Apprentissage non supervisé de flux de données massives : application aux Big Data d'assurance

Mohammed Ghesmoune

Université de Paris 13

Laboratoire d'Informatique de Paris-Nord (LIPN)

Apprentissage Artificiel et Applications (A3)

Thèse

présentée par

Mohammed Ghesmoune

pour obtenir le grade de

Docteur d'université

Spécialité : Informatique

Apprentissage non supervisé de flux de données massives : Application aux Big Data d'assurance.

---

soutenue publiquement le 25 novembre 2016

devant le jury composé de

Directeur			
Mr	Mustapha Lebbah (HDR)	-	LIPN, Université Paris 13
Co-encadrement			
Mme	Hanane Azzag (HDR)	-	LIPN, Université Paris 13
Rapporteurs			
Mr	Marc Gelgon (Pr)	-	LINA, Polytech Nantes
Mr	Allou Samé (HDR)	-	IFSTTAR, Université Paris-Est
Examinateurs			
Mme	Salima Benbernou (Pr)	-	LIPADE, Université Paris Descartes
Mr	Christophe Cérin (Pr)	-	LIPN, Université Paris 13
Mr	Marcin Detyniecki (HDR)	-	AXA Data Innovation Lab et UPMC
Mme	Céline Rouveirol (Pr)	-	LIPN, Université Paris 13

Abstract

The research outlined in this thesis concerns the development of approaches based on growing neural gas (GNG) for clustering of data streams. We propose three algorithmic extensions of the GNG approaches: *sequential*, *distributed and parallel*, and *hierarchical*; as well as a model for scalability using MapReduce and its application to learn clusters from the real insurance Big Data in the form of a data stream.

We firstly propose the **G-Stream** method. G-Stream, as a "sequential" clustering method, is a one-pass data stream clustering algorithm that allows us to discover clusters of arbitrary shapes without any assumptions on the number of clusters. G-Stream uses an exponential fading function to reduce the impact of old data whose relevance diminishes over time. The links between the nodes are also weighted. A reservoir is used to hold temporarily the distant observations in order to reduce the movements of the nearest nodes to the observations.

The **batchStream** algorithm is a micro-batch based method for clustering data streams which defines a new cost function taking into account that subsets of observations arrive in discrete batches. The minimization of this function, which leads to a topological clustering, is carried out using dynamic clusters in two steps: an assignment step which assigns each observation to a cluster, followed by an optimization step which computes the prototype for each node.

A scalable model using **MapReduce** is then proposed. It consists of decomposing the data stream clustering problem into the elementary functions, Map and Reduce. The observations received in each sub-dataset (within a time interval) are processed through deterministic parallel operations (Map and Reduce) to produce the intermediate states or the final clusters.

The batchStream algorithm is validated on the insurance Big Data. A predictive and analysis system is proposed by combining the clustering results of batchStream with decision trees. The architecture and these different modules from the computational core of our Big Data project, called ***Square Predict***.

GH-Stream for both visualization and clustering tasks is our third extension. The presented approach uses a hierarchical and topological structure for both of these tasks.

Résumé

Le travail de recherche exposé dans cette thèse concerne le développement d'approches à base de growing neural gas (GNG) pour le clustering de flux de données massives. Nous proposons trois extensions de l'approche GNG : *séquentielle*, *distribuée et parallèle*, et une méthode *hiérarchique*; ainsi qu'une nouvelle modélisation pour le passage à l'échelle en utilisant le paradigme MapReduce et l'application de ce modèle pour le clustering au fil de l'eau du jeu de données d'assurance.

Nous avons d'abord proposé la méthode **G-Stream**. G-Stream, en tant que méthode "séquentielle" de clustering, permet de découvrir de manière incrémentale des clusters de formes arbitraires et **en ne faisant qu'une seule passe sur les données**. G-Stream utilise une fonction d'oubli afin de réduire l'impact des anciennes données dont la pertinence diminue au fil du temps. Les liens entre les nœuds (clusters) sont également pondérés par une fonction exponentielle. Un réservoir de données est aussi utilisé afin de maintenir, de façon temporaire, les observations très éloignées des prototypes courants.

L'algorithme **batchStream** traite les données en micro-batch (fenêtre de données) pour le clustering de flux. Nous avons défini une nouvelle fonction de coût qui tient compte des sous ensembles de données qui arrivent par paquets. La minimisation de la fonction de coût utilise l'algorithme des nuées dynamiques tout en introduisant une pondération qui permet une pénalisation des données anciennes.

Une nouvelle modélisation utilisant le paradigme **MapReduce** est proposée. Cette modélisation a pour objectif de passer à l'échelle. Elle consiste à décomposer le problème de clustering de flux en fonctions élémentaires (*Map* et *Reduce*). Ainsi de traiter chaque sous ensemble de données pour produire soit les clusters intermédiaires ou finaux. Pour l'implémentation de la modélisation proposée, nous avons utilisé la plateforme Spark.

Dans le cadre du projet ***Square Predict***, nous avons validé l'algorithme batchStream sur les données d'assurance. Un modèle prédictif combinant le résultat du clustering avec les arbres de décision est aussi présenté.

L'algorithme **GH-Stream** est notre troisième extension de GNG pour la visualisation et le clustering de flux de données massives. L'approche présentée a la particularité d'utiliser une structure hiérarchique et topologique, qui consiste en plusieurs arbres hiérarchiques représentant des clusters, pour les tâches de clustering et de visualisation.

Acknowledgements

First and foremost I want to thank my supervisors, Mustapha Lebbah and Hanane Azzag. I appreciate all their contributions of time, ideas, and funding to make my Ph.D. experience productive and stimulating. Their encouragement, supervision and support enabled me to grow up as a Ph.D. for independently carrying out research. During my Ph.D. pursuit, they taught me how to do research, gave me suggestions when I met problems, and supported me to attend summer schools as well as international conferences. I benefited a lot from their profound knowledge and rigorous attitude toward scientific research. I wish to thank Dr. Tarn Duong, for his help preparing the final thesis manuscript and for reviewing my papers.

I would like to thank the reviewers of my dissertation, Prof. Marc Gelgon and Dr. Allou Samé, for accepting to review and evaluate this thesis. I am also very thankful for Prof. Salima Benbernou, Prof. Christophe Cérin, Dr. Marcin Detyniecki and Prof. Céline Rouveirol for accepting to be the examiners of my thesis defense.

I would like to thank all members of our A3 team and my colleagues within the LIPN laboratory for sharing the good ambiance during my stay at LIPN. I also wish to thank Nathalie, our team secretary.

Finally, I am deeply thankful to my parents for their endless love and support, to my brothers, sisters, and their families. I wish to thank Asma for her patience, love, and encouragement. I wish to thank all my friends, among others, Adnan, Aïcha, Amine, Antony, Ehab, Gaël, Imad, Inès, Issam, Jérémie, Hanane, Hanen, Hyppolite, Kaïs, Leila, Luc, Marcos, Mohammed, Mohamed-Mehdi, Mouadh, Moufida, Nhat, Nouha, Pegah, Rakia, Tarn, Tugdual, Zayd, Zouheyr.

Contents

List of Figures

List of Tables

Notations

Notation	Description
$\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$	set of n (potentially infinite) data points
$\mathbf{x}_i = (x_i^1, x_i^2, ..., x_i^d)$	d-dimensional data point
t_i	time-stamp of data point $\mathbf{x}_i$
$\mathbf{w}_c$	prototype $\mathbf{w}_c = (w_c^1, w_c^2, ..., w_c^d)$ of node c
δ_c	threshold distance of node c
$error(c)$	local accumulated error variable
$weight(c)$	local weight variable
bmu or bmu_1	best matching unit (the nearest node)
bmu_2	the second nearest node
α_1	winning node (the nearest node) adaptation factor
α_2	winning node, neighbor adaptation factor
β	cycle interval between node insertions
age_{max}	oldest age allowed for an edge
λ_1	decay factor in the fading function
λ_2	strength factor in weighting edges

List of publications

International reviews with reading committee (3 papers)

- Mohammed Ghesmoune, Mustapha Lebbah, and Hanane Azzag. A new growing neural gas for clustering data streams. Neural Networks, 78:36–50, 2016. ISSN 0893-6080. doi: http://dx.doi.org/10.1016/j.neunet.2016.02.003. URL http://www.sciencedirect.com/science/article/pii/S0893608016000289. Special Issue on "Neural Network Learning in Big Data".

- Mohammed Ghesmoune, Mustapha Lebbah, and Hanane Azzag. State-of-the-art on Clustering Data Streams. Big Data Analytics, 2016. http://biomedcentral.spi-global.com/authorproofs/bmcproofs/index.php?id=LUANnOjxAd09172016100002qEBCbirPIE. *Upon invitation*.

- Hanane Azzag, Salima Benbernou, Tarn Duong, Mohammed Ghesmoune, Mustapha Lebbah, and Mourad Ouziri. Big Data: A Story from Collection to Visualization. Submitted to Machine Learning Journal: Special issue on Discovery Science, 2016.

International conferences with reading committee (4 papers)

- Mohammed Ghesmoune, Mustapha Lebbah, and Hanane Azzag. Micro-batching growing neural gas for clustering data streams using spark streaming. In INNS Conference on Big Data 2015, San Francisco, CA, USA, 8-10

August 2015, pages 158–166, 2015b. doi: 10.1016/j.procs.2015.07.290. URL http://dx.doi.org/10.1016/j.procs.2015.07.290.

- Nhat-Quang Doan, Mohammed Ghesmoune, Hanane Azzag, and Mustapha Lebbah. Growing hierarchical trees for data stream clustering and visualization. In 2015 International Joint Conference on Neural Networks, IJCNN 2015, Killarney, Ireland, July 12-17, 2015, pages 1–8, 2015. doi: 10.1109/IJCNN.2015. 7280397. URL http://dx.doi.org/10.1109/IJCNN.2015.7280397.

- Mohammed Ghesmoune, Mustapha Lebbah, and Hanane Azzag. Clustering over data streams based on growing neural gas. In Advances in Knowledge Discovery and Data Mining - 19th Pacific-Asia Conference, PAKDD 2015, Ho Chi Minh City, Vietnam, May 19-22, 2015, Proceedings, Part II, pages 134–145, 2015c. doi: 10.1007/978-3-319-18032-8 11. URL http://dx.doi.org/10.1007/978-3-319-18032-8_11.

- Mohammed Ghesmoune, Hanane Azzag, and Mustapha Lebbah. G-Stream: Growing neural gas over data stream. In Neural Information Processing - 21st International Conference, ICONIP 2014, Kuching, Malaysia, November 3-6, 2014. Proceedings, Part I, pages 207–214, 2014. doi: 10.1007/978-3-319-12637-1 26. URL http://dx.doi.org/10.1007/978-3-319-12637-1_26.

French speaking conferences with reading committee (3 papers)

- Mohammed Ghesmoune, Mustapha Lebbah and Hanane Azzag. G-Stream: une approche incrémentale pour le clustering de flux de données. In SFC 2015, 09-11 Septembre 2015, Nantes.

- Mohammed Ghesmoune, Hanane Azzag and Mustapha Lebbah. Une nouvelle méthode topologique pour le clustering de flux de données. In COSI 2015, Coloque sur l'optimisation et les systèmes d'information, Oran, 01-03 Juin 2015.

- Mohammed Ghesmoune, Mustapha Lebbah, and Hanane Azzag. Clustering topologique pour le flux de données. In 15èmes Journées Francophones Extraction et Gestion des Connaissances, EGC 2015, 27-30 Janvier 2015, Luxembourg, pages 137–142, 2015a. URL http://editions-rnti.fr/?inprocid=1002072.

To my parents. . .

Introduction

Fouille de flux de données

Ce travail de thèse concerne le domaine de l'apprentissage automatique et du Big Data. L'apprentissage automatique est définit comme étant *la capacité d'un ordinateur à apprendre sans avoir été explicitement programmé.* Plus précisément, l'apprentissage automatique vise à acquérir de la connaissance à partir des données. Ce domaine est apparenté à la fouille de données qui vise également à extraire des modèles à partir des données; bien que les deux domaines possèdent plusieurs technologies et critères commun, la fouille de données reste profondément liée aux technologies de base de données [Han et al., 2011, Zhang, 2010].

Les méthodes d'apprentissage automatique s'organisent en trois types : méthodes *supervisées, semi-supervisées, et non supervisées* [Han et al., 2011].

- *L'apprentissage supervisé* est traditionnellement synonyme de la classification. La supervision dans l'apprentissage vient du fait que les observations de la base d'apprentissage sont étiquetées.
- *L'apprentissage non supervisé* est essentiellement synonyme de clustering. Le processus d'apprentissage est non supervisé puisque les observations d'entrée ne sont pas étiquetées.
- *L'apprentissage semi-supervisé* est une classe de techniques qui font usage des deux types d'observations, étiquetées et non étiquetées, lors de l'apprentissage d'un modèle.

Les contributions présentées par la suite concernent l'apprentissage automatique et la fouille de données pour la fouille de flux de données dans le contexte

du Big Data. Un *flux de données* est une séquence, potentiellement infinie, non-stationnaire de données arrivant en continu où l'accès aléatoire aux données n'est pas possible et le stockage de toutes les données arrivant est infaisable. La fouille de flux de données a récemment fait l'objet de nombreuses études, d'une part en raison du nombre important d'applications émergentes qui manipulent des flux et d'autre part, de l'impossibilité d'utiliser directement les méthodes traditionnelles.

Le clustering consiste à partitionner un ensemble d'observations en sous-ensembles appelés clusters, tels que les observations affectées dans le même cluster soient "similaires" et les observations inter-clusters soient "dissimilaires". Les mesures de similarité ou de dissimilarité sont évaluées en fonction des valeurs des attributs décrivant les observations et impliquent souvent des mesures de distance [Han et al., 2011].

Cependant, le clustering de flux de données nécessite un processus capable de partitionner des observations de façon continue avec des restrictions au niveau de la mémoire et du temps. C'est pourquoi la fouille de flux de données est plus difficile que dans le cas traditionnel. Le travail de recherche exposé dans cette thèse concerne le développement de méthodes d'apprentissage basées sur l'approche 'Growing Neural Gas' pour la classification non-supervisée et la visualisation hiérarchique des structures topologiques dans les flux de données massives. L'objectif principal est de développer des méthodes de clustering de flux de données qui passent à l'échelle en utilisant les concepts et les technologies du Big Data.

Big Data et le projet Square Predict

Le "Big Data" est devenu un nouveau terme omniprésent. Le Big Data concerne la science, l'ingénierie, la médecine, la santé, la finance, le commerce et, finalement, notre société elle-même. En effet, les entreprises comptent de plus en plus sur le Big Data pour découvrir des corrélations et des tendances dans les données qui auraient auparavant restées cachées, et d'utiliser ensuite cette nouvelle information afin d'augmenter la qualité de leurs activités commerciales.

Actuellement, le domaine du Big Data peut être caractérisé par les 5 V : le Volume, la Vélocité, la Variété, la Valeur et la Véracité (voir le chapitre 2 pour plus de détails).

Construire des modèles d'apprentissage automatique à partir de grande masse de données (Big Data) est devenu un défi important et nécessite le développement

de nouveaux types d'algorithmes. La plupart des algorithmes d'apprentissage automatique ne passent pas à l'échelle.

Le projet *Square Predict* est un projet mené avec plusieurs partenaires dont un géant de l'assurance (AXA), trois laboratoires de recherche (LIPN, LIPADE et EISTI), deux éditeurs et un expert en protection de la vie privée, a pour objectif de bàtir une plateforme permettant aux assureurs de réaliser des prédictions en "temps réel" à partir des données de leurs assurés croisées avec celles disponibles sur Internet (réseaux sociaux, open data, etc.).

L'enjeu prévu dans le projet Square Predict est de développer une plateforme Big Data clé en main dédiée aux producteurs d'assurance afin de leur permettre de croiser, fusionner et d'exploiter l'avalanche de données locales et externes. Cette plateforme a pour objectif de fournir des modèles d'apprentissage statistique et outils pour une meilleure personnalisation des produits permettant aux assureurs de réaliser des visualisation et prédictions à partir des données de leurs assurés croisées avec celles disponibles sur le web.

Contributions

Comme nous l'avons déjà mentionné, le présent travail porte sur le développement et la modélisation d'algorithmes pour des données à grande échelle dans un flux de données, en utilisant des méthodes d'apprentissage automatique, en particulier l'approche Growing Neural Gas, et les concepts et technologies du Big Data.

Le chapitre 2 est consacré à l'introduction de l'écosystème Big Data et les principes fondamentaux de la science des données. Le chapitre 3 présente une vue d'ensemble des méthodes de clustering et celles qui peuvent passer à l'échelle et qui sont implémentées avec le paradigme MapReduce. Le chapitre 4 présente une étude approfondie de l'état de l'art des algorithmes de clustering de flux de données. Nous présentons nos principales contributions dans les chapitres suivants:

1. Dans le chapitre 5 nous présentons une première extension de l'approche GNG, appelée **G-Stream**, pour le clustering de flux de données évolutives, **ne faisant qu'une seule passe sur les données**. G-Stream, en tant que méthode "séquentielle" de clustering, permet de découvrir de manière incrémentale des clusters de formes arbitraires. Dans G-Stream, une fonction exponentielle (fonction d'oubli, appelée aussi fonction de fading) est utilisée

afin de réduire l'impact des anciennes données dont la pertinence diminue au fil du temps. Les liens entre les nœuds sont également pondérés par une fonction exponentielle. Un réservoir est aussi utilisé afin de maintenir, de façon temporaire, les données très éloignées des prototypes courants. La qualité de la méthode proposée est évaluée à la fois visuellement et en termes d'indices de performance sur des données synthétiques et réelles.

2. Dans le chapitre 6, une deuxième extension est présentée. Cette dernière consiste en une reformalisation de la méthode des nuées dynamiques en se basant sur le modèle de traitement en *micro-batch*. Nous développons une méthode traitant les données en micro-batch pour le clustering de flux de données et qui passe à l'échelle, appelée **batchStream**. Plus précisément, nous définissons une nouvelle fonction de coût tenant compte des sous ensembles de données qui arrivent par paquets. Ensuite, nous proposons la minimisation de la fonction de coût en utilisant l'algorithme des nuées dynamiques tout en introduisant une pondération qui permet une pénalisation des données anciennes. Cette minimisation est réalisée en deux étapes : une étape d'affectation de chaque observation $\mathbf{x}_i$ au cluster c en utilisant une fonction d'affectation, suivie d'une phase d'optimisation qui consiste à mettre à jour le prototype de chaque cluster c. L'algorithme batchStream, tout comme G-Stream, intègre plusieurs fonctions telle que la fonction d'oubli, la pondération des liens et la création de plusieurs noeuds à chaque interval de temps.

3. Notre troisième contribution propose une modélisation pour le passage à l'échelle en utilisant le paradigme MapReduce. Dans cette modélisation, on décompose le problème de clustering de flux de données en fonctions élémentaires, à savoir les deux fonctions *Map* et *Reduce* du paradigme MapReduce. Les données reçues dans chaque sous ensemble de données (spécifié par un intervalle de temps) sont traitées via des opérations parallèles déterministes (*Map* et *Reduce*) pour produire soit les résultats du programme (les clusters finaux) ou des états intermédiaires. Pour l'implémentation de la modélisation proposée, nous avons utilisé l'écosystème Spark, comme environnement Big Data de traitement distribué de grandes masses de données.

4. Le chapitre 7, qui a un caractère applicatif, aborde le problème d'estimation en temps réel de l'impact des dégàts causés par un événement climatique de grande ampleur, en combinant notre méthode non-supervisée à un modèle

supervisé. Le passage à l'échelle de la méthode citée ainsi que l'utilité de l'algorithme batchStream en tant que méthode d'apprentissage non-supervisé sont démontrés sur un grand jeu de données d'assurance fourni par AXA. L'architecture ainsi que les différents modules de la plateforme Big Data proposée dans le cadre du projet "Square Predict" sont aussi présentés.

5. Enfin, une troisième extension pour la visualisation et le clustering de flux de données massives est décrite dans le chapitre 8. L'approche que nous développons, qui se nomme **GH-Stream** (Growing Hierarchical Trees over Data Stream) est une variante de G-Stream en incluant à ce dernier une composante hiérarchique. Cette dernière consiste en plusieurs arbres hiérarchiques représentant des clusters qui permettent de décrire l'évolution des flux de données, et ensuite d'analyser explicitement leur similitude. Cette structure adaptative peut être exploitée en descendant d'un niveau topologique à n'importe quel niveau de la hiérarchie.

