Extending the Growing Hierarchical Self Organizing Maps for a Large Mixed-Attribute Dataset Using Spark MapReduce

by

Ameya Mohan Malondkar

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the degree in
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
Faculty of Graduate and Postdoctoral Studies
University of Ottawa

© Ameya Mohan Malondkar, Ottawa, Canada, 2015

Abstract

In this thesis work, we propose a Map-Reduce variant of the Growing Hierarchical Self Organizing Map (GHSOM) called *MR-GHSOM*, which is capable of handling mixed attribute datasets of massive size. The Self Organizing Map (SOM) has proved to be a useful unsupervised data analysis algorithm. It projects a high dimensional data onto a lower dimensional grid of neurons. However, the SOM has some limitations owing to its static structure and the incapability to mirror the hierarchical relations in the data. The GHSOM overcomes these shortcomings of the SOM by providing a dynamic structure that adapts its shape according to the input data. It is capable of growing dynamically in terms of the size of the individual neuron layers to represent data at the desired granularity as well as in depth to model the hierarchical relations in the data.

However, the training of the GHSOM requires multiple passes over an input dataset. This makes it difficult to use the GHSOM for massive datasets. In this thesis work, we propose a Map-Reduce variant of the GHSOM called MR-GHSOM, which is capable of processing massive datasets. The MR-GHSOM is implemented using the Apache Spark cluster computing engine and leverages the popular Map-Reduce programming model. This enables us to exploit the usefulness and dynamic capabilities of the GHSOM even for a large dataset.

Moreover, the conventional GHSOM algorithm can handle datasets with numeric attributes only. This is owing to the fact that it relies heavily on the Euclidean space dissimilarity measures of the attribute vectors. The MR-GHSOM further extends the GHSOM to handle mixed attribute - numeric and categorical - datasets. It accomplishes this by adopting the *distance hierarchy* approach of managing mixed attribute datasets.

The proposed MR-GHSOM is thus capable of handling massive datasets containing mixed attributes. To demonstrate the effectiveness of the MR-GHSOM in terms of clustering of mixed attribute datasets, we present the results produced by the MR-GHSOM on some popular datasets. We further train our MR-GHSOM on a Census dataset containing mixed attributes and provide an analysis of the results.

Acknowledgements

I had the great pleasure of working with my supervisors Dr. Iluju Kiringa and Dr. Nathalie Japkowicz, both eminent researchers in their domains. I would like to thank them for accepting me as their student and bringing out the researcher in me. They not only helped me understand the research methodologies, but also provided the useful direction required for my work. Their guidance and support was instrumental in making this thesis work possible.

Further, I would like to thank Dr. Mengchi Liu, Dr. Diana Inkpen and Dr. Tet Yeap for being on my thesis committee and for the insightful feedback.

I would also like to thank Netfore Systems Inc. and Mr. Scott Brookes for providing the infrastructure, technical guidance and motivation for this work.

I would also like to extend my thanks to Kate, Jay, Aman, Meha and Revanth for their support and help in all possible forms over the past two years.

Last but not the least, I wish to devote a special vote of thanks to my Mom, Dad and Jayesh for their unconditional love. I would always be indebted to them for believing in me and my decisions. This work would not have been remotely possible without their support.

Table of Contents

List of Tables vii

List of Figures viii

Nomenclature x

1 Introduction 1

1.1 Motivation and Objective . . . 1

1.2 Contributions . . . 4

1.3 Thesis Organization . . . 5

2 Background 6

2.1 Map-Reduce Framework . . . 6

2.1.1 Map-Reduce Programming Model . . . 6

2.1.2 Apache Spark . . . 8

2.2 Self Organizing Map . . . 11

2.2.1 Theory of the Self Organizing Map . . . 12

2.2.2 Visualizations of SOM . . . 18

2.3 Handling Mixed-attributes . . . 18

2.3.1 Binary Encoding . . . 19

2.3.2 Simple Matching . . . 19

2.3.3 Distance Hierarchy . . . 21

2.4 Conclusion . . . 24

3 Growing Hierarchical Self Organizing Map for Mixed Attributes 26

3.1 Growing Hierarchical Self Organizing Map 26

3.1.1 Architecture of GHSOM 27

3.1.2 Training and Growth Process 28

3.1.3 Effect of τ_1 and τ_2 33

3.2 Faster two-dimensional growth using the *batch growth* 34

3.3 Extending GHSOM for Mixed Attributes 35

3.3.1 Mixed Attribute GHSOM using *Variance* and *Distance Hierarchy* . 37

3.4 Conclusion 46

4 MR-GHSOM 47

4.1 Scope of Parallelism in GHSOM 47

4.1.1 GHSOM Stages for parallelizing on a cluster 48

4.2 MR-GHSOM Algorithms 50

4.2.1 Map-Reduce for computing var_0 51

4.2.2 Training of individual SOM 52

4.2.3 Evaluating the quality of SOM for expansion 54

4.2.4 Hierarchical Expansion 55

4.2.5 Complete MR-GHSOM 57

4.3 Conclusion 59

5 Experiments 60

5.1 Configuration and precursor information 60

5.2 Evaluation Approach 61

5.3 Evaluation of two-dimensional Growth 62

5.3.1 Focus of Evaluation 62

5.3.2 Evaluation 63

5.3.3 Learnings from the evaluation 69

5.4 Evaluation of the *batch growth* 69

5.4.1 Focus of Evaluation . . . 69
5.4.2 Evaluation . . . 70
5.4.3 Learnings from the evaluation . . . 71
5.5 Evaluation of Hierarchical Growth . . . 71
5.5.1 Focus of Evaluation . . . 71
5.5.2 Evaluation . . . 72
5.5.3 Learnings from the evaluation . . . 73
5.6 Experiments on the Census Dataset . . . 74
5.6.1 Focus of Evaluation . . . 74
5.6.2 Evaluation . . . 75
5.6.3 Learnings from the evaluation . . . 80
5.7 Conclusion . . . 80

6 Conclusion and Future Work **82**
6.1 Conclusion . . . 82
6.2 Directions for Future Work . . . 84

References **85**

List of Tables

5.1 Datasets for two-dimensional growth evaluation 62

5.2 Topographic error: MR-GHSOM versus sequential SOM 68

List of Figures

2.1 Map-Reduce Programming Model 7

2.2 Apache Spark Architecture 9

2.3 Adaptation of neurons to input instance 14

2.4 Distance Hierarchy for Mixed attributes 22

2.5 Adaption of Neuron M to instance X 24

3.1 Typical structure of a trained GHSOM 27

3.2 Adding new row or column between *error neuron* and *dissimilar neighbour neuron* 30

3.3 Initialization of neurons in new map[5] 32

3.4 Distance hierarchy for the mixed attribute GHSOM 39

3.5 Addition of points in distance hierarchy 43

5.1 Results of MR-GHSOM on the Iris dataset 63

5.2 Results of the sequential SOM on the Iris dataset 64

5.3 Results of MR-GHSOM on the Wine dataset 65

5.4 Results of the sequential SOM on the Wine dataset 65

5.5 Results of MR-GHSOM on the Mushroom dataset 66

5.6 Results of the sequential SOM on the Mushroom dataset 66

5.7 Results of MR-GHSOM on the Credit dataset 67

5.8 Results of the sequential SOM on the Credit dataset 68

5.9 Evaluation of the *batch growth* approach 70

5.10 Hierarchical SOMs for zoo dataset 73

5.11 U-Matrix and Component Planes for Level 1 SOM of the Census Dataset . 76

5.12 U-Matrix and Component Planes for Level 2 SOM of the Census Dataset . 78

Nomenclature

GHSOM	Growing Hierarchical Self Organizing Map
MR-GHSOM	Map-Reduce variant of the Growing Hierarchical Self Organizing Map
RDD	Resilient Distributed Datasets
SOM	Self Organizing Map
HDFS	Hadoop Distributed File System
mqe_k	Mean Quantization Error of a neuron k
MQE_m	Mean Quantization Error of map m
MV_m	Mean Variance of map m
var_k	Variance of a neuron k

Chapter 1

Introduction

This thesis work proposes a Map-Reduce variant of the Growing Hierarchical Self Organizing Map (GHSOM) called MR-GHSOM. The proposed algorithm is built on the Map-Reduce framework of Apache Spark making it scalable to handle large datasets. Further, we extend the algorithm to handle mixed attribute datasets containing both numeric and categorical attributes, which are prevalent in most real-world datasets.

1.1 Motivation and Objective

With the advent of technologies for capturing and storing digital information, the amount of data accumulating in enterprise data centres is growing exponentially. This data holds a lot of latent information which can be mined to extract knowledge from it. Data mining is a process of discovering such interesting patterns and knowledge from massive amounts of data [17]. *Clustering* is one such method of data mining. It is used to make sense of the data, especially when characteristics of the underlying dataset are unknown. It partitions the instances of the dataset into clusters, such that the instances within a cluster have a high degree of similarity in comparison to the instances belonging to other clusters. Cluster analysis can help one understand the distribution of an underlying dataset, observe the characteristics of the clusters and moreover, narrow the focus to a particular set of cluters. It can also be used as a preprocessing step for classification and attribute subset selection problems of data mining [17]. Clustering has been used in a variety of applications such as information retrieval [32], image segmentation [8, 36], bioinformatics [2], document clustering & retrieval [50] and marketing research [37]. In marketing, the primary use of clustering is market segmentation [37], which is identifying the groups of entities that share common

characteristics in terms of demographics, attitudes and behaviour. In this thesis work, we shall be performing a similar analysis of a census dataset. This analysis can be used in marketing by an enterprise to understand the characteristics of their customer population. Similarly, it can be used in the formulation of strategies for an election campaign where campaign officers need to devise their campaigns and strategies to persuade voters to vote in favour of their political parties.

One of the crucial requirements of data mining algorithms is the need to handle very large datasets (gigabyte, terabytes or even petabytes). Moreover, the data instances usually have a high-dimensionality which further complicates the analysis of the data, especially with regards to visualization. Further, the attributes of most real world datasets are of mixed types - numeric and categorical. Thus, the usual Euclidean space computations do not apply to such datasets. Most conventional data mining algorithms are formulated considering only numeric attributes and hence there is a need to extend them for non-numeric or categorical attributes. These properties of real-world data make the problem of data analysis further complex. The problems concerning the scalability of clustering algorithms, the handling of high-dimensional data and the modelling of mixed attribute dataset, have been of interest in the research community for quite some time now.

The Self Organizing Map (SOM) [25] is a clustering algorithm based on the neural network model. It maps a high-dimensional input data onto a 2-dimensional (or 3-dimensional) space, also called a feature map, which is typically a grid of neurons. An important advantage of using the SOM is that, it preserves the topology of an underlying dataset. The generated feature map can be used to understand the structure of the underlying dataset as well as to find clusters within the data. The SOM is also capable of handling large datasets and scales linearly with the number of input data samples [48]. However, in the conventional SOM, the shape and the number of neurons in the feature map needs to be predetermined and provided before the training of the SOM begins. The final structure and dimensions of the feature map are usually determined by the trial-and-error process. This becomes cumbersome and difficult to determine when nothing is known about the underlying dataset and the size of the dataset is massive. Also, hierarchical relations within the input dataset cannot be identified in the conventional SOM, as they are shown in the same feature space.

To overcome these limitations of the SOM, a variant of the SOM called the Growing Hierarchical Self Organizing Map (GHSOM) was proposed [10]. The GHSOM has a multi-level tree-like architecture formed of individual SOMs arranged in a hierarchy. Each SOM layer adapts its size and structure to the input data incident on it. Moreover, the hier-

archical structure of the GHSOM provides a zooming capability - starting from the first map representing the complete data, one can zoom into the next hierarchical layer to see the finer granularity of the dataset. Thus, the GHSOM has a capability to grow in two-dimensions to adapt to the underlying dataset as well as expand in depth to provide the details at a finer granularity. The GHSOM provides the necessary dynamic capability to the SOM. It thus proves to be a good candidate for dealing with a large dataset especially when nothing is known about it and its distribution.

The training of the GHSOM involves multiple passes over an input dataset. This will be difficult for a sequential algorithm on a single machine when dealing with a massive dataset. On a single machine, this would involve reading the massive dataset from the disk multiple times since the dataset cannot fit in memory, thus hampering the performance of the algorithm. Like other clustering algorithms [53, 44, 39, 31, 41], there is a need for leveraging the dynamic capabilities of the GHSOM for processing massive datasets by porting the GHSOM algorithm to run in a distributed environment on a cluster of nodes.

Map-Reduce [9] has emerged as a popular choice for distributed parallel processing of massive datasets. The Map-Reduce framework simplifies writing programs in a distributed computing environment. The tasks of parallelization, fault tolerance, data distribution and load balancing are managed by the framework, enabling developers to concentrate on the business logic. A developer only needs to specify a *map* and a *reduce* function. The *map* function or the *mapper* task performs the required computations and/or transformations on an input data record and outputs an intermediate set of *key-value* pairs. The *mapper* tasks execute in parallel on the nodes in the cluster called the *mapper* nodes. The *reduce* function or *reducer* task receives a list of all values associated with a key from all the *mapper* tasks and combines these values to compute an output. This output could be the final result or the input to the next cycle of *map-reduce*. Similar to the *mapper* tasks, the *reducer* tasks are also executed in parallel on the reducer nodes.

Apache Spark [51] is a relatively new entrant in this field of distributed data processing of large datasets which supports the Map-Reduce programming model. Spark can access data from distributed data sources such as Hadoop Distributed File System (HDFS), Apache Cassandra and Apache HBase, to name a few. Apache Spark provides a faster parallel data processing than Hadoop in case of applications where multiple iterations are required over the same dataset [51]. In [51], results show that Spark outperforms Hadoop by a factor of 10x for the class of applications involving iterative algorithms. Since, the GHSOM also falls in this class of applications, we decided to use Apache Spark as our underlying Map-Reduce engine to come up with a variant capable of processing massive

datasets.

The conventional SOM and GHSOM can process dataset involving numeric attributes only. The mathematical operations on attribute vectors are performed assuming a Euclidean space. However, most real world datasets contain non-numeric or categorical attributes alongside numeric ones. To handle such mixed attribute datasets, we adopted an approach called *distance hierarchy* presented in [19]. The approach provides a uniform method for handling both numeric and categorical components of a mixed attribute dataset. Using the *distance hierarchy* technique, the proposed MR-GHSOM is capable of processing high-dimensional datasets containing mixed attributes.

1.2 Contributions

This thesis work proposes a Map-Reduce variant of the GHSOM algorithm called MR-GHSOM, which is capable of processing massive high-dimensional mixed-attribute datasets in a distributed environment setting. The algorithm, as named, is implemented leveraging the Map-Reduce programming model. As the GHSOM, like other data mining algorithms, requires multiple iterations over an input dataset, we used the Apache Spark platform for implementing our algorithm. Apache Spark, using an abstraction called the Resilient Distributed Datasets (RDD) and caching, facilitates better performance of machine learning jobs requiring multiple iterations over a dataset. Further, we also show how we can use the *distance hierarchy* technique to extend the capability of the MR-GHSOM to handle datasets containing mixed attributes, which is a common characteristic of most real world datasets. We also propose a *batch growth* approach for speeding up the two-dimensional growth process of the conventional GHSOM. We analyze our algorithm with respect to clustering by running it on some popular classification datasets from UCI [29]. Lastly, we use the Census-Income dataset from UCI repository [29] as our test bed and perform cluster analysis on it.

In summary, the contributions of this work are as follows:

- Propose *MR-GHSOM* - a Map-Reduce based algorithm for the GHSOM using Apache Spark
- Extend the conventional GHSOM algorithm to handle high-dimensional mixed attribute - numerical and categorical - datasets using the *distance hierarchy* based approach

- Propose a modification to speed up the two-dimensional growth process of the GHSOM

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 provides the necessary background about the Map-Reduce programming model, SOM and the current trends in handling mixed attribute datasets in SOM. We provide the theory behind GHSOM in Chapter 3 and introduce the technique to speed up the two-dimensional growth process. We also describe the process of extending GHSOM to handle mixed attributes using the *distance hierarchy* approach. Chapter 4 presents the algorithms of our Map-Reduce variant of GHSOM formally. In Chapter 5, we provide the results of the experiments conducted using our proposed MR-GHSOM. Finally, we provide the summary of the results and discuss the possible future work in Chapter 6.

Chapter 2

Background

In this chapter, we will present a description of the Map-Reduce programming model for processing massive datasets. Further, we shall provide some theoretical understanding of the SOM and discuss some of its variants. We shall also look at the current trends in handling of mixed attributes for SOMs.

2.1 Map-Reduce Framework

2.1.1 Map-Reduce Programming Model

The Map-Reduce Programming Model [9] was introduced in 2004 and has been a popular approach for processing of large datasets ever since. The programming model reduces the complexity of developing programs for a distributed computing environment. In Map-Reduce, the complications of scheduling & parallelizing tasks, managing node/task failures, and handling inter-node communications, are taken care of by the framework, enabling the developer to concentrate on the business logic of the task at hand. A programmer needs to provide a *map* and a *reduce* function only. The *map* function (also called Mapper task) runs on the mapper nodes of the cluster. It contains computations to be performed on each record of the dataset and emits an intermediate set of *key-value* pairs. The framework then shuffles and partitions the output from the mapper nodes and sends it to the reducer nodes running *reduce* functions (also called Reducer tasks). The partitioning is done in such a way that all the values associated with a key are sent to a particular reducer node. In other words, each reducer node is assigned an exclusive set of keys from the domain of all keys generated by the mapper nodes. The reducer nodes executing the reducer task

iterates through all values associated with a key and aggregates them to produce a result corresponding to the key. All input data instances are processed in parallel on the mapper nodes and all keys and the associated values are processed in parallel on the reducer nodes. A typical Map-Reduce operation is shown in Figure 2.1. A typical Map-Reduce job is

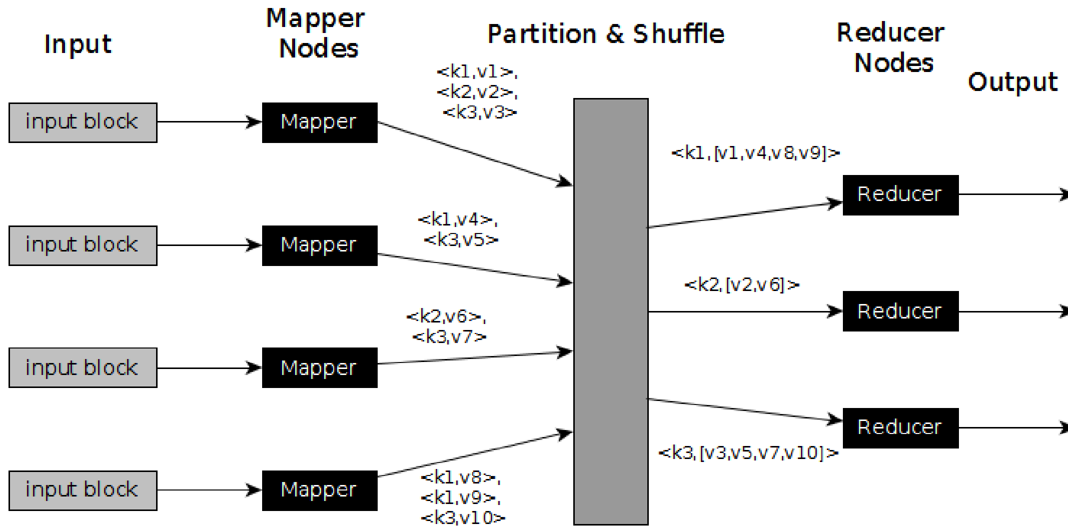


Figure 2.1: Map-Reduce Programming Model

composed of following stages:

- *Mapper*:
  A mapper task performs the required computations or transformations on each input record. The mapper tasks emit a *key-value* pair or a set of *key-value* pairs for each input record.
- *Partition, Shuffle and Sort*:
  The *key-value* pairs from the mappers are required to be sent to the reducers. The shuffling process takes care of this. It transmits the output from the mappers to the reducers. However, each reducer is expected to process a unique subset of all the keys produced by all mapper tasks. Thus, the *key-value* pairs need to be partitioned based on the keys and then sent to the reducers. This partitioning is done by the Partitioner process. The Map-Reduce library sorts the *key-value* pairs by keys so that all the values associated with a key are grouped together.
- *Reducer*:
  Each reducer task is responsible for a subset of keys and the values associated with them. The input to the *reduce* function is one key at a time and a list of values associated with the key from all mapper nodes. The reducer iterates over all the

values and processes them (usually an aggregate operation) to compute an output for the key. It then processes the next key and its associated values.

- *Combiner*:
  This is an optional optimization process but a very useful one to improve the performance of a Map-Reduce job. The mapper produces a set of *key-value* pairs which are transmitted to the reducer nodes. In a normal scenario without the combiner, the total number of such pairs transmitted from the mapper to the reducer is $k \times n$, where k is the number of keys/neurons and n is the number of input instances (assuming one *key-value* pair per input instance). A combiner behaves like a reduce task but executes locally on the mapper node. It performs a reduce-like operation for all *key-value* pairs generated at that mapper node. It combines or groups the *key-value* pairs by keys so that the number of records transmitted from the mapper to the reducer node is considerably reduced. With the introduction of the combiner, the number of records transmitted from the mapper nodes to the reducers, reduces from $k \times n$ to $k \times m$ where k is the number of keys, m is the number of mapper tasks and $m << n$.

2.1.2 Apache Spark

Apache Hadoop [16] and Apache Spark [51] are two of the several open-source frameworks supporting the Map-Reduce programming model. Over the years, Hadoop has gained popularity as a useful framework in the processing of large datasets using Map-Reduce. However, for iterative machine learning tasks such as clustering which involves multiple iterations over an input dataset, Hadoop is not an optimal choice[51]. Each iteration on the dataset involves a Map-Reduce job and every successive iteration in Hadoop would need to reload the data from the disk, thus, affecting the performance of the overall application. This shortcoming of Apache Hadoop was addressed by Apache Spark.

Apache Spark Architecture

As stated on the Apache Spark website[40], "Apache Spark is a fast and general engine for large-scale data processing". A typical Apache Spark architecture is shown in Figure 2.2. The main components are the *driver*, the *cluster manager* or *master* and one or more *workers* or *slaves*. The *driver* is a process which launches the *SparkContext*. It is the program which contains the *main* method and is launched on the node from where we submit the Spark job. *SparkContext* is a connection to the Spark Cluster and is the main

entry point of the Spark functionality[43]. The SparkContext can connect to different types of cluster managers such as the Spark standalone, Apache Mesos, or Apache YARN. The job of a cluster manager is to allocate resources for the Spark application. Each worker node runs *executor* processes concerned with running computations (executing tasks) and storing the data for an application. Once Spark acquires the connections to the *executor* processes, it sends the code or jar libraries to the *executors* for execution. The output of Spark jobs can be returned to the *driver* through *actions* such as *reduce*(explained later) or can be written to the distributed file system using *actions* such as *saveXXX*(explained later). In conventional data processing applications, the data is transmitted from the data server to the application server. Map-Reduce based applications instead, rely on the concept of transmitting the application code to the data servers, since the size of the data is the major bottleneck in the processing of massive datasets.

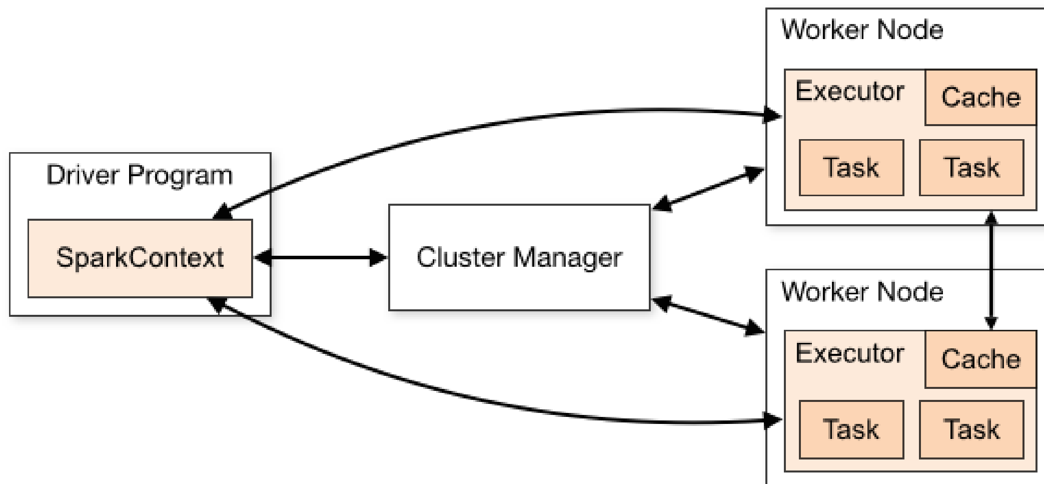


Figure 2.2: Apache Spark Architecture
(Image source: http://spark.apache.org/docs/latest/img/cluster-overview.png)

Apache Spark Concepts

Apache Spark uses an abstraction called Resilient Distributed Datasets (RDD). It is based on the abstraction of Distributed Shared Memory. It is a fault-tolerant immutable collection of elements partitioned across a cluster of machines. The elements of an RDD are the instances of the input dataset and can be operated in parallel by the cluster machines. The RDDs support two kinds of operations : transformations and actions [42]. Transformations are the operations that are executed on the dataset elements in parallel. They perform the necessary computations or transformations on the elements of RDDs. RDDs

being immutable, the transformations do not alter the input RDDs, but return a new RDD dataset with the transformed elements. Examples of some transformations are:

- *map()*: iterates and performs computations on all elements of dataset.
- *filter()*: performs a filter operation on the dataset. Only the elements satisfying the filter criteria are selected in this operation.

