# Incremental Learning for Large-Scale Data Stream Analytics in a Complex Environment

Choiru Za'in

A thesis submitted in total fulfilment of
the requirements for the degree of
Doctor of Philosophy

School of Engineering and Mathematical Sciences

La Trobe University
Melbourne, Australia

August 2020

## Dedication

To my parents, my family, and my children

## Declaration

Except where reference is made in the text of the thesis, this thesis contains no material published elsewhere or extracted in whole or in part from a thesis accepted for the award of any other degree or diploma. No other person's work has been used without due acknowledgment in the main text of the thesis. This thesis has not been submitted for the award of any degree or diploma in any other tertiary institution.

*Choiru Za'in*

*08 August 2020*

# Acknowledgements

The completion of this PhD would not have been possible without the blessing of Allah SWT and the continuous support and encouragement of my supervisors, family, and colleagues.

First and foremost, I would like to acknowledge Allah, the Almighty, the Greatest of all. His continuous grace and mercy was with me throughout my life and evermore during my PhD journey.

I would like to thank my supervisors, Dr. Mahardhika Pratama, Dr. Eric Pardede, and Dr. Zhen He, for their continuous support, dedication, and valuable guidance throughout my candidature.

For my family, I would like to express my sincere thank you to my lovely wife, Dwi Anggraini Puspita Rahayu, who relentlessly support my decision to take the PhD degree. Thank you for your understanding and sacrifices during my candidature, especially during my study visit in Singapore. I would also express my love to my children, Fathan Adhyaksa Zain and Ghayda Atiqah Zain who give me spirit to keep moving. To my mother, Asliyah and Yulianti who support me with their prayer day and night, visiting us, and accompanying our family, during my stay in Singapore. My lovely sister, Nurul Fitriani, who also many times visited us to help us. Also all my brothers and sisters in Bandung, Malang, and Tangerang, Indonesia who also support me. My prayer also to my passed away fathers: Imam Rifa'i and Didik Murhadi (may Allah have mercy on them). They inspired me to work hard, never give up, and reach the line till the end.

I sincerely acknowledge the efforts of all those who directly or indirectly helped me in completing my thesis. I acknowledge all my colleagues both at La Trobe University and NTU Singapore, especially the one from SCSE Lab NTU Singapore, with whom I spent most of my time, to finalize my thesis chapters.

This research was supported by:

- La Trobe University Full Fee Research Scholarship (LTUFFRS) and La Trobe University Postgraduate Research Scholarship (LTUPRS).
- NTU Start-up grant
- Nectar Research Cloud, a collaborative Australian research platform supported by the National Collaborative Research Infrastructure Strategy (NCRIS).
- Pawsey Supercomputing Centre with funding from the Australian Government and the Government of Western Australia.

# Publications

Some of the publications listed in the following journal/conferences have become the content of this thesis.

**Published Journals/Conferences**

- **C. Za'in**, M. Pratama, E. Lughofer, and S. G. Anavatti, "Evolving type-2 web news mining," Applied Soft Computing, vol. 54, pp. 200–220, 2017. [The contents of this paper form the motivation of the thesis].

- M. Pratama, **C. Za'in**, A. Ashfahani, Y. S. Ong, and W. Ding, "Automatic construction of multi-layer perceptron network from streaming examples," in Proceedings of the 28th ACM International CIKM, 2019. [Some parts of this contents share the idea of structural evolution of evolving systems in reaction to drift, chapter 5].

- **C. Za'in**, M. Pratama, A. Ashfahani, E. Pardede, and H. Sheng, "Big data analytic based on scalable panfis for rfid localization," in 2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp. 1687–1692, IEEE, 2018. [The contents of this paper form part of chapter 5].

- **C. Za'in**, M. Pratama, M. Prasad, D. Puthal, C. P. Lim, and M. Seera, "Motor fault detection and diagnosis based on a meta-cognitive random vector functional link network," in Fault Diagnosis of Hybrid Dynamic and Complex Systems, pp. 15–44, Springer, 2018. [The contents of this paper form the motivation of the thesis]

- **C. Za'in**, M. Pratama, E. Lughofer, M. Ferdaus, Q. Cai, and M. Prasad, "Big data analytics based on panfis mapreduce," Procedia Computer Science, vol. 144, pp. 140–152, 2018. [The contents of this paper form part of chapter 5].

- **C. Za'in**, M. Pratama, and E. Pardede, "Evolving large-scale data stream analytics based on scalable panfis," Knowledge-Based Systems, vol. 166, pp. 186–197, 2019. [The contents of this paper form the main work of chapter 5].

- **C. Za'in**, A. Ashfahani, M. Pratama, E. Lughofer, and E. Pardede, "Scalable teacher forcing networks under spark environments for large-scale streaming problems," in 2020 IEEE Conference on Evolving and Adaptive Intelligent Systems (EAIS), pp. 1–8, 2020. [The contents of this paper form the main work of chapter 6]

- A. M. Shiddiqi, W. Wibisono, and **C. Za'in**, "An enhanced sensor placement strategy to quantify small leaks in water distribution networks using weighted lean graphs," in 2019 International Conference on Computer, Control, Informatics and its Applications (IC3INA), pp. 134–139, IEEE, 2019. [The work was done in collaboration with other researchers].

## Abstract

Data streams are not only characterized by continuous and non-stationary characteristics, but they also arrive at a rapid rate from multiple sources, generating high volumes of data which might be beyond the capacity of single processing node machine learning. While most evolving algorithms are adaptive, have an open structure and operate in a single-pass learning mode, they are not designed to run in a distributed processing environment. A novel distributed evolving system, a large-scale data stream analytics framework based on a parsimonious network of fuzzy inference system (PANFIS), termed Scalable PANFIS, is proposed. Scalable PANFIS utilizes the PANFIS evolving algorithm distributed across Spark computing nodes to train large-scale data streams. Scalable PANFIS can deal with large-scale examples, generating a fast-evolving distributed model using an expertly designed merging model mechanism.

Furthermore, the active learning (AL) method is designed to run with PANFIS to accelerate the PANFIS learning mechanism further. We design four-structure Scalable PANFIS algorithms using a combination of PANFIS with/without AL and the merging/majority voting method. The results show that all these combinations produce comparable accuracy across all datasets. Of these four structures, the combination of merging models with AL trains large-scale datasets with a significantly faster running time. In comparison to the other algorithms in the Spark library, Scalable PANFIS methods yield higher accuracy, despite a slightly slower training time for some cases.

Another large-scale data stream analytics framework, namely scalable teacher-forcing networks under the Spark environment for large-scale streaming problems (ScatterNet), is also proposed. While Scalable PANFIS generates an evolving distributed model via the merging procedure in one task, ScatterNet evaluates data streams in sequential tasks under the prequential test- then-train scheme, a standard evaluation for data stream learning. Furthermore, Scalable PANFIS does not adopt an elastic structure evolution in learning from data streams which is vital to handle concept drift.

The ensemble structure of ScatterNet evolves in reaction to concept drift on both local and global scales. Concept drift on a local scale is carried out by the teacher forcing network, considering the temporal characteristic of data streams without using a time-intensive iterative gradient calculation. On the global scale, the drift

# Contents

# List of Figures

# List of Tables

# 1 Introduction

## 1.1 Background

Nowadays, in the era of the Internet of Things (IoT), data are generated from devices/sensors in the form of text, images, videos, etc. Data from sensors arrive continuously from multiple sources, different environments as data streams [1]. As a result, vast volumes of data can be generated in the cloud over time. Furthermore, data streams are also characterized by non-stationary environments that come from real-world applications [2].

Due to the high business demands, these enormous volumes of data streams need to be learned immediately as they arrive for decision-making purposes [3]. While large-scale data streams have a high potential to improve effective decision making, learning from this data is challenging. Based on [4], a general problem large-scale/big data of big data includes 5V (Variety, Velocity, Volume, Value, and Veracity) characteristics. In machine learning literature, of these five characteristics, there are at least two main issues in learning from data streams which trigger further investigation: huge volumes and velocity. Another issue of data streams is the non-stationary characteristics of data streams [5]. The vast volumes and velocity characteristics causes the generation of big data due to the high speed of arrival data streams. In contrast, the non-stationary characteristic is related to the changing of data distribution over time. These challenges provide excellent opportunities for many research directions.

In the realm of learning from data streams, **online algorithms** and **distributed algorithms** have become essential research topics over the last two decades in dealing with data stream characteristics [6, 7, 8]. The online algorithms are capable of continuously updating their parameters and modifying their structures to adapt to the available new samples of data streams. Furthermore, their single-pass learning mechanisms enable them to learn the infinite tasks without a retraining phase, using a limited memory capacity which makes them different from traditional offline learning algorithms [9]. On the other hand, due to the growth of distributed processing technology, from the perspective of offline learning, distributed algorithms have become the alternative solution to handle the vast volumes of data [10].

From an online learning perspective, the research in this area has grown significantly, which is marked by a significant amount of work that has been conducted in recent years [11, 12, 13, 14, 15]. Of these online algorithms, work on **evolving fuzzy systems** [9, 16] and ensemble-based **incremental learning** [17] is the most frequently published in the literature. These algorithms have been developed to successfully handle web-scale datasets such as web news mining [18] and a real-world problem demonstrated in [19].

The similarity between these two algorithms lies in their evolving/adaptive model, either single or multiple models, and their single-pass learning mechanisms. In particular, the ensemble-based incremental learning algorithms provide a mechanism to relinquish a previously valid concept while acquiring new knowledge, the property of which the incremental learning algorithms require.

Despite their online learning properties, they may suffer from problem of huge volumes of data which need to be processed immediately due to the rapid arrival of data streams. The volume of data being processed is beyond the capacity of single processing node machine learning [10, 20]. The problem of data streams has scaled up into large-scale data streams which online algorithms alone cannot handle.

From a distributed processing perspective, MapReduce [21] was recognized as the first distributed learning framework, which was initially run on the Hadoop distributed processing

platform [22]. Following this, many distributed processing platforms appeared. Apache Spark (Spark) [23] is one of the latest distributed processing platform and provides a built-in machine learning framework called MLlib [24]. Spark features in-memory computation where the computation process is undertaken using random access memory instead of disk drives. As a result, Spark can significantly reduce the processing time compared to other distributed platforms. Despite the apparent advantages of the Spark platform, most of the distributed algorithms are still built based on offline algorithms, which are computationally ineffective because they need a retraining procedure to acquire the whole historical model. Therefore, they are not feasible for processing ever-growing large-scale data streams.

While online learning and distributed processing have become emerging research topics for large-scale data streams over the last decade, to the best of our knowledge, the synergy between them is rarely reported. The vast majority of online algorithms in the literature have been built under a single processing node environment. This property hinders them from processing large-scale data streams. While a lot of studies have been devoted to coping with data stream characteristics, learning from data streams remains an open issue. Ideally, the desirable properties of the algorithms should be capable of learning from new examples (adapting to the recent concept drift), retaining the previously valid knowledge, and processing samples in a distributed manner using the Spark platform.

The remainder of this chapter is organized as follows: section 1.2 discusses research motivations. Section 1.3 discusses the objectives of the thesis, whereas section 1.4 details of the contributions of the thesis. Finally, the structure of the thesis is summarized in section 1.5.

## 1.2 Research Motivations

This section presents the research motivations which briefly highlight the current state-of-the-art of online algorithms, and their further development (e.g. incorporating distributed features using Spark into the existing frameworks) based on the current challenges mentioned previously

in section 1.1. To cope with these challenges, our motivations are driven by the development of frameworks which are capable of learning from new samples (either in the form of batch or instance data), responding quickly to a change in the data distribution, keeping the useful existing knowledge, and performing distributed learning.

### 1.2.1 The Taxonomy of Online Algorithms and Further Development Work

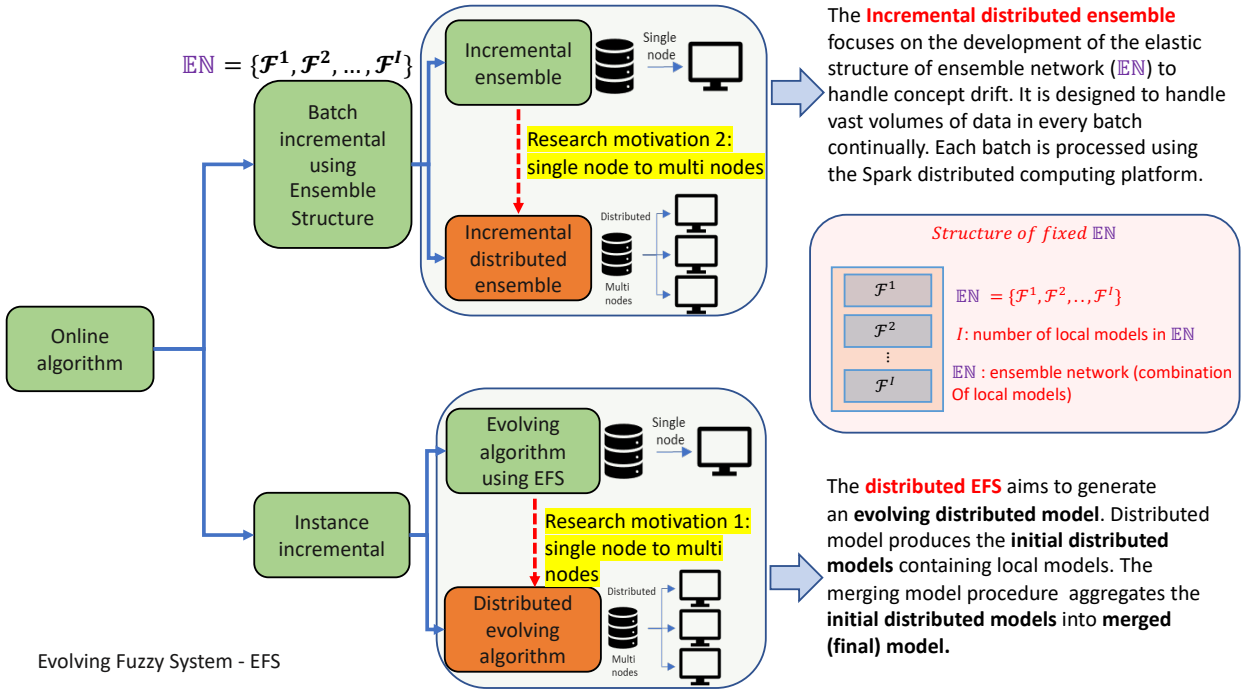


Figure 1.1: The taxonomy of online algorithms and their structure models (single/multiple models) along with the map of our research motivations

Due to the broad range of types of online algorithms, we illustrate the taxonomy of online algorithms, as depicted in Fig. 1.1 to give an insight regarding the possible area in existing online algorithms that can be scaled in a distributed learning framework. It can be seen that online algorithms are the only possible solution to coping with the data stream problems. To date, abundant seminal works have been carried out in this area as in [11, 12, 25, 13, 26], all of which have the incremental feature, allowing them to evolve their models (structure and parameter) to learn ever-growing data streams. Their structural evolution is an essential

procedure to adjust the model to the new instance(s) patterns [16]. Despite their incremental feature, none of these algorithms are designed to be operated in the distributed processing environment.

Our thesis motivations are to build large-scale data stream frameworks. The learning mechanism of the vast majority of online algorithms can only run in a single processing node environment. As a result, they may have difficulty processing the enormous volumes of data due to their inability to process data streams in parallel mode. In this case, modifying the existing online algorithms so they are able to operate in a distributed processing environment without deteriorating the performance of the algorithms is essential, which entails some challenges.

### 1.2.2 The Types of Online Algorithms and the Reasons for Using these Algorithms for Further Development

The majority of problems in data streams can be handled by using online algorithms. While abundant work has been carried out in this area, there is a need to develop more advanced frameworks (e.g. incorporating a distributed computing platform so that it can deal with vast volumes of data streams). For this reason, it is essential to investigate the trend/current state of some architectures of online algorithms to determine which area needs enhancement.

Firstly, we start our work with the evolving fuzzy systems (EFSs) whose models can be quickly adapted to the system conditions at any state (specifically for every sample) during online real-world processes as in [11]. We are motivated to scale the EFS into a distributed evolving fuzzy system, where the challenge of this work lies in generating an evolving distributed model. This is understandable because every node in the Spark cluster processes different partitions of data streams, generating many local models. In this work, handling different local models efficiently using either model merging or model combination is the main focus.

For the second motivation, we focus on developing an incremental learning approach [16], particularly a distributed incremental ensemble. We are motivated to develop incremental

learning capable of processing large-scale data streams utilizing a Spark distributed processing platform. In practice, the task undertaken in every batch is conducted in a distributed manner due to the large size of the batch data. The model is built on a dynamic ensemble structure, which evolves in each task.

We visualize our motivations in Fig. 1.1. The orange highlighted frameworks are the area we want to develop, whereas the grey highlighted frameworks are the existing developed frameworks. The underlying reasons for choosing these types of algorithms as our research motivations are summarized as follows:

- Our first motivation is to develop **a distributed evolving fuzzy system framework (distributed EFS)**. This framework is designed to scale the computational capability of a single processing node EFS into multiple processing nodes of distributed EFS. This procedure aims to speed up the learning process of EFS, while keeping the same level of accuracy of EFS [3]. EFS is a fuzzy-based evolving model characterized by single-pass learning and adopts the open-structure property. EFSs have become a well-established research area in the data stream community with an extensive array of published work. However, little attention has been paid so far to designing the EFS in a distributed manner using the Spark ecosystem. Thus, it is worth investigation.

- Our second motivation is to develop a **distributed incremental ensemble framework**, which is ensemble-based incremental learning which learns large-scale sequential batches. This procedure is different from the distributed EFS, which is designed to learn large-scale data streams in one task/batch. This framework also demonstrates the evolution of the ensemble structure in every task, a feature that is absent from the first framework, to react to concept drift. The reasons for choosing an ensemble method as the base-learning algorithm for this framework is two-fold. Firstly, the ensemble algorithms tend to cope better with the concept drift problem in data streams as presented in [27, 28, 29]. The second reason is that the incremental learning ability of ensemble methods has been widely used in the literature [30, 31], but there is little evidence found for their distributed

versions. The incremental and distributed properties of this framework give us an answer to the research challenge as to whether our frameworks are capable of handling massive data streams which arrive continuously in large volumes over time [17].

### 1.2.3 The Connection between Frameworks in Our Thesis and the Underlying Reasons for Developing Such Architectures

#### 1.2.3.1 Tackling the Volume and Velocity Characteristics of Data Streams

In [8], it is discussed that evolving algorithms and distributed algorithms are two possible approaches to deal with data streams. These approaches aim to improve the scalability of the algorithms, so they become capable of handling large-scale dataset due to the volume and velocity characteristics of data streams.

In the first approach, it is clear that evolving algorithms are scalable because they adopt the online working principle, optimizing the limited resources (memory and storage) to process the data streams as demonstrated in [11, 12, 32]. In practice, examples are learned instance-by-instance incrementally in a single-pass scenario.

In the second approach, distributed algorithms, scalability is achieved through parallelization as in [21], which aims to accelerate the training process. The main advantage of distributed algorithms lies in their capability to process large-scale datasets at once. Their drawback, however, lies in their offline learning mechanism, which hinders them from learning ever-growing data streams. Learning new data requires a retraining procedure which leads to the problem of catastrophic forgetting [33].

Evolving algorithms are limited by their single processing node capacity, which hinders them from processing vast volumes of data. For this reason, it is necessary to endow evolving algorithms with a distributed processing feature. Therefore, a distributed evolving algorithm framework, namely **distributed EFS**, is designed in the Spark platform. In this case, EFS is

used as a base structure to develop distributed EFS.

#### 1.2.3.2 Tackling the Multiple issues of Data Streams

The primary motivation of this subsection is to scale the incremental ensemble algorithms. While the research on incremental ensemble algorithms has recently become state-of-the-art with its remarkable performance [28], the issue of scalability remains a challenge due to the problem of the rapid generation of data streams which is beyond the capacity of a single processing node to process. In the literature, the state-of-the-art on distributed incremental learning algorithms was undertaken in [34]. However, its base-learning algorithm still uses a batch algorithm, which is static. Thus, it is not capable of handling the concept drift issue. To the best of our knowledge, there are not many current works which address this issue. Therefore, a **distributed incremental ensemble** is designed to tackle the multiple issues (non-stationary, volume, and velocity issue characteristic) of data streams.

## 1.3 Objectives

The primary objective of this work is to develop effective algorithms that can deal with the major problems of large-scale data streams: volume, velocity, and non-stationary traits, as described in section 1.1. Our algorithms are mainly proposed to solve the classification task. Based on our research motivations, we aim to develop large-scale data stream analytics frameworks extended from existing state-of-the-art algorithms (EFS and incremental ensemble learning). For each corresponding motivation mentioned in section 1.2, our proposed frameworks should achieve the following objectives:

- To cope with the velocity issue of rapid data streams, a **distributed EFS** framework implemented on the Spark platform is proposed. This framework should be able to speed up the training process without reducing accuracy compared to its base-learning algorithm,

EFS, which only can be deployed in a single processing node machine. This framework should demonstrate the scalability of EFS, which can adapt to the new environment as well as work in a distributed fashion. As a result, this framework can learn large-scale data streams characterized by huge volumes, high speed, and non-stationary traits of data streams.

- A **distributed incremental ensemble** learning framework implemented on the Spark platform is proposed to address the three problems of data streams: volume, speed, and the non-stationary characteristics of data streams. This framework should demonstrate a capability to deal with never-ending large-scale data stream problems. This framework should also achieve comparable accuracy with the single processing node of the evolving algorithm. This framework aims to equip an existing incremental ensemble with a distributed feature using Spark to scale its training capacity while keeping the advantageous progressive learning feature. However, this task is challenging due to the scalability issue, the increase of network capacity as the data batches grow, and it potentially suffers from accuracy degradation when the model merging procedure is not adequately designed.

## 1.4 Contributions

In this thesis, two frameworks are compiled according to the corresponding objectives. The contributions of this thesis can be summarized as follows:

- We propose Scalable PANFIS, a novel distributed EFS framework, which can deal with the large-scale data stream classification problem. This framework is scalable so it can cope with a data stream's changing patterns. Scalable PANFIS introduces **four structure learnings** resulting from two types of model aggregation procedures in distributed training task and two types of testing tasks. A distributed training task is undertaken using the Spark platform where large-scale data streams are divided into several partitions, and from each partition, a local model is generated. Therefore, for a distributed training

task, several local models are generated, and the collection of local models is called the initial distributed models. Each local model is formed as a result of the data partitions' training using PANFIS (with or without **Active Learning (AL)**) as an evolving algorithm. The AL mechanism is embedded in PANFIS to accelerate the learning process at the local level (a partition of training data). The testing task of Scalable PANFIS utilizes two types of model aggregations: **model merging** and **majority voting**. The novel model merging method extracts, selects, and eliminates inconsequential rules across the initial distributed models before the merging process. For majority voting, the initial distributed models are used directly for the prediction task without merging. The final output is determined from the majority class voted by local models. In other words, the final result is a composite output of several local models.

- A novel distributed incremental ensemble, namely ScatterNet, is proposed. The incremental feature of ScatterNet offers a solution of never-ending batches of massive data stream learning problems under the distributed computing platform of Spark. The basic structure of ScatterNet is an ensemble network containing a stack of base models which is updated sequentially for every task/batch. For each batch, the distributed training task is performed at the global level forming initial distributed models, where the initial distributed models contain local models. Each local model is generated from training data partitions in the Spark node using the **teacher-forcing network** method, taking into account the temporal characteristic of data streams without iterative gradient calculation. The merging of the initial distributed models produces a merged model, as a candidate of the base model, which is carried out using **online model selection** and **zero-shot merging approaches** which are capable of keeping structural complexity under control while retaining generalization performance. The dynamic structure of ScatterNet is controlled by a **drift detection method** at the global level (across-the-batch) and advances network significance with the incorporation of the forgetting mechanism based on the bias-variance decomposition method at the local level. That is, if drift is detected, a merged model is stacked in the current ensemble network. Conversely, base models are

removed by using **the model pruning mechanism**, employing a statistical measurement to assess the insignificant models. At the local level, in the case of high bias, hidden nodes grow, whereas in the case of high variance, hidden node pruning takes place.

## 1.5 Structure of the Thesis

In this thesis, the advanced algorithms are compiled to learn large-scale data streams. The structure of the thesis is as follows:

- Chapter 1 presents the concept of learning from data streams and introduces three main problems: velocity, volume, and the non-stationary traits of data streams. This chapter provides the motivation for developing large-scale data stream frameworks which have properties to deal with large-scale data stream problems. The thesis objectives which address the challenges are also discussed, followed by the thesis contributions.

- Chapter 2 introduces the research context, providing a broad overview of machine learning algorithms related to the large-scale data stream frameworks. For each type of algorithm, the learning mechanism and the data flow in both the testing and training task are elaborated.

- Chapter 3 introduces the literature review related to the proposed method, especially the state-of-the-art distributed algorithms to cope with the velocity of data streams. It also reviews some of the seminal incremental learning algorithms which offer solutions to deal with the dynamic changes of data streams.

- Chapter 4 presents our work, namely Scalable PANFIS, a type of distributed EFS algorithm, as our first contribution. This algorithm can speed up the training process using multi-node processing instead of the single-node processing of evolving algorithm, namely PANFIS [11] without suffering a loss of accuracy in comparison to the single processing node of PANFIS.

- Chapter 5 presents one of the works on distributed incremental ensemble algorithms, namely ScatterNet. This chapter proposes the idea of an incremental learning scheme in a distributed fashion. In this work, the structure of the ensemble network dynamically evolves in reaction to the concept drift which exists in data streams.
- Chapter 6 covers the conclusion of the thesis.
- Chapter 7 provides the future direction.

# 2

# Research Context - Architecture and Problem Definition

In this chapter, the background knowledge and methods related to the thesis are discussed. We introduce the general problem, architecture, terminology, notation, learning, and inference mechanisms used in learning from large-scale data streams. We structure this chapter into the following discussions. Firstly, the general large-scale data challenges are discussed in subsection 2.1, which covers studies to cope with large-scale data problems. This part generally covers the introduction of large-scale data analytics. Then, the details of the platform of large-scale data analytics and algorithms attached to it as the learning and inference engine is discussed in subsection 2.2 and 2.3 respectively. As our thesis is focusing on the development of algorithms to handle large-scale data stream challenges, the rest of the chapter is discussing various algorithms and their architecture. The architecture covers batch learning algorithms into evolving algorithms as the learning and inference engine in large-scale data analytics. Note that the algorithms developed in this thesis are supervised learning algorithms striving to solve the classification problem.

## 2.1 Introduction to Large-scale Data Analytics

Large-scale analytics is commonly associated with scalable machine learning using an advanced computational platform. This platform can be operated either by using vertical or horizontal

scaling techniques [35]. Vertical scaling technique refers to the enhancement of single machine computational capabilities (e.g.Graphics Processing Unit (GPU)), whereas the horizontal one refers to the distribution of tasks across multiple nodes. Large-scale data analytics is also commonly known as a process to find hidden insight into the data to help organizations in making business decisions. This process includes a complex process such as data preprocessing and analysis. While standard machine-learning algorithms can only process the simple dataset, the significant challenges imposed by large-scale data according to [35] are described as follows:

- Scalability: Large-scale data is associated with large or complex datasets from various sensors. Thus the algorithm must be scalable to cope with the volume and velocity of data.
- Decentralization: This problem is related to the capability of the algorithm to be distributed to handle multiple sources of data.
- Dynamic datasets: The algorithm should be able to operate on a dynamic dataset from the dynamic sensors. It may also be capable of exploiting a dynamic graph for visualization.
- Nonuniform Dataset: Data may come from different formats such as audio, weblogs, social media. This problem is also a challenge for large-scale data processing.
- Privacy: This challenge is related to the fact that many data is confidential and often raises privacy concerns. While this data is beneficial to be processed, it cannot be handled by traditional methods due to its unavailability to protect the data privacy.

As explained above, large-scale data analytics impose a wide range of problems. In general, the study of large-scale data processing involves two significant steps: 1) The platform; 2) The learning and inference. The first part includes setting the stage for large-scale data processing: big data model selection, storage selection (e.g. Hadoop distributed file systems - HDFS), and computational framework selection. The second part is related to the main algorithm that is attached into the large-scale data platform. Our thesis focuses on the second part, specifically the development of the algorithms which can deal with large-scale data stream problems.

## 2.2 Large-scale Data Analytics Platform

The explosion of data has led to the emergence of scalable data processing. According to [35], big data computational frameworks could be categorized into three models: batch, real-time processing, and hybrid model. These three models of scalable data processing are summarized as follows:

- Batch model refers to the conventional distributed processing whose architecture is depicted in Fig. 2.5, which more focuses on solving the volume characteristics of data. The example for this is MapReduce which has two main function Map and Reduce. Despite its simplicity, MapReduce suffers from communication cost, availability to perform the iteration process, and availability to process data in a real-time manner [36].
- Real-time processing model refers to the adaptability to process the continuous data as stream. This model focuses on the velocity characteristic of the data stream. The example platform that can accommodate this model is Storm. Another advantage of Storm also lies on its latency. Despite its real-time characteristics, it is lack of iteration [35].
- The hybrid model offers the solution as the framework that can handle both volume and velocity characteristics of big data (explosion of data). This model architecture can handle both batch, real-time, and combination of both. The example of this hybrid model is Spark which can operate for both batch and real-time model.

  In this subchapter, the platform of large-scale data analytics using Spark is discussed. In particular, we discuss about external file system such as HDFS and the interoperability between Spark and HDFS.

### 2.2.1 Hadoop

Hadoop is an open-source data management platform which was initially developed by Google to handle web-scale datasets as a result of cloud-based big data generation. Hadoop makes

use of the storage layer called HDFSs to run on commodity hardware. As a result, it can save the operational cost. HDFS has many similarities with other standard distributed file systems. However, a significant improvement from the previous distributed file systems is that Hadoop is highly fault-tolerant.

A cluster of HDFS adopts the master-slaves architecture which consists of a single Namenode as a master server and several Datanodes as the slaves where the actual data are stored. Namenode manages the file system namespace which has the authority to let clients access the files in a cluster. To support this operation, Namenode has a metadata which records all the files stored in the HDFS's cluster, e.g. the location of the blocks stored, the size of the files, permissions, hierarchy, etc. Datanodes are responsible for providing requests from clients for either read and write requests. In the case of the write operation, Datanodes are also responsible for splitting the file into several blocks whose size is defined by the user. Right after the writing operation has been executed, Datanodes create the replica for the blocks that have been made, a standard procedure in HDFS, which aims to maintain the high availability of data. HDFS architecture is depicted in Fig. 2.1.

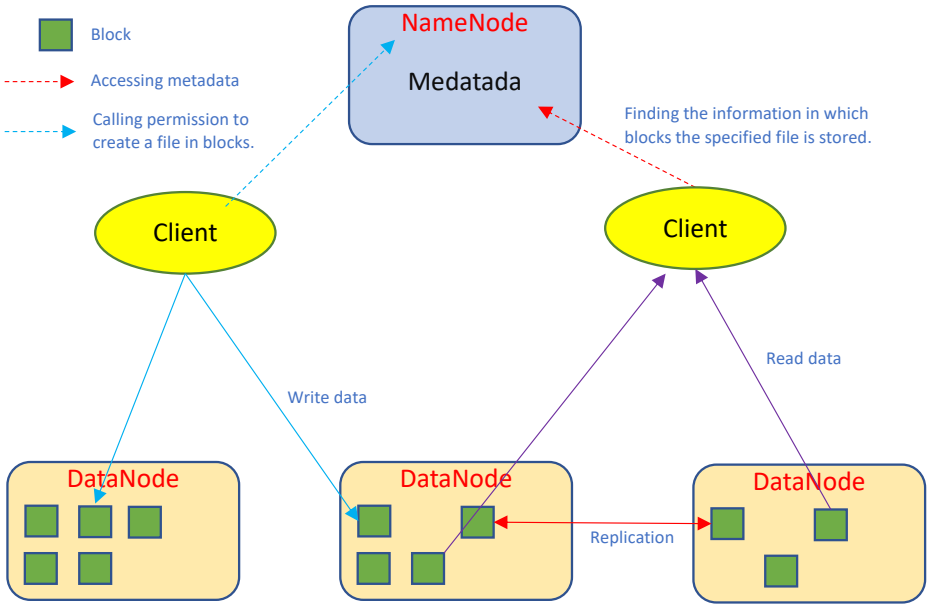


Figure 2.1: HDFS architecture

MapReduce was one of the first programming models implemented in Hadoop to be used for distributed learning operation. From a process point of view, a MapReduce program is split

into two procedures: 1) Map, and 2) Reduce. In the realm of parallel data processing, the Map procedure is undertaken in the cloud, which processes a batch of data to be processed distributively. A Map procedure performs partitioning, filtering, sorting, and training data from the machine learning point of view. The results of the Map operation are stored on the local disc. This mechanism aims to avoid replication in the cluster. A Reduce procedure executes an aggregating operation such as (summation, averaging). The MapReduce sequence is depicted in Fig. 2.2.

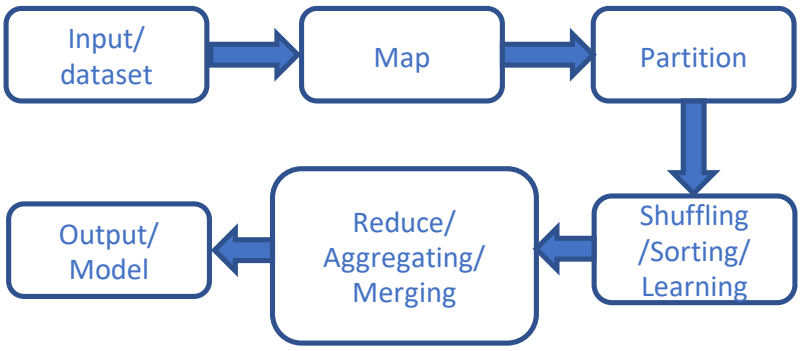


Figure 2.2: MapReduce process sequence

However, there are some drawbacks related to MapReduce programming due to its inability to perform iterative and interactive analytics tasks. This problem is because MapReduce is a file-intensive operation. When the Map and Reduce perform their operations, files are generated into several blocks. This condition is not beneficial for read/write operation. Furthermore, Hadoop read/write operation is carried out to HDFS storage, which is slower than using RDD type storage. As a result, it will cause a significant delay in the overall machine learning tasks.