Chapter 1

Introduction

1.1 Mining data streams

The present work pertains to the fields of Machine Learning (ML) and Big Data. ML is defined as *the science of getting computers to act without being explicitly programmed*. Specifically ML aims at acquiring knowledge from data. The sister domain of Data Mining (DM) likewise aims at extracting patterns from data; while both domains have many core technologies and criteria in common, they mostly differ as DM is deeply related to the database technologies [Han et al., 2011, Zhang, 2010].

Machine Learning methods traditionally fall into three categories: *supervised*, *semi-supervised*, and *unsupervised* methods [Han et al., 2011].

- *Supervised learning* is basically a synonym for classification. The supervision in the learning comes from the labeled observations in the training data set.
- *Unsupervised learning* is essentially a synonym for clustering. The learning process is unsupervised since the input observations are not class labeled.
- *Semi-supervised learning* is a class of machine learning techniques that make use of both labeled and unlabeled observations when learning a model.

The presented contributions are concerned with ML and DM for streaming data in the Big Data context. A *data stream* is a sequence of potentially infinite,

non-stationary data arriving continuously (which requires a single pass through the data) where random access to the data is not feasible and storing all the arriving data is impractical. Mining data streams is motivated by key large-scale applications such as network intrusion detection, transaction streams, phone records, web click-streams, social streams, weather monitoring, etc.

Clustering is the process of partitioning a set of observations into multiple groups or *clusters* so that observations within a cluster have high similarity, but are very dissimilar to observations in other clusters. Dissimilarities and similarities are assessed based on the attribute values describing the observations and often involve distance measures [Han et al., 2011].

However, when applying data mining techniques, and specifically clustering algorithms, to data streams, restrictions in execution time and memory have to be considered carefully. This is why mining data streams is more challenging than the traditional case. The research work discussed in this thesis concerns the development of learning methods based on the Growing Neural Gas approach for unsupervised learning (clustering) and hierarchical visualization of topological structures in data streams. The main objective is to develop scalable data stream clustering methods using Big Data concepts and technologies.

1.2 Big Data and *Square Predict* project

In recent years, "Big Data" has become a new ubiquitous term. Big Data is transforming science, engineering, medicine, healthcare, finance, business, and ultimately our society itself. Currently, the Big Data domain can be characterized by the 5 V's: Volume, Velocity, Variety, Value and Veracity (see chapter 2 for more details).

Learning from Big Data has become a significant challenge and requires development of new types of algorithms. Most machine learning algorithms can not easily scale up to Big Data. MapReduce is a simplified programming model for processing large datasets in a distributed and parallel manner.

Organisations are increasingly relying on Big Data to provide the opportunities to discover correlations and patterns in data that would have previously remained hidden, and to subsequently use this new information to increase the quality of their business activities.

The *Square Predict* project gathers 3 public research labs and 4 private companies including AXA Data Innovation Lab. This project aims to provide the insurance industry a platform for real-time predictive analytics that can analyze the information published on social networks coupled with the information available in Open Data, e.g. to assess the rapidly the severity of a natural disaster and its impact on housing insurance payouts.

The *Square Predict* platform is designed in a modular way from the initial data collection to the final visualization, passing by the data fusion, and the analysis and clustering tasks.

1.3 Our contributions

As already mentioned, the present work is concerned with the modelling of large-scale data within a data streaming framework, using Machine Learning methods, specifically the Growing Neural Gas approach, and Big Data concepts and technologies.

Chapter 2 is devoted to introducing the Big Data ecosystem and the fundamentals for data science. Chapter 3 surveys clustering and scalable clustering methods implemented with MapReduce. Chapter 4 presents a thorough survey of the state-of-the-art for a wide range of data stream clustering algorithms. In the subsequent chapters are our main contributions, summarized as follows:

1. In chapter 5 we present a first extension of the GNG approach to deal with streaming data, called **G-Stream**, which is a **one-pass streaming clustering algorithm**. G-Stream, as a "sequential" clustering method, allows us to discover clusters of arbitrary shapes without any assumptions on the number of clusters. In G-Stream, an exponential fading function is used to reduce the impact of old data whose relevance diminishes over time. For the same reason, links between nodes are also weighted by an exponential function. A reservoir is used to hold temporarily the distant observations in order to avoid needless movements of the nearest nodes to observations. The quality of the proposed method is evaluated visually and in terms of various performance criteria on synthetic and real-world datasets.

2. In chapter 6, a second extension is presented, which consists of a novel re-formalization of the dynamic clusters "nuées dynamiques". We develop

a micro-batch method for scalable clustering data streams, named **batchStream**. Specifically, we define a new cost function taking into account the subsets of observations arriving in batches. The minimization of this function, which leads to the topological clustering, is made using dynamic clusters in two steps: an assignment step which assigns each observation $\mathbf{x}_i$ to one cluster c using the assignment function, followed by an optimisation step which computes the prototype for each cell c. We introduce a fading function which "penalizes" the old data since that they are less relevant compared to the recent data. The batchStream algorithm incorporates, like G-Stream, several characteristics e.g the exponential fading function, the time weighting of the edges, and the creation of more than one node in each interval.

3. After that, we present our third contribution consisting of proposing a model for scalability using MapReduce. This model consists of decomposing the data stream clustering problem into the elementary functions, Map and Reduce. The received data-points in each sub data set (specified by a time interval) are processed through deterministic parallel operations (Map and Reduce) to produce either the algorithm output (the final clusters) or the intermediate states. Its implementation is assured in the Spark Streaming platform.

4. In chapter 7, we present an application of our batchStream algorithm on the insurance Big Data provided by AXA. The latter scalable approach in 3 is demonstrated and validated on the insurance Big Data while the utility of the batchStream algorithm in 2 as an example of unsupervised learning. Afterwards, a predictive and analysis system is proposed by combining the clustering result with decision trees. The architecture and the different modules of the proposed Big Data framework are presented as part of the Big Data project, named *Square Predict*.

5. Last but not least, a third extension for both visualization and clustering tasks is described in chapter 8. The presented approach, named **GH-Stream**, uses a hierarchical and topological structure for both clustering and visualization. The topological network is represented by a graph in which each node represents a set of similar data points and neighbor nodes are connected by edges. The hierarchical component consists of a multiple tree-like hierarchy of clusters which allow us to describe the evolution of a data

stream, and then to analyze explicitly their similarity. The main idea for this algorithm compared to the others is in addition to using the fading function and present the data window by window, this algorithm allows to enhance the assignment time since we can move the data by packet (sub-tree).

The different assessments carried out in this thesis (performance measurements and visualizations) obtained promising results.

The thesis manuscript is organized as follows. Chapter 2 gives an introduction to the Big Data ecosystem and discusses the fundamentals that a data scientist needs in order to extract knowledge or insights from large data in various forms, with a focus on the data stream use case. Chapter 3 reviews and discusses the state-of-the-art related to both clustering and scalable clustering methods implemented with MapReduce. Chapter 4 presents a thorough survey of the state-of-the-art for a wide range of data stream clustering algorithms. Chapter 5 presents our G-Stream algorithm concerned with extending the GNG approach to deal with streaming data; experimental validation on benchmark datasets from the clustering literature is reported and discussed. Chapter 6 introduces the batchStream algorithm designed for large-scale data stream clustering. Chapter 7 presents the *Square Predict* project and describes the validation results of batchStream on the insurance Big Data. Chapter 8 finally describes our hierarchical and topological structure for both clustering and visualization tasks. Some conclusions and perspectives for further research are presented in chapter 9.

Chapter 2

Fundamentals of Big Data

This chapter gives an introduction to the Big Data ecosystem. Indeed, we will review and discuss the fundamentals that a data scientist needs in order to extract knowledge or insights from large data in various forms, with a focus on the data stream use case.

2.1 Big Data

To our knowledge, the term "Big Data" appeared for first time in 1998 in a Silicon Graphics (SGI) slide deck by John Mashey with the title of "Big Data and the Next Wave of InfraStress" [Mashey, 1998]. It is a term used to identify the datasets that due to their large size and complexity, we can not manage them with our current methodologies or data mining software tools [Fan and Bifet, 2013]. Despite that the "Big Data" has become a new buzz-word, there is no consistent definition for Big Data, or any detailed analysis of this new emerging technology. Most discussions until now have been going in the blogosphere where active contributors have generally converged on the most important features and incentives of the Big Data [Demchenko et al., 2013].

The work presented in [Laney, 2001] was the first one to talk about 3 Vs in Big Data management, i.e., Volume (great volume), Velocity (rapid generation), Variety (various modalities), to which were added Value (huge value but very low density) [Gantz and Reinsel, 2011] and Veracity (consistency and trustworthiness) [Demchenko et al., 2013] more recently proposed. Figure 2.1 resumes the 5 Vs of Big Data [Demchenko et al., 2013]:

- Volume: there is more data than ever before, their size continues to increase, but not the percent age of data that our tools can process
- Velocity: data are arriving continuously as streams of data, and we are interested in obtaining useful information from it in real time
- Variety: data type diversity in a given stream (text, video, audio, static image, etc.); also differences in data processability (structured, semi-structured, unstructured data)
- Value: business value that gives an organization a competitive advantage, due to the ability to make decisions based in answering questions that were previously considered beyond reach
- Veracity: it includes two aspects: data consistency (or certainty) as defined by their statistical reliability; and data trustworthiness that includes data origin, collection and processing methods (trusted infrastructure and facilities).

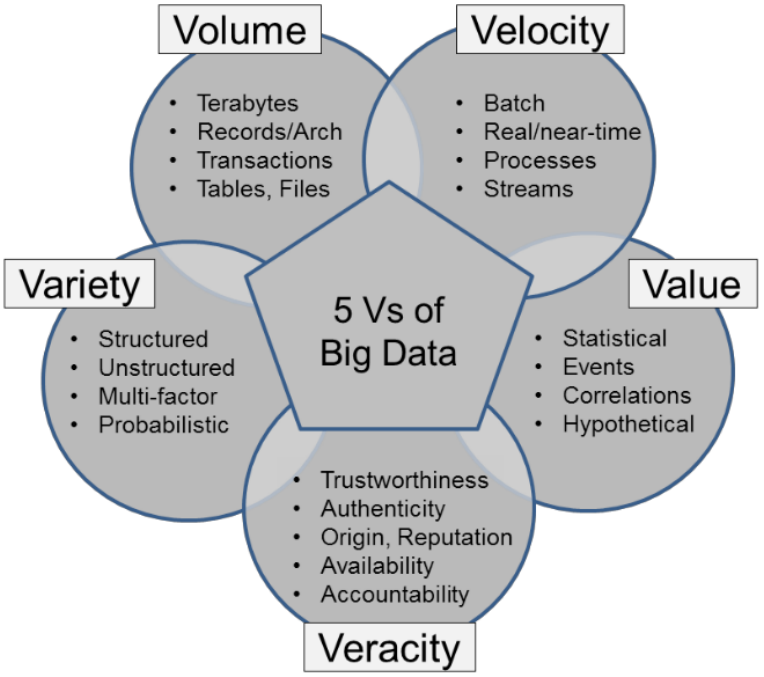


FIGURE 2.1: 5 Vs of Big Data [Demchenko et al., 2013]

We offer an alternative definition: Big Data is also a multidisciplinary area for exchange and collaboration. This definition emphasis the processes involved with Big Data rather than attempting to define intrinsic characteristics like the 5 Vs.

2.2 Distributed data storage systems

2.2.1 Google File System (GFS)

GFS [Ghemawat et al., 2003] uses a simple design with a single *master* server for hosting the entire metadata (the namespace, access control information, the mapping from files to chunks, and the current locations of chunks) and where the data is split into chunks and stored in *chunk-servers*. Files are divided into fixed-size *chunks*. Chunkservers store chunks on local disks and read or write chunk data specified by a chunk handle and byte range. For reliability, each chunk is replicated on multiple chunkservers. However the GFS master is now made fault tolerant using the Chubby [Burrows, 2006] abstraction.

2.2.2 Hadoop Distributed File System (HDFS)

HDFS [Borthakur, 2007] is a distributed file system designed to run on top of the local file systems of the cluster nodes and store extremely large files. HDFS consists of two types of nodes, namely, a namenode called "master" and several datanodes called "slaves". HDFS can also include secondary namenodes. The namenode manages the hierarchy of file systems and director namespace (i.e., metadata). File systems are presented in a form of a namenode that registers attributes, such as access time, modification, permission, and disk space quotas. The file content is split into large blocks, and each block of the file is independently replicated across datanodes for redundancy and to periodically send a report of all existing blocks to the namenode.

HDFS is highly fault tolerant and can scale up from a single server to thousands of machines, each offering local computation and storage. For example, according to Figure 2.2, the record #2 is replicated on nodes A, B, and D. When a process needs this record, it can retrieve it from the node which optimises the response time.

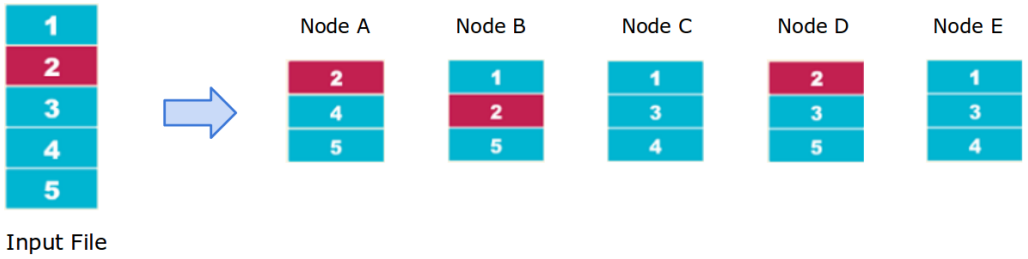


FIGURE 2.2: HDFS Data Distribution

2.3 MapReduce: Basic Concept

MapReduce [Dean and Ghemawat, 2008] is a simplified programming model for processing large numbers of datasets pioneered by Google for data-intensive applications. The MapReduce model was developed based on GFS [Ghemawat et al., 2003] and is adopted through an open-source Hadoop implementation, which was popularized by Yahoo. MapReduce enables programmers who have no experience with distributed systems to write applications that process huge datasets in a large cluster of commodity machines; it manages data partitioning, task scheduling, and nodes failure.
Indeed, MapReduce allows an unexperienced programmer to develop parallel programs and create a program capable of using computers in a cloud. The MapReduce programming model can be explained as follows. The computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: *Map* and *Reduce*.

- *Map*: written by the user, takes an input pair and produces a set of intermediate key/value pairs. For example, given the word count example that it is displayed in Figure 2.3, each mapper takes a line as input and breaks it into words. It then emits a key/value pair of <"word", 1> (<"D", 1>, <"B", 1>, etc.).

- *Shuffle*: is an intermediate step which is done automatically by the system. It starts after finishing the Map step and before the Reduce step. It takes the intermediate data generated by each Map task, sorts this data with intermediate data from other nodes, divides this data into regions to be processed by the reduce tasks.

- *Reduce*: also written by the user, accepts an intermediate key and a set of values for that key. It merges together these values to form a possibly smaller set of values. According to example of Figure 2.3, each reducer sums the counts for each word and emits a single key/value with the word and sum. The final result can be collected into one file that contains each word associated with its frequency (<"A", 2>, <"B", 2>, <"C", 3>, <"D", 2>).

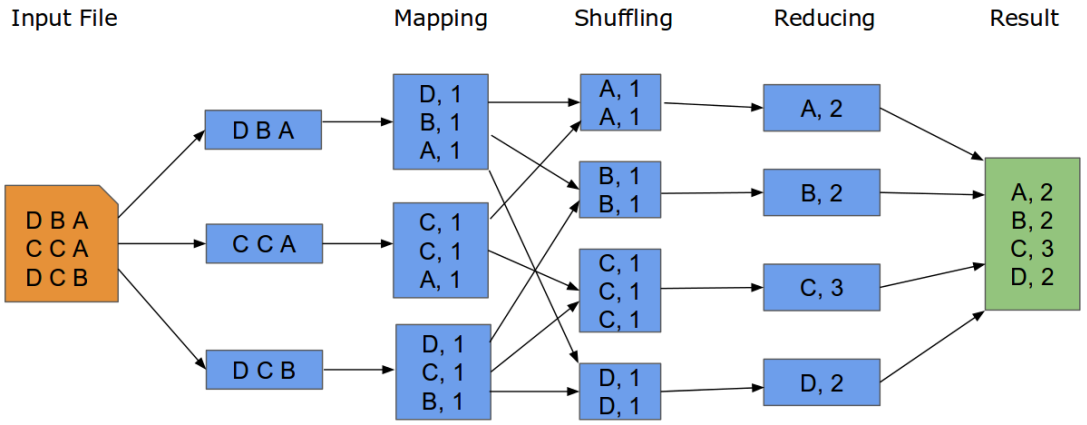


FIGURE 2.3: MapReduce processes for counting the number of occurrences for each word in a document

2.4 Distributed platforms

2.4.1 Hadoop

Apache Hadoop[1] is one of the most well-established software platforms that allow for the distributed processing of large data sets across clusters of computers using simple programming models. It implements the MapReduce paradigm. It is designed to scale up from single servers to thousands of machines, with each offering local computation and storage. Typiquely, the Hadoop framework uses the Hadoop Distributed File System (HDFS) to save large datasets in a distributed manner. Users code their queries and programs using Java. Therefore, the I/O performance of a Hadoop MapReduce job strongly depends on HDFS. Indeed, the HDFS has an non-negligible access time; reads and writes are sufficiently long (because of the use of hard disc to save intermediate data as shown in Figure 2.4)

[1]See http://hadoop.apache.org/

that maching learning processes which generally make many iterations on data are inefficient.

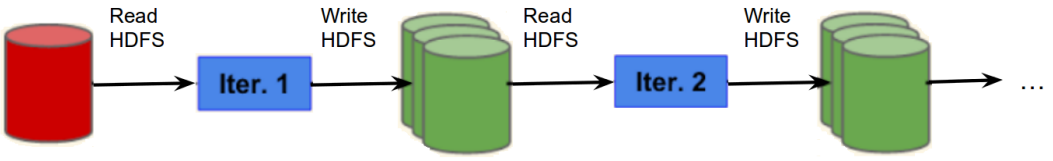


FIGURE 2.4: HDFS reads and writes in iterative machine learning algorithms

In terms of a Hadoop cluster, there are two kinds of nodes in the Hadoop infrastructure: master nodes and worker nodes. The master node takes the input, divides it into smaller sub-problems, and distributes them to worker nodes in the Map step. Afterwards, the master node collects the answers to all the sub-problems and combines them in some way to form the output in the Reduce step.

2.4.2 Spark

Spark is a cluster computing system originally developed by UC Berkeley AMPLab [Zaharia et al., 2012a]. Now it is an umbrellaed project of the Apache Foundation[2]. The main abstraction in Spark is that of a resilient distributed dataset (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations. RDDs achieve fault tolerance through a notion of lineage: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition. Indeed, the elements of an RDD need not exist in physical storage; instead, a handle to an RDD contains enough information to compute the RDD from data in reliable storage. This means that RDDs can always be reconstructed if nodes fail. Although RDDs are not a general shared memory abstraction, they represent a sweet-spot between expressivity on the one hand and scalability and reliability on the other hand, and they are well-suited to a variety of applications [Zaharia et al., 2010].

Spark is intended to be easy to use where users can write their applications quickly in Java, Scala, Python, or R. Figure 2.5 shows that starting from the second iteration, the intermediate data are saved in-memory (RAM) where Spark

[2]See `http://spark.apache.org/`

can retrieve them by accessing the RAM rather than the hard disc, which makes the execution much faster.

