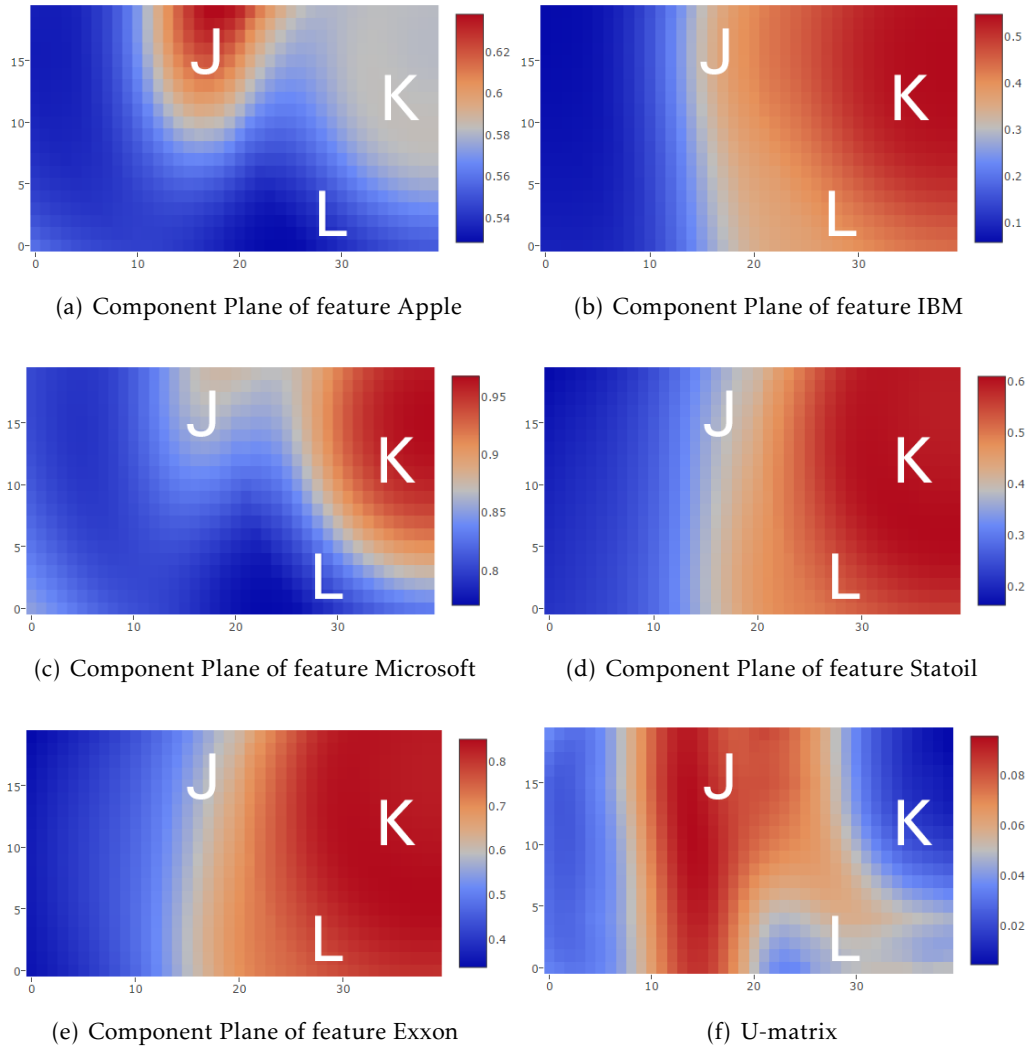
(c) Component Plane of feature WebX

(d) Component Plane of feature WebY

(e) U-matrix

Figure 6.21: Case study – Stock Market, Global node clusters

Figure 6.22 shows the mapping of the Global node clusters to the TechOil node component planes. We can see that the K, L clusters map to the zones where the values of IBM, Statoil and Exxon are high, with the exception of Microsoft, where L has low/intermediate values and K has high values . Whereas, cluster J presents high values only on the Apple component plane and has intermediate values on the others component planes.



(a) Component Plane of feature Apple

(b) Component Plane of feature IBM

(c) Component Plane of feature Microsoft

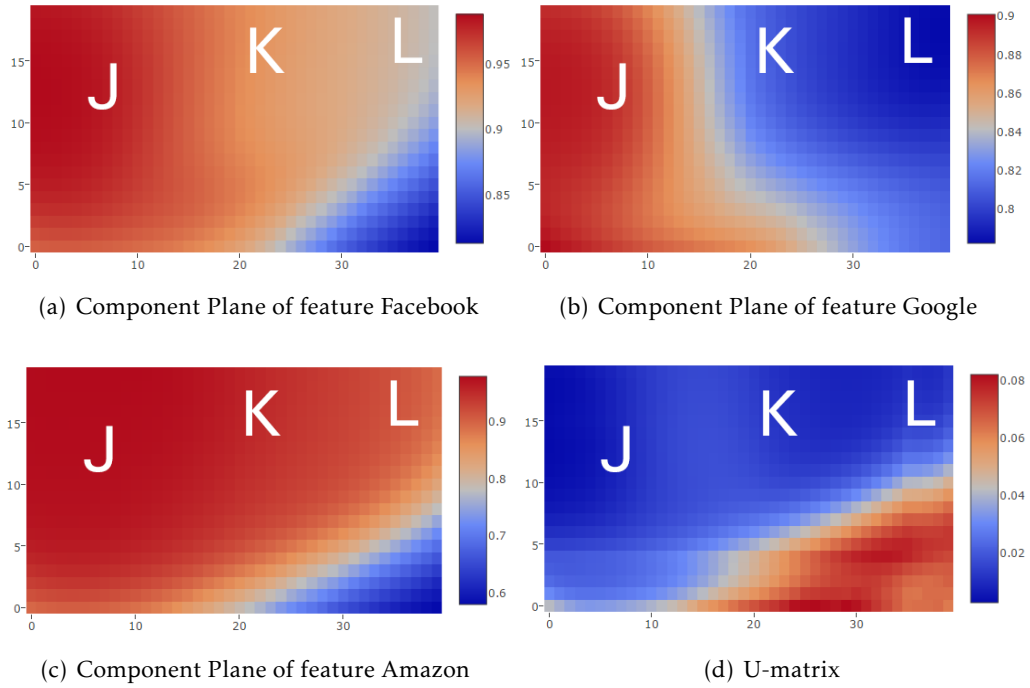
(d) Component Plane of feature Statoil

(e) Component Plane of feature Exxon

(f) U-matrix

Figure 6.22: Case study – Stock Market, TechOil node clusters

Figure 6.23 shows that Google only has high values on cluster J, Facebook has high values on cluster J and K and intermediate/low values on cluster L. Amazon has high values on all of the clusters.



(a) Component Plane of feature Facebook

(b) Component Plane of feature Google

(c) Component Plane of feature Amazon

(d) U-matrix

Figure 6.23: Case study – Stock Market, Web node clusters

It is interesting to see that Apple is the only feature of the TechOil node with high values on cluster J, moreover, cluster J is the only cluster that has high values on the three features of of node Web, Facebook, Google and Amazon. This can possibly mean that between Apple, Microsoft and IBM, Apple is the one that is more similar with the internet/web companies (Facebook, Google and Amazon). Additionally, Google is the only feature of the Global node that presents intermediate/low value on clusters K and L, where the features IBM, Statoil and Exxon of node TechOil have shown to have high values and Apple has low values on those clusters.

6.3 Discussion

Section 6.1 used the UbiFactory service capabilities to analyze if the UbiHSOM nodes behaved as expected and how they reacted to a change in the data that started by sending data points regarding the square data set and suddenly changed and started sending points regarding the triangle data set. The UbiHSOM nodes were able to adapt to those the stream changes as well as present the correct results.

Section 6.2 presented three case studies using the UbiHSOM API. The first case study aimed to test UbiHSOM to see if it was able to represent the eight vertices of the cube in a third tier node. The points of a cube were split into x,y and y,z data points and fed into two lower tier nodes that connected to a global node with the objective of having the global node associate the y component of the data points, which successfully associated and identified the eight vertices on the global node. We have also seen that the data streamers configuration can have a huge impact in the results. In addition, we analyzed the Iris data set (in section 6.2.2) in order to see if the UbiHSOM was able to separate the distinct iris flower species, the *Iris Setosa* and *Iris Virginica*. The UbiHSOM was not only able to separate those to species but it also able to separate the *Iris Versicolor* specie too.

Regarding the stock market case study (please refer to section 6.2.3), we can verify that the taxonomy construction needs particular attention, i.e., the comparison between similar features decreases the comparable terms of the higher tier node (in this case, node C). Which was verified in node B (the one with Facebook, Google and Amazon features), where it emphasized the price variations of Google in comparison with those of Facebook and Amazon that could have been less if compared with other features (e.g. the oil companies).

The UbiHSOM analysis process turned out to be a very exhaustive and non-intuitive one due to the complexity of analyzing various models simultaneously. To correctly analyze the UbiHSOM maps the clusters from the higher tier nodes were cross-referenced with labels (J,K,L) to the lower tier ones. We have seen in the cube case study (in section 6.2.1) that the UbiHSOM was able to correctly map the y feature in the same component of both maps, moreover, the UbiHSOM was also able to separate the most distinct classes of the iris data set, the *Iris Setosa* and *Iris Virginica*. Giving evidence that the UbiHSOM adds values to the data analysis because it takes into account not only the correlations between maps but also a temporal component (in comparison with the classic HSOM).

Chapter 7

Conclusions and Future Work

7.1 Conclusions

A set of microservices that integrate the functionalities identified in chapter 3 were specified in chapter 4 along with some usage examples, also, further detail about their implementation was given in chapter 5. Nonetheless there are still some bugs to fix as well as performance optimization regarding system features in order to release a stable version ready for production.

There is still many room for improvement, namely in the *DataStreamers* service. At this stage the *DataStreamers* are still very basic with not much computation associated that if implemented could potentially increase the system utility and flexibility in dealing with data. For instance, it could offer the possibility that users can define a regular expression, which is applied to the forwarded data. Besides that, the *DB DataStreamers* lack diversity because the service only offers a very limited collection of data sets. Also, the UbiHSOM nodes input channels can directly receive a data stream, e.g., data provided by devices in a Internet-of-Things setting. Although this would required more case studies that were considered out of scope. A good example of such automation would be a smart verticle that could crawl and download from a database of data sets[67] the desired data set and automatically insert it and stream it, without human intervention.

The *UbiFactory* service can also be significantly improved specially the *UbiHSOM node*. Even thought some features were added to the UbiSOM model, there are many others waiting to be implemented, e.g, prototype labeling, dynamic feature scaling normalization, dynamic dimensionality adaptation, data re-sampling policies, among many others. In addition to the dynamic feature scaling normalization there are many other forms of normalization that can be implemented which could increase UbiSOM results precision

and reliability. Since UbiSOM reacts to a stream of data that is unstructured, the underlying data dimension can change without warning, the current solution cannot cope with this problem. Thus dynamic dimension adaptation would increase the UbiSOM flexibility and ubiquity. The SOM-ward[75] could also be added as a feature of the UbiHSOM node thus increasing the user analysis power and efficiency. Since the service uses *MongoDB* for persistence the underlying map of the deployed nodes could save a snapshot from time to time, those snapshots could later be batch processed using the native *MongoDB* MapReduce functionality.

Regarding the UbiHSOM service, its improvements could range from switching to a multi-tree implementation to complex map merging. This is, the current solution features a directed graph implementation which could be replaced by other data structure. Furthermore the merging of the maps is done through the activated BMUs from lower tier maps, there are many others alternatives that could have been used, e.g. [61], [21], [33], [34], [37], [41].

All of the services do not imply model ownership, i.e., the created *DataStreamers*, *UbiHSOM nodes* and UbiHSOMs do not have a owner, the system does not log to whom the assets belong to. This can be achieved by implementing another microservice that deals with all the security related details such as registration, authentication, authorization, ownership, privacy, etc. In order to ensure ownership the user would have to follow a basic login model:

1. Register in the system
2. Confirm registration
3. Authenticate using its credentials
4. Ticket issuing

After that the system would issue a ticket[44] that the user would use for every future interaction. This is an example of a model that secures data ownership thus enforcing that each user could only see and manipulate his data. This type of approach could possibly be used to control what type of resources can each user interact with and the bandwidth allowed for each of them.

The UbiHSOM use case diagram that is illustrated in Figure 4.3, shown the actors that interact with the system as well as some of the system requirements. It is possible to observe that the *Mongo service* actor has the higher number of interactions. This means that it can become a bottleneck. There are many ways to tackle this issue, one of them (and probably the go-to solution) is to distribute the database. This solution can however not be the ideal solution, because there are many details that have to be taken into account. For instance, it could be advantageous to decouple the persistence services and have one per microservice but careful must be taken, because such solution could break important system boundaries.

It was shown in section 6.1 that it was possible to cope with real-time data analysis using the suggested services. The SOM variant, UbiSOM, was adapted and extended to deal with the financial data (section 2.7) and find correlations between it. The proposed solution in section 1.3 aimed for a system that could increase developers and data scientists options when it comes to build UbiSOM based solutions. A GitHub repository that hosts an implementation of those services is offered (please see section 5.6). The case studies in chapter 6, give evidence that the suggested services can indeed be used to build powerful and lightweight solutions. Furthermore, by decoupling concepts, a more generic solution was also obtained, one that can cope not only with the financial data but with other data taxonomies. A highly scalable, ubiquitous and responsive solution was achieved.

The proposed UbiHSOM model has shown that it can improve data analysis capabilities when it comes to finding correlations in data that follows a hierarchic taxonomy, moreover, it does not only find correlations between the data but it also adds a temporal component that the classic HSOM does not have. The studied data sets verified that it was able to find those correlations without loosing the underlying data dimensionality, e.g., in the cube case study it was able to retain the y component. Additionally, we have seen how the data streamers configurations can impact the UbiHSOM results, however, further tests must be made in order to identify which configuration is better. The stock market case study shown that an UbiHSOM taxonomy that has nodes with few similar features could possibly reduce the comparable terms of the higher tier UbiHSOM nodes, nonetheless, further tests should be conducted. We can conclude from the case studies chapter, that the UbiHSOM analysis process is exhaustive and non-intuitive due to the cross-referencing process between the higher tier UbiHSOM nodes and the lower tier ones since they have to be simultaneously analyzed. In section 6.2, the UbiHSOM analysis of the various case studies was accomplish through the cross reference method, i.e., we manually compared the various areas (clusters) of the maps resulting in a table with the mapping between the higher tier nodes and the lower tier nodes.

7.2 Future Work

This sections focuses on discussing the possible new scenarios brought by the solution and alternatives that could have been used to tackle the challenges.

Apache Spark and Storm frameworks were introduced as being a potential tool to implemented the desired system. Vert.x was chosen at the end but that does not close the possibility of a solution that is based on either Apache Spark or Storm. If reliability is preferred over ubiquity and agility, Spark or Storm are probably the way to go. Spark is advertised as being one hundred times faster than MapReduce in memory or ten times on disk. Moreover it offers the DAG feature that is really similar to the proposed graph solution (see section 3.2). Storm on the other hand is said to be the Hadoop of real-time

processing, by offering the Spout + Bolt mechanism one can build really powerful and extensible solutions. Please note the similarities between DataStreamers plus UbiHSOM node and Spouts plus Bolts, the model is really identical.

AWS Kinesis and Lambda The 6.1 case study demonstrated that the solution can be deployed on the AWS public cloud. AWS offers an enormous amount of services that ease the application development and maintenance. Two of them are the Amazon Kinesis and AWS Lambda, early introduced in section 2.13.2. The proposed solution could be modeled using such services. Even though they are payed services, they ensure reliability, scalability and monitoring. It also widens improvements because the AWS cloud offers so many services like GPU computing, which could be used to batch process old UbiSOM snapshots. Moreover it easily integrates with other tools (e.g. Spark and Storm).

Hadoop MapReduce Some researches have put effort into adapting SOM to the MapReduce model [72], [62],[74]. Even though MapReduce does not deal with real-time data, it can still be used to process past UbiSOM snapshots. A solution that takes as input old UbiSOM snapshots and tries to extract more knowledge from it (possibly empowering future correlation analysis) is an interesting one.

Cross-referencing automation We have seen that the cross-referencing process is an exhaustive and non-intuitive one, because in section 6.2 the cross-referencing process was done manually through tables. Hence an interactive tool that cool automate the process [40] would potentially increase the UbiHSOM analysis process.

The list above shows a few possible suggestion of matters that result from this thesis solution as alternative solutions or extensions of the proposed one, since the source-code is publicly available (see section 5.6) and can be used by anyone to conduct experiments like the ones in chapter 6. The UbiHSOM itself has a fair bit of space for improvements, e.g., due to its nature, i.e., the directed graph, one could view the model as a very primitive one of a DeepMind-like based SOM solution, dubbed DeepSOM, for instance.

Bibliography

[1] *Agile coding in enterprise IT: Code small and local*. URL: http://usblogs.pwc.com/emerging-technology/agile-coding-in-enterprise-it-code-small-and-local/.

[2] *Akka Java Documentation*. Report. Typesafe Inc, 2015.

[3] *Alistair Cockburn*. URL: http://alistair.cockburn.us/Hexagonalarchitecture.

[4] *Amazon Kinesis Streams – Amazon Web Services (AWS)*. URL: https://aws.amazon.com/kinesis/streams/.

[5] *Amazon Web Services (AWS) - Cloud Computing Services*. URL: https://aws.amazon.com/.

[6] *Apache Spark™ - Lightning-Fast Cluster Computing*. URL: http://spark.apache.org/.

[7] *Apache Storm*. URL: http://storm.apache.org/.

[8] *AWS Lambda - Serverless Compute*. URL: https://aws.amazon.com/lambda/.

[9] *AWS Service Limits*. URL: http://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html.

[10] F. Bação, V. S. Lobo, and M. Painho. "Self-organizing Maps as Substitutes for K-Means Clustering". In: *Computational Science – (5th ICCS'05, Part III)*. Ed. by V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. Dongarra. Vol. 3516. Lecture Notes in Computer Science (LNCS). Reading, UK: Springer-Verlag (New York), May 2006, pp. 476–483.

[11] C. M. Bishop. *Pattern Recognition and Machine Learning*. eng. 2006. URL: http://cds.cern.ch/record/998831.

[12] J. Bonér, D. Farley, R. Kuhn, and M. Thompson. *The reactive manifesto*. 2014.

[13] *BSON*. URL: http://bsonspec.org/.

[14] Carlosbate. *carlosbate/ubifactory-javascript-example*. 2017. URL: https://github.com/carlosbate/ubifactory-javascript-example.

[15] *carlosbate (rubito)*. URL: https://github.com/carlosbate.

[16] P. Clifford and R. Bhandari. *Database management system*. US Patent App. 10/488,592. 2001.

[17] *Cloud computing - statistics on the use by enterprises*. URL: http://ec.europa.eu/eurostat/statistics-explained/index.php/Cloud_computing_-_statistics_on_the_use_by_enterprises.