### 2.2.2 Apache Spark

Apache Spark (Spark) is an open-source data processing platform that allows machine learning (e.g. it can be any application) to learn large-scale dataset in a distributed manner across multiple workers/nodes/computers. Similar to Hadoop, the Spark cluster uses master-slave architecture which comprises three parts: a driver node as a master, a cluster manager to manage jobs (tasks and data flow), and many executors that run across as worker nodes in

the cluster. The interconnection between Spark components is depicted in Fig. 2.3. Unlike Hadoop, which can only be used for batch algorithms, Spark offers real-time processing as one of the additional features compared to Hadoop.

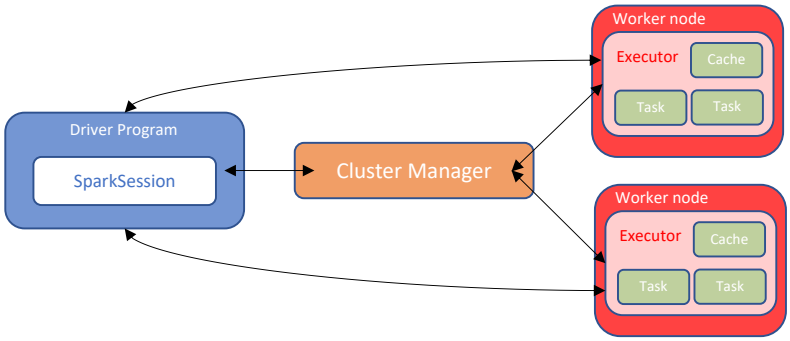


Figure 2.3: Spark cluster architecture

The working principle of Spark is defined as follows. The driver generates SparkSession, the entry point of Spark functionality, which allows the Spark application to access the cluster with the help of a cluster manager, including converting the dataset into the Spark DataFrame as a Spark data abstraction. While Spark is regarded as an advanced data processing platform, it does not have a dedicated storage file system. As a result, Spark interoperate with the other external storages such as HDFS. A Cluster Manager bridges the interoperability between Spark and HDFS. It controls the requests from a driver program into a namenode of HDFS and acquire a data from it. This procedure is similar to a Map function in the Hadoop ecosystem, but in Spark, the data being taken from HDFS is converted first into specific Spark data abstraction. Note that there are several types of Spark data abstractions such as the well-known **resilient distributed dataset**. In this thesis, we use the Spark DataFrame as the Spark data abstraction. After this, the Spark driver translates the user's written code into Spark jobs and passes these jobs to the cluster manager. Then, the cluster manager distributes the smaller jobs and the Spark DataFrame partitions into all the worker nodes. Each worker node may receive more than one partition depending on the settings of parameters. Each worker node processes/learns all the corresponding Spark DataFrame partitions in a serial manner, and sends the results back to the SparkSession.

Spark contains built-in modules encapsulated as a Spark ecosystem to enable the user to design

user-defined commands (e.g. algorithm) to manipulate the Spark DataFrame inside the Spark cluster via the SparkSession. The Spark ecosystem comprises the Spark library and Spark core. Spark core consists of several basic programming languages such as R, Python, Java, and Scala, whereas the Spark library covers several components such as Spark SQL, Spark Streaming, Spark MLib, Spark GraphX, and Spark R, all of which are created to support data manipulation in the cluster. Of these Spark components, MLib is a standard module which consists of several built-in algorithms that are mainly used for a distributed machine learning task.

## 2.3 The Learning and Inference Engine of Large-scale Data Analytics

This subsection aims to define terms used in several types of algorithm to distinguish their architectures, characteristics, and working principles, including the samples used to generate their model. We introduce the term of **base-learning algorithm**. In this thesis, a base-learning algorithm is a machine learning which is used to learn the data and produces a **single model**. Base-learning algorithm can be regarded as the learning and inference engine that learns the data, produces model, and infers the unlabelled data.

Conventional base-learning algorithms cover popular machine learning algorithms including but not limited to: linear discriminant analysis, decision trees, naive Bayes classifiers, k-nearest neighbours, and standard fuzzy inference systems such as an adaptive-network-based fuzzy inference system (ANFIS) [37]. These algorithms are based on the offline working principle, with a single and static model structure, and running on a single processing node as portrayed in Fig. 2.5. When these base-learning algorithms are used in a more complex structure such as ensemble or distributed algorithms, the **multiple models** are generated as depicted in Fig. 2.6 and Fig. 2.7.

An example of ensemble algorithms is presented in subsection 2.4. We introduce the ANFIS

structure in subsection 2.5. Then, in subsection 2.6, we review in detail the learning mechanisms of all the related algorithms which adopt the offline working principle. In subsection 2.7, we discuss the online algorithms which include batch incremental learning and instance incremental learning. Finally, in subsection 2.8, we introduce the learning policy scenario of our proposed algorithms.

As our work deals with the data stream problems, we utilize the evolving algorithms such as a parsimonious network based on a fuzzy inference system (PANFIS) [11] and a parsimonious learning machine (PALM) [32] as the base-learning algorithms. As the evolving algorithms, their components (rules) are evolving following the data stream conditions. The types of evolving algorithms are varied. Although ANFIS adopts an offline working principle, its structure is adopted by many works as a basis on which to build their evolving systems, including PANFIS and PALM. Thus, the ANFIS structure is worth discussing.

Furthermore, it is also essential to discuss the structure of ensemble and distributed algorithms, including incorporating the base-learning algorithms into their learning systems. In general, the structure and learning mechanisms of all the related algorithms are summarized as follows:

- The standard **distributed algorithms** use a base-learning algorithm distributed into several nodes. Suppose that large-scale static batch data is divided and distributed evenly into several nodes. Each node may learn several partitions which come to them. The data partitions are learned locally in their corresponding nodes, such that from each partition a local model is generated. At the end of the distributed training task, several local models are constructed, and we refer to these models as the **initial distributed models**. A **merged model** is obtained by merging the initial distributed models.

- The standard **ensemble algorithms** make use of multiple base-learning algorithms to learn a static dataset using a single processing node. The learning output of an ensemble algorithm is multiple models wher

## 2.4 The Ensemble Algorithms: Examples

The aim of this subsection is to provide some examples of seminal ensemble algorithms which exist in the literature. The ensemble algorithms are built based on the hypothesis that combining several base models for the testing task can often yield a better generalization performance. Note that ensemble algorithms are not originally designed to handle large data. On the other hand, it slows the inference process as inference tasks need to be carried out multiple times. However, the ensemble algorithm can be attached in the large-scale data analytics platform. Thus, this algorithm can yield a higher accuracy in acceptable running time.

The critical success in machine learning tasks (e.g. classification or regression) is determined by selecting a suitable base model for prediction. The implementation of the model selection can be carried out by assigning the weight for each base model in the network. Based on how the base models are updated, generally, the ensemble methods can be categorized in two groups: boosting and bagging. In this subsection, the two ensemble methods are discussed: boosting and bagging.

### 2.4.1 Boosting

The boosting methods aim to improve the performance of the weak learners (base models or base classifiers in a classification problem) by converting them into a strong learner. Suppose that a base model will work on any data distribution in a binary classification task where instances are classified into positive and negative. Suppose that the training instances are classified into positive and negative. Suppose that the training instances are denoted as $\mathbb{D} = [X, Y]$. $X \in \Re^{N \times n}$ represents the input data and the labels are denoted as ($Y \in \Re^{N \times m}$). $Y$ is also called the ground-truth function of the training data, which is denoted as $\mathcal{F}^{ground}$.

After the training, the ensemble hypothesis is formed denoted as $\mathbb{EN} = \{\mathcal{F}1, ..., \mathcal{F}i, ..., \mathcal{F}I\}$. $\mathbb{EN}$ is also called an ensemble network, and $I$ denotes the number of base models in the $\mathbb{EN}$. The

pseudocode of the boosting algorithm is presented in Algorithm 1. The idea of boosting is to learn from the mistakes made by the previous base models to boost the ensemble performance. That is, the next base model focuses more on incorrectly classified training samples.

**Algorithm 1:** The General Boosting Algorithm

**Input** :
Dataset samples: $\mathbb{D} \in \Re^{N\times(n+m)} = (X_1, Y_1), ..., (X_N, Y_N)$
% $n$ denotes the dimension of input data;
% $m$ denotes the number of classes (2 in binary classification case);
Base-learning algorithm: $\mathcal{L}$
Number of learning rounds to be the number of base models: $I$
**Output** : Ensemble network ($\mathbb{EN} = \{\mathcal{F}1, ..., \mathcal{F}i, ..., \mathcal{F}I\}$), a stack of base models
**Initialization**:
Sample distribution: $\mathcal{D} \in \Re^{1\times N}$
$\mathcal{D}_1 = \frac{1}{N}$ - The equal weight for all samples
**for** $i = 1$ to $I$ **do**
- $\mathcal{F}i = \mathcal{L}(D_i)$% Training weak learner from distribution $\mathcal{D}_i$ -
$\epsilon_i^{boost} = P_{x\mathcal{D}_t(\mathcal{F}i(x)\neq\mathcal{F}^{ground}(x))}$//Evaluate the error of $\mathcal{F}i$% -
$\mathcal{D}_{i+1} = Adjust\_Distribution(\mathcal{D}_i, \epsilon_i^{boost})$
**end for**
Inference : $CombineBoostingOutputs(X')$
% $X' \in \Re^{N\times n}$ denotes the testing samples

The procedure described in algorithm 1 is the general form of boosting. The unspecified parts such as *Adjust_Distribution* and *Combine_Ouputs* vary for every boosting-based ensemble algorithm. One of the notable boosting algorithms is Adaboost [38]

### 2.4.2 Bagging

Unlike boosting which generates base models in an iterative manner, in bagging methods, base models can be possibly generated in parallel as one base model is not dependent on previous base models. An example of this method is in [39]. The parallel mechanism is motivated by the fact that combining independent base models can reduce error dramatically due to the independence between base models. Bagging is also called bootstrap aggregation, which applies bootstrap sampling [40] to form several data subsets with replacement. Every subset is trained using a base-learning algorithm to form a base model.

Bagging embraces the most used methods for combining the inferences from the base models. The most notable one is the voting method in the case of the classification task and averaging for the regression task. For the inference task, the future samples are fed into all the base models, and the output from all the base models are voted and the winner labels are taken as the final prediction. The pseudocode of the bagging algorithm as depicted in Algorithm 2.

**Algorithm 2:** The General Bagging Algorithm

**Input** :
Dataset samples: $\mathbb{D} \in \Re^{N\times(n+m)} = (X_1, Y_1), ..., (X_N, Y_N)$
% $n$ denotes the dimension of input data;
% $m$ denotes the number of classes (2 in binary classification case);
Base-learning algorithm: $\mathcal{L}$
Number of base-models: $I$
**for** $i = 1$ to $I$ **do**
  $\mathcal{F}i = \mathcal{L}(\mathbb{D}, \mathcal{D}_{bootstrap})$ % $\mathcal{D}_{bootstrap}$ is the bootstrap distribution
**end for**
Inference:

## 2.5 An Offline Base-Learning Algorithm

### 2.5.1 ANFIS

ANFIS [37] is one of the well-known algorithms in the fuzzy systems area. Accurately, ANFIS is categorized as a neuro-fuzzy system (NFS) because it uses the hybrid intelligent system represented by a graphical network of Sugeno-Type Fuzzy Systems endowed with neural network learning capabilities. Due to its ability to cope with the non-linear systems of real-world applications, ANFIS has been used by many researchers to develop many variations of evolving neuro-fuzzy systems (ENFSs) such as PANFIS [11]. While PANFIS adopts an online working principle where its structure is evolving, the basic structure of PANFIS is adopted from ANFIS. Therefore, ANFIS architecture is worth discussion.

The main characteristic of ANFIS lies in the use of examples to define its membership function to construct a fuzzy inference system (FIS). An example of ANFIS's architecture consists of

two input variables $x$ and $y$ with $f(x, y)$ as an output. Three associated membership functions (MFs) for each input are illustrated in Fig. 2.4

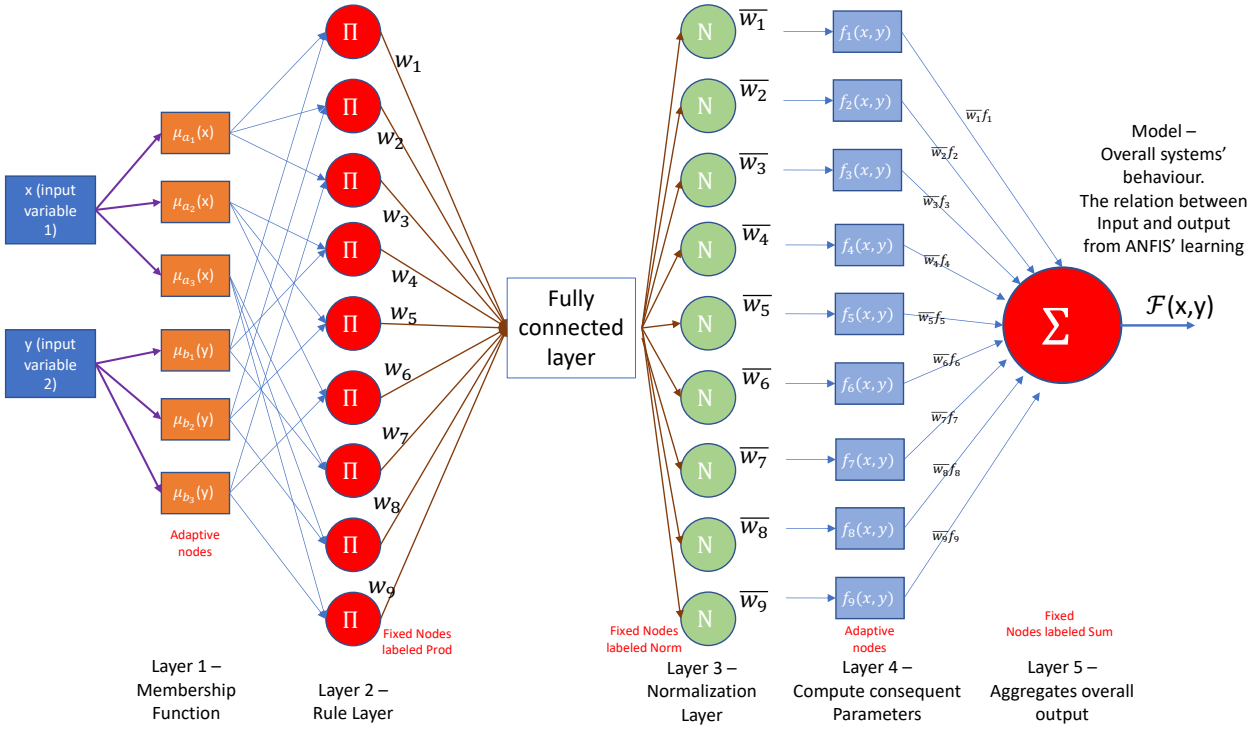


Figure 2.4: Architecture of ANFIS

The two fuzzy IF-THEN rules using the first-order Takagi-Sugeno fuzzy model [41] with nine rules are defined as follows:

Rule 1: if $x$ is $a_1$ and $y$ is $b_1$, then $f_1 = p_1x + q_1y + r_1$

Rule 2: if $x$ is $a_1$ and $y$ is $b_2$, then $f_2 = p_2x + q_2y + r_2$

Rule 3: if $x$ is $a_1$ and $y$ is $b_3$, then $f_3 = p_3x + q_3y + r_3$

Rule 4: if $x$ is $a_2$ and $y$ is $b_1$, then $f_4 = p_4x + q_4y + r_4$

Rule 5: if $x$ is $a_2$ and $y$ is $b_2$, then $f_5 = p_5x + q_5y + r_5$

Rule 6: if $x$ is $a_2$ and $y$ is $b_3$, then $f_6 = p_6x + q_6y + r_6$

Rule 7: if $x$ is $a_3$ and $y$ is $b_1$, then $f_7 = p_7x + q_7y + r_7$

Rule 8: if $x$ is $a_3$ and $y$ is $b_2$, then $f_8 = p_8x + q_8y + r_8$

Rule 9: if $x$ is $a_3$ and $y$ is $b_3$, then $f_9 = p_9x + q_9y + r_9$

where $a_i$, and $b_j$ denote the linguistic label or fuzzy sets of the first input and second input variable respectively, $i = \{1, 2, 3\}$ and $j = \{1, 2, 3\}$, the index of fuzzy set $a_i$ and $b_j$ respectively.

$p_r$, $q_r$, and $r_r$ represent the consequent parameters which can be obtained by ANFIS' training procedure using the examples as training data, whereas $r = \{1, ..., 9\}$. The number of rules is obtained from the combination of the number of fuzzy sets in all input variables.

ANFIS architecture basically consists of five layers as follows:

- Layer 1: Fuzzification layer - also called the membership function layer whose function is to determine the membership degree of the input variables to each fuzzy set for that particular input. An example of a membership function is a Gaussian function. Thus the membership degree of fuzzy set $a_i$ (using a Gaussian function) can be expressed as follows:

$$\mu_{a_i}(x) = \exp(-\frac{(c_i - x)^2}{2\sigma_i^2}) \tag{2.1}$$

  $\mu_{a_i}$ is the membership degree of first input $x$ with respect to $i^{th}$ fuzzy set. $c_i$ and $\sigma_i$ represent the center and width of fuzzy set $a_i$.

- Layer 2: Rule layer: where each node in the first layer ($\mu_{a_i}(x)$ and $\mu_{b_j}(y)$) calculate the firing strength of all rules. The function used can be MIN, PROD, AND, or any other fuzzy operation. Suppose that the function calculating the firing strength uses the PROD function. The firing strength of rule $r$ is denoted as :

$$w_r = \mu_{a_i}(x).\mu_{b_j}(y) \tag{2.2}$$

- Layer 3 : Normalization layer - This layer's function is simply to normalize the all rules' firing strength. The normalized firing strength of $r^{th}$ rule is denoted as :

$$\bar{w_r} = w_r / \sum_{r=1}^{9} w_r \tag{2.3}$$

  where $w_r$ denotes an $r^{th}$ rule's firing strength. $w_r$ is then normalised as a normalised firing strength $\bar{w_r}$.

- Layer 4: Defuzzification layer: In this layer, every node $v$ is an adaptive node and the output is the parameter output of $r^{th}$ rule denoted as :

$$\bar{w_r} f_r(x, y) = \bar{w_r}(p_r x + q_r y + r_r); r = \{1, .., 9\} \tag{2.4}$$

  where $p_r$, $q_r$, and $r_r$ are the parameters set for the $r^{th}$ rule/node. Parameters in this layer are called consequent parameters or the consequent function of the $r^{th}$ rule.

- Layer 5: Aggregation layer: This layer normally contains a single node that aggregates the overall outputs from the previous layer as the summation of all incoming signals denoted as:

$$\mathcal{F}(x, y) = \sum_{r=1}^{9} \bar{w_r} f_r(x, y) / \sum_{r=1}^{9} w_r \tag{2.5}$$

From Fig. 2.4 and the output of the ANFIS (layer 5), it can be seen that FIS which is depicted by the model denoted as $\mathcal{F}(x, y)$ represents the relationship between input and output variables $x$, $y$, and $z$ describing the overall system's behaviour.

### 2.5.2 Decision Tree

Decision tree is one of the popular supervised learning algorithm. Like ANFIS, decision tree algorithm can solve the problem of regression and classification. In decision tree, the training data is used to build the model in the form of decision rules the predict the class or value of the target variable.

The decision tree procedure starts from finding the first deciding node as a root. The root can be determined from several criteria, such as: Entropy, Information Gain, Gini Index, Gain Ratio, Reduction in Variance, and Chi-Square. These criteria are used to calculate values for every attribute.

Assume that the Information Gain is used as a criterion. The first step of building the tree is calculating the entropy. Entropy is known as a measure of the uncertainty about source of

messages. The entropy is formulated as follows:

$$Entropy = \sum_{i=1}^{c} -p_i * log_2(p_i) \tag{2.6}$$

where $p_i$ denotes the proportion of (collection S) belonging to $i^{th}$ class. When the value of entropy is small, it means that the less information required to define a class. Entropy is zero if all the members of S have the same class.

In order to determine the decision node, the Information Gain is calculated as follows:

$$Gain(S, A) = Entropy(S) - \sum_{v \in values(A)} \frac{|S_v|}{|S|} \tag{2.7}$$

where $A$ denotes the attribute, $S$ represents each value $v$ of all possible values of attribute $A$. $S_v$ denotes the subset of $S$ when attribute $A$ has value of $v$. $|S_v|$ and $|S|$ denote the number of $S_v$ and $S$ respectively.

The highest value of the criteria (e.g. Information Gain), is assigned as a root. Once root is obtained, a tree is built by creating the branch from the root. The splitting procedure is carried out based on both splitting rules and classification features. This procedure is repeated until the stopping criteria is reached (e.g. the value in the subset node has an equal value with the target variable).

## 2.6 Learning Mechanism of Offline Algorithms

This subsection introduces three standard offline algorithms: the batch algorithms, the distributed algorithms, and the ensemble algorithms. They adopt an offline working principle and utilize the static/fixed-size training data before the learning process.

### 2.6.1 The Batch Algorithms

The batch algorithms are commonly known as standard data mining algorithms. They use a fixed size training data for learning, having a static single model, and perform a standard testing task as depicted in Fig. 2.5. Some tasks and terms used in the batch algorithms are described as follows:

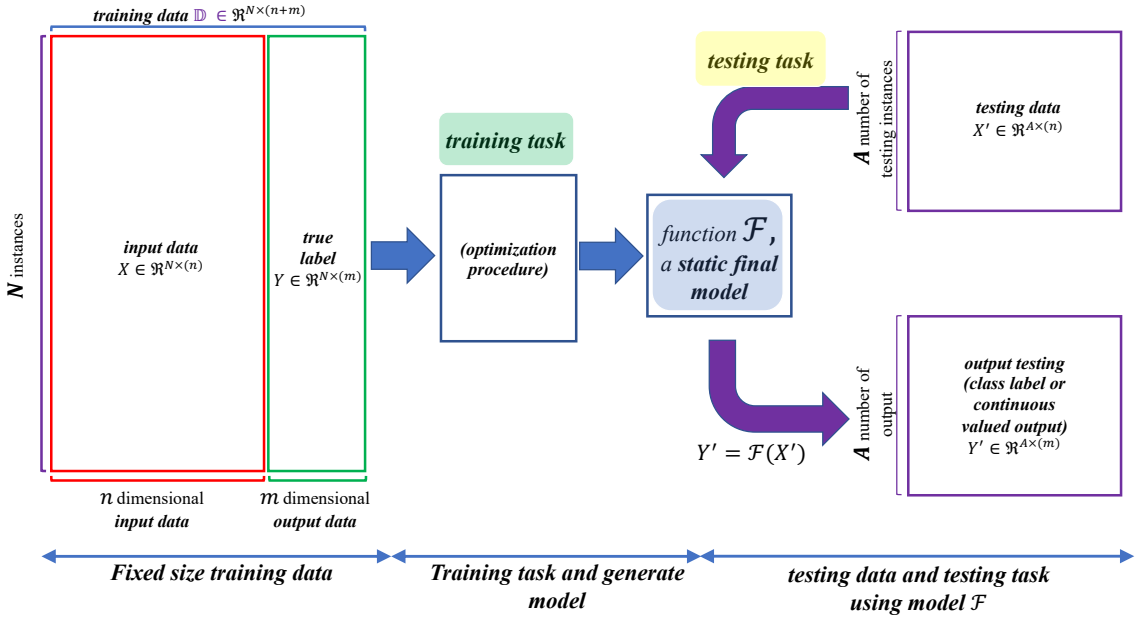


Figure 2.5: The standard batch algorithm architecture.

**Definition 2.1. Training data** $\mathbb{D}$ : The fixed number of observations consists of a pair of input data $X$ and true label $Y$ denoted as $\mathbb{D} = (X, Y)$. $X$ and $Y$ contain a set of instances $X = (X_1, .., X_i, ..., X_N)$ and $Y = (Y_1, .., Y_i, ..., Y_N)$ respectively. The properties of $\mathbb{D}$ are defined in Table 2.1.

**Definition 2.2. Training task** : The task of approximation of an unknown function $\mathcal{F} : X \rightarrow Y$ from the observations $\mathbb{D}$. $\mathcal{F}$ is searched using the optimization method so that the error $(\mathcal{F}(X) - Y)$, is minimized. The optimized function $\mathcal{F}$ is also called a model.

Note: $\mathcal{F}$ represents a model generated from the training task in general. In this thesis, this model is used for the inference task in a classification problem as a classifier.

Table 2.1: Properties of training data $\mathbb{D}$ used in the batch algorithms

| No | Term | Symbol | Dimension | Note |
|---|---|---|---|---|
| 1 | input data | $X$ | $X \in \Re^{N \times n}$ | $N$ denotes the fixed number of instances, $n$ denotes the dimension of the input data (static data) |
| 2 | true label | $Y$ | $Y \in \Re^{N \times m}$ | $m$ denotes the dimension of the true label (static data) |
| 3 | training data | $\mathbb{D}$ | $\mathbb{D} \in \Re^{N \times (n+m)}$ | training data is also called a pair of observations $(X, Y)$ which consists of input data $X$ and true label $Y$ (static data) |

**Definition 2.3. Testing data** $\hat{X} \in \Re^{A \times n}$ represents the unlabelled/unseen future data for the purpose of the testing task. This data is separate from the training data. Similar to the training data, it has $n$ dimension of input without labels, where $A$ represents the number of instances in the testing data.

**Definition 2.4. Testing task** : This is also called the inference task. This is the process when a model $\mathcal{F}$ generates an output testing $\hat{Y}$ from the given $\hat{X}$. This task is denoted as $\hat{Y} = \mathcal{F}(\hat{X})$. $\hat{Y}$ is the output in a form of a class label in the classification task or continuous valued output in the regression task.

### 2.6.2 The Ensemble Algorithms

The standard ensemble algorithms utilize a fixed size training data $\mathbb{D}$ in both the training and testing task. They use the multiple models to improve the system's overall performance. The architecture and the learning properties of the standard ensemble algorithms are depicted in Fig. 2.6 and Table 2.2 respectively.

**Definition 2.5. Ensemble training task**: A learning mechanism that involves several base-learning algorithms to learn either the same observations $\mathbb{D} = (X, Y)$, disjoint data partition, or overlapping data partition to generate several base models. The collection of base models

Table 2.2: Properties of standard Ensemble Learning

| No | Term | Symbol | Dimension | Note |
|---|---|---|---|---|
| 1 | Input data | $X$ | $X \in \Re^{N \times n}$ | Fixed size of input data<br>the same data as the batch algorithms<br>(static data) |
| 2 | True label | $Y$ | $Y \in \Re^{N \times m}$ | $k^{th}$ label (static data) |
| 3 | Base model | $\mathcal{F}i$ | NA | $i^{th}$ base model of ensemble network $\mathbb{EN}$. |
| 4 | Ensemble network | $\mathbb{EN}$ | NA | $\mathbb{EN} = \{\mathcal{F}1, ..., \mathcal{F}i, ..., \mathcal{F}I\}$<br>A stack of $I$ base models. |

is called ensemble network $\mathbb{EN} = \{\mathcal{F}1, ..., \mathcal{F}i, ..., \mathcal{F}I\}$, where $I$ denotes the number of base-learning algorithms or base models which exist in the $\mathbb{EN}$.

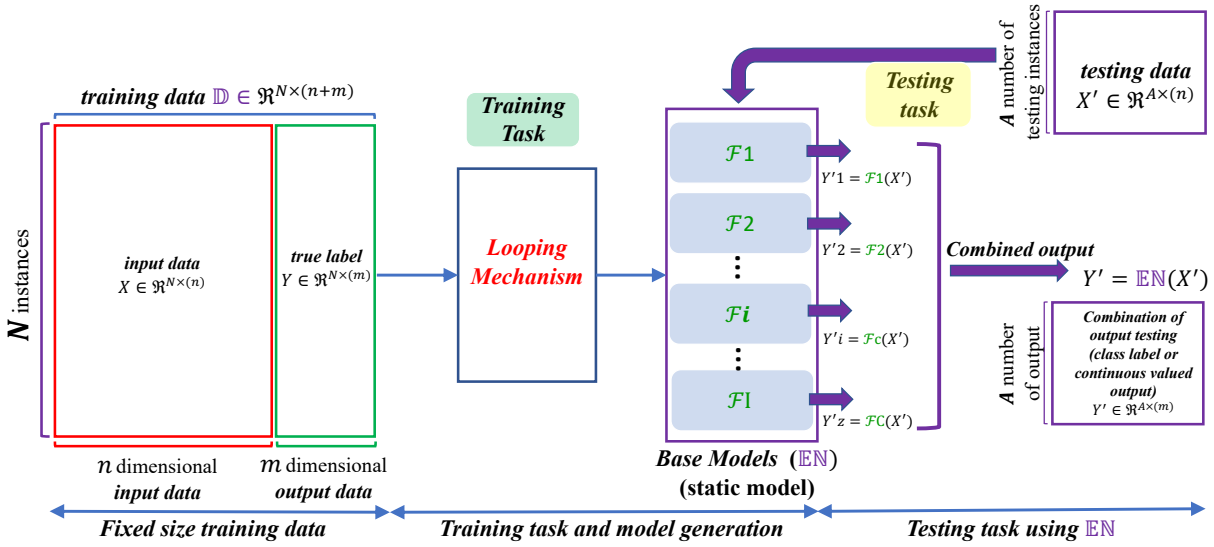


Figure 2.6: The standard ensemble algorithm architecture - standard training and testing scenario.

**Definition 2.6. Ensemble testing task**: The output of $i^{th}$ base model can be modelled as $\hat{Y}_i = \mathcal{F}i(\hat{X})$. The final output (which is denoted as $\hat{Y} = \mathbb{EN}(\hat{X})$) is the **combination** from all the base models' inferences after the voting weight mechanism is applied ($\hat{Y} = \sum_{i=1}^{I} \hat{Y}_i * \beta_i$). $\hat{Y}_i$ and $\beta_i$ are the inference output and voting weight of $i^{th}$ base model respectively, where the voting weight can be determined through the performance evaluation mechanism (e.g. penalty and reward scenario).

### 2.6.3 Distributed Algorithms

While standard batch algorithms are considered to be standard data mining algorithms, standard distributed algorithms are also introduced to expand the processing capacity of the batch algorithms by processing $\mathbb{D}$ in a distributed manner. Thus, the distributed algorithms are capable of processing a much larger scale of instances. Both batch and distributed algorithms are designed to process a fixed number of observations $\mathbb{D}$. The tasks and terms used in the distributed algorithms are defined as follows:

**Definition 2.7. Distributed training task**: A training task undertaken in a distributed manner using several ($\mathcal{Q}$) processing nodes. Suppose that a fixed number of instances of $\mathbb{D}$ consists of $X \in \Re^{N \times n}$ and $Y \in \Re^{N \times m}$. In the distributed algorithm task, $X$ and $Y$ are divided into $P$ partitions called **input data partition** $X^p \in \Re^{V \times n}$ and **true label partition** $Y^p \in \Re^{V \times m}$ respectively. The typical distributed algorithms learn all $P$ number of partitions of training data $\mathbb{D}^p$, and for each $\mathbb{D}^p = [X^p, Y^p]$, a local model $\mathcal{F}^p$ is generated resulting in the generation of **initial distributed models** $\mathcal{F}^{initial} = \{\mathcal{F}^1, ..., \mathcal{F}^p, ..., \mathcal{F}^P\}$ as the learning outputs.

$\mathbb{D}^p$ represents **a partition of training data**, where $p = \{1, 2, ..., P\}$, and $V$ denotes a number of instances in the partition of $X^p$, $Y^p$, and $\mathbb{D}^p$. Note that the $V$ can be obtained from $V \approx \frac{N}{P}$. A distributed learning platform (Spark) will automatically duplicate a small number of instances to some partitions if ($N$ mod $P \neq 0$) to make the number of $V$ even for all partitions.

In the distributed algorithm task, all $\mathbb{D}^p$s are distributed evenly into $\mathcal{Q}$ nodes in the cloud. Each node trains around $G$ number of $\mathbb{D}^p$ (in a serial manner), where $G \approx \frac{P}{Q}$. Please also note that the number of $G$ may not be even in all nodes (if ($P$ mod $Q \neq 0$)). However, Spark will automatically distribute all partitions as evenly as possible to all nodes. The architecture of the standard distributed algorithms is depicted in Fig. 2.7, and its properties is summarized in Table 2.3.

**Definition 2.8. Merging (local) model task**: The initial distributed models $\mathcal{F}^{initial}$ contain

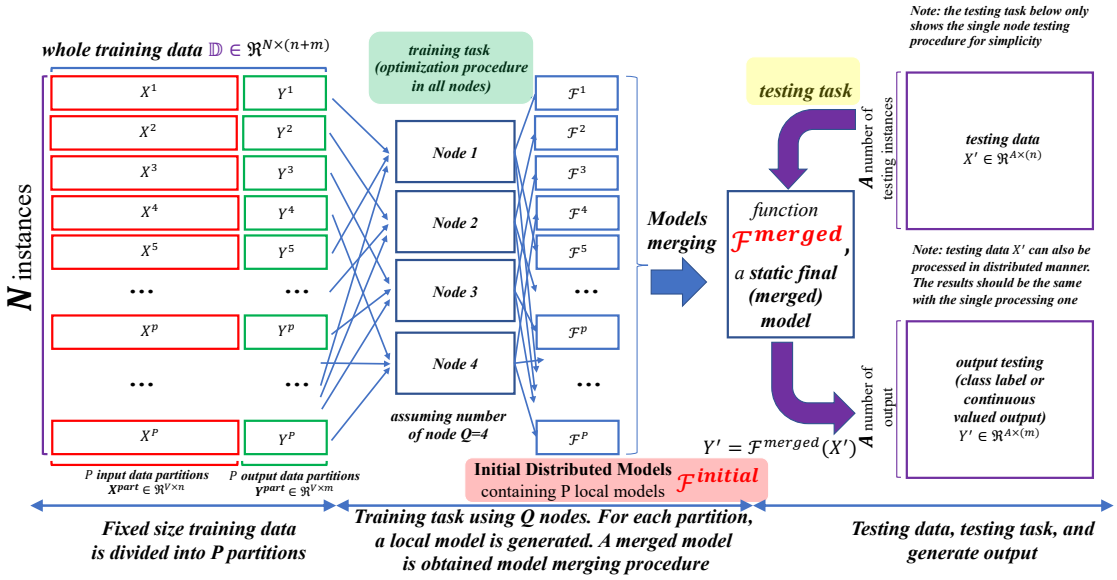


Figure 2.7: The Standard distributed algorithm architecture.

$P$ number of local models (**multiple models**). The model merging procedure aggregates initial distributed models $\mathcal{F}^{initial}$ into a merged model $\mathcal{F}^{merged}$ (**single model**). This procedure is denoted as $\mathcal{F}^{merged} = merge(\mathcal{F}^{initial})$.