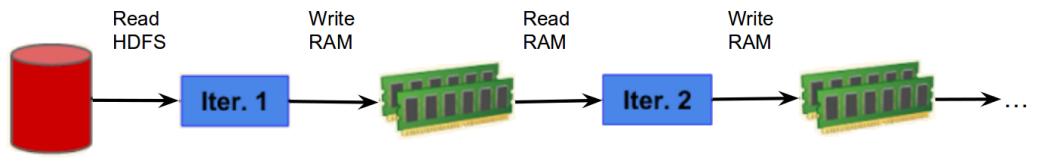


FIGURE 2.5: Iterative machine learning algorithms in Spark

In terms of comparison between Spark and Hadoop, Spark runs programs up to 100× faster than Hadoop MapReduce in memory, or 10× faster on disk, in iterative machine learnig algorithms. As example, Figure 2.6 compares Hadoop and Spark in terms of running time for the k-means and logistic regression algorithms. Another point of comparison between the two frameworks is that Spark offers more choice and flexibility to the programmers because it allows them to write their code in Scala, Java, Python, or R while Hadoop offers only Java.
Given these arguments, it appears that the choice of Spark over Hadoop is obvious.

FIGURE 2.6: Running time of k-means and logistic regression in Hadoop and Spark [Zaharia et al., 2012a]

2.5 Streaming platforms

In today's applications, evolving data streams are ubiquitous. As the need by industry for real time analysis has emerged, the number of systems which support real-time data integration and analytics have increased in recent years. Generally, there exists two types of streaming processing systems: (a) traditional streaming platforms, on which we can implement a streaming algorithm using a *traditional* programming language in a *sequential* manner; (b) Distributed streaming platforms, where the data is distributed across a cluster of machines and the processing model is implemented using the MapReduce framework. This section gives a survey on the most well-known streaming platforms with a focus on the streaming

clustering task. [Liu et al., 2014] gives a general survey on real-time processing systems for Big Data.

2.5.1 Spark Streaming

Spark Streaming [Zaharia et al., 2012b, 2013] is an extension of the Apache Spark [Zaharia et al., 2010] project which adds the ability to perform online processing through a similar functional interface to Spark, such as map, filter, reduce, etc. Spark Streaming runs streaming computations as a series of short batch jobs on RDDs within a programming model called *discretized streams* (D-Streams), as illustrated in Figure 2.7.

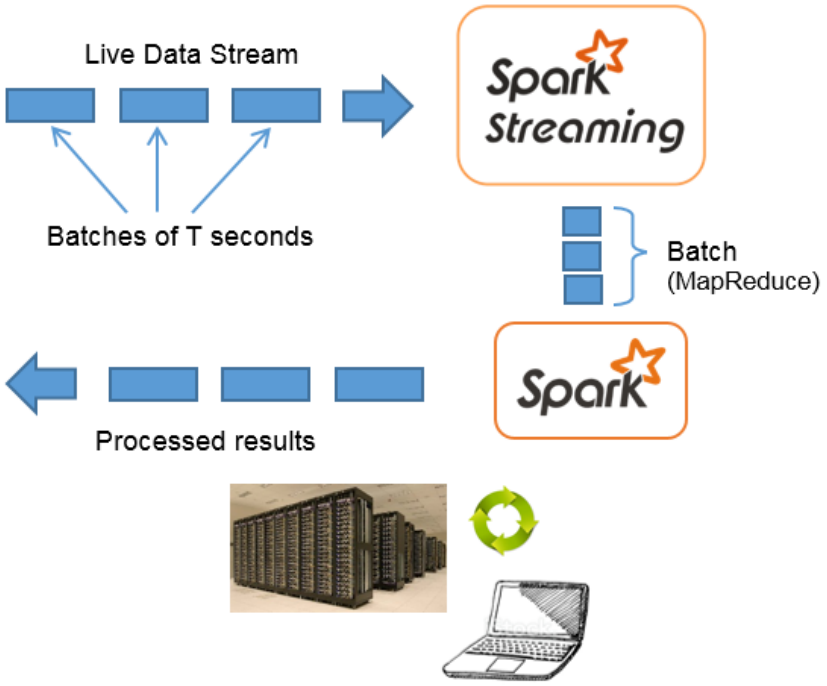


FIGURE 2.7: The internal workflow in Spark Streaming

The key idea behind D-Streams is to treat a streaming computation as a series of deterministic batch computations on small time intervals. For example, we might place the data received each second into a new interval, and run a MapReduce operation on each interval to compute a count. Similarly, we can perform a running count over several intervals by adding the new counts from each interval to the previous result. Spark Streaming can automatically parallelize the jobs across the nodes in a cluster.

Spark Streaming comes with a new approach for fault recovery, while classical

streaming systems update the mutable state on a per-record basis and use either replication or upstream backup for fault recovery. The replication approach creates two or more copies of each process in the data flow graph [Balazinska et al., 2008]. This can double the hardware cost, and if two nodes in the same replica fail, the system is not recoverable. In upstream backup [Hwang et al., 2005], upstream nodes act as backups for their downstream neighbors by preserving tuples in their output queues while their downstream neighbors process them. If a server fails, its upstream nodes replay the logged tuples on a recovery node. The disadvantage of this approach is long recovery times, as the system must wait for the standby node to catch up.

To address these issues, D-Streams employ another approach: *parallel recovery*. The system periodically checkpoints some of the state RDDs, by asynchronously replicating them to other nodes. For example, in a view count program computing hourly windows, the system could checkpoint results every minute. When a node fails, the system detects the missing RDD partitions and launches the tasks to recover them from the latest checkpoint [Zaharia et al., 2013].

In the streaming clustering point of view, Spartakus[3] is an open-source project on top of Spark-notebook[4] which provides front-end packages for some clustering algorithms implemented using the MapReduce framework. This includes our MBG-Stream[5] algorithm [Ghesmoune et al., 2015a] (detailed in chapter 6) with an integrated interface for execution and visualization checks. MLlib [Meng et al., 2015] gives implementations of some clustering algorithms, especially a Streaming k-means[6] open-source code. Another open source software for mining Big Data streams using Spark Streaming is streamDM[7] which is developed at Huawei Noah's Ark Lab. For streaming clustering, it includes Clustream [Aggarwal et al., 2003] and StreamKM++ [Ackermann et al., 2012].

2.5.2 Flink

Flink[8] is an open source platform for distributed stream and batch data processing. The core of Flink is a streaming iterative data flow engine. On top of the engine,

[3] See https://hub.docker.com/r/spartakus/coliseum/
[4] See http://spark-notebook.io/
[5] See https://github.com/mghesmoune/spark-streaming-clustering
[6] See http://spark.apache.org/docs/latest/mllib-clustering.html#streaming-k-means
[7] See http://huawei-noah.github.io/streamDM/
[8] See https://flink.apache.org/

Flink exposes two language-embedded fluent APIs: the DataSet API for processing batch data sources and the DataStream API for processing event streams. The key idea behind Flink is the optimistic recovery mechanism that does not checkpoint every state [Schelter et al., 2013]. Therefore, it provides optimal failure-free performance and simultaneously uses less resources in the cluster than traditional approaches. Instead of restoring such a state from a previously written checkpoint and restarting the execution, a user-defined, algorithm-specific *compensation* function is applied. In case of a failure, this function restores a consistent algorithm state and allows the system to continue the execution.

2.5.3 Massive On-line Analysis (MOA)

MOA[9] (Massive On-line Analysis) is a framework for data stream mining [Bifet et al., 2010]. It includes tools for evaluation and a collection of machine learning algorithms. Related to the WEKA project[10] (Waikato Environment for Knowledge Analysis), it is also written in Java, while scaling to more demanding problems. The goal of MOA is a benchmark framework for running experiments in the data stream mining context by proving storable settings for data streams (real and synthetic) for repeatable experiments, a set of existing algorithms and measures from the literature for comparison, and an easily extendable framework for new streams, algorithms and evaluation methods. MOA currently supports stream classification, stream clustering, outlier detection, change detection and concept drift and recommender systems. In the streaming case, MOA contains several stream clustering methods including: StreamKM++ [Ackermann et al., 2012], CluStream [Aggarwal et al., 2003], ClusTree [Kranen et al., 2011], DenStream [Cao et al., 2006], D-Stream [Chen and Tu, 2007].

2.5.4 Scalable Advanced Massive Online Analysis (SAMOA)

SAMOA[11] (Scalable Advanced Massive Online Analysis) is distributed streaming machine learning (ML) framework that contains a programing abstraction for distributed streaming ML algorithms. It is a project started at Yahoo Labs Barcelona. SAMOA is both a framework and a library [Morales and Bifet, 2015].

[9] See `http://moa.cms.waikato.ac.nz/`
[10] See `http://weka.wikispaces.com/`
[11] See `http://samoa-project.net/`

As a framework, it allows algorithm developers to abstract from the underlying execution engine, and therefore reuse their code on different engines. It features a pluggable architecture that allows it to run on several distributed stream processing engines such as Storm[12], S4[13], and Samza[14]. As a library, SAMOA contains implementations of state-of-the-art algorithms for distributed machine learning on streams. For streaming clustering, it includes an algorithm based on CluStream [Aggarwal et al., 2003].

2.6 Conclusion

Big Data are becoming a new technology focus both in data science and in industry. The domain of Big Data gathers all the techniques and algorithms recently proposed where conventional methods fail when applied to data, as well as accompanying platforms and technologies. Indeed, it has been necessary to reconsider platforms, programming languages, and paradigms usually used for statistical learning. Handling large volumes of data and developing scalable models using computing power (clusters and clouds) was not possible with the usually existing platforms. These new platforms are usually deployed after extensive development on traditional platforms e.g. Matlab, R, etc., as they remain attractive for rapid scientific prototyping.

In the next chapter, we will review and discusse the state of the art related to clustering methods as well as the detail of some scalable clustering methods implemented with MapReduce.

[12]See http://storm.apache.org
[13]See http://incubator.apache.org/s4
[14]See http://samza.incubator.apache.org

Chapter 3

Clustering and Scalable Algorithms

The first part of this chapter reviews and discusses the state of the art related to clustering methods. In the second part, we detail some scalable clustering methods implemented with MapReduce, allowing the reader to have a clear idea on how to scale any data clustering algorithm using the MapReduce paradigm.

There are too many clustering algorithms to cover comprehensively here so we will focus on the algorithms which we have utilised ourselves or those which appear to be most relevant to our work.

3.1 Introduction

Clustering is a key data mining task. This is the problem of partitioning a set of observations into clusters such that observations assigned in the same cluster are similar (or close) and the inter-cluster observations are dissimilar (or distant). The other objective of clustering is to quantify the data by replacing a group of observations (cluster) with one representative observation (prototype).

This chapter reviews and discusses the state of the art related to clustering methods. Even if we do not propose an exhaustive survey, we argue that we present in detail the most well-known data clustering algorithms as we cited in chapter 4. Furthermore, we present an understandable section on how to scale traditional clustering algorithms using the MapReduce paradigm.

We assume that a set of n data-points $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$ are given, where

$\mathbf{x}_i = (x_i^1, x_i^2, ..., x_i^d)$ is a vector in the $\mathbb{R}^d$ space. We denote by $\mathcal{C}$ the set of clusters produced by the clustering task. Each cluster has a prototype variable, denoted by $\mathbf{w}_c = (w_c^1, w_c^2, ..., w_c^d)$, which represents the position of the cluster in $\mathbb{R}^d$.

3.2 Data clustering algorithms

3.2.1 *k*-means

The most common example of clustering algorithms is k-means [Jain and Dubes, 1988]. Clusters are represented by a mean vector called the weighted vector or prototype $\mathbf{w}_j$, where $j = 1, ..., k$, which may not necessarily be a physical point in the data space. Thus we can re-define the clustering problem as an optimization problem: find the cluster centers such that the intra-class variance is minimized, i.e., the sum of squared distances from each object within a cluster to its corresponding prototype. k-means finds k classes from a set of n observations, by minimizing the following cost function:

$$\mathcal{R}_{k-means}(\phi, \mathcal{W}) = \sum_{i=1}^{n} \sum_{j=1}^{k} \|\mathbf{x}_i - \mathbf{w}_j\|^2 \tag{3.1}$$

The method used for the minimization of the function $\mathcal{R}_{k-means}(\phi, \mathcal{W})$ is an iterative method whose basic iteration has two phases:

- **Assignment step**: it is, in this phase, to minimize the function $\mathcal{R}_{k-means}(\phi, \mathcal{W})$ with respect to the assignment function ϕ assuming that the prototype vectors $\mathcal{W}$ are constant; The minimization is achieved by assigning each observation $\mathbf{x}_i$ to the referent $\mathbf{w}_c$ using the assignment function ϕ:

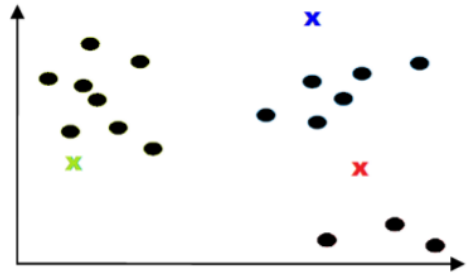
$$\phi(\mathbf{x}_i) = \arg \min_{j=1,...,k} \|\mathbf{x}_i - \mathbf{w}_j\|^2 \tag{3.2}$$

  assign data points to the nearest prototype (best match unit). This assures that the cost function $\mathcal{R}(\phi, \mathcal{W})$ is minimized with respect to the assignment function ϕ assuming that the prototype vectors are constant. Additionally, this step maps data to the network nodes.

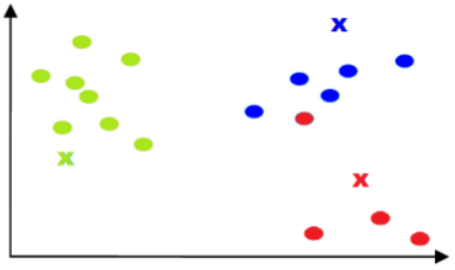
- **Minimization step**: the second phase of iteration decreases again $\mathcal{R}_{k-means}(\phi, \mathcal{W})$ according to the set of referents $\mathcal{W}$. It is assumed in this case that ϕ is fixed at the current value. The referents $\mathbf{w}_c$ are calculated using the following equation:

$$\mathbf{w}_c = \frac{1}{n_c} \sum_{i=1}^{n_c} \mathbf{x}_i \tag{3.3}$$

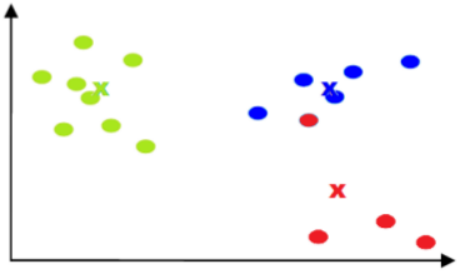
The k-means algorithm is summarised in Figure 3.1 and written in Algorithm 1.



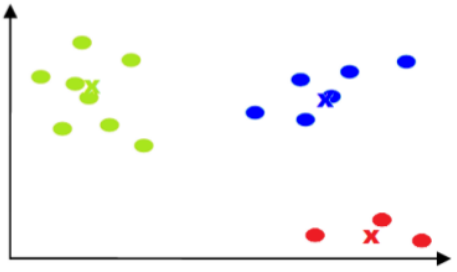
(a) Initial step: Data distrubition is given and $k = 3$ initial prototypes are randomly generated within the data space.



(b) Assignment step: The clusters are formed by assigning every single object to the cluster whose prototype is the nearest to this object.



(c) Update step: The prototypes are updated and move to the local minimum and to cover data distrubition.



(d) The algorithm repeats until convergence or a stopping criterion has been fulfilled.

FIGURE 3.1: Clustering with k-means

k-means is a heuristic algorithm which has few shortcomings: it can converge to a local optimum, and the result depends on k and the initial prototypes. Therefore, some variants have been developped including topological models for example: SOM and Neural Gas that suffer less from these problems, because of the topological preservation.

Algorithm 1: k-means

```
Initialize randomly k prototypes;
repeat
    for i = 1 to n do
        Find the nearest cluster c_j to x_i according to Equation (3.2);
        Assign x_i to cluster c_j;
    for j = 1 to k do
        Update the prototype w_j according to Equation (3.3).
until stopping criterion has been fulfilled;
```

3.2.2 *k*-means++

The k-means algorithm is a simple, fast, and well-known algorithm. However, its performance is sensitive to the initialisation step of the k clusters. [Arthur and Vassilvitskii, 2007] proposed a specific way of choosing the initial prototypes for the k-means algorithm. Let $dist(\mathbf{x})$ denote the shortest distance from a data point $\mathbf{x}$ to the closest prototype that we have already chosen. The k-means++ method is described in Algorithm 2.

The main idea in the k-means++ algorithm is to choose the prototypes one by one in a controlled fashion, where the current set of chosen prototypes will stochastically bias the choice of the next prototype. The central drawback of the k-means++ initialization from a scalability point of view is its inherent sequential nature: the choice of the next prototype depends on the current set of prototypes [Bahmani et al., 2012].

Algorithm 2: k-means++

1. Take one prototype $\mathbf{w}_{c_1}$, chosen uniformly at random from the set of data, $\mathcal{X}$;
2. Take a new prototype $\mathbf{w}_{c_i}$, choosing $\mathbf{x} \in \mathcal{X}$ with probability $\frac{dist(\mathbf{x})^2}{\sum_{\mathbf{x}\in\mathcal{X}} dist(\mathbf{x})^2}$;
3. Repeat Step 2 until we have taken k clusters altogether;
4. Proceed as with the standard k-means method (Algorithm 1);

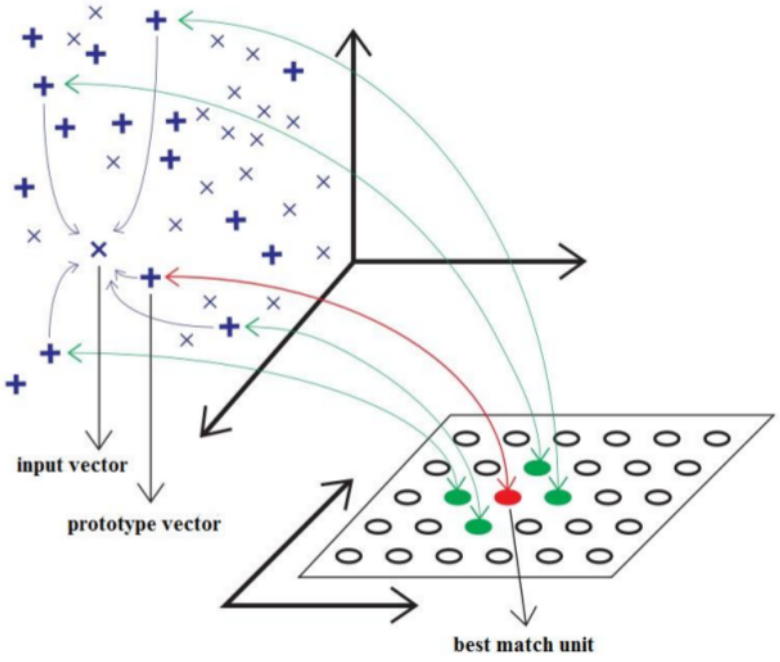
3.2.3 Self-Organizing Map (SOM)

The SOM algorithm, proposed by Kohonen [Kaski et al., 1998], is a type of artificial neural network for unsupervised learning. SOM has the ability of creating spatially organized internal representations of input objects and their abstractions.

As in Figure 3.2, SOM produces a low-dimensional (1D or 2D) discretized representation (called a map or network) from the high-dimensional space of the input data. SOM uses a neighborhood function to preserve the topological properties of the input space, and forms a discretely topological map where similar objects are grouped close together and dissimilar ones apart. Like most artificial neural networks [Haykin, 1998], SOM has a two-fold objective:

1. **Training map**: build the topological map using the input data. A map consists of a number of network nodes arranged according to a structure defined a priori. The usual arrangement of the network nodes is a 1D or 2D, hexagonal or rectangular grid. Associated with each network node is a prototype, $\mathbf{w}_c$, of the same dimension as the input data points.

2. **Mapping (quantization)**: put the input data into a non-linear, discrete map. Vector quantization techniques assign a data point to a prototype such that the distance from this point to the best match prototype is the smallest. This process will respect the neighborhood function to preserve data topology. Data points which are similar into the input space will be put onto neighbor network nodes.