[18] *Cloud Services Pricing – Amazon Web Services (AWS)*. URL: https://aws.amazon.com/pricing/services/.

[19] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[20] *Eclipse Vert.x toolkit*. URL: http://vertx.io/.

[21] M. Endo, M. Ueno, and T. Tanabe. "A clustering method using hierarchical self-organizing maps". In: *The Journal of VLSI Signal Processing* 32.1 (2002), pp. 105–118.

[22] C. Escoffier. *Reactive Microservices with Eclipse Vert.x*. URL: https://www.eclipse.org/community/eclipse_newsletter/2016/october/article4.php.

[23] R. T. Fielding. "Architectural styles and the design of network-based software architectures". PhD thesis. University of California, Irvine, 2000.

[24] M. Fowler and J. Lewis. "Microservices". In: *ThoughtWorks. http://martinfowler.com/articles/microservices. html [last accessed on February 17, 2015]* (2014).

[25] *Graphic Figure of Pearson's r correlation*. URL: https://statistics.laerd.com/statistical-guides/pearson-correlation-coefficient-statistical-guide.php.

[26] M. D. Hanson. "The Client/Server Architecture". In: *Server Management* (2000), p. 3.

[27] J. A. Hartigan and M. A. Wong. "Algorithm AS 136: A k-means clustering algorithm". In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28.1 (1979), pp. 100–108.

[28] *Hazelcast the Leading In-Memory Data Grid*. URL: https://hazelcast.com/.

[29] R. A. P. Henriques. *Artificial Intelligence in Geospatial Analysis: applications of Self-Organizing Maps in the context of Geographic Information Science*. Report. Universidade Nova de Lisboa – ISEGI, 2010.

[30] P. S. Hoey. *Statistical Analysis of the Iris Flower Dataset*. 2002.

[31] *HTTP/1.1: Method Definitions*. URL: https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html.

[32] *HTTP/1.1: Status Code Definitions*. URL: https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

[33] H. Ichiki, M. Hagiwara, and M. Nakagawa. "Self-organizing multilayer semantic maps". In: *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*. Vol. 1. IEEE. 1991, pp. 357–360.

[34] N Kasabov and E Peev. "Phoneme Recognition with Hierarchical Self Organised Neural Networks and Fuzzy Systems-A Case Study". In: *ICANN'94*. Springer, 1994, pp. 201–204.

[35] *KDnuggets*. URL: http://www.kdnuggets.com/2015/06/cognitive-computing-solving-big-data-problem.html.

[36] T. Kohonen. "The self-organizing map". In: *Proc. IEEE* 78.9 (Sept. 1990), pp. 1464–1480.

[37] P. Koikkalainen and E. Oja. "Self-organizing hierarchical feature maps". In: *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*. IEEE. 1990, pp. 279–284.

[38] J. Lampinen and E. Oja. "Clustering properties of hierarchical self-organizing maps". In: *Journal of Mathematical Imaging and Vision* 2.2-3 (1992), pp. 261–272.

[39] N. Marques. "Agrupamento sobre uma matriz de distâncias UMAT–uma aplicação sobre dados financeiros". In: (2014).

[40] N. C. Marques, B. Silva, and H. Santos. "An Interactive Interface for Multi-dimensional Data Stream Analysis". In: *Information Visualisation (IV), 2016 20th International Conference*. IEEE. 2016, pp. 223–229.

[41] R. Miikkulainen. "Script recognition with hierarchical feature maps". In: *Connectionist Natural Language Processing*. Springer, 1992, pp. 196–214.

[42] *MongoDB Map-Reduce*. URL: https://docs.mongodb.com/manual/core/map-reduce/.

[43] B. Naveh. *JGraphT*. URL: http://jgrapht.org/.

[44] B. C. Neuman and T. Ts'o. "Kerberos: An authentication service for computer networks". In: *IEEE Communications magazine* 32.9 (1994), pp. 33–38.

[45] S. Newman. *Building microservices*. " O'Reilly Media, Inc.", 2015.

[46] M. Owens and G. Allen. *SQLite*. Springer, 2010.

[47] L. Oy, S. T. F. Matlab, J. Vesanto, J. Vesanto, J. Himberg, J. Himberg, E. Alhoniemi, E. Alhoniemi, J. Parhankangas, and J. Parhankangas. *SOM Toolbox for Matlab 5*. en. Tech. rep. 2000. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.7561;http://www.cis.hut.fi/projects/somtoolbox/documentation/../package/papers/techrep.ps.

[48] G. C. Panosso. *Análise do Mercado Financeiro baseada em Análise Técnica com Self-Organizing Maps*. Report. Universidade Nova de Lisboa – ISEGI, 2013.

[49] *Pattern: API Gateway / Backend for Front-End*. URL: http://microservices.io/patterns/apigateway.html.

[50] R. Perrey and M. Lycett. "Service-oriented architecture". In: *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*. IEEE. 2003, pp. 116–119.

[51] *Pokemon Dataset*. URL: http://srea64.github.io/msan622/project/dataset.html.

[52] L. F. Report, D. . 2015, D. B. C. L. J. Lyman, M. A. B. Sector, M. A. B. Acquirer, F. shown indicate number of transactions, C. M. (other), Channels, and Sectors. *2016 Trends in Development, DevOps IT Ops*. URL: https://451research.com/report-long?icid=3600.

[53] V. Sarcar. "Abstract Factory Patterns". In: *Java Design Patterns*. Springer, 2016, pp. 109–114.

[54] P. Sarlin and T. A. Peltonen. "Mapping the State of Financial Stability". In: *SSRN Electronic Journal* (). DOI: 10.2139/ssrn.1914294.

[55] *Scaling Writes on Amazon DynamoDB Tables with Global Secondary Indexes*. 2015. URL: https://aws.amazon.com/blogs/big-data/scaling-writes-on-amazon-dynamodb-tables-with-global-secondary-indexes/.

[56] A. Silberschatz, H. F. Korth, S. Sudarshan, et al. *Database system concepts*. Vol. 4. McGraw-Hill New York, 1997.

[57] B. Silva. *A Study of a Hybrid Parallel SOM Algorithm for Large Maps in Data Mining*. Report. Universidade Nova de Lisboa – FCT, 2008.

[58] B. Silva. *Exploratory Cluster Analysis from Ubiquitous Data Streams using Self-Organizing Maps*. Report. Universidade Nova de Lisboa – FCT, 2016.

[59] B. Silva and N. C. Marques. "Ubiquitous Self-Organizing Maps". In: *UDM@IJCAI*. Ed. by J. Gama, M. May, N. C. Marques, P. C. 0001, and C. A. Ferreira. Vol. 1088. CEUR Workshop Proceedings. CEUR-WS.org, 2013, p. 54. URL: http://ceur-ws.org/Vol-1088.

[60] B. Silva and N. C. Marques. "The ubiquitous self-organizing map for non-stationary data streams". In: *Journal of Big Data* 2.1 (2015), pp. 1–22. ISSN: 2196-1115. DOI: 10.1186/s40537-015-0033-0. URL: http://dx.doi.org/10.1186/s40537-015-0033-0.

[61] P. N. Suganthan. "Hierarchical overlapped SOM's for pattern classification". In: *IEEE Transactions on Neural Networks* 10.1 (1999), pp. 193–196.

[62] S.-J. Sul and A. Tovchigrechko. "Parallelizing BLAST and SOM algorithms with MapReduce-MPI library". In: *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE. 2011, pp. 481–489.

[63] S. A. H. T. Samad. "Self-organization with partial data." In: *Network: Computation in Neural Systems* (1992), pp. 205,212.

[64] P. Tiple, L. Cavique, and N. Cavalheiro Marques. "Ramex-Forum: a tool for displaying and analysing complex sequential patterns of financial products". In: *Expert Systems* (2016).

[65] P. S. Tiple. *Tool for discovering sequential patterns in financial marketss*. Report. Universidade Nova de Lisboa – FCT, 2014.

[66] *Trends in the Open Source Cloud: A Shift to Microservices and the Public Cloud*. 2016. URL: https://www.linux.com/blog/trends-open-source-cloud-shift-microservices-and-public-cloud.

[67] *UCI Machine Learning Repository: Data Sets*. URL: https://archive.ics.uci.edu/ml/datasets.html.

[68] *Unirest for Java*. URL: http://unirest.io/java.html.

[69] *Vert.x Core Manual - The Golden Rule*. URL: http://vertx.io/docs/vertx-core/groovy/#golden_rule.

[70] *Vert.x MongoDB Client API*. URL: http://vertx.io/docs/vertx-mongo-client/java/.

[71] *Web Services Glossary*. URL: https://www.w3.org/TR/ws-gloss/.

[72] C. Weichel. "Adapting Self-Organizing Maps to the MapReduce Programming Paradigm." In: *STeP*. 2010, pp. 119–131.

[73] E. Wilde. "Putting things to REST". In: *School of Information* (2007).

[74] P. Wittek and S. Darányi. "Accelerating text mining workloads in a MapReduce-based distributed GPU environment". In: *Journal of Parallel and Distributed Computing* 73.2 (2013), pp. 198–206.

[75] Z. Yao, T. Eklund, and B. Back. "Using som-ward clustering and predictive analytics for conducting customer segmentation". In: *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE. 2010, pp. 639–646.

Appendix

API Usage Examples

This appendix regards the *JSON* models that are used in the different services endpoints. The Postman *Google Chrome* extension will be used to illustrate the interactions between a user (e.g. an application or a human) and the DataStreamers, UbiFactory and UbiHSOM services.

In order to analyze it the reader must pay attention to the zones depicted in Figure A.1.

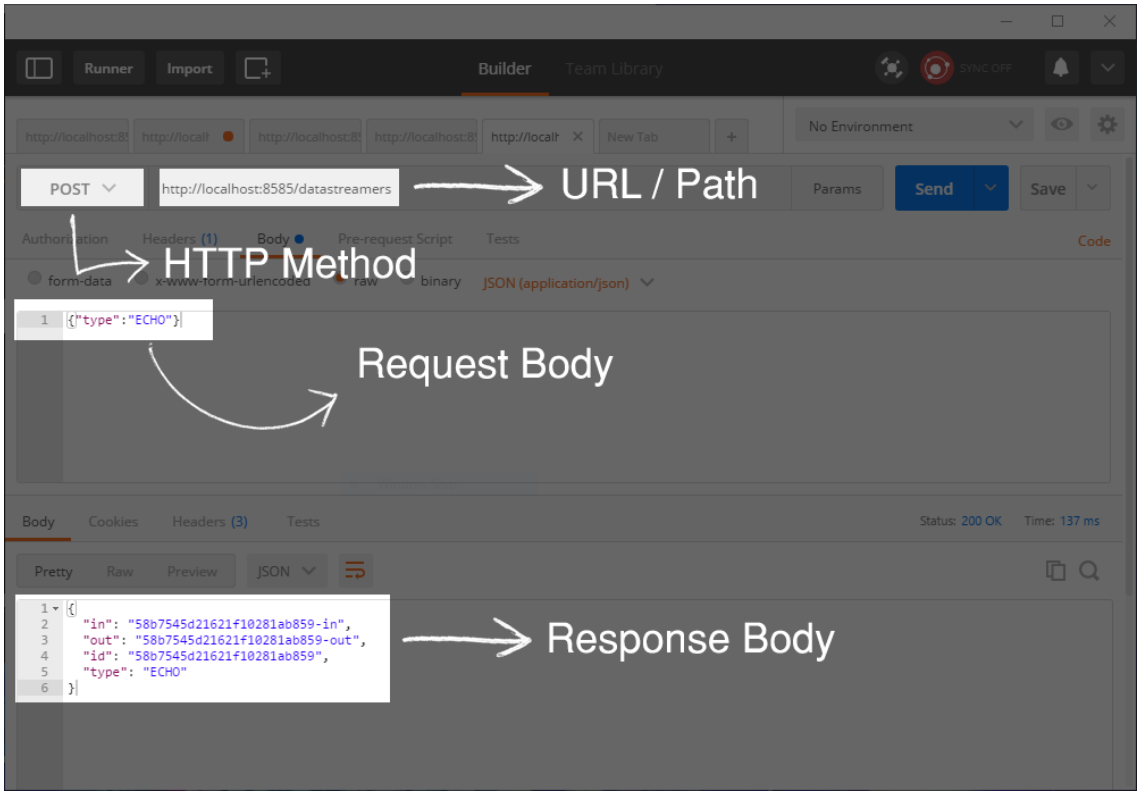


Figure A.1: Postman Important Zones

A.1 DataStreamers

The list of figures bellow shows the Postman screens for the different requests:

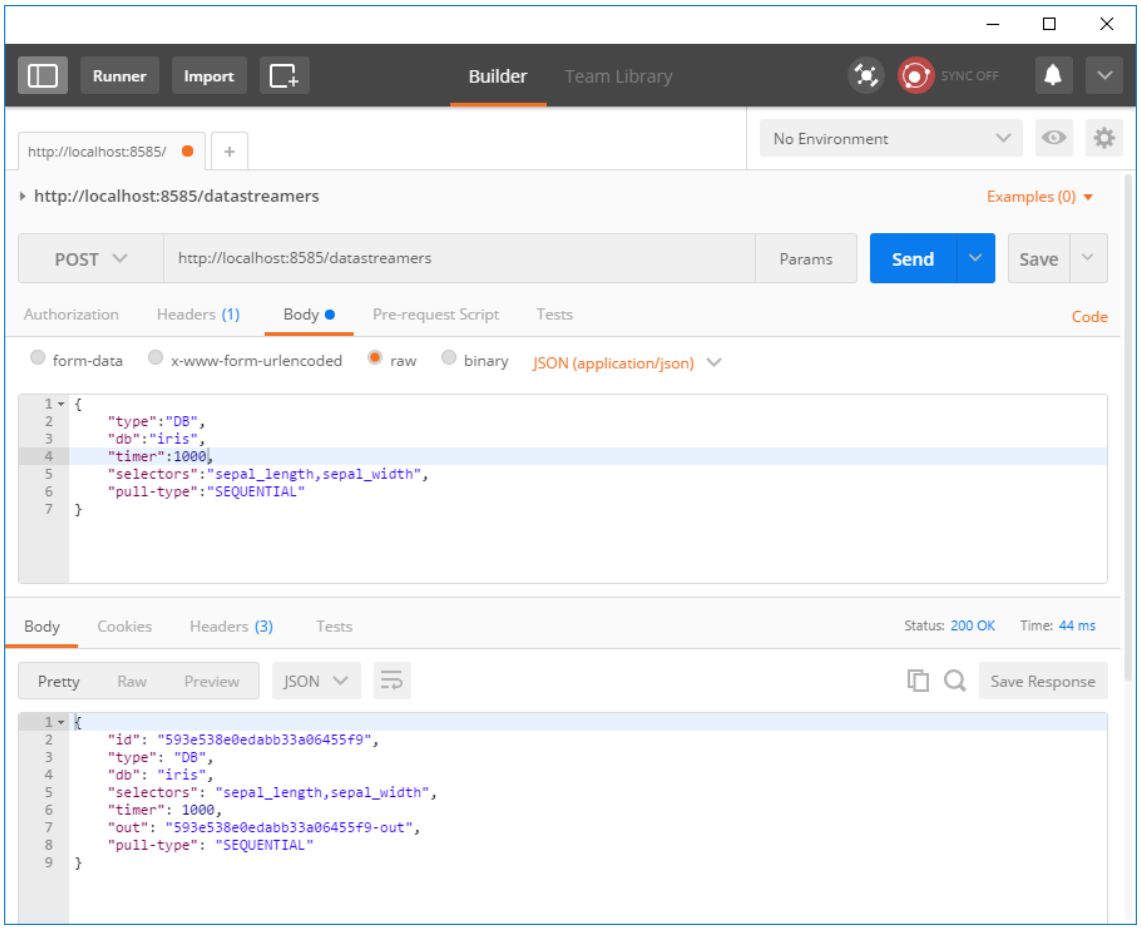


Figure A.2: DataStreamers, Postman – POST DB DataStreamer (Sequential)

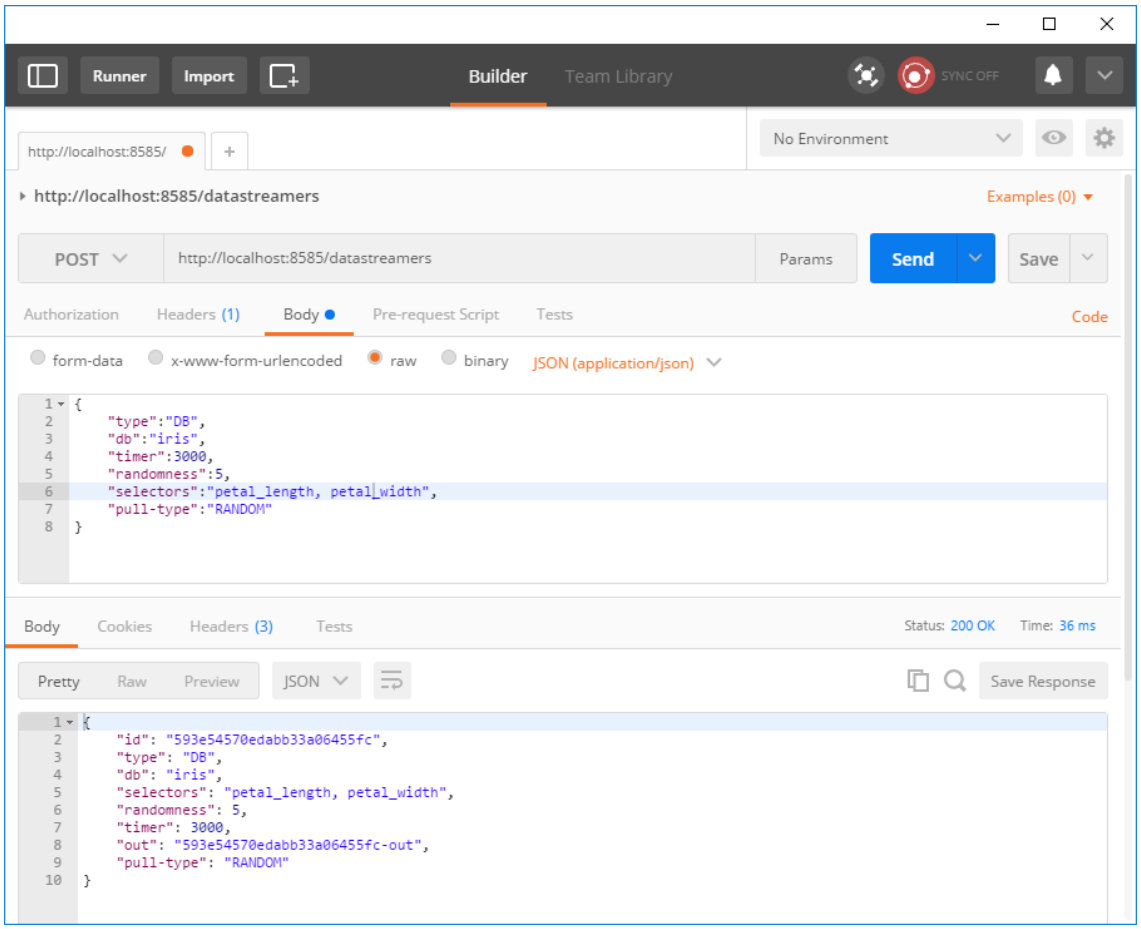


Figure A.3: DataStreamers, Postman – POST DB DataStreamer (Random)