**Definition 2.9. Distributed testing task**: The output $\hat{Y}$ can be obtained in two ways, either using a merged model $\mathcal{F}^{merged}$ or using the initial distributed models $\mathcal{F}^{initial}$. The former testing task is carried out by simply feeding the $\hat{X}$ into $\mathcal{F}^{merged}$ denoted as $\hat{Y} = \mathcal{F}^{merged}(\hat{X})$ as the testing task depicted in Fig. 2.7. The latter testing task is undertaken by feeding $\hat{X}$ into $\mathcal{F}^{initial}$ ($\hat{Y} = \mathcal{F}^{initial}(\hat{X})$) similar to the testing task conducted in the ensemble algorithms discussed in subsection 2.6.2.

Note : The initial distributed models $\mathcal{F}^{initial}$ structure is similar to the base models of ensemble network (𝔼𝕀𝕅). In other words, $\mathcal{F}^{initial}$ can be perceived as an 𝔼𝕀𝕅 because the structure of a local model in $\mathcal{F}^{initial}$ is the same as a base model in 𝔼𝕀𝕅. One distributed training task can be regarded as training of $P$ same type and same parameter of base-learning algorithms trained by the $P$ disjoint partitions of training data $\mathbb{D}$ using an ensemble algorithm.

In summary, while the typical batch, ensemble, and distributed algorithms have different learning schemes, they are similar in terms of the use of $N$ fixed instances of $\mathbb{D}$, which is known

Table 2.3: Properties of the distributed algorithms

| No | Term | Symbol | Dimension | Note |
|---|---|---|---|---|
| 1 | input data partition | $X^p$ | $X^p \in \Re^{V \times n}$ | $V$ denotes the number of instances in $X^p$ $p = \{1, 2, ..., P\}$ |
| 2 | true label partition | $Y^p$ | $Y^p \in \Re^{V \times m}$ | |
| 3 | partition of training data | $\mathbb{D}^p$ | $\mathbb{D}^p \in \Re^{V \times (n+m)}$ | $\mathbb{D}^p = (X^p, Y^p)$; a pair of observations $(X^p, Y^p)$ |
| 4 | local model | $\mathcal{F}^p$ | NA | A model of $\mathbb{D}^p$; an approximated function learned from $\mathbb{D}^p$ |
| 5 | initial distributed models | $\mathcal{F}^{initial}$ | NA | The initial distributed models; Stack of $P$ number of $\mathcal{F}^p$ |
| 6 | merged model | $\mathcal{F}^{merged}$ | NA | The final model in the distributed methods framework; |

before the training task. Their models ($\mathbb{EIN}$, $\mathcal{F}^{merged}$, and $\mathcal{F}$) are used to infer testing data $\hat{X}$.

In the case of data streams where instances arrive continuously, learning from data streams using these three types of algorithms is impractical as they are designed to learn static training data. To gain knowledge of all samples, a retraining phase should be carried out upon the arrival of new instances (including previously learned samples). However, the retraining phase could lead to the problem of catastrophic forgetting because data streams are generated from a non-stationary environment. For this reason, the algorithms suitable for this problem are the online algorithms which are explained in section 2.7.

## 2.7 Learning Mechanism of Online Algorithms

In the previous subsection 2.6, three types of offline algorithms: the batch, ensemble, and distributed algorithms whose corresponding models ($\mathcal{F}$, $\mathbb{EIN}$, $\mathcal{F}^{merged}$) are generated from a fixed training data, were presented. In this subsection, the online algorithms to learn from data streams are introduced.

**Definition 2.10. Data streams $\mathcal{S}$**: the sequential arrival of unbounded batches of training data streams denoted as $\mathcal{S} = \{\mathbb{B}_1.\mathbb{B}_2, ..., \mathbb{B}_k, ...\}$, where $\mathbb{B}_k = (X_k, Y_k)$, $\mathbb{B}_k \in \Re^{N\times(n+m)}$. $X_k \in \Re^{N\times n}$ represents the batch of input data, $Y_k \in \Re^{N\times m}$ represents the batch of true label, and $k$ represents the timestamp of arrival $\mathbb{B}_k$, $k = \{1, 2, 3, ..., K, ...\}$. $K$ is possibly an infinite number, and $N$ represents number of instances in $\mathbb{B}_k$.

From definition 2.10, it can be seen that learning from data streams requires algorithms which can process sequential data by incrementally updating their models in every $k^{th}$ timestamp. Furthermore, the algorithms must be adaptive so they can deal with the non-stationary trait of data streams which can only be handled by using incremental learning algorithms.

The online or incremental methods which were inspired by the concept of an incremental heuristic search [42] are deemed as appropriate solutions to handle data stream issues because they feature the continual learning property to learn data in a **task-by-task** manner. Basically, for every training task, they use the previous timestamp's model $\mathcal{F}_{k-1}$ to build the current timestamp model $\mathcal{F}_k$ given the new data streams $\mathbb{B}_k$.

The new data streams can either be in the form of batch or instance data. Their learning mechanisms can be called **batch incremental** for incremental learning algorithms and **instance incremental** for evolving algorithms which are depicted in Fig. 2.8 and Fig. 2.9 respectively. The formal mathematical point of view for the incremental learning model update is modeled as follows:

$$\mathcal{F}_k = \mathcal{L}(\mathcal{F}_{k-1}, \mathbb{B}_k); \mathbb{B}_k = (X_{k1,...,kN}; Y_{k1,...,kN}); \mathbb{B}_k \in \Re^{N\times(n+m)}, \tag{2.8}$$

where $\mathbb{B}_k$, $X_k$, and $Y_k$ denote training data, input data, and true label respectively, at $k^{th}$ timestamp. $N$ denotes the number of instances in the current timestamp. $n$ and $m$ denote the dimensional feature of the input variable of $X_k$ and the output variable of $Y_k$ respectively. Similarly, the formal mathematical point of view for the evolving algorithm's model update is modeled as follows:

$$\mathcal{F}_k = \mathcal{L}(\mathcal{F}_{k-1}, \mathbb{I}_k); \mathbb{I}_k = (X_k; Y_k); \mathbb{I}_k \in \Re^{1\times(n+m)}, \tag{2.9}$$

where $\mathbb{I}_k$, $X_k$, and $Y_k$ denote the instance stream, instance input stream, and instance label stream respectively at current ($k^{th}$) timestamp.

It is apparent from the learning mechanism's point of view that both the evolving algorithms and the incremental learning algorithms share a similar procedure, where they process the data streams sequentially. The difference between the evolving algorithms and the incremental learning algorithms lie in the number of instances processed at the $k^{th}$ timestamp. The incremental learning algorithms update their models per batch ($N >> 1$), whereas the evolving algorithms update their models per instance ($N = 1$).

Furthermore, unlike the standard batch algorithms where the labels are available prior to the training task, in data stream processing, data arrive without labels (labels are available after some time due to a delay in the labelling effort). This means that in data stream processing, for every timestamp, the testing task is carried out first followed by the training task. This procedure is called the prequential test-then-train task.

**Definition 2.11.** Prequential test-then-train task: The sequential learning of the data streams, where the testing task is conducted first followed by the training task for every timestamp in $\mathbb{B}_k$.

While prequential test-then-train can be used for both evolving algorithm and incremental learning, this term is commonly used for the standard procedure in incremental learning. The details of the prequential test-then-train task procedure in incremental learning is illustrated as follows. When $X_k$ arrives, the testing task is conducted first to infer $X_k$ which results in output testing batch $\hat{Y}_k$. The testing task is denoted as $\hat{Y}_k = \mathcal{F}_{(k-1)}(X_k)$. Then, the training task is conducted afterward denoted as $\mathcal{F}_k = \mathcal{L}(\mathcal{F}_{k-1}, \mathbb{B}_k)$. The architectural diagram for prequential test-then-train is depicted in Fig. 2.8.

### 2.7.1 The Incremental Learning Algorithms - Batch Incremental Learning Algorithms

For incremental learning algorithms, especially in the prequential test-then-train scenario, the learning procedure is described as follows. For every $k^{th}$ timestamp, the test-then-train task of a batch $\mathbb{B}_k$ is repeated and the batch model $\mathcal{F}_k$ is incrementally updated until the end of the batch (or possibly infinite $K^{th}$ batch). The the properties of the test-then-train task scenario is depicted in Table 2.4.

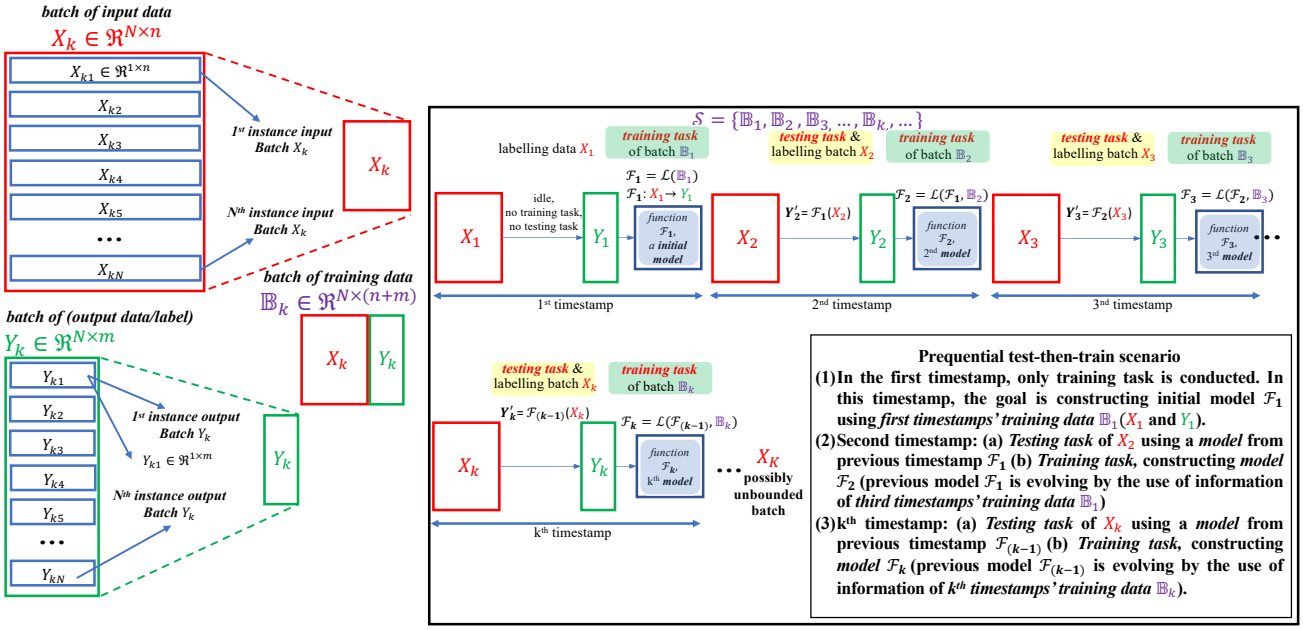


Figure 2.8: Online algorithm - standard batch incremental learning algorithm architecture using prequential test-then-train scenario

### 2.7.2 The Evolving Algorithms - The Instance Incremental Learning Algorithms

The slight difference between evolving algorithms and incremental learnings is that in the evolving algorithms, the model update is carried out instance-by-instance instead of batch-by-batch for every timestamp. Data fed in the $k^{th}$ timestamp is an instance called instance stream $\mathbb{I}_k$ instead of a batch of instances called the batch of training data $\mathbb{B}_k$.

Table 2.4: Properties of the Standard Batch Incremental Learning Algorithms

| No | Term | Symbol | Dimension | Note |
|---|---|---|---|---|
| 1 | batch of input data | $X_k$ | $X_k \in \Re^{N \times n}$ | $N$ denotes the number of instances in $X_k$ |
| 2 | an instance in a batch of input data | $X_{kt}$ | $X_{kt} \in \Re^{1 \times n}$ | $t$ denotes index of instance in a batch $X_k$ ; $t = \{1, 2, 3, ..., t, ..., N\}$ |
| 3 | a batch of true label | $Y_k$ | $Y_k \in \Re^{N \times m}$ | |
| 4 | an instance in a batch of true label | $Y_{kt}$ | $Y_{kt} \in \Re^{1 \times n}$ | $t$ denotes index of instance in a batch $Y_k$ ; $t = \{1, 2, 3, ..., t, ..., N\}$ |
| 5 | a batch of training data | $\mathbb{B}_k$ | $\mathbb{B}_k \in \Re^{N \times (n+m)}$ | a batch of training data consisting of $(X^k, Y^k)$ |
| 6 | timestamp model | $\mathcal{F}_k$ | NA | an evolving model at timestamp k. This model is evolving every timestamp adapting to the pattern of $\mathbb{B}_k = (X^k, Y^k)$ while also considering previous merged model $\mathcal{F}_{k-1}$ |

The learning scenario of the common **evolving algorithms** is described as follows. For every timestamp $k$, a test-then-train task of instance stream $\mathbb{I}_k$ is repeated and the model $\mathcal{F}_k$ is updated until the end of the batch where the number of timestamp $k$ is possibly unbounded. The architecture and the properties of the learning and evaluation scenario are depicted in Fig. 2.9 the Table 2.5 respectively.

Table 2.5: Properties of the Evolving Algorithms

| No | Term | Symbol | Dimension | Note |
|---|---|---|---|---|
| 1 | instance input stream | $X_k$ | $X_k \in \Re^{1 \times n}$ | k$^{th}$ instance input of unbounded data stream |
| 2 | instance label stream | $Y_k$ | $Y_k \in \Re^{1 \times m}$ | k$^{th}$ instance label of unbounded data stream |
| 3 | instance stream | $\mathbb{I}_k$ | $\mathbb{I}_k \in \Re^{1 \times (n+m)}$ | The pair of $(X_k, Y_k)$, a training data at k$^{th}$ timestamp $\mathbb{B}_k$, $\mathbb{B}_k = \mathbb{I}_k$, means a training data is an instance stream |

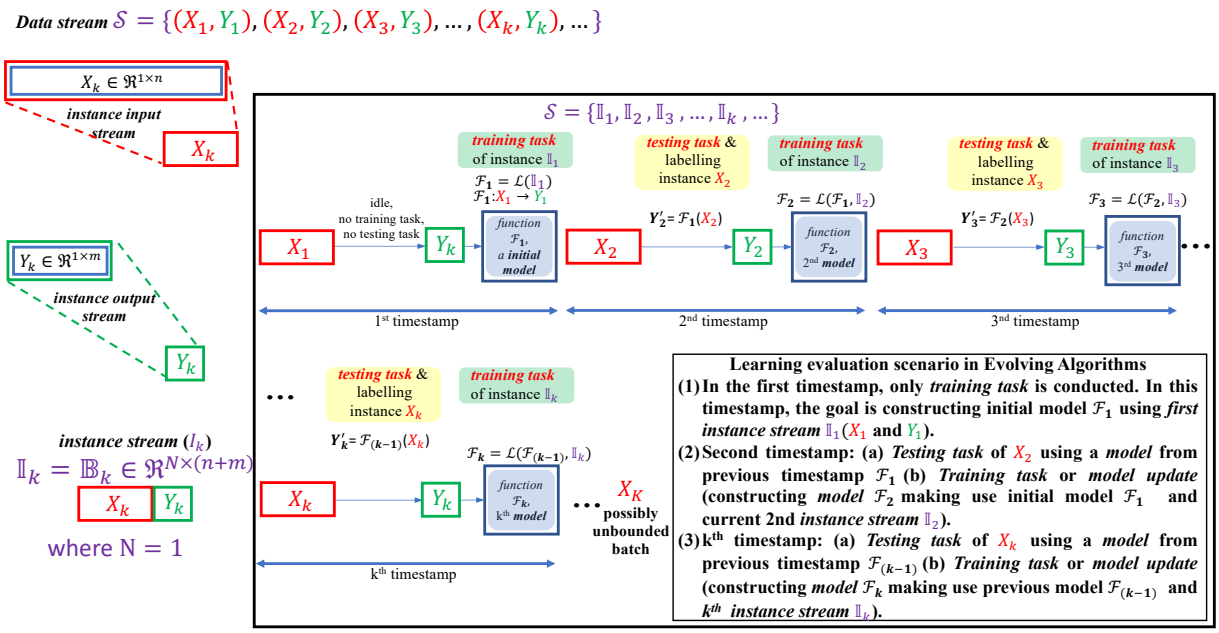


Figure 2.9: Online algorithm - evolving algorithm architecture - standard instance incremental learning algorithm

## 2.8 Towards the Implementation of Distributed Incremental Ensemble Algorithms

This subsection discusses two methods: incremental ensemble and distributed incremental ensemble algorithms. Both are categorized as incremental learning, but the latter runs in the distributed environment. The most challenging part of incremental learning algorithms is keeping their model updated at any timestamp without losing the generalization performance when the model is used to infer future data. The problem with distributed incremental learning algorithms is much more challenging due to the problem of structural complexity. That is, the initial output of distributed algorithms in each task is the initial distributed models whose structure needs to be simplified to warrant the incremental learning procedure.

### 2.8.1 Dynamic Structure of the Incremental Ensemble

The backbone of this algorithm is the architecture of the ensemble algorithms implemented in the incremental setting, as explained in subsections 2.6.2 and 2.7.1 respectively. An example of this kind of algorithm is applied in pENsemble [28]. In pENsemble, data arrive in batches as in the incremental learning algorithms. A prequential test-then-train is applied for every batch. In the testing part, an $\mathbb{EIN}_{k-1}$ infers $X_k$, and it produces the combination of output testing $\hat{Y}_k$ similar to the inference mechanism in the ensemble algorithms.

**The dynamic structure of incremental ensemble algorithms:** The structure of incremental ensemble algorithms evolve in every task. In [28], a new base model $\mathcal{F}^{I+1}$ is added into the existing $\mathbb{EIN}_k$ when drift occurs by employing a drift detection mechanism at the $k^{th}$ timestamp. Conversely, any statistical measurements can be applied as the pruning criterion to remove some of existing models in the $\mathbb{EIN}$ if they are deemed to be no longer significant to the ensemble network $\mathbb{EIN}$.

**An example scenario for the dynamic structure of the incremental ensemble algorithms:** At the $9^{th}$ timestamp, $k = 9$, a $9^{th}$ batch of input data $X_9$ arrives and is fed to the ensemble network (assuming that at the previous timestamp ($8^{th}$), the ensemble network has two base models) $\mathbb{EIN}_8 = \{\mathcal{F}^1, \mathcal{F}^2\}$. The update of the $\mathbb{EIN}$ in the $9^{th}$ timestamp is performed based on the following steps:

- **Testing task at batch 9:** The testing task is performed first following the testing procedure in the ensemble algorithm as shown in Fig. 2.6 resulting in the combination of output testing $\hat{Y}_9$ denoted as $\hat{Y}_9 = \mathbb{EIN}_8(X_9)$.
- **Model removal:** In this step, the models' performance is measured. The model removal procedure is based on the performance of each base model inside the $\mathbb{EIN}$. The insignificant base models are removed from the network.
- **Model selection or model (inside $\mathbb{EIN}_8$) selection:** Of the existing base models, only one model needs to be updated as the winning model. The selection mechanism is based

on the testing task performance in the current (testing task) step. For example, suppose the winning model is $\mathcal{F}^2$, thus $\mathcal{F}^2$ will be used as the model to be updated in the training task step.

- **The drift detection mechanism:** This method measures the drift rate in the current batch ensemble output $\hat{Y}_9$.

- **Training task or model update:** In this step, the ensemble network modifies its models based on the following procedures:
  (1) Update the winning model: Assuming that the batch 9 true label $X_9$ arrives, the batch of training data can be obtained $\mathbb{B}_9 = (X_9, Y_9)$. The winning model update is denoted as $\mathcal{F}^{update} = \mathcal{L}(\mathcal{F}^2, \mathbb{B}_9)$, $\mathcal{F}^2 = \mathcal{F}^{winning}$, which means that an updated model is obtained given a new batch of training data $\mathbb{B}_9$ using a winning model $\mathcal{F}^{winning}$.
  (2) Modify the network structure: (a) In a case where drift is detected, a new base model $\mathcal{F}^{update}$ is added to the ensemble network, so that $\mathbb{EN}_9 = \{\mathcal{F}^1, \mathcal{F}^2, \mathcal{F}^{update}\}$. (b) In the case that drift is not detected, the updated model replaces the winning model ($\mathcal{F}^2$), so that $\mathbb{EN}_9 = \{\mathcal{F}^1, \mathcal{F}^{update}\}$.

From the example above, it can be seen that the structural evolution in the incremental ensemble algorithms occur sequentially in data stream learning triggered by a drift. When drift is detected in the $9^{th}$ batch, the ensemble network evolves from $\mathbb{EN}_8 = \{\mathcal{F}^1, \mathcal{F}^2\}$ into $\mathbb{EN}_9 = \{\mathcal{F}^1, \mathcal{F}^2, \mathcal{F}^{update}\}$. Otherwise, if no drift is detected in the $9^{th}$ batch, the ensemble network evolves from $\mathbb{EN}_8 = \{\mathcal{F}^1, \mathcal{F}^2\}$ into $\mathbb{EN}_9 = \{\mathcal{F}^1, \mathcal{F}^{update}\}$. Note that in this case, the model's performance and drift detection task are evaluated at the end of the instance in the batch $\mathbb{B}_9$, thus the structural ensemble evolution is performed afterwards.

### 2.8.2 The Distributed Incremental Ensemble frameworks

Distributed incremental ensemble algorithms are the extension of typical incremental ensemble algorithms implemented in a distributed processing platform. Their dynamic structure is similar

to incremental ensemble algorithms in terms of the structural evolution mechanism, except both the training and the testing task are performed in a distributed manner instead of single-node processing, thus it requires a merging procedure once the local models are generated from all partition learning, as depicted in Fig. 2.7 (the distributed training task). The dynamic architecture of this algorithm is illustrated in Fig. 5.1. The learning scheme for this algorithm also uses the prequential test-then-train similar to the incremental ensemble algorithms.

**An example scenario for the dynamic structure of the distributed incremental ensemble algorithms (using the same scenario as the incremental ensemble algorithms):** The steps of the learning mechanism in each timestamp are summarized as follows:

- **Distributed Testing task:** At the $9^{th}$ timestamp, the distributed testing task is performed first denoted as a $9^{th}$, $\hat{Y}_9 = \mathcal{F}_8^{network;distributed}(X_9)$.
- **Model removal:** The model's performance is measured based on the distributed testing task performance. The superfluous models are removed.
- **Model selection:** This aims to determine which model (inside ensemble network at $8^{th}$ timestamp ) should be used for the model update process in the distributed training task. The selected model is called the winning model.
- **The drift detection mechanism:** The drift detection method used in this algorithm is similar to that of the one used in the incremental ensemble algorithms. This method is applied to measure the drift rate in the current batch output $\hat{Y}_9$ ($9^{th}$ timestamp).
- **Distributed training task and model update:** This step is the most challenging part due to the distributed nature of the training procedure. This step is broken down into several substeps:
  - (1) Update the winning model:

    (a) Pick the winning model - Suppose that the winning model in $\mathbb{EIN}$ is $\mathcal{F}^2$ ($2^{nd}$ base model).

(b) In the same way as for the incremental ensemble algorithms, the training data $\mathbb{B}_9$ can be obtained after the labels arrive.

(c) Divide batch $\mathbb{B}_9$ into $P$ partitions evenly and distribute them into $\mathcal{Q}$ nodes. In each partition, the training/model update is performed, $\mathcal{F}_9^p = \mathcal{L}(\mathcal{F}^2, \mathbb{D}_9^p)$. $\mathcal{F}_9^p$ denotes $p^{th}$ local models at $9^{th}$ timestamp.

(d) Collect all $P$ local models. The collection of all $P$ sub-models is denoted as $\mathcal{F}_9^{initial} = \{\mathcal{F}_9^1, ..., \mathcal{F}_9^p, ..., \mathcal{F}_9^P\}$.

- (2) Merging all the collected $P$ local models to generate a merged model: merged model $\mathcal{F}_9^{merged} = Merge(\mathcal{F}_9^{initial})$ as a result of the distributed algorithms of batch $\mathbb{B}_9$. Previously, we had a winning model at $8^{th}$ timestamp $\mathcal{F}^2$ ($2^{nd}$ model).
- (3) Model Update : candidate model $\mathcal{F}^{update} = \mathcal{F}_9^{merged}$.
- (4) Modify the ensemble network structure: (a) In the case that drift is detected, the new model $\mathcal{F}^{update}$ is added to the ensemble network, so that $\mathbb{EN}_9 = \{\mathcal{F}^1, \mathcal{F}^2, \mathcal{F}^{update}\}$. (b) In the case that drift is not detected, the updated model replaces the winning model, so that $\mathbb{EN}_9 = \{\mathcal{F}^1, \mathcal{F}^{update}\}$.

## 2.9 Summary

This chapter presents the related backgrounds (e.g. terms, notations, and architectures) used in the large-scale data stream framework. In general, we utilize online algorithms because we deal with the problems of data streams. We categorize online algorithms into two categories based on the number of sample(s) learned in every task. The first category is instance incremental learning, which updates its model in every instance. The second is batch incremental learning, which learns and updates its model in every batch/task. In this thesis, we use the term **evolving algorithm** to refer to instance incremental learning. The term **incremental learning** refers to batch incremental learning.

A **base-learning algorithm** is introduced as a learning and inference engine component for

both ensemble algorithms and distributed algorithms. Once a base-learning component is used in either ensemble or distributed algorithms, the learning output is the multiple models. While the ensemble methods aim to increase accuracy by combining multiple models, the distributed methods aim to speed up the single processing node algorithm. When an evolving algorithm is implemented as a base-learning algorithm in a distributed training task, the output is the evolving initial distributed models. These models contain multiple models similar to the ensemble model structure. The initial distributed models can be processed in two ways: model merging and a combination mechanism (e.g. weighted voting, majority voting). The testing mechanism can be carried out either using a merged model obtained from the model merging or using the initial distributed models directly, similar to the testing task of the ensemble algorithms. For the case of using the initial distributed models, the final output is obtained from the combination of outputs from several models.

Batch incremental learning algorithms aim to learn the continual data streams. One of the prevalent methods in incremental learning are the incremental ensemble algorithms. In the literature, some of the earlier methods are built based on static EN without structural evolution. Recently, incremental learning with the dynamic evolution of EN was proposed, and has been proven to be effective in handling concept drift. The dynamic evolution of EN is also further enhanced into its distributed version, a distributed incremental ensemble algorithm.

In the next chapter, we present the literature review and the research gap between our frameworks and the existing online learning algorithms.

# 3 Literature Review

This chapter presents a review of the related literature used in this thesis. Section 3.1 presents the current development of the evolving algorithms based on fuzzy systems and distributed algorithms followed by identifying the research gap between them. In section 3.2, the types of algorithms in incremental learning, particularly ensemble-based batch incremental learning algorithms are reviewed followed by a discussion of the research gap.

## 3.1 Learning from Data Streams in a Distributed Environment

In this section, we present the state-of-the-art algorithms related to the distributed EFS framework and the underlying research gap in the framework.

### 3.1.1 The Current Development of Evolving Fuzzy Systems

The general term for evolving algorithms in the literature is evolving intelligent systems. Overall, the research on evolving algorithms has enjoyed great success for the last two decades, as evidenced by the appearance of seminal works conducted by the soft computing community for both evolving neural networks [43], evolving fuzzy systems [44], and evolving neuro-fuzzy systems [45].

One of the works in the evolving fuzzy systems area was pioneered by dynamic evolving neural-fuzzy inference systems (DENFIS) [46]. This work was based on the evolving structure of fuzzy systems architecture employing a recursive clustering method namely the evolving clustering method. Following DENFIS, another seminal work, namely evolving Takagi-Sugeno fuzzy systems (eTS) was carried out in [47]. eTS makes use of the potential of new incoming data to recursively update its models' structure which was later extended in evolving Takagi-Sugeno fuzzy systems from streaming data (eTS+) by introducing the concept of rule ages, utility function, and zone of influence. Yager's participatory learning concept [48] was introduced to extend eTS in the work of on evolving participatory learning (ePL) [49] and sequential adaptive fuzzy inference systems (SAFIS) [50], where SAFIS introduced the fuzzy rule contribution to the system output concept to define the fuzzy rule's influence. Rule contribution was later used as the basis on which to remove rules from the system to make the system more compact.

Further, in the area of fuzzy systems, several seminal works using more advanced approaches such as a variation of rule merging, generalized rules, incremental feature weighting, input pruning scenario etc. were introduced in [11, 32, 51, 52, 12, 53, 54, 55, 56]. In this thesis, a seminal EFS, namely a parsimonious network based on a fuzzy inference system (PANFIS) [11] is used as a base-learning algorithm in our distributed EFS. A comprehensive review of EFS can be found in [16].

### 3.1.2 Distributed Algorithms

Distributed algorithms were initially developed to cope with the problem of big data stored in temporary storage, as a result of the arrival of data streams from real-time world applications. MapReduce [21] was one of the first groundbreaking distributed computing platforms and is capable of processing data on a large-scale.

MapReduce controls the parallelization for the computation across the storage (large-scale database cluster) with its functions named Map and Reduce. Even though MapReduce is no longer popular as a distributed processing platform, it has been the inspiration for the

development of other platforms. Spark [23] has become one the most popular platforms in distributed processing, particularly for scalable machine learning tasks. Unlike MapReduce, Spark provides a built-in library which makes it possible for users to use it directly. Moreover, its speed is much faster than other available platforms due to the use of memory clusters for storing temporary datasets. However, their built in algorithms are still based on traditional algorithms which are offline in nature.

Big data is often stored on the cloud due to its large volume to support the extensibility and scalability of local storage. In order to efficiently extract valuable information from big data, there is an urgent need to modify existing data mining techniques, so they are scalable in processing large-scale datasets. This issue led to the need to develop a distributed or parallel scenario to process big data. Big data are also generated by the continuous arrival of new instances either in batches or one by one, which is known as a data stream and is generated by real-world applications [57, 58]. Therefore, it is necessary for a machine learning algorithm to adapt to a rapidly changing non-stationary data stream. Stream processing/mining in the web news domain was conducted in [18] using eT2Class [59], which is able to handle streaming data [60]. This triggered the development of evolving learning algorithms, which are able to learn big data continuously [1] by evolving its model to adjust the shift and drift of big data patterns.

### 3.1.3 Research Gap

In the era of big data, the processing speed of data analytics is paramount. Initially, several algorithms were developed to cope with the volume of big data by increasing the scalability of machine learning algorithms as discussed in [61, 62, 63, 64]. Recently, a large amount of attention has been focused on the development of big data analytics by considering the velocity and volume of data streams. For example, big data stream clustering [65, 66] processes big data by distributing the data stream for processing in many processors/nodes to speed up the computational time. Despite the increase in computational speed, these works still suffer from

changing data patterns, which in turn, possibly lead to the fusion of non-related clusters [67]. This means that the final generated model does not represent the knowledge of the data, which can lead to misleading inferences in prediction and clustering results.

The development of algorithms and techniques which are scalable for handling large-scale data streams and can also cope with the changing environment is crucial. This procedure enables data streams to be processed in a parallel mode by distributing them into many processors, where each processor processes/learns a block of a data stream (data chunk). In addition, it also processes every data chunk in an online mode with the capability of changing its data pattern incrementally.

To the best of our knowledge, only a few studies have addressed online big data processing and analytics in both distributed and online modes. Parallel_TEDA was introduced in [68], and is based on the parallel mode which processes every data chunk in a single pass mode using its evolving characteristic which is inherited from an evolving algorithm, namely typicality based empirical data analysis (TEDA) [69], which supports the recursive form of calculation and is parameter free. Even though Parallel_TEDA applies a parallel strategy and an evolving algorithm to improve the scalability and speed of the learning process, its model is still generated using a simple fusion technique, which can cause a redundancy problem.

Distributed algorithms have emerged as methods to solve the problem of data volume which grows beyond the capabilities of a single processing node. Big data platforms such as MapReduce [21] and Apache Spark [23] have been introduced in the last decade to scale out computation from single to multiple processing nodes to perform the distributed learning task. Work on distributed learning was conducted in [20] where distributed fuzzy rule-based classification systems (FRBCSs) are distributed [70] utilizing MapReduce. Moreover, the Spark-based distributed random forest algorithm was introduced in [3]. Distributed learning with the feature selection method was conducted in [71] to reduce system complexity. Association rules in large-scale data based on the genetic programming algorithm were proposed in [72]. Nonetheless, they were built based on an offline algorithm. The gaps and/or challenges of the existing

evolving algorithms are summarized as follows:

- Most of the evolving algorithms are built based on single-node processing so they are unable to cope with the problem of data explosion when data grows beyond the processor's capability to process it. To address this problem, an evolving algorithm needs to be developed in a distributed manner.

- One of the most challenging problems in distributed processing is aggregating the models generated from different training data partitions. To address this problem, the robust model merging procedure is designed to deal with large-scale distributed data stream training. The merging procedure should be handled carefully to reduce system complexity and the redundancy problems among the model components (rules).

## 3.2 Incremental Learning Based on Ensembles in a Distributed Environment

Incremental ensemble algorithms are one of the popular types of incremental learning algorithms. In this subsection, we divide the review into two topics: ensemble algorithms and the challenges of coping with the data stream problems, and the research gap.

### 3.2.1 Ensemble Algorithms and the Challenges

The original ensemble algorithm was introduced by Dasarathy and Sheela in [73] where a fixed number of models are built from different partitions of training data (the partition used in forming each model ) which may overlap each other. The ensemble algorithm was initially built for a static system where high accuracy was the main purpose without considering its scalability [74]. Further, many seminal ensemble algorithms have been introduced in either multiple-model same type models (bagging [39], boosting (AdaBoost) [38] and a mixture of expert

classifiers [75, 76]) or multiple-model different types of models such as in [77]. In this thesis, distributed incremental ensemble algorithms utilize multiple models of same type using PALM [32] as a base-learning algorithm similar to the incremental ensemble algorithm implemented in pENsemble [28].

There are three main challenges in building ensemble algorithms [78]: (1) data sampling/selection; (2) training the base models in the ensemble network; and (3) combining the base models to determine the aggregated output. The first challenge, data sampling selection, is to achieve diversity among models, which can be done in several ways, such as using different subsets of training data, using different subsets of the available features to train each model, known as the random subspace method as in [79], using different sets of parameters of the model, or different types of models inside the ensemble network. The second challenge is related to the strategy used to train models, where methods such as bagging, boosting, stack generalization and hierarchical MoE are the most frequently used approaches to train the ensemble's models. To address the third challenge, methods such as majority voting, and weighted majority voting can be used to combine the predictions from each base model. The algebraic combiners and weighted average, minimum/ maximum/ median/ rule, product rule, generalized mean, and decision template can be used to combine the continuous output of several models.