(a)

FIGURE 3.2: SOM principles: mapping and quantization

At the start of the learning, a discrete topological map of size $p \times q = k$ is initialized. We denote $\mathcal{C} = \{c_1, ..., c_k\}$ where c_i $(i = 1, ..., k)$ is a network node. $\mathcal{C}$ is associated with $\mathcal{W} = \{\mathbf{w}_1, .., \mathbf{w}_k\}$ where $\mathbf{w}_i = (w_i^1, ..., w_i^k)$ is the prototype associated with the network node c_i. For each pair of network nodes c_r and c_s in $\mathcal{C}$, their mutual influence is defined by the function $\mathcal{K}^T(\delta(c_r, c_s))$ as in Equation (3.4). A Gaussian function is a common choice for $\mathcal{K}$ that will shrink with time.

$$\mathcal{K}^T(\delta(c_r, c_s)) = e^{\frac{-\delta(c_r, c_s)}{T}} \tag{3.4}$$

where T represents the temperature which decreases the value of T between two values T_{max} and T_{min}, to control the size of the neighborhood influencing a given cell on the map :

$$T = T_{max}\left(\frac{T_{min}}{T_{max}}\right)^{\frac{ith}{N_{iter}-1}} \tag{3.5}$$

N_{iter} is the number of iterations, and $\delta(c_r, c_s)$ is defined as the shortest distance between the two network nodes c_r and c_s.

Due to the use of this function $\mathcal{K}$, in the training the whole neighborhood network nodes move along in the same direction towards the learning data, similar data tend to be put in the adjacent network nodes [Kohonen et al., 2001].

There are mainly two versions of SOM algorithm: stochastic and batch, both aim to minimize the cost function presented in equation 3.6.

$$\mathcal{R}_{SOM}(\phi, \mathcal{W}) = \sum_{i=1}^{n}\sum_{j=1}^{k} \mathcal{K}^T\big(\delta(\phi(\mathbf{x}_i), c_j)\big)\|\mathbf{x}_i - \mathbf{w}_j\|^2 \tag{3.6}$$

where $\phi(\mathbf{x}_i)$ is the assignment function which returns the network node to which $\mathbf{x}_i$ is assigned:

$$\phi(\mathbf{x}_i) = \arg\min_{j=1,...,k} \|\mathbf{x}_i - \mathbf{w}_j\|^2 \tag{3.7}$$

The learning steps are similar to the steps of k-means:

1. **Initialization step**: initialize the map structure, i.e., the number of network nodes (or k clusters), the arrangement shape: hexagonal or rectangular and the initial prototypes.

2. **Assignment step**: assign data points to the nearest prototype (best match unit). This assures that the cost function $\mathcal{R}(\phi, \mathcal{W})$ is minimized with respect to the assignment function ϕ assuming that the prototype vectors are constant. Additionally, this step maps data to the network nodes.

3. **Update step**: re-compute the prototype vectors. The prototypes and their neighbors move along together towards the assigned data such that the map tends to approximate the data distribution. It includes minimizing the cost function $\mathcal{R}(\phi, \mathcal{W})$ with respect to the prototypes vectors assuming that data are all assigned to the best match unit.

Batch SOM

In batch version, the prototypes are updated according to the following equation:

$$\mathbf{w}_c = \frac{\sum_{r=1} \mathcal{K}^T(\delta(c,r)) \sum_{i=1}^{n_r} \mathbf{x}_i}{\sum_{r=1} \mathcal{K}^T(\delta(c,r)) n_r} \tag{3.8}$$

where n_r is the number of data assigned to cluster r. This formula is obtained by fixing ϕ and minimizing $\mathcal{R}$ with respect to $\mathcal{W}$. The assignment function in the batch version is calculated according to the following equation:

$$\phi(\mathbf{x}_i) = \arg \min_{j=1,\ldots,k} \mathcal{K}^T(\delta(\mathbf{x}_i, \mathbf{w}_j)) \|\mathbf{x}_i - \mathbf{w}_j\|^2 \tag{3.9}$$

Algorithm 3: Batch SOM version

```
Initialize k prototypes and W;
while stopping criteria have not been fulfilled do
    for i = 1 → n do
        Find the best match unit to the current selected input data
          according to Equation (3.9);
        c_φ(x_i) = c_φ(x_i) ∪ {x_i} ;          // Put x_i into cluster φ(x_i)
    for j = 1 → k do
        Update prototype vectors according to Equation (3.8);
```

Stochastic SOM

In the stochastic version, each iteration consists of presenting the SOM map a data point randomly selected. The best match unit (the neatest node) as well as

its neighbors move to the input point (see Figure 3.2).

Unlike the batch version, the stochastic version uses the gradient descent method in order to update prototypes:

$$\mathbf{w}_c^t = \mathbf{w}_c^{t-1} - \mu^t \mathcal{K}^T(\delta(c, c_{\phi(\mathbf{x}_i)}))(\mathbf{w}_c^{t-1} - \mathbf{x}_i) \tag{3.10}$$

where μ^t is an adaptation parameter, called "the learning rate" which decreases with time t.

Algorithm 4: Stochastic SOM version

```
1 Initialize k prototypes and 𝒲;
2 while stopping criteria have not been fulfilled do
3     for i = 1 → n do
4         Find the best match unit to the current selected input data
           according to Equation (3.7);
5         forall c_r is a neighbor of φ(x_i) (including φ(x_i) itself) do
6             Update the nodes in the neighborhood of φ(x_i) according to
              Equation (3.10) (including the node φ(x_i) itself) by pulling
              them closer to the input data;
```

3.2.4 Neural Gas

Neural Gas (NG) [Martinetz and Schulten, 1991] is inspired by the SOM. While the SOM map dimensionality must be chosen a priori; depending on the data distribution, the topological network of neural gas may have a different arrangement. Neural Gas is a more flexible network capable of quantizing topological data and learning the similarity among the input data without defining a network topology. Unlike SOM, the adaptation strength in Neural Gas is constant over time and only the best match prototype and its direct topological neighbors are adapted.

Given a network of k clusters $\mathcal{C} = \{c_1, ..., c_k\}$ associated with k prototypes $\mathcal{W} = \{\mathbf{w}_1, ..., \mathbf{w}_k\}$, they are adapted independently of any topological arrangement of the network nodes within the network. Instead, the adaptation step is affected by the topological arrangement within the input space. For each data point $\mathbf{x}_i$ is selected, prototypes will be ajusted by distortions $\mathcal{D}(\mathbf{x}_i, c_j) = \|\mathbf{x}_i - \mathbf{w}_j\|, \forall j = 1, ..., k$. The resulting adaptation rule can be described as a "winner takes most" instead of a "winner takes all" rule [Fritzke, 1991]. The winner network node denoted by

j_0 is determined by the assignment function

$$j_0 = \phi(\mathbf{x}_i) = \arg \min_{j=1,...,k} \|\mathbf{x}_i - \mathbf{w}_j\|^2. \tag{3.11}$$

An edge that connects the network node adjacent, denoted by j_1, to the winner node j_0 which is then stored in a matrix $\mathcal{S}$ representing the neighborhood relationships among the input data:

$$\mathcal{S}_{ij} = \begin{cases} 1 & \text{if a connection exists between } c_i \text{ and } c_j \ (\forall i, j = 1, ..., k, i \neq j) \\ 0 & \text{otherwise} \end{cases}$$

When an observation is selected, the prototypes move toward it by adjusting the distortion $\mathcal{D}(\mathbf{x}_i, c_{j_0})$, controlled by a neighborhood function $\mathcal{K}^T$. In [Fritzke, 1991], this function is fixed, e.g. $\mathcal{K}^T = \exp^{knn_j/T}$ where knn_j is the number of neighborhood network nodes of c_j. This affects directly to the adaptation step for $\mathbf{w}_j$ which is determined by:

$$\mathbf{w}_j^t = \mathbf{w}_j^{t-1} - \varepsilon \mathcal{K}^T(\delta(c_j, c_{\phi(\mathbf{x}_i)}))(\mathbf{x}_i - \mathbf{w}_j) \tag{3.12}$$

To capture the topological relations between the prototypes, each time an observation is presented, the connection between j_0 and j_1 is incremented by one. Each connection is associated with an "age" variable. Only the connection between j_0 and j_1 is reset, the other connections of j_0 age, i.e. their age increment. When the age of a connection exceeds a specific lifetime Max_{age}, it is removed. The way to update the age of the connections is to increase with each incoming input object is learnt. Finally, Neural Gas can be summarized by the Algorithm 5.

In this algorithm, stopping criteria can be either:

- a number of iterations
- a threshold for the quantization error.

3.2.5 Growing Neural Gas

Growing Neural Gas (GNG) [Fritzke, 1994] is an incremental self-organizing approach which belongs to the family of topological maps such as Self-Organizing Maps (SOM) [Kohonen et al., 2001] or Neural Gas (NG) [Martinetz and Schulten,

Algorithm 5: Neural Gas

Initialize k prototypes and set all $\mathcal{S}_{ij}$ to zero;
forall $\mathbf{x}_i \in \mathcal{X}$ **do**
  Determine the sequence $(c_{j_0}, c_{j_1}, ..., c_{j_{n-1}})$ such that

$$\|\mathbf{x}_i - \mathbf{w}_{j_0}\| < \|\mathbf{x}_i - \mathbf{w}_{j_1}\| < ... < \|\mathbf{x}_i - \mathbf{w}_{j_{k-1}}\|$$

  `// ` $\mathbf{w}_{j_0}$ `is the best match prototype, i.e., the nearest prototype;` $\mathbf{w}_{j_1}$ `is the second nearest prototype to` $\mathbf{x}_i$
  forall c_j *with* $\mathcal{S}_{j_0,j} == 1$ **do**
    Perform an adaptation step for the prototypes according to Equation (3.12);
  if $\mathcal{S}_{j_0,j_1} == 0$ **then**
    Create a topological connection between c_{j_0} and c_{j_1}, i.e., $\mathcal{S}_{j_0,j_1} = 1$;
    Set age for this connection, i.e., $age_{j_0,j_1} = 0$;
  forall c_j *with* $\mathcal{S}_{j_0,j} == 1$ **do**
    Increase the age of all connections of j_0 by one, i.e., $age_{j_0,j} = age_{j_0,j} + 1$;
    if $age_{j_0,j} > Max_{age}$ **then**
      Remove all connections of j_0 which exceeded their age, i.e., $\mathcal{S}_{j_0,j} = 0$;

1991]. It is an unsupervised clustering algorithm capable of representing a high dimensional input space in a low dimensional feature map. Typically, it is used for finding topological structures that closely reflect the structure of the input distribution. Therefore, it is used for *visualization tasks* in a number of domains [Martinetz and Schulten, 1991, Beyer and Cimiano, 2012] as neurons (nodes), which represent prototypes, are easy to understand and interpret.

The GNG method has no input parameters which change over time and is able to continue learning, adding network units and connections. As an incremental variant of Neural Gas, GNG inherits its principle; however it does not impose the strict network-topology preservation rule. The network incrementally learns the topological relationships inherent in the dataset, and continues until a stopping criterion is fulfilled. Before learning, only $k = 2$ prototypes are initialized. Step by step, after a certain number of iterations (called epoch), a new network node is successively added into the topological network. But how to add a new network node? Now, this relates to quantization error.

In the clustering problem, the goal is always to minimize the quantization error of datasets or data within the clusters. Therefore, the cluster that provides a high

value of quantization error is not a good one. We should divide this cluster into smaller clusters. GNG finds the two clusters c_1 and c_2 which have the highest quantization error. Then a new node is inserted halfway between these two nodes by the following expression:

$$\mathbf{w}_{new} = \frac{1}{2}(\mathbf{w}_1 + \mathbf{w}_2) \tag{3.13}$$

The node insertion will be repeated until a stoping criterion is fulfilled.

3.2.6 Affinity Propagation

The Affinity Propagation (AP) approach proposes an equivalent formalization of the k-medoids problem, defined in terms of energy minimization. It solves the optimization problem

$$\mathbf{c}^* = \arg\min(E[\mathbf{c}]) \tag{3.14}$$

with

$$E[\mathbf{c}] = -\sum_{i=1}^{n} S(\mathbf{x}_i, \mathbf{w}_{c_i}) - \sum_{i=1}^{k} \ln \chi_i^{(p)}[\mathbf{c}] \tag{3.15}$$

where $\mathbf{c} = (c_1, c_2, ..., c_k)$ is the mapping between the data and prototypes, $S(\mathbf{x}_i, \mathbf{w}_{c_i})$ is the similarity between $\mathbf{x}_i$ and its prototype $\mathbf{x}_{c_i} \in \mathcal{X}$, set to the negative squared distance $-\|\mathbf{x}_i - \mathbf{x}_{c_i}\|^2$ if $i \neq \mathbf{c}_i$. A tuning parameter called the *preference penalty* is the cost incurred for being a self prototype:

$$\sigma = S(\mathbf{x}_i, \mathbf{w}_{c_i}), \qquad \forall i, \tag{3.16}$$

$\chi_i^{(p)}[\mathbf{c}]$ is a set of constraints controlling the clustering structure. The quantity $\ln \chi_i^{(p)}[\mathbf{c}] \to -\infty$ if $\mathbf{w}_{c_i} \neq \mathbf{x}_i$, which implies that if $\mathbf{x}_i$ is selected as an prototype by some items, it has to be its own prototype. Otherwise, $\ln \chi_i^{(p)}[\mathbf{c}] = 0$. The energy function thus enforces a tradeoff between the distortion, i.e., the sum over all items of the squared error, $\|\mathbf{x}_i - \mathbf{w}_{c_i}\|^2$, and the cost of the model, that is $\sigma|\mathbf{c}|$ if $|\mathbf{c}|$ denotes the number of prototypes retained. Equation 3.15 thus does not directly specify the number of prototypes to be found, as opposed to k-medoids. Instead, the number of prototypes in the solution depends on penalty σ; note that $\sigma = 0$ yields a trivial solution, selecting every item as an prototype.

A message passing algorithm is employed to solve the optimization problem defined by equation 3.15, considering two types of messages: availability messages

$a(i,k)$ express the accumulated evidence for $\mathbf{x}_k$ to be selected as the best prototype for $\mathbf{x}_i$; responsibility messages $r(i,k)$ express the fact that $\mathbf{x}_k$ is suitable to be the prototype of $\mathbf{x}_i$ [Zhang et al., 2008, Frey and Dueck, 2007].

3.2.7 DBSCAN

Density-based clustering has the ability to discover arbitrary-shape clusters and to handle noise [Amini et al., 2014]. In density-based clustering methods, clusters are formed based on the dense areas that are separated by sparse areas. DBSCAN (Density-Based Spatial Clustering of Applications with Noise) [Ester et al., 1996] is one of the most well-known density-based clustering algorithms.

DBSCAN is developed for clustering large spatial databases with noise, based on connected regions with high density. The density of each point is defined based on the number of points close to that particular point called the point's neighborhood. The dense neighborhood is defined based on two user-specified parameters: the radius (ε) of the neighborhood (ε-neighborhood), and the number of the objects in the neighborhood ($MinPts$).

The basic definitions in DBSCAN are introduced in the following, where $\mathcal{X}$ is a current set of data points:

- ε-neighborhood of a point: the neighborhood within a radius of ε of a point $\mathbf{x}_p$ is denoted by $N_\varepsilon(\mathbf{x}_p)$:

$$N_\varepsilon(\mathbf{x}_p) = \{\mathbf{x}_q \in \mathcal{X} | dist(\mathbf{x}_p, \mathbf{x}_q) \leqslant \varepsilon\},$$

  where $dist(\mathbf{x}_p, \mathbf{x}_q)$ denotes the Euclidean distance between points $\mathbf{x}_p$ and $\mathbf{x}_q$;

- $MinPts$: the minimum number of points around a data point in the ε-neighborhood;

- core point: a point where the cardinality of its ε-neighborhood is at least $MinPts$ (see Figure 3.3);

- border point: a point is a border point if the cardinality of its ε-neighborhood is less than $MinPts$ and at least one of its ε-neighbors is a core point (see Figure 3.3);

- noise point: a point is a noise point if the cardinality of its ε-neighborhood is less than $MinPts$ and none of its neighbors is a core point (see Figure 3.3);

- directly density reachable: a point $\mathbf{x}_p$ is directly density reachable from point $\mathbf{x}_q$, if $\mathbf{x}_p$ is in the ε-neighborhood of $\mathbf{x}_q$ and $\mathbf{x}_q$ is a core point;
- density reachable: a point $\mathbf{x}_p$ is density reachable from point $\mathbf{x}_q$, if $\mathbf{x}_p$ is in the ε-neighborhood of $\mathbf{x}_q$ and $\mathbf{x}_q$ is not a core point but they are reachable through chains of directly density reachable points;
- density-connected: if two points $\mathbf{x}_p$ and $\mathbf{x}_q$ are density reachable from a core point $\mathbf{x}_o$, $\mathbf{x}_p$ and $\mathbf{x}_q$ are density-connected;
- cluster: a maximal set of density-connected points.

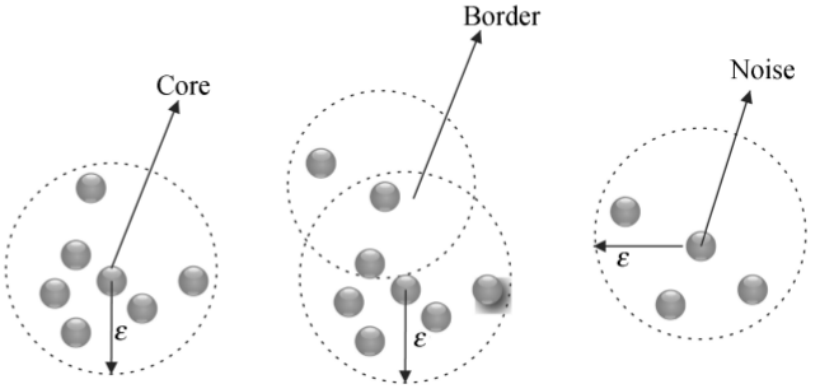


FIGURE 3.3: DBSCAN: core, border, and noise points [Ester et al., 1996].

DBSCAN starts by randomly selecting a point and checking whether the ε-neighborhood of the point contains at least $MinPts$ points. If not, it is considered as a noise point, otherwise it is considered as a core point and a new cluster is created. DBSCAN iteratively adds the data points, which do not belong to any cluster and are directly density reachable from the core points of a new cluster. If the new cluster can no longer be expanded, the new cluster is completed. In order to find the next cluster, DBSCAN randomly selects the unvisited data points and the clustering process continues until all the points are visited and no new point is added to any cluster [Ester et al., 1996, Amini et al., 2014].

3.2.8 EM Algorithm

In the probabilistic approach, data is considered to be a sample independently drawn from a mixture model of several probability distribution [Berkhin, 2006]. Each component of the mixture corresponds to a cluster; additional criteria are

used to automatically determine the number of clusters [Fraley and Raftery, 1998]. The Expectation Maximization (EM) algorithm [Dempster et al., 1977, McLachlan and Krishnan, 2007] is a general approach to maximum likelihood in the presence of incomplete data.

The overall likelihood of the training data is its probability to be drawn from a given mixture model.

$$\mathcal{V}(\mathbf{x}_1, ..., \mathbf{x}_n; \theta) = \prod_{i=1}^{n} \sum_{j=1}^{k} \pi_j \varphi_j(\mathbf{x}_i; \alpha_j) \tag{3.17}$$

with

$\forall j = 1, ..., k, \pi_j \in [0, 1]$ and $\sum_{j=1}^{k} \pi_j = 1$
where:

- $\varphi_j(\mathbf{x}_i; \alpha_j)$ represents the probability density.
- π_j denotes the probability that an element of the sample follows the law φ.
- $\theta = (\pi_1, ..., \pi_k; \alpha_1, ..., \alpha_k)$ represents the unknown parameter of the mixture model.

By introducing the log-likelihood, the Equation (3.17) can be rewritten as follows:

$$\mathcal{L}(\mathbf{x}_1, ..., \mathbf{x}_n; \theta) = \sum_{i=1}^{n} \log \Big(\sum_{j=1}^{k} \pi_j \varphi_j(\mathbf{x}_i; \alpha_j) \Big) \tag{3.18}$$

Log-likelihood serves as an objective function, which gives rise to the EM method. EM is a two-step iterative optimization:

- The Step E estimates probabilities $\varphi_j(\mathbf{x}_i; \alpha_j)$, which is equivalent to a soft reassignment.
- The Step M finds an approximation to a mixture model, given current soft assignments.

This boils down to finding mixture model parameters that maximize log-likelihood. The process continues until log-likelihood convergence is achieved.