Actions, on the other hand, return the results to the *driver* process. Examples of some actions are :

- *reduce()*: performs aggregate like operations on the dataset and returns a single result to the driver.
- *collect()*: returns the elements of the dataset as an array to the driver
- *count()*: returns the count of the elements in the dataset
- *saveXXX()*: saves the elements of an RDD to the distributed file system in different formats (*XXX* can be one of the formats such as *AsTextFile*, *AsObjectFile*, or *AsCassandraTable*)

The *transformations* on RDDs are lazy i.e. RDDs are not materialized when the transformation operations are executed. They are materialized only when an *action* is executed on them. A handle to the RDD stores information about all transformations performed on the RDD, starting from the point when dataset was in a reliable state (e.g. a file). This enables Spark RDDs to be fault-tolerant in case an RDD is lost. Another important feature of a Spark RDD, that assists in faster performance, is the notion of caching a computed RDD. As stated, RDDs may be recomputed every time an action occurs on it. The cached RDDs, however, are computed during the first iteration and are stored on the cluster (memory of individual nodes [1]) after the first materialization for a faster access on the subsequent iterations.

Apache Spark and Map-Reduce

The *map()* and *reduce()* functions in Spark are not equivalents of the conventional *map* and *reduce* functions of the Map-Reduce programming model. In the conventional Map-Reduce, a *map* function emits *key-value* pairs for each input element while a *reduce* function gets a

[1] The storage configuration of a computed RDD is configurable. Details of that are not in the scope of this thesis

key and a list of associated values as input and computes an output per key. In Spark, there is no restriction on the *map()* and *reduce()* to work with key-value pairs. In fact, Spark's *reduce()* aggregates all the values from the *map()* and returns a single value, unlike the Map-Reduce's *reduce()* function. However, to achieve the Map-Reduce like equivalents in Spark, we use the notion of *PairRDDs* available in Spark. *PairRDDs* provide operations such as *reduceByKey()* which is the closest equivalent of the conventional Map-Reduce *reduce()* function. *reduceByKey()* can aggregate the input RDD by key like the conventional Map-Reduce's *reduce()*. To obtain a *PairRDD*, the RDD needs to be transformed into an RDD of 2-tuple pairs of the form $(element_1, element_2)$, where $element_1$ is treated as the key while $element_2$ is treated as a value. Additionally, *reduceByKey()* does the job of a combiner as well, hence contributing to the better performance of Apache Spark jobs.

In this section, we saw a brief introduction to the Map-Reduce programming model and Apache Spark. It was crucial to understand these concepts, which will assist in understanding of the MR-GHSOM algorithm to be presented in Chapter 4.

2.2 Self Organizing Map

The Self-Organizing Map (SOM) introduced by Kohonen [25] is based on the artificial neural network model. It produces an orderly mapping of a high-dimensional data onto a low-dimensional grid. During this mapping, it preserves the topological ordering of an underlying input dataset. The topology preserving property of the SOM reflects the similarities of the input data in terms of the distance in the output space, i.e. similar models of input data are mapped closer to each other than the dissimilar ones. It also makes the SOM flexible with handling of different types of non-linear data distributions.

It was shown in [52], that the SOM gives satisfactory results for various distributions of data. Also, in cases where there are no inherent clusters in the data, the SOM could be used for understanding its distribution and also the distributions of different attributes of the dataset. Most traditional clustering methods may produce clusters even if there are no clustering relations. Thus, the SOM can be considered as a safe algorithm for using directly on any dataset, especially, when nothing is known about the data. Moreover, the visualizations of the SOM (described later) also facilitate an easy understanding of the data characteristics. Owing to the simplicity of the SOM, its capability of depicting different distributions of input data whilst preserving the topology and available visualizations, it proves to be a good candidate for clustering or data analysis. The bibliography of the SOM [24, 35] contains more than 5000 papers which corroborates the popularity and usefulness

of the SOM approach in various domains such as bioinformatics[45], document and web classification[27, 18], medicine[6], image segmentation[3] and market segmentation[22].

2.2.1 Theory of the Self Organizing Map

Architecture of the SOM

The SOM is composed of a grid of neurons called as *feature map*. Each neuron represents a model for a domain of input instances. In other words, each neuron represents a summary or a prototype for a set of instances. The grid can be a hexagonal grid (each neuron has 6 neighbours) or a rectangular grid (each neuron has 4 neighbours). Consider an input dataset X containing n instances, $X = \{x_1, x_2, x_3, \cdots, x_n\}$. Each instance x_i is represented by a d-dimensional attribute vector (instance vector), $x_i = (a_{i,1}, a_{i,2}, a_{i,3}, \cdots, a_{i,d})$. Let k be the number of neurons in the feature map M, $M = \{m_1, m_2, m_3, \cdots, m_k\}$. Each neuron is associated with a d-dimensional weight vector (neuron vector) having the same dimensionality as the input instance vector. Initially the neuron weight vectors are initialized randomly. All input instances are presented to the SOM one by one. A complete pass over the input set is called an epoch, $0 \leq t \leq n$. A complete training of the SOM requires multiple passes over the input data (multiple epochs).

SOM Training

The two main stages of the SOM training are:

- *Finding the best matching neuron (winner)*:
  At a given time instant t, a random input instance $x(t)$ is presented to the neuron map. $m_k(t)$ is the weight vector of the neuron m_k at instant t. The input instance is compared with each neuron using a distance metric d (usually the Euclidean distance for a Euclidean feature space).

$$d_k(t) = \|x(t) - m_k(t)\| \quad \text{for Euclidean spaces} \tag{2.1}$$

  The neuron with the minimum distance from the instance is selected as the winner neuron $c(t)$.

$$c(t) = arg \min_k d_k(t) \tag{2.2}$$

- *Adaptation of model vectors*:
  After identifying the winner neuron, all neuron vectors in the map are adapted to the presented input $x(t)$. The degree of adaptation with respect to the input instance gradually decreases with t as well as the distance from the winner, $c(t)$. In other words, at $t = 0$, almost all the neurons in the neuron map are updated, while towards the end of an epoch, only the winner neuron is updated. The neuron vectors in the map are updated (adapted to the input instance $x(t)$) using,

$$m_k(t+1) = m_k(t) + \alpha(t)h_{ck}(t)[x(t) - m_k(t)] \tag{2.3}$$

  Here, $\alpha(t)$ is the learning rate and $h_{ck}(t)$ is the neighbourhood function. The learning rate controls the amount of correction to the neuron vectors and decreases with time. The neighbourhood function controls the number of units that are adapted as well as the degree of adaptation. During an epoch, the neurons closer to the winner are adapted more to $x(t)$ than the neurons further from the winner. Usually the standard Gaussian neighbourhood function is used for computing the neighbourhood factor.

$$h_{ck}(t) = exp\left(-\frac{\|r_c - r_k\|^2}{2\sigma(t)^2}\right) \tag{2.4}$$

  where, r_c and r_k denote the position of neurons c and k on the two-dimensional grid. $\sigma(t)$ corresponds to the width of the neighbourhood function decreasing with time t.

To summarize, for every input instance, the winner neuron is identified and the weight vectors of all neurons are adapted with respect to this instance. The Figure 2.3 outlines the adjustment phase of the SOM. The black neuron being the winner is adapted the most. The amount of adaptation to the input is depicted by the color gradient (the black colour indicating the most impact while the white colour indicating the least impact).

This variant of the SOM described above is called the *Serial SOM* or the *Online SOM*. It adjusts the neurons on every presentation of an input.

Batch SOM: The Serial SOM is slow in performance as the neuron vectors are updated after presentation of every input data instance. To speed up the SOM training, Kohonen proposed a batch version of the SOM called the Batch SOM [26]. In the Batch SOM, Equation 2.3 is rewritten as,

$$m_k(t_e) = \frac{\sum_{t=t_s}^{t_e} h_{ck}(t) \cdot x(t)}{\sum_{t=t_s}^{t_e} h_{ck}(t)} \tag{2.5}$$

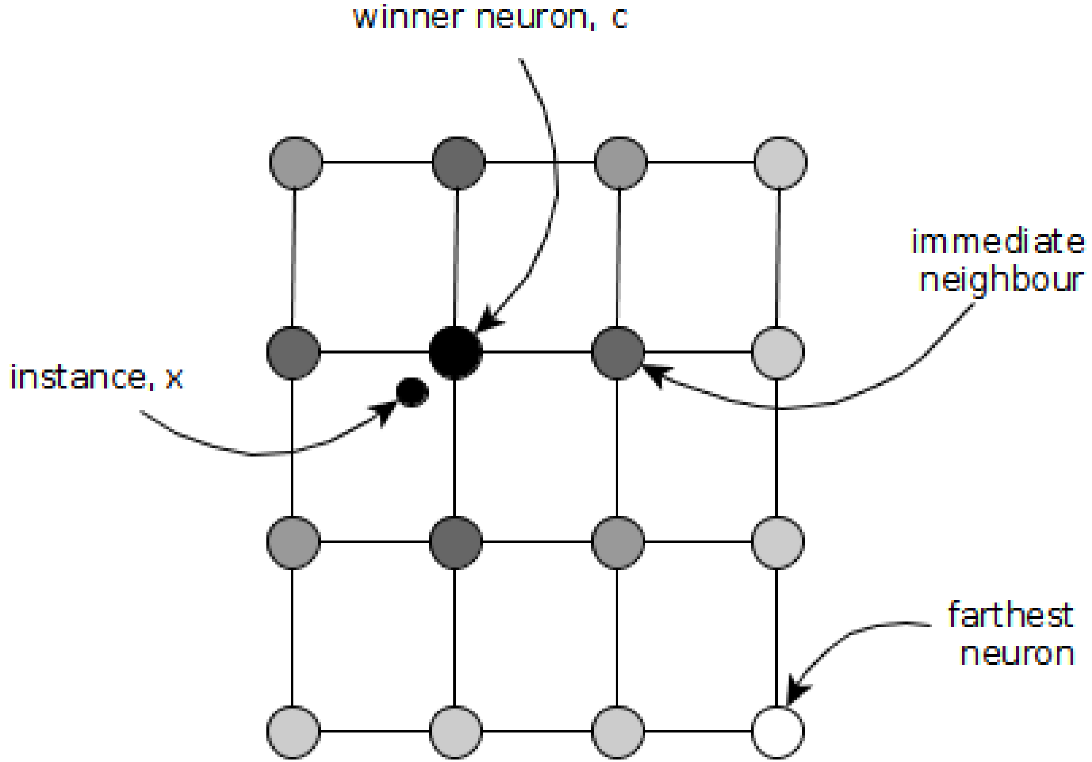


Figure 2.3: Adaptation of neurons to input instance

where, t_s and t_e denote the start and end of an epoch i.e. a pass over the input dataset and $m_k(t_e)$ is the neuron vector at the end of epoch $t = t_e$. For the computation of the winner neuron c, the distance metric in Equation 2.1 is modified as,

$$d_k(t) = \|x(t) - m_k(t_s)\| \quad \text{for Euclidean spaces} \tag{2.6}$$

In the Batch SOM, we do not need to provide the learning rate $\alpha(t)$, which is susceptible to poor convergence when not selected properly [28]. Moreover, the Batch SOM algorithm is well-suited for parallelization. Batches of input data can be processed in parallel and neuron vectors can be updated at the end of all batches. We shall be leveraging the Batch SOM for our Map-Reduce implementation. Algorithm for the Batch SOM is presented in Algorithm 1.

Limitations of SOM and its variants

There are certain limitations of the SOM, as outlined below.

- *Fixed architecture of the SOM*:

Algorithm 1 Batch SOM algorithm [28]

for $epoch = 0$ to n_{epoch} **do**
  for all neurons in the grid **do**
    $numerator = 0$
    $denominator = 0$
  end for
  for all instances x in the input dataset **do**
    for all neurons in the grid **do**
      Find the neuron with the minimum distance from the instance i.e. the winner
    end for
    for all neurons in the grid **do**
      Add to the numerator and denominator using Equation 2.5
    end for
  end for
  for all neurons in the grid **do**
    update the weight vectors
  end for
end for

Before the training of the SOM can commence, the number of neurons and the dimensions of the feature map needs to be provided. This becomes difficult to predict when you are dealing with a dataset with no prior information and also when such a dataset is massive. Deducing an optimal size of the SOM layer involves trial-and-error runs of the SOM training which becomes difficult and cumbersome.

- *Hierarchical relations in data*:
  A dataset may contain hierarchical relations within the input data. In the SOM, the hierarchical relations are depicted by clusters containing neurons topologically closer to each other. It does not properly reflect the hierarchical relations.

Variants of SOM

Over the years, several versions of the SOM have evolved. In this section, we will look at some of the SOM versions or architectures similar to the SOM.

- *Neural Gas Algorithm[33]*:
  This algorithm starts with a set of floating neurons. On presentation of an input, a distance order of neurons is obtained with respect to the input. An edge is created between the winner neuron and the next closest neuron to the input and the error is computed with respect to the input. The edges have an associated age and the

old edges are removed during the process of training. However, this algorithm also requires the number of neurons to be specified before training like the SOM. Growing Neural Gas Algorithm[14] was devised to overcome this limitation of Neural Gas Algorithm, which adds and removes neurons based on neurons' accumulated error.

- *Incremental Grid Growing[4]*:
  This algorithm features a dynamically growing map. The growth occurs only at the boundary neurons which are expanded to more neurons in the available spaces. The algorithm adds connections between neurons that are closer to each other in terms distance. Similarly, it deletes connections between neurons that are distant from each other beyond a certain threshold. Another similar approach called GSOM based on Incremental Grid Growing was proposed in [1]. The proposed algorithm outlines a different technique for initializing the weights of new nodes and introduces a parameter of *spread factor* to control the growth of the map.

- *Growing Grid[13]*:
  This approach again features a dynamically growing map. It starts with a 2×2 grid of neurons and grows by adding a complete row or column, unlike Incremental Grid Growing which grows by adding nodes at the boundary nodes only. Thus, Growing Grid maintains a rectangular structure.

- *Hierarchical Self Organizing Map[3]*:
  This hierarchical variant of the SOM was proposed for the purpose of image segmentation. The structure of the Hierarchical Self Organizing Map involves SOMs arranged in a hierarchy such that each level contains only one SOM layer. Also the number of neurons in a SOM layer is more than the SOM at the above level. The size of the SOM layers from top to bottom are 1×1, 2×2, 4×4, $\ldots$, $N \times N$, where N depends on the size of the input image. Each layer thus represents a higher level of abstraction than the layer below. The input is fed to the bottommost layer and the layer is trained. The trained vectors from the bottommost layer are fed to the layer above and the second last layer is trained and so on.

- *Hierarchical Feature Map[34]*:
  The Hierarchical Feature Map presents a hierarchical variant of the SOM. It consists of a hierarchy of individual SOM layers, each layer representing the input model at a finer granularity. However, the number of layers and the dimensions of individual SOMs need to be specified in advance, which again brings us back to the fixed architecture shortcoming of the SOM.

- *Hierarchical Self-organizing segmentation model (HSOS)[22]*:

HSOS is another hierarchical variant of the SOM. It is similar to the GHSOM algorithm we shall be using (described in the next chapter) in many aspects. However, unlike the GHSOM, it specifies a static layer (fixed dimensions) on the first level of hierarchy. Secondly, uses a concept of *semantic label* which is similar to class labels in supervised learning. Essentially, every record is associated with a semantic label. After training the layer, the neurons are assigned these semantic labels based on the associated input instances. The neurons with the same semantic label are then grouped as a single unit and expanded onto the next level of the hierarchy. Moreover, each layer corresponds to the topological ordering based on a subset of input vector attributes (referred to as segmenting variables). This approach requires semantic labels which are not available in most clustering real world datasets.

- *Map-Reduce Variants of SOM*:
  Considering the usefulness of the SOM, research community sensed the need for migrating the SOM algorithm to train on massive datasets. These datasets could be of the size of gigabytes, terabytes or even petabytes. Some of the Map-Reduce variants of the SOM developed in the process are:
  - Weichel[49] presents a Map-Reduce variant of the SOM implemented on the Hadoop framework. The number of *key-value* pairs shuffled from the map to the reduce task is equal to $n \times k$, where n is the total number of input instances and k is the number of neurons in the layer. This algorithm does not use the concept of *Combiner*, which would greatly reduce the number of *key-value* pairs transmitted over the network.
  - Another Map-Reduce variant of the SOM based on Apache Spark was proposed recently in [39]. It proposes two algorithms. The first one is similar to the one mentioned in [49] where the number of *key-value* pairs generated is $n \times k$, where n is the total number of input instances and k is the number of neurons in the SOM layer. In the second algorithm, the output of the mapper task is a matrix constituted by rows of input vectors multiplied by the neighbourhood factor and a neighbourhood vector consisting of the neighbourhood factors. The size of the matrix is $k \times n$ where k is the number of neurons and n is the number of input vectors, while the size of neighbourhood vector is equal to the number of neurons.

In the next chapter, we shall discuss another dynamic variant of the SOM algorithm and also discuss how it is useful and flexible in comparison to the above mentioned variants.

2.2.2 Visualizations of SOM

One of the important features of the SOM is the useful visualizations that have been developed over the years, which makes the analysis of the SOM layer and hence, the data distribution very easy and intuitive. The visualizations of a SOM neuron map enable the identification of clusters and understand the distribution of different attribute values. Vesanto[47] outlines different methods used for visualizing SOM feature maps. We shall describe two such methods - UMatrix and Component Planes - which will be used in our visualizations.

- *U-Matrix*:
  U-Matrix[46] is a visualizing technique for showing the cluster boundaries in the feature map. The number of cells in the matrix is almost double the number of neurons in the neuron map. To compute a U-Matrix, a distance representing cell (distance cell) is inserted between every pair of neurons and it contains the distance value between the adjacent neurons. The U-Matrix is generally shown in gray-scale. Higher values (dark shade) in distance cells represent large distance or dissimilarity between the adjacent neurons, while lower values (light shade) in the distance cells represent similarity between the adjacent neurons. Thus, the dark shaded distance cells depict the cluster boundaries.
- *Component Planes*:
  Component Planes are usually drawn for each attribute a_i, for $i = 0, 1, 2, \cdots, d$, of a neuron vector. The component planes depict the distribution of an attribute a_i across the SOM layer. To compute the component plane for an attribute a_i, another grid of the same size as the neuron map is created and the values in the component plane cells are set equal to the values of the attribute a_i for the corresponding neuron weight vector in the feature map.

2.3 Handling Mixed-attributes

For data mining applications, it is crucial to find a method to compare two objects with respect to similarity and dissimilarity. When objects have numeric attributes, this comparison is done using distance measures such as the Manhattan distance or the Euclidean distance. The scenario changes when the objects also have categorical attributes. The domain of categorical attributes contain a discrete set of values. There is no inherent ordering or comparison mechanism for these values. Some examples of categorical attributes

are marital status {single, married, divorced, widowed}, gender {male, female} and type of house {apartment, condomonium, townhouse}.

Most clustering algorithms are conventionally formulated for numeric attributes. However, the real-world datasets usually contain both numeric and categorical attributes, also referred to as *mixed* attributes. In this section, we shall see how clustering algorithms formulated for numeric attributes have been extended to handle mixed attribute datasets.

2.3.1 Binary Encoding

One of the simplest and the most popular approach for extending an algorithm to mixed attribute domain is *Binary encoding*. In this approach, categorical attributes in a dataset are transformed into a set of binary attributes. The number of binary attributes created is equal to the number of discrete values in the domain of the categorical attribute. Each binary attribute corresponds to one distinct value of the categorical attribute. After transformation, all binary attributes are treated as numerical attributes and the dataset can be processed normally. There are several disadvantages of this approach[19]. Firstly, the resulting dataset after binary encoding has an increased dimensionality which increases the space and computational complexity of the processing algorithm. A categorical attribute with 100 distinct values could result in a dataset where each record has atleast 100 binary attributes. Also, the transformed dataset is very sparse since out of 100 attributes only one attribute has a value of 1. Secondly, an addition of a new value to the domain of categorical attributes needs a change in schema of the dataset.

2.3.2 Simple Matching

Another approach that is used is called *Simple Matching*. In this, two categorical values have a distance of 0 if they are equal and have a distance of 1 if they are not equal. For two instances x and y having d_n numeric attributes and d_c categorical attributes, the total distance is calculated as the sum of distance for numeric attributes and the sum of distance of categorical attributes.

$$dist(x, y) = \sum_{i=1}^{d_n} \|x_i - y_i\| + \sum_{i=1}^{d_c} \delta(x_i, y_i) \tag{2.7}$$

where,

$$\delta(x_i, y_i) = \begin{cases} 0 & x_i = y_i \\ 1 & x_i \neq y_i \end{cases} \tag{2.8}$$

This approach is used in *k-modes*[21] and *k-prototypes*[20] algorithms which are variants of the *k-means* clustering algorithm for handling categorical and mixed-attribute dataset respectively. For updating the cluster centres in the variants of k-means algorithm, a frequency based approach was used. The value of the jth categorical attribute of a cluster centre was set to the most frequent value of the jth attribute for the instances in that cluster.

A variant of the SOM called NCSOM[7], used a similar simple matching and frequency based neuron update approach for the SOM. To find the winning neuron c, the distance of instance x_i from neuron m_k is calculated as,

$$dist(x_i, m_k) = \sum_{l=1}^{d_n} \|x_{il} - m_{kl}\| + \sum_{l=1}^{d_c} \delta(x_{il}, m_{kl}), \quad \delta(x_{il}, m_{kl}) = \begin{cases} 0 & x_{il} = m_{kl} \\ 1 & x_{il} \neq m_{kl} \end{cases} \tag{2.9}$$

For updating the neurons for the Batch SOM, the numeric attributes were updated as,

$$m_k(t_e) = \frac{\sum_{t=t_s}^{t_e} h_{ck}(t) \cdot x(t)}{\sum_{t=t_s}^{t_e} h_{ck}(t)} \tag{2.10}$$

where, t_s and t_e denote the start and end of the epoch, $x(t)$ is the instance presented at time t, h_{ck} is the neighbourhood function for the neuron m_k with respect to the winner neuron, c. For updating the categorical attributes of neurons, the frequency of each categorical value is computed as,

$$F(\alpha_l^r, m_{kj}(t_s)) = \frac{\sum_{t=t_s}^{t_e} h_{ck}(t) \mid x_l(t) = \alpha_l^r}{\sum_{t=t_s}^{t_e} h_{ck}(t)} \tag{2.11}$$

where, α_l^r is the rth value of the lth categorical attribute of instance $x(t)$. The value α_l^c having the maximum frequency is accepted for the attribute m_{kl} for neuron m_k if its frequency is more than the total frequency of the other values or accepted randomly based

on threshold θ. This is depicted in Equation 2.12.

$$m_{kl}(t_e) = \begin{cases} \alpha_l^c & \text{if } F(\alpha_l^c, m_{kl}(t_s)) > \sum_{r=1, r\neq c}^{n_{\alpha_l}} F(\alpha_l^r, m_{kl}(t_s)) \\ \alpha_l^c & \text{else if } random(0,1) > \theta \\ m_{kl}(t_s) & \text{otherwise} \end{cases} \tag{2.12}$$

2.3.3 Distance Hierarchy

Hsu[19] proposed a *distance hierarchy* method for handling categorical attributes in the SOM. This variant of the SOM was called GSOM (Generalized Self Organizing Map). The approach extends the concept hierarchy technique[17] by giving weights to the links. It provides a method for calculating the distance between mixed, numeric and/or categorical data in a uniform manner. This is accomplished by mapping the values to the *distance hierarchy* of attributes and calculating their distance in the hierarchies.