Furthermore, in [30], the taxonomy of ensemble learning from data streams, especially in supervised learning for the classification task, is divided into four categories: 1) batch incremental ensemble algorithm stationary environments; 2) instance incremental ensemble algorithm stationary environments; 3) batch incremental ensemble algorithm non-stationary environments; and 4) instance incremental ensemble algorithm non-stationary environments. As we focus on the third category, concept drift adaptation is employed from the new batch data to create a new model. In this way, ensemble network base models may represent a mixture of different distributions (concepts) that are present in the data stream because their models are formed from training from different chunks. Furthermore, the removal of the model in the ensemble network can also be done by evaluating the performance output of every model, where the poorest performing model is removed. This offers a way to dynamically control the number of

models inside the ensemble network to maintain the network's system complexity.

### 3.2.2 The State-Of-The-Art and the Research Gap

Most of the earliest batch incremental learning algorithms feature a dynamic number of models based on the evaluation of each testing and training for each timestamp. The streaming ensemble algorithm (SEA) [13] as a composite model (e.g. classifiers if the task is a classification task) system is one of the earliest incremental ensemble learning algorithms. It iteratively replaces the poorest performing model with a new one in every timestamp. Accuracy weighted ensemble (AWE) [80] works by assigning a weight to all models based on their prediction. A new model is created from the latest training chunk and the existing models whose performance is below this new model are removed. There are several other incremental ensemble algorithms such as dynamic ensemble pruning [81], and the recently published pENsemble [28]. However, the way they maintain the number of ensemble models, either with a static or dynamic structure and how they combine the network output is different.

While various incremental ensemble algorithms have been designed, their base models are generated from the static base-learning algorithm. Evolving algorithms were used as base-learning algorithms in incremental ensemble algorithms in [13, 82]. They processed training data in a single pass manner so unbounded data can be processed without a retraining mechanism. The advantage of using evolving algorithms as a base classifier is that they can increase network adaptability, where a classifier can adapt to a changing data distributions.

The further development of incremental learning lies in the adaptability of its structure, where most the incremental learning algorithms are built based on the static ensemble network where the number of its base models are fixed. The fixed network hinders incremental learning in making its base classifiers diverse, a condition where the network classifiers' decision boundaries are significantly different from those of the others which may improve its performance. In the fixed ensemble network, although the classifier's diversity can be reached through the use of different user-defined parameters, different data partitions, and a combination according to

[28, 83, 78], concept drift remains the major issue due to the fixed structure of the ensemble network.

A recent incremental learning approach, pENsemble [28], an evolving ensemble fuzzy classifier, offers a solution of a flexible network structure, where its base classifier is built from an evolving classifier (pClass [84]) and its ensemble network is also dynamic. While intensive research has been undertaken in the area of incremental learning to cope with data stream problems, the issue of scalability remains an open issue because the proposed solutions are mostly designed in a single processing node.

The research gaps in the distributed incremental ensemble algorithms are summarized as follows:

- Multiple models of ensemble systems in batch incremental learning have been proven to be effective in handling concept drift which exists in data streams [78]. While common ensemble systems have some advantages, they are not scalable because they are designed for small datasets which cannot cope with the problem of continuous data. In addition, the merging mechanism of local models in every timestamp/task must be handled carefully.

## 3.3 Summary

This section presents a review of the state-of-the art approaches related to our proposed algorithms (distributed EFS and the distributed incremental ensemble algorithms). To summarize and comprehend between our algorithms, we provide an overview of several types of algorithms that related to our work in Table 3.1.

For the distributed EFS, the current work on fuzzy-based evolving algorithms is reviewed, followed by a review of the distributed algorithms. The second proposed method, a distributed incremental ensemble algorithm, the state-of-the art of the ensemble methods is reviewed. Then, the study of incremental learning based on ensembles with the elastic structure is also pre-

Table 3.1: Overview of several types of algorithms

| No | Type of Algorithm | Online learning mechanism | Deep Structure Network | Dynamic Network Structure | Architecture | Form of Model | Additional Notes | Reference |
|---|---|---|---|---|---|---|---|---|
| 1 | Batch Algorithms | ✘ | ✘ | ✘ | Offline base-learning algorithm working on **single node** | Rules, Trees, Nodes, etc. as a single base model | (1)**Fixed structure of single model**; (2)Only can learn **static data** | Example: ANFIS, Decision Tree Architecture: Subchapter 2.6.1 |
| 2 | Distributed Algorithms | ✘ | ✘ | ✘ | Offline base-learning algorithm working on **multiple nodes** | Rules, Trees, Nodes, etc. as a single base model | (1)**Fixed structure of single model** as a **merged model** generated from a collection of local models; (2)Only can learn **static data** | Example: ANFIS, Decision Tree, which are attached to the the Spark platform Architecture: Subchapter 2.6.3 |
| 3 | Ensemble Algorithms | ✘ | ✔ | ✘ | Offline base-learning algorithm working on **single node** forming a multiple models | Stack of base models | (1)**Fixed structure of multiple models**; (2)Only can learn **static data** | Example: ANFIS, Decision Tree, which are attached to the ensemble learning mechanism such as bagging and boosting Architecture: Subchapter 2.4 Subchapter 3.2.1 |
| 4 | Evolving or Instance Incremental Algorithms | ✔ | ✘ | ✘ | Online base-learning algorithm working on **single node** | Rules, Trees or Nodes as single **evolving** base model | (1)Adaptive single model; (2)Features **continual learning** | Example: PANFIS, PALM Architecture: Subchapter 2.7.2 Subchapter 3.1.1 |
| 5 | Incremental Ensemble Algorithms | ✔ | ✔ | ✔ | Online base-learning algorithm working on **single node** processing data batch-by-batch forming multiple models | Stack of **evolving** base models | (1)**Dynamic structure of multiple models** (2)Features **continual learning** | Example: pENsemble Architecture: Subchapter 2.8.1 Subchapter 3.2.1 |
| **6** | **Distributed Evolving Algorithm** | ✔ | ✘ | ✘ | Online base-learning algorithm working on **multiple nodes** forming multiple models | single **evolving** base model from merging model mechanism | (1)**Adaptive single model** as a **merged model**; (2)Features **continual learning** | Our proposed method Subchapter 3.1.3 Implementation: Chapter 4 |
| **7** | **Distributed incremental Ensemble Algorithm** | ✔ | ✔ | ✔ | Online base-learning algorithm working on **multiple nodes** processing data batch-by-batch forming multiple models | Stack of evolving base models. Each base model is formed from merging mechanism | (1)**Dynamic structure of multiple adaptive models.** Each model is formed from a merged model constructed during distributed learning; (2)Features **continual learning** | Our proposed method Subchapter 2.8.2 Implementation: Chapter 5 |

sented. Finally, the research gap between the incremental ensemble algorithms in a distributed environment is presented. In the next chapter, the implementation of thesis contribution is discussed.

# 4

# Evolving Large Scale Data Stream Analytics Based On PANFIS - Scalable PANFIS

## Abstract

In this chapter, a type of distributed evolving fuzzy systems (distributed EFS), namely an evolving large-scale data stream analytics framework based on a parsimonious network of a fuzzy inference system (Scalable PANFIS) is proposed. Scalable PANFIS makes use of the PANFIS evolving algorithm distributed across Spark computing nodes. The Scalable PANFIS inherits evolving algorithm traits which are adaptive to the new instance patterns and capable of demonstrating the single-pass learning mechanism. Furthermore, the distributed learning mechanism allows Scalable PANFIS to deal with the vast volumes of training data streams. The main feature of Scalable PANFIS over the single processing node PANFIS lies in the fast generation of the **evolving distributed model** without suffering from a loss of accuracy. On the training side, the evolving distributed model is obtained through two main steps: 1) the generation of **initial distributed models** as a result of learning from several data stream partitions in a task; 2) **model merging** which aggregates the initial distributed models into a **merged model**. The Scalable PANFIS framework also features active learning (AL) embedded to PANFIS to further accelerate the training process of data stream partitions. The majority voting method is also utilized as an alternative to model merging to be used for testing. On the testing side, the output of the Scalable PANFIS can be obtained from either using a model

merging mechanism or using a majority voting mechanism. The combination of the Scalable PANFIS' training scheme in terms of initial distributed model generation (PANFIS with or without AL) and the testing scheme (model merging or majority voting) resulted in four types of Scalable PANFIS structures. Extensive experiments on this framework are validated by measuring the accuracy and running time of four types of Scalable PANFIS and other Spark-based built-in algorithms. The results indicate that Scalable PANFIS with AL improves the training time to be almost two times faster than Scalable PANFIS without AL. The results also show that both the model merging and majority voting mechanisms yield similar accuracy in general among Scalable PANFIS algorithms, and they are generally better than Spark-based algorithms. In terms of running time, the Scalable PANFIS training time outperforms all the Spark-based algorithms when classifying a multi-class label dataset.

## 4.1 Introduction

Large-scale data stream analytics has become one of the emerging areas in data science [85, 86, 87]. A large volume of data in many forms (e.g. text, picture, sound, video, signals etc.) can be generated from numerous sources (e.g. IoT, Web 2.0, and social networks) in the internet era. This information is essential for many companies/corporations to support their urgent decision-making to ensure they maintain their competitive advantage.

Extracting valuable knowledge from large-scale data streams is challenging due to its 5V characteristics: volume, velocity, variety, value, and veracity [1]. In fact, data streams are mostly generated by real-world applications which arrive continuously in non-stationary environments in a rapid and high-volume condition. Hence, it is paramount to obtain knowledge from large-scale data efficiently (in a reasonable timeframe without a reduction in the algorithm's accuracy) [88].

Large-scale data stream analytics problems can be solved in two ways: 1) distributed algorithms ; and 2) evolving algorithms [8]. Distributed algorithms focus on how to distribute/parallelize

data processing from single-node into a multi-node processing framework [89], thus accelerating the learning time. The evolving algorithm processes/learns data at high speed, in a single pass, and online manner. Its structure is evolving following an update of the current instance. The evolving algorithm does not require historical data as the current information/pattern/model is updated after the last instance has been learnt. This feature can help to reduce the amount of stored data because the historical data can be discarded.

Recent work on large-scale data analytics was reported in [20], utilizing the MapReduce [21] method. In this work, the distributed algorithm used to learn the data partition was still based on the offline algorithm, namely Fuzzy Rule-Based Classification Systems (FRBCSs) [70] to model complex problems. However, the offline learning of FRBCS is not efficient, especially in handling rapidly varying and large-scale data streams. On the other hand, processing large-scale data using a single-node evolving algorithm is limited by the memory and bandwidth of a single machine. This issue remains the main challenge for further developments in large-scale data analytics. Considering the benefits of both distributed algorithms and the online learning mechanism, a large-scale data stream analytics framework should accommodate the scalability of distributed algorithms and the efficiency of an evolving algorithm.

In this work, we propose an evolving large-scale data stream analytics framework based on Scalable PANFIS, where PANFIS [11] is a seminal evolving algorithm based on a hybrid neuro-fuzzy system (NFS) which has the capability of learning data streams in the single pass mode to cope with the velocity and changing patterns of data stream. As an evolving version of NFS, PANFIS is capable of learning the non-linear systems in real-world applications. A Scalable PANFIS framework learning and inference mechanism involves three methods: 1) the AL which is embedded along with PANFIS to generate the fast initial distributed models which contain many local models as a result of learning from many partitions of training data; 2) model merging to aggregate the initial distributed models; and 3) majority voting to combine all the local models' output.

The training phase of this framework is conducted by distributing the PANFIS algorithm (with

or without AL) across the worker nodes. AL is the method embedded in PANFIS whose function is to accelerate the learning process by selecting only the important instances of training data. Note that PANFIS (with or without AL) operates on the local level, which means it learns a partition of the data stream and yields a local model. The output of Scalable PANFIS is the initial distributed models as depicted in Fig. 4.1. A model merging procedure is developed in Scalable PANFIS to aggregate the initial distributed models into a merged model. The output of the system can be obtained from either the inference of a **merged model** or **combination of the initial distributed model inferences** to the testing data using majority voting.

To summarize, the main contributions of this work are as follows:

- We present a Scalable PANFIS framework, an evolving large-scale data stream analytics framework, a distributed EFS, which can deal with large-scale data stream classification problems. This framework is scalable/distributable and it can cope with the changing patterns of data streams.
- We present the robust model merging method to solve the aggregation problem in large-scale distributed data stream training, where initial distributed models (the collection of local models) are generated. Each local model consists of one or more rules as the representation of a training data partition. The extracted rules are obtained from concatenation of rules in the initial distributed models. The extracted rules cannot be used as components in the model merging process. This is because the distribution of training data partitions are different to each other. Our model merging method can solve the aggregation problem and yields a stable performance (accuracy) for all datasets.

The rest of the chapter is organized as follows. Section 4.2 discusses the related research: PANFIS algorithm and its learning policy. Section 4.3 discusses the Scalable PANFIS architecture and problem formulation. Section 4.4 explains the architecture of the Scalable PANFIS framework with its four structures. Section 4.5 discusses the numerical study: experimental objective, experiment setup, and the results of evaluating large-scale data analytics. Section 4.5.3 presents the discussion and section 4.6 concludes the chapter.

## 4.2 PANFIS

PANFIS is an evolving algorithm which is built based on evolving neuro-fuzzy systems (ENFS), an extension of the well-known classic neuro-fuzzy systems (NFSs) [90] such as ANFIS [37]. NFSs combine fuzzy systems which imitate human reasoning and neural networks which have a learning ability, parallelism, and robustness characteristics. Basically, ENFSs are the evolving version of NFSs and have the capability of evolving their structure (rules) so they can adapt to the changing environment, which is essential to cope with the non-stationary environment of data streams.

The PANFIS evolving algorithm can learn the data without an initial structure. During the learning, its structures (fuzzy rules and its parameters) are evolving, so the new rule can be generated, updated, and pruned. In addition, the merging process is carried out by identifying identical (or similar) fuzzy rule sets to simplify the rules' complexity.

The main feature of PANFIS is the construction of ellipsoids in arbitrary positions to the support the multidimensional membership function in the feature space. These ellipsoids in arbitrary positions are projected to fuzzy sets to form the antecedent parts of fuzzy sets which are easy for the user to interpret. The inference scheme of PANFIS still uses high-dimensional ellipsoidal representation.

The evolution of rules is controlled by the datum significance (DS) criterion which represents the potential of the current instance being learned in the system. DS was initially proposed by [43] and [50] to identify the high-potential of the instance which can be measured by the statistical contribution of the instance to PANFIS' output. Once its value is high, this instance is considered to have high descriptive power and generalization potential and is worth being hired as a new rule.

The rule adaptation policy is executed when the arrival instance falls in the current clusters. In this case, the winning rule parameters are adjusted to determine the new coverage/span of the winning rule. Rule adaption in the original PANFIS utilizes an evolving self-organizing map

(ESOM) [91]. However, this method has the drawback of instability which requires reinversion once the inverse covariance matrix is ill-conditioned (e.g., due to redundant input features). As a result, the adaptation formula of GENEFIS [12], pClass [84], and GEN-SMART-EFS [53] is adopted instead.

The three properties of the winning rule are updated as follows:

$$C_{win}(update) = \frac{Sup_{win}(prev)}{Sup_{win}(prev)+1} + \frac{X - C_{win}(prev)}{Sup_{win}(prev)+1} \tag{4.1}$$

$$\begin{aligned}\sum_{win}(update)^{-1} &= \frac{\sum_{win}(prev)^{-1}}{1-\alpha} + \frac{\alpha}{1-\alpha} \\ &= \frac{\sum_{win}(prev)^{-1}(X - C_{win}(update)))(\sum_{win}(prev)^{-1}(X - C_{win}(update)))^T}{1+\alpha(X - C_{win}(prev))\sum_{win}(prev)^{-1}(X - C_{win}(prev))^T}\end{aligned} \tag{4.2}$$

$$Sup_{win}(update) = Sup_{win}(prev) + 1, \tag{4.3}$$

where $C_{win}(update)$, $\sum_{win}(update)^{-1}$, and $Sup_{win}(update)$ denote the updated focal point, dispersion matrix, and the population of the winning rule respectively prior to instance arrival. These rule property updates also show that the winning rule (part of the rule base system) is evolving based on the value of the current instance.

The rule pruning of PANFIS is driven by an extended rule significance (ERS) concept which represents the contribution of every rule to the system output. ERS is inspired by the concept of the SAFIS method [50] by integrating hyperplane consequents and generalizing to ellipsoids in an arbitrary position which enable rules to be pruned in the high-dimensional learning space.

The fuzzy sets merging in PANFIS is carried out when some fuzzy sets are overlapped, which means they have a similar membership function. This is done to reduce fuzzy rule redundancies,

thus forming a more interpretable rule base. The similarity calculation between two fuzzy sets can be found in [92]. The two fuzzy sets can be merged if the similarity $S_{ker} \geq 0.8$.

The fuzzy consequences adjustment of PANFIS is driven by the enhanced recursive least squares (ERLS), which is inspired by conventional recursive least squares (RLS) [93]. The main function of ERLS is to support the convergence of the system error, which is used for weight vector updates.

## 4.3 Scalable PANFIS Framework - Architecture and Problem Formulation

In this section, we discuss the architecture and the problem formulation of Scalable PANFIS. The first part (subsection 4.3.1) discusses the Scalable PANFIS framework architecture which focuses on the interaction between components in Apache Spark (Spark), a distributed processing platform. The second part (subsection 4.3.2) discusses the problem formulation of Scalable PANFIS.

### 4.3.1 Scalable PANFIS Framework Architecture

The Spark platform [23] is the latest platform for distributed-based data processing. In comparison with the older platforms, such as Hadoop, Spark improves performance significantly in terms of the speed of data processing because it supports an in-memory based instead of disk-based programming model. The Spark ecosystem comprises two parts: 1) spark-core ; and 2) programming interface core. Spark-core lies in the lower level library of the Spark ecosystem to serve the programming interface core. The programming interface core is integrated by Spark APIs which support many programming languages such as Scala, Java, Python, and R. Furthermore, Spark API also provides a machine learning library (Spark MLib), GraphX for analysis, a stream processing module of Spark Streaming, and SQL for structured data process-

ing. For large-scale data analytics, these Spark ecosystem components enable the framework to conduct parallel data processing and support the real-time insight/knowledge generation of large-scale data streams.

The R language is chosen as the main operating programming language in the Scalable PANFIS framework as it is a well-known programming language which is commonly used for data analysis. In order to bridge the operation between R and Spark, the SparkR library (as a backend R and Java Virtual Machine (JVM)) is utilized to manipulate and process large-scale data in a parallel/distributed manner. The type of data used in processing large-scale data in parallel mode is Spark DataFrames, a unique Spark data abstraction which is stored in memory cluster computing.

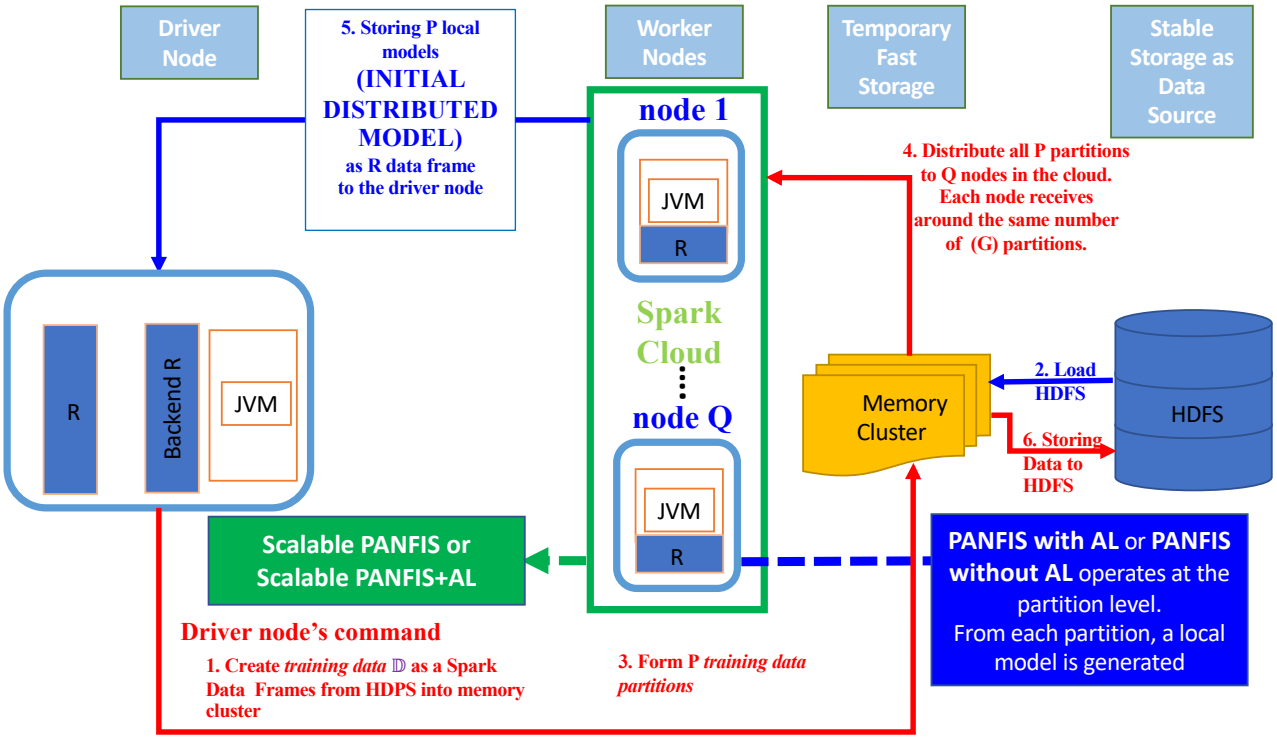


Figure 4.1: The Data flow architecture of the Scalable PANFIS framework during the data stream training phase in the Spark platform

Fig. 4.1 shows the data flow architecture of the Scalable PANFIS framework in the distributed training phase utilizing the Spark platform which focuses on the interaction between Spark's components (driver node, worker nodes, memory cluster, and HDFS). This is different from the distributed learning diagram portrayed in Fig. 2.7, which emphasizes the visualization

distributed learning process in detail which involves the distributed training of all training data partition $\mathbb{D}^p$, initial distributed model generation $\mathcal{F}^{initials}$, merged model $\mathcal{F}^{merged}$, and inference procedure $\hat{Y} = \mathcal{F}^{merged}(\hat{X})$. Note that training data $\mathbb{D}$ and its partition $\mathbb{D}^p$ used in the distributed algorithm portrayed in Fig. 2.7 is in the form of static data.

Furthermore, PANFIS is chosen as a base-learning algorithm because it is of leading fuzzy-based evolving system. While nowadays, deep learning (deep neural network) algorithm has arised as a state of the art approach, they are considered as a black box approach as the relation of input and output cannot be mapped easily. Each rule in PANFIS represents the local model of the system. Thus, the merging process can be done with ease.

### 4.3.2 Problem Formulation of Scalable PANFIS

Suppose that the data stream is formulated as $\mathcal{S} = \{\mathbb{B}_1, \mathbb{B}_2, \mathbb{B}_3, .., \mathbb{B}_k, ...\}$ demonstrating the batch-by-batch arrival of data on a large scale. $\mathbb{B}_k$ denotes the batch of data at $k^{th}$ timestamp, $\mathbb{B}_k \in \Re^{N\times(n+m)}$. Scalable PANFIS is a large-scale data stream analytics framework which learns the first batch of training data $\mathbb{B}_1$. The Scalable PANFIS framework, in this case, has not been evaluated in an incremental fashion. In this chapter, the proposed method is used to develop a large-scale data stream model, the **evolving distributed model**, since every batch partition was learned by the PANFIS evolving algorithm generating an evolving model. The important challenge here is how to combine or merge the local models obtained from the batch partition learning without decreasing the accuracy in the testing task. Note that in this chapter, Scalable PANFIS is used to solve the classification problem. Therefore, in Scalable PANFIS, the model can be seen as the classifier.

Learning from data streams can be perceived as generating an evolving model sequentially in every timestamp. In Scalable PANFIS, suppose that $\mathbb{B} \in \Re^{N\times(n+m)}$ denotes the batch of the training data which has $N$ instances (in large-scale). In this chapter, we call $\mathbb{B}$ as the training data since Scalable PANFIS only evaluates the first batch of training data. Based on Fig.

4.1 (interaction between Spark's components) and Fig. 2.7 (the architecture of distributed algorithms), we draw a problem formulation in the Scalable PANFIS framework and the steps are summarized as follows:

- Load an R DataFrames consisting of training data $\mathbb{B}$ from HDFS and convert it into a Spark DataFrames of training data $\mathbb{B}$ in the memory cluster. $N$ represents the number of instances in the training data, and $n$ and $m$ are the dimensional input and output vector of the training data, respectively.

- The driver node sends the third instruction which divides DataFrames in the memory cluster into $P$ number of partitions. All the partitions are denoted as $\mathbb{B}^p \in \Re^{V \times (n+m)}$, where $p = \{1, ..., p, ..., P\}$ and $V$ is the number of instances in a partition $\mathbb{B}^p$.

- Send all $P$ training data partitions $\mathbb{B}^p$ into $\mathcal{Q}$ nodes, so that each node receives around an equal number of $G$ partitions to process. The partition distribution mechanism is explained in section 2.6.3).

- Each node, on which PANFIS and AL are embedded, processes/trains $G$ number of training data partitions in order (serial mode). This process is a local training, resulting in $G$ number of local models generated in each node (see the solid blue box with the dashed arrow where PANFIS and AL algorithms embed every single node).

- This part marks the training process of Scalable PANFIS overall. Once all nodes have processed all $G$ partitions coming into them, these results are collected and stored in a driver node. The collection of the results is an R DataFrames consisting of $P$ local models as the initial distributed models denoted as $\mathcal{F}^{initials} = \{\mathcal{F}^1, ..., \mathcal{F}^p, ..., \mathcal{F}^P\}$.

- The last step is an optional step depending on whether the models should be saved in the stable storage for back up purposes or directly used for the next process (and majority voting).

## 4.4 Structure of the Scalable PANFIS framework model

This subsection details the four structures (learning mechanism) the of Scalable PANFIS framework due to the combination of distributed training and the testing mechanism in the framework. The distributed training mechanism of Scalable PANFIS has two approaches in terms of how a local model is generated:

- Using **PANFIS** as an evolving *base-learning algorithm.*
- Using **PANFIS with AL** as an evolving *base-learning algorithm*

Furthermore, the testing/inference mechanism of Scalable PANFIS can be conducted using two approaches:

- Using a merged model $\mathcal{F}^{merged} = merge(\mathcal{F}^{initials})$ generated from merging the initial distributed models. $\mathcal{F}^{merged}$ can be used to infer the *testing data* $\hat{X}$ using $\mathcal{F}^{merged}$ similar to the testing task in the distributed algorithms (Fig. 2.7). The output of this inference can be denoted as $\hat{Y} = \mathcal{F}^{merged}(\hat{X})$. Note that model merging is executed at the rule level as the smallest component of a model.
- Using the initial distributed models $\mathcal{F}^{initials} = \{\mathcal{F}1, ..., \mathcal{F}i, ..., \mathcal{F}I\}$ as a result of training data stream partitions. In the case of distributed learning, $I = P$, where $P$ denotes the number of partitions in a distributed training task. The output is determined from the combination output from all base models using the **majority voting** method. The majority voting method can be seen as the testing task in the ensemble learning framework illustrated in Fig. 2.6, particularly in the testing part, but the voting weight ($\beta_i$, $i$ is the index of the model in $\mathcal{F}^{initials}$, $i = \{1, ..., i, ..., I\}$, and $I$ represents the number of models in $\mathcal{F}^{initials}$) for all the base models in the majority voting are set to be equal.

Taking two combinations for each distributed training and testing mechanism in Scalable PANFIS, in this thesis, we present the four structures of the Scalable PANFIS framework: (1)

Scalable **PANFIS** using the **model merging** method; (2) Scalable **PANFIS** using the **majority voting** method; (3) Scalable **PANFIS with AL** using the **model merging** method; (4) Scalable **PANFIS with AL** using the **majority voting** method. Note that PANFIS can learn a set of training data and form a model with or without the AL method.

### 4.4.1 Scalable PANFIS Framework using the Model Merging Method

This subsection discusses the first structure of Scalable PANFIS which utilizes the **PANFIS** evolving algorithm in the distributed training task to generate initial distributed models. The **model merging** methodology is discussed for the testing task.

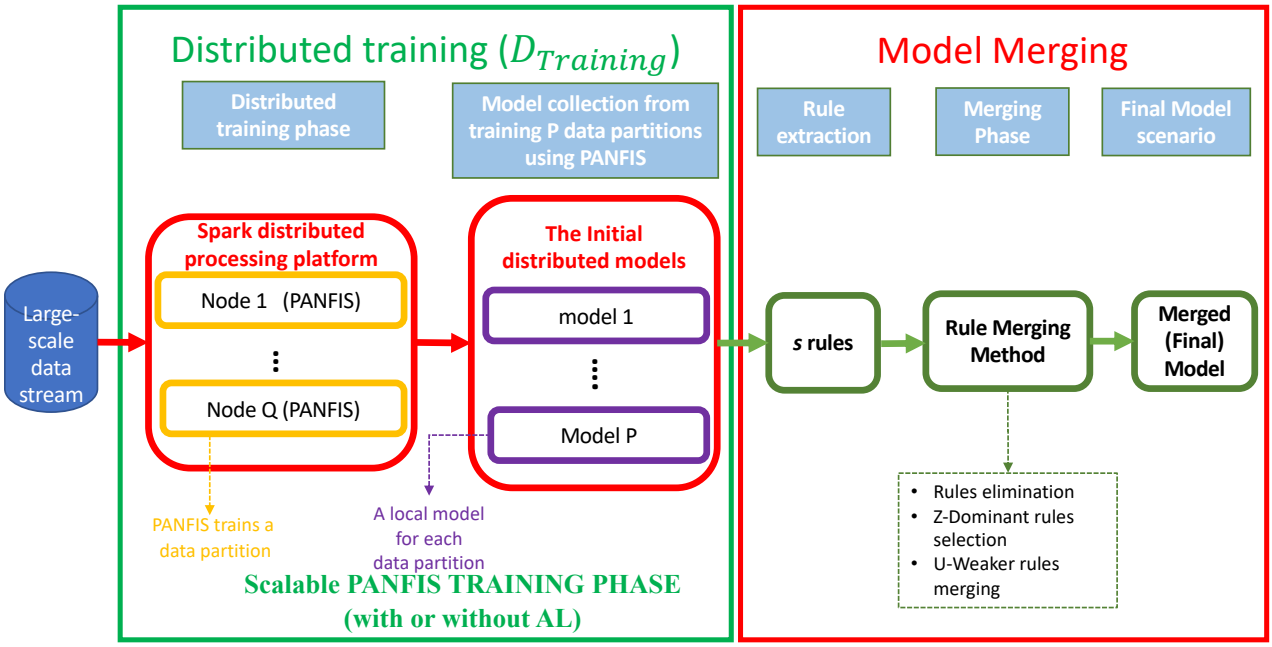


Figure 4.2: The structure of the Scalable PANFIS framework using the model merging method at the rule level.

#### 4.4.1.1 The Initial Distributed Models and Their Components

##### 4.4.1.1.1 The Underlying Reason for Using a Rule as a Merging Component

As can be seen in Fig. 4.2 the initial distributed models are the direct representation of large-scale training data streams which consist of $P$ local models resulting from training all the

*training data partitions* $\mathbb{B}^p$, $p = \{1, ..., p, ..., P\}$. It is worth noting that the merging of the initial distributed models should consider merging at the *rule* level instead of the *local model* level. This is because a local model in PANFIS is a multiple fuzzy if-then rules, and each rule contributes to the overall inference output. Since there are $P$ local models inside the initial distributed models, all rules as the smallest component of the initial distributed models are concatenated in order to form the overall output of the fuzzy inference system. Assuming that $s$ denotes the number of concatenated rules from the initial distributed models (each local model may contain one or more rules), the number of concatenated rules is possibly higher than the number of local models in the initial distributed models ($s \geq P$). In this thesis, the model merging procedure is conducted at the rule level. The model merging (at the rule level) method has previously been applied in other distributed PANFIS frameworks in [94] and [95].

#### 4.4.1.1.2 The Need to Select and Remove Inconsequential Concatenated Rules Prior to Model Merging

From the previous discussion, it is clear that initial distributed models contain $P$ local models or $s$ concatenated rules extracted from the initial distributed models. In the testing task, using a merged model from the extracted rules directly for inference is impractical. The extracted rules contain many rules which are overlapping each other and this condition deteriorates the generalization power of the system's model.

We investigate whether the decrease in accuracy is caused by the difference in the data distribution of all data partitions, thus leading to misclassification output when the data is close to the rules which have a lower generalization power. Furthermore, during data partition training, outliers may appear and generate new rules with a very small population. Thus, these rules cannot represent the distribution of the data partition. To validate our hypothesis, we conduct an experimental test on the HEPMASS dataset to show that the initial **rule selection** (selecting $Z$ best rules) and **rule removal** (eliminating rules which have a small population) prior to the model merging influence the model's performance. The results of this experiment are

illustrated in Table 4.1.

Table 4.1: The accuracy of the HEPMASS testing dataset for different $Z$ best initial rule selection with and without rule removal prior to model merging

| k | Accuracy(rule removal) | Accuracy(without rule removal) |
|---|---|---|
| 1 | 83.87 | 83.57 |
| 3 | 83.47 | 83.68 |
| 5 | 83.15 | 83.37 |
| 45 | 83.63 | 83.46 |
| 50 | 83.52 | 82.82 |
| 55 | 77.28 | 62.16 |
| 60 | 71.11 | 61.64 |

It can be seen from the empirical result in Table 4.1 that not all rules in the classification system provide the same classification output. It is clear that at $Z = 55$, the performance of the HEPMASS testing dataset decreases with a higher rate of without rule removal. In the case of rule removal, there are no outliers involved, so the performance decrease is purely due to the existence of rules which have a low generalization power. For the case of without rule removal, the number of rules which have a low generalization power is higher than the model with rule removal, thus the result is worse. This indicates that some outliers are wrongly chosen because they have a higher ranking than other rules.

From the above investigation, it is clear that rule removal and rule selection are the steps that need to be carried out prior to model merging. Rule selection is inspired by the work in [20], where they select the rules with the highest weight which are among the same antecedent in the same partition. The same procedure is repeated where the rules selected from every partition are compared with the rules selected from the other partitions. Rules which have the highest weight among the same antecedent are selected. At the end of the process, only the clusters/rules which have the highest weight for all unique antecedents will exist and can be used as the component in generating the final merged model.