Algorithm	Complexity
k-means	$O(kn)$
k-means++	$O(kn)$
SOM	$O(kn \log n)$
NG	$O(kn \log n)$
GNG	$O(kn^2)$
AP	$O(n^2 \log n)$
DBSCAN	$O(n^2)$; by using spatial indices, it decreases to $O(n \log n)$
AntTree	$O(n \log n)$
EM	$O(kn)$

TABLE 3.1: Computational complexity of clustering algorithms

In [Attar et al., 2013, El Attar, 2012], the authors have proposed an estimation of probability distribution over a data set which is distributed into subsets located on the nodes of a distributed system. More precisely, the global distribution is estimated by aggregating local distributions which are modelled as a Gaussian mixture.

3.2.9 Computational complexity

In Table 3.1, we report the computational complexity of the data clustering algorithms presented above, where n is the number of data points and k is the number of network nodes (or clusters).

3.3 Scalable clustering

In this section, we will describe in details the implementation of some of the most well-known and commonly used clustering methods, using the MapReduce paradigm. This will give the reader a clear idea on how to scale any data clustering algorithm in MapReduce.

As described in chapter 2, MapReduce [Dean and Ghemawat, 2008, Lämmel, 2008] is a programming model and an associated implementation for processing and generating large datasets that is amenable to a broad variety of real-world tasks. Users specify the computation in terms of a *map* and a *reduce* function, and the underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks.

3.3.1 General Framework

In contrast to the typical single machine clustering, parallel and distributed (scalable) algorithms use multiple machine to speed up the computation and increase the scalability.

Most parallel and distributed clustering algorithms follow the general framework depicted in Figure 3.4 [Januzaj et al., 2004, Sarazin et al., 2014, Zhao et al., 2009]

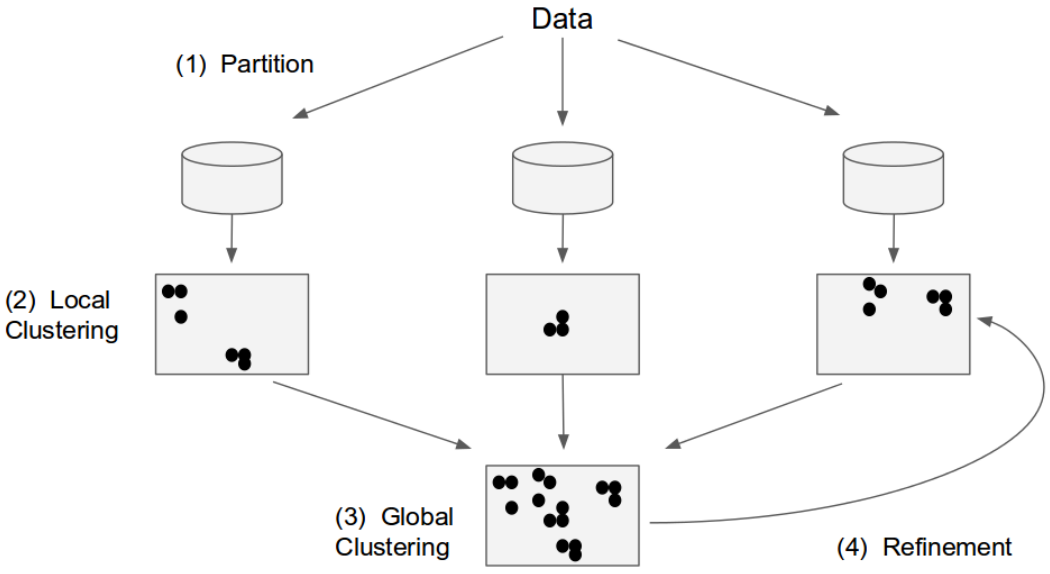


FIGURE 3.4: The general framework of most parallel and distributed clustering algorithms [Aggarwal and Reddy, 2014].

1. **Partition.** Data are partitioned and distributed over machines.
2. **Local Clustering.** Each machine performs local clustering on its partion of the data.
3. **Global Clustering.** The cluster information from the previous step is aggregated globally to produce global clusters.
4. **Refinement of Local Clusters.** Optionally, the global clusters are sent back to each machine to refine the local clusters.

3.3.2 Scalable *k*-means using MapReduce

Zhao et al. [2009] proposed a parallel and distributed implementation of k-means in MapReduce. The proposed algorithm, called PKMeans, is implemented using Hadoop[1] to make the clustering method applicable to large scale data.

Since the most intensive calculation to occur in k-means is the calculation of distances, the idea of PKMeans is to execute in a parallel manner these distance computations between different observations with prototypes. In a nutshell, the map function performs the procedure of assigning each data-point to the closest cluster while the reduce function performs the procedure of updating the new clusters. In order to decrease the cost of network communication, a combiner function is developed to deal with partial combination of the intermediate values with the same key within the same map task.

The input dataset is stored in an HDFS [Shvachko et al., 2010] as a sequence file of $< key, value >$ pairs, each of which represents a record in the dataset. The key is the offset in bytes of this record to the start point of the data file, and the value is a string of the content of this record. The dataset is split and globally broadcast to all mappers. Consequently, the distance computations are executed in parallel. For each map task, PKMeans construct a global variable *clusters* which is an array containing the information about centers of the clusters. Given the information, a mapper can compute the closest cluster for each data-point. The intermediate values are then composed of two parts: the index of the closest cluster and the data-point information [Zhao et al., 2009]. The pseudocode of the map function is shown in Algorithm 6.

In the *combine* function, we partially sum the values of the points assigned to the same cluster. In order to calculate the mean value of the objects for each cluster, we should record the number of data-points in the same cluster in the same map task. This procedure does not consume the communication cost because the intermediate data is stored in local disk of the host. The pseudocode for the combine function is shown in Algorithm 7.

In the *reduce* function, we sum all the data-points and compute the total number of data-points assigned to the same cluster. Therefore, we can obtain the new cluster centers which are used for next iteration. The pseudocode for the reduce function is shown in Algorithm 8.

[1] http://lucene.apache.org/hadoop/

Algorithm 6: map(key, value)

Data: Global variable *clusters*, the offset *key*, the data-point *value*
Result: $< key', value' >$ pair, where the key' is the index of the closest cluster and $value'$ is a string comprise of data-point information

```
1  Construct the data-point instance from value;
2  minDist = Double.MAX_VALUE;
3  index = −1;
4  for each cluster c_i ∈ C do
5      dist = ComputeDistance(instance, c_i);
6      if dist < minDist then
7          minDist = dist;
8          index = i;
9  Take index as key′;
10 Construct value′ as a string comprise of the values of different dimensions;
11 output < key′, value′ > pair;
```

Algorithm 7: combine(key, V)

Data: *key* is the index of the cluster, V is the list of the data-points assigned to the same cluster
Result: $< key', value' >$ pair, where the key' is the index of the cluster and $value'$ is a string comprised of sum of the data-points in the same cluster and the data-point number

```
1  Initialize one array to record the sum of value of each dimensions of the
   data-points contained in the same cluster, i.e. the data-points in the list V;
2  Initialize a counter num as 0 to record the number of data-points in the
   same cluster;
3  for each value v ∈ V do
4      Construct the data-point instance from v;
5      Add the values of different dimensions of instance to the array;
6      num = num + 1;
7  Take key as key′;
8  Construct value′ as a string comprised of the sum values of different
   dimensions and num;
9  output < key′, value′ > pair;
```

Algorithm 8: reduce(key, V)

Data: *key* is the index of the cluster, V is the list of the partial sums from different host

Result: $< key', value' >$ pair, where the key' is the index of the cluster and $value'$ is a string representing a new cluster center

```
Initialize one array record the sum of value of each dimensions of the
  data-points contained in the same cluster, e.g. the data-points in the list
  V;
Initialize a counter NUM as 0 to record the number of data-points in the
  same cluster;
for each value v ∈ V do
    Construct the data-point instance from v;
    Add the values of different dimensions of instance to the array;
    NUM = NUM + num;
Divide the entries of the array by NUM to get the new cluster's coordinates;
Take key as key';
Construct value' as a string comprise the cluster's coordinates;
output < key', value' > pair;
```

3.3.3 Scalable Self-Organizing Map using MapReduce

Sarazin et al. [2014] present two scalable implementations of the SOM-MapReduce algorithm coded in Spark. The pseudocode of the first version is as follows.

1. Randomly initialize the prototypes
2. **Map**: For each data-point $\mathbf{x}_i \in \mathcal{X}$
   (a) Assign $\mathbf{x}_i$ to its nearest cluster using the Euclidean distance
   (b) Compute the numerator and the denominator for each cluster c

$$MapNumerator_c = \mathcal{K}^T(\delta(c, \phi(\mathbf{x}_i)))\mathbf{x}_i$$

$$MapDenominator_c = \mathcal{K}^T(\delta(c, \phi(\mathbf{x_i})))$$

3. **Reduce**: Update the prototypes of all clusters by summing up the output of the Map tasks

$$\mathbf{w}_c = \frac{\sum_c MapNumerator_c}{\sum_c MapDenominator_c} \tag{3.19}$$

In the second version, map outputs are merged in one value, so the key of the output is not used. The Map value of the output is a matrix and a neighborhood

vector. The matrix consists of the rows of data vectors $\mathbf{x}_i$ who are themselves multiplied by the neighborhood factors $\mathcal{K}^T(\delta(c, \phi(\mathbf{x_i})))$ [Sarazin et al., 2014].

3.3.4 Density-based Distributed Clustering (DBDC)

DBDC is a density distributed clustering algorithm. Density-based clustering aims to discover clusters of arbitrary shape. Each cluster has a density of points which is considerably higher than outside of the cluster. Also, the density whithin the areas of noise is lower than the density in many of the clusters [Januzaj et al., 2004].

DBDC is an exemplary algorithm that follows the general framework given in Section 3.3.1. Initially the data is partitioned over machines. At the local clustering step, each machine performs a carefully designed clustering algorithm to output a set of a small number of representatives which has an accurate desciption of local clusters. The local models consist of a set of representatives for each locally found cluster. Each representative is a concrete observation from the observations stored on the local site.

The global model is created by analyzing the local representatives. This analysis is similar to a new clustering of the representatives with suitable global clustering parameters. To each local representative a global cluster-identifier is assigned. In fact, the representatives are merged in the global clustering step using DBSCAN [Ester et al., 1996], a single-machine density-based clustering algorithm. Then the global clusters are sent back to all clients sites which relabel all observations located on their site independently of each other.

The experimental results clearly show the advantage of the distributed clustering. the running time of DBDC is more than 30 times faster than the serial clustering counterpart. Moreover, DBDC yields almost the same clustering quality as the serial algorithm.

3.3.5 Scalable DBSCAN using MapReduce

A recent proposed algorithm is MR-DBSCAN [He et al., 2014] which is a scalable MapReduce-based DBSCAN algorithm. Three major drawbacks are existed in parallel DBSCAN algorithms which MR-DBSCAN is fulfilling [Shirkhorshidi et al., 2014]:

1. They are not successful to balance the load between the parallel nodes
2. These algorithms are limited in scalability because all critical sub procedures are not parallelized
3. Their architecture and design limit them to less portability to emerging parallel processing paradigms.

MR-DBSCAN proposes a novel data partitioning method based on computation cost emission as well as a scalable DBSCAN algorithm in which all critical sub-procedures are fully parallelized. The MR-DBSCAN algorithm consists of three stages: data partitioning, local clustering, and global merging.

The first stage divides the whole dataset into smaller partitions according to spatial proximity. In the second stage, each partition is clustered independently. Then the partial clustering results are aggregated in the last stage to generate the global clusters. Experiments on large datasets confirm the scalability and efficiency of MR-DBSCAN.

3.3.6 Scalable EM using MapReduce

Expectation Maximization (EM) is used to learn the maximum likelihood parameters in the presence of incomplete data.

Many works have been proposed to scale-up the EM algorithm [Das et al., 2007, Cui et al., Basak et al., 2012]. The parallel implementation of EM proposed in [Cui et al.] is coded in Spark.

- Each E-step is a Spark map transformation which runs in parallel mapping each $\mathbf{x}_i$ to a vector of conditional probability densities.
- Each M-step is a reduce action which goes through all the observations in the RDD, aggregating results from E-step.

In their implementation, each iteration consists of two map operations and two reduce operations. In the first map operation, we calculate the responsibility (the log-likelihood, $\mathcal{L}$) of each cluster for each data point, along with the product of the data point and $\mathcal{L}$ and the sum of the products for all clusters. Then we do a reduce operation to calculate the new centers for each cluster. In the las step, we do another map and reduce to calculate the covariance of each cluster.

3.3.7 MapReduce-based Models and Libraries

Due to the interest of the MapReduce framework, some studies have used it for scaling clustering algorithms. As examples, we can cite the implementation of the EM algorithm in MapReduce [Das et al., 2007], the parallel version of the k-means++ initialization algorithm [Bahmani et al., 2012], and the work considered in [Ene et al., 2011] which is a MapReduce implementation of the k-medean problem.

Currently, more and more libraries have emerged offering MapReduce-based implementations of machine learning algorithms:

- **MLlib.**[2] This is Spark's machine learning library. It consists of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as lower-level optimization primitives and higher-level pipeline APIs.
- **Apache Mahout.**[3] Is a project of the Apache Software Foundation to produce free implementations of distributed or otherwise scalable machine learning algorithms focused primarily in the areas of collaborative filtering, clustering and classification. Currently, the supported algebraic platforms are Apache Spark[4] and H20[5], and Apache Flink[6]. Since April 2014, support for Hadoop MapReduce[7] algorithms is being gradually phased out.

3.4 Conclusion

As data clustering has attracted a significant amount of research attention, many clustering algorithms have been proposed in the past decades. However, the engrowing volumes of information made possible by technological advances, makes clustering of very large data a challenging task.

Currently, the MapReduce paradigm has met with a resounding success in this era of Data Science due to, amongst others, its simplicity. The challenge in scaling

[2] http://spark.apache.org/docs/latest/mllib-guide.html
[3] http://mahout.apache.org/
[4] http://spark.apache.org/docs/latest/index.html
[5] http://www.h2o.ai/
[6] http://hadoop.apache.org/
[7] http://hadoop.apache.org/

a data clustering method is not only to use the MapReduce paradigm but also to decompose the problem in small functions, the *map* and *reduce* functions. Usually, scaling an algorithm using MapReduce needs a redefintion of the initial problem.

In the next chapter, we will review and discusse the state of the art related to data stream clustering methods.

Chapter 4

State of the art on Clustering Data Streams

This chapter is devoted to the problem of clustering data in the form of a stream, i.e., a sequence of potentially infinite, non-stationary data (the probability distribution of the unknown data generation process may change over time) arriving continuously (which requires a single pass through the data) where random access to the data is not feasible and storing all the arriving data is impractical.

4.1 Introduction

In today's applications, evolving data streams are ubiquitous. Indeed, examples of applications relevant to streaming data are becoming more numerous and more important, including network intrusion detection, transaction streams, phone records, web click-streams, social streams, weather monitoring, etc.

When applying data mining techniques, and specifically clustering algorithms, to data streams, restrictions in execution time and memory have to be considered carefully. To deal with time and memory restrictions, many of the existing data stream clustering algorithms modify traditional non-streaming methods to use the two-phase framework proposed in [Aggarwal et al., 2003] to deal with streaming data, e.g., DenStream [Cao et al., 2006] is an extension of DBSCAN algorithm, StreamKM++ [Ackermann et al., 2012] of k-means++, StrAP [Zhang et al., 2008] of AP, etc.

General surveys have been recently published in the literature for mining data streams [Aggarwal, 2013, Nguyen et al., 2015, Khalilian and Mustapha, 2010, Yogita and Toshniwal, 2013, Mousavi et al., 2015]. The authors of [de Andrade Silva et al., 2013] introduced a taxonomy to classify data stream clustering algorithms. The work presented in [Amini et al., 2014] is a thorough survey of state-of-the-art density-based clustering algorithms over data streams.

This chapter presents a thorough survey of the state-of-the-art for a wide range of data stream clustering algorithms.

4.2 Fundamental concepts for streaming data

A definition of a data stream has been given in [Golab and Özsu, 2003] as follows: *A data stream is a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items. It is impossible to control the order in which items arrive, nor is it feasible to locally store a stream in its entirety. Likewise, queries over streams run continuously over a period of time and incrementally return new results as new data arrive. These are known as long-running, continuous, standing, and persistent queries.*

As already mentioned, Data Streaming became a hot research topic since the early 2000s not only does it raise challenging scientific issues, it also appears as the only way to handle data sources such as sensor networks, web logs, telecommunications or Web traffic [Gaber et al., 2005].

4.2.1 Window models

In most data stream scenarios, more recent information from the stream can reflect the emerging of new trends or changes in the data distribution. This information can be used to explain the evolution of the process under observation. Systems that give equal importance to outdated and recent data do not capture the evolving characteristics of stream data [de Andrade Silva et al., 2013]. There are three commonly-studied models in data streams [Zhu and Shasha, 2002]: i) sliding windows; ii) damped windows; and iii) landmark windows.

Sliding window model

In the sliding window model, only the most recent information from the data stream are stored in a data structure whose size can be variable or fixed. The observations are manipulated based on the principles of queue processing, where the first observation added to the queue will be the first one to be removed. Figure 4.1 presents an example of the sliding window model.

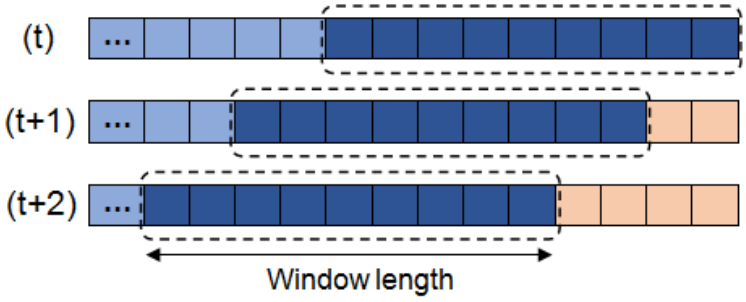


FIGURE 4.1: Sliding window model

Damped window model

Differently from sliding windows, the damped window model, also referred to as the time-fading model, considers the most recent information by associating weights to observations from the data stream. An illustrative example of the damped window model is presented in Figure 4.2, where the weight of the observations exponentially decays from black (most recent) to white (expired).

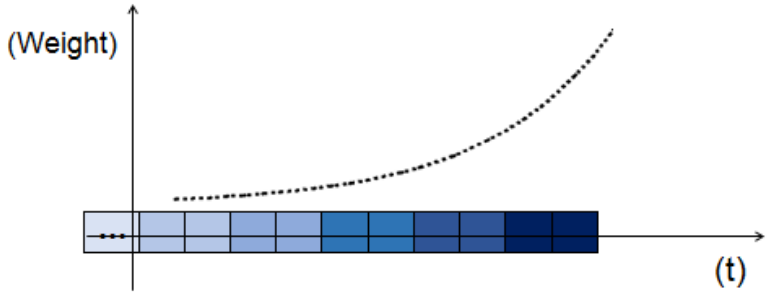


FIGURE 4.2: Damped window model

Landmark window model

Processing a stream based on landmark windows requires handling disjoint portions of the streams (chunks), which are separated by landmarks (relevant observations). Landmarks can be defined either in terms of time, (e.g., on daily or weekly basis) or in terms of the number of elements observed since the previous landmark [Metwally et al., 2005]. All observations that have arrived after the landmark onwards are kept or summarized into a window of recent data.

When a new landmark is reached, all observations kept in the window are removed and the new observations from the current landmark are kept in the window until a new landmark is reached. Figure 4.3 illustrates an example of landmark window.

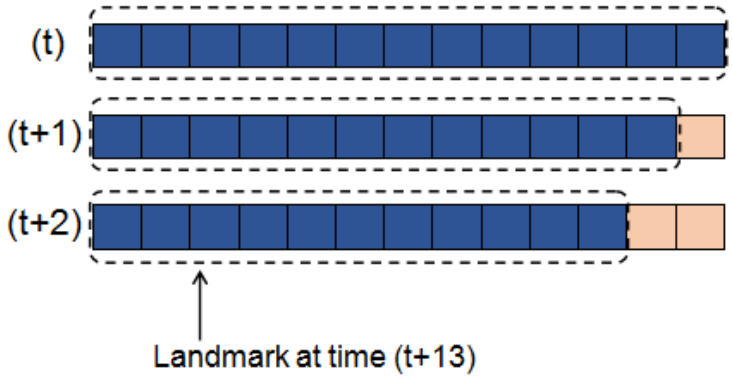


FIGURE 4.3: Landmark window model

4.2.2 Change detection

A key difference between data streaming and online learning, as already mentioned, is the fact that the underlying distribution of the data is not necessarily stationary. This phenomenon, also known as concept drift, means that the concept about which data is obtained may shift from time to time after certain delay.