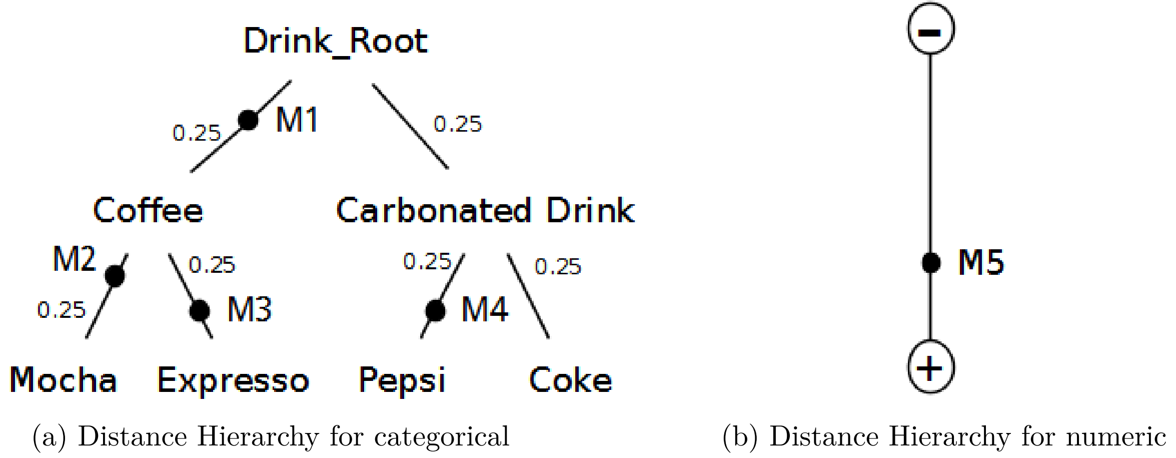
The *distance hierarchy* approach provides a better representation of the similarities or dissimilarities between the categorical values. For example, if we consider a categorical attribute of *Drink* with values {Mocha, Expresso, Pepsi, Coke}, Mocha and Expresso (types of coffee) are more similar to each other than Mocha when compared with Pepsi (carbonated drink). To compute the distance between such categorical values, we construct a *distance hierarchy* for the categorical attribute. The distance hierarchy for *Drink* attribute is shown in Figure 2.4a. The labels at the leaf nodes represent the distinct categorical values for the attribute. Similar values according to the concept hierarchy are placed under a common parent (Coffee or Carbonated Drink) which represents an abstract concept. Each link is given a weight (0.25 in the figure). The link weights are usually assigned by domain experts. A point X in the *distance hierarchy* is represented by two parts: *(anchor, offset)* denoted as (N_X, d_X). The *anchor* is equal to a value from the domain of the leaf node values and the *offset* is the distance of the point from root of the hierarchy. The *offset* value for the leaf nodes is equal to the total path length from the root to the leaf. For any two points, X and Y in the hierarchy, there is a point which is referred to as the *least common point* of X and Y, denoted as $LCP(X, Y)$. $LCP(X, Y)$ is defined as one of the following:

1. either X or Y, if X and Y refer to the same point i.e. $N_X = N_Y$ and $d_X = d_Y$, or
2. X if X is an ancestor of Y, i.e. X lies on the path from the root to Y, or
3. least common ancestor of X and Y

The distance $dist(X, Y)$ between two points X and Y is calculated as

$$|X - Y| = d_X + d_Y - 2d_{LCP(X,Y)} \tag{2.13}$$

where $d_{LCP(X,Y)}$ is the distance of the *least common point* from the root.



(a) Distance Hierarchy for categorical (b) Distance Hierarchy for numeric

Figure 2.4: Distance Hierarchy for Mixed attributes

Let $M1$, $M2$, $M3$ and $M4$ be some points in the hierarchy such that $M1 = (Mocha, 0.1)$, $M2 = (Mocha, 0.3)$, $M3 = (Expresso, 0.4)$ and $M4 = (Pepsi, 0.4)$ as shown in Figure 2.4a. The distances between these points can be calculated as,

$$dist(M1, M2) = 0.1 + 0.3 - 2 \times 0.1 = 0.2 \qquad \ldots LCP(M1, M2) = M1$$
$$dist(M1, M3) = 0.1 + 0.4 - 2 \times 0.1 = 0.3 \qquad \ldots LCP(M1, M3) = M1$$
$$dist(M1, M4) = 0.1 + 0.4 - 2 \times 0 = 0.5 \qquad \ldots LCP(M2, M3) = Drink_Root$$
$$dist(M2, M3) = 0.3 + 0.4 - 2 \times 0.25 = 0.2 \qquad \ldots LCP(M2, M3) = Coffee$$
$$dist(M2, M4) = 0.3 + 0.4 - 2 \times 0 = 0.7 \qquad \ldots LCP(M2, M4) = Drink_Root$$

As we can see, this approach replicates the fact that Mocha and Expresso are less dissimilar from each other as compared to Mocha and Pepsi.

For numeric attributes, the distance hierarchy can be constructed as shown in Figure 2.4b. The hierarchy contains two nodes - *"-"* and *"+"* - representing the minimum and maximum value for a numeric attribute. A point in this hierarchy has an *anchor* value of *"+"* and an *offset* value equal to the actual numeric value of the attribute. The equation for the distance between any two points in this hierarchy remains same as Equation 2.13.

As the equation for computing the distance between numeric and categorical attributes

remains the same, the total distance between two instances X and Y having d mixed attributes, can be computed as,

$$d(x,y) = \left(\sum_{i=1}^{d} |dh_i(X) - dh_i(Y)|^2 \right)^{1/2} \tag{2.14}$$

where, $dh_i(X)$ and $dh_i(Y)$ are the mapping of points in the distance hierarchy for attribute i.

With respect to the SOM, the attribute values of data instances and neuron vectors are mapped to the corresponding attribute's distance hierarchy. A distance hierarchy mapping of a categorical attribute l_c for a dataset instance x is (N_{xl_c}, d_{xl_c}), where N_{xl_c} is the symbol value of the categorical attribute and d_{xl_c} is the offset of the corresponding leaf node from the root. For a numeric attribute l_n of instance x, the distance hierarchy mapping is $(+, d_{xl_n})$ where, d_{xl_n} is the numeric value of the attribute. Value of the root node "$-$" is the minimum value of the attribute l_n. As we know, before training begins, neuron vectors of the SOM are initialized randomly. For a neuron m of the map, a numeric attribute l_n is randomly initialized as $(+, d_{ml_n})$ where d_{ml_n} is a random value between the minimum and the maximum value of the attribute l_n. To initialize a categorical attribute l_c for the neuron, a random symbol value from the domain of attribute l_c is selected as the *anchor*. For the offset d_{ml_c}, a random value between 0 and the offset of the leaf corresponding to the anchor is taken. Thus, distance hierarchy mappings for instance attributes correspond to leaf nodes while those for the neuron vector attributes correspond to intermediate points in the distance hierarchy. Then, the training of the SOM proceeds as described earlier - find the winner neuron for each input instance by computing the distance using Equation 2.14 and adapt the winner neuron and its neighbourhood neurons to the input instance.

During the adaption process, the distance hierarchy points for the neurons move towards the respective leaf nodes of the input instance. Let (N_M, d_M) represent the mapping for the neuron M, (N_X, d_X) be the mapping for the instance X, P be the conceptual parent of X at an offset d_P from root and δ be the adaption amount of M towards X. Following cases arise during the adaption (Figure 2.5):

- *Case 1:* When M is the ancestor of X and P, and after adjustment $d_M + \delta$, M does not cross over P, then the new value M' is $(N_M, d_M + \delta)$. This is illustrated in Figure 2.5a. The anchor of M does not change.
- *Case 2:* When M is the ancestor of X and P, and after adjustment $d_M + \delta$, M crosses over P, then the new value M' is $(N_X, d_M + \delta)$. This is illustrated in Figure 2.5b.

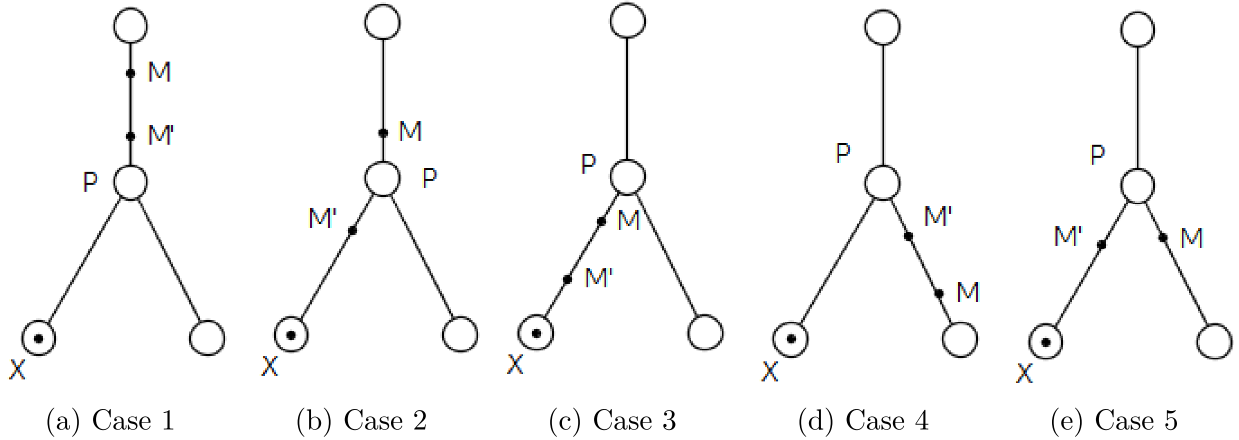


Figure 2.5: Adaption of Neuron M to instance X

The anchor of M becomes equal to the anchor of X, $N_M = N_X$.

- *Case 3:* When P is the ancestor of M, M is in the path between P and X ($N_M = N_X$), then the new value M' is $(N_X, d_M + \delta)$. This is illustrated in Figure 2.5c. The anchor of M does not change (it is already same as the anchor of X).
- *Case 4:* When P is the ancestor of M, M is not in the path between P and X ($N_M \neq N_X$) and M does not cross over P, then the new value M' is $(N_M, d_M - \delta)$. This is illustrated in Figure 2.5d. The anchor of M does not change ($N_M \neq N_X$).
- *Case 5:* When P is the ancestor of M, M is not in the path between P and X ($N_M \neq N_X$) and M crosses over P, then the new value M' is $(N_X, 2d_P - d_M + \delta)$. This is illustrated in Figure 2.5e. The anchor of M changes to N_X.

The *distance hierarchy* approach, thus provides a uniform distance measurement for mixed attributes, and also a finer measure of dissimilarity between categorical attributes. However, creating a distance hierarchy requires intervention of a domain expert and can be used when the domain of values is static and known before hand. Also, it needs additional space for storing the distance hierarchies which could affect the performance when the domain of attribute values is large.

2.4 Conclusion

In this chapter, we looked at the Map-Reduce programming model and a framework supporting this programming model called *Apache Spark*. Apache Spark using the notion

of RDDs and caching, proves to be a useful Map-Reduce computation framework, especially, for applications which requires multiple iterations over a dataset such as data mining algorithms.

We also discussed the theory behind the Self Organizing Map. The Self Organizing Map is an effective data analysis technique. It preserves the topology of a high-dimensional dataset and enables understanding of its characteristics. We also outlined some of the variants of the SOM that have been developed over the years to overcome the limitations of the traditional SOM and used in different domains.

We also looked at some of the commonly used approaches in handling of mixed attributes in a dataset. The *distance hierarchy* approach came out as a very promising technique for handling both numeric and categorical attributes in a uniform manner.

In the next chapter, we will discuss another dynamic variant of the SOM called Growing Hierarchical Self Organizing Map (GHSOM), and also show how we can tailor it to handle a mixed attribute dataset by leveraging techniques from the *distance hierarchy* approach.

Chapter 3

Growing Hierarchical Self Organizing Map for Mixed Attributes

In this chapter, we will introduce another dynamic variant of the SOM called the Growing Hierarchical Self Organizing Map (GHSOM)[10]. First, we shall present the theory behind it. Then, we shall discuss how we adopted the *distance hierarchy* approach in the GHSOM so that it can be extended for handling a mixed attribute dataset.

3.1 Growing Hierarchical Self Organizing Map

The SOM has proved to be an effective tool for data analysis and exploration of high-dimensional data. However, it has some limitations. Firstly, the size of a map, in terms of the number of neurons, needs to be specified before training. For a dataset with no prior knowledge, it is difficult to ascertain the size required to get a satisfactory result and is deduced by the trial-and-error approach. Secondly, some datasets may have inherent hierarchical relations in the data. Such relations are not reflected in a straight-forward manner by the traditional SOM. We saw some variants of the SOM in the previous chapter aimed to address these limitations but none of them addresses both these limitations properly.

The GHSOM addresses both limitations of the SOM giving a very dynamic structure that orients itself according to an underlying data. Moreover, the GHSOM has a flexible structure which can be controlled using appropriate parameters - τ_1 and τ_2 (explained later in this section). The GHSOM can be configured using these parameters, to create a

single large SOM layer whose final size was determined dynamically or create a hierarchical tree-like SOM which shows hierarchies in data.

3.1.1 Architecture of GHSOM

The GHSOM was introduced in [10] and explained in more detail in [38] and [12]. It has a multilevel hierarchical arhitecture. Each level is composed of one or more independent SOM layers. Each independent neuron layer is capable of growing in two dimensions (by adding rows or columns) according to the input data incident on it. Each map grows till it attains a representation of the data at a particular level of detail. The map at the top level usually depicts a generalized data at a coarser granularity. In a SOM layer, each neuron respresents a summary or a prototype of the represented set of instances. The similiar neurons lying next each other form larger clusters. The neurons in the parent layers that have too diverse data mapped onto them or representing a diverse set of instances spin off another SOM layer at the lower level. This lower level SOM again grows according to the data represented by the parent neuron and represents it at a finer level of granularity. Neurons with homogenous data mapped onto it do not spin off or expand into another SOM at a lower level. Thus, the GHSOM is capable of a two dimensional growth as well a tree-like growth in depth for depicting the hierarchical structure. The final structure of the GHSOM, hence depends on the input data and its distribution.

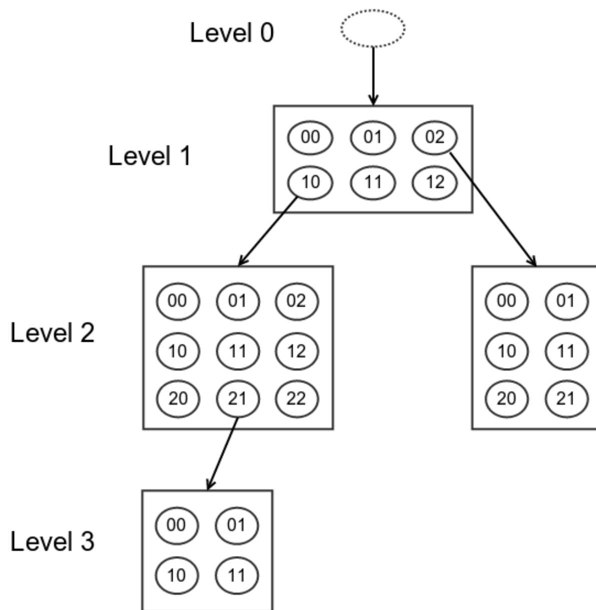


Figure 3.1: Typical structure of a trained GHSOM

Figure 3.1 shows a typical structure of the GHSOM. Map at level 0 is a fictitious map

with a single neuron representing the mean of all input instances (explained later). For this trained GHSOM, the map at level 1 is the first SOM layer which started with dimensions of 2×2 neurons and grew to a map of 2×3. From the level 1 map, neurons "10" and "02" expanded into new independent maps at level 2. The first map in level 2 grew to a size of 3×3 neurons and neuron "21" expanded into another map at level 3. The other map in level 2 grew to a size of 3×2 neurons and no neurons expanded any further. As we can see, the overall structure of the GHSOM is quite dynamic. This structure can be controlled by the values of the parameters τ_1 and τ_2 as we will see in the next section.

3.1.2 Training and Growth Process

The GHSOM adapts its structure according to the input data. This adaptation revolves around the notion of *Mean Quantization Error* as explained below. To keep things simple, we shall consider only numerical attributes for now and the distance measure as Euclidean distance.

- *Mean Quantization Error of a Neuron (mqe)*:
  Mean Quantization Error mqe_k of a neuron m_k in map m, as defined by [12], is the deviation between its weight vector and the input vectors x_i which are elements of the set of input vectors C_k represented by the neuron.

$$mqe_k = \frac{1}{|C_k|} \sum_{x_i \in C_k} \|m_k - x_i\| \tag{3.1}$$

- *Mean Quantization Error of a Map (MQE)*:
  Mean Quantization Error MQE_m of a map m is defined as the mean of the mean quantization errors of only those neurons in the map onto which the data instances were mapped. Let K be the total neurons in map m, and $U \subseteq K$ be the subset of neurons onto which data is mapped, then,

$$MQE_m = \frac{1}{|U|} \sum_{k \in U} mqe_k \tag{3.2}$$

These measures are used to assess the quality of the SOM layer. The decision of whether to grow the current layer in two-dimensions or expand into new maps at lower levels is guided by these measures.

Level 0 Map Initialization

Before the training begins, a level 0 map is created consisting of only one neuron. The weight vector for this neuron is initialized to the mean of all input vectors m_0 in the dataset. Then, we compute the mqe_0 for it with respect to the input using equation,

$$mqe_0 = \frac{1}{|C_n|} \sum_{x_i \in C_n} \|m_0 - x_i\|, \tag{3.3}$$

where C_n is the set of all n input instances. mqe_0 represents the overall dissimilarity in the input dataset and will direct the growth process of the GHSOM as we will see.

Training and 2D-Growth of SOM maps

The first neuron map is created at level 1 consisting of 2×2 neurons. This map is trained using the conventional *Serial* or *Batch SOM* training procedure. We used the *Batch SOM* algorithm because it is faster and more suited to parallelization. Like the conventional SOM, the training process involves multiple epochs or passes over an input dataset. In each epoch, we find the winner neuron and compute the adaptation of all the neurons in the neighbourhood of the winning neuron. At the end of each epoch, the weight vectors of all the neurons are updated using,

$$m_k(t_e) = \frac{\sum_{t=t_s}^{t_e} h_{ck}(t) \cdot x(t)}{\sum_{t=t_s}^{t_e} h_{ck}(t)} \tag{3.4}$$

where, t_s and t_e denote the start and end of an epoch, $h_{ck}(t)$ is the neighbourhood factor for a neuron m_k with respect to the winner neuron c for input instance $x(t)$ presented at time t.

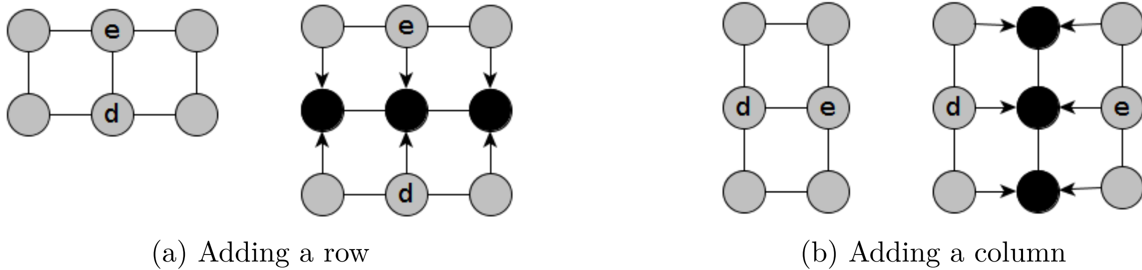
After the training is complete, the map is analysed and the mean quantization error for the map MQE_m is computed using Equation 3.2. Higher value of MQE_m signifies that the map m does not represent the input data well and requires more neurons to produce a better representation of the input domain. Formally, this is governed by the equation,

$$MQE_m < \tau_1 \cdot mqe_p \tag{3.5}$$

where, mqe_p is the mean quantization error of the parent neuron in the upper level map from which this map m is expanded. The map will grow until the condition in Equation 3.5 evaluates to true. While the condition is false, the map grows by adding a row or a column.

For the map at the first level, the condition is $MQE_1 < \tau_1 \cdot mqe_0$, where MQE_1 is the MQE of the map 1 at level 1 and mqe_0 is the mean quantization error of the neuron in level 0 map (Equation 3.3). A smaller value of τ_1 results in a large flatter map while a high value of τ_1 creates a map with relatively less neurons.

Two-dimensional Growth of a map: While the criterion given by Equation 3.5 is not satisfied, the neuron map layer grows by adding a row or column to the map. For growing the map, the neuron with the highest *mqe* is identified called as the *error* neuron *e*. The high value of *mqe* for this neuron indicates that this neuron does not represent the mapped input domain well. Next, the most dissimilar direct neighbour *d* of the *error neuron* is identified. The most dissimilar neighbour *d* is the neuron which has the maximum distance in terms of weight vector from *e*. A new row or column is inserted between *e* and *d*. The vectors of the new neurons in this new row or column are initialized as the average of the weight vectors of their corresponding adjacent neighbours. Figure 3.2 shows how a new row or column is added between *e* and *d*. The black coloured row/column of neurons is the new row/column added. The arrows indicate the neighbour neurons used by the new neurons for the initialization.



(a) Adding a row

(b) Adding a column

Figure 3.2: Adding new row or column between *error neuron* and *dissimilar neighbour neuron*

Hierarchical Growth

After the condition in Equation 3.5 is satisfied, the training of the map is complete and it represents data at the desired level of granularity. All the neurons are then analysed for expansion into another map at a level below. This is governed by the criterion mentioned in Equation 3.6. It is controlled by the parameter τ_2 which depicts the minimum level of granularity of data, that is required to be represented by each neuron. Equation 3.6 is the global stopping criteria for the complete GHSOM training. It defined as a fraction of the

dissimilarity indicated by mqe_0.

$$mqe_k < \tau_2 \cdot mqe_0 \tag{3.6}$$

The neurons which do not satisfy Equation 3.6 are expanded into a new map at the next level of hierarchy. τ_2 criterion is always computed with respect to mqe_0 for all neurons in all maps at all levels.

The initial dimensions of the new map are 2×2 and it is trained and grown as described earlier. The new map, however, is trained only on the input vectors that are represented by the corresponding parent neuron in the upper level. The new map trains and grows till it adheres to the τ_1 criterion (Equation 3.5). Similarly, all the expanded maps are trained and grown. For each map, after the training is complete, its neurons are analysed for expansion and expanded into new maps if required. The growth and training of the GHSOM stops when the criterion in Equation 3.6 is satisfied by all neurons in the lowest levels.

Initialization of new maps in hierarchy: When the new maps are expanded from parent neurons, the weight vectors of the neurons in the new map can be initialized randomly. However, this will affect the global topology of the map. Since, topology preservation is one of the important characteristics of the SOM, the maps in the lower levels should preserve the topology orientation of the parent neuron.

Random initialization would not enable the new map to adhere to the parent neuron's topology orientation with respect to its neighbours. There are quite a few approaches to preserve this topology orientation of the child maps. The simplest approach would be to copy the weight vectors of the parent neuron's neighbours for the initialization of the neurons in the child map. [11] proposes another approach based on the average of the weight vectors of the parent neuron's neighbours. It uses different initialization techiques for the map based on whether the parent neuron is located on the edge or is surrounded by eight neighbours.