Furthermore, apart from rule removal, we propose rule selection by first selecting the $Z$ best rules. This is done by observing the highest classification training accuracy among local models when data partitions were trained using the Scalable PANFIS framework. The training accuracy reflects the confidence level of the local model to be recruited as the preferred local model. Since

each local model is constructed by one or more fuzzy rules, the weight of the rules are assigned by the weight of their corresponding local models.

#### 4.4.1.2 Model Merging Implementation at the Rule Level

From the previous explanation, we can summarize that preliminary steps need to be carried out prior to model merging. As the first step, the rule extraction is carried out by simply concatenating all rules in the system, resulting in $s$ rules being concatenated from $P$ local models, $s \geq P$. The next step is assigning the rule weight which can be acquired from the training accuracy of the local model corresponding to it. Rule removal in step three aims to remove the outliers. The outliers can be identified as they have a lower population than the others, thus they contribute less during their lifespan. We apply a 5 percent threshold population of the rule in the total population of the cluster. If it does not meet the requirement of the threshold, the rules are removed from the system. From this step, $o$ number of rules are extracted, where $P \leq o \leq s$. Up to this step, we have $o$ rule candidates to be fed in the **rule merging** process (rule merging is part of model merging). The implementation of **rule selection** is initialized by choosing the most $Z$ influential rules among the other $o$ rule candidates. We call the $Z$-most influential rules the **Dominant** rules, whereas the $U$ number of rules are called the **Weaker** rules thus, $o = U + Z$. If the set of **Dominant** rules is denoted by $D_z = \{D_1, .., D_z, ..., D_Z\}$, and the **Weaker** rules is denoted by $W_u = \{W_1, .., W_u, ..., W_U\}$. The merging process between rules occurs by following the procedure illustrated in algorithm 3.

The first $Z$-**Dominant** rules are selected, and they become the reference of the other $U$ number of **Weaker** rules, assuming that the **Dominant** rules have more optimum results than the **Weaker** ones. From algorithm 3, it can be seen that the first loop process aims to assign each of the **Weaker** rules to the closest **Dominant** rule as well as discards the rule if the maximum similarity obtained is less than or equal to $\theta$. The value of $\theta$ is set to 0.9 in this experiment. This value is set with the assumption that the higher the level of the threshold, the more

similar rules will be merged to avoid a decrease in classification performance. In other words, the reason of using the high value of $\theta = 0.9$ is to maintain the high quality of rules/clusters having the high generalization power. As it can be seen in Table 4.1, that the more rules to be included in the merging process, the higher chance the accuracy will decrease. At the end of the rule similarity calculation, only the shortlisted number of rules ($var$) are selected, $var \leq U$, to be merged to $Z$-**Dominant** rules. The rest of **Weaker** ($U - var$) rules are discarded.

The similarity calculation between two rules is adopted from the method proposed in [53]. This is based on the degree of deviation in the hyper-planes's gradient information, where the deviation is calculated based on the dihedral angle of the two hyper-planes they span formulated as follows:

$$\phi = arccos(\frac{a^T b}{|a||b|}), \tag{4.4}$$

where $a = (D_{z;1}D_{z;2}D_{z;3} - 1)^T$ and $b = (W_{u;1}W_{u;2}W_{u;3} + 1)^T$ the normal vectors of the two planes corresponding to rules $D_z$ and $W_u$, showing in the opposite direction with respect to target $\hat{y}$ (-1 and +1 in the last coordinate). Thus, the similarity of two hyper-planes is formulated as follows:

$$Sim_{(D_z,W_u)} = \frac{\phi}{\pi}, \tag{4.5}$$

Two hyper-planes are regarded as similar if the similarity degree $Sim_{(D_z,W_u)} \geq \theta$.

The next loop in Algorithm 3 performs the merging of $var$ assigned rules to the associated **Dominant** rules. The parameter update of the dominant rules is carried out iteratively, where **Dominant** rules can only be updated with the list of the **Weaker** rules assigned to them.

Note that PANFIS *base-learning algorithm* adopts TSK fuzzy system, whose rules/clusters describe the input-output relations as depicted in formula 4.11. PANFIS rule premise part is built upon ellipsoids shape in arbitrary position as the fuzzy sets. The consequent inference part itself is built upon polynomial function as depicted in formula 4.12 as the hyperplanes.

While some rules can be directly merged using a simple Euclidean distance criterion from the premise parameters (e.g. centres of the clusters) [51] or using membership function criterion as

**Algorithm 3:** Model merging algorithm

**Input** : Set of Dominant and Weaker Rules (D and W) after rule selection and rule removal are applied from the extracted rules.
**Ouput** : Set of Updated parameter of Dominant Rules
*Initialization*:
$Z$ : number of **Dominant** rules
$U$ : number of **Weaker** rules
$D_z$ : index $z$ of **Dominant** rules
$W_u$ : index $u$ of **Weaker** rules
$list_{assign} = [\ ]$ ; Array of empty set to store the index of recruited weaker rules
*Loop Process*:
-Assign Weaker Rules to the closest Dominant Rules
**for** $u = 1$ to $U$ **do**
  **for** $z = 1$ to $Z$ **do**
    Count **Similarity** between $W_u$ and $D_z$ ($Sim_{D_z,W_u}$) (**(Formula 4.5)**)
  **end for**
  Determine winning rule: calculating maximum similarity
  $D_z(winning) = \arg\max_{z=1,..,Z}(Sim_{D_z,W_u})$
  **if** ($Sim_{D_z,W_u}(winning) \geq \theta$ ) **(Formula 4.5) then**
    Rule $W_u$ is recruited to be merged with the rule $D_z$
    $list_{assign} = [list_{assign}\ u]$
  **else**
    Discard $W_u$ ($W_u$ *is regarded has a low similarity over current* ***Dominant*** *rules*)
  **end if**
**end for**
Some rules may be eliminated, resulted in $var = sizeof(list_{assign})$ assigned rules to be used for the next merging process ($var \leq U$).
*Loop Process*:
**Rule merging:** merging of assigned Weaker rules to the closest Dominant Rules
**for** $z = 1$ to $Z$ **do**
  **for** $u = 1$ to $U$ **do**
    - Iteratively update Dominant rule parameters by merging it with the assigned list of Weaker rules
    **if** ($u$ not in $list_{assign}$) **then**
      skip for the next $D_z$
    **else**
      **if** ( the condition of blow-up effect is met- **Formula (4.6)** ) **then**
        $D_z(Update)$=Merging of rule $D_z(current)$ and rule $W_u$ **(Formula 4.7 4.8 4.9 4.10)**
        $D_z(Current) = D_z(Update)$
      **else**
        Cancel merging, discard rule $W_u$
      **end if**
    **end if**
  **end for**
**end for**

initially carried out in PANFIS, we adopt a geometric-based criteria as carried out in [53]. In particular, our approach adopts the calculation of hyperplanes' gradient information to measure the direction of the rules.

While we only use a gradient information in the merging process [53], we calculate the blowup effect to ensure that both merged rules should form a homogeneous shape (ellipsoids) and direction (gradient). This measurement represents the accurate representation of the two rules/-clusters when merged to ensure a homogeneous joint region. The blowup effect is formulated as:

$$Vol_{merged} \leqslant n(Vol_{Dominant_z} + Vol_{Weaker_u}), \tag{4.6}$$

where $Vol$ stands for volume and $n$ is the dimension of input attribute. After these two conditions are fulfilled (formula 4.5 and formula 4.6), the updated parameters of the merged rule referring to the work in [12], is as follows:

$$c_{Dom_z}(update) = \frac{c_{Dom_z}(cur)Sup_{Dom_z}(cur)+c_{Weak_u}(cur)Sup_{Weak_u}(cur)}{Sup_{Dom_z}(cur)+Sup_{Weak_u}(cur)}, \tag{4.7}$$

$$\sum\nolimits_{Dom_z}^{-1}(update) = \frac{\sum_{Dom_z}^{-1}(cur)*Sup_{Dom_z}(cur)+\sum_{Weak_u}^{-1}(cur)*Sup_{Weak_u}(cur)}{Sup_{Dom_z}(cur)+Sup_{Weak_u}(cur)}, \tag{4.8}$$

$$Sup_{Dom_z}(update) = Sup_{Dom_z}(cur) + Sup_{Weak_u}(cur)\ , \tag{4.9}$$

$$w_{Dom_z}(update) = \frac{w_{Dom_z}(cur)*Sup_{Dom_z}(cur)+w_{Weak_u}(cur)*Sup_{Weak_u}(cur)}{Sup_{Dom_z}(cur)+Sup_{Weak_u}(cur)}\ , \tag{4.10}$$

where $c_{Dom_z}(update)$ and $\sum_{Dom_z}^{-1}(update)$ are the updated antecedent parameters of the merged rule and $w_{Dom_z}(update)$ is the updated consequent parameter of the merged rule.

### 4.4.2 Scalable PANFIS Framework using the Majority Voting method

This subsection discusses the second structure of Scalable PANFIS. The distributed training task of this structure utilizes **PANFIS**, the same base-learning algorithm as the one used in the

first structure, whereas on the testing task, a **majority voting** method is used to aggregate the output. Unlike the first structure, where initial distributed models are merged first prior to the testing task, in this structure, the initial distributed models are used directly as the multiple-classifiers to form the output. The majority voting inference scheme can be seen as an ensemble algorithm inference scheme where its output is drawn from the composite of multiple output using the same voting weight for all local models. That is, every local model infers the same instance. The composite output of the instance is determined from the most voted class among the local models. The second Scalable PANFIS structure is depicted in Fig. 4.3.

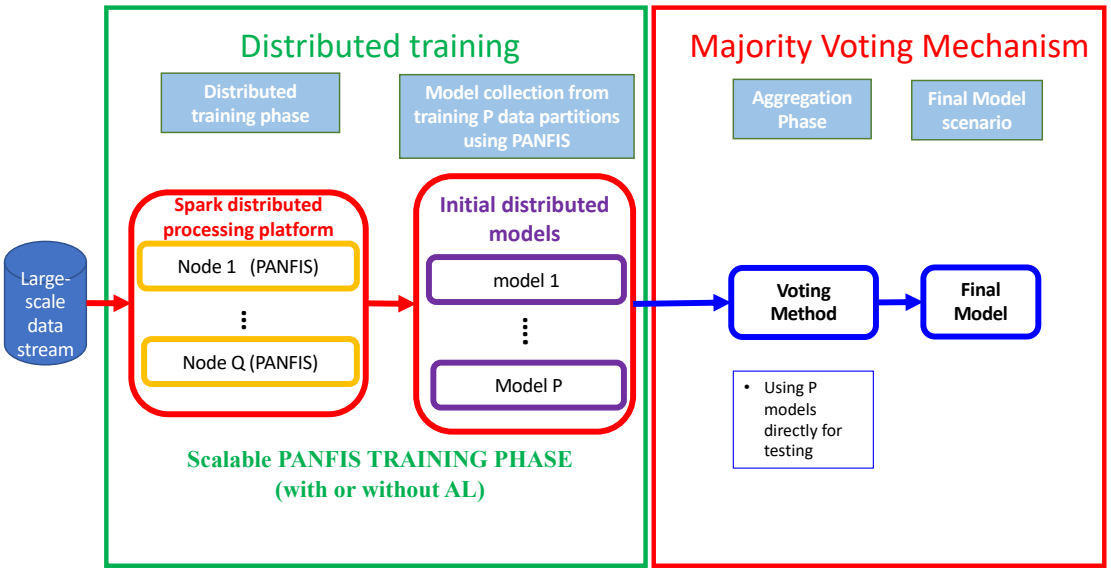


Figure 4.3: The structure of the Scalable PANFIS framework using the majority voting method

Voting methods have become popular to combine multiple models, such as those in the early works in [77, 96, 97]. A voting method in the fuzzy rule-based classification system was pioneered by Ishibuchi et al. in [98]. In the realm of fuzzy rule-based classification systems, there are two kinds of fuzzy rule-based voting schemes: 1) multiple fuzzy if-then rules in a single fuzzy rule-based classification system; and 2) multiple fuzzy rule-based classification systems. For majority voting, we adopt the second method where the voting procedure is carried out using multiple fuzzy rule-based classification systems. In this case, the voting procedure is conducted at the model level instead of the rule level.

In PANFIS, the $n$-dimensional pattern classification problem is defined by the fuzzy rule-based

classification system. The classification system makes use of a generalized form of the Takagi Sugeno Kang (TSK) fuzzy system. Its antecedent part is developed by the ellipsoids in arbitrary positions connected with a new projection concept in terms of linguistic terms (fuzzy sets). The ellipsoids are generated by the multivariate Gaussian function. The generalized fuzzy rule is formally written as follows:

$$R_r : If\ x_1\ is\ close\ to\ A_1^1\ and\ ...\ and\ x_n\ is\ close\ to\ A_r^n$$
$$THEN \hat{y_r} = w_r \phi_r \tag{4.11}$$

where $w_r$ and $\phi_r$ are the particular output parameter and the firing strength of particular $r^{th}$ rule respectively which reflect the consequent part of the fuzzy rule. The output $\hat{y}$ in the fuzzy system is determined by the following formula:

$$\hat{y} = \sum_{r=1}^{v} \phi_r w_r, \tag{4.12}$$

where $v$ denotes the number of rules in the system. PANFIS was originally designed for the regression problem. In the case of classification, the output of a given instance of testing data is structured following the multi-input-multi-output (MIMO) form. The output parameter $w_r$ of $r^{th}$ rule in the MIMO form is expressed as follows:

$$w_r = \begin{bmatrix} w_{r0}^1, w_{r0}^2, ..., w_{r0}^m \\ w_{r1}^1, w_{r1}^2, ..., w_{r1}^m \\ ..... \\ w_{rn}^1, w_{rn}^2, ..., w_{rn}^m \end{bmatrix}, \tag{4.13}$$

where $m$ and $n$ denote the number of classes and input dimension, respectively. Thus, the multiplication of output parameter $w_r \in \Re^{(n+1)\times m}$ and the firing strength $\phi_r \in \Re^{1\times((n+1))}$ will result in the output value $\hat{y} \in \Re^{1\times m}$. The classification decision of a particular instance is

determined by observing the highest activation degree of output over all rules which is expressed as follows:

$$O = \arg \max_{c=1,..,m} (O_c) \tag{4.14}$$

Note that for PANFIS, the $n$-dimensional classification model is conducted in a particular node of the large-scale data analytics framework which processes particular $G$ partitions of data in a serial manner, while parallel across nodes. In the case of distributed learning, there are many partitions to be processed/learnt in large-scale data analytics training phase. Therefore, a $P$ number ($P$ is set by user) of local models is generated in the distributed training phase. In the case of the majority voting procedure for the final classification decision, all models generated in the large-scale data training phase influence the classification decision of every instance in the testing data. An illustration of the models generated in the training phase in inferencing an instance from the testing dataset is depicted in Fig. 4.4.

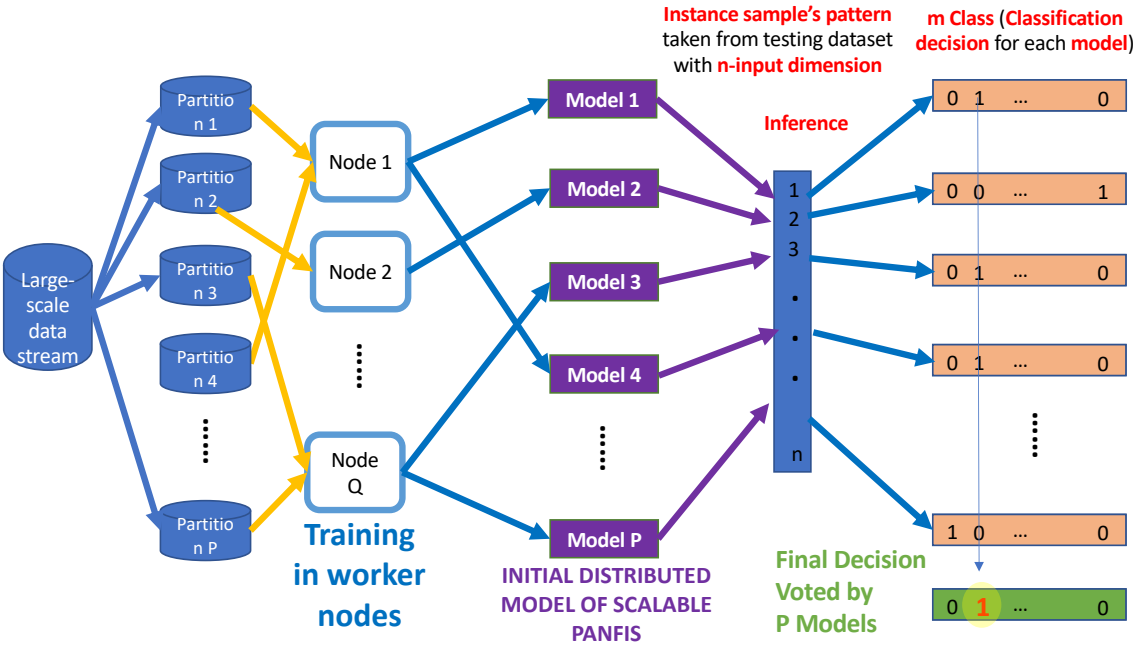


Figure 4.4: The voting mechanism scheme in the distributed machine learning PANFIS architecture

The majority voting mechanism is applied in the Scalable PANFIS framework because it adopts the multiple model classification technique as in the ensemble algorithm where the weight of all local models is set to be equal. Classification systems are usually formed by the combination

of either many uniform or different types of local models, which often show better performance than a single local model [99].

### 4.4.3 Scalable PANFIS Framework with AL and the Model Merging Method

This subsection mainly discusses the third structure of Scalable PANFIS, where **active learning (AL)** is involved in the distributed training task. The learning mechanism in this structure is similar to the one described in subsection 4.4.1. However, this Scalable PANFIS framework structure uses AL to train large-scale data streams. In other words, the number of samples trained to generate a local model from a *training data partition* is less than the number of samples in the original one. Furthermore, the procedure of generating the initial distributed model generation remains the same. That is, every worker node sends their results ($G$ local models) to be accumulated with the results from the other nodes into a driver node. A set of local models is collected in the driver node as an initial distributed model. In this structure, initial distributed models are then merged using the model merging method. This structure is illustrated in Fig. 4.2.

The key feature in large-scale data stream analytics is the ability of the framework to train the data efficiently in terms of running time and accuracy. In addition to distributed data stream processing, sample selection (also known as prototype reduction) is used to reduce computational learning time. Sample selection in this work is inspired by the certainty-based active learning (AL) concept. The main difference in prototype reduction is that AL evaluates the data in an unsupervised mode.This concept is able to define the relevant instances and minimize the labour-intensive labelling effort.

The AL was inspired by the what-to-learn method of the traditional meta-cognitive model in [100][101], which still completely depends on labeled training samples. The online AL was put forward in [102] to improve the previous concept accommodating the online learning scenario. However, none of these concepts involve concept drift in the data stream. Therefore, in order

to cope with this issue, a certainty-based active learning scenario is proposed in [103].

The certainty-based AL scenario is developed by virtue of the Bayesian concept, where the Bayesian posterior probability determines the conflict level between input and output spaces. This Bayesian concept is more preferable than the firing strength criterion because the Bayesian concept is more robust to outliers. In addition, the variable of the uncertainty strategy [104] counterbalances the effects of concept drift. This strategy adjusts the conflict threshold correspondingly to an up-to-date system dynamic. The substantial conflict in the output spaces is triggered by the instance occupying an adjacent proximity to the decision boundary. The model's truncated output defines the conflict in the output which is expressed as follows:

$$\begin{aligned} p(\hat{y}_s|X)^{output} &= min(max(conf_{final}, 0), 1), conf_{final} \\ &= \frac{\hat{y}_1}{\hat{y}_1 + \hat{y}_2}, \end{aligned} \tag{4.15}$$

where $p(\hat{y}_s|X)^{output}$ represents the output posterior probability. The first and second dominant outputs are denoted as $\hat{y}_1$ and $\hat{y}_2$, respectively. The conflict in the output space is determined by the quality of the decision boundary, whereas the conflict of the input space is caused by the unclean cluster, where the cluster has a different class sample. We calculate the posterior probability using the joint-category and class probability $p(\hat{y}_s|R_i)$ estimation as follows:

$$p(\hat{y}_s|X) = \frac{\sum_{i=1}^{r} p(\hat{y}_s|R_i)p(X|R_i)p(R_i)}{\sum_{o=1}^{m}\sum_{i=1}^{r} p(\hat{y}_s|X)p(X|R_i)p(R_i)}, \tag{4.16}$$

$$p(\hat{y}_s|R_i) = \frac{log(Sup_i^s + 1)}{\sum_{o=1}^{m} log(Sup_i^s + 1)}, \tag{4.17}$$

where $Sup_i^s$ represents the number of the $i^{th}$ cluster falling to the $s^{th}$ class, $m$ denotes the number of output dimensions or classes, and $p(\hat{y}_s|X)^{input}$ represents the input posterior probability. If the model exhibits a strong confusion, the training samples are allowed to update the model.

This criterion is defined as follows:

$$p(\hat{y}_s|X)^{output} < \upsilon \; and \; p(\hat{y}_s|X)^{input} < \upsilon, \tag{4.18}$$

where $\upsilon$ represents the conflict threshold. The variable budget $Bud$ is introduced to determine the maximum number of samples allowed to be annotated during the training process. With the assumption that data is uniformly distributed, $\upsilon$ is initialized as $\upsilon = \frac{1}{m} + Bud(1 - \frac{1}{m})$. When the conflict in the output space or the conflict in the input space is higher than the conflict threshold, the sample does not need to be trained. Otherwise, the sample needs to be trained. The value of $\upsilon$ is dynamically changing, increasing when the sample is trained, and decreasing whenever the sample is discarded from the training process.

### 4.4.4 Scalable PANFIS Framework with AL and the Majority Voting Method

The fourth structure of Scalable PANFIS employs **PANFIS with AL** for the distributed training task while **majority voting** is used for the testing task. In the distributed training task, initial distributed models are obtained from the collection of local models, where each local model is the representation of the corresponding *training data partition* trained by PANFIS with AL. The inference procedure follows the majority voting conducted in subsection 4.4.2.

The initial distributed models contain $P$ number of models (e.g. PANFIS with AL). The final model is obtained by processing the initial distributed models using majority voting, as explained in subsection 4.4.2. This final model is then used to classify large-scale data stream testing. The fourth structure of the Scalable PANFIS framework (using AL) sequence from the initial distributed models until the final model is generated is illustrated in Fig. 4.3.

## 4.5 Numerical Study

This section describes the numerical study of large-scale data stream analytics using Scalable PANFIS and the Spark-based algorithm. Subsection 4.5.1 details the experiment procedure including the environmental setup, performance measures, datasets, and methods used in the experiment. The results of the experiment are discussed in subsection 4.5.2.

### 4.5.1 Experiment Setup

In this work, all the experiments are performed in the Spark platform, under the Nimbus Pawsey Supercomputing Centre Australia. The Spark platform is built by one master node and 6 worker nodes, where the Pawsey version of Ubuntu 20.04 Focal Fossa is installed for all nodes as the operating system. Each node has the maximum specification 32GB RAM. For the total memory used in the cluster, we configure only 24GB for each worker node to be allocated in the memory cluster, leaving the rest for other operations, thus the total memory cluster is 144GB. For the driver node, we allocate 8GB for the Spark operation, leaving the rest (24GB) for other operations, considering there may be a lot of variables stored in the local memory of the driver node. For software, we use Apache Spark version 3.0.0 and R version 3.6.3.

In this experiment, eight algorithms are compared in order to measure their performance in terms of running time, accuracy, and the number of rules generated after the merging. The first four algorithms are the Scalable PANFIS algorithms and the other algorithms are the algorithms which are built in the SparkR API library. For the sake of simplicity, we abbreviate the four structures of the Scalable PANFIS algorithms which employ a combination of three techniques (e.g, the active learning, the model merging, and the majority voting) as shown in Table 4.2.

For a clearer explanation, Fig. 4.2 and Fig. 4.3 show the Scalable PANFIS model sequence using the model merging and majority voting methods as the aggregation method after initial

Table 4.2: Algorithm description

| No | **Algorithm** | **Description** |
|---|---|---|
| 1 | Scalable PANFIS Merging | Scalable PANFIS using **Model Merging** Technique |
| 2 | Scalable PANFIS Voting | Scalable PANFIS using **Majority Voting** Technique |
| 3 | Scalable PANFIS with AL Merging | Scalable PANFIS with **AL** **u**sing **Model Merging** Technique |
| 4 | Scalable PANFIS with AL Voting | Scalable PANFIS with **AL** using **Majority Voting** Technique |
| 5 | Spark.KMeans | K-Means |
| 6 | Spark.GLM | Spark Generalized Linear Model |
| 7 | Spark.GBT | Spark Gradient Boosted Tree |
| 8 | Spark.RF | Spark Random Forest |

distributed models are generated. We utilize six datasets taken from the UCI dataset repository [105] for the large-scale data analytics framework: SUSY, HIGGS, HEPMASS, Poker Hand, RLCPS, and KDDCup, their specifications being shown in Table 4.3. All datasets are divided into 80% training data and 20% testing data using 5-fold cross-validations.

Susy, Higgs, Hepmass, and RLCPS datasets are commonly used for large-scale data classification problems, such as in [106, 20]. Poker Hand and KDDCup are multiclass datasets with 10 and 22 classes, respectively. While the Poker Hand dataset is trained using all 10 classes to perform classification task based on [106], the KDDCup dataset is trained using only two classes: "normal" and "on attack" mode based on the work in [20].

Table 4.3: Dataset description

| **Dataset** | **#Samples** | **#IA** | **#Class** |
|---|---|---|---|
| SUSY | 500000 | 18 | 2 |
| HIGGS | 11000000 | 28 | 2 |
| HEPMASS | 1050000 | 28 | 2 |
| RLCPS | 5174219 | 9 | 2 |
| Poker Hand | 1025011 | 10 | 10 |
| KDDCup | 4898431 | 41 | 2 |

Samples: data points, IA: Input Attribute

There are 96 data partitions set in this work for all datasets to perform Scalable PANFIS training. Every partition is mapped into eight worker nodes to be processed on the Spark platform in parallel mode. For each data partition, one model is generated. This number of

partition (96 partitions) is chosen based on the consideration that our system has 96 cores with 8 driver nodes, so every driver node processes an equal number of partitions (12 partitions). Increasing the number of partitions will generally speed up the training process, such as the work conducted in [106]. However, our main concern in this experiment is to investigate the performance of Scalable PANFIS using model merging and Scalable PANFIS using majority voting (with and without AL) in terms of accuracy and running time. The model merging method is our main contribution which is able to merge many rules and yields stable accuracy for classification. For each dataset executed by the algorithms shown in Table 4.2, we measure the following performance:

1. Accuracy : The percentage of correctly classified testing data over all testing data.

2. Compression rate: The performance measure of the AL technique embedded in the PANFIS algorithm. This represents the percentage of instances learned over all instances in the training data in particular data partitions. It is also called compression ratio. Note that the compression ratio is only calculated by the Scalable-PANFIS algorithm with the AL method embedded on it. Otherwise, the compression rate is 100 percent, meaning that all samples in the dataset are trained.

3. Running time: The measurement of the training time required for all distributed machine learning algorithms (Table 4.2) in processing a large-scale data (Table 4.3) (from distributing the data partition into the worker nodes until the large-scale data model is generated in the driver node). The results are presented in the Table 4.5.

4. Testing time: The measurement of inference time required for Scalable PANFIS based algorithms. This measurement aims to compare the effectiveness of inference task between model merging and majority voting methods. The results are presented in Table 4.7.

5. Number of rules: The measurement of the number of rules shows the complexity of the model system. The lower the number of rules, the lower the complexity of the model system.

### 4.5.2 Results

Two group of experiments are conducted to measure the performance of the algorithms. The first group compares Scalable PANFIS without AL (Algorithm 1-2) and Scalable PANFIS with AL (Algorithm 3-4). The second group compares all the algorithms as shown in Table 4.2 (Scalable PANFIS algorithms and Spark-based algorithms). For the first group, we measure the following performance: 1)accuracy; 2)compression rate; 3)running time; 4) testing time; and 5)number of rules generated after merging. The second group measures the performance of all the algorithms (both Scalable PANFIS and Spark-based) in terms of both accuracy and running time as illustrated in Table 4.8 and Table 4.9. The hyperparameters for Scalable PANFIS are hand-tuned to yield the optimum results for all datasets and kept fix for all experiments. Three parameters are set as follows: growing threshold $k_{grow} = 1$, pruning threshold $k_{prune} = 0.25$, and safety width $kfs = 0.05$ for the PANFIS local learning.

#### 4.5.2.1 Scalable PANFIS discussion

##### 4.5.2.1.1 The effect of AL in the Scalable PANFIS performance - Scalable PANFIS with and without AL comparison

As previously discussed, the difference between Scalable PANFIS with AL and Scalable PANFIS without AL lies in the generation of the collected model of all partitions after the training phase (initial distributed models) as shown in Fig. 4.2 and Fig. 4.3. Scalable PANFIS with AL only trains selected samples in each data partition, whereas Scalable PANFIS without AL trains all samples in each data partition. Therefore, it is important to compare Scalable PANFIS with AL and Scalable PANFIS without AL.

Table 4.4 shows that the average accuracy for both Scalable PANFIS with and without AL is comparable, although Scalable PANFIS without AL performs slightly better accuracy in general with maximum difference only 0.3 per cent in Susy dataset in comparison to Scalable PANFIS with AL. However, Scalable PANFIS without AL-only trains around 40 per cent of

Table 4.4: The average of performance (compression rate and accuracy of Scalable PANFIS with and without AL using voting and merging method

| Dataset | Aggregating Method | Performance using (AL) Method | | | Performance without using (AL) Method | | |
|---|---|---|---|---|---|---|---|
| | | **Overall Average Accuracy (5-fold CV)** | **Average Merging+ Voting** | **Average Compression Rate using AL(%)** | **Overall Average Accuracy (5-fold CV)** | **Average Merging+ Voting** | **Average Compression Rate without using AL (Normal)(%)** |
| Susy | Merging<br>Voting | 76.08±0.13<br>75.38±0.13 | 75.73 | 40.12±0 | 76.46±0.18<br>75.66±0.04 | **76.06** | 100 |
| Higgs | Merging<br>Voting | 63.93±0.18<br>63.76±0.22 | 63.85 | 40.07±0 | 63.95±0.15<br>64.05±0.20 | **64** | 100 |
| Hepmass | Merging<br>Voting | 83.49±0.03<br>83.47±0.04 | 83.48 | 40.08±0 | 83.49±0.07<br>83.51±0.05 | **83.5** | 100 |
| RLCPS | Merging<br>Voting | 99.93±0.05<br>99.91±0.15 | 99.92 | 40.12±0 | 99.91±0.13<br>99.98±0 | **99.95** | 100 |
| PokerHand | Merging<br>Voting | 50.12±0.1<br>48.60±0.19 | **49.35** | 40.41±0 | 50.12±0.09<br>48.23±0.5 | 49.18 | 100 |
| KDDCup | Merging<br>Voting | 99.65±0.03<br>99.64±0.03 | 99.65 | 40.12±0 | 99.68±0<br>99.7±0.01 | **99.69** | 100 |

the samples in the worker nodes on average in comparison with Scalable PANFIS without AL which trains all the samples coming to the nodes.

From this, we can conclude that AL can operate in the training process without a significant loss of accuracy. This also means that fewer resources are required to process the samples/streams to generate the model which produces a comparable performance to Scalable PANFIS without AL. Note that average accuracy is the average of the classification result for Scalable PANFIS with AL or Scalable PANFIS without AL for both model merging and majority voting, where each of them is carried out using 5-fold cross-validations.

Table 4.5: The effect of the Active Learning Method in the distributed machine learning PANFIS training algorithm on the running time using performance 5-fold cross validations

| **Dataset** | **Average Compression Rate (%)** | **Average Running Time without AL (s)** | **Average Running Time with AL (s)** |
|---|---|---|---|
| Susy | 40.12 | 842.11±18.11 | 417.02±12.76 |
| Higgs | 40.07 | 4279.80±131.44 | 2009.97±57.32 |
| Hepmass | 40.08 | 3877.84±90.08 | 1852.66±50.94 |
| RLCPS | 40.12 | 842.11 ± 18.11 | 341.40±38.42 |
| PokerHand | 40.41 | 164.24±7.77 | 86.91±5.91 |
| KDDCup | 40.12 | 3443.68±116.89 | 1530.83±31.22 |

A similar comparison between Scalable PANFIS with AL and without AL is shown in Table

Table 4.6: Number of rules generated before and after the model merging for initial distributed models generated with Scalable PANFIS (with and without AL)

| Dataset | Scalable PANFIS Before Merging # Rule | Scalable PANFIS After Merging # Rule | Scalable PANFIS with AL Before Merging # Rule | Scalable PANFIS with AL After Merging # Rule |
|---|---|---|---|---|
| SUSY | 107 | 5 | 103 | 5 |
| HIGGS | 102 | 5 | 119 | 5 |
| HEPMASS | 135 | 5 | 137 | 5 |
| RLCPS | 96 | 5 | 96 | 5 |
| Poker Hand | 126 | 5 | 125 | 5 |
| KDDCup | 96 | 5 | 96 | 5 |

4.5 in terms of running time. It is noted that the average compression rate is linear with the speed of the training partition. For example, in the case of training SUSY dataset using 5-fold cross validations, Scalable PANFIS requires around 842 seconds to generate the large-scale data model, whereas Scalable PANFIS with AL needs around half the time, this being around 417 seconds. This trend is similar with the other datasets where the running time of Scalable PANFIS with AL is around half the running time of Scalable PANFIS without AL.

#### 4.5.2.1.2 Determining the number of initial rules before the merging process

The number of rules generated by Scalable PANFIS with and without AL (initial distributed models) and the number of rules after the merging is depicted in Table 4.6. It is clear that the proposed model merging method reduces the complexity of rules in the system from before and after merging. The initial constant number of rules after merging for all datasets is constant (5 rules) as we set the five best initial rules prior to the merging process. Note that this initial number is determined solely from empirical experiment as the higher number initial rules to be picked before merging at some point may deteriorate the generalization power due to the wrong rules being picked as the outliers as depicted in table 4.1. The dynamic selection of initial rules to be picked before merging is still beyond the scope of this chapter. In summary, merging method using a few number of initial rules is able to classify large-scale data streams.

#### 4.5.2.1.3 The Merging and Voting methods comparison

Table 4.7 shows the comparison of performance between Scalable PANFIS using merging and voting method for both accuracy and testing time. It can be seen that the accuracy of the merging method slightly outperforms the voting method on Pokerhand and Susy dataset by around 1.5 and 0.7 per cent respectively.