The problem of data evolution is interesting from two perspectives [Aggarwal, 2007]:

- For a given data stream, we would like to find the significant changes which have occurred in the data stream. The aim of this approach is to provide a

direct understanding of the underlying changes in the stream. Methods such as [Aggarwal, 2003, Kifer et al., 2004] fall in this cathegory.

- The second class of problems relevant to data evolution is that of updating data mining models when a change has occurred. There is a considerable amount of work in the literature with a focus on incremental maintenance of models in the context of evolving data [Donjerkovic et al., 2000, Ganti et al., 2002].

4.3 Data stream clustering methods

This section discusses previous works on data stream clustering problems, and highlights the most relevant algorithms proposed in the literature to deal with this problem. Most of the existing algorithms (e.g. *CluStream* [Aggarwal et al., 2003], *DenStream* [Cao et al., 2006], *StreamKM++* [Ackermann et al., 2012], or *ClusTree* [Kranen et al., 2011]) divide the clustering process in two phases:

1. *Online*, the data will be summarized;
2. *Offline*, the final clusters will be generated.

Figure 4.4 is a flowchart of the data stream clustering algorithms presented in this paper. These algorithms are categorized according to the nature of their underlying clustering approach.

4.3.1 Hierarchical stream methods

A hierarchical clustering method groups the given data into a tree of clusters which is useful for data summarization and visualization. This is a binary-tree based data structure called the *dendrogram*. Once the dendrogram is constructed, one can automatically choose the right number of clusters by splitting the tree at different levels to obtain different clustering solutions for the same dataset without rerunning the clustering algorithm again. Hierarchical clustering can be achieved in two different ways, namely, bottom-up and top-down clustering. Though both of these approaches utilize the concept of dendrogram while clustering the data,

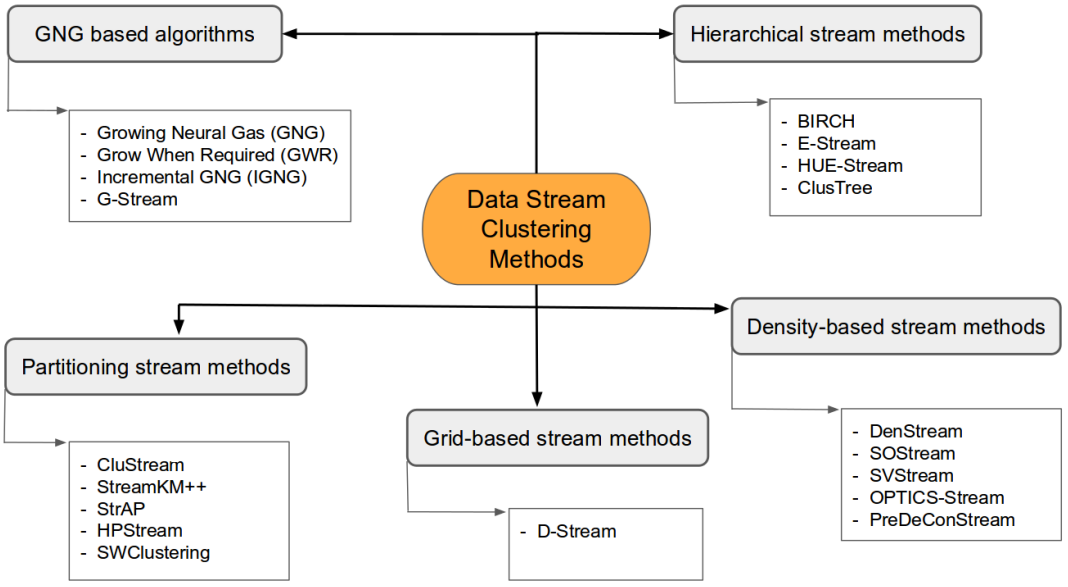


FIGURE 4.4: Data stream clustering methods: the presented algorithms categorized according to the nature of their underlying clustering approach.

they might yield entirely different sets of results depending on the criterion used during the clustering process [Aggarwal and Reddy, 2014].

4.3.1.1 Balanced Iterative Reducing and Clustering using Hierarchies (BIRCH)

BIRCH incrementally and dynamically clusters multi-dimensional data points to try to produce the best quality clustering with the available resources (i.e., memory and time constraints) by making a single scan of the data, and to improve the quality further with a few additional scans. It should be noted that the BIRCH method is not designed for clustering data streams and cannot address the concept drift problem.

The key characteristic of the BIRCH is to introduce a new data structure called a *clustering feature* (CF) as well as a CF-tree. The CF can be regarded as a concise summary of each cluster. This is motivated by the fact that not every data point is equally important for clustering and we cannot afford to keep every data point in the limited main memory. On the other hand, for the purposes of clustering, it is often enough to keep up to the second order of data moment. In other words, CF is not only efficient, but also sufficient to cluster the entire data set [Aggarwal and Reddy, 2014, Zhang et al., 1996].

More precisely, a CF structure is a triple (N, LS, SS), where

- N is the number of data points in the cluster;
- $LS = \sum_{\mathbf{x} \in \mathcal{X}} \mathbf{x}$ is the linear sum of the N data points;
- $SS = \sum_{\mathbf{x} \in \mathcal{X}} \mathbf{x}^2$ is the squared sum of the N data points.

The CF vector has two main properties giving the incremental aspect, in an intuitive way, to any algorithm that uses this structure:

- **Incrementality**

  If a point $\mathbf{x}$ is added to the cluster, the sufficient statistics are updated as follows:

$$N_i \leftarrow N_i + 1; \tag{4.1}$$

$$LS_i \leftarrow LS_i + \mathbf{x}; \tag{4.2}$$

$$SS_i \leftarrow SS_i + \mathbf{x}^2; \tag{4.3}$$

- **Additivity**

  If $CF_1 = (N_1, LS_1, SS_1)$ and $CF_2 = (N_2, LS_2, SS_2)$ are the CF vectors of two disjoint clusters, merging them is equal to the sum of their parts. The additive property allows us to merge sub-clusters incrementally without accessing the original data set.

$$CF_1 + CF_2 = (N_1 + N_2, LS_1 + LS_2, SS_1 + SS_2). \tag{4.4}$$

Figure 4.5 presents the CF-tree structure in BIRCH. The CF-tree is a height-balanced tree which keeps track of the hierarchical clustering structure for the entire data set. BIRCH requires two user defined parameters: B the branch factor or the maximum number of entries in each non-leaf node; and T the maximum diameter (or radius) of any CF in a leaf node. The maximum diameter T defines the examples that can be absorbed by a CF. By increasing T, then more examples can be absorbed by a CF node and smaller CF-Trees are generated. Each node in the CF-tree represents a cluster which is in turn made up of at most B sub-clusters. All the leaf nodes are chained together for the purposes of efficient scanning.

When a data point is available, it traverses down the current tree from the root, until it finds the appropriate leaf following the *closest*-CF path, with respect to the L_1 or L_2 norms. Its insertion in the CF-tree can be performed in a similar

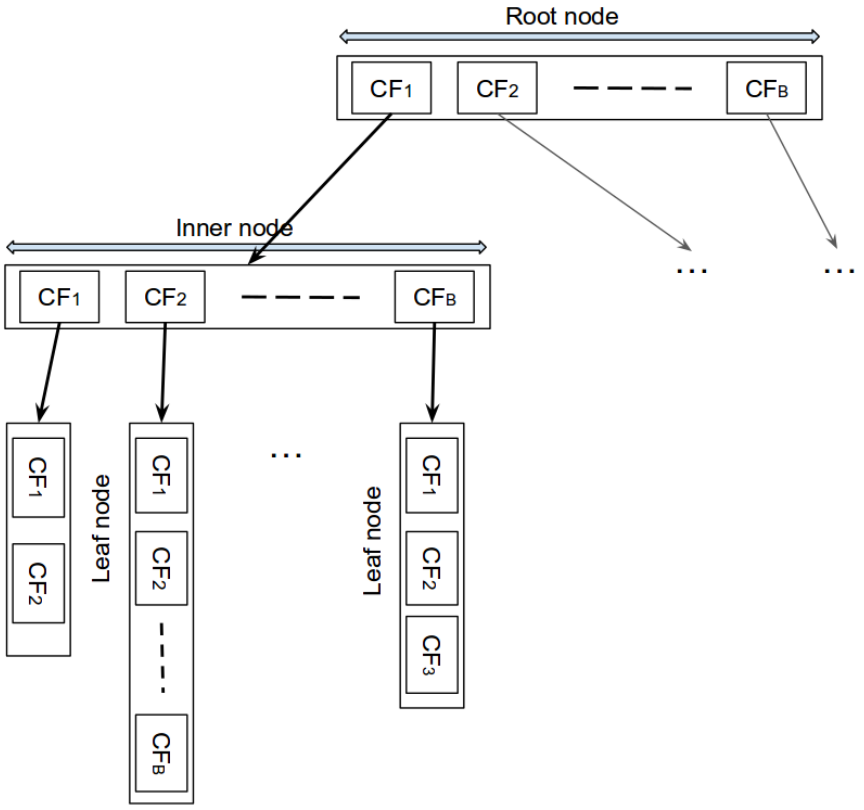


FIGURE 4.5: The Clustering Feature Tree in BIRCH. B is the maximum number of CFs in a level of the tree

way to the insertion in the classic B-tree. If the closest-CF in the leaf cannot absorb the data point, a new CF entry is created. If there is no room for new leaf, the parent node is split.

A leaf node might be expanded due to the constraints imposed by B and T. The process consists of taking the two farthest CFs and creates two new leaf nodes. BIRCH operates in two main steps: the first step builds an initial CF-tree in memory using the given amount of memory and recycling space on disk; the second step tries to cluster all the sub-clusters in the leaf nodes, called also the "global clustering". There are two optional steps: the"tree condensing" step which aims to refine the initial CF-tree by re-inserting its leaf entries; and the "clustering refinement" step which re-assigns all the data points based on the cluster centroid produced by the global clustering step.

4.3.1.2 Evolution-based technique for stream clustering (E-Stream)

E-Stream [Udommanetanakit et al., 2007] classifies the evolution of data into five categories: appearance, disappearance, self evolution, merge, and split. This algorithm is an *evolution-based stream clustering* method, i.e., a stream clustering method that supports the monitoring and the change detection of clustering structures. It uses another data structure for saving summary statistics, named the α-bin histogram [Udommanetanakit et al., 2007].

Indeed, each cluster is represented as a *Fading Cluster Structure* (FCS), which is a weighted CF, utilizing an α-bin histogram for each feature of the dataset. A histogram of the cluster data values is utilized to identify cluster splits.

When the maximum or minimum value changes, a new range is calculated and the values in each range are updated from the intersection between the new and old ranges. Each cluster has a histogram of feature values, but the histogram is utilized only for the split of active clusters. Only an active cluster can assemble an incoming data point. If a statistically significant valley is found between two peaks in any of the marginal histograms, the cluster is split. Figure 4.6 illustrates the histogram management in a split. *E-Stream* starts empty, and every new point

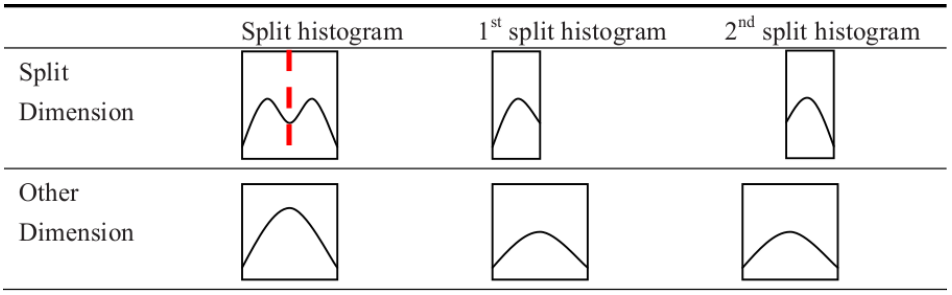


FIGURE 4.6: Histogram management in a split dimensionop and other dimension [Udommanetanakit et al., 2007]

either is mapped onto one of the existing clusters (based on a radius threshold) or a new cluster is created around it. Any cluster not meeting a predefined density level is considered inactive and remains isolated until achieving a desired weight. The weight of a cluster is the number of data elements assigned to this cluster. The algorithm employs an exponential decay function to weigh down the influence of older data, thus focuses on keeping an up-to-date view of the data distribution. Clusters which are not active for a certain time period may be deleted from the data space.

Algorithm 9: E-Stream

```
retrieve new data x_i;
FadingAll;
CheckSplit;
MergeOverlapCluster;
LimitMaximumCluster;
FlagActiveCluster;
(minDistance, index) ← FindClosestCluster;
if minDistance < radius_factor then
    add x_i to the cluster c_index;
else
    create a new cluster from x_i;
waiting for new data;
```

4.3.1.3 Evolution-based clustering for heterogeneous data streams with uncertainty

HUE-Stream [Meesuksabai et al., 2011] extends E-Stream in order to support uncertainty in heterogeneous data, i.e., including numerical and categorical attributes simultaneously. Uncertain data streams pose a special challenge because of the dual complexity of high volume and data uncertainty. This uncertainty is due to errors in the reading of sensors or other hardware collection technology.

In many of these cases, the data errors can be approximated either in terms of statistical parameters, such as the standard deviation, or the probability density functions [Aggarwal, 2013]. The Uncertain MICROclustering (*UMicro*) algorithm is proposed as a method for clustering uncertain data streams, which enhances the micro-clusters with additional information about the uncertainty of the data points in the clusters [Aggarwal and Yu, 2008]. This information is used to improve the quality of the distance functions for the cluster assignments.

HCluStream [Yang and Zhou, 2006] extends the definition of the cluster feature vector to include categorical features, by replacing the modified k-means clustering with the corresponding k-prototypes clustering which is able to handle heterogeneous attributes. The centroid of continuous attributes and the histogram of the discrete attributes are used to represent the micro-clusters, and the k-prototype clustering algorithm is used to create the micro-clusters and macro-clusters.

The distance function, cluster representation, and histogram management are introduced for the different types of clustering structure evolution. A *distance function* between the probability distributions of two observations is introduced

to support uncertainty in categorical attributes.

To detect changes in the clustering structure, the proposed distance function is used to merge clusters and find the closest cluster of a given incoming data and the proposed histogram management to split clusters for categorical data. To decrease the weight of old data over time, a fading function is used. Experimental results show that HUE-Stream gives better cluster quality, in terms of *purity* and the *F-measure*, compared to UMicro for the KDD-CUP'99 dataset [Meesuksabai et al., 2011].

4.3.1.4 ClusTree

ClusTree [Kranen et al., 2011] is a non-parametric stream clustering algorithm that is capable of processing the stream in a single pass, with limited memory usage. It always maintains an up-to-date cluster model and reports concept drift, novelty, and outliers. This is ensured by weighing data points with an exponential time-dependent decay function. Moreover, this approach makes no a priori assumptions on the size of the clustering model, but dynamically self-adapts.

ClusTree is an anytime algorithm that organizes micro-clusters in a tree structure for faster access and automatically adapts micro-cluster sizes based on the variance of the assigned data points. Anytime algorithms denote approaches that are capable of delivering a result at any given point in time, and of using more time if it is available to refine the result. The tree used in ClusTree is a balanced multi-dimensional indexing structure with the following properties:

- an inner node contains between m and M entries. Leaf nodes contain between l and L entries. The root has at least one entry (m, M, l and L are input parameters).
- an entry in an inner node stores: (i) a cluster feature of the observations that it represents. (ii) a cluster feature of the observations in the buffer. (iii) a pointer to its child node.
- an entry in a leaf stores a cluster feature of the data point(s) it represents.
- a path from the root to any leaf node has always the same length (balanced).

So, it uses also the micro-cluster structure as a compact representation of the data distribution. The basic idea is to maintain measures for incremental computation

of the mean and variance of micro-clusters so that the infeasible access to all past stream observations is no longer necessary. We recall that a micro-cluster is a cluster feature tuple (or a variant of it) $CF = (N, LS, SS)$ (as defined in section 4.3.1.1) of the number N of represented data points, their linear sum LS, and their squared sum SS.

In the proposed method, CFs are created and updated by extending index structures from the R-tree family [Guttman, 1984]. Such hierarchical indexing structures provide the means for efficiently locating the correct place to insert any observation from the stream into a micro-cluster. The idea is to build a hierarchy of micro-clusters at different levels of granularity.

Given enough time, the algorithm descends the hierarchy in the index to reach the leaf entry that contains the micro-cluster that is the most similar to the current observation. If this micro-cluster is similar enough, it is updated incrementally by this observation's values. Otherwise, a new micro-cluster may be formed [Kranen et al., 2011].

However, in anytime clustering of streaming data, there might not always be enough time to reach leaf level to insert the observation. To deal with this, the authors provide some strategies for anytime inserts. By incorporating local aggregates, i.e., temporary buffers for "hitchhikers", a solution is provided for the easy interruption of the insertion process so that it can be simply summarized at any later point in time. For very fast streams, aggregates of similar observations allow insertion of groups instead of single observations for even faster processing. For slower stream settings, alternative insertion strategies that exploit possible idle times of the algorithm to improve the quality of the resulting clustering are proposed [Kranen et al., 2011].

Taking the means of the CFs as representatives, we can apply a k-center clustering or density based clustering (e.g. k-means or DBSCAN) to detect clusters of arbitrary shape.

4.3.2 Partitioning stream methods

A partitioning-based clustering algorithm organizes the observations into k disjoint clusters. The clusters are formed based on a distance function. As example of partitioning methods, the k-means algorithm which leads to finding only spherical clusters and the clustering results are usually influenced by noise.

4.3.2.1 CluStream

The idea behind the *CluStream* [Aggarwal et al., 2003] method is to divide the clustering process into an online component which periodically stores detailed summary statistics and an offline component which uses only this summary statistics. The offline component is utilized by the analyst who can use a wide variety of inputs (such as the time horizon or number of clusters) in order to provide a quick understanding of the broad clusters in the data stream. The summary information is defined by the following structures:

- **Micro-clusters:** Statistical information about the data locality in terms micro-clusters are maintained. The *micro-cluster* structure is a temporal extension of the *cluster feature vector* 4.3.1.1 [Zhang et al., 1996]. The additivity property of the micro-clusters makes them a natural choice for the data stream problem. More precisely, a micro-cluster is tuple (N, LS, SS, LST, SST) where
  - (N, LS, SS) are the three components of the CF vector, as introduced in section 4.3.1.1 (namely, the number of data points in the cluster, N; the linear sum of the N data points, LS; and the squared sum of the N data points, SS).
  - The two other components are $LST = \sum_i T_i$ and $SST = \sum_i T_i^2$ (the sum and the sum of the squares of the time stamps of the N data points).
- **Pyramidal time frame:** The micro-clusters are stored at time snapshots which follow a pyramidal pattern. This pattern provides an effective trade-off between the storage requirements and the ability to recall summary statistics from different time horizons.

The data stream clustering algorithm proposed in [Aggarwal et al., 2003] can generate approximate clusters in any user-specified length of history from the current moment. The online phase stores q micro-clusters in (secondary) memory, where q is an input parameter.

Each new point is assigned to its closest micro-cluster (according to the Euclidean distance) if the distance between the new point and the closest centroid falls within the maximum boundary. If so, the point is absorbed by the cluster and

its summary statistics are updated. If none of the micro-clusters can absorb the point, a new micro-cluster is created. This is accomplished by either deleting the oldest micro-cluster or by merging two micro-clusters. The oldest micro-cluster is deleted if its time-stamp is below a given threshold δ (input parameter).

The q micro-clusters are stored in a secondary storage device in particular time intervals that decrease exponentially, which are referred to as *snapshots*. These snapshots allow the user to search for clusters in different time horizons through a *pyramidal time window* concept. This summary information in the micro-clusters is used by an offline component which is dependent upon a wide variety of user inputs such as the time horizon or the granularity of clustering.