[5] uses a similar approach but considers only four immediate neighbours and provides a uniform method of initialization irrespective of the location of parent neuron. For the parent neurons on edges, it creates virtual neurons which are mirrors of the existing neighbours. The virtual neurons are shown in Figure 3.3 using dotted circles. For neuron a, the

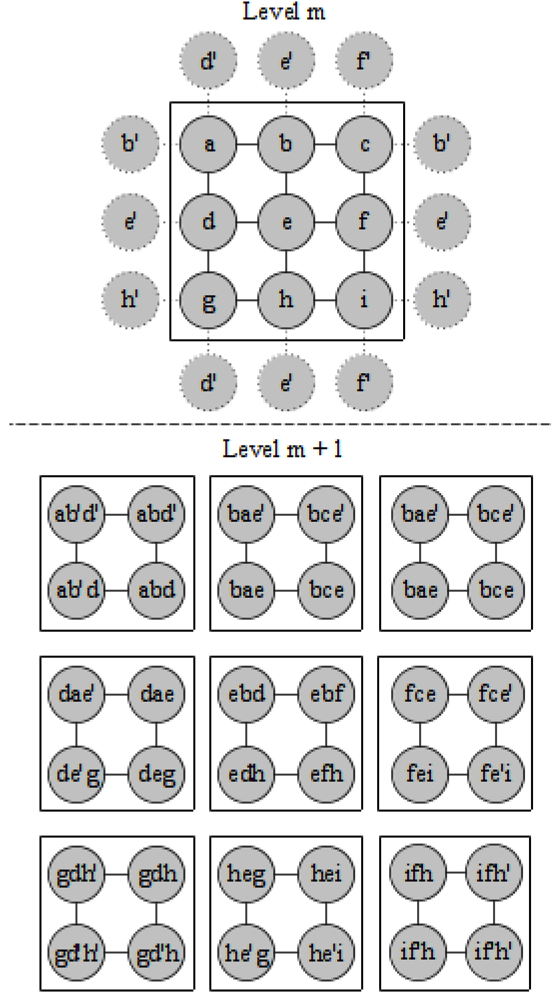


Figure 3.3: Initialization of neurons in new map[5]

virtual neuron b' on the left is created using,

$$b' = a + (a - b) \tag{3.7}$$

Initialization of the neurons in the child neuron layer is done using the average of the weight vectors of the parent and its corresponding neighbours. For instance, the neuron *abd* of the child map is initialized as,

$$abd = \frac{a + b + d}{3} \tag{3.8}$$

For our implementation, we used this method because of its simplicity of initialization irrespective of the location of the parent neuron.

The complete training of the GHSOM, thus involves various stages starting from com-

puting the mqe_0 and ends when each neuron in the lowest level layers satisfy the Equation 3.6. This concludes the overall training process of the GHSOM. Next, we shall investigate the impact of the τ_1 and τ_2 parameters on the overall structure of the GHSOM.

3.1.3 Effect of τ_1 and τ_2

τ_1 and τ_2 are the crucial parameters which determine the final structure of the GHSOM. The training of the GHSOM may not result in a balanced hierarchy of maps in most cases. This is in adherence to the fact that the GHSOM adapts it structure completely according to the input data. [12] explains the impact of these parameters and how they can be used to control the shape and the size of the GHSOM.

- τ_1: It controls the growth process of the GHSOM.
  - A lower value of τ_1 creates shallow hierarchies with large maps at each level. This is because the expected MQE from each map is low and hence each map grows into more neurons to represent the data. A very low value will result in a single large map of the SOM ignoring the hierarchies in the data.
  - A high value of τ_1 creates a deep hierarchical structure of the SOM layers. The maps at each level represent lesser characteristics of the data. In this case, the expected MQE from each map is high and has less number of neurons to represent the data. A high value of τ_1 should be used when exploring the hierarchical relations in the underlying data is the motive of the analysis.
- τ_2: It controls the minimum granularity of data expected to be represented by each neuron in the GHSOM. When a diverse data is mapped onto a neuron, either an additional neuron row/column will be added to represent this diverse data in the same map or a new child map will be spun off to represent this diverse data at a finer level. The expected minimum granularity is based on the inherent dissimilarity of input data indicated by mqe_0. This parameter signifies the global stopping criterion for the GHSOM. The lower the value of τ_2, more neurons can be expected in the complete GHSOM as each neuron is expected to represent fairly homogenous data.

3.2 Faster two-dimensional growth using the *batch growth*

In the conventional GHSOM, every two-dimensional growth iteration adds only a single row or column to the SOM layer. The initial size of an individual SOM layer in the GHSOM is 2×2 neurons. Often the dataset cannot be represented by just four neurons. The SOM layer needs to grow to a size of tens or thousands of neurons, to represent the input data at the desired quality of representation. This final size depends on the input data and the value of the parameter τ_1. Thus, for a highly heterogenous data or a low value of τ_1, several growth iterations would be required to grow a map of four neurons to a map of tens, hundreds or thousands of neurons. Moreover, every growth iteration in the GHSOM is followed by an iteration of the SOM training. This training of the SOM is the most computationally expensive part of the GHSOM. Thus, reducing the number of growth iterations and hence, the training iterations, would result in significant savings, in terms of the overall training time of the GHSOM.

To speed up this growth process, we used a *batch growth* approach in our MR-GHSOM. In this approach, instead of adding only one row or column in every growth iteration, we add a batch of rows and/or columns to the map. To do this, while looking for the *error* neuron, rather than selecting just one neuron, we select a set of *error* neurons E and their respective dissimilar neighbours. To compute this set E, we select all the neurons whose mqe is greater than $\tau_1 \cdot mqe_p$, $E = \{m_k \mid mqe_k > \tau_1 \cdot mqe_p\}$. The new rows and/or columns are inserted in the map for all the neurons in E. These neurons are primarily responsible for the high MQE of the map and, hence need more surrounding neurons to express the data represented by them. However, this batch growth is used in a controlled manner, to prevent adding too many unnecessary neurons to the map. The MQE_m of the map m reduces as more neurons are added to it. We use the batch growth approach only till the MQE_m of the map is greater than the target criterion value of $\tau_1 \cdot mqe_p$ by an amount θ. If the difference between the current MQE_m of the map and the target criterion is less than θ, we grow in the conventional way (one row/column at a time).

The results from the experiments shown in Section 5.4 confirm that, this batch growth approach does not have any adverse effects on the final result in terms of the size of the map or the quality of the map. It only helps to speedup the overall training process by reducing the number of growth iterations required to reach the final desired size of the SOM layer.

3.3 Extending GHSOM for Mixed Attributes

The GHSOM discussed so far is applicable to datasets with numeric attributes only. However, most real world datasets contain mixed attributes - numeric and categorical. The *binary encoding* approach for handling mixed attributes is not an optimal choice as it increases the dimensionality of datasets, hence increasing the computational cost of training. As seen earlier, methods for handling mixed attributes in the SOM without resorting to the *binary encoding* approach have been proposed in the literature (NCSOM[7], GSOM[19]). However, these approaches are not sufficient for the GHSOM. In addition to training an independent SOM, the GHSOM requires some more mathematical operations to be defined on the attribute vectors of the instances and the neurons. The mathematical operations required for training the GHSOM are as follows:

- *Computing* mqe_0:
  The first critical step in the GHSOM is to compute the inherent dissimilarity in the input data set indicated by mqe_0 (mean quantization error of level 0 neuron). The value of mqe_0 plays a critical role in the final structure of a trained GHSOM. To compute mqe_0, we first compute the *mean* of all input vectors and initialize the weight vector of level 0 neuron to this *mean* vector. We then compute the *mean distance* of the input vectors from the *mean* vector to deduce the value of mqe_0.

  Given a dataset with mixed attributes, *simple matching* supports the latter operation of computing the distance. However, it does not provide a straightforward mechanism to compute the mean of the attribute values. The frequency based adaptation technique could be used to consider the mean value as the value of the attribute with the highest frequency. However, this would be not be an optimal value when the attribute values are evenly distributed. Consider a categorical attribute like gender having only two values in its domain $\{m, f\}$. If the frequency of m is almost equal to the frequency of f, then the mean could be either m or f and picking either one of them would not be appropriate.

  Computing the mean of categorical attributes is not possible. Hence computing mqe_0 becomes difficult for mixed attributes because of categorical attributes. In Section 3.3.1, for the reasons stated later, we replaced the mean quantization error by variance and used it as a measure of assessing the quality of map and neurons.

- *Training of SOM*:
  The training comprises of two parts:

  - *Finding the winner neuron*: This operation requires finding the neuron with

the least distance from the input instance. Both *simple matching* and *distance hierarchy* techniques support this operation for mixed attributes. For a dataset with categorical attributes only, *Simple matching* will give whole number value of distance while *distance hierarchy* shall give a fractional value of distance and hence has more precision.

  – *Adaptation of winner and neighbour neurons*: This step is again defined by both *simple matching* and *distance hierarchy*. The former employs a frequency based approach for updating the neuron weight vectors. However, as shown in [19], *distance hierarchy* results in better adaptation and hence topology ordering owing to its *concept hierarchy* model. However, [19] employs the approach on *Serial SOM* while we apply it on the *Batch SOM*. For extending it to *Batch SOM*, we need to define some more operations on distance hierarchy approach (Section 3.3.1).

- *Two-dimensional Growth of Map*:
  This step requires computing the mean quantization error of the map using Equation 3.2 and finding the *error neuron* e and its most *dissimilar neighbour* d. Both operations requires computing the distance between two attribute vectors. Distance computations are supported by both, *simple matching* and *distance hierarchy*.

  After identifying e and d, a new row or column is inserted in the map. The neurons of this new row or column are initialized as the average of the weight vectors of their neighbouring neurons. This is similar to computing the mean of the mixed attribute vectors. *Simple matching* would not be an appropriate method for computing the mean for the reasons stated earlier. As we shall see in Section 3.3.1, we can employ the *distance hierarchy* approach here.

- *Hierarchical Growth of Map*:
  After the two-dimensional growth is complete, we find the neuron which does not satisfy Equation 3.6 and expand a new map from it. For initialization of neurons in this new map, we need to define addition, subtraction and average operations on attribute vectors (Equations 3.7, 3.8).

  Simple matching is not a favourable approach here. In particular, the subtraction operation, is not defined for two categorical values. We shall see in Section 3.3.1, how we can extend the *distance hierarchy* approach to support the addition, subtraction and average operations required for this step.

3.3.1 Mixed Attribute GHSOM using *Variance* and *Distance Hierarchy*

In this section, we will describe how we used *Variance* as a measure of dissimilarity and *Distance Hierarchy* in the GHSOM, and hence extended the GHSOM to handle mixed attributes.

Variance instead of *Mean Quantization Error*

The first step of the GHSOM training is computing the mqe_0 for the entire dataset. mqe_0 is the measure of the overall dissimilarity of the input data. However, to compute mqe_0, we need to compute the mean of the input dataset. This is trivial for numerical attributes, however, there is no standard definition of *mean* for categorical attributes. Computing mqe_0 for categorical attributes, thus, becomes impossible.

In the GHSOM, the *mean quantization error* is the deviation of the input instances from the *mean* vector of level 0 neuron or the deviation of mapped instances from the weight vector of a neuron. This concept is similar to the *mean absolute deviation*. The *mean absolute deviation* D is the sum of distances of the input instances from the mean divided by the total number of instances (Equation 3.9). *Variance* is another method of calculating the deviation in the input. *Variance* V is defined as the sum of squared distances of the input from the mean divided by the total number of instances (Equation 3.10). Both measures compute the deviation of input from the mean or a reference. Hence, *variance* can also be used as a measure of dissimilarity in place of the *mean absolute deviation* or *mean quantization error*.

$$D = \frac{\sum_{i=1}^{n} |x_i - \bar{x}|}{n} \tag{3.9}$$

$$V = \frac{\sum_{i=1}^{n} \|x_i - \bar{x}\|^2}{n} \tag{3.10}$$

where x_i is an instance and $\bar{x}$ is the mean of all n instances.

[23] states that for categorical attributes, "unalikeability" is a more natural concept than "variation about mean". It further proposes the *coefficient of unalikeability* to measure the variability in categorical attributes. "Unalikeability" measures how often the categorical

values differ from one another. It is based on the relative frequency of the values. The *coefficient of unalikeability* for a category attribute l is defined as,

$$u_2 = \sum_{i \in Domain(l)} p_i(1 - p_i) \tag{3.11}$$

where $p_i = \frac{frequency(l_i)}{n}$. This equation is based on the findings in [15] which states that variance can be computed independent of the mean by computing the distance between all pairs of entities. However, this variance is twice the variance computed with respect to the mean.

To maintain uniformity throughout the GHSOM training, we replace the *mean quantization error* mqe_k of a neuron k by *variance* var_k on the neuron k. Also *Mean Quantization Error* MQE_m of the map m is replaced by *Mean Variance* MV_m of the map m. Thus, we replace Equation 3.1 and 3.2 by Equations 3.12 and 3.13, respectively.

$$var_k = \frac{1}{|C_k|} \sum_{x_i \in C_k} (\|m_k - x_i\|)^2 \tag{3.12}$$

where, x_i is an instance from the set C_k of instances mapped onto neuron m_k.

$$MV_m = \frac{1}{|U|} \sum_{k \in U} var_k \tag{3.13}$$

where U represents the subset of neurons of map m onto which instances are mapped.

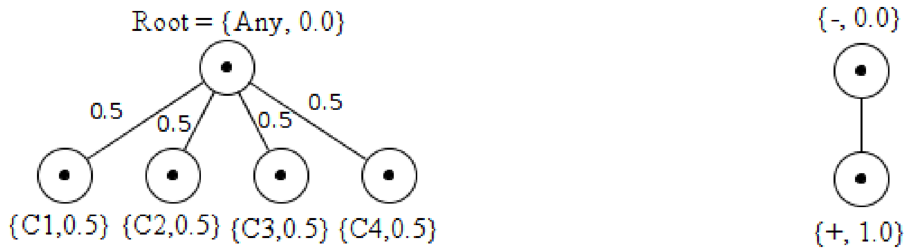
Using *distance hierarchy* in GHSOM for mixed attributes

The *distance hierarchy* approach for handling mixed attributes was introduced in Section 2.3.3. It provides a uniform mechanism for dealing with both, numeric and categorical attributes in the input data. In this method, each attribute value is represented by a point in the distance hierarchy defined by a pair (N_X, d_X) where N_X is the *anchor* or the symbolic value of the attribute and d_X is the *offset* or the path length of the point from the root of the distance hierarchy. For categorical attributes, N_X is one of the values from the domain of the attribute. For numeric attributes, N_X is usually the "+" symbol, and the value of d_X represents the magnitude of the numeric attribute. In terms of categorical attributes, d_X provides the necessary fractional component to the attribute. This helps in calculating fractional distances between the instances instead of whole distances (0 or 1 in simple matching). This fractional component also assists in arithmetic computations at

various stages of the GHSOM.

In the distance hierarchy of a categorical attribute, the non-leaf points in the hierarchy represent neurons' attributes while the leaf points represent the attributes of the instances. The *offset* or the fractional component depicts the fraction by which the neuron attribute is adapted to the *anchor* or the categorical attribute. When we present an instance to the neuron, the neuron's weight vector is adapted by a certain amount to match the instance vector. This adjustment is guided by the neighbourhood factor (Equation 2.4). For the winner neuron, this adjustment amount is 1.0 i.e. it completely updates itself to match the input instance. In other words, when an instance is presented, it pulls the neuron point towards its leaf. The magnitude of the pull is defined by the neighbourhood factor.

In our work, we do not constrain the value of the *offset* to be limited between 0 (root) and the path length of the leaf node corresponding to the *anchor*. Also, the use of the *coefficient of unalikeability* limits our algorithm to use distance hierarchies with two-levels only, as the coefficient does not take into account the notion of concept hierarchies. It treats all the values in a category as equally distinct from each other. The equation of the *coefficient of unalikeability* can be extended to make use of the degree of similarity or dissimilarity between the attributes by using concept hierarchies. However, this is not included in the scope of this thesis work. It will be a part of future work. Hence, for categorical attributes, we created distance hierarchies of two-levels only and treat all the values in the domain as equally distant from each other.



(a) Distance hierarchy for categorical attribute (b) Distance hierarchy for numeric attribute

Figure 3.4: Distance hierarchy for the mixed attribute GHSOM

A typical distance hierarchy for a categorical and numerical attribute, we used in our work, is as shown in Figure 3.4. The distance hierarchy for a categorical attribute with domain values of {C1, C2, C3, C4} is shown in Figure 3.4a. Figure 3.4b shows the distance hierarchy for a numeric attribute. These figures depict the distance hierarchies for the attributes in the normalized form. The link weights for the categorical attribute is set to 0.5, so that the maximum distance between any two leaf nodes is 1.0. For a normalized

numeric attribute, the root node $(-, 0.0)$ represents the minimum value, while $(+, 1.0)$ represents the maximum value of the attribute.

Training of GHSOM for mixed attributes

Now, we shall formally describe the training process of the GHSOM for mixed attributes using the distance hierarchy based approach.

- *Computing* var_0:

  As stated before, we replace the mean quantization error as a measure of dissimilarity by variance. We compute the variance of numerical attributes var_{num} using Equation 3.14.

  $$var_{num} = \frac{1}{|C_n|} \sum_{x_i \in C_n} \sum_{j=1}^{d_n} \|m_{0j} - x_{ij}\|^2 \tag{3.14}$$

  where, C_n is a set of all n input instances, d_n is the number of numerical attributes, x_{ij} denotes a numeric attribute j of input instance and m_{0j} denotes the numerical attribute j of the mean vector of all instances. To compute the variance for categorical attributes var_{cat}, we use the *coefficient of unalikeability*.

  $$var_{cat} = \sum_{l=1}^{d_c} \sum_{i \in Domain(l)} p_{li}(1 - p_{li}) \tag{3.15}$$

  where, $p_{li} = \dfrac{frequency(r_{li})}{n}$ for the categorical attribute l with value r_{li} and d_c is the total number of categorical attributes. Hence, the total variance in the input data var_0 is the sum of var_{num} and $var_{cat}/2$ (as stated earlier this variance is twice the variance computed with respect to the mean).

- *Training of SOM*:

  This is the core part of the GHSOM obviously. We employ the *Batch SOM* algorithm for training an individual neuron layer. The first level neuron map layer is created with 4 neurons (map of 2×2) and the weight vectors of these neurons are initialized randomly. The attribute values of these neuron vectors are basically points in the corresponding distance hierarchies of the attributes. For a neuron m_k, an attribute l is represented in its distance hierarchy as $m_{kl} = (N_{kl}, d_{kl})$. If l is a numeric attribute, $N_{kl} =$ "+" and d_{kl} is set to a random value between the minimum and the maximum value of the attribute. When l is a categorical attribute, N_{kl} is set to a randomly

selected value from the $Domain(l)$ and d_{kl} is set to a random value between 0 and path length of the leaf node corresponding to N_{kl}.

For an instance x and attribute l, the distance hierarchy point is defined as $x_l = (N_{xl}, d_{xl})$. Usually, the values of the attributes are normalized to a range of 0 and 1. If l is a numeric attribute, $N_{xl} =$"+" and d_{xl} is equal to the value of the numeric attribute. If l is a categorical attribute, N_{xl} is set to the categorical value and d_{xl} is set to the path length of the leaf representing N_{xl}. For normalization with respect to categorical attributes, the link weights are designed such that the maximum value of the path length between any two leaf points is 1. Since, we use a two level hierarchy and consider all attribute values as equally distinct from each other, we assign link weight as 0.5.

The training of the SOM involves several iterations over the input data called epochs. Let t_s and t_e denote start and end of an epoch respectively. During an epoch, instances are presented to the map one by one. Let $x(t)$ be the instance presented to the map at time t. For each instance, we first identify the winner neuron $c(t)$, the neuron with the minimum distance from $x(t)$.

$$c(t) = arg\min_k dist_k(t) \tag{3.16}$$

where $dist_k(t)$ is the distance between $x(t)$ and neuron m_k, defined by,

$$dist_k(t) = \left(\sum_{l=1}^{d} \| dh_l(x(t)) - dh_l(m_k(t_s)) \|^2 \right)^{1/2} \tag{3.17}$$

where, $dh_l(x(t))$ and $dh_l(m_k(t_s))$ represent the distance hierarchy mappings of the attributes l of $x(t)$ and $m_k(t_s)$ and d is the number of attributes in the vector. Note that, the weight vector of the neuron at the beginning of epoch t_s is used for identifying the winner neuron c in the *Batch SOM*. The distance between any two points in a distance hierarchy is computed using the Equation 3.18.

$$|X - Y| = d_X + d_Y - 2 \cdot d_{LCP(X,Y)} \tag{3.18}$$

where, X and Y represent distance hierarchy points of two entities, d_X and d_Y represent the offset of the points in the distance hierarchy and $d_{LCP(X,Y)}$ is the offset of the *least common point* of X and Y.

Once the winner neuron c is identified, the adaptation of neighbourhood neurons

is computed using Equation 3.19.

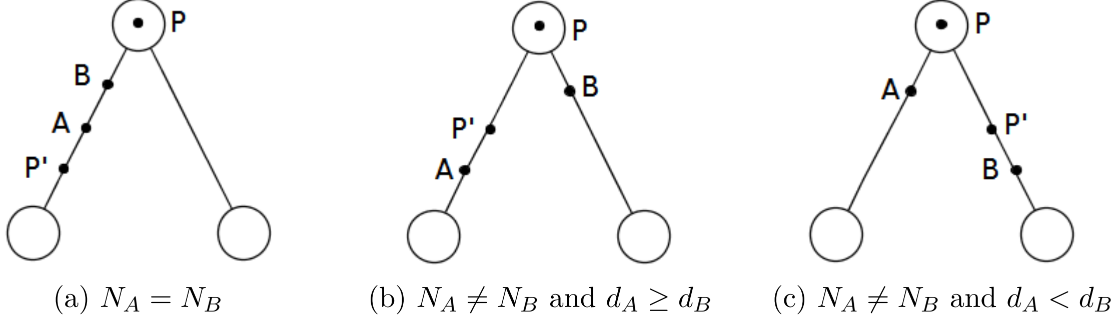
$$m_k(t_e) = \frac{\sum_{t=t_s}^{t_e} h_{ck}(t) \cdot \{dh(x_l(t)) \mid 1 \leq l \leq d\}}{\sum_{t=t_s}^{t_e} h_{ck}(t)} \tag{3.19}$$

where $dh(x_l(t))$ represent the distance hierarchy mapping of attribute l of instance x having d attributes and h_{ck} is the neighbourhood factor for neuron m_k with respect to the winner c. For the multiplication operation in the numerator part, we multiply the *offset* part of distance hierarchy point with the neighbourhood factor. For summating all the terms in the numerator with respect to the distance hierarchy, we defined the addition operation for points in a two level hierarchy, as decribed next.

Addition of two distance hierarchy points: To define the addition operation of two points in distance hierarchy, we introduce a concept of *pliable point.* A *pliable point* P is a point in the distance hierarchy that moves along the paths in the hierarchy as other points are applied to it. Initially, the location of P is the root of the tree. Hence, the initial value of P is (N_P, d_P) where value of $N_P = Any$ and value of $d_P = 0$. The addition operation of two or more distance hierarchy points can be considered as applying each point one after the other to P and adapting or pulling P towards the anchor of the applied point.

Let $A = (N_A, d_A)$ and $B = (N_B, d_B)$ be the two points to be added. So when A is applied to P, the value of P becomes (N_A, d_A) i.e. equal to A. Then we apply point B to this displaced P. B pulls P towards it anchor by an amount equal to the offset value of B, d_B. The final value of the addition is the final location of point P. When, the anchors of A and B are equal i.e. have the same attribute values, it means that the direction of the pull is same for both points. In this case, we simply add the offsets of the two points. When anchors of A and B are different, then the direction of the pull is opposite. Thus, we subtract the offsets. The final value of anchor of P, N_P is set to the closest leaf anchor. Formally, this can be written as,

$$(N_A, d_A) + (N_B, d_B) = \begin{cases} (N_A, d_A + d_B) & \text{case when } N_A = N_B \\ (N_A, d_A - d_B) & \text{case when } N_A \neq N_B \text{ and } d_A \geq d_B \\ (N_B, d_B - d_A) & \text{case when } N_A \neq N_B \text{ and } d_A < d_B \end{cases} \tag{3.20}$$



(a) $N_A = N_B$ (b) $N_A \neq N_B$ and $d_A \geq d_B$ (c) $N_A \neq N_B$ and $d_A < d_B$

Figure 3.5: Addition of points in distance hierarchy

The addition operation for the distance hierarchy points is shown in Figure 3.5. P indicates the initial location of the *pliable point*. A and B are the two points applied to P and are the operands of the addition operation. P' is the final location of P and denotes the result of the addition operation.

All the operations discussed so far, are uniform for both, numerical and categorical attributes. The summation in the numerator with respect to all the n instances, may cause the *pliable point* P to have an offset value greater than the maximum path length of the leaf corresponding to N_P. However, the next operation in Equation 3.19 is dividing this sum by the sum of all neighbourhood factors (denominator of Equation 3.19). We divide the *offset* value of P by the value in the denominator. This will cause the point to fall back inside the distance hierarchy i.e. P will lie between the root and the leaf corresponding to anchor N_P. The attribute l of the neuron's weight vector is updated at the end of an epoch and set to the final value of P.