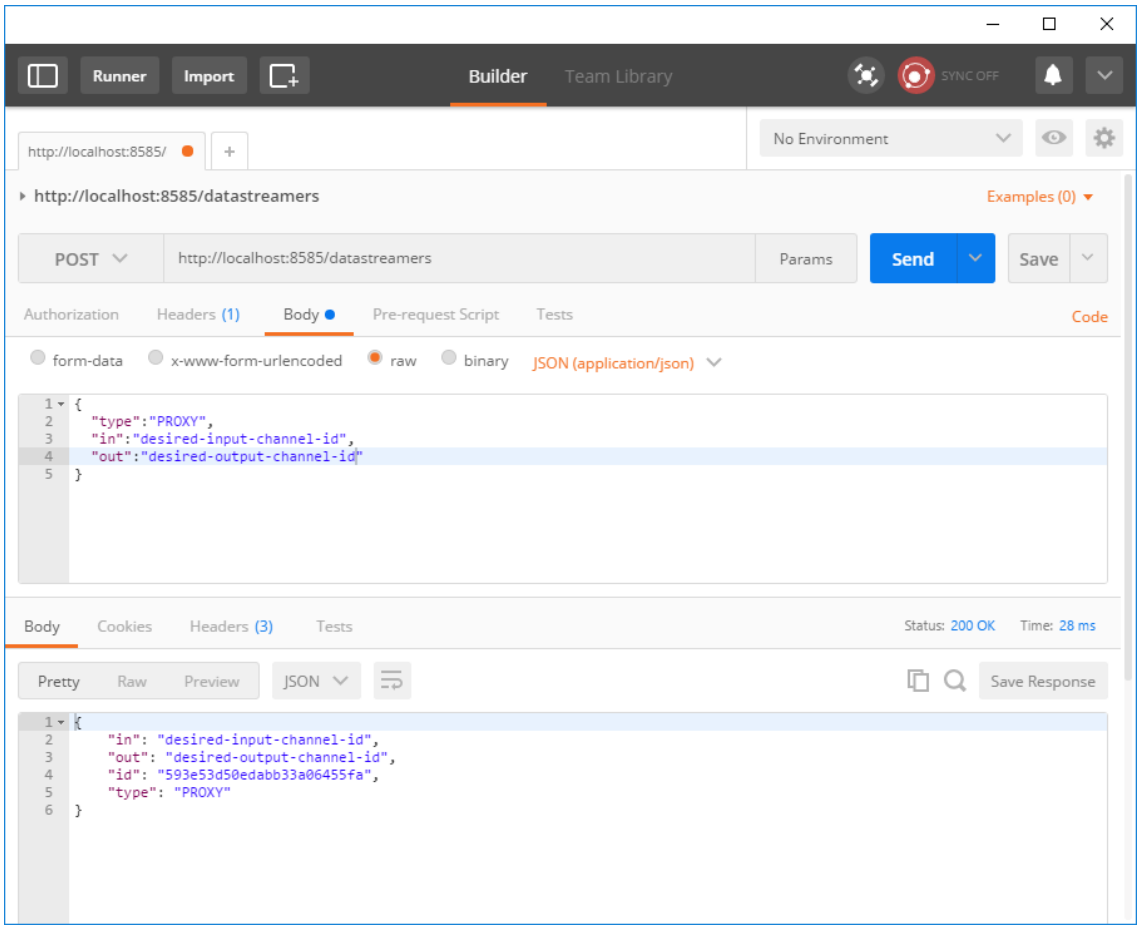


Figure A.4: DataStreamers, Postman – POST Proxy DataStreamer

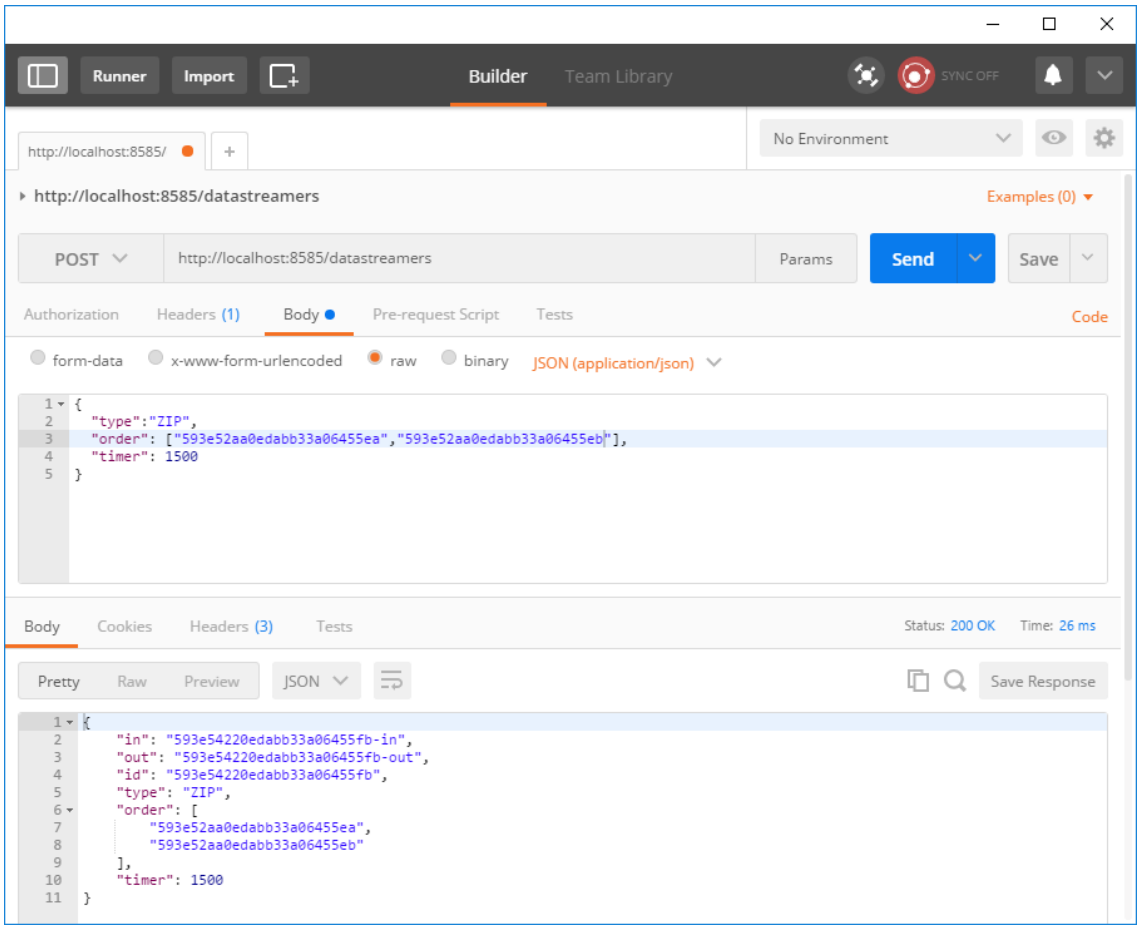


Figure A.5: DataStreamers, Postman – POST Zip DataStreamer

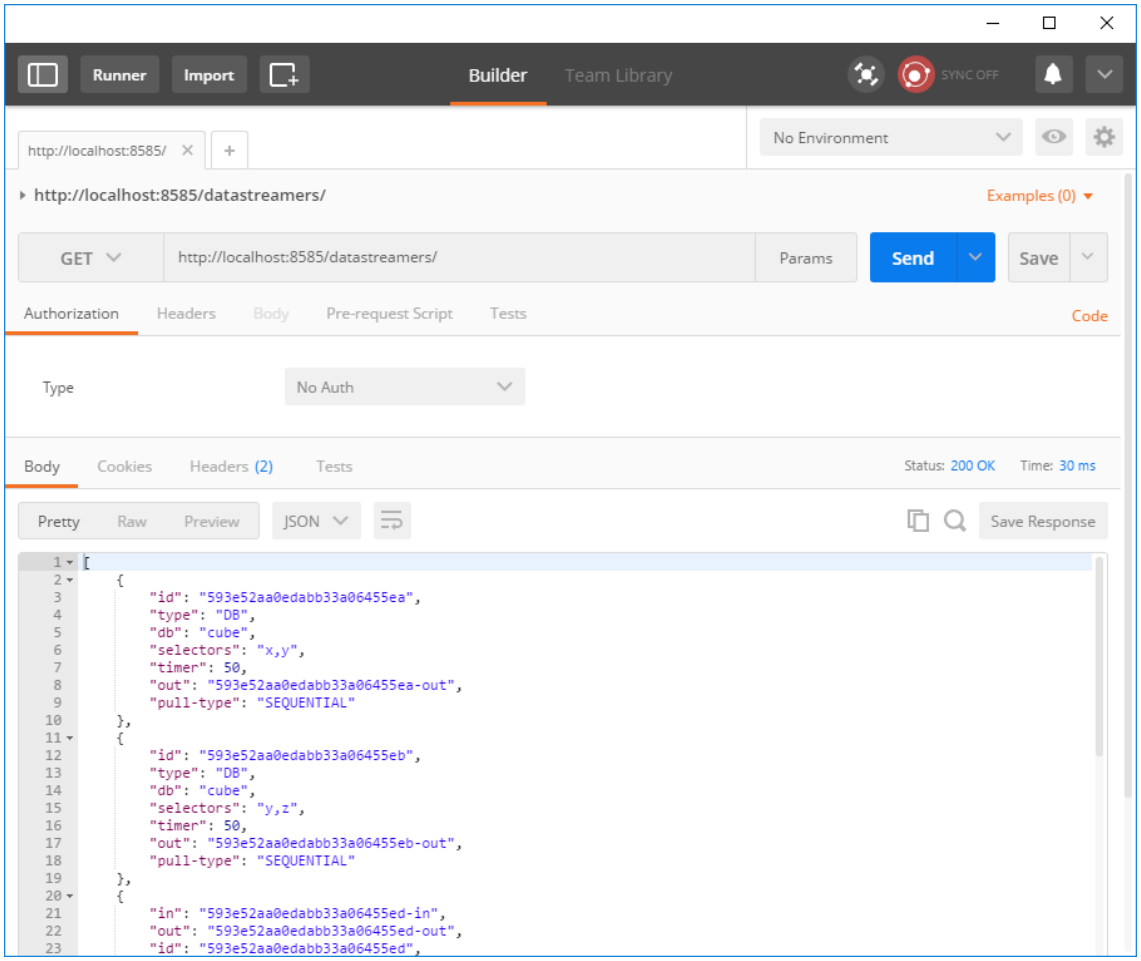


Figure A.6: DataStreamers, Postman – GET

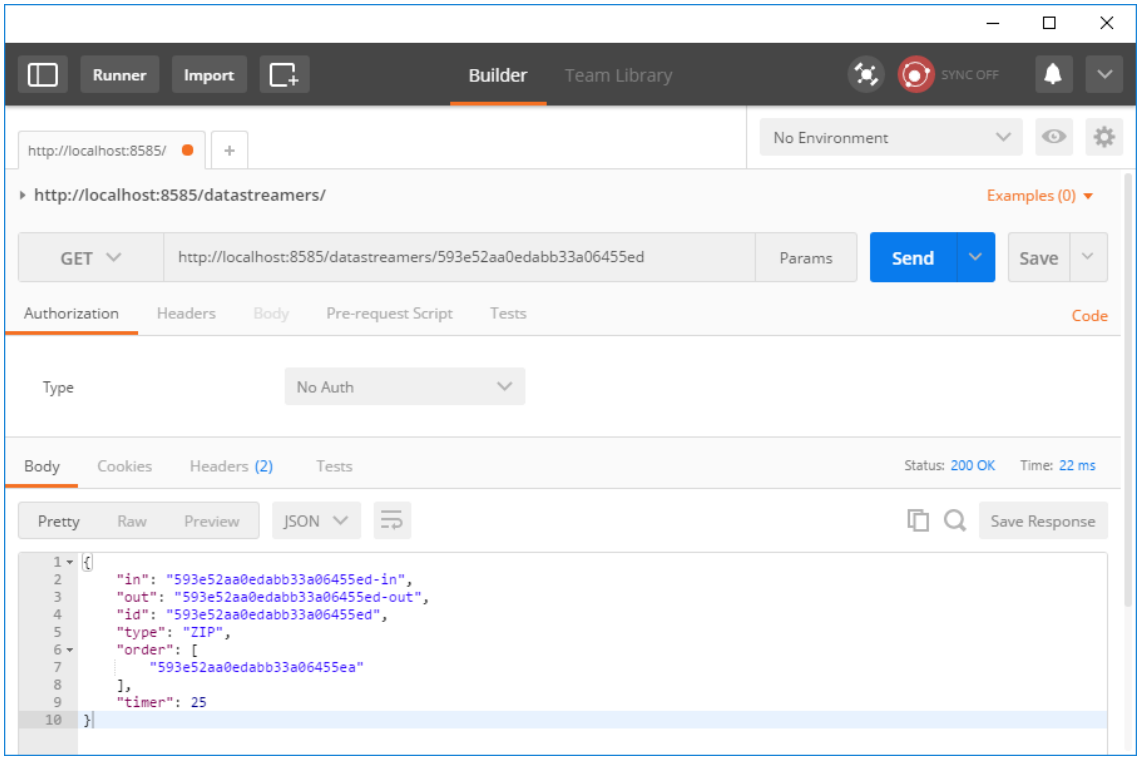


Figure A.7: Postman – GET with identifier

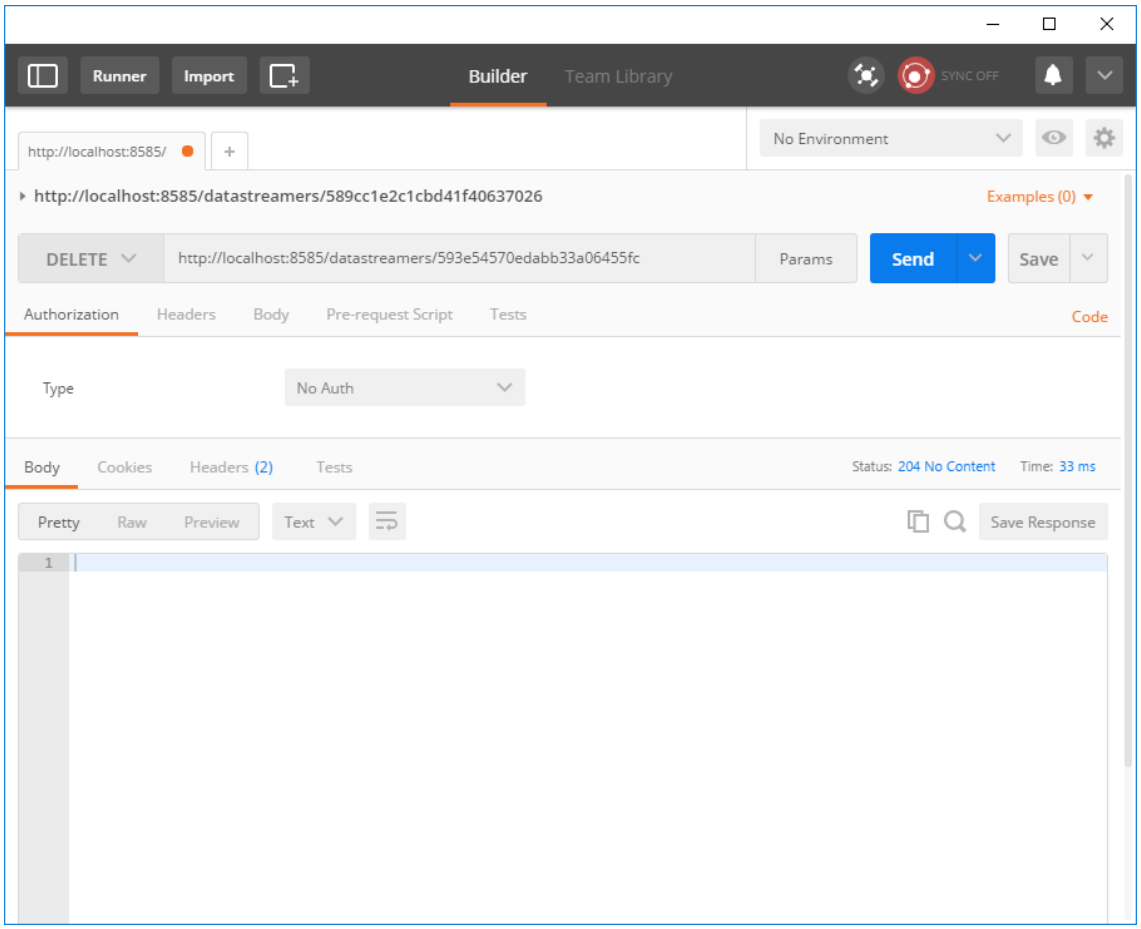


Figure A.8: DataStreamers, Postman – DELETE with identifier

A.2 UbiFactory

The list of figures bellow show the Postman screens for the different requests:

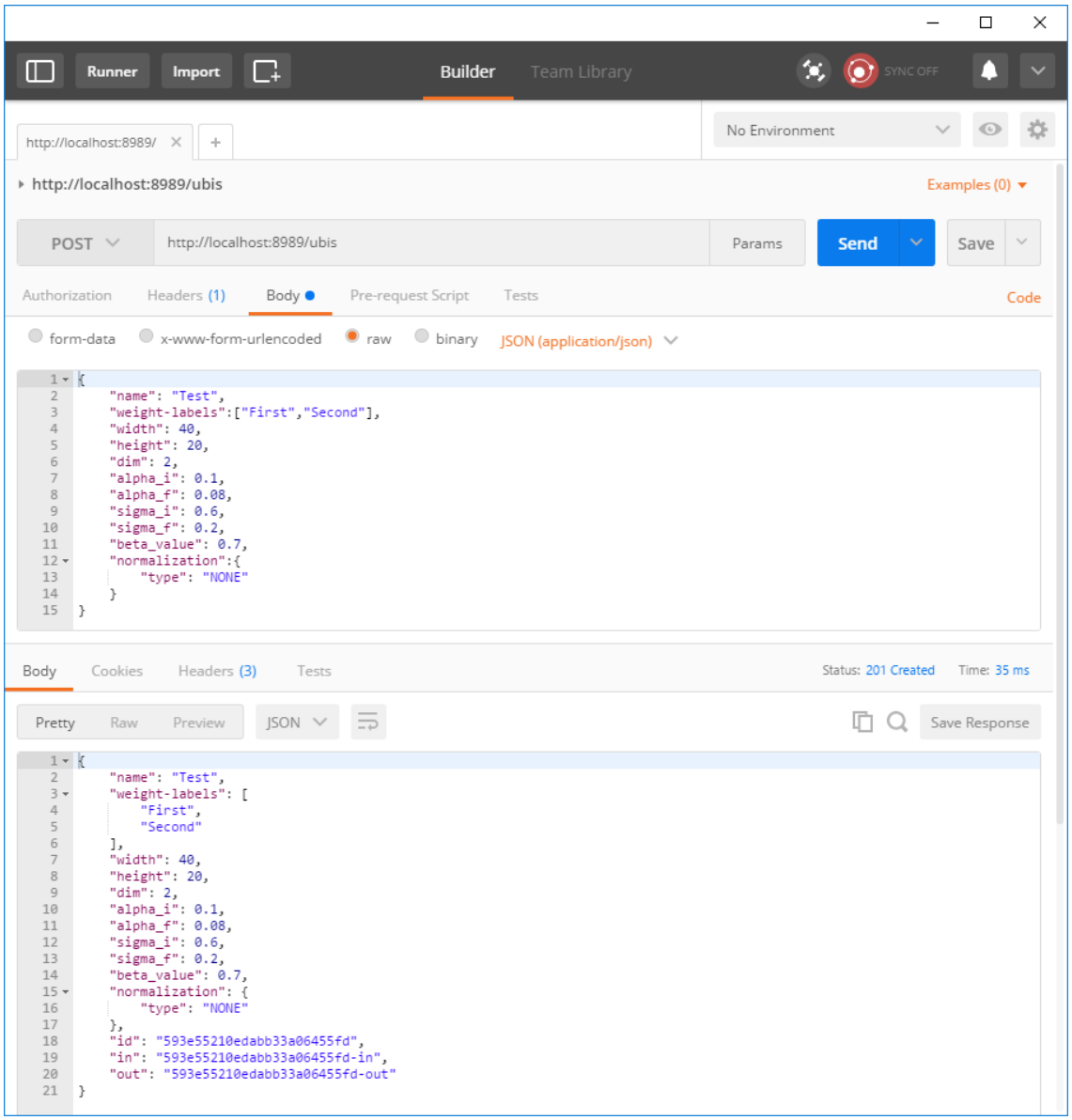


Figure A.9: UbiFactory, Postman – POST UbiHSOM node

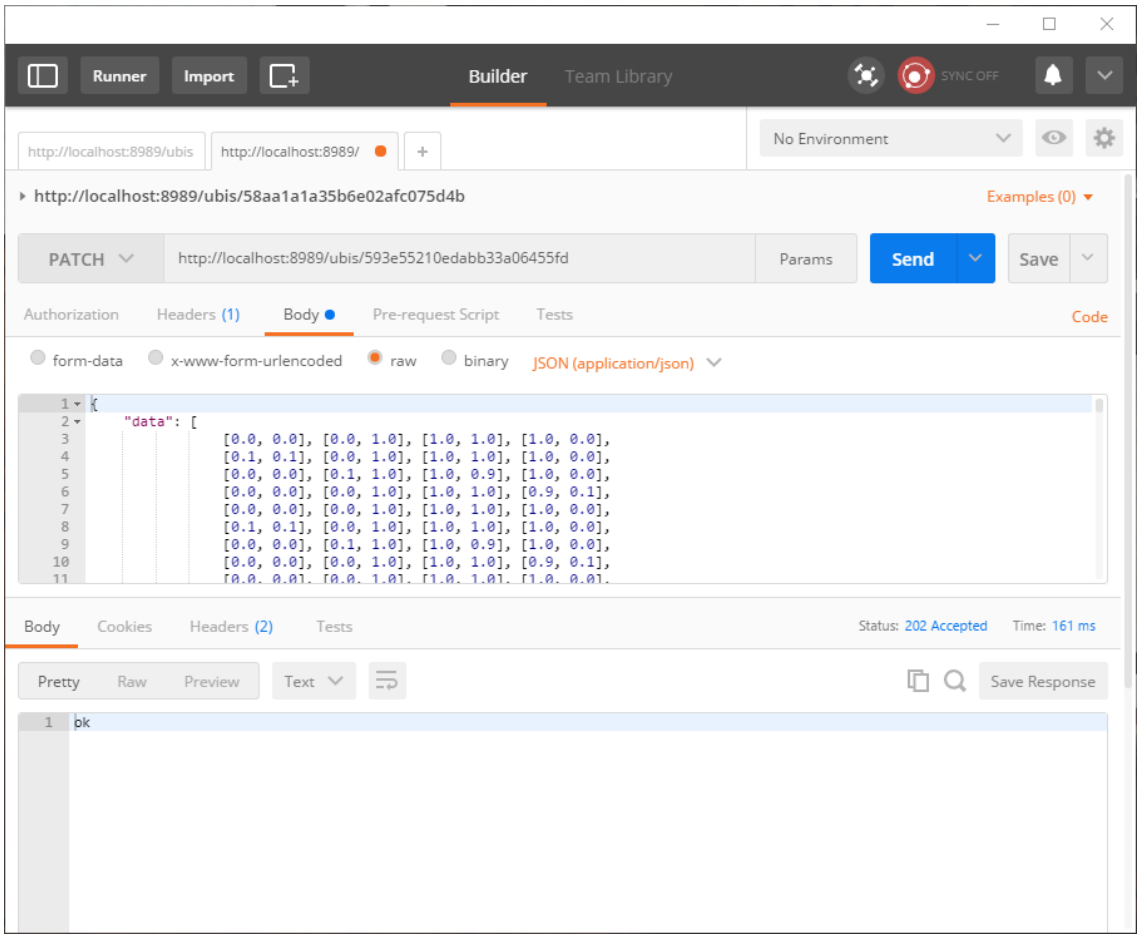


Figure A.10: UbiFactory, Postman – PATCH UbiHSOM node (send data)