When the user specifies a particular time horizon of length h over which to find the clusters, then we need to find micro-clusters which are specific to that time-horizon. For this purpose, we find the additive property of the cluster feature vector very useful. The final clusters are determined by using a modification of a k-means algorithm. In this technique, the micro-clusters are treated as *pseudo-points* which are re-clustered in order to determine higher level clusters.

4.3.2.2 StreamKM++

StreamKM++ [Ackermann et al., 2012] is a two-phase (online-offline) algorithm which maintains a small outline of the input data using the *merge-and-reduce* technique. The merge step is performed by via a data structure, named the *bucket set*, which is a set of L buckets (also named buffers), where L is an input parameter. The reduce step is performed by a significantly different summary data structure that is suitable for high-dimensional data, the *coreset tree*, when we consider that it reduces $2m$ data points to m data points (m is an input parameter).

The advantage of such a coreset is that we can apply any fast approximation algorithm (for the weighted problem) on the usually much smaller coreset to compute an approximate solution for the original dataset more efficiently.

The *coreset tree* is constructed as follow:

- First, the tree has only the root node v, which contains all the $2m$ data points in the set of data points X_v, where X_v is the of observations assigned to the node v. The prototype of the root node $\mathbf{w}_v$ is chosen randomly from X_v.

- Afterwards, two child nodes for v are created: v_1 and v_2. To create these nodes, the data point that is farthest away from $\mathbf{w}_v$ has the highest probability of being selected We call the selected data point $\mathbf{x}_{v'}$.

- The next step is to allocate the data points in X_v to X_{v1} and X_{v2}, such that:

$$X_{v_1} = \{\mathbf{x} \in X_v \mid dist(\mathbf{x}, \mathbf{x}_v) < dist(\mathbf{x}, \mathbf{x}_{v'})\}, \tag{4.5}$$

$$X_{v_2} = X_v \setminus X_{v_1}. \tag{4.6}$$

Consequently, the summary statistics of the child nodes v_1 and v_2 are updated. This is the *expansion* step of the tree, which creates two child nodes for a given inner node. When the tree has many leaf nodes, we have to decide which one should be expanded first. In this case, we start from the root node of the coreset tree and descend it by iteratively selecting a child node with probability proportional to $\frac{SSE_{child}}{SSE_{parent}}$, until a leaf node is reached for the expansion step to be re-started, where

$$SSE_v = \sum_{\mathbf{x} \in X_v} \|\mathbf{x} - \mathbf{w}_v\|^2. \tag{4.7}$$

The coreset tree expansion stops when the number of leaf nodes is m.

When a new data point arrives, it is stored in the first bucket. If the first bucket is full, all of its data are moved to the second bucket. If the second bucket is full, the two buckets are merged resulting in $2m$ data points, which are then reduced to m data points, by the construction of a *coreset tree*, as previously detailed. The resulting m data points are stored in the third bucket, unless it is also full, and then again a new merge-and-reduce step is needed [Ackermann et al., 2012, de Andrade Silva et al., 2013].

In its offline phase, the k-means++ [Arthur and Vassilvitskii, 2007], which is executed on an input set of size m, is used for finding the final clusters. The k-means++ method (presented in chapter 3) is a seeding procedure for the k-means algorithm that guarantees a solution with a certain quality and gives good practical results.

4.3.2.3 Data stream clustering with Affinity Propagation (StrAP)

StrAP [Zhang et al., 2008] is an extension of the Affinity Propagation (AP) [Frey and Dueck, 2007] algorithm for data streams, which uses a reservoir for saving

potential outliers. The Affinity Propagation approach (presented in chapter 3) proposes an equivalent formulation of the k-medoids problem in the sense that a prototype is an effective data point, with the difference that the number of clusters to be found is not fixed.

The StrAP algorithm, as an online version of AP, proceeds by incrementally updating the current model if the current data point fits the model, and putting it in a reservoir otherwise. A change point detection test enables StrAP to catch drifting exemplars that significantly deviate away.

StrAP involves four main steps as illustrated in Algorithm 10 with a diagram in Figure 4.7 [Zhang et al., 2008]:

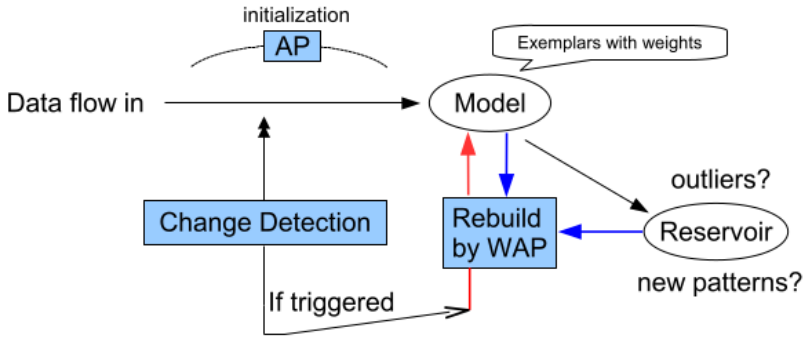


FIGURE 4.7: Diagram of StrAP algorithm [Zhang et al., 2008]

- The first batch of data is used by AP to identify the first clusters and initialize the stream model.
- As the stream flows in, each data point $\mathbf{x}_t$ is compared to the prototypes; if it is too far from the nearest exemplar, $\mathbf{x}_t$ is put in the *reservoir*, otherwise the stream model is updated accordingly.
- The data distribution is checked for change point detection, using the Page-Hinkley significance test.
- Upon triggering the change detection test, or if the number of outliers exceeds the reservoir size, the stream model is rebuilt based on the current model and reservoir, using a weighted version of AP (WAP).

The model of the data stream used in StrAP is inspired by DenStream [Cao et al., 2006] (presented in section 4.3.4). It consists of a set of 4-tuples $(c_i, N_i, \Sigma_i, T_i)$, where

- c_i ranges over the clusters;
- N_i is the number of items associated to cluster c_i;
- Σ_i is the distortion of c_i (sum of $d(\mathbf{x}, c_i)^2$, where $\mathbf{x}$ ranges over all data points associated to c_i);
- T_i is the last time stamp when a data point was associated to c_i.

At time t, the data point $\mathbf{x}_t$ is considered and its nearest cluster c_i (w.r.t. distance d) in the current model is selected; if $d(\mathbf{x}_t, c_i)$ is less than some threshold δ, heuristically set to the average distance between points and clusters in the initial model, $\mathbf{x}_t$ is assigned to the i-th cluster and the model is updated accordingly; otherwise, $\mathbf{x}_t$ is considered to be an outlier, and put into the reservoir [Zhang et al., 2008].

Algorithm 10: StrAP

Data: $\mathcal{DS} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$, fit threshold ϵ
Init: StrAP Model $\leftarrow$ AP$(\mathbf{x}_1, ..., \mathbf{x}_T)$;
Reservoir = {};
for $t > T$ **do**
    Compute c_i = nearest cluster to $\mathbf{x}_t$;
    if $dist(c_i, \mathbf{x}_t) < \epsilon$ **then**
        Update StrAP model;
    else
        Reservoir $\leftarrow \mathbf{x}_t$;
    if *Restart criterion* **then**
        Rebuild StrAP model;
        Reservoir = {};

4.3.3 Gaussian mixture models of data streams under block evolution

[Patist et al., 2006] introduced a local approach for maintaining a Gaussian mixture model of a data stream under block evolution with restricted window. In the proposed algorithm, block evolution is considered with a restricted window consisting of a fixed number, b, of the most recently collected blocks of data. The window is updated one block at a time by inserting a new block and deleting the oldest one. The method constructs b local mixtures, one for each block. Mixtures

are stored as lists of k components, i.e., k Gaussian density functions.

When a new block of data arrives, all components from the oldest block are removed and the EM procedure (which is presented in chapter 3) is applied to the latest block to find a local mixture model for this block. Finally, all bk local components are combined with help of a greedy merge procedure to form a global model with k components. The greedy merge procedure systematically searches for two closest components and merges them with help of the above formulas until there are exactly k components left.

The method is efficient both computationally and in terms of memory requirements; it is 1-2 orders of magnitude more efficient than the standard EM algorithm.

In [Samé and Assaad, 2014, EL ASSAAD, 2014], the authors proposed a dynamic probabilistic approach within a temporal data clustering framework. An online variational EM for dynamic mixture model has been also proposed for the estimation of model parameters associated with this approach.

4.3.4 Density-based stream methods

Density-based algorithms are based on the connection between regions and density functions. In these types of algorithms, dense areas of observations in the data space are considered as clusters, which are segregated by low density area (noise). These algorithms find clusters of arbitrary shapes and generally they require two parameters: the radius and the minimum number of data points within a cluster.

The main challenge in the streaming scenario is to construct density-based algorithms which can be efficiently executed in a single pass of the data, since the process of density estimation may be computationally intensive [Aggarwal, 2013]. [Amini et al., 2014] gives a survey on recent density-based data streams clustering algorithms.

4.3.4.1 Density-based clustering over an evolving data stream with noise (DenStream)

DenStream [Cao et al., 2006] is a density-based data stream clustering algorithm that also uses a feature vector, called micro-clusters, based on the *CF* vector which is introduced in section 4.3.1. By creating two kinds of micro-clusters (*potential* and *outlier micro-clusters*), in its online phase, *DenStream* overcomes one

of the drawbacks of *CluStream*, its sensitivity to noise. Potential and outlier micro-clusters are kept in separate memories since they require different processing.

Each *potential-micro-cluster* structure has an associated weight w that indicates its importance based on temporality. The weight of each data point decreases exponentially with time t via a fading function

$$f(t) = 2^{-\lambda t} \tag{4.8}$$

where $\lambda > 0$. If the weight $w = \sum_{j=1}^{n} f(t - T_{ij})$ is above a threshold input parameter μ then the corresponding cluster is considered as a *core-micro-cluster*, where $T_{i1}, ..., T_{in}$ are timestamps of data points $\mathbf{x}_{i1}, ..., \mathbf{x}_{in}$.

At time t, if $w \geq \beta\mu$ then the micro-cluster is considered as *potential-micro-cluster*, else it is an *outlier-micro-cluster*, where β is the threshold of the outlier relative to core-micro-clusters ($0 < \beta < 1$). Micro-clusters with no recent points tend to lose importance, i.e. their respective weights continuously decrease over time in *outdated-micro-clusters*. However, the latter could grow into a potential micro-cluster when, by adding new points, its weight exceeds the threshold. The weights of micro-clusters are periodically calculated and the decision about removing or keeping them is made based on the weight threshold.

When a new data point arrives, the algorithm tries to insert it into its nearest *potential-micro-cluster* based on its updated radius. If the insertion is not successful, the algorithm tries to insert the data point into its closest *outlier micro-cluster*. If the insertion is successful, the cluster summary statistics will be updated accordingly. Otherwise, a new *outlier micro-cluster* is created to absorb this point. The Euclidean distance between the new data point and the center of the nearest potential or outlier micro-cluster is measured. A micro-cluster is chosen with the distance less than or equal to the radius threshold.

DenStream has a pruning method in which it frequently checks the weights of the outlier-micro-clusters in the outlier buffer to guarantee the recognition of the real outliers. However, the non-release of the allocated memory when either deleting a micro-cluster or merging two old micro-clusters is considered as a limitation of the *DenStream* algorithm as well as the time-consuming pruning phase for removing outliers [Amini et al., 2014].

In the offline phase, the potential-micro-clusters found during the online phase are considered as *pseudo-points* and will be passed to a variant of the DBSCAN algorithm in order to determine the final clusters.

Algorithm 11: DenStream

Data: $\mathcal{DS}$, ϵ, β, μ, λ

$T_{\mathbf{x}} = \left\lceil \frac{1}{\lambda} \log(\frac{\beta\mu}{\beta\mu - 1}) \right\rceil$;
Get the next point $\mathbf{x}$ at current time t from data stream $\mathcal{DS}$;
Merging($\mathbf{x}_t$);
if $(t \mod T_{\mathbf{x}}) = 0$ **then**
  for *each p-micro-cluster* c_p **do**
    if w_p *(the weight of* c_p *)* $< \beta\mu$ **then**
      Delete c_p;
  for *each o-micro-cluster* c_o **do**
    $\xi = \frac{2^{-\lambda(t-t_0+T_p)}-1}{2^{-\lambda T_p}-1}$ **if** w_o *(the weight of* c_o *)* $< \xi$ **then**
      Delete c_o;
if *a clustering request arrives* **then**
  Generating clusters ;

4.3.4.2 Self organizing density-based clustering over data stream (SOStream)

SOStream [Isaksson et al., 2012] is a density-based clustering algorithm inspired by both the principle of the DBSCAN algorithm and self-organizing maps (SOM) [Kohonen et al., 2001], in the sense that a winner influences its immediate neighborhood. Generally speaking, density-based clustering algorithms require a manually set threshold (similarity threshold, grid size, etc.) for which is difficult to choose the most suitable value and if it is set to an unsuitable value, then the algorithm will suffer from over-fitting, or from unstable clustering. SOStream addresses this problem by using a dynamically learned threshold value for each cluster based on the idea of building neighborhoods with a minimum number of points.

SOStream is also represented by a set of micro-clusters where for each cluster a cluster feature (CF) vector is stored, which is a tuple with three elements $CF_i = (N_i, r_i, c_i)$, N_i is the number of data points assigned to c_i, r_i is the cluster's radius and c_i is the prototype.

When a new data-point arrives, the nearest cluster is selected, and then it absorbs this data-point if the calculated distance is less than a dynamically defined threshold. It also assigns the micro-clusters' neighbors to the nearest cluster, i.e., the prototypes of clusters sufficiently close to the winning cluster have their prototypes modified to be closer to the winning cluster's prototype.

This approach is used to assist in merging similar clusters and increasing separation between different clusters. The neighborhood of the winner is defined based

on the idea of a *MinPts* distance given by a minimum number of neighboring observations [Cao et al., 2006]. This distance is found by computing the Euclidean distance from any existing clusters to the winning cluster. If the new point is not absorbed by any micro-cluster, a new micro-cluster is created for it.

In the *SOStream* algorithm, merging, updating and adapting dynamically the threshold value for each cluster are performed in an online manner. Clusters are merged if they overlap with a distance that is less than the merge-threshold, i.e., the spheres in d-dimensional space defined by the radius of each cluster overlap. Hence, the threshold value is a determining factor for the number of clusters. When two clusters are merged, the largest radius of these two clusters is chosen to be the radius of the cluster to avoid losing any data points within the clusters. However, no split feature is proposed in the algorithm. *SOStream* also uses an exponential fading function to reduce the impact of old data whose relevance diminishes over time.

4.3.4.3 SVStream

SVStream (Support Vector based Stream clustering) [Wang et al., 2013] is a data stream clustering algorithm based on support vector clustering (SVC) and support vector domain description (SVDD).

In the Support Vector Clustering (SVC) [Ben-Hur et al., 2001] algorithm data points are mapped from the data space to a high dimensional feature space using a Gaussian kernel. In the feature space we look for the smallest sphere that encloses the image of the data. This sphere is mapped back to data space, where it forms a set of contours which enclose the data points. These contours are interpreted as cluster boundaries. Points enclosed by each separate contour are associated with the same cluster. Support vectors are used to construct cluster boundaries of arbitrary shape in SVC.

Support vector domain description (SVDD) [Tax and Duin, 1999] is a one-class classifier inspired by the support vector classifier. The idea is to use kernels to project data into a feature space and then to find the sphere enclosing almost all data, namely not including outliers. SVDD has the possibility to reject a fraction of the training data points, when this sufficiently decreases the volume of the hypersphere. One inherent drawback of SVDD, which significantly affects not only its outlier detection performance but also its general properties, is that the resulting description is highly sensitive to the selection of the trade-off parameter,

which is difficult to estimate in practice.

Given a set of M data elements, the Gaussian kernel parameter q and the trade-off parameter C, the sphere structure S is defined as

$$S = \{SV, BSV, \|\mu\|^2, R_{SV}, R_{BSV}\}.$$

where,

- SV is a support vector set.
- BSV is a bounded support vector set.
- $\|\mu\|^2$ is the squared length of the sphere center μ.
- R_{SV} is the radius of the sphere.
- R_{BSV} is the maximum Euclidean distance of the bounded support vectors from the sphere center μ.

The multi-sphere set SS is defined as a set consisting of multiple spheres, that is, $SS = \{S^1, ..., S^{|SS|}\}$, where the superscript denotes the index of a sphere. In SVStream, the elements of a data stream are mapped to a kernel space, and the support vectors are used as the summary information of the historical elements to construct the cluster boundaries of arbitrary shape. To adapt both sudden and gradual changes, multiple spheres are dynamically maintained, each describing the corresponding data domain presented in the data stream.

When a new data batch arrives, if a sudden change occurs, a new sphere is created; otherwise, only the existing spheres are updated to take into account the new batch. The data elements of this new batch are assigned with cluster labels according to the cluster boundaries constructed by the sphere set. Bounded support vector (BSVs) and a newly designed BSV decaying mechanism are introduced so as to respectively identify overlapping clusters and automatically detect outliers (noise) [Wang et al., 2013]. In the clustering process, if two spheres are too close to each other, they should be merged. In addition, eliminating old BSVs by the BSV decaying mechanism would help detect the tendency of a cluster to shrink or split.

4.3.5 Grid-based stream methods

Grid-based clustering is another group of the clustering methods for data streams where the data space is quantized into finite number of cells which form the grid structure and perform clustering on the grids. Grid-based clustering maps the infinite number of data records in data streams to a finite number of grids. Then, the grids are clustered based on their density.

4.3.5.1 D-Stream

D-Stream [Chen and Tu, 2007] is also a two-phase scheme which consists of an online component that processes input data stream and produces summary statistics and an offline component that uses the summary data to generate clusters. In the online component, the algorithm maps each input data point onto a grid whereas in the offline component, it computes the grid density and clusters the grids based on the density. The algorithm adopts a density decaying technique to capture the dynamic changes of a data stream and it can find clusters of arbitrary shapes.

Unlike other algorithms such as CluStream [Aggarwal et al., 2003], D-Stream automatically and dynamically adjusts the clusters without requiring user specification of target time horizon and number of clusters. Algorithm 12 outlines the overall architecture of *D-Stream*.

For a data stream, at each time step, the online component of *D-Stream* continuously reads a new data point, places the multi-dimensional data into a corresponding discretized density grid in the multi-dimensional space, and updates the characteristic vector of the density grid (Lines 4-7 of Algorithm 12). The density for a grid g, at a given time t, $D(g,t)$ is defined as the sum of the density coefficients of all data records that are mapped to g. That is the density of g at t is:

$$D(g,t) = \sum_{\mathbf{x} \in E(g,t)} D(\mathbf{x},t) \tag{4.9}$$

where $E(g,t)$ is the set of data points that are mapped to g at or before time t. The density of any grid is constantly changing. However, the updating operation is executed only when a new data record is mapped to that grid.

D-Stream uses the CF vector concept associated to each grid. This is a tuple $(t_g, t_m, D, label, status)$, where t_g is the last time when g is updated, t_m is the last time when g is removed from grid list as a sporadic grid (if ever), D is the

grid density at the last update, *label* is the class label of the grid, and *status* = { *SPORADIC, NORMAL*} is a label used for removing sporadic grids.

The procedures *initial_clustering* (used in Line 9 of Algorithm 12) and *adjust_clustering* (used in Line 12 of 12) first update the density of all active grids to the current time. Once the density of grids are determined at the given time, the clustering procedure is similar to the standard method used by density-based clustering.

The offline phase dynamically adjusts the clusters every *gap* time steps, where *gap* is an integer parameter. After the first *gap*, the algorithm generates the initial cluster (Lines 8-9). Then, the algorithm periodically removes sporadic grids and adjusts the clusters (Lines 10-12) [Chen and Tu, 2007].

Algorithm 12: D-Stream

```
1  time_c = 0;
2  initialize an empty hash table grid_list;
3  while there is a data point to proceed do
4      Get the next data point in the data stream, x = (x_1, x_2, ..., x_d);
5      Determine the density grid g that contains x;
6      if g not in grid_list then Insert g to grid_list;
7      Update the characteristic vector of g;
8      if time_c = gap then
9          Call initial_clustering(grid_list);
10     if time_c mod gap == 0 then
11         Detect and remove sporadic grids from grid_list;
12         Call adjust_clustering(grid_list);
13     time_c = time_c + 1;
```

One weakness of the approach is that a significant number of non-empty grid cells need to be discarded in order to keep the memory requirements in check. In many cases, such grid-cells occur at the borders of the clusters. The discarding of such cells may lead to a degradation in cluster quality [Aggarwal, 2013].