The training of an individual map goes on for a defined number of epochs. After the training is complete, we evaluate the map to check whether the neurons were sufficient to represent data at the desired quality. If not, we grow the map in two-dimensions and train it again.

- *Two-dimensional growth of Map*:
For this step, we compute the variance of all the mapped instances on each neuron (Equation 3.12) and compute the mean variance of the map MV_m (Equation 3.13). We then evaluate the quality of the map with respect to the variance var_p of the parent neuron for the map m.

$$MV_m < \tau_1 \cdot var_p \tag{3.21}$$

If the map does not satisfy the criterion in Equation 3.21, we identify the set of error neurons E using the *batch growth* approach described earlier and the corresponding

set of dissimilar neighbours D. As a next step, we insert rows and/or columns for each pair of e and d from the set of E and D. We initialize the new neuron weight vectors by the average of the weight vectors of their corresponding neighbour neurons. To initialize an attribute l of a neuron weight vector for neuron z between neurons x and y, we compute the average of the $dh(x_l)$ and $dh(y_l)$.

$$dh(z_l) = \frac{dh(x_l) + dh(y_l)}{2} \tag{3.22}$$

This is similar to computing the result of addition of two distance hierarchy points (Equation 3.20) and dividing the offset of the result by 2. After growing the map, we retrain the map as described earlier.

- *Hierarchical Growth of Map*:
  After the neuron layer achieves the desired quality and satisfies the criterion in Equation 3.21, we evaluate each neuron of the map for the hierarchical growth. We identify neurons that do not represent the data at the desired quality i.e. do not satisfy the criterion in Equation 3.23 and expand them onto a new 2×2 layer.

$$var_k < \tau_2 \cdot var_0 \tag{3.23}$$

  The new expanded map is trained only on instances represented by the parent neuron. Figure 3.3 shows how the new map is initialized for each possible location of the parent neuron (having 3 neighbours or 4 neighbours or 8 neighbours). If the parent neuron is on the edge of the map, we first create virtual neighbouring neuron(s) shown by dotted circles in the figure. These neurons are mirrors of the existing neighbours. With reference to the figure, the neuron b' is created as,

$$b' = a + (a - b) \tag{3.24}$$

  Due to this virtual neuron strategy, the parent neurons at any location in the parent map have exactly 4 direct neighbours. The child map is initialized using the weight vector of the parent neuron and its corresponding direct neighbours. For example, the child neuron 00 is initialized using the weight vectors of the parent neuron, its neighbour at the top and the neighbour on the left, the neuron 01 is initialized using the parent neuron, its neighbour at the top and on the right, 10 is initialized using the parent neuron, its neighbour at the bottom and the neighbour on the left, and lastly, the neuron 11 is initialized using the parent neuron, its neighbour at the bottom and the one on the right. The same is depicted in Figure 3.3. The new neuron vector is

initialized as the average of the 3 vectors (parent and its two neighbours in respective direction).

$$newNeuron = \frac{parent + neighbour1 + neighbour2}{3} \tag{3.25}$$

As we can see, this initialization process involves the addition, subtraction and division operations on the distance hierarchy points. We have already described the addition and division operation previously. So now, we shall define how a subtraction operation can be performed on two distance hierachy points. A subtraction operation can be considered as an opposite of addition operation. In addition, one point pulls the other point towards its anchor. In subtraction operation instead, it pushes the other point away from its anchor. More formally, we define the subtraction between point A and B in the two level distance hierarchy as,

$$(N_A, d_A) - (N_B, d_B) = \begin{cases} (N_A, d_A + d_B) & \text{when } N_A \neq N_B \\ (N_A, |d_A - d_B|) & \text{when } N_A = N_B \end{cases} \tag{3.26}$$

For a two level distance hierarchy, both addition and subtraction, reflects the theory of vectors in physics for vectors pointing in the same or opposite direction. When the vectors are pointing in the same direction ($N_A = N_B$), for the addition of two vectors ($A + B$), we simply add their magnitudes ($d_A + d_B$). For subtraction ($A - B$), we reverse the direction of the other vector and subtract their magnitudes (hence, $d_A - d_B$). When the vectors are pointing in the opposite direction ($N_A \neq N_B$), for vector addition ($A + B$), we subtract their magnitudes ($|d_A - d_B|$). The direction of the resultant vector is the same as the vector with the larger magnitude (hence different cases when $d_A \geq d_B$ and $d_A < d_B$ to decide the anchor value of the result). For the subtraction operation ($A - B$), we reverse the direction of the other vector and add their magnitudes (hence, $d_A + d_B$).

We can thus initialize the neuron weight vectors of the new map in a straightforward manner using the addition, subtraction and division operations on distance hierarchies. After initializing the new child map, the same process of training and growth is performed on this new map as described earlier. The process of hierarchical growth continues till every neuron in every map of the GHSOM satisfies the criterion in Equation 3.23.

This concludes the process of training the GHSOM using the distance hierachy for handling mixed attributes. Though we have limited ourselves to a distance hierarchy of

only two levels, this method can be extended to a distance hierarchy of more than two levels. As mentioned earlier, this is not in the scope of this thesis work and will be included as a part of future work. The current approach, however, still has its advantages over using more than two levels of distance hierarchy. Since all the categorical values are treated as equally distant from each other, we do not have to maintain the distance hierarchies, especially for storing the information of the *least common parent*. Each value is represented by its anchor value (categorical value) and an offset value. For performing the operations, there is no need to perform lookups in the distance hierarchy. If the anchor values of the operands are same, we treat them as points on same path; else we treat them as points on different paths. In case of distance hierarchy of more than two-levels, we need to query the distance hierarchy structure to identify the location of points with respect to their *least common parent* and to maintain this information, every attribute will need to store this information. Hence, the two-level hierarchy approach is efficient in terms of space and the handling of mixed attributes causes no significant overhead in terms of space and computations.

3.4 Conclusion

In this chapter, we presented the theory behind the traditional GHSOM. We proposed the *batch growth* approach to speedup the growth process of an individual map of the GHSOM. We also showed, how we can adopt the distance hierarchy based approach to enable the GHSOM to handle datasets with mixed attributes. Although, there are techniques proposed to extend the SOM for mixed attribute datasets, the GHSOM requires some more operations to be defined on the attributes such as computing the average, addition and subtraction. We defined these operations in terms of distance hierarchy points for the GHSOM, and thus enabling the GHSOM to handle mixed attribute datasets.

In the current setting, the GHSOM is capable of processing mixed attribute datasets but it is still ill-suited for massive datasets. It needs to be extended to run on a distributed cluster to handle massive datasets in parallel. In the next chapter, we will introduce the Map-Reduce variant of the GHSOM called MR-GHSOM. The MR-GHSOM extends the potential of the GHSOM to be used for the analysis of massive datasets, owing to its capability to run in parallel on a cluster of machines.

Chapter 4

MR-GHSOM

So far, we have outlined the usefulness of the GHSOM over the traditional SOM owing to its dynamic characteristics. The GHSOM can adapt its structure according to the underlying data and can also reveal the hierarchical relations in the data. In the last chapter, we extended the existing GHSOM to handle a mixed attribute dataset using the distance hierarchy technique. However, it is still difficult to use the GHSOM for large datasets as it requires multiple passes over the input. This becomes a difficult task on a single machine, especially when the dataset becomes so large that it cannot fit in the memory at a time.

In this chapter, we will formally introduce the Map-Reduce variant of the GHSOM which shall enable the GHSOM to be ported to a distributed computing environment to process large datasets in parallel. We have christened the variant as MR-GHSOM, which stands for Map-Reduced Growing Hierachical Self Organizing Map. Thus, our MR-GHSOM has the capability to process a large mixed attribute dataset.

This chapter is organized as follows. We first briefly discuss the stages in the GHSOM algorithm which can leverage parallelism. Also, we will discuss some challenges encountered with regards to the hierarchical expansion. Then we will formally introduce the Map-Reduce algorithm for each stage and the overall algorithm for the MR-GHSOM.

4.1 Scope of Parallelism in GHSOM

In this section, we will have a look at the stages in the GHSOM which can exploit parallelism on a cluster of nodes. Also we will briefly discuss the issues which could surface in the processing of massive datasets, especially during the hierarchical growth of the GHSOM.

4.1.1 GHSOM Stages for parallelizing on a cluster

We now proceed with the description, at a very high level, of the possible approaches for introducing parallelism in the GHSOM. If we concentrate on the training of an individual SOM which is the core of the GHSOM, one possible parallel approach would be splitting the neuron layer into regions and train the regions in parallel. However, this approach would be network intensive. The nodes working on different regions of the layer in parallel would need to communicate with each other to identify the winner neuron and convey the updates with respect to the neighbourhood factor. Another approach could be through exploiting the hierarchical structure of the GHSOM. Since the GHSOM is composed of multiple individual SOMs and each SOM works on an exclusive subset of data, we could train the SOM layers in parallel with each node responsible for an independent SOM. This parallelism can be introduced only after the layer at level-1 is trained (there is only one SOM layer to be trained in the GHSOM initially) and the parallelism could be introduced for the subsequently spinned off layers thereafter. The approach could run into out of memory issues if the data that needs to be processed by an individual SOM is too large to be processed on a single machine. This case could arise when we have deep hierarchies and small individual SOMs at each level.

Yet another approach for parallelism can be processing the dataset in parallel. The large dataset can be partitioned and distributed on the nodes of the cluster. Each partition of data can be processed in parallel on the nodes and used for training. This approach is flexible in terms of the size of the dataset. Smaller datasets would need less number of cluster machines. As the size of the dataset increases, more nodes can be added to the cluster to handle the increased size. Also, this approach is intuitive with the concept of the Map-Reduce programming model, and we are therefore adopting it.

Now, let us identify the stages of the GHSOM for mixed attributes where parallelism is feasible using the approach of processing the data in parallel.

Calculation of var_0: This is the first step in the GHSOM training and a good candidate for parallelism. The computation of the variance for numeric and categorical values can be performed simultaneously but using different techniques as described below.

For numerical attributes, we need to compute the mean of each numeric attribute with respect to the entire dataset. The mean is computed as the sum of values divided by the number of values. To compute the sum, the dataset can be split into chunks and the sum for each of these partitions can be calculated. This can be done in parallel. We can

then summate the intermediate sums for each partition and compute the total sum. The division of this sum by the total number of instances will yield the required mean value of the numeric attribute. The next step for computing var_0 is deducing the total variance, i.e. the sum of the squared distances of each input record from the mean divided by the total number of records. This requires another pass of the dataset. This can again be performed in parallel - compute the sum of squared distances for each partition of data and combine the intermediate sums to get the total sum of squared distances. The final variance value can be deduced by dividing this sum by the number of instances.

For categorical attributes, the variance value of an attribute is computed using the relative frequency of each value in the domain of the attribute. We can compute the frequency of each attribute value parallely for each partition of dataset. The intermediate sums of the value frequencies can then be collected and summated to get the total frequency for all the values of the attribute. Finally, we can divide the total frequency of each value by the total number of instances to get the required relative frequency.

The final variance of the dataset is the sum of the variance of the numerical attributes and the categorical attributes. Thus, the calculation of var_0 shall require atleast two passes over the dataset - one for calculating the mean (numeric attributes) and the total frequency of each distinct value (categorical attributes). The second pass of dataset would be required for the numeric attributes only to compute the distance from the mean for each input record.

Training of the SOM: This is the core step of the GHSOM and the most computationally expensive one. To parallelize this step, we can partition the dataset and distribute it across nodes in the cluster. The *Batch SOM* algorithm suits this approach. Remember that, in the Batch SOM, during each epoch the neuron layer is used in a read-only fashion to find the winner neuron. The neuron layer is only updated at the end of the epoch. Thus, we can share the read-only neuron layer across nodes in the cluster. Each node can train the layer on the local partition of data. The updates can be shared across the cluster at the end of the epoch and the neuron layer can be updated. This step shall require one pass of data for each epoch.

Evaluating the quality of the SOM for expansion: After the neuron layer is trained, it is evaluated for both, two-dimensional growth as well as hierarchical growth. For this, we need to compute the variance of each neuron with respect to the represented input instances. This requires identifying the best matching winner neuron for each input record

and its distance from the winner neuron's weight vector. This can be computed in parallel on the distributed partitions of dataset. The trained neuron layer can be shared across nodes and the distance from the winner neuron for each input record can be computed locally on each node. The total of all distances from the represented set of inputs for each neuron can then be accumulated from each node and the variance can be computed. This step shall require one pass of the input dataset.

Hierarchical Expansion: Once an individual SOM is trained and it satisfies the two-dimensional growth criterion (Equation 3.21), each neuron in the layer is evaluated according to the hierarchical growth criterion (Equation 3.23). For the neurons which do not satisfy the criterion, a new SOM layer is expanded from them. This new layer is trained only on input instances that are represented by the parent neuron. This step would again require a pass over the input dataset to identify the representing neuron for each instance. A concern to address at this stage would be, *"how do we store the represented subset of the dataset for each neuron, to be used for training later on"*. It would not be an optimal idea to store it in memory since the represented dataset may be too large to store in memory. Another approach is to store this dataset on the disk but we need to develop a mechanism to keep a track of the association between the parent neuron and the mapped set of instances.

The above mentioned stages in the GHSOM are suitable for parallel processing of a distributed dataset. In the next section, we will formally provide the Map-Reduce algorithms of the MR-GHSOM to realize the mentioned parallelisms.

4.2 MR-GHSOM Algorithms

In this section, we shall describe the Map-Reduce algorithms for training the MR-GHSOM on an input dataset in parallel. Since the MR-GHSOM requires multiple passes over the input dataset, *Apache Spark* was an optimal choice for implementing the algorithms. Using the notion of *Resilient Distributed Datasets(RDD)*, which is an abstraction of the shared-memory dataset mechanism, Apache Spark provides an improved performance in comparison to the popular Map-Reduce framework of *Hadoop*. Apache Spark is well-suited for iterative tasks such as ours in which multiple iterations over the input dataset are required.

We shall provide algorithms for every stage outlined before. Almost all algorithms contain two parts - *map* and *reduce* or *reduceByKey*. The *map* part executes in parallel on

the mapper nodes and emits an output or a set of outputs for each input record. Remember that, Apache Spark does not mandate the mapper and reducer tasks to emit key-value pairs. Each record in the input dataset is referred to as an instance. It is a representation of the attribute vector of the instance. The $emit()$ function in the algorithms that will follow, indicate the output produced by the $map()$ and $reduceByKey()$ functions on a per input record or per key basis respectively.

4.2.1 Map-Reduce for computing var_0

In this stage, we find the overall dissimilarity or variance in the dataset. The algorithm for computing the variance is as shown in Algorithm 2. The *map()* function on each node receives the subset of the data, one record at a time. For each input instance, it emits a pair of $(instance, 1)$. The second term will be used to compute the total number of instances in the dataset.

Algorithm 2 Map-Reduce job for computing variance of input data

```
1: function MAP(instance)
2:     emit(instance, 1)
3: end function

4: function REDUCE(pair_list = [(instance, count), ...])
5:     sum_instance ← initialize the zero instance
6:     total_instances ← 0
7:     for all tuple ∈ pair_list do
8:         add the tuple.instance to the sum_instance
9:         total_instances ← total_instances + tuple.count
10:    end for
11:    mean_instance ← compute the mean using sum_instance and total_instances
12:    return (mean_instance, total_instances)
13: end function
```

The *reduce()* function receives a list of all the pairs of $(instance, 1)$ from all the mapper tasks : *pair_list*. The *reduce()* function in Spark collates all the pairs to produce a single output and return the output to the driver. For numerical attributes, we need to compute the *mean* of all the instance attributes while for categorical attributes, we need to compute the *relative frequency* of each value of each categorical attribute. We first create an empty instance (*sum_instance*) - numerical attributes are initialized to 0 and categorical attributes are initialized by setting the frequency counts of each value in their respective domains to 0. Every pair in the *pair_list* is added to the *sum_instance* and the

value of the *sum_instance* is updated(line 8). For numerical attributes, values for each attribute are added to respective *sum_instance* attributes while in case of categorical attributes it increments the frequency count for the corresponding value of each attribute. The *total_instances* variable is also incremented for every record. The *reduce()* function then computes the mean instance using the sum of all instances (*sum_instance*) and the total count (*total_instances*). For numerical attributes, it divides the sum total for each attribute by *total_instances*. For categorical attributes, the relative frequency of each categorical value is computed by dividing their frequency counts by the *total_instances*.

The *mean_instance* is then used to compute the variance of the dataset at the driver. For categorical attributes, the variance can be calculated using the relative frequencies of all values for each attribute using Equation 3.15. For computing the variance for numerical attributes, another iteration of Map-Reduce is required. This is shown in Algorithm 3. The *map()* function computes the distance between each instance and the *mean_instance* computed before (line 2) and emits the distance value for each instance. The *reduce()* function computes the sum of the squares for these distances and emits the numerical variance. The output of this *reduce()* is the required variance for numerical attributes. Thus, the total variance var_0 of the dataset can be computed.

Algorithm 3 Additional Map-Reduce iteration for computing variance of numerical attributes

1: **function** MAP(instance)
2:     *distance* ← compute the distance betwen *mean_instance* and *instance*
3:     *emit*(*distance*)
4: **end function**

5: **function** REDUCE($distance_list = [distance_1, distance_2, \ldots]$)
6:     *sq_distance_sum* ← 0
7:     **for all** *distance* ∈ *distance_list* **do**
8:         $sq_distance \leftarrow distance^2$
9:         $sq_distance_sum \leftarrow sq_distance_sum + sq_distance$
10:     **end for**
11:     $numerical_variance \leftarrow sq_distance_sum/total_instances$
12:     **return** *numerical_variance*
13: **end function**

4.2.2 Training of individual SOM

To train an individual SOM on the input dataset, we used the Batch SOM algorithm. A high-level algorithm is as shown in Algorithm 4. Before the training begins, a SOM layer

with dimensions 2×2 is created. If this is the layer at level 1, then the neuron weight vectors are initialized randomly. For any other subsequent levels, the layers are initialized as per the heuristic mentioned in Section 3.3.1.

Algorithm 4 Training of SOM

```
1: function SOMTrain(dataset, neuron_map)
2:     dataset.cache()
3:     initialize neuron_map
4:     for current_epoch ← 0, epochs do
5:         neuron_updates ←Train SOM using Algorithm 5
6:         apply the updates to neuron_map
7:         current_epoch ← current_epoch + 1
8:     end for
9: end function
```

As discussed earlier, the training of the SOM involves a defined number of epochs over the dataset. Hence before we start with the training, we cache the dataset in memory using the *cache*() method of Spark RDDs (line 2 of Algorithm 4). During each epoch, the neuron layer is published to the mapper nodes. The new weight vectors for the neurons (*neuron_updates*) are computed in a Map-Reduce iteration shown in Algorithm 5. At the end of the epoch, the neuron updates are collected at the driver. The driver updates the neuron layer with the new weight vectors and publishes it to the mapper nodes for the next epoch.

The details of the algorithm's Map-Reduce iteration are given in Algorithm 5. The *map*() function (line 1) is the mapper task and executes on every mapper node of the cluster. Each mapper task processes a partition of the dataset. For each instance x in the partition, it computes the numerator and denominator part of Equation 3.19 for each neuron k in the SOM layer. It emits a corresponding *key-value* pair (line 6) for each neuron. *neuron.id* is the key part and value part is a pair of $(h_{ck} \cdot x, h_{ck})$.

The *reduceByKey*() function (line 9) represents the reducer task. It receives a list of pairs of numerator part (*num_part*) and denominator part (*den_part*) corresponding to the neuron identifer(*neuron_id*) from all the mapper tasks. It summates the numerator and the denominator parts to compute the total value of the numerator and the denominator of Equation 3.19 for the corresponding neuron. The division of the total numerator by the total denominator yields the updated weight vector for the neuron. The *reduceByKey*() function also emits a key-value pair (line 17) where the key is the neuron identifier and the value is the updated neuron weight vector. The updated neuron weight vectors are collected at the driver. The driver then applies the updates to the neuron layer.

Algorithm 5 Map-Reduce iteration in SOM Training

```
1: function MAP(instance, neuron_layer)
2:     find the winning neuron c in the SOM layer for instance
3:     for all neuron ∈ neuron_layer do
4:         num_part ← h_ck · x
5:         den_part ← h_ck
6:         emit(neuron.id, (num_part, den_part))
7:     end for
8: end function

9: function REDUCEBYKEY(neuron_id, val_list = [(num_part, den_part), ...])
10:     numerator ← 0
11:     denominator ← 0
12:     for all partial_update ∈ val_list do
13:         numerator ← numerator + partial_update.num_part
14:         denominator ← denominator + partial_update.den_part
15:     end for
16:     updated_weight_vector ← numerator/denominator
17:     emit(neuron_id, updated_weight_vector)
18: end function
```

Note that, in Apache Spark, *reduceByKey*() is an equivalent of the *reduce*() function in the conventional Map-Reduce. It works in a distributed manner on reducer nodes and each reducer node is responsible for an exclusive subset of keys or neurons. Moreover, the *reduceByKey*() function also behaves as a *combiner*. As described in Section 2.1.2, a combiner is like a mini-reducer which works on the mapper nodes and performs the reducer operation on the subset of *key-value* pairs generated on the mapper nodes locally. It helps in optimizing the network performance by reducing the amount of data to be transferred from the mapper to the reducer. In this case, the combiner aspect of *reduceByKey*() reduces the number of records transmitted from the mapper to reducer nodes from $n \times k$ to $m \times k$, where n is the number of records, m is the number of mapper tasks generated in the cluster and k is the number of keys or the number of neurons. This is a significant optimization since $m << n$.

4.2.3 Evaluating the quality of SOM for expansion

After a SOM is trained, it is evaluated for both, two-dimensional growth as well as hierarchical growth. This step is similar to the first step of computing var_0. The Map-Reduce algorithm for this step is provided in Algorithm 6.

Algorithm 6 Evaluating SOM quality using Map-Reduce

1: **function** MAP(instance)
2: $winner \leftarrow$ find the winning neuron c in the SOM layer
3: $distance \leftarrow$ compute the distance of $instance$ from c
4: $emit(winner.id, (distance, 1))$
5: **end function**

6: **function** REDUCEBYKEY($neuron_id, valList = [(distance, count), \ldots]$)
7: $sum_sq_dist \leftarrow 0$ ▷ sum of squared distances
8: $total_count \leftarrow 0$ ▷ count of represented instances
9: **for all** $tuple \in val_list$ **do**
10: $sum_sq_dist \leftarrow sum_sq_dist + tuple.distance^2$
11: $total_count \leftarrow total_count + tuple.count$
12: **end for**
13: $neuron_variance \leftarrow sum_sq_dist/total_count$
14: $emit(neuron_id, neuron_variance)$
15: **end function**

In the *map*() function, for every instance, we compute the distance from its winner neuron. The *map*() function emits a key-value pair (line 4) where the key is the identifier for the winner neuron and the value is a pair of $(distance, 1)$. The latter component of the pair is used for computing the total number of represented instances by the neuron. The $reduceByKey$() function receives a list of such pairs for every unique neuron identifier key. This function executes on the reducer nodes. It computes the total variance for the neuron with respect to the represented instances. For all the pairs in the list for a neuron, it computes the sum of the squared distances and the total of the represented instances. Dividing these two terms gives the variance of the represented instances at the neuron. The function emits a *key-value* pair where neuron identifier is the key and the value is the variance at this neuron. The values are collected at the driver which evaluates the neurons and their variances to identify the neurons for the two-dimensional growth and the hierarchical growth.

4.2.4 Hierarchical Expansion

After the two-dimensional growth is complete for a layer, each neuron's quality is evaluated for the hierarchical growth. Each identified neuron is expanded onto a new layer at the lower level in the hierarchy. Each new layer is trained only on the subset of instances represented by the corresponding parent neuron. So before the new layer can be trained, we need to filter out these subset of instances from the original dataset. Since the size of

these subsets may be too large to fit in the memory, we need a mechanism to keep a track of instances in the original dataset and their corresponding neurons in parent layer. A simple approach would be - parse the dataset every time a new layer is created to identify the subset of instances. However, this will require multiple passes over the dataset for every expanded layer.