Table 4.7: Performance of Scalable PANFIS on merging and voting method

| Dataset | Training Method | Merging method performance | | | | Voting method performance | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | **Average accuracy** using 5-fold CV (%) | **Overall average merging** (with and without AL) (%) | **Average testing time** using 5-fold CV (s) | **Overall average testing time** on **merging** (s) | **Average accuracy** using 5-fold CV (%) | **Overall average voting** (with and without AL) (%) | **Average testing time** using 5-fold CV (s) | **Overall average testing time** on **voting** (s) |
| Susy | with AL | 76.08+0.13 | **76.27** | 24.42+1.06 | 24.07 | 75.38+0.13 | 75.52 | 2196.55+22.20 | 2179.62 |
| | without AL | 76.46+0.18 | | 23.73+0.45 | | 75.66+0.04 | | 2162.69+52.67 | |
| Higgs | with AL | 63.93+0.20 | **63.94** | 75.32+4.19 | 75.43 | 63.76+0.22 | 63.91 | 6688.43+452.53 | 6725.81 |
| | without AL | 63.95+0.15 | | 75.54+9.58 | | 64.05+0.20 | | 6762.99+310.79 | |
| Hepmass | with AL | 83.49+0.03 | 83.49 | 68.57+1.96 | 68.99 | 83.47+0.04 | 83.49 | 6117.43+212.54 | 6135.82 |
| | without AL | 83.49+0.07 | | 69.42+1.99 | | 83.51+0.05 | | 6154.21+150.82 | |
| RLCPS | with AL | 99.93+0.05 | 99.92 | 22.89+2.36 | 22.89+2.36 | 99.91+0.15 | **99.95** | 1897.69+71.91 | 1893.41 |
| | without AL | 99.91+0.13 | | 22.47±1.61 | | 99.98+0 | | 1889.13+114.13 | |
| PokerHand | with AL | 50.12+0.1 | **50.12** | 5.83+0.51 | 5.88 | 48.60+0.19 | 48.42 | 492.367+31.07 | 493.49 |
| | without AL | 50.12+0.09 | | 5.93+0.55 | | 48.23+0.5 | | 494.62+18.58 | |
| KDDCup | with AL | 99.65+0.03 | 99.67 | 40.70+1.18 | 41.15 | 99.64+0.03 | 99.67 | 3878.41+46.16 | 3884.23 |
| | without AL | 99.68+0 | | 41.59+1.83 | | 99.7+0.01 | | 3897.04+50.52 | |

The similar trend is also shown in the testing performance. Voting method suffers from computational inefficiency. The testing time required for the voting method for all datasets was almost 100 times slower than those of testing time carried out by the merging method. This problem is understandable because the number of testing is carried out in every sub-models (model before merging). As the number partition of the Scalable PANFIS framework uses 96 partitions, the number of testing is carried out at least 96 times.

### 4.5.2.2 Scalable PANFIS and Spark-based Algorithms Comparisons

The performance comparison between all Scalable PANFIS frameworks and Spark-based algorithms are summarized in Table 4.8 and Table 4.9 in terms of accuracy and running time, respectively.Note that this experiment is performed using five times cross-validations procedure.

In general, in terms of accuracy (Table 4.8), all Scalable PANFIS algorithms have a similar performance for all datasets. For example, for the SUSY, HIGGS, HEPMASS, RLCPS, Poker

Hand, and KDDCup datasets, the Scalable PANFIS algorithms demonstrate an accuracy of around 76, 64, 83, 99, 50, and 99 percent, respectively. Conversely, for the Spark-based algorithms, the accuracy for some of the datasets is not the same. For the SUSY and HIGGS datasets, for example, the Spark.KMeans algorithm is outperformed by its counterparts with only 49.73 and 48.82 percent accuracy in comparison with other Spark-based algorithms, with around 75 and 60 percent accuracy. Table 4.8 also shows that Scalable PANFIS algorithms outperform Spark-based algorithms in terms of accuracy with a significance difference of accuracy in dataset SUSY, HIGGS, and HEPMASS (around 3 percent). However, for the remaining datasets (RLCPS, Poker Hand and KDD Cup), most of Scalable PANFIS algorithms accuracy are slightly lower than Spark.RF.

Table 4.9 shows that most of the Spark-based algorithms perform faster when they train large-scale data. However, in one case (Poker Hand dataset), all the Spark-based algorithms (Algorithm 5-8) perform slower than the Scalable PANFIS algorithms (Algorithm 1-4). Of the Spark-based algorithms, Spark-GLM consumes more time than the other Spark-based algorithms. In some cases (SUSY and Poker Hand dataset), Spark-GLM also performs slower than Scalable PANFIS with AL algorithms.

Table 4.8: Accuracy for all datasets and algorithms using 5-fold cross validations

| **Algorithm** | **Accuracy (%)** | | | | | |
|---|---|---|---|---|---|---|
| | SUSY | HIGGS | HEPMASS | RLCPS | Poker Hand | KDD Cup |
| Scalable PANFIS Merging | 76.46+0.18 | 63.9+0.15 | 83.49+0.07 | 99.91+0.13 | 50.12+0.09 | 99.68+0 |
| Scalable PANFIS Voting | 75.66+0.04 | 64.05+0.20 | 83.51+0.05 | 99.98+0 | 48.23+0.5 | 99.7+0.01 |
| Scalable PANFIS with AL Merging | 76.08+0.13 | 63.93+0.18 | 83.49+0.03 | 99.93+0.05 | 50.12+0.1 | 99.65+0.03 |
| Scalable PANFIS with AL Voting | 75.38+0.13 | 63.76+0.22 | 83.47+0.04 | 99.91+0.15 | 48.60+0.19 | 99.64+0.03 |
| Spark.KMeans | 49.73+0.48 | 48.82+1.26 | 50.09+0.55 | 70.34+18.43 | 50.12+0.09 | 39.62+12.52 |
| Spark.GLM | 74.91+0.03 | 63.80+0.13 | 83.48+0.06 | 99.98+0 | 50.12+0.09 | 99.70+0.01 |
| Spark.GBT | 73.5+0.09 | 59.35+6.18 | 77.8+1.02 | 99.64+0 | 50.12+0.09 | 97.04+0.02 |
| Spark.RF | 75.88+0.26 | 59.77+0.51 | 80.91+0.27 | 99.98+0 | 50.63+0.60 | 99.9+0.04 |

#### 4.5.2.3 Statistical Testing

In this thesis, we conduct a statistical testing (hypothesis testing) to measure the performance of all algorithms over multiple datasets. We use Wilcoxon signed-ranked test, a non-parametric statistical test which considers the magnitude of difference and sign between two paired samples

Table 4.9: Running time for all datasets and algorithms using 5-fold cross validations

| **Algorithm** | **Running Time (s)** | | | | | |
|---|---|---|---|---|---|---|
| | SUSY | HIGGS | HEPMASS | RLCPS | Poker Hand | KDD Cup |
| Scalable PANFIS **without AL** (merging and voting) | 842.11+18.11 | 4279.8+131.44 | 3877.84+90.08 | 527.81+68.30 | 164.24+7.77 | 3443.68+116.89 |
| Scalable PANFIS **with AL** (merging and voting) | 417.02+12.76 | 2009.97+57.32 | 1852.66+50.94 | 341.40+38.42 | 86.91+5.91 | 1530.83+31.22 |
| Spark.KMeans | 132.56+6.52 | 562.41+153.47 | 475.33+69.44 | 77.28+9.43 | 333.76+19.78 | 415.99+74.91 |
| Spark.GLM | 436.26+21.25 | 1223.4+112.6 | 1200.32+80.2 | 308.43+8.55 | 953.02+63.47 | 812.71+35.52 |
| Spark.GBT | 346.4+34.52 | 813.45+193.38 | 644.05+47.04 | 191.75+43.92 | 815.89+33.41 | 712.31+79.88 |
| Spark.RF | 147.54+5.12 | 377.27+203.69 | 454.72+60.1 | 174.79+16.8 | 542.47+48.61 | 457.44+46.56 |

(e.g. vector of accuracies of two algorithms). The accuracy is considered as the only performance metric we used for testing since the running time performance is obviously different one to the others—for example, the running time of Spark.KMeans (algorithm 5) is obviously faster than Scalable PANFIS with AL (algorithm 4) on all datasets (Table 4.9).

For each algorithm, there are 30 observations (vector of accuracies) to be compared as a result of performing five times cross-validation over six datasets as illustrated in the Table 4.10. The vector v1 reflects the accuracy of Scalable PANFIS Merging or Scalable PANFIS Voting on SUSY dataset on first cross-validation. The Vector v2 up to v30 reflect the results of both algorithm on SUSY dataset on the second cross-validation all the way to KDDCup on the fifth cross-validation.

Table 4.10: Statistical Testing of Two Paired Algorithms using Wilcoxon Signed-Ranked Test

| Algorithm | Vector of accuracies | | | | | | | | | | | | **Statistical Test Results** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | v1 | v2 | v3 | v4 | v5 | .. | .. | v26 | v27 | v28 | v29 | v30 | $V_{statistic}$ | $p_{Value}$ |
| Algorithm 1 | 76.47 | 76.51 | 76.16 | 76.57 | 76.61 | ... | ... | 99.68 | 99.68 | 99.68 | 99.68 | 99.67 | **283** | **0.309** |
| Algorithm 2 | 75.66 | 75.65 | 75.71 | 75.68 | 75.61 | ... | ... | 99.71 | 99.70 | 99.70 | 99.70 | 99.68 | **182** | **0.309** |

It can be seen in Table 4.10 that the statistical test results show that algorithm 1 performs better in accuracy, having higher $V_{statistic}$ value than algorithm 2 (283 against 182), despite the significance difference $p_{Value}$ is still higher than significance level $\alpha_{significance}$.

Since there are eight algorithms to be compared, all of the statistical testing using Wilcoxon signed-ranked are stored in $8 \times 8$ performance matrix. Table 4.11 shows the matrix of statistical testing which store $V_{statistic}$ values and $p_{values}$ for every pair of compared algorithms. In this context, $V_{statistic}$ reflects the score obtained from the sum of ranks multiplied by positive sign

when an algorithm outperforms another algorithm at a pair of observation, whereas $p_{value}$ reflects the measure of significance difference between two compared algorithms. For example, the statistical testing measure of accuracy between Scalable PANFIS Merging (algorithm 1) and Scalable PANFIS Voting (algorithm 2) is stored in first row and second column of Table 4.11, where $V_{statistic}$ value and $p_{value}$ of this test are **283** and **0.309** respectively.

From the same Table 4.11 it is clear that algorithm 1 statistically better than algorithm 2, having higher $V_{statistic}$ value than algorithm 2. However, using significance level $\alpha_{significance} = 0.05$, it is considered that algorithm 1 is not significantly better than algorithm 2, where the measure of significance difference $p_{Value} = 0.309$ is greater than $\alpha_{significance}$, thereby the null hypothesis is not rejected. Note that the significance level $\alpha_{significance} = 0.05$ is equivalent with $V^{left}_{critical} = 137$ for left tail and $V^{right}_{critical} = 328$ for right tail for 30 observations. $V^{right}_{critical}$ is the minimum statistical value (sum of ranks) an algorithm is considered significantly better than another, and vice versa.

Furthermore, algorithm 1 (Scalable PANFIS Merging) also outperforms of other algorithms, especially compared to algorithm 4, 5, and 7 significantly where the values of significance difference are 1.89e-04, 2.61e-08, and 1.11e-04 respectively (all of them are below the $\alpha_{significance}$). In contrast, algorithm 5 (Spark.KMeans) is outperformed by other algorithms significantly, having $V_{statistic}$ below $V^{left}_{critical}$ and $p_{Value} < \alpha_{significance}$. In Table 4.11, we highlight the statistical measure in bold, the one that provides non-significance difference between two algorithms.

Of the Scalable PANFIS structure, only algorithm 4 is outperformed by one of Spark-based algorithm, particularly Spark.GLM. However, there is no significance difference of statistical result as its $p_{value} = 0.25$ greater than $\alpha_{significance}$. Of the Spark-based algorithms, Spark.GLM is considered as the best performer, significantly better than Spark.KMeans and Spark.GBT, and slightly better than Spark.RF, having higher $V_{statistic}$ than Spark.RF (255 against 210).

In summary, Scalable PANFIS algorithms (1-4) are generally better than Spark-based algorithms (5-8) statistically, even though for particular algorithms (Spark.GLM and Spark.RF) the statistical test results show that there is no evidence that Scalable PANFIS methods are significantly better in accuracy. Furthermore, Wilcoxon signed ranked test provide a better

description about overall performance. For most notable example is the comparison between Scalable PANFIS Merging and Spark.RF. Scalable PANFIS Merging better than Spark.RF in the three datasets (Susy, Higgs, and Hepmass), whereas Spark.RF outperforms Scalable PANFIS Merging in RLCPS, PokerHand and KDDCup. In fact that Spark.RF only exceeds the Scalable PANFIS Merging with small amount of difference (around 0.3 percent), whereas Scalable PANFIS Merging achieve accuracies with much higher difference (around 4 percent). Wilcoxon signed ranked test considers the magnitude (differences of accuracies), where the higher gap performance is multiplied by the higher value. It can be seen that $V_{statistic}$ of Scalable PANFIS is higher than $V_{statistic}$ of Spark.RF (329 against 136).

Table 4.11: Matrix of statistical testing using Two-tailed Wilcoxon signed ranked test on 8 algorithms using $V_{statistic}$ values and $p_{Values}$

| Comparison | $V_{statistic}$ values/$p_{values}$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Scalable PANFIS Merging (1) | Scalable PANFIS Voting (2) | Scalable PANFIS with AL Merging (3) | Scalable PANFIS with AL Voting (4) | Spark. KMeans (5) | Spark. GLM (6) | Spark. GBT (7) | Spark. RF (8) |
| Scalable PANFIS Merging (1) | NA | **283/0.309** | **295/0.205** | 404/1.88e-04 | 459/2.60e-08 | **293/0.221** | 458/3.54e-08 | 329/0.047 |
| Scalable PANFIS Voting (2) | **182/0.309** | NA | **194/0.44** | 310/0.047 | 450/2.55e-07 | **267/0.49** | 409/1.11e-04 | **264/0.529** |
| Scalable PANFIS with AL Merging (3) | **170/0.205** | **271/0.44** | NA | 390/0.001 | 458/3.54e-08 | **283/0.309** | 458/3.54e-08 | **299/0.177** |
| Scalable PANFIS with AL Voting (4) | 61/1.89e-04 | 125/0.047 | 75/0.001 | NA | 450/2.55e-07 | **179/0.28** | 418/3.90e-05 | **261/0.57** |
| Spark.KMeans (5) | 6/2.61e-08 | 15/2.55e-07 | 7/3.54e-08 | 15/2.55e-07 | NA | 0/1.31e-05 | 1/1.48e-05 | 0/1.86e-09 |
| Spark.GLM (6) | **172/0.221** | **198/0.49** | **182/0.309** | **286/0.28** | 325/1.31e-05 | NA | 325/1.31e-05 | **255/0.655** |
| Spark.GBT (7) | 7/3.54e-08 | 56/1.11e-04 | 7/3.54-08 | 47/3.90e-05 | 324/1.48e-05 | 0/1.31e-05 | NA | 54/8.85e-05 |
| Spark.RF (8) | 136/0.047 | **201/0.529** | **166/0.177** | **204/0.57** | 465/1.86e-09 | **210/0.655** | 411/8.86e-05 | NA |

### 4.5.3 Summary Discussion

This section summarises the methods, algorithms, and results obtained in this experiment. There are at least six points we can discuss in relation to this experiment. As discussed in the results section, firstly, the AL strategy is embedded in the PANFIS machine learning algorithm to speed up the training process by selecting samples to be trained in the worker node of the Scalable PANFIS framework. For many cases of large-scale data stream processing, reducing the number of samples to be trained slightly reduces the accuracy without a significance difference.

For smaller datasets, such as Poker Hand, distributed machine learning PANFIS using a voting technique yields a lower accuracy than the others (48.23 and 48.60 percent for algorithm 1 and 3 compared to the others). This is due to the small size of the Poker Hand dataset compared to the other datasets. With around 800k of total samples, if it is divided into 96 partitions, each partition will have around 8k samples. With the further AL applied on each partition, the samples trained in each chunk is around 3.2k (assuming the compression rate is around 0.4). Hence, the training process in all the partitions is not converged.

Secondly, both the model merging and voting techniques yield similar performance results in terms of accuracy. The voting mechanism discards the less supported decision made by the **Weaker** models which generate a false classification output, whereas the model merging mechanism discards the models which have a lower confidence level (lower weight/lower classification training results) thus resulting in better inference results. Furthermore, the over-complex rule base leads to the overfitting issue and thus deteriorates the generalization ability.

Thirdly, the PANFIS architecture is designed for MIMO architecture. In the case of binary classification problems, the Spark-based algorithm can directly process the data. However, for the multi-class classification problem, Spark-based algorithms need to be modified into the One-Versus-All (OVA) form. Therefore, for the Poker Hand dataset, Spark-based algorithms require a longer time to process the training data, as shown in Table 4.9.

Fourthly, the large-scale data stream framework based on PANFIS accelerates the training process by processing all the data partitions in parallel mode. From each partition, one single model is generated by PANFIS with or without AL. In order to gain a final model, model merging methods, such as model merging and the voting mechanism are applied because simply concatenating the data partition model can result in the overfitting issue deteriorating the generalization ability of the concatenated model because some rules may be overlapping.

Fifthly, in general, all Scalable PANFIS algorithms produce similar accuracy without a significance difference. However, it can be concluded that the use of voting method, in general, slightly decreases the accuracy. This is because the number of local models that have lower gen-

eralization powers influence the final results. This fact also emphasizes that the model selection should be considered prior the voting rather than using all local models directly to infer the testing data. Therefore, using limited rules/local models as described in Table 4.6 is one of the way to keep the quality of the distributed model. Furthermore, using voting method is costly when the number of local models involved is very high (more than 96 local models). Table 4.7 shows that the testing/inference time of voting method is almost 100 times slower than inference of model generated from model merging, due the compact of its structure. This fact also confirms that our designed model merging method robust to handle evolving distributed data model, especially in dealing with the outliers.

Finally, we design the robust model merging method by selecting the initial rules which have the highest weight and applying rule removal before the model merging process, as can be seen in the performance (accuracy and number of rules after merging). The model merging process is explained in algorithm 3. Rule removal is performed based on the consideration that the rules which have less support are considered as outliers, thus this could reduce the generalization capability of the classification performance.

## 4.6 Conclusion

The evolving large-scale data stream analytics framework, namely Scalable PANFIS, demonstrates distributed data stream processing. Scalable PANFIS combines two ways of dealing with the large volume of large-scale data: streaming algorithms and distributed computing. It makes use of a PANFIS evolving algorithm which is adaptive to the changing data stream patterns and works on the single-pass learning scheme. The Spark distributed processing platform ensures that PANFIS can be scaled up into the desired number of nodes, enabling it to process large-scale data stream examples. This combination ensures that the final model generated from Scalable PANFIS (the initial distributed models followed by the model merging/aggregation) is kept up to date, and the generalization performance does not decrease. On the training side, the AL method is also applied in the framework to further speed up

the PANFIS training process in the partition level with comparable accuracy. On the testing side, the majority voting method is also implemented. The results show that model merging is slightly better to the majority voting method. All in all, the four structures of the Scalable PANFIS approaches demonstrate the comparable accuracy with a slight decrease of accuracy in voting method. Furthermore, a model selection should be incorporated before applying the voting method because over complex model not only slows the testing time but also degrades the generalization performance.

From this point, we can conclude that the needs of the advanced framework is vital for processing data streams. That is, a framework should be effective in coping with the vast volumes of data generated, the changing patterns, and the continuous arrival of data streams. While Scalable PANFIS has been proven to effectively handle the large-scale and changing patterns of data streams, it has not been evaluated in the continual learning environment which is vital to ensure the scalability of the algorithm. In the next chapter, the incremental learning framework is proposed to cope with sequential data streams, which arrive batch-by-batch. Furthermore, a prequential Scalable PANFIS is also considered as a benchmark algorithm to show the effectiveness of a new framework whose architecture is detailed in the subchapter 2.7.1. The learning performance is evaluated task-by-task using prequential test-then-train, a standard evaluation method for data stream learning.

# 5

# Scalable Teacher-Forcing Networks under Spark Environments for Large-Scale Streaming Problems

## Abstract

A novel self-organizing structure of ensemble under incremental learning scheme using a distributed processing platform is proposed. Unlike Scalable PANFIS, ScatterNet differs from Scalable PANFIS in that its learning mechanism is evaluated under the test-then-train protocol allowing it to process never-ending batches of training data. While the generation of the fast-evolving distributed model in Scalable PANFIS is undertaken in one task, the distributed learning performed in ScatterNet takes place in sequential tasks. Furthermore, Scalable PANFIS does not adopt an elastic structure evolution in learning from data streams which is vital to handle concept drift. ScatterNet adopts the elastic network principle, where the number of base models (the base classifiers for a classification problem) in the ensemble network ($\mathbb{EN}$) evolves over time. Network performance is evaluated for every task, where a winning model is obtained. Each base model's performance in the $\mathbb{EN}$ is also assessed, and the base models are removed from the $\mathbb{EN}$ if they are considered to be inconsequential base models using a statistical measure. The distributed training of ScatterNet makes use of a winning model to be updated in a distributed learning system. Every partition is processed in the corresponding assigned node using a teacher-forcing method distributed across the Spark nodes. The initial

distributed models are formed for the first time after the distributed training task. Then, a data-free model merging compresses them into a merged model as a candidate base model in the $\mathbb{EIN}$. The $\mathbb{EIN}$ is expanded and a candidate base model is added into the $\mathbb{EIN}$, once drift across the batch is detected. Otherwise, a candidate model replaces the winning model.

## 5.1 Introduction

The problems of large-scale data streams are related to learning from large-sized data whose size is beyond the traditional computing platform to process [107]. In the era of IoT, smart devices which are interconnected might produce a large amount of data as streams, and this can lead to a data explosion problem. While vast volumes of data are beneficial for decision-making purposes, it also demands advanced machine learning methods to handle the large-size and fast sampling rate of data streams [108]. While data stream problems are commonly treated by various incremental or online learners as described in [30] with their ability to handle continuous data, the issue of large-scale data streams has not received adequate research study. Most of the existing approaches are built under a standard single processing node architecture. While data streams might arrive in batches periodically, the single node architecture might not be able to keep up to process a large-size data stream batch.

Learning using a distributed computing platform is becoming the dominant way of dealing with large-scale data [109]. This approach aims to learn huge volumes of static data distributively in a single shot process. However, for continuous large-scale data streams, the learning approach should be designed to cope with data stream characteristics. The distributed learning approach faces several challenges when dealing with data streams. Firstly, the problem of the structural complexity of a model might exponentially increase concerning the generation of several models for each data stream accumulated over time (caused by the nature of distributed computing platform, drifts, shifts, and new operation modes or dynamic system states). Secondly, there is a decrease in the model's accuracy (due to a complex structure which leads to the flexibility issue). Our experimental finding shows that a simple combination of neurons or rules from

distributed nodes deteriorates the model's accuracy because they are trained using different partitions of a massive data batch and are optimized independently with a lack of synergy among them.

Apache Spark (Spark) is a distributed data processing platform that is suitable for use in a wide range of computational tasks. Particularly, in big data and machine learning tasks, the Spark platform is powerful enough to process massive datasets across multiple nodes [23]. The Spark platform is more effective than MapReduce [109] in streaming environments because it is designed to be fast for iterative algorithms and process the data directly from the memory instead of the disk. The kNN-based distributed algorithm was developed in [34] using the Spark platform to handle large-scale data streams. It utilizes the distributed metric tree for sample selection to speed up the calculation of the distance of kNN. However, kNN differs from neural networks, where the underlying challenge lies in model selection if being undertaken in the distributed computing environment. While [110] also performs large-scale data streams analytics using evolving fuzzy rule-based classifiers, it has not been configured to evaluate sequential data in the prequential test-then-train scheme and the lifelong setting. Work on distributed learning was also carried out using the graphics processing unit (GPU) instead of Spark in [111]. It puts forward the evolving rule-based classifier, which is constructed using the genetic programming method. Note that a GPU can also be used as an alternative platform on which to perform distributed learning tasks, although it requires a unique setting to demonstrate mini-batch execution. The advantage of Spark over the GPU is that Spark is a cloud-based platform, whose computing nodes can be expanded anytime on demand without changing the algorithm operating in each computing node.

A large-scale streaming algorithm, namely Scalable Teacher-Forcing Networks (ScatterNet), is proposed in this chapter under the Spark platform. ScatterNet makes use of the teacher-forcing concept. The teacher-forcing concept offers an internal memory to capture the temporal dynamics of data streams while avoiding iterative gradient calculations. The teacher-forcing used in this work is the extension of [112] where the parameter learning mechanism is performed without using the back-propagation method. The issue of massive data streams is solved using

a distributed learning strategy utilizing Apache Spark. A zero-shot merging mechanism is proposed to effectively merge the several models generated from learning from every large-scale data stream batch. The merging procedure solves the structural complexity due to the nature of distributed processing without deteriorating the generalization performance. The model simplification procedure, model merging, can help to address the problem of lifelong learning under a distributed computing environment.

ScatterNet adopts an open structure capable of coping with **concept drift on local and global scales**. Note that local drift here refers to drift evaluation using the teacher-forcing module at the partition level, where large-scale data streams are partitioned and distributed to all nodes across the cluster. At the global level, the drift detection method is employed to mark global concept drift followed by introducing a new model representing the new, drifted concept.

The major contribution of ScatterNet is as follows:

- ScatterNet solves the problem of learning from large-scale data streams utilizing the distributed computing platform, namely Apache Spark. ScatterNet employs the teacher-forcing network [32, 113, 112] to evaluate data streams at the local level (a data stream partition) taking into account the temporal characteristic of data streams without using a time-intensive iterative gradient calculation.

- ScatterNet introduces the data-free model merging strategy which encapsulates both the zero-shot merging strategy and online model selection. The former is capable of controlling the complexity of the model structure, whereas the latter provides the model selection mechanism before the merging procedure. Data-free model merging offers the same level of accuracy as that of its single node implementation.

- ScatterNet solves the local or within-the-batch drift making use of the teacher-forcing mechanism whose network is adaptively changing through the addition or pruning of its hidden nodes based on streaming examples. Furthermore, global or across-the-batch drift

is resolved using the drift detection method. A new model is incorporated into the existing EIN if a concept drift is signalled. Conversely, a model pruning mechanism is employed to alleviate the heavy network structure by removing insignificant models in the EIN.

- ScatterNet advances the network significance (NS) method [15, 114, 115] by incorporating the forgetting mechanism. The forgetting mechanism solves the concept drift issue which causes changes between the distribution of input data and the true label, $P(Y_k, X_k) \neq P(Y_{k-1}, X_{k-1})$. This forgetting mechanism is undertaken by out-weighing older learned (distributive) relations.

ScatterNet's performance is numerically evaluated under two scenarios, **large-size-small-number** and **small-size-large-number** using six large-scale data stream problems. Our numerical results demonstrate the improvements of performance over related work in the aspect of execution time while retaining comparable accuracy.

## 5.2 Problem Formulation of ScatterNet

The problem of large-scale data streams is defined as a situation where a process continuously creates data formulated as $\mathbb{B}_1, \mathbb{B}_2, .., \mathbb{B}_k, ..., \mathbb{B}_K \in \Re^N$ [1]. $K$ indicates the number of data batches which may be unbounded. $N$ expresses the size of a data batch. The size of $N$ is considered very large and it either cannot or is too costly to be learned by the single-node machine learning algorithm.

Data streams often appear without the presence of class labels $\mathbb{B}_k = X_k \in \Re^{N \times n}$ where $n$ is the dimensionality of input space. A labelling process is carried out to provide the input vector $X_k$ to their corresponding true class labels denoted as $Y_k \in \Re^N$. The one-hot-encoding scheme translates the target vector into an m-dimensional class vector so that a data stream can be expressed mathematically as $\mathbb{B}_k = (X, Y) \in \Re^{N \times n+m}$. $n$ and $m$ denote the dimensionality of input and output space, respectively. Large-scale data streams are simulated under the

prequential test-then-train scheme [1] where numerical evaluation is independently evaluated per data batch $\mathbb{B}_k$.

The large-scale data stream problem illustrates a real-world case where a batch of data samples must be processed promptly without decreasing its generalization power. In this case, $\mathbb{B}_k$ cannot be efficiently processed under a single computing node algorithm as they cannot keep pace with the large size of a batch of samples. From an offline learning point of view, the standard distributed learning strategy might be suitable to learn a typical big data problem. However, this technique might not be ideal for large-scale data stream problems where its underlying issue lies in controlling structural complexity, where the model merging procedure has to be carried out many times to deal with the never-ending data generation process. In practice, the model merging strategy should be carried out to maintain structural complexity without deteriorating accuracy, as achieved under a single processing node [110]. Furthermore, the issue of drift may occur both locally (within the stream) and globally (across the stream), where the drift handling strategy should be constructed thoughtfully.

## 5.3 Preliminaries

Apache Spark (Spark) is regarded as one of the latest distributed-computing platforms. In Spark, the data processing mechanism is executed in the memory cluster instead of on a disk. The ecosystem of Spark is partitioned into two parts: 1) spark-core; 2) programming interface core. While the former serves the latter by defining the instructions received from it using its lower-level library, the latter constructs a series of instructions to manipulate a large-scale data batch $\mathbb{B}_k$ using a high-level library namely Spark API. Spark API provides several programming languages such as Scala, Java, Python, R, and Spark MLib employing the Java Virtual Machine. In Spark, $\mathbb{B}_k$ is processed through the subsequent steps:

- Obtaining $\mathbb{B}_k$ as a data frame from the cluster and covert it into Spark DataFrames in the memory cluster.

- Several job tasks are created, and the necessary operations (e.g. training or testing data partition) are performed over the nodes.
- The output from the several processes are collected (e.g. training models and predictions collection).

Spark API also provides custom operations using its library, making it easy to manipulate Spark DataFrames. Furthermore, these operations are carried out in parallel, utilizing the memory cluster. As a result, it can speed up the data processing time. Due to this mechanism, Spark can accommodate the iterative processing, which is vital for incremental learning scheme.

In Spark, the incremental learning mechanism is carried out by processing data streams batch-by-batch or task-by-task. For each task, the prequential testing and training of data of $\mathbb{B}_k$ are performed in a distributed manner. A large-scale data stream batch $\mathbb{B}_k$ is divided into $P$ number of partitions where the driver node controls the distribution of $\mathbb{B}_k^p$ data partitions over several worker nodes in the cloud. For each partition processed on any node, a result is obtained (training model or an inference output). The same operation (the distributed prequential test-then-train) is repeated for the future batches utilizing models/parameters stored in the global environment generated from previous learning. Note that both testing and training data are executed in the Spark environment (green part in Fig. 5.1).

## 5.4 Scalable Teacher-Forcing Network

The learning policy of ScatterNet is illustrated in Fig. 5.1. ScatterNet adopts an open structure which assumes that the learning procedure is carried out from scratch without a pre-existing model. This means that the initial distributed models are obtained from the first data stream batch $\mathbb{B}_1$. ScatterNet characterizes a self-organizing network structure where hidden nodes evolve following the distributional variation of data streams. The evaluation of ScatterNet is carried out under the prequential-test-then-train scheme. Under this scheme, for every batch $\mathbb{B}_k$, the testing procedure is undertaken first followed by the training procedure. Once the

testing for each phase is done, the prequential error is recorded, producing an accuracy matrix $A \in \Re^N$. The matrix $A$ contains the value of 1 and 0, which represent a false and correct prediction, respectively.

ScatterNet is built based on the ensemble concept containing a stack of base models called the $\mathbb{EIN}$. Every model in the $\mathbb{EIN}$ is assigned a voting weight $\beta_i$, and its value reflects its relevance to cover the current concept. The changes of the voting weight are controlled by the penalty-reward procedure based on the prediction output made by every model (base classifier). In the case of a wrong prediction, the penalty scheme is applied so that the voting weight of the corresponding model decreases. Conversely, the voting weight increases if the base model predicts the test sample correctly. At the end of each task, every model has its weight according to their performance on the test samples. A model which has the highest testing accuracy is deemed to be the winning model.