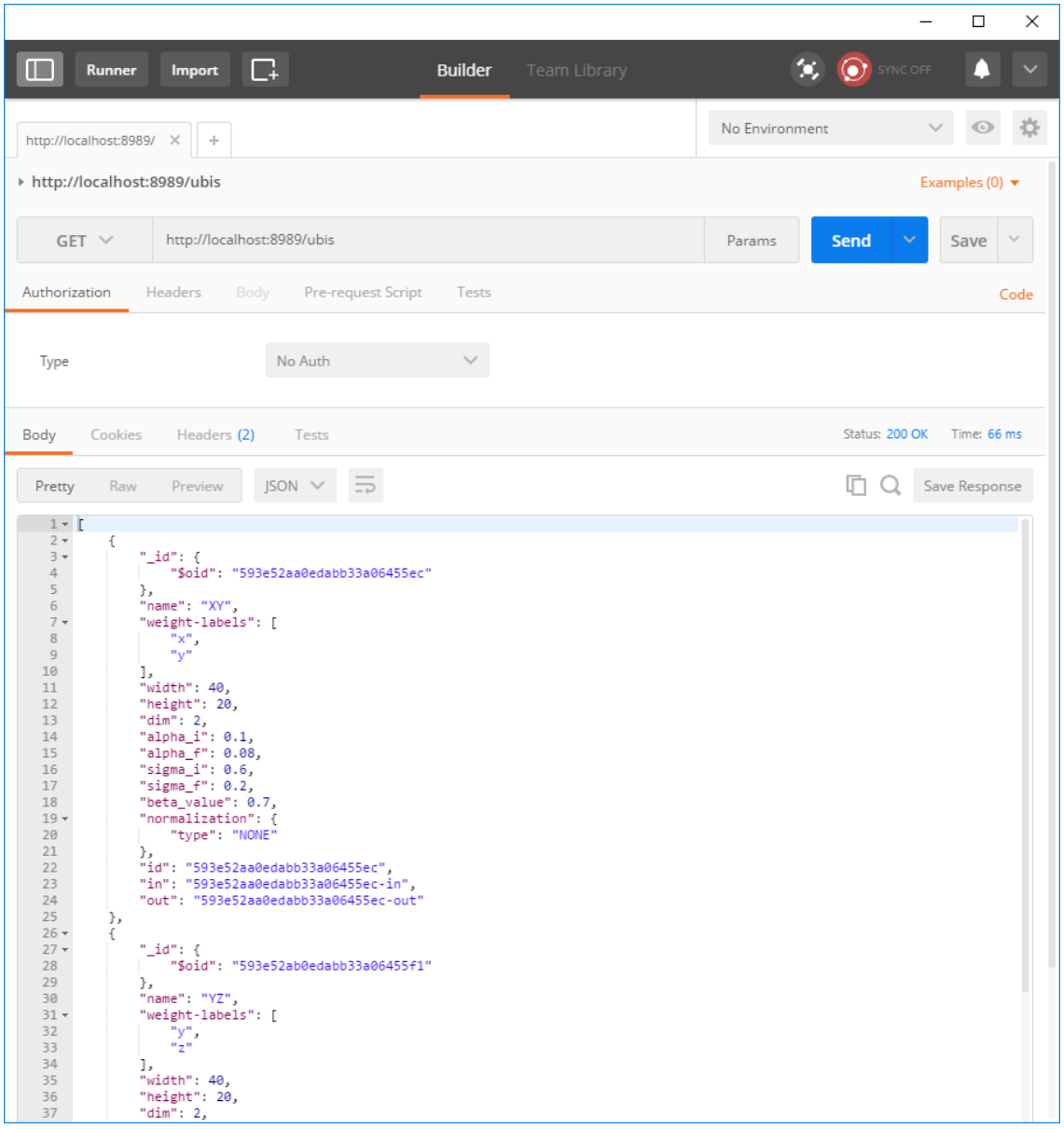


Figure A.11: UbiFactory, Postman – GET

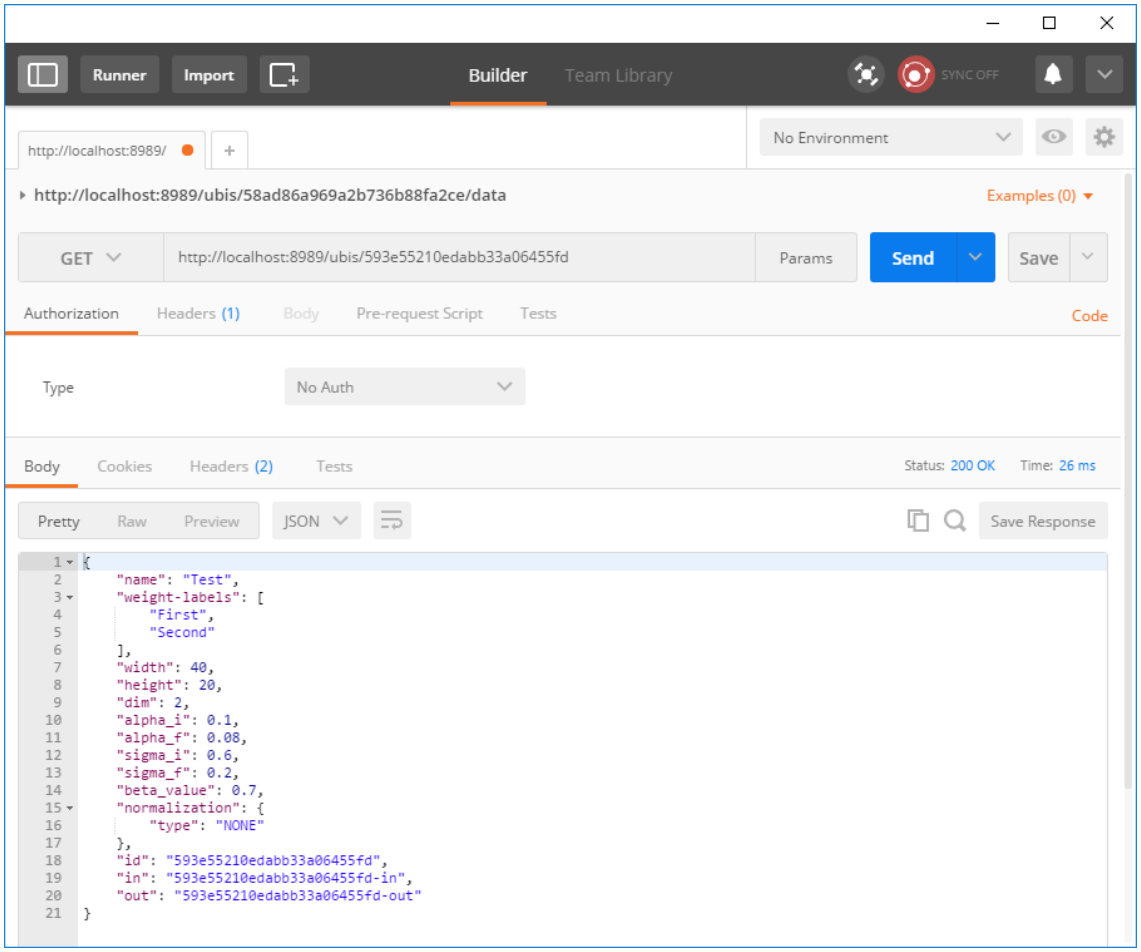


Figure A.12: UbiFactory, Postman – GET with identifier

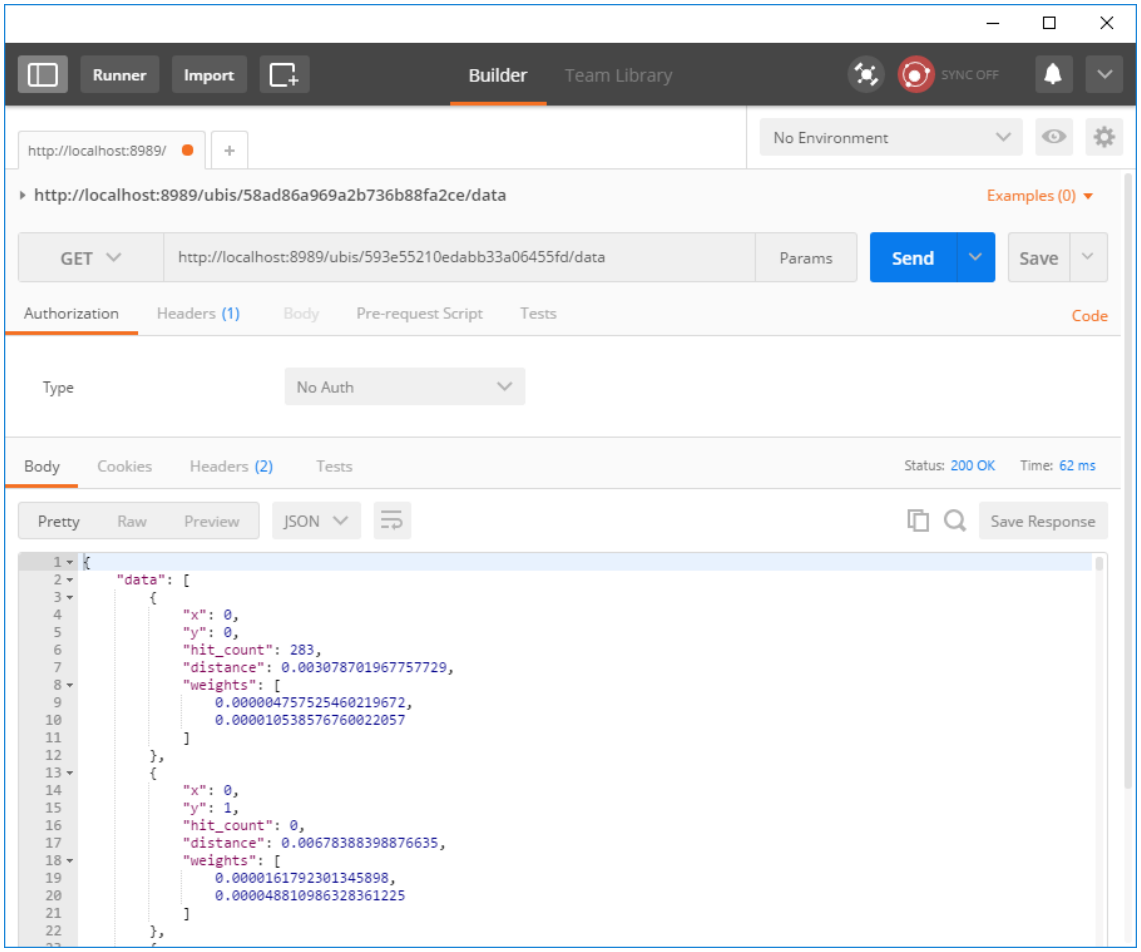


Figure A.13: UbiFactory, Postman – GET data with identifier

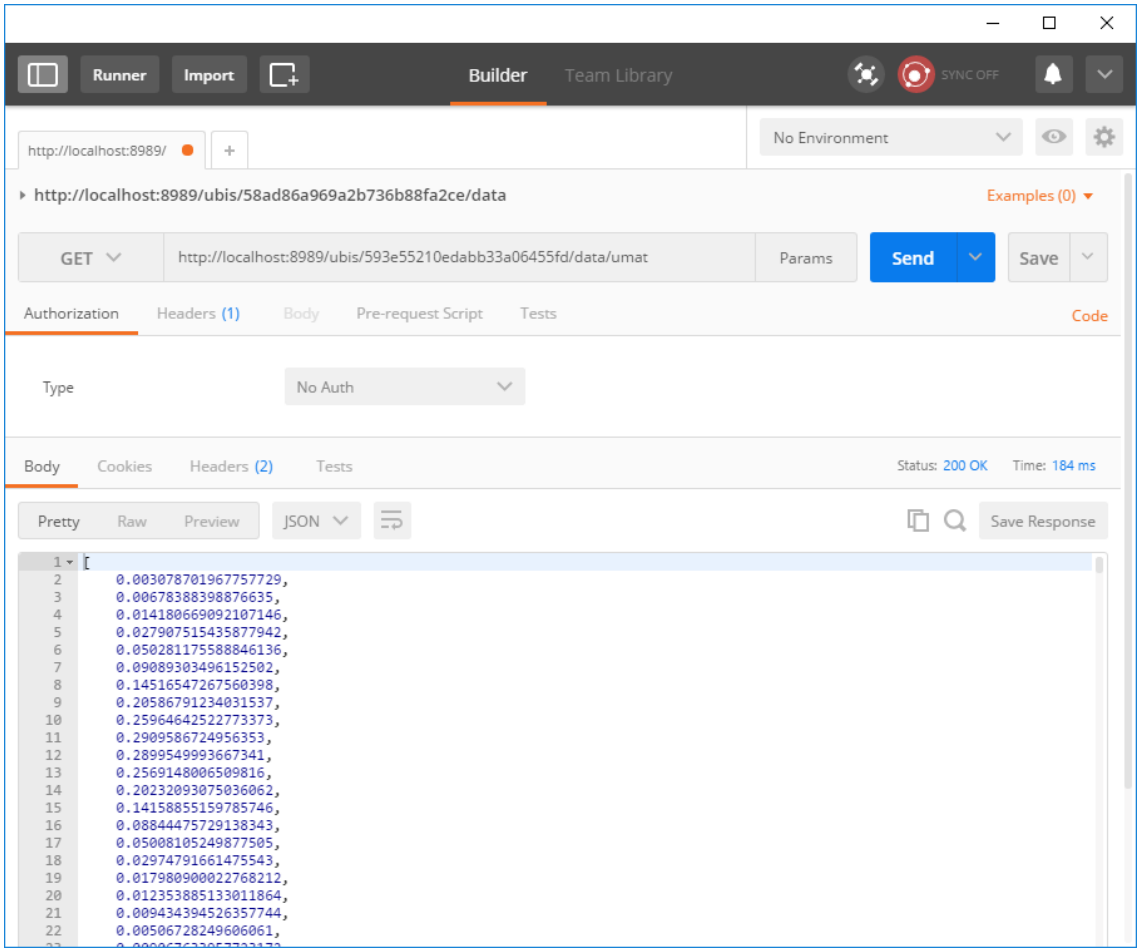


Figure A.14: UbiFactory, Postman – GET U-Matrix with identifier

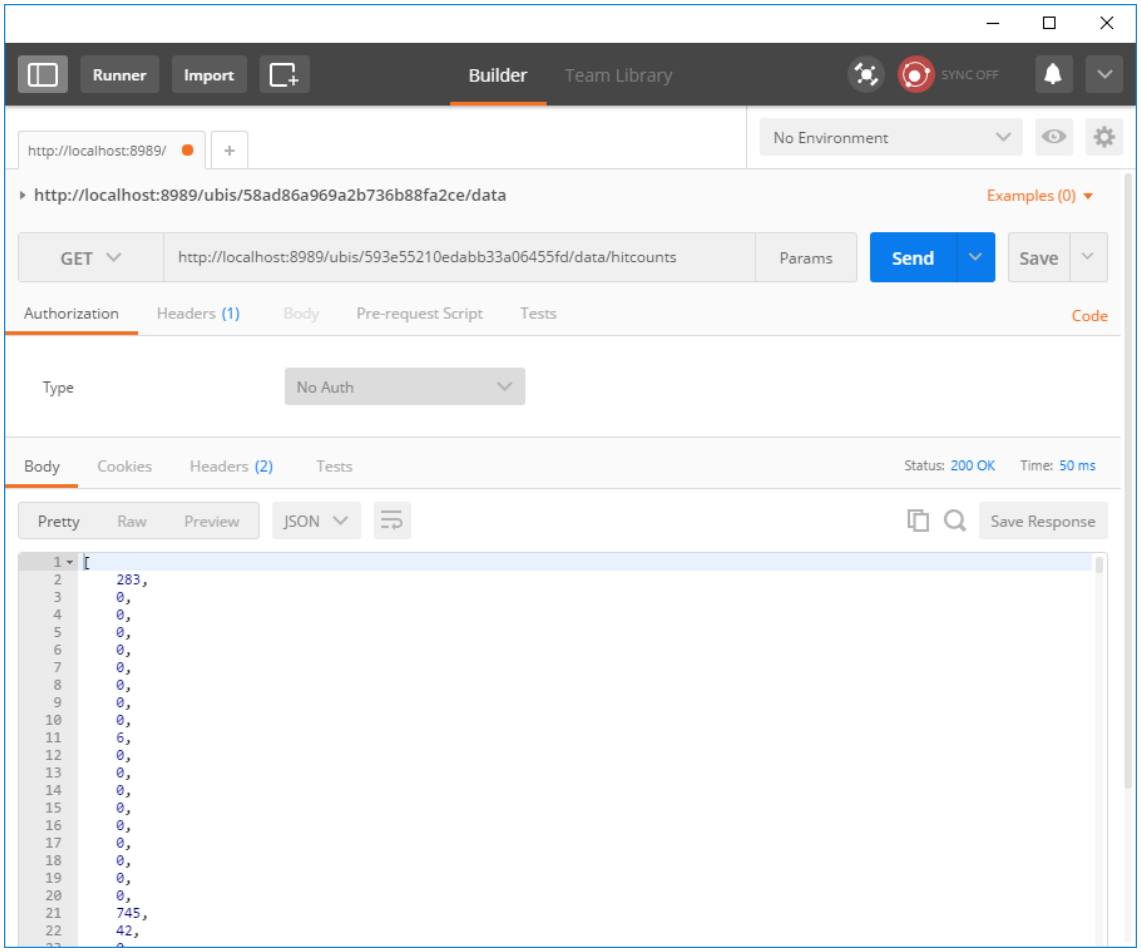


Figure A.15: UbiFactory, Postman – GET hit-count with identifier

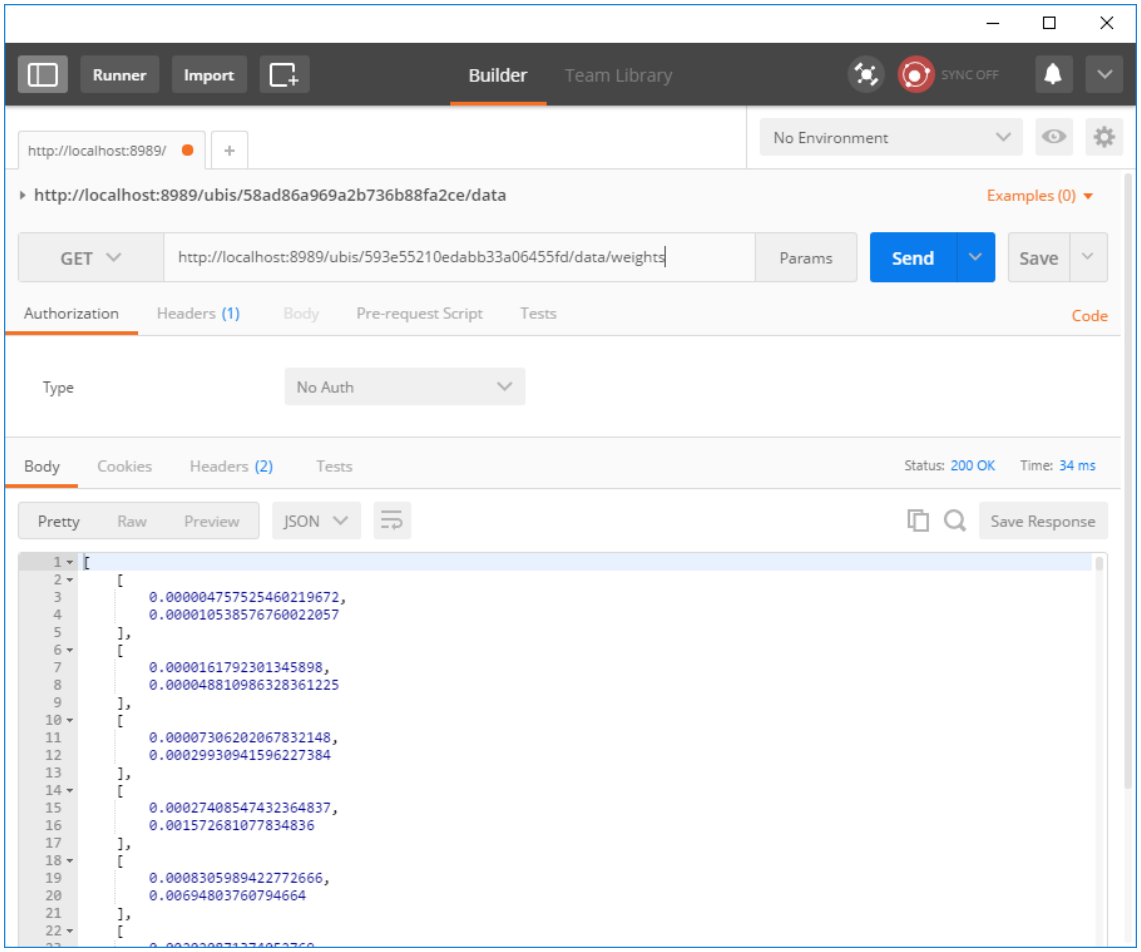


Figure A.16: UbiFactory, Postman – GET prototypes weights with identifier

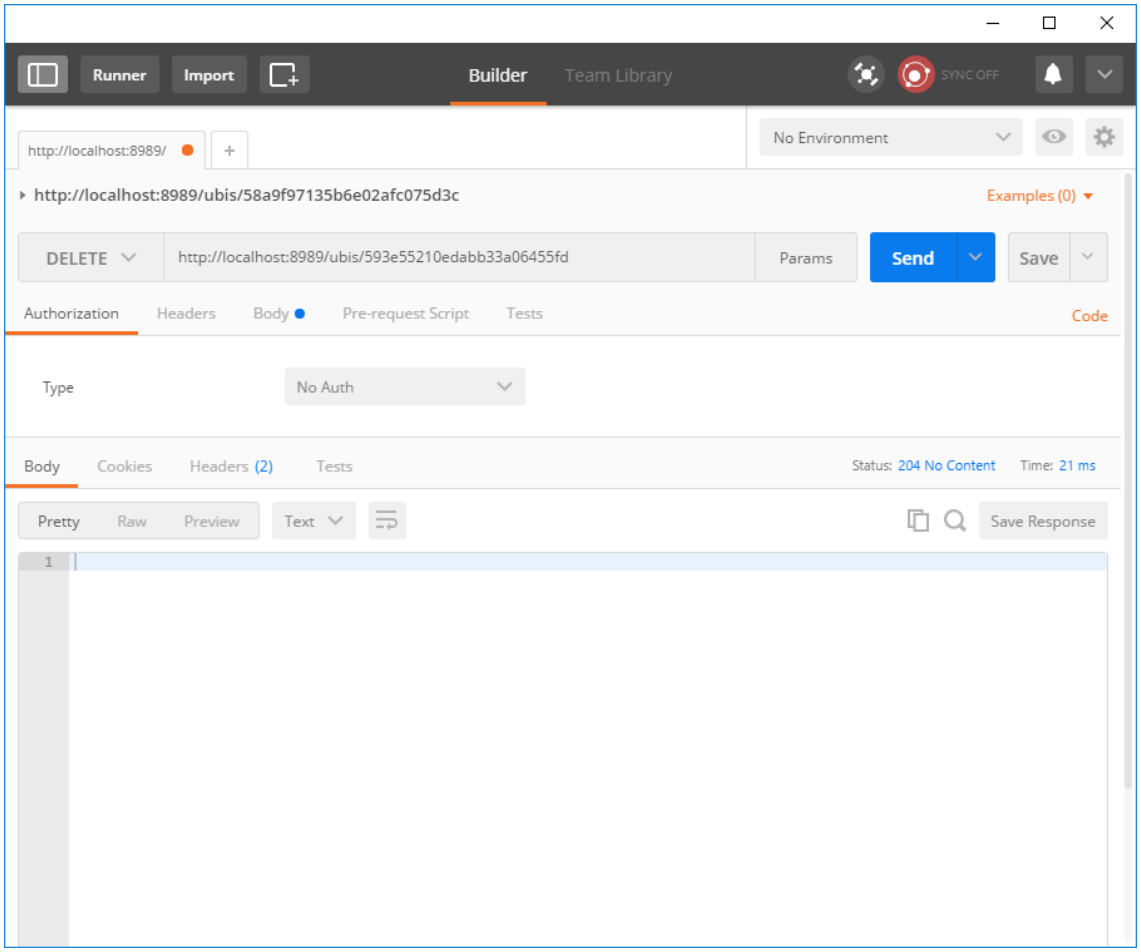


Figure A.17: UbiFactory, Postman – DELETE with identifier

A.3 UbiHSOM

The list of figures bellow show the Postman screens for the different requests:

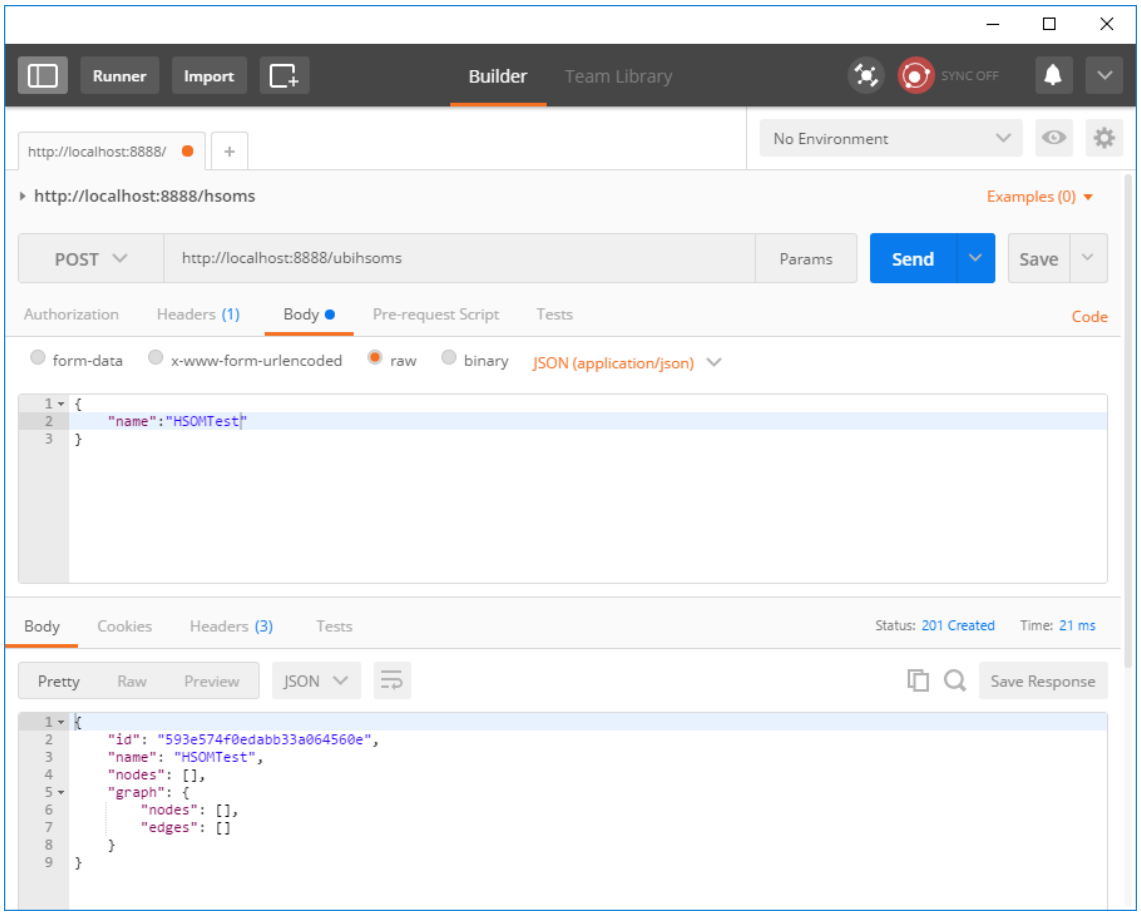


Figure A.18: UbiHSOM, Postman – POST UbiHSOM

The delete screens are very similar to the ones of the previous sections.