For our MR-GHSOM, we adopted a technique that leverages the shared-memory aspect of the Spark RDD. An RDD can reside in the memory on the cluster nodes as long as the memory is available on the cluster machines. Also, RDDs provide a mechanism to filter out a subset of the data based on a criterion using the *filter()* transformation. The *filter()* transformation is executed in parallel on the cluster nodes. In the MR-GHSOM, we transformed an RDD of dataset instances into an RDD of tuples of the form - *(parent_layer, parent_neuron, instance)* as depicted in Equation 4.1. That is, we associate each instance with its representing neuron identifier and the layer identifier of the neuron.

$$dataset(instance) \rightarrow dataset(parent_layer, parent_neuron, instance) \tag{4.1}$$

We delegate the task of managing the storage of this transformed dataset to the Apache Spark library. It may store it in memory on the cluster machines or store it on disk if the memory is not sufficient or partially store it in memory and partially on the disk. This would depend on the memory available on the cluster machines. The algorithm for performing this mapping using only the mapper task is shown in Algorithm 7

Algorithm 7 Mapping instances to their winner neurons

1: **function** MAP($instance, expand_neuron_set, curr_layer_id$)
2: $\quad winner \leftarrow$ find the winning neuron c in the layer
3: $\quad$ **if** $winner \in expand_neuron_set$ **then**
4: $\quad\quad emit(curr_layer_id, winner.id, instance)$
5: $\quad$ **end if**
6: **end function**

The $map()$ function receives the set of neurons to expand for the hierarchical growth and the identifier of the current SOM layer. It finds the winner neuron for the instance and if the winner neuron is in the set of neurons, it emits a tuple of the form *(parent_layer, parent_neuron, instance)* (line 4). The final output of the $map()$ function is an RDD of such tuples in which an instance is associated to a parent layer and a parent neuron.

When a new layer is expanded from the neuron, we have the information about the parent neuron and the parent layer. We can use this information to filter out the subset of

data to be used for training this new layer using the *filter()* transformation. The training process for a new layer remains the same as described earlier.

4.2.5 Complete MR-GHSOM

Now we are positioned to join the pieces of the MR-GHSOM together and formulate the end-to-end algorithm as shown in Algorithm 8.

The first step of the algorithm is computing the overall variance in the dataset. Since we train several layers arranged in a tree hierarchy in a breadth-first manner, we maintain a queue to keep a track of the layers to process - *parent_queue*. Every layer is associated with an identifier - *curr_layer_id*. Initial contents of the queue for first layer at level 1 is a pair $(0, 0)$ indicating the parent layer as level-0 layer and parent neuron as the only neuron in level-0 layer. Next, we transform the dataset (line 7) for the future hierarchical expansion as described in Section 4.2.4 into a dataset of tuples as shown in Equation 4.1.

For every entry in the queue, we create a SOM layer (line 13) and train it (line 17) using the Algorithm 4. Before the training begins, we filter out the instances, as required, for the current layer using the parent layer id and parent neuron id from the queue (line 11). After the iteration of the SOM training, we evaluate the layer for the two-dimensional growth (lines 19-24). The SOM is grown in two-dimensions if the criterion in Equation 3.21 is not satisfied and the grown layer is trained again. After the layer satisfies the Equation 3.21, we identify the neurons for expanding onto a new layer of the SOM (*expand_neuron_set*) using the criterion in Equation 3.23. The next step in the process (lines 26-33) is mapping the instances in the current dataset to their winner neuron in the current layer for the reasons described in Section 4.2.4. Note that, the dataset for the current map is cached for the entire duration of training, the two-dimensional growth, and the mapping of the instances for training the SOM in the next level. This allows faster processing of dataset over multiple passes.

Thus, we described our proposed Map-Reduce variant of the GHSOM called MR-GHSOM. The MR-GHSOM not only extends the capability of the GHSOM to handle datasets containing mixed attributes but also extends its usability for processing massive datasets in a distributed computing environment.

Algorithm 8 MR-GHSOM

```
1: function MR-GHSOM(dataset)
2:     compute var_0 using Algorithm 2
3:     parent_queue ← create queue to track parent layer and neuron of current layer
4:     parent_layer ← 0                                              ▷ For level-0 layer
5:     parent_neuron ← 0                                     ▷ Only neuron in level-0 layer
6:     parent_queue.enqueue(parent_layer, parent_neuron)        ▷ for the first level layer
7:     mapped_dataset ← transform dataset as described in Equation 4.1
8:     curr_layer_id ← 0
9:     while parent_queue is not empty do
10:        (curr_parent_layer, curr_parent_neuron) ← parent_queue.dequeue()
11:        curr_dataset ← filter instances from mapped_dataset for current layer
12:        curr_dataset.cache()
13:        curr_layer ← create a new layer of 2 × 2 neurons
14:        curr_layer_id ← curr_layer_id + 1
15:        is_2d_growth ← false
16:        repeat
17:            train curr_layer SOM using Algorithm 4
18:            evaluate quality of curr_layer using Algorithm 6
19:            if curr_layer does not satisfy 2-D growth criterion Equation 3.21 then
20:                grow the layer
21:                is_2d_growth ← true
22:            else
23:                is_2d_growth ← false
24:            end if
25:        until is_2d_growth is true
26:        expand_neuron_set ← create empty set
27:        for all neuron ∈ curr_layer do
28:            if neuron does not satisfy hierarchical growth criterion Equation 3.23 then
29:                add neuron to expand_neuron_set
30:                parent_queue.enqueue(curr_layer_id, neuron.id)
31:            end if
32:        end for
33:        mapped_dataset ← associate instances to their winner neurons (Algorithm 7)
34:        save the current SOM
35:        curr_dataset.uncache()
36:    end while
37: end function
```

4.3 Conclusion

In this chapter, we described our MR-GHSOM more formally while outlining the overall algorithm as well as the component algorithms in detail. We made use of the Map-Reduce programming paradigm using Apache Spark enabling the MR-GHSOM to process massive dataset on a distributed cluster of nodes. Owing to the notion of RDDs and caching of RDDs, Apache Spark enables faster processing of a large dataset, especially for tasks which requires multiple passes over the dataset. We leveraged these concepts of Spark for our implementation of the MR-GHSOM.

In the next chapter, we shall describe the experiments we performed on the MR-GHSOM to evaluate its clustering behaviour and its ability to mirror the hierarchical structure. The chapter shall ascertain the usefulness of the MR-GHSOM in clustering and data analysis of large mixed attribute datasets.

Chapter 5

Experiments

So far, we have provided a theoretical understanding of how the MR-GHSOM works. In the last chapter, we provided the complete end-to-end Map-Reduce based algorithm of the MR-GHSOM. In this chapter, we will further ascertain its usefulness by way of testing it on some popular known datasets.

This chapter is organized as follows. Initially, we will discuss the approach that we adopted to evaluate the results generated by the MR-GHSOM. We then discuss the testing of the MR-GHSOM with regards to its two-dimensional growth property and the ability to represent the input data on a single SOM layer. We also evaluate the impact of the *batch growth* technique that we proposed in Section 3.2. Next, we study the results of the MR-GHSOM in terms of hierarchical clustering. Lastly, we will run our algorithm on the Census-Income dataset and perform an analysis of the results.

5.1 Configuration and precursor information

All the experiments were conducted on a cluster of 2 worker nodes. The data was stored on the HDFS. Each node had 4GB RAM and 1 core. The amount of memory on each node needs to be sufficient to cache the data partition in memory on each node. This small cluster was sufficient for our Census-Income dataset ($\sim 300K$ records). The working model of the MR-GHSOM on a small cluster for a $\sim 300K$ record dataset can be extended for a larger dataset by using a more powerful and larger cluster as required. The experiments are aimed to prove that the MR-GHSOM is capable of processing a large mixed attribute dataset in a distributed environment and still producing satisfactory results.

For all datasets used in these experiments, the attributes were normalized. Numeric attributes were normalized to a range of 0.0 to 1.0. For each categorical attribute, a two-level distance hierarchy was created and the link weight was set to 0.5 (making the maximum path length or distance between any two leaf nodes as 1.0).

The results of experiments are mostly in the form of SOM layers generated by the MR-GHSOM. While describing a generated map, a reference to a cell in the SOM layer is made using the notation $[x, y]$ where x is the row number and y is the column number of the neuron cell in the map. Also rows and columns are numbered starting from 0. $[0, 0]$ represents the cell in the top left corner i.e. row = 0 and column = 0.

5.2 Evaluation Approach

With reference to the SOM algorithm, the most common evaluation metric used to assess the quality of the map is *quantization error*. *Quantization error* indicates the quality of the data representation by the SOM layer and is calculated as the average distance between each input instance and its winner neuron. It decreases with the increase in the SOM layer size. This measure is not applicable to dynamic variants of the SOM like the GHSOM or our MR-GHSOM. The GHSOM or MR-GHSOM are designed to grow till they satisfy a criterion governed by the *mean quantization error* or *variance* respectively. The *mean quantization error* or *variance* are also indicators of the quality of data representation by the map and in the GHSOM or MR-GHSOM, we can control this quality using the parameters τ_1 and τ_2. Owing to this, such dynamic variants of the SOM are evaluated based on empirical evidence only. Accordingly, we shall also use empirical results, in most cases, to support the MR-GHSOM. The empirical results will be shown using the U-Matrix and component planes for the attributes. For datasets having labelled classes, we shall show the distribution of class labels throughout the map. Note that, in maps showing class label distribution, cells without any labels represent neurons which were not selected as winner neurons for any input instance.

To evaluate the topology preservation property of a SOM layer, we used the *topographic error* metric. It is a ratio of input instances for which the winner neuron, i.e. the best matching neuron and the next best matching neuron are not adjacent. It has a value in the range of 0 and 1, where 0 indicates the best topology preservation while 1 indicates the worst. The topographic error is an indicator of the quality of projection of high-dimensional dataset onto a two-dimensional grid of neurons. Topology preservation is not useful for small maps since for a small map the probability of finding the best and the second best

winner neuron to be adjacent, is high. Hence, we shall be evaluating only large maps with respect to the *topographic error*.

5.3 Evaluation of two-dimensional Growth

5.3.1 Focus of Evaluation

In this section, we will evaluate how our MR-GHSOM performs in terms of modelling an individual SOM layer on the input data while growing the SOM layer dynamically. To perform this evaluation, we shall compare the results of the MR-GHSOM with the results generated by a sequential SOM. We created a sequential implementation of the SOM, that runs on a single machine and handles a mixed attribute dataset using the distance hierarchy approach as presented in [19]. Remember that, the sequential SOM has a static size of the layer which needs to be specified before the training begins. To make the comparison fair, we set the dimensions of the sequential SOM layer same as that of the layer generated by the MR-GHSOM while growing dynamically. We run our MR-GHSOM and the sequential SOM on some popular classification datasets from UCI[29]. We then compare their results in terms of the U-Matrix generated for the trained SOM layers. Since we are using classification datasets, we also present the class label distribution generated by the two versions of the SOM: sequential SOM and MR-GHSOM. Note that, we are not trying to demonstrate whether the MR-GHSOM is better or worse than the sequential SOM. We expect these experiments to provide us an assurance that the MR-GHSOM produces similar results to the sequential SOM while running in parallel on a distributed cluster and growing the SOM layer dynamically.

To perform the evaluation, the datasets from UCI[29] we shall be using are Iris, Wine, Mushroom and Credit. Iris and Wine are purely numerical datasets while Mushroom is a purely categorical dataset. Lastly, Credit dataset is a mixed attribute dataset containing both numeric and categorical attibutes. The details of datasets are shown in Table 5.1.

Dataset	Type of attributes	# of attributes	# of instances	# of classes
Iris	Numerical	4	150	3
Wine	Numerical	13	178	3
Mushroom	Categorical	22	8124	2
Credit	Mixed	15	690	2

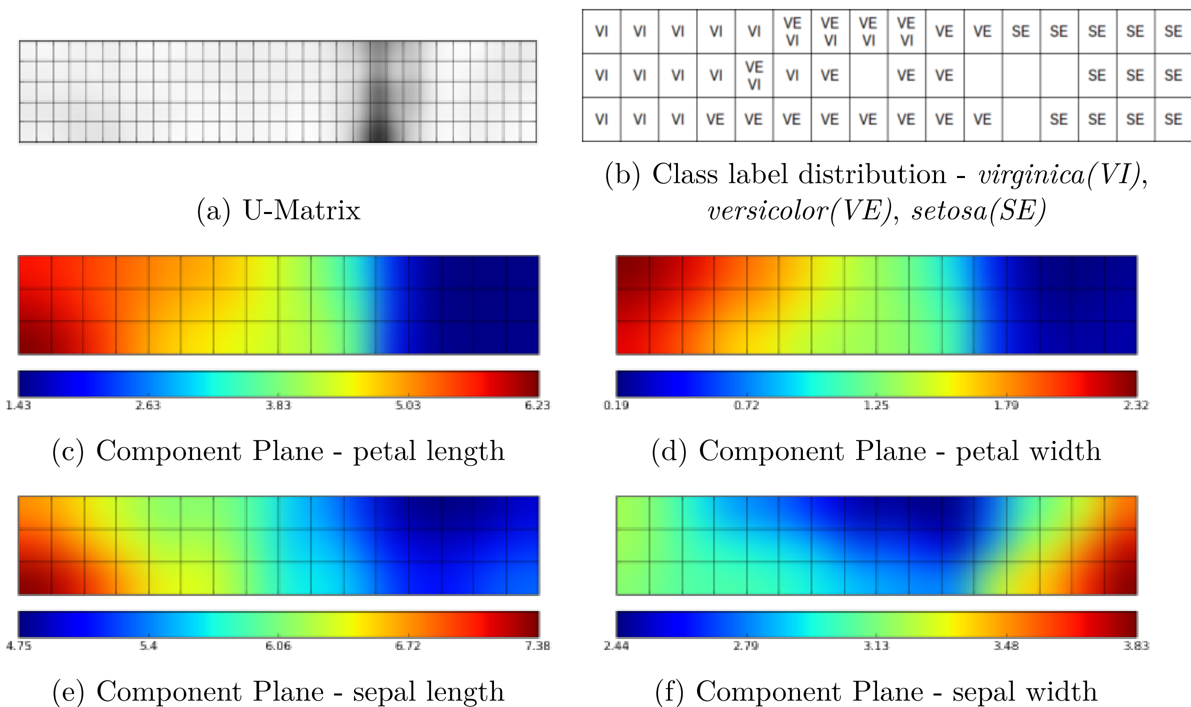
Table 5.1: Datasets for two-dimensional growth evaluation

5.3.2 Evaluation

Iris dataset

We shall first describe the results for the Iris dataset in detail with the help of the U-Matrix and component planes generated for it. We will also use this dataset's results to show, how we can leverage the U-Matrix and the component planes in unison to understand the characteristics of the data. The Iris dataset has three classes namely Iris-setosa, Iris-virginica and Iris-versicolor. Iris-setosa is distinguishable from the other two class instances; however, the remaining two class instances are not separable from each other [30].

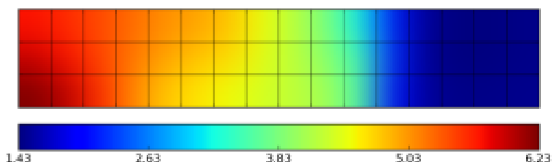
The parameters τ_1 and τ_2 were set to 0.05 and 1.0 respectively. Since we are interested in only two-dimensional growth, τ_2 was set to a high value to avoid the expansion into lower levels. Figure 5.1 shows the U-Matrix, component planes and class label distribution for Iris dataset for the SOM layer generated by the MR-GHSOM. The resulting SOM layer had the dimensions of 3×16 neurons.



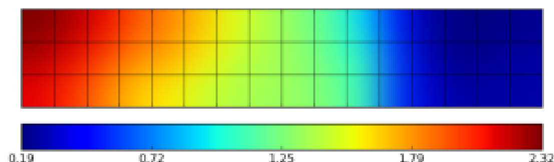
(a) U-Matrix

VI	VI	VI	VI	VI	VE VI	VE VI	VE VI	VE VI	VE	VE	SE	SE	SE	SE	SE
VI	VI	VI	VI	VE VI	VI	VE		VE	VE				SE	SE	SE
VI	VI	VI	VE	VE	VE	VE	VE	VE	VE	VE		SE	SE	SE	SE

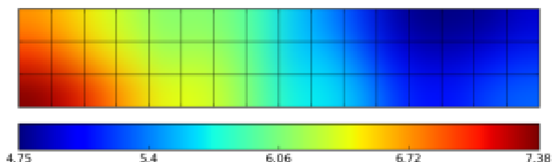
(b) Class label distribution - *virginica(VI)*, *versicolor(VE)*, *setosa(SE)*



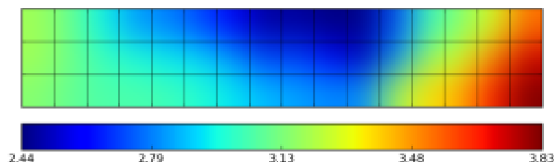
(c) Component Plane - petal length



(d) Component Plane - petal width



(e) Component Plane - sepal length



(f) Component Plane - sepal width

Figure 5.1: Results of MR-GHSOM on the Iris dataset

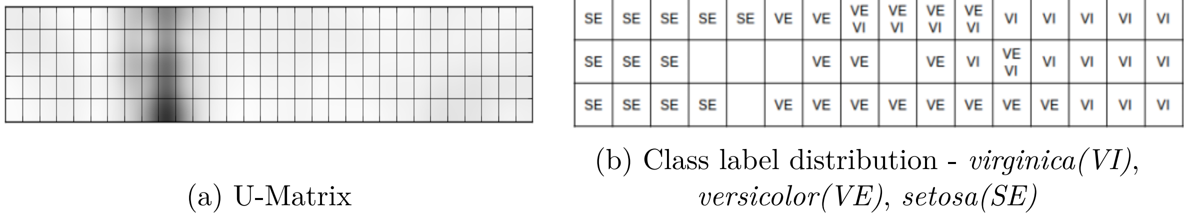
Analysis of the SOM layer: The U-Matrix for the SOM layer generated by the MR-GHSOM (Figure 5.1a) shows two distinct clusters separated by a dark area at column 22 (area near the two-thirds of the U-Matrix from left). The class label distribution shows

that these two regions are related to the classes iris-virginica and iris-versicolor on the left of the dark separator, and the class iris-setosa on the right. The class label distribution also shows that the classes on the left are grouped into two groups of iris-virginica and iris-versicolor. The U-Matrix does not reflect a distinct separator between iris-virginica and iris-versicolor owing to the fact that vectors associated with these classes are not separable.

If we move our attention to the component planes, we see that the attributes - petal width and petal length - are related to each other. Further their distribution is similar to the sepal length attribute. We also see that iris-setosa is distinguished from other classes mostly on the attribute of petal length and petal width. Iris-setosa has lower values of petal length and width while higher values of sepal width. The results of the MR-GHSOM on the Iris dataset helps us understand, how we can benefit from the component plane visualization to understand the relation between attributes in the data distribution.

Thus, the generated SOM layer can be used not only to identify clusters in the dataset, but also for data analysis purposes to study the correlation between different attributes.

Comparison with sequential SOM: We trained the sequential SOM with a layer size of 3×16 on the same dataset. The U-Matrix and class label distribution is shown in Figure 5.2. As we can see, the MR-GHSOM generated similar results for the Iris dataset - the orientation of the SOM layer generated by the MR-GHSOM is a mirror image of the one generated by the sequential SOM. Both results show identical distribution of the class labels in the dataset, thus verifying the results of the MR-GHSOM on the Iris dataset.



(a) U-Matrix

(b) Class label distribution - *virginica(VI)*, *versicolor(VE)*, *setosa(SE)*

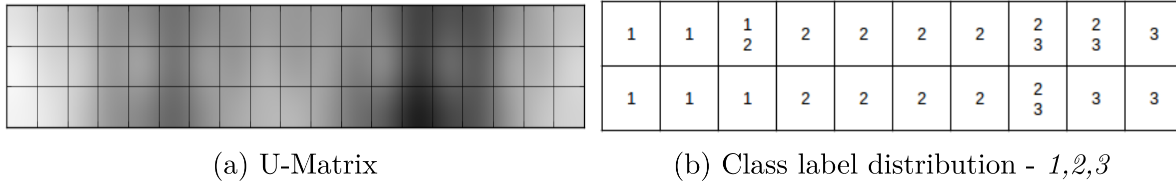
Figure 5.2: Results of the sequential SOM on the Iris dataset

Next, we modelled the two-dimensional SOM layers for the Wine, Mushroom and Credit datasets. We will only show the U-Matrix and class label distribution for these datasets since they are sufficient to verify the results for this section of experiments.

Wine Dataset

The Wine dataset is a dataset with only numeric attributes. This dataset has three classes - $1, 2, 3$ - for the wine instances. For this dataset, the value of τ_1 was set to 0.4 while τ_2

was again set to 1.0 to avoid the expansion into lower level layers. The resulting SOM had 2×10 neurons. The results are shown in Figure 5.3.

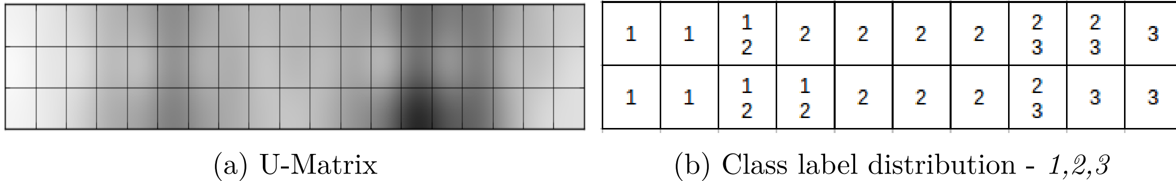


(a) U-Matrix (b) Class label distribution - *1,2,3*

Figure 5.3: Results of MR-GHSOM on the Wine dataset

Analysis of the SOM layer: The U-Matrix (Figure 5.3a) shows a clear separation between the three classes. As depicted in the class label distribution (Figure 5.3b), the cluster on the left corresponds to the class *1*, the one in the middle represents class *2* and the right one represents class *3*.

Comparison with sequential SOM: We trained a SOM layer of size 2×10 for the Wine dataset using the sequential SOM. The results of training are shown in Figure 5.4. We can see that the U-Matrix for this SOM layer (Figure 5.4a) is identical to the one generated by our MR-GHSOM. It also shows clear separation between the three classes. The class label distribution shows some additional overlaps of the class labels of *1* and *2*; however, this is acceptable as the results of clustering can vary with every execution.

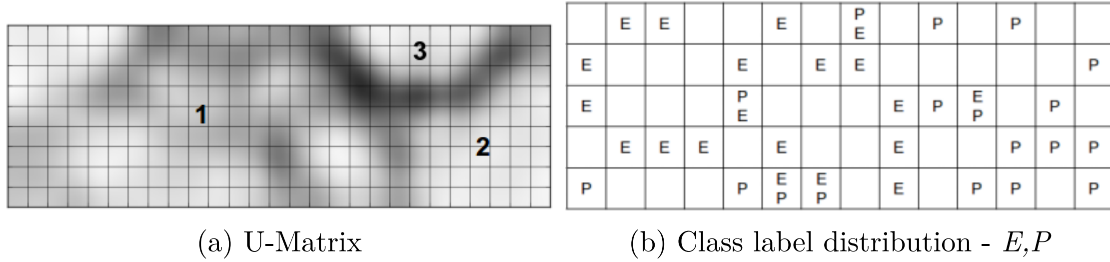


(a) U-Matrix (b) Class label distribution - *1,2,3*

Figure 5.4: Results of the sequential SOM on the Wine dataset

Mushroom dataset

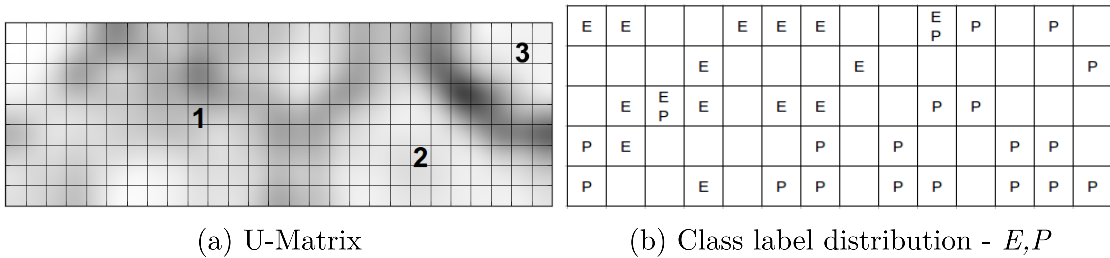
Having shown the results for the numericals datasets, we now test our MR-GHSOM for a purely categorical dataset - Mushroom. The parameter τ_1 was set to 0.45 and τ_2 was given 1.0 to avoid the expansion into lower levels. All the boolean attributes were treated as categorical attributes having two values in their domain. This dataset contains two class labels namely *E* (edible) and *P* (poisonous). The SOM layer for the Mushroom dataset grew from a four neuron map to attain dimensions of 5×14. The U-Matrix and the class label distribution generated by MR-GHSOM is shown in the Figure 5.5.