The implementation of the drift detection mechanism is undertaken to detect drift across the batch utilizing the accuracy matrix. Once drift occurs, a new model is appended. Conversely, the system is regarded as a stable condition. In this case, $\mathbb{EIN}$ renews one of its base models, a winning model. The zero-shot merging simplifies the structure of the complex network resulting from the generation of $P$ distributed models of each distributed training task under the Spark environment. $P$ denotes the number of data partitions. ScatterNet can be deemed to be similar to the work of MUSE-RNN [113] where the multi-layer RNN using the teacher-forcing concept is developed. However, [113] has not been designed to cope with large-scale data streams. Furthermore, ScatterNet does not apply the back-propagation method for parameter learning. This practice hinders the issue of vanishing or exploding gradients and to escape the local optima trap. ScatterNet also demonstrates a distributed process for both testing and learning datasets, which are performed under Spark. The presentation of ScatterNet's learning policy is illustrated sequentially as per its order of Fig. 5.1. Note that in Fig. 5.1, a base model (classifier) generated at $k^{th}$ task is denoted by $y_k$. A base model $y_k$ has the same structure as $\mathcal{F}i$, a member of ensemble network $\mathbb{EIN}$ as depicted in Fig. 2.6. The use of symbol $y_k$ in Fig. 5.1 only aims to illustrate the evolution of base models during the evolution of $\mathbb{EIN}$. Otherwise,

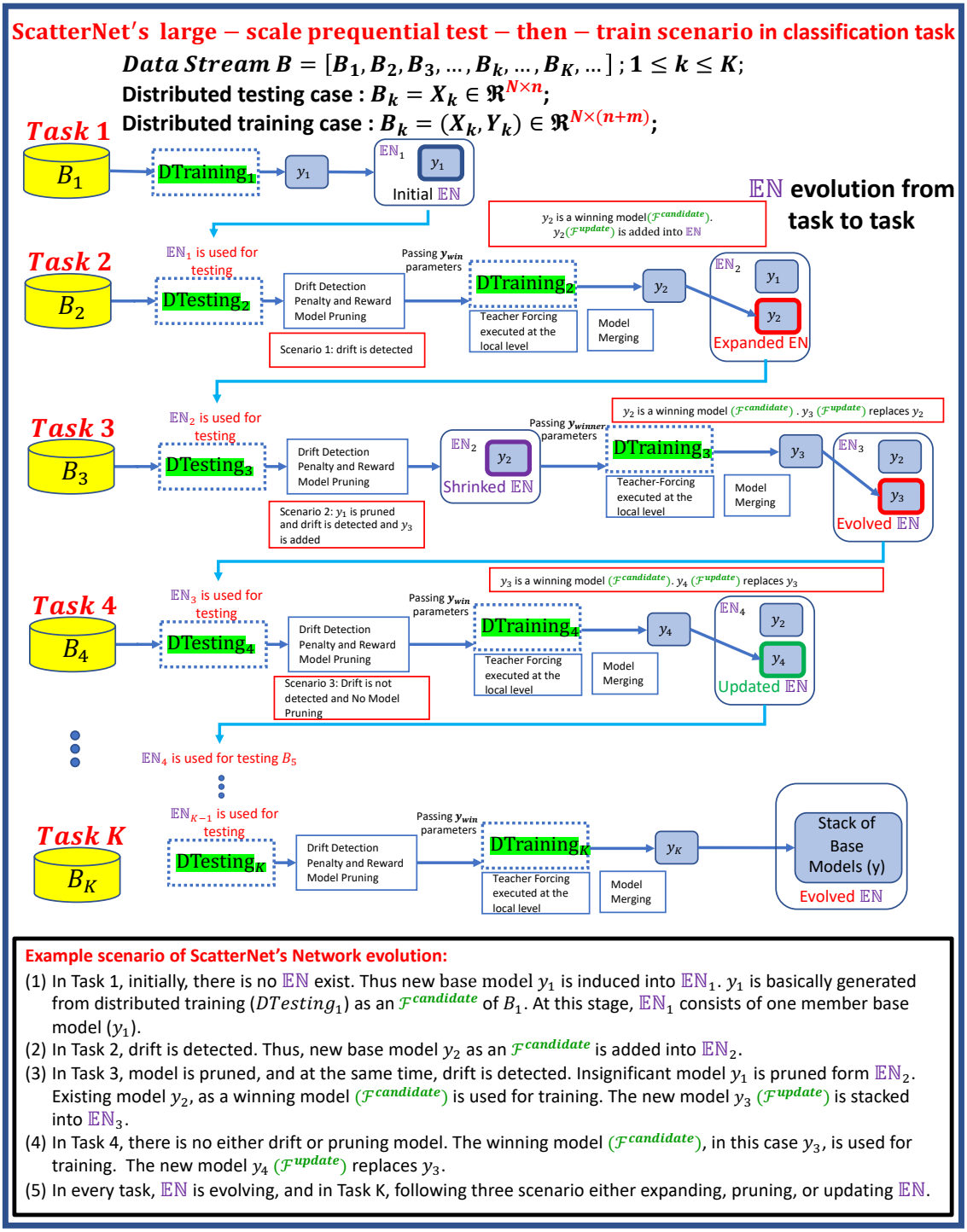


Figure 5.1: ScatterNet's learning policy and network evolution

we denote the evolved ensemble network as $\mathbb{EN}_k = \{\mathcal{F}1, ..., \mathcal{F}i, ..., \mathcal{F}I\}$ to describe that there are $I^{th}$ base models at $k^{th}$ task.

### 5.4.1 Penalty and Reward Mechanism

The penalty and reward mechanism controls the voting weight of a model $\beta_i$. The higher the voting weight of a model, the more substantial influence of a model on the classification decision (ensemble inference). In contrast, the models with less voting weight have less impact on the ensemble inference. A model receives a reward when it classifies samples correctly, whereas a penalty is imposed in the opposite situation. The penalty and reward mechanism is formulated as follows:

$$\begin{aligned} \beta_i &= \min(\beta_i(1 + fac), 1), \ (reward) \\ \beta_i &= \beta_i * fac, \ (penalty) \end{aligned} \tag{5.1}$$

where $fac \in [0, 1]$ denotes an adjustment factor. The minimum operator in the reward function (5.1) is inserted to limit the voting weight in the range of value $[0, 1]$. This strategy is conceived to achieve a stable weighted voting mechanism while pushing a model to be more sensitive to the current concept. In this way, its voting weight can be mitigated or recovered quickly.

The mechanisms of penalty and reward refer to some recent work [116, 15], in which they successfully implement similar strategies concerning the weighted voting method. Practically, in the distributed learning environment, the penalty and reward mechanisms of the particular model are undertaken in each data partition. Once, the voting weight of every data partition $\beta_i^p$ is collected, the final voting weight $\beta_i$ can be measured by solely using the average of all voting weights $\beta_i = (\sum_{p=1}^{P} \beta_i^p)/P$ from all partitions. This process is performed in a driver node as it only requires low computational resources.

### 5.4.2 Drift Detection Method

The drift detection method is implemented to identify the status of a data stream, e.g. whether the existing concept remains fit. The state of *stable* means that the current concept remains

relevant, whereas the status of *drift* implies that the concept may be slightly irrelevant to the data stream condition. In this chapter, we adopt the same principle as the work in [116, 15, 115] derived from the drift detection method in [117]. This method is initialized by determining the cutting point of accuracy describing the rise of the population means. The rise of the population indicates the model's performance has been compromised because of the binary characteristic of the accuracy matrix $A$ - 1 (false prediction), 0 (true prediction). The cutting point is established if it meets the following criteria: $\hat{A} + \epsilon_A \leq \hat{B} + \epsilon_B$. $\hat{A}$ and $\hat{B}$ stand for the statistics of the accuracy matrices $A$ and $B$, respectively. The $\epsilon_A$ and $\epsilon_B$ denote the Hoeffding bound of $A$ and $B$, respectively. $B \in \Re^{cut}$ is a partition of the accuracy matrix $A$ recording up to *cut* entries. The Hoeffding bound is calculated as in the work in [116, 15, 115] where $\epsilon_{A,B} = (b-a)\sqrt{\frac{size}{2*cut*size} ln(\frac{1}{\alpha})}$. $a$ and $b$ are the minimum and maximum points of interest while $size$ is the size of the matrix of interest. $\alpha$ is the significance level and is inversely proportional to the confidence level $1 - \alpha$.

Like other schemes in ScatterNet, drift is also monitored in every partition $B_k^p$ of a large-scale data stream. The cutting point is evaluated in every partition. Once eliciting the cutting point, the accuracy matrix of $B_k^p$ is divided into two subpartitions in respect to the cutting point *cut*. Assume that the two matrixes are established denoted as $B \in \Re^{cut}, C \in \Re^{N-cut}$. The drift condition is flagged by rejecting the null hypothesis $|\hat{B} - \hat{C}| \leq \epsilon$. The opposite case, in contrast, implies a stable situation. A drift situation means that the current model is under-fitting [116, 15].

To alleviate the under-fitting problem, a candidate base model $\mathcal{F}^{candidate}$ is appended to the existing $\mathbb{EN}$ as a new base model. $\mathcal{F}^{candidate}$ is basically a merged model obtained from three sequential processes: 1) selecting a winning base model $\mathcal{F}^{winning}$; 2) updating $\mathcal{F}^{winning}$ using the current data streams $\mathbb{B}_k$ (in the distributed training task); 3) merging the initial distributed models. In the mathematical notation, these three sequential processes are denoted as $\mathcal{F}^{candidate} = merge(\mathcal{L}(\mathcal{F}^{winning}, \mathbb{B}_k))$. The current $\mathbb{EN}$ after the addition of a candidate base model is denoted as $\mathbb{EN}_k = [\mathbb{EN}_{k-1}; \mathcal{F}^{candidate}]$, and $I = I + 1$. $\mathbb{EN}_{k-1}$ expresses the ensemble network containing $I^{th}$ number of base models before distributed training is conducted at $k^{th}$

timestamp. $\mathcal{F}I$ denotes $I^{th}$ base model, a new base model.

In another case, if drift is not detected, the stable condition, $\mathcal{F}^{candidate}$ replaces an $\mathcal{F}^{winning}$. In other words, the winning base model is updated to keep improving the predictive performance. Note that there is no warning case incorporated here as implemented in [115]. A warning presents a case where a drift requires substantiation of the next samples. This situation is unlikely to happen in ScatterNet due to the massive size of a data stream. Note that the drift situation is evaluated in every partition of a data stream $B_k^p$.

### 5.4.3 Model Pruning Mechanism

The focus of drift detection is to increase the complexity of the $\mathbb{EN}$ by adding a new base model in the network as a result of concept drift. As a counterbalance, model pruning mechanism is designed to remove the insignificant base models $\mathcal{F}i$. This mechanism is undertaken by evaluating every voting weight $\beta_i$ of $\mathcal{F}i$ in each task (in the distributed testing task), where voting weight $\beta_i$ is dynamically updated using a penalty and reward mechanism. The voting weight directly reflects the performance of each base model $\mathcal{F}i$. The lower the voting weight, the less important the model is to the ensemble output. Thus, it is considered should be removed to reduce the network complexity [118].

### 5.4.4 Data Stream Learning Phase

Once the testing phase of ScatterNet is executed, ensemble accuracy is evaluated followed by the execution of the three following methods: drift detection, penalty and reward, and the model pruning mechanism. The learning/training process is then undertaken, where the winning model as evaluated from the testing is updated in the distributed training system ($D_{training}$). Assume that the previous task's winning model is denoted as $\mathcal{F}^{winning}$ and the learning output of the distributed learning is denoted as $\mathcal{F}^{candidate}$. The detail of the transformation or process from $\mathcal{F}^{winning}$ to $\mathcal{F}^{candidate}$ is described as follows. The learning phase is undertaken in a

distributed fashion, where initially the distributed learning engine is induced by $P$ partitions $\mathbb{B}_k^p$. When we look particularly into a data partition $\mathbb{B}_k^p$ at the local level, the teacher-forcing mechanism (sub subsection 5.4.4.1) takes the initial parameter of $\mathcal{F}^{winning}$ to be evolved based on a data stream partition $\mathbb{B}_k^p$ condition. As a result, a local model denoted as $\mathcal{F}_k^p$ is produced. From this point, $P$ local models $\mathcal{F}_k^p$ are produced as the initial distributed models. Then, the zero-shot merging process compresses the initial distributed models into $\mathcal{F}^{candidate}$. Note that this merging is similar to the procedure of Scalable PANFIS in Fig. 4.2 with some of extensions.

#### 5.4.4.1 Scalable Teacher-Forcing Network

The ScatterNet's main algorithm, teacher-forcing, is derived from the concept of the hyperplane activation function, which is based on the notion of hyperplane clustering. [32, 119]. This idea takes into account the development of a hidden node utilizing a hyperplane defined as $W_r \in \Re^{(n+1)\times m}$. For each node, the activation degree is determined via the distance between point to hyperplane [32, 119] formulated as follows:

$$\begin{aligned} dis_r^c &= \frac{|(y_d^c \quad \textbf{or} \quad \hat{y}_{t-1}^c) - x_e * W_r^c|}{\sqrt{1 + \sum_{d=0}^{D-1} W_{r,d,c}^2}} \\ h_r &= \exp\left(-(\gamma dis_r^c)/(\max_{r=1,\ldots,R} dis_r^c)\right) \end{aligned} \tag{5.2}$$

where $\gamma$ and $h_r$ denote a control parameter and the hyperplane activation function of $r^{th}$ hidden node, respectively. $R$ represents the number of hidden nodes, $x_e = [1, x] \in \Re^{1\times(n+1)}$ is the extended input vector, whereas $y_d^c$ is the $i^{th}$ target variable.

The teacher-forcing mechanism [112] is depicted in (5.2) where the activation degree is determined from either target variable $y_d^c$ or the predictive output at the $t-1$ time instant with an equal proportion while the network output during the testing phase is fully produced by $\hat{y}_{t-1}^c$. That is, 50 percent of training samples are captured by $y_d^c$ while the remainder is processed by $\hat{y}_{t-1}^c$. The use of $\hat{y}_{t-1}^c$ generates an implicit recurrent connection offering short-term memory to cope with the temporal nature of data streams [113]. While teacher-forcing mechanism

was initially introduced in a recurrent neural network to address the slow convergence and instability during the training, it had not been applied in neuro-fuzzy system algorithm until PALM [32] was developed. Note that in the realm of data streams, instances arrive without label. The teacher-forcing mechanism is beneficial for data stream application as it overcomes the dependency to the label (target output) in order to calculate the distance for membership function.

The predictive output is obtained from the shared network parameters of the activation function and the hyperplane $W$ functions using a weighted average mechanism, which is formulated as follows:

$$\hat{y}_c = (\sum_{r=1}^{R} h_r x_e W_r)/(\sum_{r=1}^{R} h_r) \tag{5.3}$$

where the normalization is applied in (5.3) to enforce the partition of unity. ScatterNet make use of the shared network parameters in which the hyperplane $W_r$ generates activation degree $h_r$ as in formula(5.2). Both hyperplane $W_r$ and activation degree $h_r$ are used to infer the network output. ScatterNet's unique property lies in keeping the local model property in which each $r^{th}$ local model can produce its local output. The activation degree $h_r$ takes into consideration the granularity of input space in terms of local proximity, which can be drawn by the popular fuzzy **IF-Then** rules. In practice, the network output is similar to a standard Takagi-Sugeno Fuzzy Systems inference scheme for output estimation/prediction.

#### 5.4.4.2 Structural Learning of ScatterNet

The network significance (NS) governs the structural learning method of ScatterNet. NS determines the generalization performance of the model based on the bias-variance decomposition [15, 114, 115]. When high bias is signalled, the network implies the underfitting condition and hidden node growing is undertaken. Conversely, the high variance signifies the overfitting

condition. In this case, hidden node pruning is carried out. Bias and variance are defined as $Bias = (y - E[\hat{y}])^2$ and $Var = E[\hat{y}^2] - E[\hat{y}]^2$, respectively. The expected output $E[\hat{y}]$ is formulated as $E[\hat{y}] = \sum_{r=1}^{R} W_r \int_{-\infty}^{\infty} h_r(x; W_r) p(x) dx$.

The simplification of the hyperplane activation function $h_r$ is implemented as its integral solution which is difficult to estimate. $h_r$ is assumed to be at the maximum point when $h_r = 1$. Suppose the case of the normal distribution, $E[\hat{y}]$ can be solved as follows:

$$E[\hat{y}] = \sum_{r=1}^{R} \mu_e W_r \tag{5.4}$$

where $\mu_e \in \Re^{n+1} = [1, \mu]$ denotes the mean of data points. The $\mu$ here is obtained from the recursive calculation of historical data points. As the application of static $\mu$ is ineffective in coping with concept drift causing a change of data density $P(x)_t \neq P(X)_{t-1}$, some methods should be implemented to avoid the static $\mu$.

As per the implementation in [120, 121], the AGMM method is put forward to ease the strict normal distribution. The AGMM, however, incurs notable extra computational and space complexities because a mixture of Gaussian models has to be processed. As an alternative, we put forward the dynamic forgetting strategy because of the strict normal distribution. In this case, the recursive mean calculation [122] can be adjusted as follows:

$$\mu_t = \mu_{t-1} + (f_t / F_t)(X_t - \mu_{t-1}) \tag{5.5}$$

where $F_t = F_{t-1} + f_t$. In the case where the sample is treated equally, then $F_t$ and $f_t$ are defined as $F_t = t$ and $f_t = 1$, respectively. As with [117], $\Gamma = \exp(-Rate)$ denotes the forgetting factor. Moreover, the forgetting factor $f_t$ is scaled into the range $[0.9, 1]$. This value of range aims to enable smooth forgetting to avoid an unstable calculation [123] in the case of a too small forgetting factor. The lowest value of 0.9 presents the maximum forgetting case, whereas the highest value of 1 means there is no forgetting factor. The contribution of our method lies in the setting of $Rate$, which is dynamic depending on the drift rate. When the

drift rate is higher, the forgetting factor should be stronger.

In this chapter, we follow the definition of the drift rate implemented in [124] outlined from a distance between two consecutive mini-batches defined as follows:

$$Rate_t = \lim_{\Delta \longrightarrow \infty} \Delta D(t - 0.5/\Delta, t + 0.5/\Delta) \tag{5.6}$$

where $Rate_t$ is the drift rate of the $t^{th}$ data stream. The total variation distance is used here where the discretization is applied. Moreover, two data distributions are represented by forming two non-overlapping data groups at the mid-point of data streams $\mathbb{B}_k^p$. Note that $\mathbb{B}_k^p$ refers to the $p^{th}$ data partition of Spark operation task (either distributed training or distributed testing task). The use of the drift rate allows us to fit the so-called sweet path in [124]. That is, a high-bias-low-variance is generated in the case of a high drift rate while a low-bias-high-variance model is produced in the case of a low drift rate.

The statistical process governs the growing and pruning criteria [125]. However, the variable is directly obtained from bias ad variance instead of via the conversion of binomial distribution. The criteria of bias and variance are set as follows:

$$\mu_{Bias}^t + \sigma_{bias}^t \geq \mu_{Bias}^{min} + 2 * k_1 * \sigma_{bias}^{min}, \quad (Growing) \tag{5.7}$$

$$\mu_{var}^t + \sigma_{var}^t \geq \mu_{var}^{min} + k_2 * \sigma_{var}^{min}, \quad (Pruning) \tag{5.8}$$

Equations (5.7) and (5.8) are the extension of the famous $k$ sigma rule employing the strict Gaussian assumption. In this chapter, it is eased here by implementing the dynamic $k$ ranging from $[1, 2]$ formulated as in [115]:

$$k_1 = 1.25exp(-Bias^2) + 0.75 \tag{5.9}$$

$$k_2 = 1.25exp(-Var^2) + 0.75 \tag{5.10}$$

In practice, a new node is introduced in the model in the case of a high bias. In the case of a high variance, an insignificant node is pruned. Every time these two conditions occur, $\mu_{Bias}^{min}, \sigma_{bias}^{min}, \mu_{var}^{min}, \sigma_{var}^{min}$ are reset. In equation 5.7, a growing condition formula, term 2 is inserted showing that the growing mechanism is less preferred than the pruning mechanism. This choice aims to circumvent the exponential increase in network complexity. Note that a lot of new nodes are introduced concurrently in the parallelization process.

Condition (5.7) presents a high bias condition which triggers the expansion of a new node. Due to the local property of ScatterNet, only one node is integrated. The new hyperplane $W_{r+1}$ and its support $Sup_{r+1}$ are respectively initialized as :

$$W_{R+1} = wamp * \mathbb{1}_{(n+1),r}; \quad Sup_{R+1} = 1 \tag{5.11}$$

where *wamp* denote the predefined initial weight. The hidden node pruning process is carried out to the weakest node determined from (5.4) $\min_{r=1,\ldots,R} E[\hat{y}_r] = \mu_e W_r$ once (5.8) is satisfied. The support of pruned node is accumulated to the winning node ($Sup_{win} = Sup_{win} + Sup_{pruned}$). The pruning process aims to mitigate the high variance dilemma. If no growing or pruning mechanism is triggered, a sample is associated to a winning node and its support is incremented as per $Sup_{win} = Sup_{win} + 1$ [32].

#### 5.4.4.3 Parameter Learning of ScatterNet

The tuning of hyperplane $W_r$ is performed using the widely known fuzzily weighted generalized recursive least squares (FWGRLS), which are also adopted in evolving fuzzy systems [11, 12]. This approach is an extension of fuzzily weighted recursive least squares (FWRLS) in [47] where the major difference lies in the application of a regularization term $\zeta_r$ to prevent the overfitting problem and to retain the small and bounded weight parameter [11, 12].

The system updates the network parameters $W$ locally per node, where every node has its own output covariance matrix $\Omega_r \in \Re^{(n+1)\times(n+1)}$. Instead of the gradient descent method, the

covariance matrix is preferred here as it is formed as a well-converging recursive version of the least squares (LS) solution. In practice, the covariance matrix makes convergence to the global optimum within one iteration step possible, i.e. within one update step on a new incoming sample. However, the drawback of using the covariance matrix over the gradient descent method lies in dealing with the high-dimensional feature. The covariance matrix method imposes the use of feature selection to compress the size of covariance matrix $\Omega_r$. In terms of ScatterNet, the high-dimensional problem is beyond the scope of this chapter.

#### 5.4.4.4 Data-Free Model Merging

The aim of the model merging strategy is to simplify network complexity as a result of distributed training in every task ($D_{Training}$). The model merging procedure in ScatterNet is depicted in Fig. 5.2. Data-free model merging is the extension of model merging implemented in Scalable PANFIS. Model merging in ScatterNet makes use of data-free online model selection without pre-model selection as implemented in Scalable PANFIS.

Data-free model merging comprises two main components: zero-shot merging and online model selection. Commonly, model merging controls the structural complexity during the distributed training where the initial distributed models are compressed into a merged model [110]. Note that initial distributed models which contain $P$ sub models are induced by $P$ data partitions of Spark for every data stream. Model merging is necessary to control structural complexity, as the merged model can be used for a continual learning situation to be used for learning large-scale data streams of future tasks.

Suppose that $K, R$ stand for the number of data streams and the average number of hidden nodes across $P$ models, the total number of nodes goes up to $P*K*R$. Since $K$ is unknown and possibly unbounded, the structural complexity becomes untenable. Another issue is related to the issue of predictive accuracy because $P$ models are generated by different data distributions. Thanks to the local property of ScatterNet, an aggregated model does not suffer significantly from the issue of accuracy drop. This issue is obvious if a model is trained under a global

optimization strategy such as conventional recursive least squares with a single global covariance matrix $\Omega \in \Re^{(n+1)*R \times (n+1)*R}$.

The model merging strategy is crafted from the concept of hyperplane merging. Two hyperplanes are considered to be similar to each other if they possess similar angles [53] and are close to each other $Dist(r, r')$ [126] where $r \neq r'$. The distance and similarity of two hyperplanes are thus measured as follows:

$$Dist(r, r') = (||W_r - W_{r'}||)/(||W_r + W_{r'}||) \tag{5.12}$$

$$Sim(r, r') = \phi/\pi;\ \phi = \arccos(a^T b)/(|a||b|) \tag{5.13}$$

where (5.12) stands for the normalized distance between two hyperplanes while (5.13) expresses the similarity based on the dihedral angle between the two normal vectors $a = [W_{r,1}, W_{r,2}, ..., W_{r,n}, -1], b = [-W_{r',1}, -W_{r',2}, ..., -W_{r',D}, 1]$ [53] of the two hyper-planes (represented by the regression coefficients of the inputs and $-1*y$, because $W_{r,1}*x_1+...+W_{r,n}*x_n-y = -W_{r,0}$, equivalent to the original regression formulation $y = W_{r,1} * x_1 + ... + W_{r,n} * x_n + W_{r,0}$, denotes the normal vector form of a plane, with $W_{r,0}$ the intercept). The normal vector of the second plane $b$ has to be set to the opposite direction (thus being multiplied by $-1$), to achieve a (desired) high similarity when the two normal vectors point to the same direction (and thus have a low original angle $\phi$, which is then turned into an angle of $\pi - \phi$, leading to a similarity $Sim(r, r')$ close to 1).

**Zero-Shot Merging Mechanism**: the hidden node merging process has to be carried out carefully due to the presence of poor hidden nodes significantly compromising the generalization of a merged node. That is, the accuracy of a resultant node deteriorates if the merging process involves poor nodes [110]. A poor hidden node is one which has minor support or poor accuracy. The minor support is interpreted as a hidden node covering a low variance direction of data distribution. Accuracy here refers to the training classification rate when learning $\mathbb{B}_k^p$ data partitions. The hidden node merging process starts with the removal of inconsequential hidden nodes, defined as hidden nodes with a low number of supports formalized as $Sup_r \leq q\% * N$,

where $q$ is set to 2 in all the experiments.

Hidden node merging based on hyperplane similarity was proposed in [53] and is implemented in a one-to-one fashion. ScatterNet differs by implementing the $Z$-best-nodes-merging strategy. That is, the $Z$-best-nodes are extracted based on their training accuracy [110]. Once finding the $Z$ best nodes, other nodes are merged to those $Z$ nodes by following the angle and distance concepts in (5.12, 5.13). Fig. 5.2 on the left-hand side visualizes data-free model merging. Unlike its predecessor in [110] where the number of best nodes $Z$ are blindly selected, an online model selection process is implemented here to determine the number of $Z$.

**Online Model Selection**: the online model selection process is carried out to select the parameter $Z$ on the fly rather than setting it as a hyper-parameter leading ScatterNet to be ad hoc. Online model selection initiates with the generation of $Z$ candidates where $Z = 3, 5, 8, 10$ are selected. The training accuracy is used to induce the $Z$ best nodes. That is, other nodes are coalesced to the $Z$ best nodes. The online model selection mechanism of the 4 candidates is performed by minimizing the tradeoff between the bias $Bias$ and variance $Var$ as follows:

$$\min_{Z=3,5,8,10} |Bias - Var| \tag{5.14}$$

The candidate minimizing (5.14) is chosen. In the case of drifts, the selected model is added as a new base classifier embracing the new concept of the data stream. On the other hand, it will replace the winning base classifier if there is no drift detected in $\mathbb{B}_k$ as an effort to improve the predictive performance.

The online model selection strategy drives the ScatterNet base classifier to arrive at an optimal complexity satisfying bias and variance trade-off. As a result, the generalization power of the network can be preserved. Note that $Bias$ and $Var$ are enumerated by exploiting the aggregated mean across $P$ data partitions $\sum_{p=1}^{P} \frac{\mu_p}{P}$. Because of $Bias = (y - E[\hat{y}])^2$, $P$ data samples are randomly sampled from $P$ data partitions meaning that a single sample of each data partition is picked.

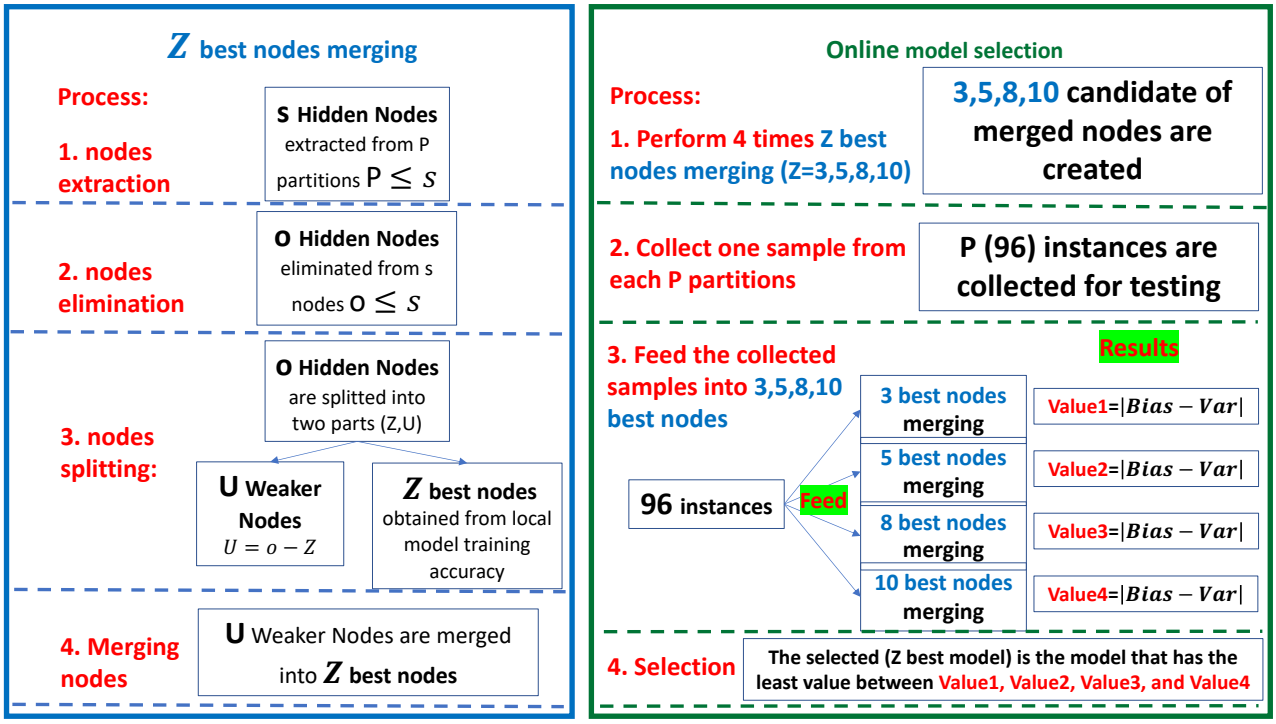


Figure 5.2: Model merging mechanism

## 5.5 Numerical Results

In this section, the effectiveness of ScatterNet is examined in two cases: **large-size-small-number** and **small-size-large-number**. An ablation study is conducted to analyze each learning component. **The source codes of the consolidated algorithms and raw numerical results of all the experiments are available at this link** `https://bit.ly/2TSRGFJ`.

### 5.5.1 Dataset

The advantage of ScatterNet is demonstrated using six popular big data problems namely Hepmass, Higgs [127], Susy [127], RLCPS [128], KDDCup [129], and Poker Hand [130]. All problems are of a large size to simulate real data stream environments. The properties of the datasets are given in Table 5.1.

A prequential test-then-train procedure [1] is used to demonstrate ScatterNet's lifelong learning over $K$ sequential tasks, where each task is tested first before it is trained in a distributed

manner as depicted in Fig. 5.1. The experimental setting depicted in Table 5.2 shows that the ScatterNet processes the flow of data batches sequentially.

Table 5.1: Properties of Datasets

| Dataset | #IA | #C | Size (# instances) |
|---|---|---|---|
| Higgs | 18 | 2 | 11,000,000 |
| Hepmass | 28 | 2 | 10,500,000 |
| Susy | 18 | 2 | 5,000,000 |
| RLCPS | 9 | 2 | 5,000,000 |
| KDDCup | 2 | 41 | 4,898,432 |
| Poker Hand | 10 | 2 | 1,025,011 |

IA: input attributes, C: classes

### 5.5.2 Algorithms and Parameters

The conducted experiment has been compared with the previous work namely Scalable PANFIS [110]. It does not feature the lifelong learning scheme. Some modifications are applied from the original code of Scalable PANFIS to make it applicable in the lifelong learning scenario.

The hyperparameters are hand-tuned to find the best possible results for all datasets. Once they are tuned, they are kept fix for all experiments. ScatterNet parameters $\alpha_{drift}$ is set to 0.0005 which controls the drift rate of the output at the global level. At the local level, the control parameter $\gamma$ is set to 0.7, whereas *wamp*, a coefficient for weight initialization of the first

Table 5.2: Experimental Setting

| Dataset | **ScatterNet** Large size small number scenario | | | | **ScatterNet** Small size large number scenario | | | |
|---|---|---|---|---|---|---|---|---|
| | #Batch | #Partition per batch | Batch size (# instances) | Partition size (# instances) | #Batch | #Partition per batch | Batch size (# instances) | Partition size (# instances) |
| Higgs | 66 | 96 | 166,666 | 1,736 | 189 | 96 | 55,555 | 578 |
| Hepmass | 63 | 96 | 166,666 | 1,736 | 198 | 96 | 55,555 | 578 |
| Susy | 30 | 96 | 166,666 | 1,736 | 90 | 96 | 55,555 | 578 |
| RLCPS | 30 | 96 | 166,666 | 1,736 | 90 | 96 | 55,555 | 578 |
| KDDCup | 29 | 96 | 168,911 | 1,759 | 87 | 96 | 56,303 | 586 |
| Poker Hand | 6 | 96 | 170,835 | 1,779 | 18 | 96 | 56,945 | 593 |

sample, is set to 0.2. The merging parameter threshold, *dist* and *angle*, are set to 0.6 and 0.4 as the criteria used for similarity distance and dihedral angle of two hyperplanes, respectively. For Scalable PANFIS, three parameters are set: $k_{grow} = 2$, $k_{prune} = 3$, and $kfs = 0.05$ which represent the growing threshold, pruning threshold, and safety width respectively. Note that all of these parameters are fixed in all experiments.

### 5.5.3 Environmental Setting: Spark Architecture, Hardware and Software

Apache Spark is installed across the nodes in the NeCTAR Cloud, a flexible scalable computing infrastructure. One driver node and eight worker nodes are deployed where each of them has NeCTAR Ubuntu 16.04 LTS (Xenial) amd64 as an operating system, 390 GB disk capacity, and 48GB RAM. The memory configuration for the driver node is 30GB out of 48GB leaving 18GB for other processes. For all worker nodes, the memory capacity is set to 48GB out of 48GB, so that the total of the 384GB memory is occupied for the Spark cluster.

### 5.5.4 Results and Discussion

The final [22] numerical results are reported in Table 5.3 by averaging the numerical results across all tasks. It is observed that ScatterNet is better than Scalable PANFIS in terms of accuracy in several datasets. Specifically, it attains performance improvements with around a 1% margin on: Higgs, Hepmass, KDDcup and Poker Hand in the case of large-size-small-number and small-size-large-number. To put it plainly, this exhibits the advantage of the teacher-forcing mechanism which is capable of capturing the temporal dynamic of data streams while incurring low network parameters. These results also demonstrate the effectiveness of the model merging strategy in combining the local model without decreasing the predictive performance. As a further point, ScatterNet generates an acceptable number of base classifiers. This shows that drift detection and the model pruning mechanisms are capable of governing the

Table 5.3: Numerical Results

| Algorithm | Dataset | Average accuracy per batch (%) | Average training time per batch (s) | Average testing time per batch (s) | Average # base classifier per task |
|---|---|---|---|---|---|
| **ScatterNet** Large size small number scenario | Higgs | **63.80** | **34.67** | 90.88 | 3.2 |
| | Hepmass | **83.43** | **34.42** | 62.97 | 2.31 |
| | Susy | 75.35 | **27.08** | 53.38 | 2.48 |
| | RLCPS | 99.65 | **22.34** | 16.80 | 1 |
| | KDDCup | **99.59** | **43.37** | 35.59 | 1 |
| | Poker Hand | **50.13** | **36.61** | 31.33 | 1 |
| **ScatterNet** Small size large number scenario | Higgs | **63.05** | **13.87** | 23.38 | 2.35 |
| | Hepmass | **83.22** | **13.34** | 14.57 | 1.53 |
| | Susy | 74.97 | **11.65** | 18.12 | 2.25 |
| | RLCPS | 99.64 | **9.49** | 7.12 | 1 |
| | KDDCup | **99.29** | **17.29** | 12.39 | 1 |
| | Poker Hand | **50.09** | **18.09** | 11.76 | 1 |
| **Scalable PANFIS** Large size small number scenario | Higgs | 63.29 | 352.5 | 23.90 | 1 |
| | Hepmass | 83.36 | 345.08 | 24.25 | 1 |
| | Susy | **75.67** | 163.38 | 16.68 | 1 |
| | RLCPS | **99.78** | 74.20 | 11.89 | 1 |
| | KDDCup | 99.44 | 662.51 | 32.11 | 1 |
| | Poker Hand | 50.04 | 144.41 | 19.53 | 1 |
| **Scalable PANFIS** Small size large number scenario | Higgs | 61.97 | 114.85 | 8.40 | 1 |
| | Hepmass | 82.87 | 114.29 | 8.68 | 1 |
| | Susy | **75.51** | 53.44 | 6.28 | 1 |
| | RLCPS | **99.70** | 26.17 | 5.04 | 1 |
| | KDDCup | 99.18 | 216.20 | 11.34 | 1 |
| | Poker Hand | 49.96 | 48.27 | 7.11 | 1 |

evolving process to arrive at appropriate network complexity for a given data stream problem.