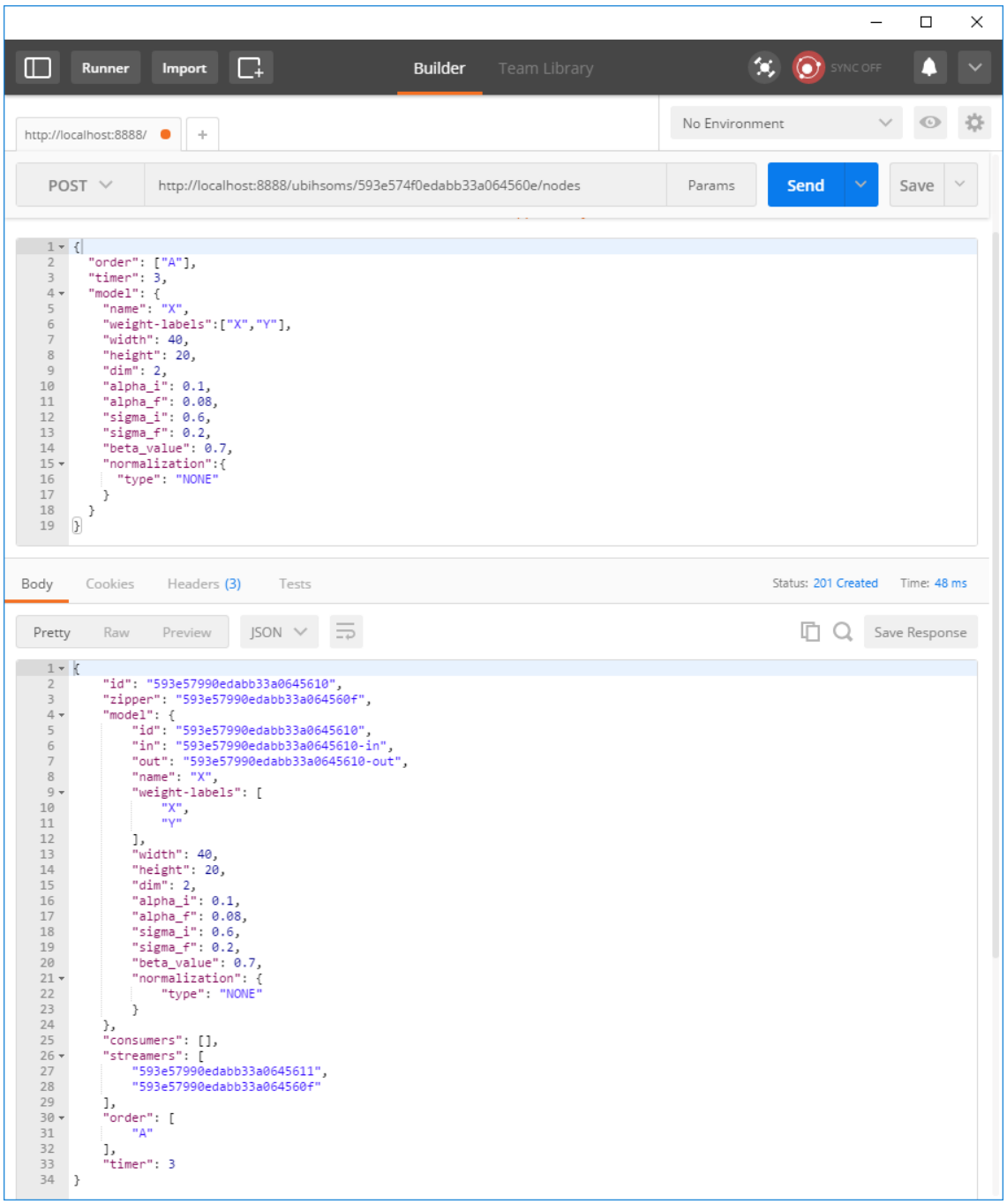


Figure A.19: UbiHSOM, Postman – POST UbiHSOM node

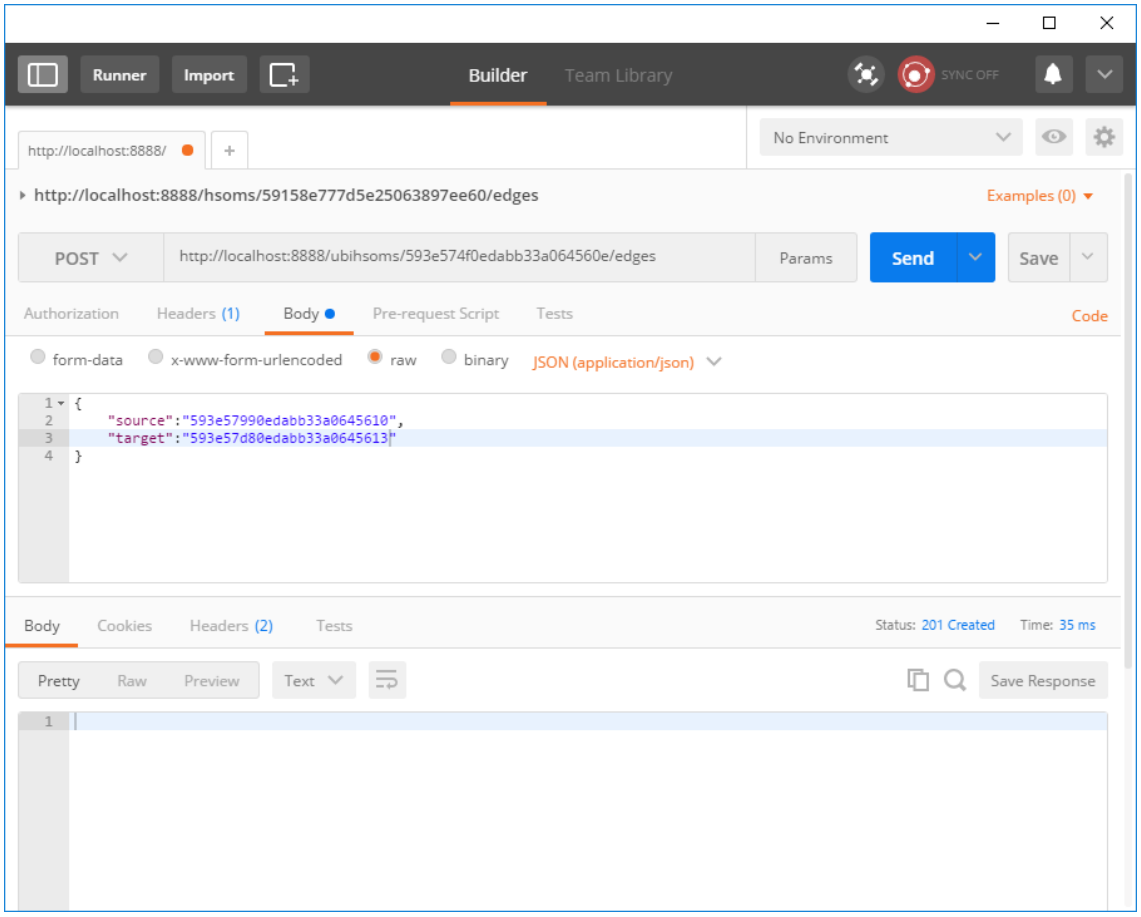


Figure A.20: UbiHSOM, Postman – POST UbiHSOM edge

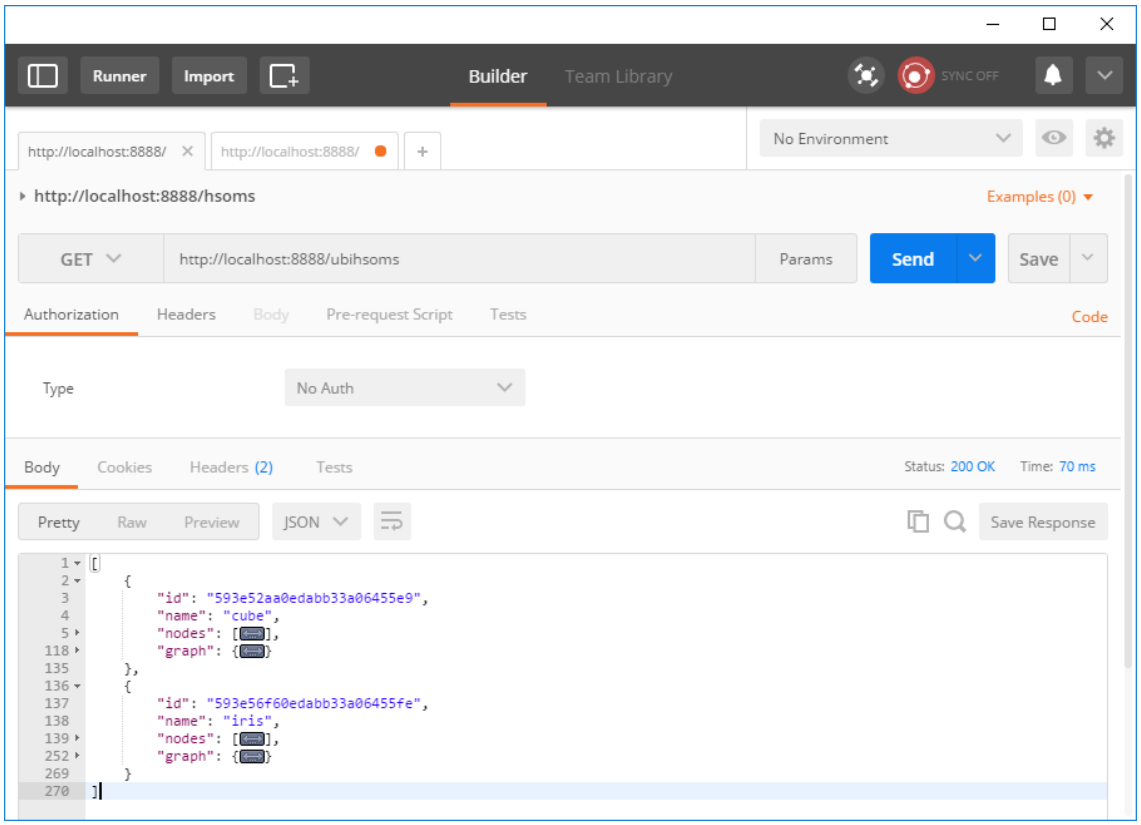


Figure A.21: UbiHSOM, Postman – GET

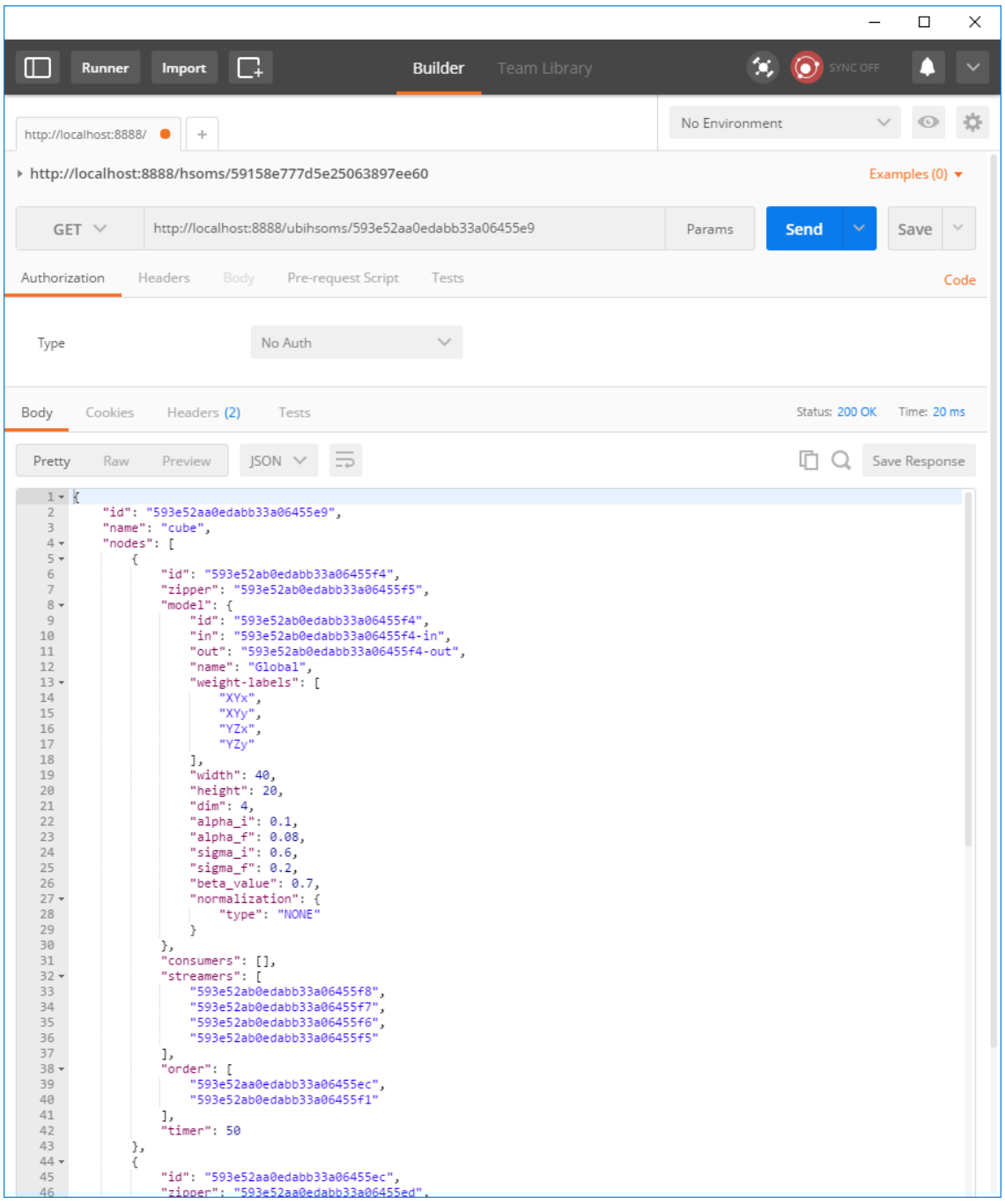


Figure A.22: UbiHSOM, Postman – GET with identifier

Appendix B

Implementation UML

This appendix presents the various UML representations of the services that were developed and described in this thesis.

B.1 DataStreamers

The figures bellow illustrated the various UML diagrams for the DataStreamers service:

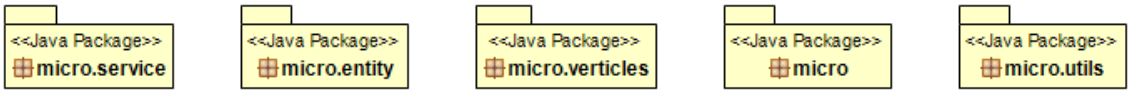


Figure B.1: DataStreamers Package Diagram

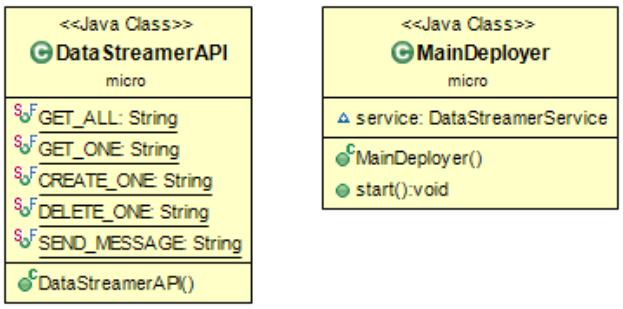


Figure B.2: DataStreamers Root Diagram

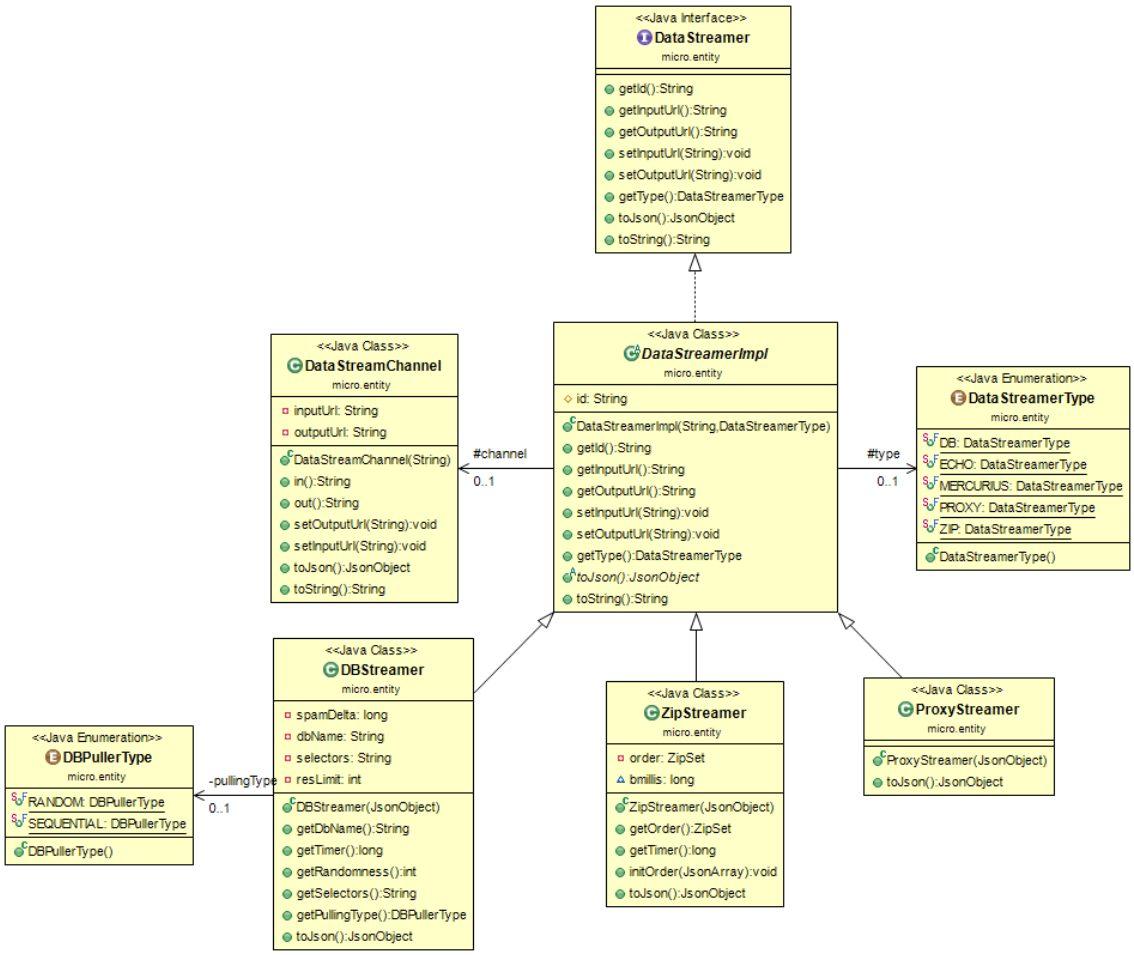


Figure B.3: DataStreamers Entity Package Class Diagram

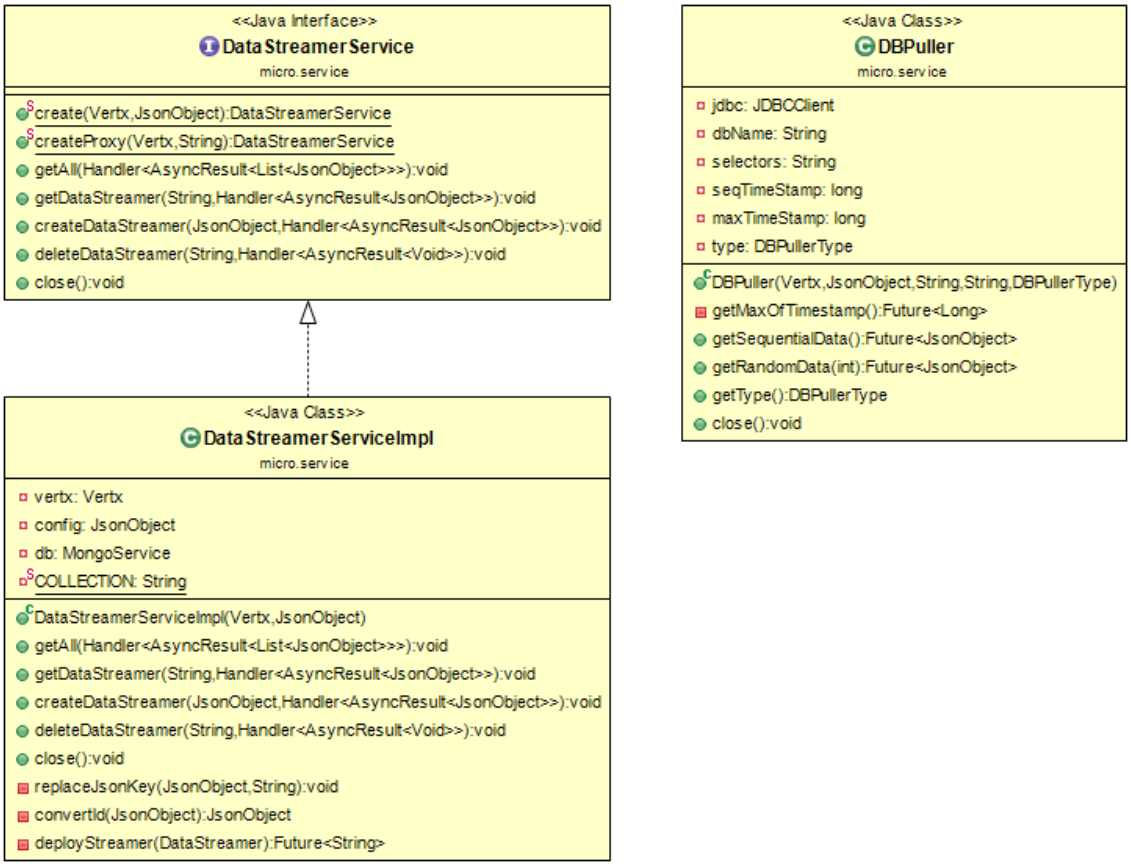


Figure B.4: DataStreamers Service Package Class Diagram

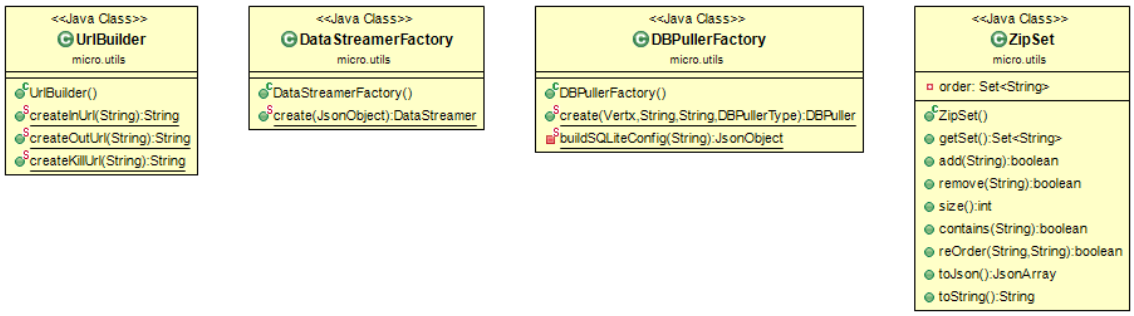


Figure B.5: DataStreamers Utility Package Class Diagram

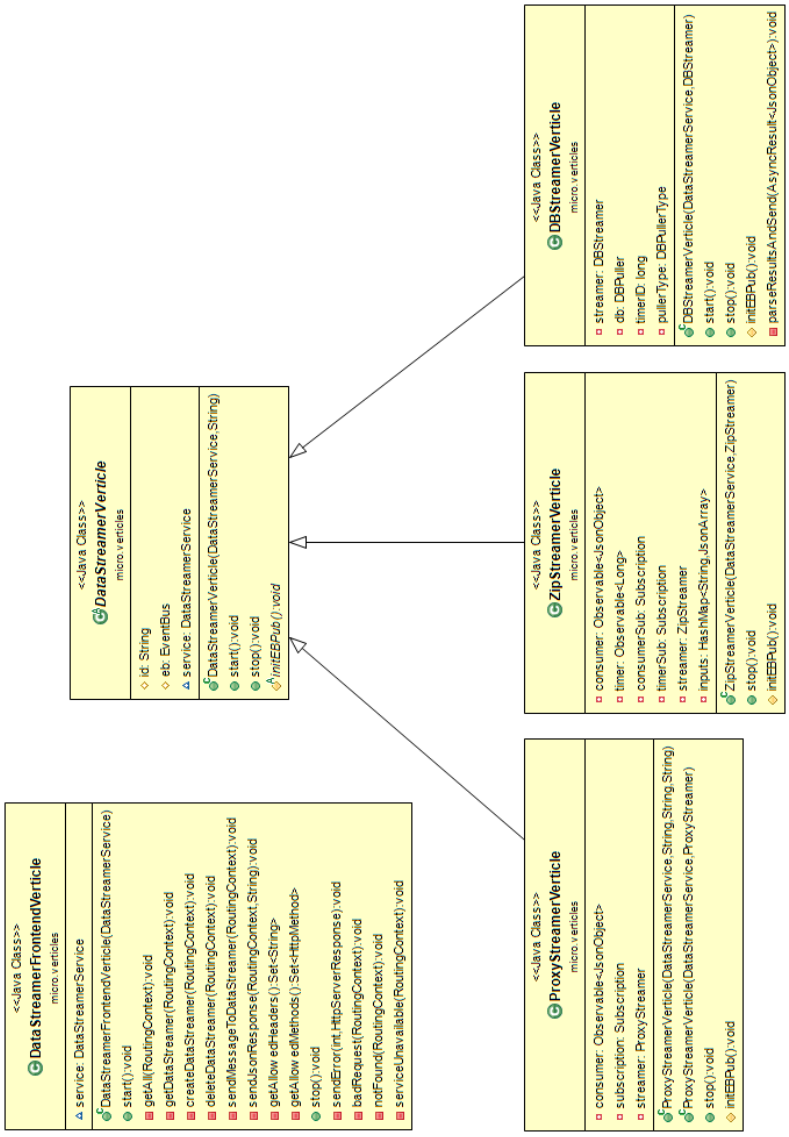


Figure B.6: DataStreamers Verticles Package Class Diagram

B.2 UbiFactory

The figures bellow illustrated the various UML diagrams for the UbiFactory service:

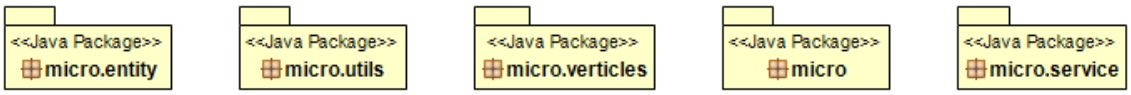


Figure B.7: UbiFactory Package Diagram

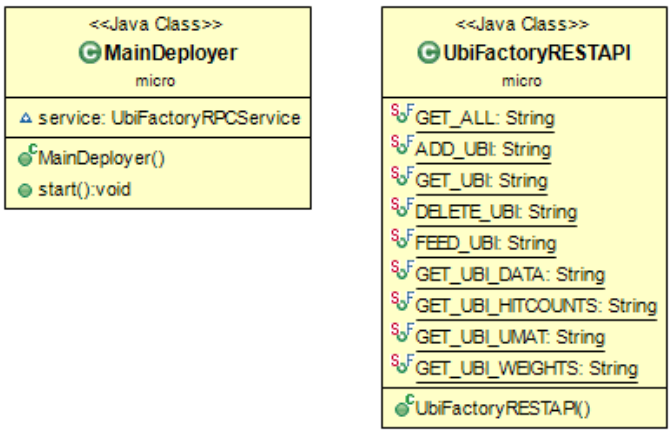


Figure B.8: UbiFactory Root Diagram

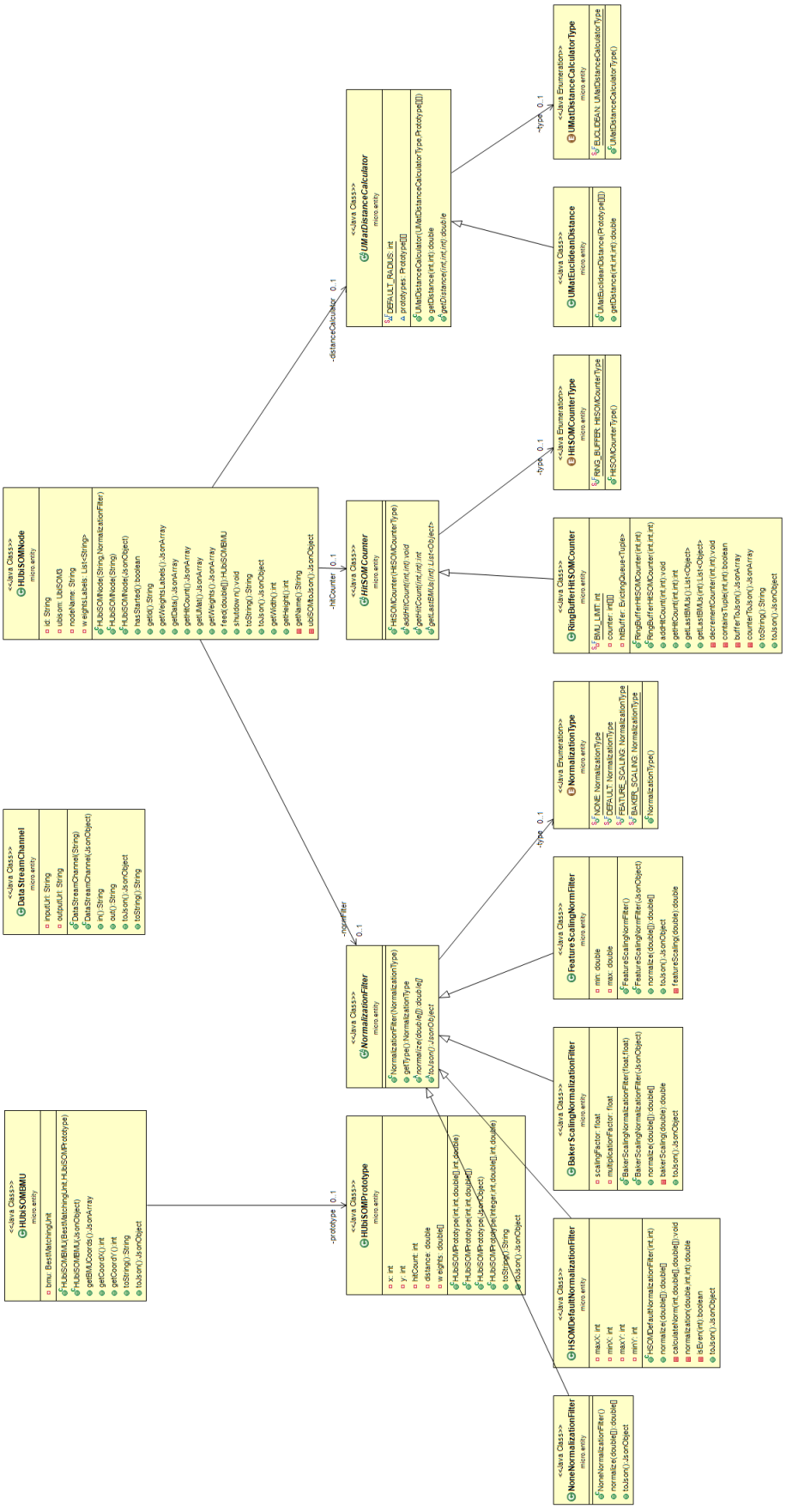


Figure B.9: UbiFactory Entity Package Class Diagram

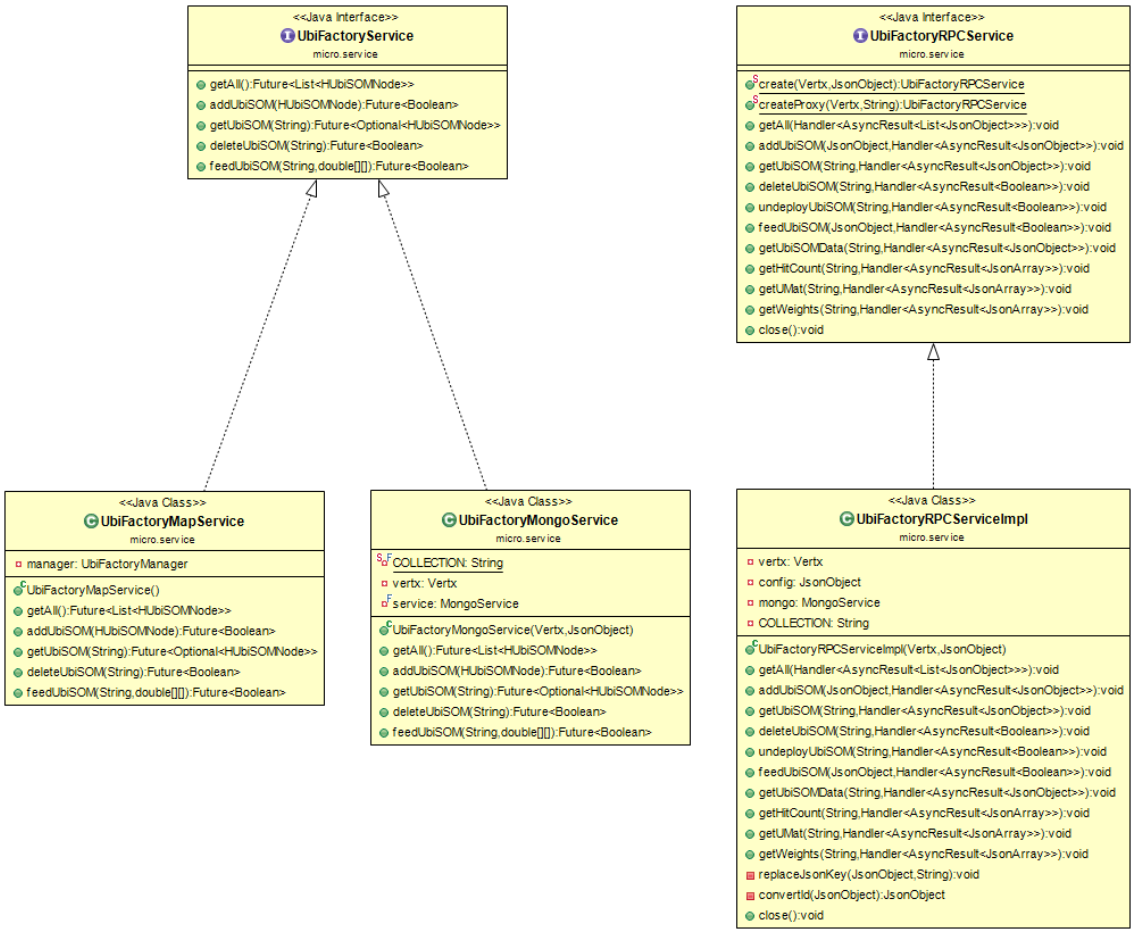


Figure B.10: UbiFactory Service Package Class Diagram

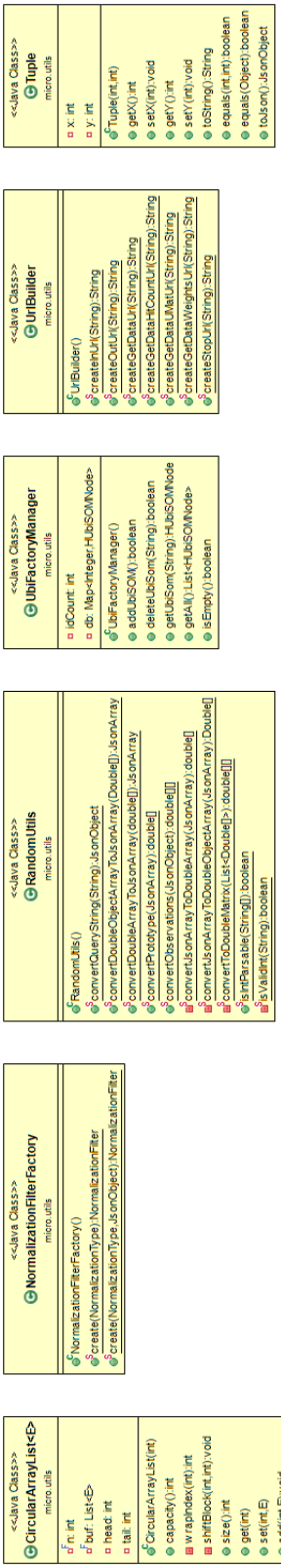


Figure B.11: UbiFactory Utility Package Class Diagram

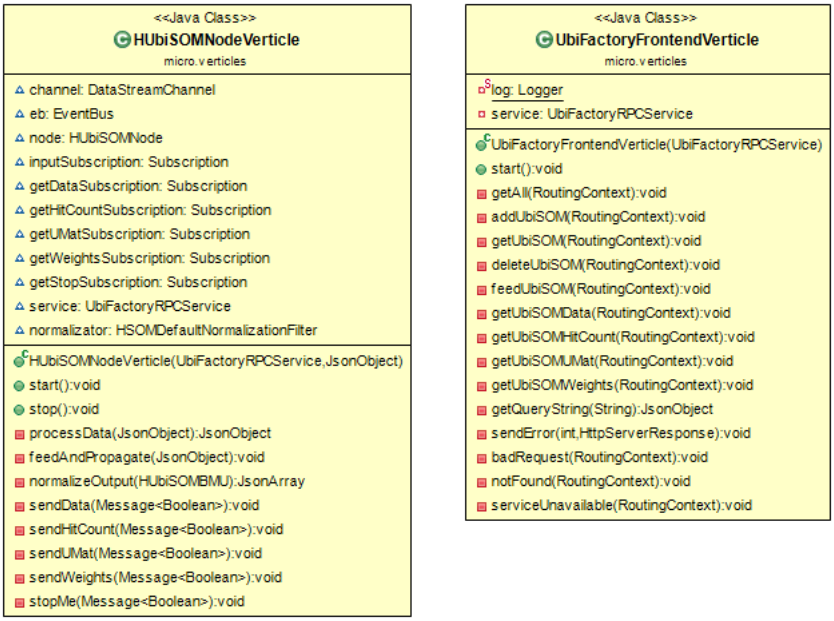


Figure B.12: UbiFactory Verticles Package Class Diagram

B.3 UbiHSOM

The figures bellow illustrated the various UML diagrams for the UbiHSOM service:

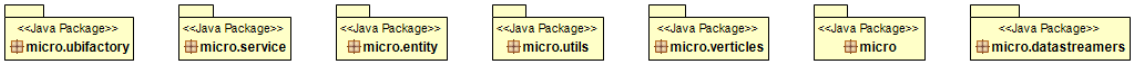


Figure B.13: UbiHSOM Package Diagram

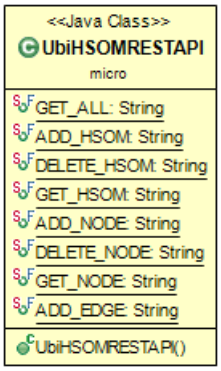


Figure B.14: UbiHSOM Root Diagram

Figure B.15: UbiHSOM DataStreamers Package Diagram

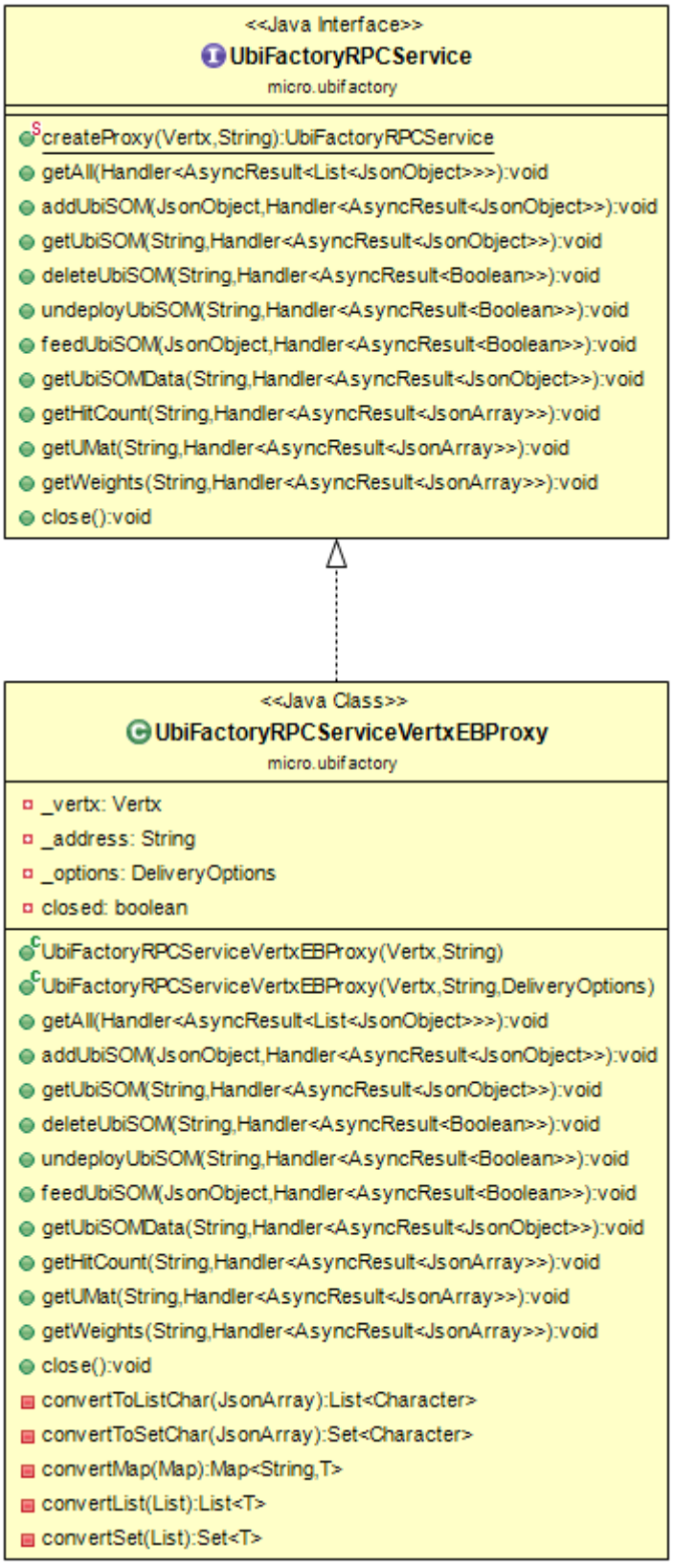


Figure B.16: UbiHSOM UbiFactory Package Diagram

Figure B.17: UbiHSOM Entity Package Class Diagram

Figure B.18: UbiHSOM Service Package Class Diagram

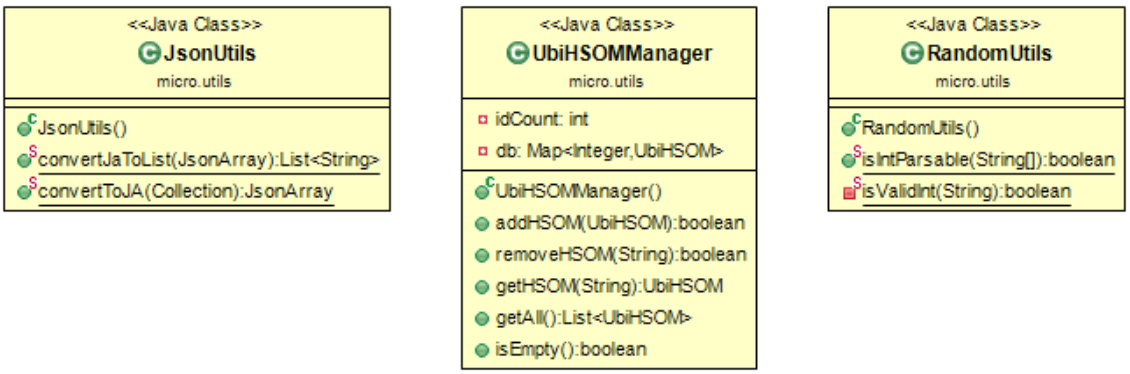


Figure B.19: UbiHSOM Utility Package Class Diagram

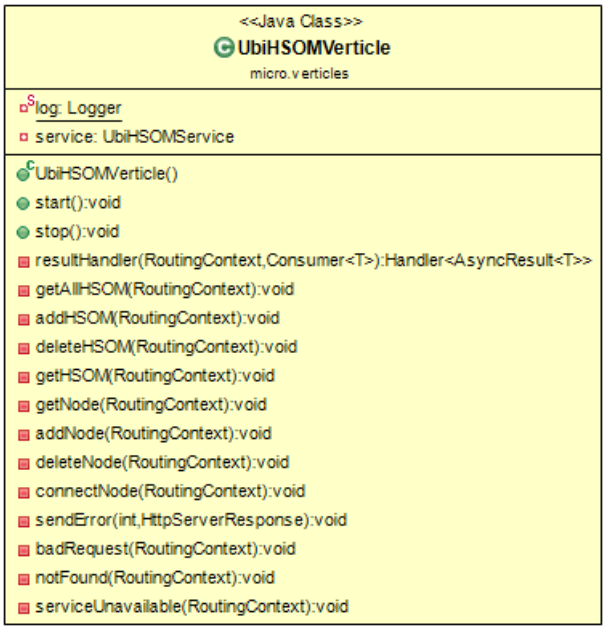


Figure B.20: UbiHSOM Verticles Package Class Diagram

Appendix

Case Study Graphics

The figures bellow illustrated the various graphic for the SOM case study (section 6):

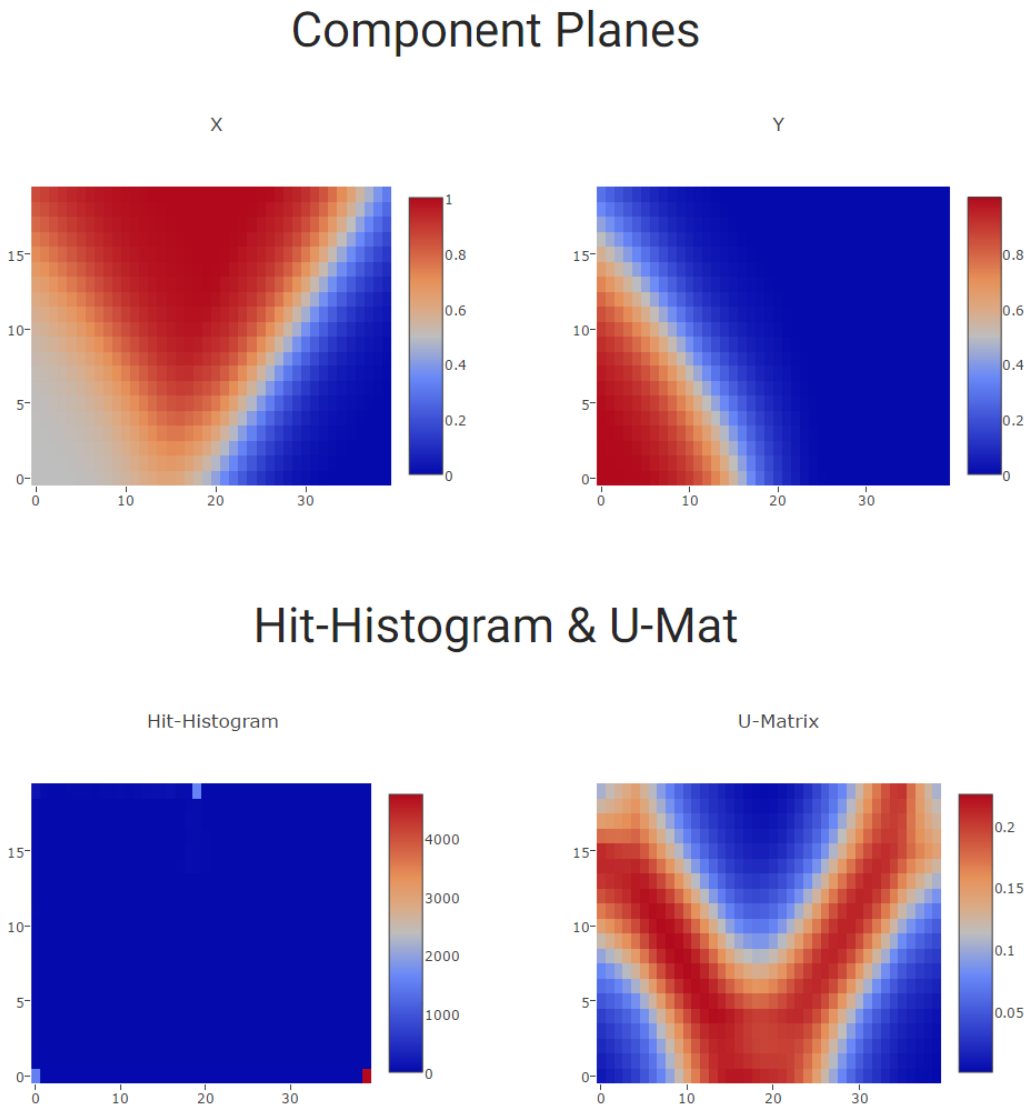


Figure C.1: Isosceles Triangle

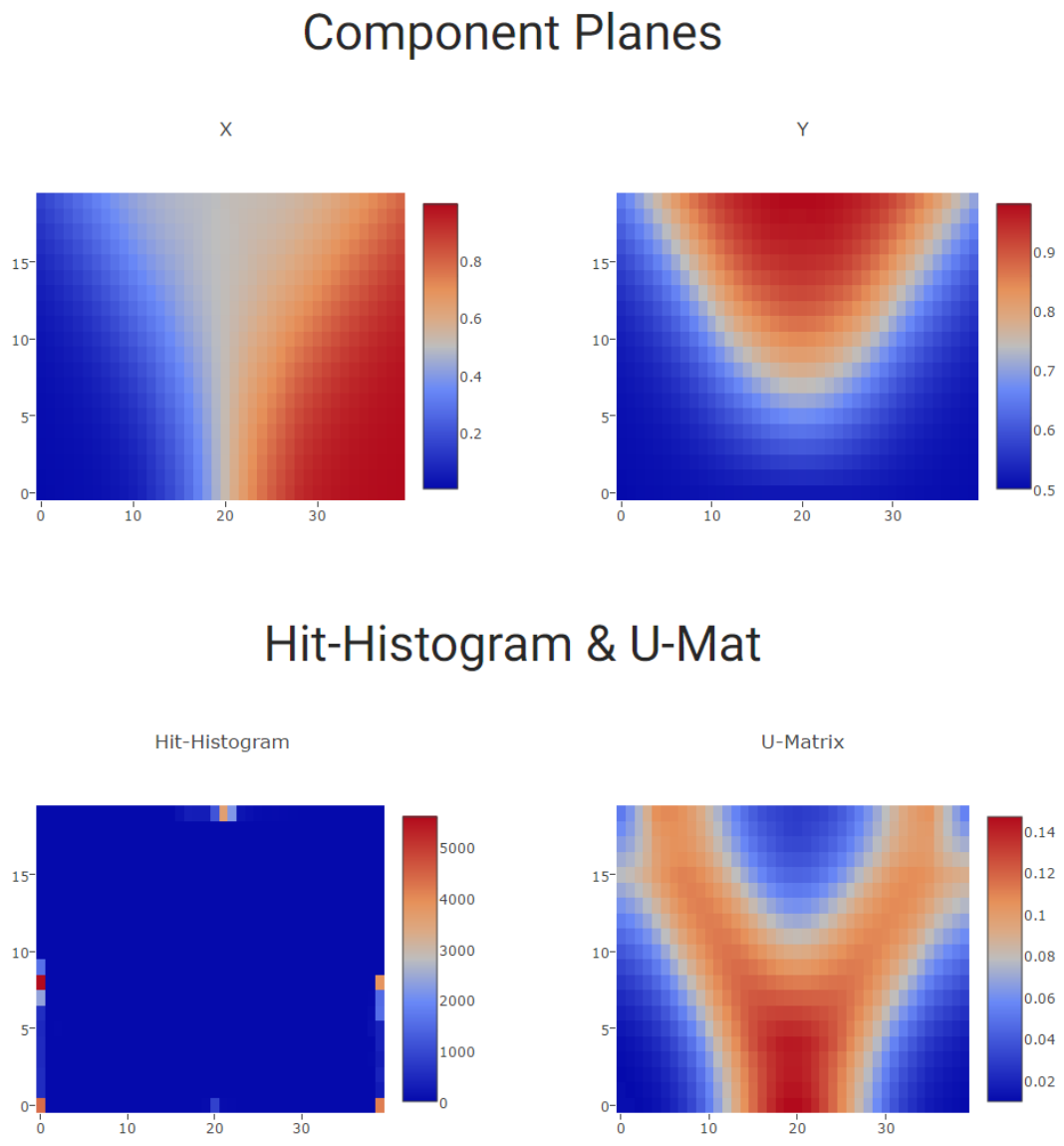


Figure C.2: Diamond

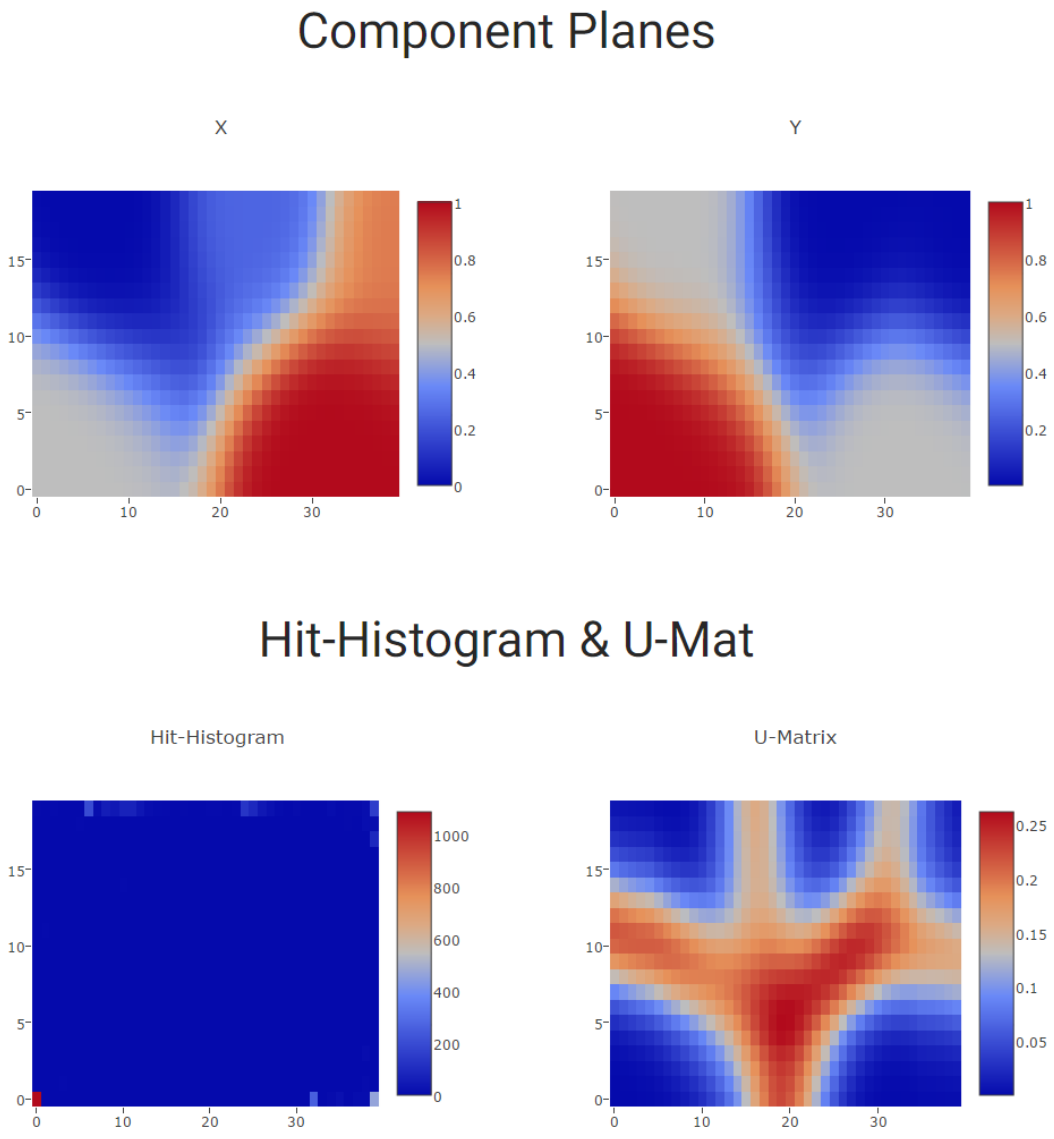


Figure C.3: Pentagon

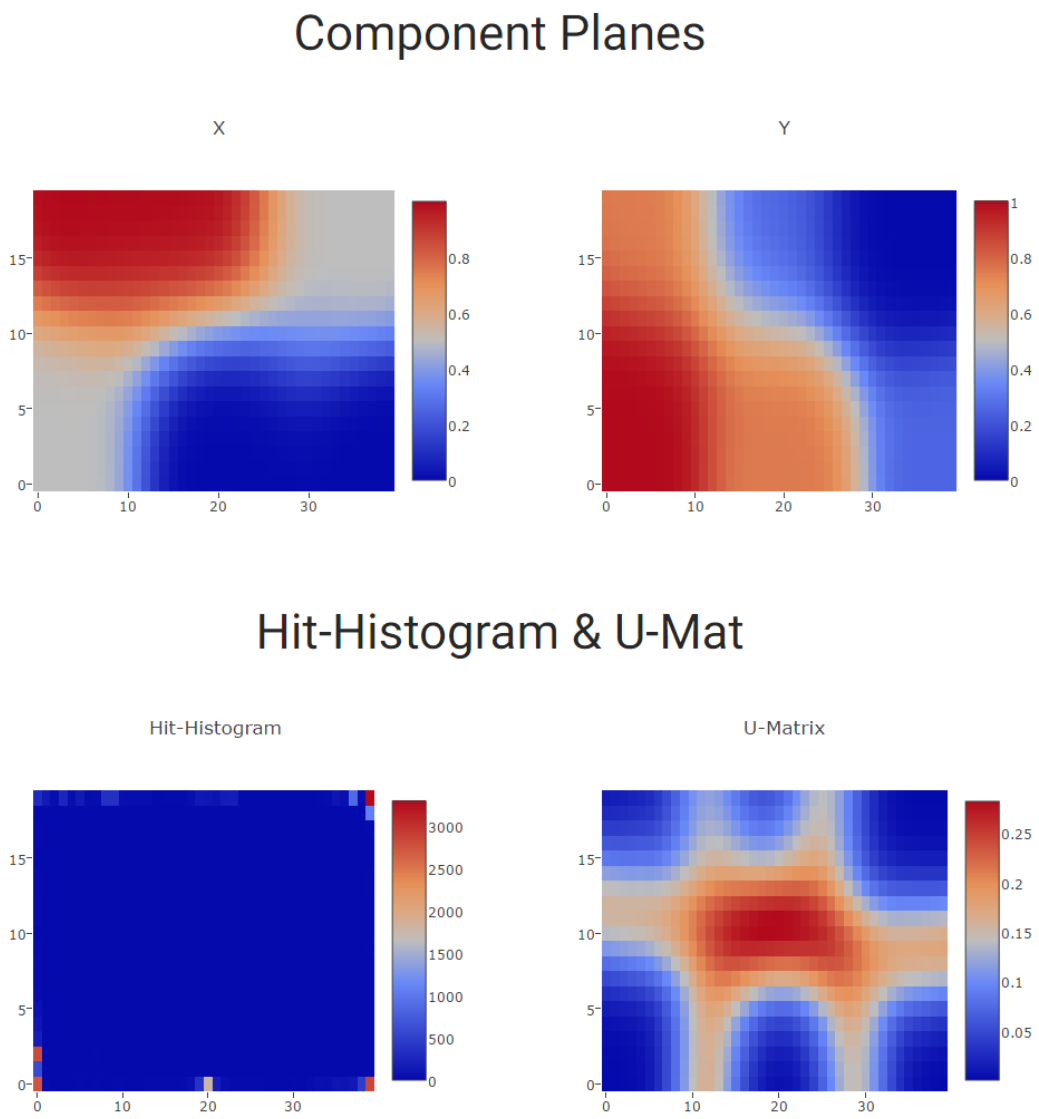


Figure C.4: Hexagon