(a) U-Matrix (b) Class label distribution - *E,P*

Figure 5.5: Results of MR-GHSOM on the Mushroom dataset

Analysis of the SOM layer: The class label distribution (Figure 5.5b) shows that the class labels are clustered together in the map (except for a few P labelled cells in the group of E). The U-Matrix (Figure 5.5a) depicts a separation between these two groups. We have labelled the cluster representing class E as 1. The clusters labelled 2 and 3 represent the group of P instances. However, U-Matrix shows more clusters within the groups of E and P. In particular, it shows two prominent clusters in the group of class P, labelled 2 and 3 in the U-Matrix. This shows that the instances within the two classes can further be split into subsequent groups. Thus, the analysis of Mushroom dataset reveals some additional clusters that are not indicated by the classification labels.



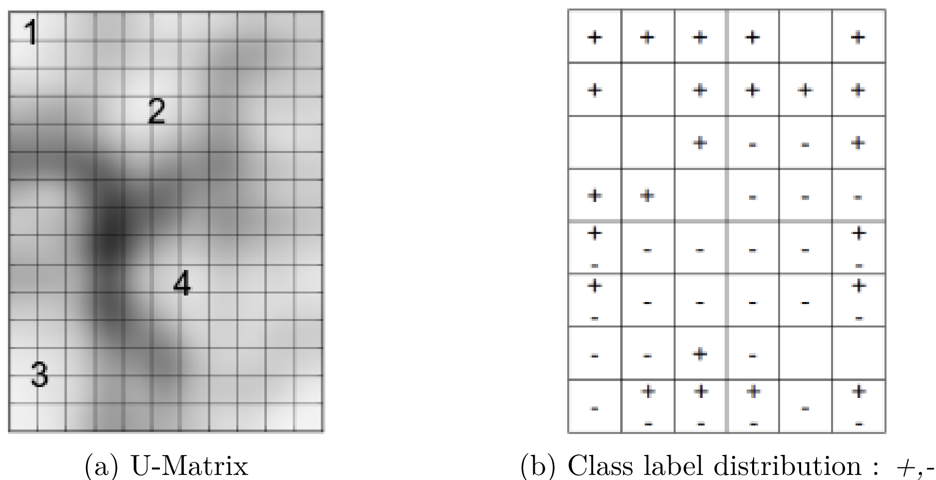
(a) U-Matrix (b) Class label distribution - *E,P*

Figure 5.6: Results of the sequential SOM on the Mushroom dataset

Comparison with sequential SOM: We trained a SOM layer of size 5×14 using the sequential SOM. The results are shown in Figure 5.6. The U-Matrix (Figure 5.6a) reveals the same clusters as we saw in case of the MR-GHSOM. The orientation of these clusters is different in this SOM layer. Again, this is acceptable as the clusters can be oriented differently across multiple trainings of the SOM. For the sake of comparison, we have shown the clusters from the MR-GHSOM results in this U-Matrix using the same labels. As long as the similar clusters lie closer to each other, the results are acceptable. The clusters labelled 2 and 3 are closer to each other in both U-Matrices, and hence we can consider the generated SOM layers as similar. Thus, we can conclude that the results generated by the MR-GHSOM are identical to those generated by the sequential SOM.

Credit dataset

Lastly, we test a mixed attribute dataset - Credit. It contains two classes for people with positive credit ("+") and others with negative credit ("−"). For this dataset, we again considered all the boolean attributes as categorical attributes. The value for parameter τ_1 was set to 0.5 and τ_2 was again set to a high value (1.0) to avoid the hierarchical expansion. The map grew to a size of 8×6. Figure 5.7 shows the U-Matrix and the class label distribution for the map generated by the MR-GHSOM.



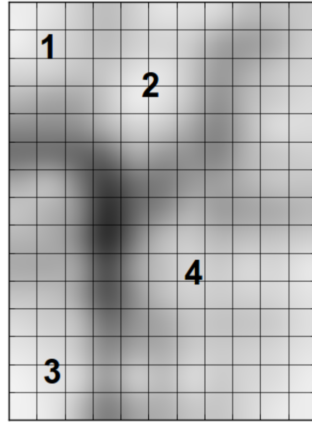
(a) U-Matrix

(b) Class label distribution : *+,-*

Figure 5.7: Results of MR-GHSOM on the Credit dataset

Analysis of the SOM layer: The label distribution (Figure 5.7b) shows that the MR-GHSOM did group the labels together: "+" is clustered at the top while "−" is clustered towards the bottom of the map. However, the U-Matrix (Figure 5.7a) depicts different groupings. It does depict a dark separator between the two classes starting at cell $[5, 0]$. If we concentrate on the lighter regions of U-Matrix indicated by labels 1 to 4, we can associate these regions to pure groups of labels in the class label distribution. For example, region 4 corresponds to a cluster of purely "−" labels while region 1 corresponds to a pure group of "+" labels.

Comparison with sequential SOM: For comparison, a SOM layer of size 8×6 was trained using the sequential SOM. The results are shown in Figure 5.8. Both, the U-Matrix and the class label distribution for this layer, are almost identical to the ones shown for the layer trained using the MR-GHSOM. In fact, for the credit dataset, the orientation of the clusters - $1, 2, 3, 4$ - is same for the layers generated by the MR-GHSOM and the



(a) U-Matrix

+	+	+	+		+
+		+	+	+	+
		+	-	-	+
+	+		-	-	-
-	-	+ -	-		+ -
+ -	-		-	-	-
-	-	+ -	-	-	+ -
-	+ -		+ -	-	+ -

(b) Class label distribution : +,-

Figure 5.8: Results of the sequential SOM on the Credit dataset

sequential SOM. Thus, we can conclude that the MR-GHSOM does a fair job at clustering of mixed attribute dataset as well.

Evaluation of topographic error:

To evaluate the topology preservation property of the MR-GHSOM, we computed the topographic error for the layers generated using the MR-GHSOM and the sequential SOM on the above mentioned four datasets. The topographic error for the SOM layers is mentioned in Table 5.2. All topographic error values for the MR-GHSOM layers have low values, corroborating the fact that the SOM layers generated by the MR-GHSOM preserved the topology of the data properly. Moreover, the topographic error for both, the sequential SOM and MR-GHSOM, are comparable. The difference in the values of topographic error is not a definitive indicator that one algorithm performed better than the other. The topographic error varies across different executions of the SOM training. As long as the values are low, we can be assured that the algorithm does a satisfactory job at preserving the topology of the data.

Dataset	MR-GHSOM	Sequential SOM
Iris	0.146	0.173
Wine	0.078	0.079
Mushroom	0.002	0.011
Credit	0.066	0.059

Table 5.2: Topographic error: MR-GHSOM versus sequential SOM

5.3.3 Learnings from the evaluation

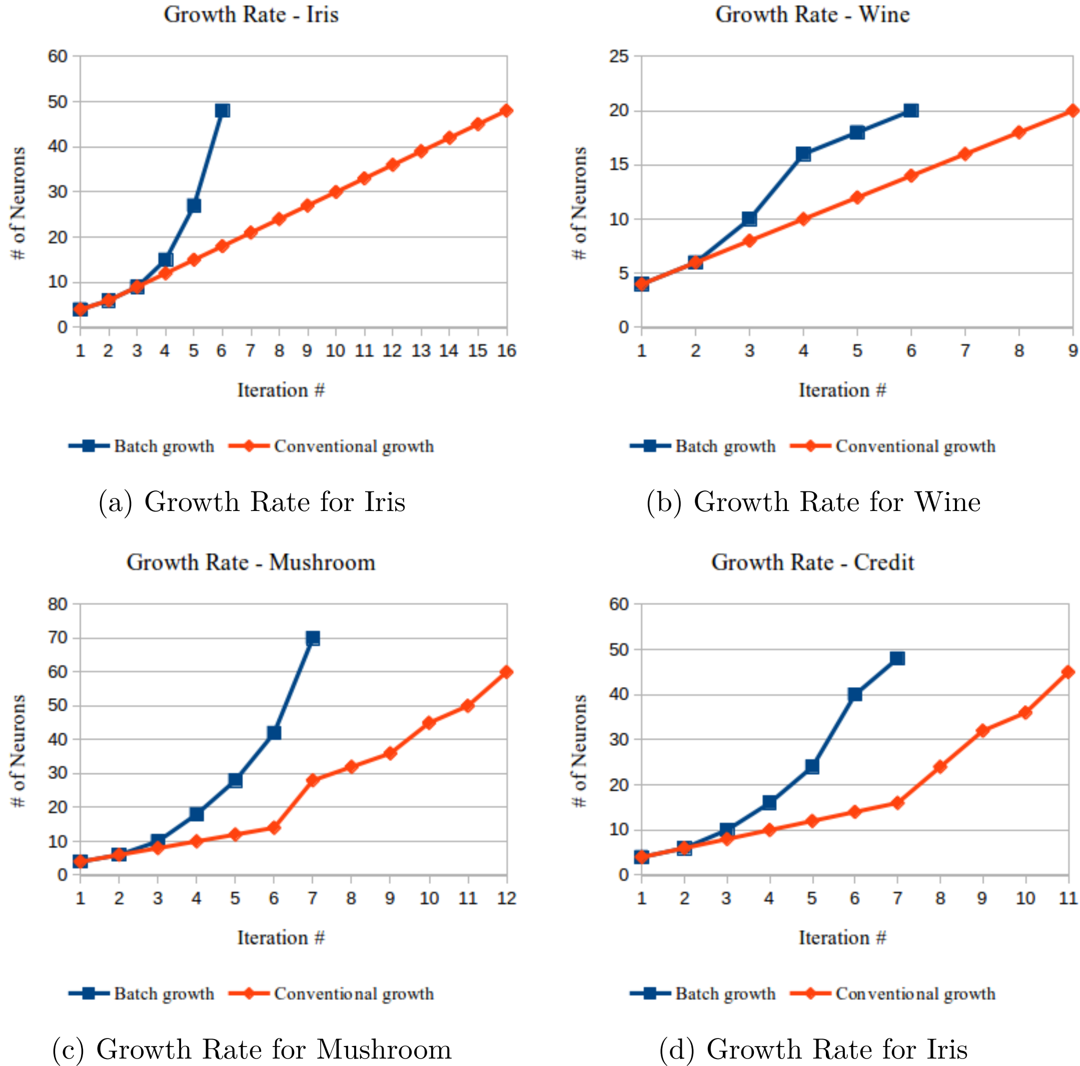
The above experiments were not meant to indicate that the MR-GHSOM is better or worse than the sequential SOM. The SOM results may vary across multiple executions. The core SOM algorithm is the same for both. Hence, the minor discrepancy in some results that we observed above is acceptable. The focus of these evaluations was to provide an assurance that the MR-GHSOM produces comparatively identical results to the sequential SOM, in spite of growing dynamically and running in parallel on a cluster. Thus, we can confirm from the results above that the MR-GHSOM does a satisfactory job of representing data on an individual SOM layer, and also maintains the topology of the input dataset as indicated by the low values of the *topographic error*.

5.4 Evaluation of the *batch growth*

In Section 3.2, we proposed a modification in the conventional GHSOM to speed up the two-dimensional growth process. We called it the *batch growth* approach and used it in MR-GHSOM. In this section, we will provide results which show that this approach is indeed an useful improvement for the dynamic two-dimensional growth process.

5.4.1 Focus of Evaluation

Every growth iteration in the conventional GHSOM is followed by an iteration of the SOM training. This SOM training requires multiple passes over the input dataset and is the most computationally expensive phase of the overall GHSOM training. Hence, any reduction in the number of SOM training iterations would lead to a reduction in the overall training time of the GHSOM. In this section, we try to evaluate how the *batch growth* approach helps us in reducing the number of growth iterations for an individual SOM layer. We will compare the number of iterations required by the MR-GHSOM (using the *batch growth* approach) and by the conventional GHSOM to reach the final required size of a SOM layer. We would also like to confirm that the *batch growth* should not result in a SOM layer with too many unnecessary neurons. We will present our results with respect to the four datasets mentioned in the last section.



(a) Growth Rate for Iris

(b) Growth Rate for Wine

(c) Growth Rate for Mushroom

(d) Growth Rate for Iris

Figure 5.9: Evaluation of the *batch growth* approach

5.4.2 Evaluation

Figure 5.9 shows the growth rate of the SOM layers on the four datasets. The X-axis denotes the growth iteration number and the Y-axis represents the number of neurons in the current SOM layer. We can see that the *batch growth* approach reaches the target SOM size comparatively quickly than the conventional one row/column at a time approach of the GHSOM. For example, in case of the Credit dataset, the two-dimensional growth criterion was satisfied in just 7 iterations of growth for *batch growth* while it took 11 iterations for the conventional approach. Also, the *batch growth* approach does not have any significant adverse effect with respect to unnecessary neurons being added to the SOM layer. The final size of the SOM in all cases is approximately similar to the size attained through the

conventional GHSOM's growth process.

To give a picture of the performance gains achieved through the *batch growth* approach, assume that the number of epochs required for the SOM training is chosen as 100 for the Mushroom dataset. By the time, the conventional GHSOM attains the desired size of the SOM layer, it would have parsed over the Mushroom dataset $12 \times 100 = 1200$ times. As against this, the MR-GHSOM using the *batch growth* approach would need $7 \times 100 = 700$ passes over the dataset. Thus, in total we would save 500 passes over the dataset, while still achieving the desired quality of data representation by the SOM layer.

5.4.3 Learnings from the evaluation

From the results, we can confirm that the *batch growth* approach requires lesser number of growth iterations to attain the desired final size of the SOM layer. Moreover, it does not add too many unnecessary neurons to the SOM layer during this bulk growth.

This reduction in the number of growth iterations can prove useful in reducing the number of passes over the input dataset and hence, reducing the overall training time of the GHSOM process. Moreover, this lessening in the number of passes over the input dataset is crucial when the underlying dataset is massive.

5.5 Evaluation of Hierarchical Growth

So far, we have established that the MR-GHSOM does a satisfactory job with respect to the two-dimensional growth. We now evaluate the MR-GHSOM for its hierarchical growth property.

5.5.1 Focus of Evaluation

In this evaluation, we will analyze the hierarchical behaviour of the MR-GHSOM. Since there does not exist any variant of the GHSOM capable of handling mixed attributes, we will not compare our results with any other algorithm. To support our MR-GHSOM, we will analyze the empirical results in this section and provide the reasoning behind the behaviour of the MR-GHSOM and the orientation of the generated SOM layers arranged in a tree-like hierarchy.

Here, we try to show that the MR-GHSOM produces layers arranged in a hierarchy such that:

- The layers lower in the hierarchy represent the data at a finer granularity than the data represented by the parent neuron in the parent layer.
- The orientation of the child layer preserves the orientation of the parent neuron and its neighbours in the parent layer.

To evaluate the MR-GHSOM for its hierarchical growth, we used the Zoo[29] dataset. The dataset consists of 101 instances of animals. Each instance has 16 attributes (excluding the name of animal and the type attribute for the class). We treated all the boolean attributes in the dataset as categorical attributes. Thus, the dataset had in 15 categorical attributes and 1 numeric attribute (legs). To understand the hierarchical structure produced by the MR-GHSOM, we will present the label distribution of the SOM layers.

5.5.2 Evaluation

Figure 5.10 shows the resulting GHSOM structure. The first layer of the SOM is shown by the blue coloured grid. The lower level layers are projected into the first layer (shown by small red coloured grids within the major blue coloured grid).

The findings from the results generated by the MR-GHSOM are as follows:

- The first layer of the generated SOM had dimensions of 5×2 (shown by blue grid). It splits the dataset into two major groups - mammals (cells $[0, 0]$, $[0, 1]$ and $[1, 0]$) and non-mammals (rows 2 to 4). The last row represents the bird family. Also, we can see that the column 0 is dominated by the aquatic creatures, while the cell $[3, 1]$ is dominated by the insect family.
- The cell $[0, 1]$, representing land animals, is further expanded into a new layer of size 2×5. In this sub-layer, columns 0 and 1 represent predators while the remaining are non-predators. We also find that, in the sub-layer, the orientation of the predator related cells is towards the cell $[0, 0]$ containing aquatic predators - mink, seal and sealion. This ascertains that the MR-GHSOM preserves the orientation of the sublayers with respect to the parent neuron and its neighbours in the parent layer.
- The cell $[2, 0]$ of the first layer is expanded into a layer of 3×2. The groups in this layer are created on the basis of catsize, predator and breathes attributes. Again, the cell $[1, 0]$ of this sublayer is oriented towards the larger animals (cell $[1, 0]$ of layer 1).
- The subcluster for cell $[3, 0]$ is split into three groups based on the number of legs.

Figure 5.10: Hierarchical SOMs for zoo dataset

- Next, we will move our attention to the subcluster of the cell $[3, 1]$. The second row in this subcluster is devoted to aerial insects (attribute: airborn). The second row is further split into two groups based on the presence or absense of hair.
- The last row of the first layer SOM is devoted to the bird family. The subcluster in cell $[4, 1]$ is divided on the basis of the predator and catsize attribute. The second column of this subcluster is dominated by small birds (catsize $= 0$) which are not predators.

5.5.3 Learnings from the evaluation

From the results of the MR-GHSOM on the Zoo dataset, we can ascertain that the MR-GHSOM does a suitable job in representing the hierarchical relations in the input data. We saw that, the MR-GHSOM represented the data at a coarser granularity in the top layer, and then provided finer separation between the represented instances in the lower

level layers. Moreover, it also preserves the orientation of the lower level layers with respect to the parent neuron and its neighbours in the parent layer.

5.6 Experiments on the Census Dataset

The experiments so far, give us an assurance that the MR-GHSOM is indeed capable of producing meaningful results with respect to modelling data in the hierarchy of the SOM layers. Further, the visualizations of the U-Matrix and component planes also prove useful in describing the clusters or the results generated. Having established the confidence in the MR-GHSOM, we now proceed to our analysis of the target dataset - Census. The analysis of a Census dataset can have applications in election campaign steering or market segmentation. For instance, in election campaign steering, the generated GHSOM can help a campaign manager to view the entire voter population at different levels of granularity. The neurons in every SOM layer in the hierarchy represent a summary or a prototype of a set of similar entities in the voter population. The neurons that lie in the interesting clusters in the current layer, can be expanded into new layers to view finer details. The clusters can then be identified using the U-Matrix visualization. Once identified, the component planes can be used to understand the properties of these clusters.

5.6.1 Focus of Evaluation

In this part of our experiments, we will showcase the use of the MR-GHSOM on a large dataset with mixed attributes. The focus of this evaluation is:

- to show how the MR-GHSOM is capable of summarizing the data at different levels of granularity
- to show how we can use the results of the MR-GHSOM to understand the data
- to demonstrate the use of the SOM visualizations to, not only depict the clusters within the data and understand the cluster attributes, but also to understand the correlations between different attributes. Together, these tools form a useful combination to perform the necessary exploratory data analysis on a given dataset.

For this analysis, we used the Census dataset. It is more popularly known as the *Census-Income (KDD) Dataset* on UCI[29]. Census dataset contains $\sim 300K$ records. It is extracted from 1994 and 1995 population surveys conducted by the U.S. Census Bureau. It

is usually used as a classification dataset. The classification attribute of *income class* was derived from the attribute of *total person income* in the original survey. The income property is an important demographic attribute. We wanted to model this dataset to represent a typical census microdata of individual entities of a population. Therefore, for our analysis, we treated the prediction class attribute as one of the analysis attributes. We combined the train and test datasets to form our dataset of $\sim 300K$ instances. The original dataset contains 42 attributes which are of mixed type - numeric and categorical. We excluded the attributes that do not give information about the instance such as year (the year of census survey) and instance weight (attribute related to stratified sampling used for creating this dataset). For our training, we ignored the missing values in the dataset.

5.6.2 Evaluation

For presentation purposes, we generated layers of size that would be legible for the report. Remember that, the size of the layers can be controlled using the parameters - τ_1 and τ_2. The generated GHSOM structure contained two levels. First level contained one SOM layer obviously. Three neurons from the first level spawned off into three SOMs at the second level. The values of τ_1 and τ_2 were set to 0.7 and 0.6 respectively. The first level map was a small map representing the data at a very coarser level of granularity. In this discussion, we will explain and analyze the first level map and one of the three SOMs that are expanded into the second level.

Map at Level 1

The first level map is a small map of 3×2 neurons. It represents the dataset at a very coarse level. The U-Matrix and relevant component planes for this map are shown in Figure 5.11. Since this is a fairly small map for a large dataset, most categorical attributes had the same values throughout this map. However, the following observations can still be made from this map:

- The U-Matrix (Figure 5.11g) shows two major groups in the dataset. These groups are based on the financial characteristics of the census population as seen from the component planes.
- We can see that the cluster in the lower part of the map (last row of the U-Matrix) corresponds to the population of individuals with low capital gains, low capital losses, low wage per hour and who work lesser number of weeks in an year.

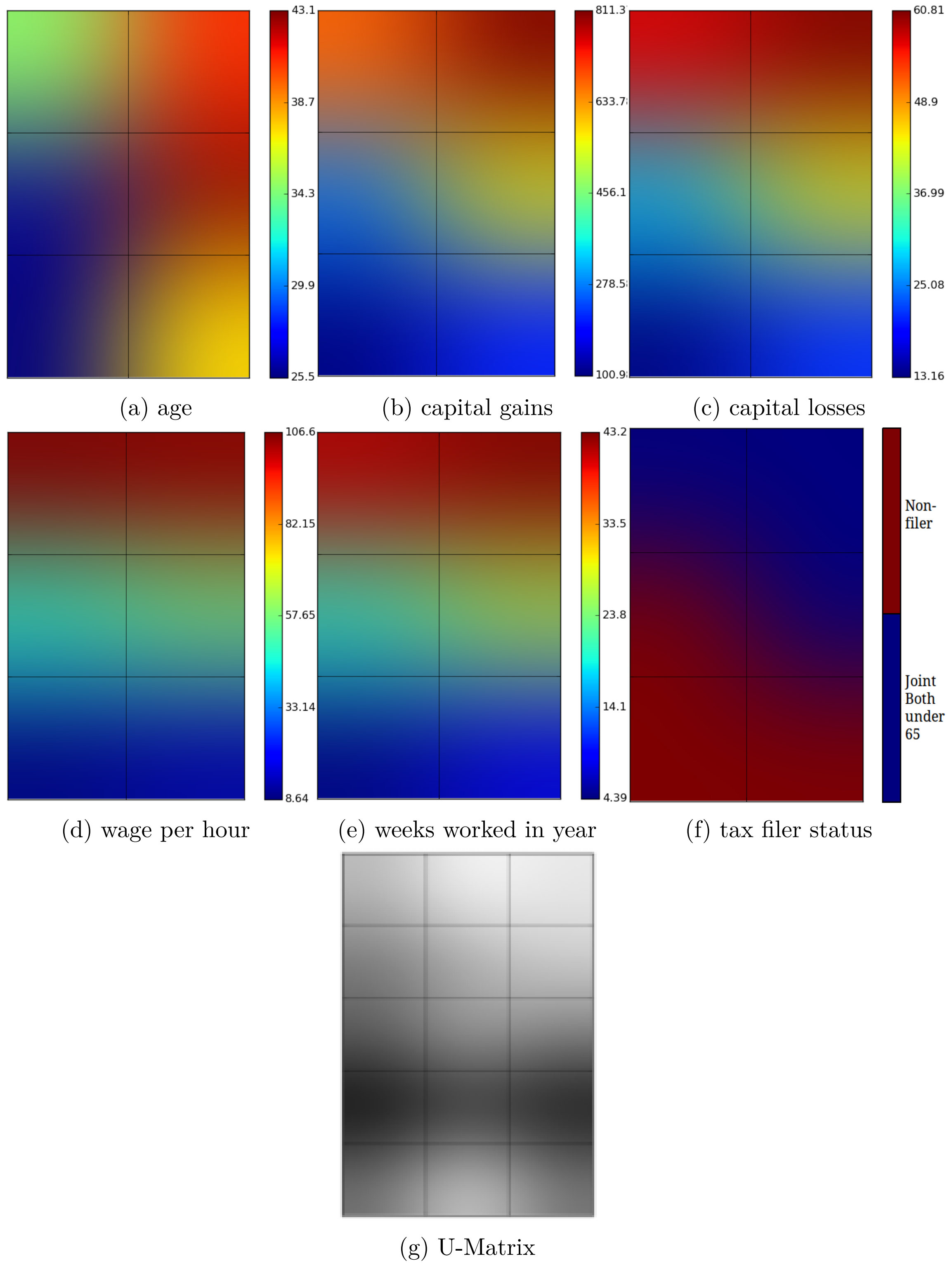


Figure 5.11: U-Matrix and Component Planes for Level 1 SOM of the Census Dataset

- We can see from the component planes of *capital gains* and *capital losses* attributes (Figures 5.11b and 5.11c) that these attributes are highly correlated. Higher capital gains imply higher capital losses and vice versa.
- The attributes of *wage per hour* and *weeks worked in year* (Figure 5.11d and 5.11e) have almost identical distributions. Moreover, the distribution of these attributes is related to capital gains and capital losses attributes, as well.
- The *age* attribute (Figure 5.11a) shows some interesting relation with other finance-related attributes. People in the mid-age of around 40 (orange shade in Figure 5.11a) have higher capital gains and more wage per hour (red shade in Figure 5.11b and 5.11d). In relation with the *tax filer status* attribute (Figure 5.11f), it shows that the population with a lower age are the non-filers of tax.