Separately, it can be observed that ScatterNet execution time (the sum of training time and testing time) is faster than Scalable PANFIS in all cases although Scalable PANFIS applies a single model. This demonstrates ScatterNet's scalability to execute a big dataset via the implementation of the MapReduce [109] paradigm and the Spark framework [23]. Also, the model merging and hidden node pruning mechanisms help to maintain model complexities without compromising the predictive performance. As a further point, the use of hyperplane-shaped activation function (5.2) being free from antecedent components helps to significantly reduce the space and computational complexities compared to the Gaussian activation function. When tested on the Higgs and Hepmass datasets in the case of large-size-small-number, ScatterNet testing times are inferior to Scalable PANFIS. This result is understandable because ScatterNet has more than one base classifier boosting its predictive performance.

Table 5.4: Ablation Study on the Susy Dataset

| Algorithm | Accuracy | Training time | Testing time | # Base classifier |
|---|---|---|---|---|
| ScatterNet large size small number | 75.49 | 29.65 | 60.69 | 2.41 |
| Single Node Configuration | 75.63 | 621.57 | 373.13 | NA |
| ScatterNet Without Merging | 76.02 | 46.84 | 218.33 | 2.45 |

### 5.5.5 Ablation Study

An ablation study of ScatterNet has been outlined in Table 5.4. It covers two scenarios: executing ScatterNet on a worker node and disabling the merging process. From Table 5.4 it is observed that both the training and testing time of ScatterNet is significantly faster than those a single node teacher-forcing network. It confirms that parallelization process helps to speed up the training process of a teacher-forcing networks. Moreover, this benefit is achieved without compromising the predictive accuracy. This claim is supported by the fact that ScatterNet and ScatterNet without the merging process achieved comparable performance when tested on the Susy dataset. This finding is likely due to the local property of ScatterNet. Separately, it is observed that the deactivation of the merging mechanism slows down the training and testing process. This is understandable as all the resulting hidden nodes in the learning process are utilized. Note that ScatterNet distributes a task among eight computation nodes during training and testing. The resulting local models are merged by the model merging method and then it is used to test the incoming data batch.

### 5.5.6 Statistical Testing

The hypothesis testing is carried out to measure how significant is the proposed method in comparison to a benchmark algorithm. In this chapter, we conduct a Wilcoxon signed-ranked test, the same procedure as the one in sub chapter 4.5.2.3. Only two algorithms to compare, ScatterNet and Scalable PANFIS in prequential setting environment. For each algorithm there are 12 observations (vector of accuracies) as a result of learning over six datasets using both large and small batch settings. The two vector of accuracies to be tested are depicted in Table

5.5. The vector v1 reflects the accuracy of ScatterNet or Scalable PANFIS on Higgs dataset, whereas v2 up to v12 are the accuracy of Hepmass on Large size scenario up to Poker Hand in small size scenario.

It can be seen in Table 5.5 that the statistical test results show that ScatterNet performs better in accuracy, having higher $V_{statistic}$ value than Scalable PANFIS (53 against 25), despite the significance difference $p_{Value}$ is still higher than significance level $\alpha_{significance}$.

Table 5.5: Statistical Testing of Two Paired Algorithms using Wilcoxon Signed-Ranked Test

| Algorithm | Vector of accuracies | | | | | | | | | | | | **Statistical Test Results** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | v1 | v2 | v3 | v4 | v5 | v6 | v7 | v8 | v9 | v10 | v11 | v12 | $p_{Value}$ | $V_{statistic}$ |
| ScatterNet | 63.80 | 83.43 | 75.35 | 99.65 | 99.59 | 50.13 | 63.05 | 83.22 | 74.97 | 99.64 | 99.29 | 50.09 | **53** | **0.3013** |
| Scalable PANFIS | 63.29 | 83.36 | 75.67 | 99.78 | 99.44 | 50.04 | 61.97 | 82.87 | 75.51 | 99.70 | 99.18 | 49.96 | **25** | **0.3013** |

## 5.6 Conclusion

We develop a novel Scalable Teacher-Forcing Network (ScatterNet) to answer the data stream challenges: non-stationary environment, high-speed arrival rate, and high volumes of data generated in the storage. ScatterNet is built upon the teacher-forcing concept possessing short-term memory. As an evolving system, the structure of ScatterNet evolves in reaction to concept drift on both local and global scales. The ScatterNet evaluation scheme uses the prequential test-then-train protocol where both the testing and training phase are executed in parallel to speed up the computation using the Spark distributed processing platform without compromising predictive performance. The issue of network complexity is undertaken by using model pruning in the ensemble network. In addition, data-free model merging compresses the initial distributed models into a merged model without compromising predictive performance. ScatterNet shows a competitive advantage in computational and memory aspects. The results are validated by both numerical results, ablation study, and statistical testing, which confirm that ScatterNet outperform Scalable PANFIS for both accuracy and training time.

# 6 Thesis Conclusions

This chapter provides a summary from this thesis. The results obtained at various stages of the research are summarized as follows:

- All of the methods developed in this thesis are designed to handle vast volumes of data streams, where the evaluation of the methods is undertaken using large-scale public datasets available in high dimensional input space which is a challenging problem.

- In the last two decades, the rise of evolving systems offers the possibility to process streaming data using limited resources (e.g. memory and hard disk capacity). Despite these significant efforts, processing data streams effectively without compromising model performance remains a crucial issue in large-scale data stream processing. A novel large-scale data stream analytics, namely Scalable PANFIS, is developed to solve this issue. The critical feature in Scalable PANFIS lies in its capability to generate a fast-evolving distributed model from large-scale streaming data using a Spark distributed learning platform.

- The distributed learning process produces the initial distributed models, a stack of several local models which also can be used to infer the future data using the combination of inference models (e.g. simple majority voting). However, this heavy structure comes at the cost of a longer testing time. A robust model merging method designed in Scalable PANFIS proves that model merging can solve the structural complexity problem without

compromising the generalization performance of the merged model.

- To further help the effectiveness of the learning process, the active learning (AL) method is implemented in the training process along with PANFIS. The results show that the use of AL does not reduce accuracy.

- Scalable PANFIS is evaluated in relation to four types of structures using the combination of (PANFIS with AL or without AL on the training side and using a merged model or majority voting on the testing side). All of the structures yield similar accuracy in all datasets, but the combination of PANFIS with AL and a merged model achieves the best training time. It can be concluded that our developed method can effectively handle large-scale data streams.

- The use of evolving algorithms helps Scalable PANFIS to identify catch the underlying data stream patterns. As a result, Scalable PANFIS produces better results in comparison to Spark-based distributed algorithms.

- ScatterNet is a large-scale batch incremental learning method implemented under a distributed processing environment. The aims of this framework are two-fold. The incremental learning mechanism ensures the scalability of the algorithm to learn data streams continually without a performance decrease. The distributed feature of the framework enables the algorithm to process large-scale data streams in one task.

- ScatterNet handles drift on both local and global scales. Concept drift on local scale is carried out by the teacher-forcing network, taking into account the temporal characteristic of data streams without using time-intensive iterative gradient calculation. On a global scale, the drift detection strategy is utilized to mark the drift event in data streams.

- ScatterNet adopts the open structure learning mechanism to allow it to adapt to the data stream conditions, where its ensemble network is expanded in reaction to the drift condition by adding the new base model into the network. As a counterbalance, ScatterNet is also equipped with the model pruning strategy to keep network complexity under control by removing the inconsequential base models inside the ensemble network.

- To deal with large-scale data streams in every task, a problem of structural complexity appears. This challenge is handled by a data-free model merging strategy, which is expertly designed to simplify the dense structure, ensuring its model is able to be passed to the next task.

- Incremental learning, in general, solves the problem of continual learning mimicking the human ability to learn an accumulated experience, where both historical and new knowledge are retained to infer future samples. However, in the realm of artificial intelligence, massive data stream processing utilizing distributed processing still receives a scant attention. ScatterNet offers a solution for processing never-ending batches of enormous data streams under a distributed computing platform.

# 7 Future Directions

This chapter provides a possibility of suggestions for future work emanating from this thesis. For scalable evolving system, the extension could be taken from several directions: architecture, data stream, and big data issues, etc. Some future directions are listed as follows:

- Dealing with the real-time system is challenging because data stream can arrive without label due to the costly labelling effort. This problem is known as weakly supervised learning. While some efforts have been made to address this problem, this study in the distributed evolving system is worth to receive a further investigation.
- The problem of imbalance data has long received the attention from the researcher. However, most of the work in high-class of imbalance big data are still using the offline distributed architecture. Finding the right architecture to address this problem could be the next research direction.
- Data quantization has been known as a method to accelerate data processing. This technique is often used to reduce the computation load (e.g. image compression). While this method is possibly applied in offline machine learning, the online quantization method in large-scale data stream would be another possible direction.

# Bibliography


[1] J. Gama, Knowledge Discovery from Data Streams. Chapman & Hall/CRC, 1st ed., 2010.

[2] G. Ditzler, M. Roveri, C. Alippi, and R. Polikar, "Learning in nonstationary environments: A survey," IEEE Computational Intelligence Magazine, vol. 10, no. 4, pp. 12–25, 2015.

[3] J. Chen, K. Li, Z. Tang, K. Bilal, S. Yu, C. Weng, and K. Li, "A parallel random forest algorithm for big data in a spark cloud computing environment," IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 4, pp. 919–933, 2017.

[4] Y. Demchenko, P. Grosso, C. De Laat, and P. Membrey, "Addressing big data issues in scientific data infrastructure," in 2013 International Conference on Collaboration Technologies and Systems (CTS), pp. 48–55, IEEE, 2013.

[5] D. Brzeziński, "Block-based and online ensembles for concept-drifting data streams," 2015.

[6] G. Krempl, I. Žliobaite, D. Brzeziński, E. Hüllermeier, M. Last, V. Lemaire, T. Noack, A. Shaker, S. Sievi, M. Spiliopoulou, et al., "Open challenges for data stream mining research," ACM SIGKDD explorations newsletter, vol. 16, no. 1, pp. 1–10, 2014.

[7] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," ACM computing surveys (CSUR), vol. 46, no. 4, pp. 1–37, 2014.

[8] A. Bifet and G. D. F. Morales, "Big data stream learning with samoa," in Data Mining Workshop (ICDMW), 2014 IEEE International Conference on, pp. 1199–1202, IEEE, 2014.

[9] E. Lughofer, "Evolving fuzzy systems—fundamentals, reliability, interpretability, useability, applications," in HANDBOOK ON COMPUTATIONAL INTELLIGENCE: Volume 1: Fuzzy Logic, Systems, Artificial Neural Networks, and Learning Systems, pp. 67–135, World Scientific, 2016.

[10] X. Wu, X. Zhu, G.-Q. Wu, and W. Ding, "Data mining with big data," IEEE transactions on knowledge and data engineering, vol. 26, no. 1, pp. 97–107, 2014.

[11] M. Pratama, S. G. Anavatti, P. P. Angelov, and E. Lughofer, "Panfis: A novel incremental learning machine," IEEE Transactions on Neural Networks and Learning Systems, vol. 25, no. 1, pp. 55–68, 2014.

[12] M. Pratama, S. G. Anavatti, and E. Lughofer, "Genefis: toward an effective localist network," IEEE Transactions on Fuzzy Systems, vol. 22, no. 3, pp. 547–562, 2014.

[13] W. N. Street and Y.-S. Kim, "A streaming ensemble algorithm (sea) for large-scale classification," in Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '01, (New York, NY, USA), pp. 377–382, ACM, 2001.

[14] D. Sahoo, Q. D. Pham, J. Lu, and S. C. Hoi, "Online deep learning: Learning deep neural networks on the fly," arXiv preprint arXiv:1711.03705, vol. abs/1711.03705, 2017.

[15] A. Ashfahani and M. Pratama, "Autonomous deep learning: Continual learning approach for dynamic environments," in In SIAM International Conference on Data Mining, 2019.

[16] I. Škrjanc, J. Iglesias, A. Sanchis, D. Leite, E. Lughofer, and F. Gomide, "Evolving fuzzy and neuro-fuzzy approaches in clustering, regression, identification, and classification: A survey," Information Sciences, 2019.

[17] H. M. Gomes, J. P. Barddal, F. Enembreck, and A. Bifet, "A survey on ensemble learning for data stream classification," ACM Computing Surveys (CSUR), vol. 50, no. 2, pp. 1–36, 2017.

[18] C. Za'in, M. Pratama, E. Lughofer, and S. G. Anavatti, "Evolving type-2 web news mining," Applied Soft Computing, vol. 54, pp. 200–220, 2017.

[19] C. Za'in, M. Pratama, M. Prasad, D. Puthal, C. P. Lim, and M. Seera, "Motor fault detection and diagnosis based on a meta-cognitive random vector functional link network," in Fault Diagnosis of Hybrid Dynamic and Complex Systems, pp. 15–44, Springer, 2018.

[20] S. del Rio, V. Lopez, J. M. Benítez, and F. Herrera, "A mapreduce approach to address big data classification problems based on the fusion of linguistic fuzzy rules," International Journal of Computational Intelligence Systems, vol. 8, no. 3, pp. 422–437, 2015.

[21] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," Communications of the ACM, vol. 51, no. 1, pp. 107–113, 2008.

[22] Apache Software Foundation, "Hadoop."

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets.," HotCloud, vol. 10, no. 10-10, p. 95, 2010.

[24] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al., "Mllib: Machine learning in apache spark," The Journal of Machine Learning Research, vol. 17, no. 1, pp. 1235–1241, 2016.

[25] M. Pratama, J. Lu, and G. Zhang, "An incremental interval type-2 neural fuzzy classifier," in Fuzzy Systems (FUZZ-IEEE), 2015 IEEE International Conference on, pp. 1–8, IEEE, 2015.

[26] R. Elwell and R. Polikar, "Incremental learning of concept drift in nonstationary environments," Trans. Neur. Netw., vol. 22, pp. 1517–1531, Oct. 2011.

[27] H. Bouchachia and E. Balaguer-Ballester, "Dela: A dynamic online ensemble learning algorithm.," in ESANN, 2014.

[28] M. Pratama, W. Pedrycz, and E. Lughofer, "Evolving ensemble fuzzy classifier," IEEE Transactions on Fuzzy Systems, pp. 1–1, 2018.

[29] L. L. Minku and X. Yao, "Ddd: A new ensemble approach for dealing with concept drift," IEEE transactions on knowledge and data engineering, vol. 24, no. 4, pp. 619–633, 2012.

[30] B. Krawczyk, L. L. Minku, J. Gama, J. Stefanowski, and M. Woźniak, "Ensemble learning for data stream analysis: A survey," Information Fusion, vol. 37, pp. 132–156, 2017.

[31] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, "Learning under concept drift: A review," IEEE Transactions on Knowledge and Data Engineering, vol. 31, no. 12, pp. 2346–2363, 2018.

[32] M. M. Ferdaus, M. Pratama, S. Anavatti, and M. A. Garratt, "Palm: An incremental construction of hyperplanes for data stream regression," IEEE Transactions on Fuzzy Systems, 2019.

[33] M. McCloskey and N. J. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," in Psychology of learning and motivation, vol. 24, pp. 109–165, Elsevier, 1989.

[34] J. Maillo, S. Ramírez, I. Triguero, and F. Herrera, "knn-is: An iterative spark-based design of the k-nearest neighbors classifier for big data," Knowledge-Based Systems, vol. 117, pp. 3–15, 2017.

[35] P. Gupta, A. Sharma, and R. Jindal, "Scalable machine-learning algorithms for big data analytics: a comprehensive review," Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, vol. 6, no. 6, pp. 194–214, 2016.

[36] C. Doulkeridis and K. NØrvåg, "A survey of large-scale analytical query processing in mapreduce," The VLDB Journal, vol. 23, no. 3, pp. 355–380, 2014.

[37] J.-S. Jang, "Anfis: adaptive-network-based fuzzy inference system," IEEE transactions on systems, man, and cybernetics, vol. 23, no. 3, pp. 665–685, 1993.

[38] R. E. Schapire, Y. Freund, P. Bartlett, W. S. Lee, et al., "Boosting the margin: A new explanation for the effectiveness of voting methods," The annals of statistics, vol. 26, no. 5, pp. 1651–1686, 1998.

[39] L. Breiman, "Bagging predictors," Machine learning, vol. 24, no. 2, pp. 123–140, 1996.

[40] R. J. Tibshirani and B. Efron, "An introduction to the bootstrap," Monographs on statistics and applied probability, vol. 57, pp. 1–436, 1993.

[41] T. Takagi and M. Sugeno, "Fuzzy identification of systems and its applications to modeling and control," IEEE transactions on systems, man, and cybernetics, no. 1, pp. 116–132, 1985.

[42] S. Koenig, M. Likhachev, Y. Liu, and D. Furcy, "Incremental heuristic search in ai," AI Magazine, vol. 25, no. 2, pp. 99–99, 2004.

[43] G.-B. Huang, P. Saratchandran, and N. Sundararajan, "A generalized growing and pruning rbf (ggap-rbf) neural network for function approximation," IEEE Transactions on Neural Networks, vol. 16, no. 1, pp. 57–67, 2005.

[44] E. Lughofer, Evolving fuzzy systems-methodologies, advanced concepts and applications, vol. 53. Springer, 2011.

[45] N. K. Kasabov, Evolving connectionist systems: the knowledge engineering approach. Springer Science & Business Media, 2007.

[46] N. K. Kasabov and Q. Song, "Denfis: dynamic evolving neural-fuzzy inference system and its application for time-series prediction," IEEE Transactions on Fuzzy Systems, vol. 10, no. 2, pp. 144–154, 2002.

[47] P. P. Angelov and D. P. Filev, "An approach to online identification of takagi-sugeno fuzzy models," IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), vol. 34, no. 1, pp. 484–498, 2004.

[48] R. R. Yager, "A model of participatory learning," IEEE Transactions on Systems, Man, and Cybernetics, vol. 20, no. 5, pp. 1229–1234, 1990.

[49] E. Lima, M. Hell, R. Ballini, and F. Gomide, "Evolving fuzzy modeling using participatory learning," Evolving intelligent systems: methodology and applications, pp. 67–86, 2010.

[50] H.-J. Rong, N. Sundararajan, G.-B. Huang, and P. Saratchandran, "Sequential adaptive fuzzy inference system (safis) for nonlinear system identification and prediction," Fuzzy Sets and Systems, vol. 157, no. 9, pp. 1260–1275, 2006.

[51] A. Lemos, W. Caminhas, and F. Gomide, "Multivariable gaussian evolving fuzzy modeling system," IEEE Transactions on Fuzzy Systems, vol. 19, no. 1, pp. 91–104, 2011.

[52] D. Dovžan, V. Logar, and I. Škrjanc, "Implementation of an evolving fuzzy model (efumo) in a monitoring system for a waste-water treatment process," IEEE Transactions on Fuzzy Systems, vol. 23, no. 5, pp. 1761–1776, 2014.

[53] E. Lughofer, C. Cernuda, S. Kindermann, and M. Pratama, "Generalized smart evolving fuzzy systems," Evolving Systems, vol. 6, no. 4, pp. 269–292, 2015.

[54] M. Pratama, G. Zhang, M. J. Er, and S. Anavatti, "An incremental type-2 meta-cognitive extreme learning machine," IEEE transactions on cybernetics, vol. 47, no. 2, pp. 339–353, 2017.

[55] R. D. Baruah, P. Angelov, and J. Andreu, "Simpl_eclass: Simplified potential-free evolving fuzzy rule-based classifiers," in 2011 IEEE International Conference on Systems, Man, and Cybernetics, pp. 2249–2254, IEEE, 2011.

[56] P. Angelov and X. Gu, "Mice: Multi-layer multi-model images classifier ensemble," in 2017 3rd IEEE International Conference on Cybernetics (CYBCONF), pp. 1–8, IEEE, 2017.

[57] L. Cohen, G. Avrahami-Bakish, M. Last, A. Kandel, and O. Kipersztok, "Real-time data mining of non-stationary data streams from sensor networks," Information Fusion, vol. 9, no. 3, pp. 344–353, 2008.

[58] M. Sayed-Mouchaweh and E. Lughofer, Learning in non-stationary environments: methods and applications. Springer Science & Business Media, 2012.

[59] M. Pratama, J. Lu, and G. Zhang, "Evolving type-2 fuzzy classifier," IEEE Transactions on Fuzzy Systems, vol. 24, no. 3, pp. 574–589, 2016.

[60] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, "Data stream mining," A practical approach, 2011.

[61] D. Warneke and O. Kao, "Nephele: efficient parallel data processing in the cloud," in Proceedings of the 2nd workshop on many-task computing on grids and supercomputers, p. 8, ACM, 2009.

[62] J. Shafer, R. Agrawal, and M. Mehta, "Sprint: A scalable parallel classi er for data mining," in Proceedings of the 22nd International Conference on Very Large Data Bases, pp. 544–555, 1996.

[63] D. Luo, C. Ding, and H. Huang, "Parallelization with multiplicative algorithms for big data mining," in IEEE 12th International Conference on Data Mining (ICDM), pp. 489–498, IEEE, 2012.

[64] R. Chen, K. Sivakumar, and H. Kargupta, "Collective mining of bayesian networks from distributed heterogeneous data," Knowledge and Information Systems, vol. 6, no. 2, pp. 164–187, 2004.

[65] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan, "Clustering data streams: Theory and practice," IEEE Transactions on Knowledge and Data Engineering, vol. 15, no. 3, pp. 515–528, 2003.

[66] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for clustering evolving data streams," in Proceedings of the 29th international conference on Very large data bases-Volume 29, pp. 81–92, VLDB Endowment, 2003.

[67] J. Gama, P. P. Rodrigues, and R. Sebastião, "Evaluating algorithms that learn from data streams," in Proceedings of the 2009 ACM symposium on Applied Computing, pp. 1496–1500, ACM, 2009.

[68] X. Gu, P. P. Angelov, G. Gutierrez, J. A. Iglesias, and A. Sanchis, "Parallel computing teda for high frequency streaming data clustering," in INNS Conference on Big Data, pp. 238–253, Springer, 2016.

[69] P. Angelov, X. Gu, D. Kangin, and J. Principe, "Teda: typicality based empirical data analysis," Submitted to Information Sciences, 2016.

[70] H. Ishibuchi, T. Nakashima, and M. Nii, "Classification and modeling with linguistic granules: Advanced information processing," 2004.

[71] S. Ramírez-Gallego, H. Mouriño-Talín, D. Martínez-Rego, V. Bolón-Canedo, J. M. Benítez, A. Alonso-Betanzos, and F. Herrera, "An information theory-based feature selection framework for big data under apache spark," IEEE Transactions on Systems, Man, and Cybernetics: Systems, vol. 48, no. 9, pp. 1441–1453, 2017.

[72] F. Padillo, J. M. Luna, F. Herrera, and S. Ventura, "Mining association rules on big data through mapreduce genetic programming," Integrated Computer-Aided Engineering, vol. 25, no. 1, pp. 31–48, 2018.

[73] B. V. Dasarathy and B. V. Sheela, "A composite classifier system design: Concepts and methodology," Proceedings of the IEEE, vol. 67, no. 5, pp. 708–713, 1979.

[74] L. I. Kuncheva, "Classifier ensembles for changing environments," in International Workshop on Multiple Classifier Systems, pp. 1–15, Springer, 2004.

[75] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, G. E. Hinton, et al., "Adaptive mixtures of local experts.," Neural computation, vol. 3, no. 1, pp. 79–87, 1991.

[76] M. I. Jordan and R. A. Jacobs, "Hierarchical mixtures of experts and the em algorithm," Neural computation, vol. 6, no. 2, pp. 181–214, 1994.

[77] L. Xu, A. Krzyzak, and C. Y. Suen, "Methods of combining multiple classifiers and their applications to handwriting recognition," IEEE transactions on systems, man, and cybernetics, vol. 22, no. 3, pp. 418–435, 1992.

[78] R. Polikar, "Ensemble learning," in Ensemble machine learning, pp. 1–34, Springer, 2012.

[79] T. K. Ho, "The random subspace method for constructing decision forests," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 20, no. 8, pp. 832–844, 1998.

[80] H. Wang, W. Fan, P. S. Yu, and J. Han, "Mining concept-drifting data streams using ensemble classifiers," in Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 226–235, 2003.

[81] W. Fan, F. Chu, H. Wang, and P. S. Yu, "Pruning and dynamic scheduling of cost-sensitive ensembles," in AAAI/IAAI, pp. 146–151, 2002.

[82] J. A. Iglesias, A. Ledezma, and A. Sanchis, "Ensemble method based on individual evolving classifiers," in 2013 IEEE Conference on Evolving and Adaptive Intelligent Systems (EAIS), pp. 56–61, IEEE, 2013.

[83] T. R. Hoens, R. Polikar, and N. V. Chawla, "Learning from streaming data with concept drift and imbalance: an overview," Progress in Artificial Intelligence, vol. 1, no. 1, pp. 89–101, 2012.

[84] M. Pratama, S. G. Anavatti, M. Joo, and E. D. Lughofer, "pclass: an effective classifier for streaming examples," IEEE Transactions on Fuzzy Systems, vol. 23, no. 2, pp. 369–386, 2015.

[85] H. Chen, R. H. Chiang, and V. C. Storey, "Business intelligence and analytics: from big data to big impact," MIS quarterly, pp. 1165–1188, 2012.

[86] A. Fernández, S. del Río, V. López, A. Bawakid, M. J. del Jesus, J. M. Benítez, and F. Herrera, "Big data with cloud computing: an insight on the computing environment, mapreduce, and programming frameworks," Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, vol. 4, no. 5, pp. 380–409, 2014.

[87] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," Information Sciences, vol. 275, pp. 314–347, 2014.

[88] S. Ramírez-Gallego, A. Fernández, S. García, M. Chen, and F. Herrera, "Big data: Tutorial and guidelines on information and process fusion for analytics algorithms with mapreduce," Information Fusion, vol. 42, pp. 51–61, 2018.

[89] H. Hu, Y. Wen, T.-S. Chua, and X. Li, "Toward scalable systems for big data analytics: A technology tutorial," IEEE access, vol. 2, pp. 652–687, 2014.

[90] C.-T. Lin et al., Neural fuzzy systems: a neuro-fuzzy synergism to intelligent systems. Prentice hall PTR, 1996.

[91] D. Deng and N. Kasabov, "On-line pattern analysis by evolving self-organizing maps," Neurocomputing, vol. 51, pp. 87–103, 2003.

[92] E. Lughofer, J.-L. Bouchot, and A. Shaker, "On-line elimination of local redundancies in evolving fuzzy systems," Evolving Systems, vol. 2, no. 3, pp. 165–187, 2011.

[93] B. Wittenmark, "Adaptive dual control methods: An overview," in Adaptive Systems in Control and Signal Processing 1995, pp. 67–72, Elsevier, 1995.

[94] C. Za'in, M. Pratama, E. Lughofer, M. Ferdaus, Q. Cai, and M. Prasad, "Big data analytics based on panfis mapreduce," Procedia Computer Science, vol. 144, pp. 140–152, 2018.

[95] C. Za'in, M. Pratama, A. Ashfahani, E. Pardede, and H. Sheng, "Big data analytic based on scalable panfis for rfid localization," in 2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp. 1687–1692, IEEE, 2018.

[96] J. A. Benediktsson and P. H. Swain, "Consensus theoretic classification methods," IEEE transactions on Systems, Man, and Cybernetics, vol. 22, no. 4, pp. 688–704, 1992.

[97] R. Battiti and A. M. Colla, "Democracy in neural nets: Voting schemes for classification," Neural Networks, vol. 7, no. 4, pp. 691–707, 1994.

[98] H. Ishibuchi, T. Nakashima, and T. Morisawa, "Voting in fuzzy rule-based systems for pattern classification problems," Fuzzy sets and systems, vol. 103, no. 2, pp. 223–238, 1999.

[99] A. Fernández, S. del Río, A. Bawakid, and F. Herrera, "Fuzzy rule based classification systems for big data with mapreduce: granularity analysis," Advances in Data Analysis and Classification, vol. 11, no. 4, pp. 711–730, 2017.

[100] K. Subramanian, S. Suresh, and N. Sundararajan, "A metacognitive neuro-fuzzy inference system (mcfis) for sequential classification problems," IEEE Transactions on Fuzzy Systems, vol. 21, no. 6, pp. 1080–1095, 2013.

[101] R. Savitha, S. Suresh, and H. Kim, "A meta-cognitive learning algorithm for an extreme learning machine classifier," Cognitive Computation, vol. 6, no. 2, pp. 253–263, 2014.

[102] E. Lughofer and O. Buchtala, "Reliable all-pairs evolving fuzzy classifiers," IEEE Transactions on Fuzzy Systems, vol. 21, no. 4, pp. 625–641, 2013.

[103] M. Pratama, J. Lu, S. Anavatti, E. Lughofer, and C.-P. Lim, "An incremental meta-cognitive-based scaffolding fuzzy neural network," Neurocomputing, vol. 171, pp. 89–105, 2016.

[104] I. Zliobaite, A. Bifet, B. Pfahringer, and G. Holmes, "Active learning with drifting streaming data," IEEE transactions on neural networks and learning systems, vol. 25, no. 1, pp. 27–39, 2014.

[105] K. Bache and M. Lichman, "Uci machine learning repository," 2013.

[106] I. Triguero, D. Peralta, J. Bacardit, S. García, and F. Herrera, "Mrpr: a mapreduce solution for prototype reduction in big data classification," neurocomputing, vol. 150, pp. 331–345, 2015.

[107] A. Bifet, G. Holmes, B. Pfahringer, R. Kirkby, and R. Gavaldà, "New ensemble methods for evolving data streams," in Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 139–148, ACM, 2009.

[108] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," Mobile networks and applications, vol. 19, no. 2, pp. 171–209, 2014.

[109] J. Dean and S. Ghemawat, "Mapreduce: a flexible data processing tool," Communications of the ACM, vol. 53, no. 1, pp. 72–77, 2010.

[110] C. Za'in, M. Pratama, and E. Pardede, "Evolving large-scale data stream analytics based on scalable panfis," Knowledge-Based Systems, vol. 166, pp. 186–197, 2019.

[111] J. Maillo, S. García, J. Luengo, F. Herrera, and I. Triguero, "Fast and scalable approaches to accelerate the fuzzy k nearest neighbors classifier for big data," IEEE Transactions on Fuzzy Systems, 2019.

[112] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," nature, vol. 521, no. 7553, p. 436, 2015.

[113] M. Das, M. Pratama, S. Savitri, and Z. Jie, "Muse-rnn: A multilayer self-evolving recurrent neural network for data stream classification," in 19th IEEE International Conference on Data Mining, 08 2019.

[114] M. Pratama, A. Ashfahani, Y.-S. Ong, S. Ramasamy, and E. Lughofer, "Autonomous deep learning: Incremental learning of denoising autoencoder for evolving data streams," arXiv preprint arXiv:1809.09081, vol. abs/1809.09081, 2018.

[115] M. Pratama, C. Za'in, A. Ashfahani, Y. S. Ong, and W. Ding, "Automatic construction of multi-layer perceptron network from streaming examples," in Proceedings of the 28th ACM International CIKM, 2019.

[116] M. Pratama, W. Pedrycz, and G. Webb, "An incremental construction of deep neuro fuzzy system for continual learning of non-stationary data streams," arXiv preprint arXiv:1808.08517, vol. abs/1808.08517, 2018.

[117] I. Frias-Blanco, J. d. Campo-Avila, G. Ramos-Jimenez, R. Morales-Bueno, A. Ortiz-Diaz, and Y. Caballero-Mota, "Online and non-parametric drift detection methods based on hoeffdings bounds," IEEE Transactions on Knowledge and Data Engineering, vol. 27, pp. 810–823, March 2015.

[118] P. P. Chan, X. Zeng, E. C. Tsang, D. S. Yeung, and J. W. Lee, "Neural network ensemble pruning using sensitivity measure in web applications," in 2007 IEEE ICSMC, 2007.

[119] W. Zou, C. Li, and N. Zhang, "At–s fuzzy model identification approach based on a modified inter type-2 frcm algorithm," IEEE Transactions on Fuzzy Systems, vol. 26, no. 3, pp. 1104–1113, 2017.

[120] M. Pratama, A. Ashfahani, and M. A. Hady, "Weakly supervised deep learning approach in streaming environments," 2019.

[121] M. Pratama, M. de Carvalho, R. Xie, E. Lughofer, and J. Lu, "Atl," Proceedings of the 28th ACM International Conference on Information and Knowledge Management - CIKM '19, 2019.

[122] T. Finch, "Incremental calculation of weighted mean and variance," University of Cambridge, vol. 4, no. 11-5, pp. 41–42, 2009.

[123] A. Shaker and E. Lughofer, "Self-adaptive and local strategies for a smooth treatment of drifts in data streams," Evolving Systems, 2014.

[124] G. I. Webb, R. Hyde, H. Cao, H. Nguyen, and F. Petitjean, "Characterizing concept drift," Data Min. Knowl. Discov., vol. 30, no. 4, 2016.

[125] J. a. Gama, R. Fernandes, and R. Rocha, "Decision trees for mining data streams," Intell. Data Anal., vol. 10, pp. 23–45, Jan. 2006.

[126] S. Srinivas and R. V. Babu, "Data-free parameter pruning for deep neural networks.," CoRR, vol. abs/1507.06149, 2015.

[127] P. Baldi, K. Cranmer, T. Faucett, P. Sadowski, and D. Whiteson, "Parameterized machine learning for high-energy physics," arXiv preprint arXiv:1601.07913, 2016.

[128] M. Sariyar, A. Borg, and K. Pommerening, "Controlling false match rates in record linkage using extreme value theory," J. of Biomedical Informatics, vol. 44, pp. 648–654, Aug. 2011.

[129] S. J. Stolfo, W. Fan, W. Lee, A. Prodromidis, and P. K. Chan, "Cost-based modeling for fraud and intrusion detection: Results from the jam project," in In Proceedings of the 2000 DARPA Information Survivability Conference and Exposition, pp. 130–144, IEEE Computer Press, 2000.

[130] D. Dua and C. Graff, "UCI machine learning repository," 2017.