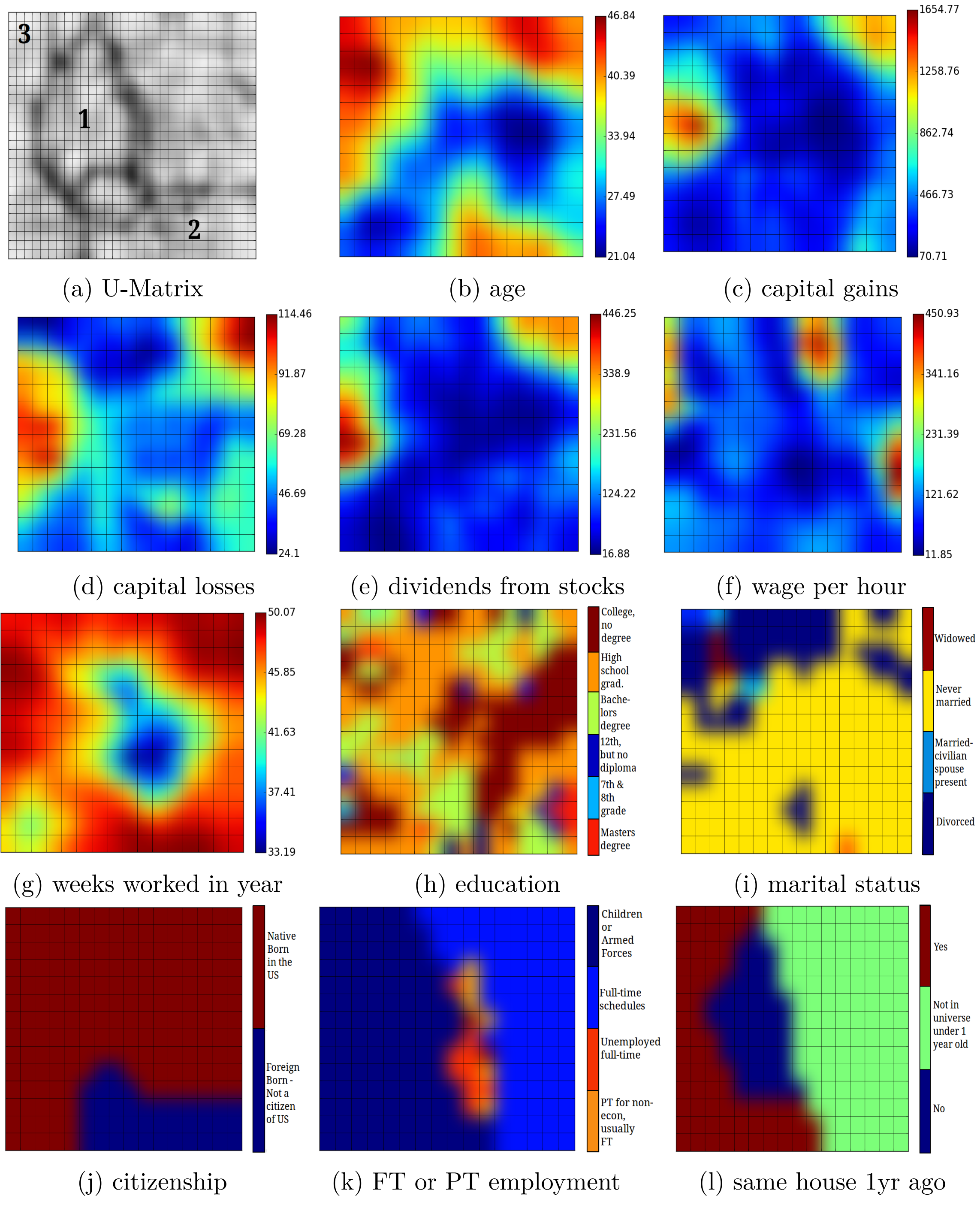
This map provides a very high-level summary of the entire dataset representing data at a very coarser level. It does not provide any detailed information regarding the clusters and attribute correlation for most attributes. Remember that, every neuron in a SOM layer can be considered as a prototype for a cluster of similar instances with similar neurons lying next to each other and forming larger clusters. Accordingly, three of such neuron clusters expanded into new SOM layers at the lower level to represent the data at a finer granularity, and thus depicting the hierarchical relations in the data. We will discuss one of these expanded layers - the one spawned off from cell $[0, 0]$.

Map at Level 2

From the neuron $[0, 0]$ in the level 1 map, a new SOM layer is created. The neuron $[0, 0]$ represented approximately $51K$ instances. The expanded SOM layer was trained on these instances and grew to a size of 14×16 neurons. We have shown the U-Matrix and the component planes for this layer in Figure 5.12. This map shows the data represented by the neuron $[0, 0]$ at a finer level of granularity, thus revealing more clusters within these $51K$ instances. Each cell in this layer represented on average 230 instances. The parent neuron summarized the instances as the ones with fairly higher finances in terms of capital gains and wages, and an average age of 35. However, this expanded map shows finer distribution of these attributes along with some other attributes.

We can deduce the following analysis from this map:

- If we observe the U-Matrix (Figure 5.12a), we can find several small clusters (marked by gray boundaries). These clusters indicate that the data represented by the parent



(a) U-Matrix (b) age (c) capital gains

(d) capital losses (e) dividends from stocks (f) wage per hour

(g) weeks worked in year (h) education (i) marital status

(j) citizenship (k) FT or PT employment (l) same house 1yr ago

Figure 5.12: U-Matrix and Component Planes for Level 2 SOM of the Census Dataset

neuron $[0, 0]$ was indeed quite diversified and needed to be refined further. Thus, it was appropriate for this layer to be spawned off from the parent neuron. We can see some major clusters (comprising of the several minor ones) marked by fairly darker boundaries. We have labelled three of them with labels 1, 2 and 3 and will focus on them.

- The cluster labelled 1 represents the population who haven't stayed in their current house for more than an year (Figure 5.12l). Moreover, this cluster contains the age group of 25 to 35 on average. Also, this population has an average wage per hour in the range of 110 to 150 (Figure 5.12f).

- The cluster labelled 2 is a cluster denoting immigrants (those not born in USA) as asserted by the component plane of the attribute *citizenship* (blue region). We observe that the majority of this population is in the age range of 30-40 (light green, yellow and light orange region near the bottom edge of Figure 5.12b). Also, the majority of this population is unmarried indicated by the yellow colour in Figure 5.12i. This population works for almost 50 hours in a year (red color in Figure 5.12g).

- The cluster labelled 3 has a distinctive attribute of marital status with the value of divorced or widowed (Figure 5.12i) and the age value of more than 45 years (Figure 5.12b). Moreover, this segment of population has stayed in their current house for more than an year (red shade in Figure 5.12l).

- The average age depicted by the parent neuron $[0, 0]$ was 35. However, we can see that there are finer relations between the age range and the other attributes. We can see that the population with a higher age range of 40 and up, work for almost 50 weeks in a year.

- From the component planes of *capital gains* (Figure 5.12c), *capital losses* (Figure 5.12d) and *dividends from stocks* (Figure 5.12e), we see that the distributions of these attributes are fairly correlated. The same was indicated by the map at level 1.

- If we add the *wage per hour* (Figure 5.12f) and *weeks worked in year* (Figure 5.12g) attributes to the above attributes in our analysis, we can deduce that the top right corner of the SOM represents a peculiar group of the population. This group has a comparatively low wage per hour (blue shade) but works for around 50 weeks a year (red shade). They invest in stocks and the dividends from stocks (Figure 5.12e) form a major contributor to their capital gains (reddish shade). Similar characteristics are shown by the population near the middle region on the left edge of the map. These two clusters are distinct from each other in the attribute of *FT or PT employment*

(Figure 5.12k) which stands for full-time or part-time employment status. The red colour region on the left edge of the *dividend from stocks* or *capital gain* attributes correspond to the population in Armed forces while the red coloured region in the top right corner corresponds to the population working full-time. Further, these groups belong to the age group of 40-45 as depicted by Figure 5.12b.

- From the *education* (Figure 5.12h) attribute, we can see that the majority of the population in this map is fairly educated. Almost entire population in this map has an education level of 12th grade or higher.

5.6.3 Learnings from the evaluation

In this section, we observed that the MR-GHSOM generated a GHSOM structure in which the first layer depicted the data at a very coarse level. The Census dataset is quite diversified and finer details of it were delegated to the lower level layers. The first layer represented the data at a coarser granularity showing two major clusters in the entire dataset. The lower level layer from the cell $[0, 0]$ of the first layer showed several segmentations in the instances represented by the cell $[0, 0]$; thus representing data at a finer granularity and showing more clusters within this data.

Using the example of the Census dataset above, we also showed that we can leverage the U-Matrix and the component plane visualizations to segment the dataset and deduce the characteristics of individual segments. Moreover, the component planes prove extremely useful in understanding the distributions and correlations between different attributes.

5.7 Conclusion

In this chapter, we studied the results generated by the MR-GHSOM on various datasets. We first verified the output of the MR-GHSOM in terms of the data representation by a single SOM layer. We also presented the performance gains of using the *batch* two-dimensional growth approach in the MR-GHSOM. We further analysed the hierarchical results generated by the MR-GHSOM on the Zoo dataset. From these results, we could conclude that the MR-GHSOM generates satisfactory results, both in terms of representing data on individual SOM layers as well as depicting the hierarchical relations in the dataset. The MR-GHSOM is thus capable of generating a tree-like structure of individual SOM layers. The SOM layers in the upper levels show data at a coarser granularity and the layers at the lower levels depict the data represented by the parent at a finer granularity.

We also performed an analysis of a mixed-attribute large dataset - the Census dataset. We discussed the results of this analysis using the U-Matrix and the component plane visualizations.

In the next chapter, we will provide a conclusion of our work. We will also discuss some learnings from our experience with the GHSOM and provide directions for future work.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The GHSOM is a dynamic variant of the Self Organizing Map which adapts its structure to the input dataset. It is capable of growing each SOM layer in two-dimensions as well as in depth to create a multi-level tree-like hierarchical structure of individual SOM layers. In this thesis work, we proposed a Map-Reduce variant of the GHSOM called MR-GHSOM, which scales the GHSOM to large datasets.

The training of the GHSOM requires multiple iterations over the input dataset. Apache Spark is a distributed computing engine supporting the Map-Reduce programming model. It is more suited for algorithms (such as the GHSOM) requiring multiple iterations over the input dataset than the popular Apache Hadoop. Apache Spark, using the notion of RDD and the ability to cache the data in memory on the cluster nodes, enables faster processing for such iterative algorithms. Thus, Apache Spark was selected as the Map-Reduce framework for implementing the MR-GHSOM.

When growing in two-dimensions, each individual SOM starts with a size of 2×2 and can grow to an order of tens or hundreds of neurons. The conventional GHSOM grows individual SOM layers, one row or column at a time. Each growth iteration is followed by a training iteration of the SOM layer, which is the most computationally expensive part of the GHSOM. Thus, the number of growth iterations can have a substantial effect on the overall training performance of the GHSOM. To avert this, we recommended a faster two-dimensional growth technique called the *batch growth*, which reaches the target size of an individual SOM in lesser number of growth iterations. We used this technique in our MR-GHSOM with a favourable outcome.

Further, we also introduced an approach that extends the GHSOM to handle datasets with mixed attributes - numeric and categorical. We adopted the *distance hierarchy* method for this. We extended the arithmetic operations in the distance hierarchy technique to support the operations required in the GHSOM algorithm. This enabled the GHSOM to handle high-dimensional mixed attribute datasets. We used the same approach in our MR-GHSOM.

Lastly, we analyzed our MR-GHSOM using some popular datasets from UCI[29]. We presented the results generated by the MR-GHSOM in terms of training an individual SOM, while growing dynamically in 2-dimensions. We also analyzed the hierarchical behaviour of the MR-GHSOM by modelling it on the Zoo dataset from UCI[29]. The results ascertained that the MR-GHSOM produced satisfactory results, both in terms of modelling an individual SOM as well as depicting the hierarchical relations in the dataset. Moreover, the orientation of the layers in the lower levels of the hierarchy was in accordance with the location of the parent neuron in the parent layer. We used the Census dataset from UCI[29] as our case study dataset. We generated a multi-level hierarchy of SOM layers and discussed the results using the U-Matrix and the component plane visualizations.

Overall, we can conclude that the MR-GHSOM is an useful extension of the GHSOM which not only scales the GHSOM to large datasets but also extends it to process high-dimensional mixed attribute datasets.

In terms of the 4Vs that define Big data processing (*Volume, Velocity, Variety and Veracity*), this thesis work concentrates on the *Volume* aspect of the dataset. For the dataset that is already present in the data center, we can create and model the GHSOM structure using the MR-GHSOM and *label* the identified clusters on the basis of the cluster properties. The new data flowing into the system can then be presented to the existing GHSOM model and classified using the existing labels. Periodically, the GHSOM model can be updated to accomodate the new set of instances. In this way, we can handle the *Velocity* aspect of the new in-flowing data. With respect to the *Variety* aspect of data from different sources or new data having a different schema, the MR-GHSOM expects that the input data is preprocessed and has a defined schema. Lastly, the *Veracity* aspect is again delegated to the data collection or the data preprocessing step and not accounted for, in the MR-GHSOM analysis.

6.2 Directions for Future Work

Before concluding this report, We would like to enlist some possible directions for extending the work in future:

- *Skewed growth of SOM layer:*
  From our experience of using the GHSOM, we found that sometimes an individual SOM during the two-dimensional growth, grows in one direction only. For example, a SOM layer may grow by only adding rows during the entire growth process. So, the final dimensions of the SOM turns out to be $n \times 2$, where n is the number of rows and n is fairly large in comparison to 2. This results in a very narrow SOM. It makes visualization, and hence the analysis of the SOM really difficult. We observed that changing the number of training epochs could avoid this skewed growth in some cases. This could be an interesting area to look at in future.

- *Initialization of the first SOM layer:*
  Over multiple executions of the GHSOM over a dataset with the same number of epochs, a non-skewed growth of the SOM layer was observed during some executions. The only variable factor over these multiple executions was the initial values assigned to the neuron weight vectors of the first layer. Thus, the random initialization of the first layer could also be a factor causing the problem of skewed growth. Some research on this initialization would also prove to be useful.

- *Attribute selection:*
  The training of an individual SOM is the most computationally expensive part of the GHSOM. The number of attributes in the dataset has a considerable impact on this, both in terms of time and space. Also, the clustering results could be improved by selecting only the distinguishing attributes. A research in this direction would be interesting as well.

- *Testing on massive datasets:*
  Last but not the least, this thesis work proposes a prototype model of a Map-Reduce algorithm capable of running in a distributed computing environment. Due to the constraints on the available infrastructure and the cluster, we could test our MR-GHSOM on a dataset of approximately $300K$ records. This dataset was large enough for the available cluster of two nodes with one core each. However, similar to the analysis carried out in [39], it would be interesting to test this algorithm on a massive dataset containing millions or billions of records. This analysis would require a more powerful cluster with more number of nodes for parallelization.

References

[1] Damminda Alahakoon, Saman K Halgamuge, and Bala Srinivasan. Dynamic self-organizing maps with controlled growth for knowledge discovery. *Neural Networks, IEEE Transactions on*, 11(3):601–614, 2000.

[2] Vicente Arnau, Sergio Mars, and Ignacio Marín. Iterative cluster analysis of protein interaction data. *Bioinformatics*, 21(3):364–378, 2005.

[3] Suchendra M Bhandarkar, Jean Koh, and Minsoo Suk. Multiscale image segmentation using a hierarchical self-organizing map. *Neurocomputing*, 14(3):241–272, 1997.

[4] Justine Blackmore and Risto Miikkulainen. Incremental grid growing: encoding high-dimensional structure into a two-dimensional feature map. In *Neural Networks, 1993., IEEE International Conference on*, pages 450–455. IEEE, 1993.

[5] Alvin Chan and Elias Pampalk. Growing hierarchical self organising map (ghsom) toolbox: visualisations and enhancements. In *Neural Information Processing, 2002. ICONIP'02. Proceedings of the 9th International Conference on*, volume 5, pages 2537–2541. IEEE, 2002.

[6] Dar-Ren Chen, Ruey-Feng Chang, and Yu-Len Huang. Breast cancer diagnosis using self-organizing map for sonography. *Ultrasound in medicine & biology*, 26(3):405–411, 2000.

[7] Ning Chen and Nuno C Marques. An extension of self-organizing maps to categorical data. In *Progress in Artificial Intelligence*, pages 304–313. Springer, 2005.

[8] Keh-Shih Chuang, Hong-Long Tzeng, Sharon Chen, Jay Wu, and Tzong-Jer Chen. Fuzzy c-means clustering with spatial information for image segmentation. *computerized medical imaging and graphics*, 30(1):9–15, 2006.

[9] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] Michael Dittenbach, Dieter Merkl, and Andreas Rauber. The growing hierarchical self-organizing map. In *ijcnn*, page 6015. IEEE, 2000.

[11] Michael Dittenbach, Andreas Rauber, and Dieter Merkl. Recent advances with the growing hierarchical self-organizing map. In *Advances in Self-Organising Maps*, pages 140–145. Springer, 2001.

[12] Michael Dittenbach, Andreas Rauber, and Dieter Merkl. Uncovering hierarchical structure in data using the growing hierarchical self-organizing map. *Neurocomputing*, 48(1):199–216, 2002.

[13] Bernd Fritzke. Growing grid - a self-organizing network with constant neighborhood range and adaptation strength. *Neural Processing Letters*, 2(5):9–13, 1995.

[14] Bernd Fritzke et al. A growing neural gas network learns topologies. *Advances in neural information processing systems*, 7:625–632, 1995.

[15] Tim Gordon. Is the standard deviation tied to the mean? *Teaching Statistics*, 8(2):40–42, 1986.

[16] Apache Hadoop. Apache Hadoop, 2012. [Online; accessed 1-September-2015].

[17] Jiawei Han, Micheline Kamber, and Jian Pei. *Data mining: concepts and techniques: concepts and techniques*. Elsevier, 2011.

[18] Timo Honkela, Samuel Kaski, Krista Lagus, and Teuvo Kohonen. WEBSOM - self-organizing maps of document collections. In *Proceedings of WSOM*, volume 97, pages 4–6, 1997.

[19] Chung-Chian Hsu. Generalizing self-organizing map for categorical data. *Neural Networks, IEEE Transactions on*, 17(2):294–304, 2006.

[20] Zhexue Huang. Clustering large data sets with mixed numeric and categorical values. In *Proceedings of the 1st Pacific-Asia Conference on Knowledge Discovery and Data Mining,(PAKDD)*, pages 21–34. Singapore, 1997.

[21] Zhexue Huang. Extensions to the k-means algorithm for clustering large data sets with categorical values. *Data mining and knowledge discovery*, 2(3):283–304, 1998.

[22] Chihli Hung and Chih-Fong Tsai. Market segmentation based on hierarchical self-organizing map for markets of multimedia on demand. *Expert systems with applications*, 34(1):780–787, 2008.

[23] Gary D Kader and Mike Perry. Variability for categorical variables. *Journal of Statistics Education*, 15(2):1–17, 2007.

[24] Samuel Kaski, Jari Kangas, and Teuvo Kohonen. Bibliography of self-organizing map (SOM) papers: 1981–1997. *Neural computing surveys*, 1(3&4):1–176, 1998.

[25] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[26] Teuvo Kohonen. The self-organizing map. *Neurocomputing*, 21(1):1–6, 1998.

[27] Teuvo Kohonen, Samuel Kaski, Krista Lagus, Jarkko Salojärvi, Jukka Honkela, Vesa Paatero, and Antti Saarela. Self organization of a massive document collection. *Neural Networks, IEEE Transactions on*, 11(3):574–585, 2000.

[28] Richard D. Lawrence, George S. Almasi, and Holly E. Rushmeier. A scalable parallel algorithm for self-organizing maps with applications to sparse data mining problems. *Data Mining and Knowledge Discovery*, 3(2):171–195, 1999.

[29] M. Lichman. UCI machine learning repository, 2013.

[30] M. Lichman. UCI machine learning repository, 2013. [Online; accessed 1-September-2015].

[31] Apache Mahout. Scalable machine learning and data mining, 2012.

[32] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[33] Thomas M Martinetz, Stanislav G Berkovich, and Klaus J Schulten. Neural-gas' network for vector quantization and its application to time-series prediction. *Neural Networks, IEEE Transactions on*, 4(4):558–569, 1993.

[34] Risto Miikkulainen. *Script recognition with hierarchical feature maps*. Springer, 1992.

[35] Merja Oja, Samuel Kaski, and Teuvo Kohonen. Bibliography of self-organizing map (SOM) papers: 1998-2001 addendum. *Neural computing surveys*, 3(1):1–156, 2003.

[36] Thrasyvoulos N Pappas. An adaptive clustering algorithm for image segmentation. *Signal Processing, IEEE Transactions on*, 40(4):901–914, 1992.

[37] Girish Punj and David W Stewart. Cluster analysis in marketing research: review and suggestions for application. *Journal of marketing research*, pages 134–148, 1983.

[38] Andreas Rauber, Dieter Merkl, and Michael Dittenbach. The growing hierarchical self-organizing map: exploratory analysis of high-dimensional data. *Neural Networks, IEEE Transactions on*, 13(6):1331–1341, 2002.

[39] Tugdual Sarazin, Hanane Azzag, and Mustapha Lebbah. SOM Clustering using Spark-MapReduce. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1727–1734. IEEE, 2014.

[40] Apache Spark. Apache Spark. [Online; accessed 1-September-2015].

[41] Apache Spark. Spark MLlib. [Online; accessed 1-September-2015].

[42] Apache Spark. Spark Resilient Distributed Datasets. [Online; accessed 1-September-2015].

[43] Apache Spark. SparkContext. [Online; accessed 1-September-2015].

[44] Seung-Jin Sul and Andrey 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*, pages 481–489. IEEE, 2011.

[45] Pablo Tamayo, Donna Slonim, Jill Mesirov, Qing Zhu, Sutisak Kitareewan, Ethan Dmitrovsky, Eric S Lander, and Todd R Golub. Interpreting patterns of gene expression with self-organizing maps: methods and application to hematopoietic differentiation. *Proceedings of the National Academy of Sciences*, 96(6):2907–2912, 1999.

[46] Alfred Ultsch. *Self-organizing neural networks for visualisation and classification.* Springer, 1993.

[47] Juha Vesanto. SOM-based data visualization methods. *Intelligent data analysis*, 3(2):111–126, 1999.

[48] Juha Vesanto and Esa Alhoniemi. Clustering of the self-organizing map. *Neural Networks, IEEE Transactions on*, 11(3):586–600, 2000.

[49] Christian Weichel. Adapting Self-Organizing Maps to the MapReduce Programming Paradigm. In *STeP*, pages 119–131, 2010.

[50] Peter Willett. Recent trends in hierarchic document clustering: a critical review. *Information Processing & Management*, 24(5):577–597, 1988.

[51] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[52] Xuegong Zhang and Yanda Li. Self-organizing map as a new method for clustering and data analysis. In *Neural Networks, 1993. IJCNN'93-Nagoya. Proceedings of 1993 International Joint Conference on*, volume 3, pages 2448–2451. IEEE, 1993.

[53] Weizhong Zhao, Huifang Ma, and Qing He. Parallel k-means clustering based on mapreduce. In *Cloud Computing*, pages 674–679. Springer, 2009.