4.3.6 GNG based algorithms

4.3.7 Online version of GNG

As presented in chapter 3, the GNG algorithm constructs a graph of nodes in which each node has its associated prototype. Prototypes can be regarded as positions

in the input space of their corresponding nodes. Pairs of nodes are connected by edges (links), which are not weighted. The purpose of these links is to define the topological structure. These links are temporal in the sense that they are subject to aging during the iteration steps of the algorithm and are removed when they become "too old" [Beyer and Cimiano, 2012].

Starting with two nodes, and as a new data point is available, the nearest and the second-nearest nodes are identified, linked by an edge, and the nearest node and its topological neighbors are moved toward the data point. Each node has an accumulated error variable. Periodically, a node is inserted into the graph between the nodes with the largest error values. Nodes can also be removed if they are identified as being superfluous. This is an advantage compared to SOM and NG, as there is no need to fix the graph size in advance. Algorithm 13 outlines an online version of the GNG approach. In this version, unlike the standard approach of GNG (which is presented in chapter 3), the data is seen only once.

Algorithm 13: GNG online

Data: $\mathcal{DS} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$
Result: set of nodes $\mathcal{C} = \{c_1, c_2, ...\}$ and their prototypes $\mathbf{W} = \{\mathbf{w}_{c_1}, \mathbf{w}_{c_2}, ...\}$

```
Initialize node set C to contain two nodes, c1 and c2: C = {c1, c2};
while there is a data point to proceed do
    Get the next data point in the data stream, x_i;
    Find the nearest node bmu_1 and the second nearest node bmu_2;
    Update edges as described in Algorithm 14;
    if the number of data points passed is an integer multiple of a parameter
     β then
        Insert a new node as described in Algorithm 15;
    Delete each isolated node;
    Finally, decrease the error of all units;
```

Algorithm 14: Edge Management

```
Increment the age of all edges emanating from bmu_1 and weight them;
if bmu_1 and bmu_2 are connected by an edge then
    set the age of this edge to zero
else
    create an edge between bmu_1 and bmu_2, and mark its time stamp;
Remove edges whose age is greater than age_max;
```

Algorithm 15: Node Insertion

1 Find node q with the maximum accumulated error;
2 Find the neighbor f of q with the largest accumulated error;
3 Add the new node, r, half-way between nodes q and f: $\mathbf{w}_r = 0.5(\mathbf{w}_q + \mathbf{w}_f)$;
4 Insert edges connecting the new node r with nodes q and f, and remove the original edge between q and f;

A number of authors have proposed variations on the Growing Neural Gas (GNG) approach [Sledge and Keller, 2008, Mendes et al., 2014, Mitsyn and Ososkov, 2011]. The GNG algorithm creates a new node every λ iterations (λ is fixed by the user as an input parameter). Hence, it is not adapted for data streams, or non-stationary datasets, or to novelty detection.

In order to deal with non-stationary datasets, the author of [Fritzke, 1997] has investigated modifying the network by proposing an on-line criterion for identifying "useless" nodes. The algorithm proposed is known as the Growing Neural Gas with Utility (GNGU). Slow changes of the distribution are handled by the adaptation of existing nodes, whereas rapid changes are handled by removal of "useless" neurons and subsequent insertions of new nodes in other places.

4.3.7.1 Grow When Required (GWR)

The GWR network [Marsland et al., 2002] may add a new node at any time, whose position is dependent on the input and the current winning node. The GWR deals with the problem of novelty detection by adding new nodes into the network structure whenever the activity of the current best-matching node is below some threshold, which implies that the best-matching node is not trained to deal with that particular input. This means that the network grows very quickly when new data is presented, but stops growing once the network has matched the data to a given accuracy.

This has benefits in that there is no need to decide in advance how large the network should be, as nodes will be added until the network is saturated. This means that for small datasets the complexity of the network is significantly reduced. In addition, if the dataset changes at some time in the future, further nodes can be added to represent the new data without disturbing the network that has already been created [Marsland et al., 2002, 2005].

Considering one iteration of the GWR algorithm, GWR has approximatively the same time complexity as one iteration of GNG. Hence, the complexity of GWR

is $O(knm)$ where k is the number of iterations, n is the number of data points of the data stream m is the number of nodes in the graph.

4.3.7.2 Incremental variants of GNG

Still in the same idea of relaxing the constraint of periodical evolution of the network, the IGNG [Prudent and Ennaji, 2005] algorithm has been proposed. In this algorithm a new neuron is created each time the distance of the current input data to the existing neuron is greater than a predefined fixed threshold σ, which is dependent on the global datasets. However, one disadvantage of this algorithm is the global character of the parameter σ and also that it must be computed prior to the learning.

In order to resolve this weakness, I2GNG [Hamza et al., 2008] associates a threshold variable σ to each neuron. However, its major drawback is the initialization of the σ values at the creation of each node. The authors of [Lamirel et al., 2010] address the problem of choosing the final winner neuron among the many input equidistant neurons. They proposed some adaptations of the IGNG and I2GNG algorithms. Notably, the use of a labeling maximization approach as a clustering similarity measure (IGNG-F) to replace the distance in the winner selection process.

The ability of self-organizing neural network models to manage real-time applications, using a modified learning algorithm for a growing neural gas network is addressed in [GarcíA-RodríGuez et al., 2012]. The proposed modification aims to satisfy real-time temporal constraints in the adaptation of the network. The proposed learning algorithm can add a dynamic number of neurons per iteration. Indeed, a detailed study has been conducted to estimate the optimal parameters that keep a good quality of representation in the available time. The authors concluded that the use of a large number of neurons made it difficult to obtain a representation of the distribution of training data with good accuracy in real-time [GarcíA-RodríGuez et al., 2012, Pimentel et al., 2014].

AING [Bouguelia et al., 2013] is an incremental GNG that learns automatically the distance thresholds of nodes based on its neighbors and data points assigned to the node of interest. It merges nodes when their number reaches a given *upper-bound*.

4.3.8 Computational complexity

In Table 4.1, we report the computational complexity of some of the data stream clustering algorithms presented above. Note: n: number of data points, k: number of network nodes (or clusters), m: number of micro-clusters in main memory, g: number of grids in the grid list, k': number of outliers in the reservoir.

Algorithm	Complexity
GNG	$O(kn^2)$ [Mendes et al., 2014]
G-Stream	$O(nk)$ (this algorithm is presented in the next chapter 5)
SOStream	$O(n^2 \log n)$ [Amini et al., 2014]
StrAP	$O((k + k'))^2$ [Zhang et al., 2008]
DenStream	$O(m)$ [Amini et al., 2014]
D-Stream	$O(1) + O(g)$ [Amini et al., 2014]

TABLE 4.1: Computational complexity of data stream clustering algorithms

4.3.9 Summary

Table 4.2 summarizes the main features offered by each algorithm considered in terms of: the basic clustering algorithm, whether the algorithm identifies a topological structure, whether the links (if they exist) between clusters (nodes) are weighted, how many phases it adopts (online and offline), the types of operations for updating clusters (remove, merge, and split cluster), and whether a *fading* function is used.

Algorithms	based on	topology	WL	phases	remove	merge	split	fade
SVStream	SVC, SVDD	✗	✗	online	✓	✓	✓	✓
StreamKM++	k-means++	✗	✗	2 phases	✓	✓	✓	✓
StrAP	AP	✗	✗	2 phases	✓	✗	✗	✓
SOStream	DBSCAN, SOM	✗	✗	online	✓	✓	✗	✓
IGNG	NG	✓	✗	online	✗	✗	✗	✗
HCluStream	k-prototypes	✗	✗	2 phases	✓	✓	✓	✓
GWR	NG	✓	✗	online	✗	✗	✗	✗
G-Stream	NG	✓	✓	online	✓	✗	✗	✓
E-Stream	k-means	✗	✗	2 phases	✓	✓	✓	✓
D-Stream	-	✗	✗	2 phases	✓	✓	✓	✓
DenStream	DBSCAN	✗	✗	2 phases	✓	offline	✗	✓
ClusTree	k-means or DBSCAN	✗	✗	2 phases	✓	offline	✓	✓
CluStream	k-means	✗	✗	2 phases	✓	offline	✗	✗
AING	NG	✓	✗	online	✗	✓	✗	✗

TABLE 4.2: Comparison between algorithms (WL: weighted links, 2 phases : online+offline).

4.4 Conclusion

Recently, examples of applications relevant to streaming data have become more numerous and more important, including network intrusion detection, transaction streams, phone records, web click-streams, social streams, weather monitoring, etc. Indeed, the data stream clustering problem has become an active research in recent years. This problem requires a process capable of partitioning observations continuously while taking into account restrictions of memory and time.

In this chapter, we surveyed, in a comprehensive manner, a number of the representative state-of-the-art algorithms for the clustering over data streams, and detailed some models. These algorithms are categorized according to the nature of their underlying clustering approach, including GNG, hierarchical, partitioning, density, and grid-based stream methods. Motivated by the need by industry for real time analysis, an increasing number of systems to support real-time data integration and analytics has emerged in recent years.

The work presented in this chapter has resulted in the following publication: Mohammed Ghesmoune, Mustapha Lebbah, and Hanane Azzag. State-of-the-art on Clustering Data Streams. Big Data Analytics, 2016. *Upon invitation*.

In the next chapter, we will detail our first contribution, concerning the G-Stream algorithm which is a clustering data stream method based on the GNG approach.

Chapter 5

G-Stream : Growing neural gas over data stream

This chapter presents our first novel contribution, concerned with extending the GNG approach to deal with streaming data. For self-containedness, this chapter begins with a description of the GNG algorithm. Afterwards, the one-pass streaming clustering algorithm titled G-Stream (Growing Neural Gas over Data Streams) is presented. After that, the quality of the proposed method is evaluated visually and in terms of various performance criteria on synthetic and real-world datasets.

5.1 Introduction

A data stream is a sequence of potentially infinite, non-stationary data (i.e., the probability distribution of the unknown data generation process may change over time) arriving continuously (which requires a single pass through the data) where random access to data is not feasible and storing all arriving data is impractical. The stream model is motivated by emerging applications involving massive datasets; for example, customer click streams, financial transactions, search queries, Twitter updates, telephone records, and observational science data are better modeled as data streams [Guha et al., 2003]. Mining data streams can be defined as the process of finding a complex structure in these large data.

While clustering is the problem of partitioning a set of observations into clusters such that observations assigned in the same cluster are similar (or close) and the inter-cluster observations are dissimilar (or distant), clustering data streams

requires, additionally, a process capable of partitioning observations continuously with restrictions of memory and time.

In the literature, many data stream clustering algorithms have been adapted from clustering algorithms, e.g., the partitioning method k-means [Ackermann et al., 2012], the density-based method DBSCAN [Cao et al., 2006, Isaksson et al., 2012], or the message passing-based method Affinity Propagation (AP) [Zhang et al., 2008].

In this work, we provide a one-pass streaming clustering algorithm titled G-Stream (Growing Neural Gas over Data Streams). We modify Growing Neural Gas (GNG) to obtain a new algorithm, whose main features and advantages are described as below:

- The topological structure is represented by a graph wherein each node represents a cluster, which is a set of "close" data points and neighboring nodes (clusters) are connected by edges. The graph size is not fixed but may evolve.
- We use an exponential fading function to reduce the impact of old data whose relevance diminishes over time. For the same reason, links between nodes are also weighted by an exponential function.
- Unlike many other data stream algorithms that start by taking a significant number of data points for initializing the model (these data points can be seen several times), G-Stream starts with only two nodes. Several nodes (clusters) are created in each iteration, unlike the traditional Growing Neural Gas algorithm [Fritzke, 1994].

5.2 Growing Neural Gas over data stream

In this section we introduce Growing Neural Gas over Data Streams (G-Stream) and highlight some of its novel features. We start by giving the model and data structure used in G-Stream. We assume that the data stream consists of a sequence $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$ of n (potentially infinite) data streams arriving in times $t_1, t_2, ..., t_n$, where $\mathbf{x}_i = (x_i^1, x_i^2, ..., x_i^d)$ is a vector in the $\mathbb{R}^d$ space. At each time, the model is represented by a graph $\mathcal{C}$ wherein each node represents a cluster. Each node $c \in \mathcal{C}$ has three variables: a prototype, a distance threshold, and an error variable. The prototype variable $\mathbf{w}_c = (w_c^1, w_c^2, ..., w_c^d)$ represents the position of the node in $\mathbb{R}^d$.

5.2.1 Growing Neural Gas

The GNG algorithm, as presented in chapter 3, constructs a graph of nodes in which each node has its associated prototype. Prototypes can be regarded as positions in the input space of their corresponding nodes. Pairs of nodes are connected by edges (links), which are not weighted. The purpose of these links is to define the topological structure. These links are temporal in the sense that they are subject to aging during the iteration steps of the algorithm and are removed when they become "too old" [Beyer and Cimiano, 2012].

Starting with two nodes, and as a new data point is available, the nearest and the second-nearest nodes are identified, linked by an edge, and the nearest node and its topological neighbors are moved toward the data point. Each node has an accumulated error variable. Periodically, a node is inserted into the graph between the nodes with the largest error values. Nodes can also be removed if they are identified as being superfluous. This is an advantage compared to SOM and NG, as there is no need to fix the graph size in advance. Algorithm 16 outlines an online version of the GNG approach. In this version, unlike the standard approach of GNG, the data is seen only once.

Algorithm 16: GNG online

Data: $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$
Result: set of nodes $\mathcal{C} = \{c_1, c_2, ...\}$ and their prototypes $\mathbf{W} = \{\mathbf{w}_{c_1}, \mathbf{w}_{c_2}, ...\}$

Initialize node set $\mathcal{C}$ to contain two nodes, c_1 and c_2: $\mathcal{C} = \{c_1, c_2\}$;
while *there is a data point to proceed* **do**
    Get the next data point in the data stream, $\mathbf{x}_i$;
    Find the nearest node bmu_1 and the second nearest node bmu_2;
    Update edges as described in Algorithm 17;
    if *the number of data points passed is an integer multiple of a parameter* β **then**
        Insert a new node as described in Algorithm 18;
    Delete each isolated node;
    Finally, decrease the error of all units;

5.2.2 G-Stream

This section aims to extend GNG to data streaming, especially to achieve a one-pass clustering. The proposed algorithm, called G-Stream, uses a reservoir to keep

temporarily the farthest data points in order to avoid needless movements of the nearest nodes to data points. It also takes into account the history of the data points by applying a fading function. The age of edges is not a simple increment, but we consider the temporal aspect in updating edges via an exponential function.

In G-Stream, the distance threshold δ_c, which is the distance from the node to the farthest data point assigned to it, is used when to decide if the data point will be added to the reservoir or the model will be updated accordingly. The first batch of data is passed without making a distance comparison. After that, the distance threshold for each node will take the distance to the farthest data point assigned to it. Figure 5.1 represents a schematic diagram of the G-Stream algorithm.

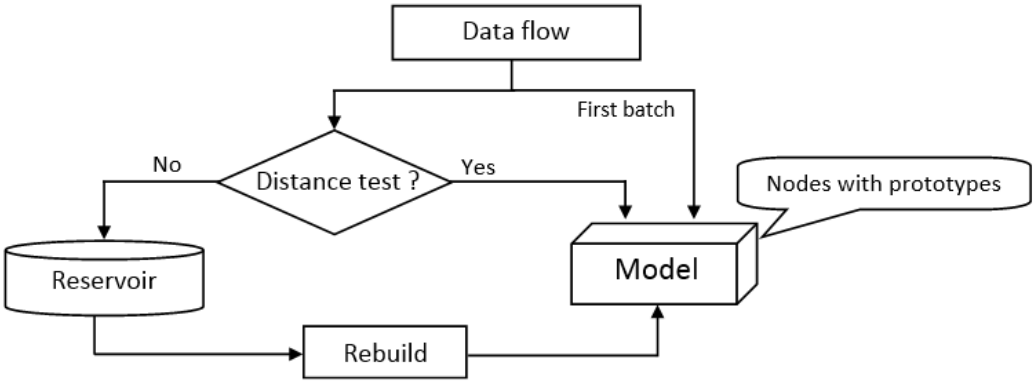


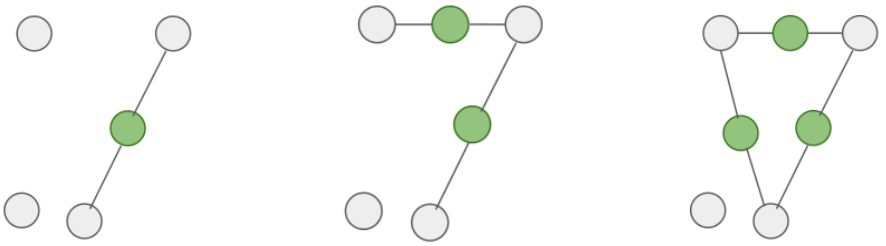
FIGURE 5.1: Diagram of G-Stream algorithm.

Starting with two nodes, and as a new data point is available, the nearest and the second-nearest nodes are identified, linked by an edge, and the nearest node with its topological neighbors are moved toward the data point. Each node q has an accumulated error variable and a weight rather than its prototype.

The error variable is used in the refinement step (where new nodes should be inserted). The weight variable varies over time using the fading function. Using the edge management procedure, one, two or three nodes are inserted into the graph between the nodes with the largest error values (Figure 5.2). Nodes can also be removed if they are identified as being superfluous.

5.2.2.1 Fading function

In most data stream scenarios, more recent data can reflect the emergence of new trends or changes in the data distribution [de Andrade Silva et al., 2013]. There are three window models commonly studied in data streams: landmark, sliding



(a) Insertion of one node (b) Insertion of two nodes (c) Insertion of three nodes

FIGURE 5.2: Insertion of one, two or three nodes in G-Stream.

and damped (as presented in chapter 4).

We consider, like many others, the damped window model, in which the weight of each data point decreases exponentially with time t via a fading function (Figure 5.3)

$$f(t) = 2^{-\lambda_1(t-t_0)} \tag{5.1}$$

where $\lambda_1 > 0$, defines the rate of decay of the weight over time, t denotes the current time and t_0 is the timestamp of the data point. Note that data points are passed according to the sliding windows principle. We use the number of the window to mark the timestamps of data points belonging to this window. The weight of a node is based on data points associated therewith:

$$\pi_c = \sum_{i=1}^{n_c} 2^{-\lambda_1(t-t_{i_0})} \tag{5.2}$$

where n_c is the number of points assigned to the node c at the current time t. If the weight of a node is smaller than a threshold value then this node is considered as outdated and then deleted (with its links).

5.2.2.2 Edge management

The edge management procedure performs operations related to updating graph edges, as illustrated in Algorithm 17. The way to increase the age of edges is inspired by the fading function in the sense that the creation time of a link is taken into account. Contrary to the *fading* function, the age of links will be

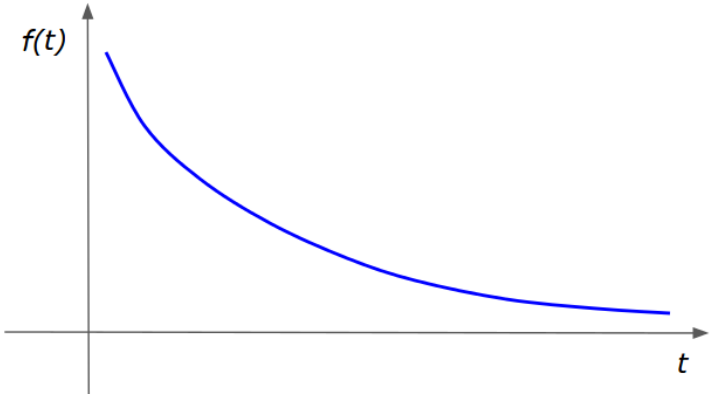


FIGURE 5.3: Plot of a fading function.

strengthened by the exponential function (Figure 5.4)

$$g(t) = 2^{\lambda_2(t-t_0)} \tag{5.3}$$

where $\lambda_2 > 0$, defines the rate of growth of the age over time, t denotes the current time and t_0 is the creation time of the edge.

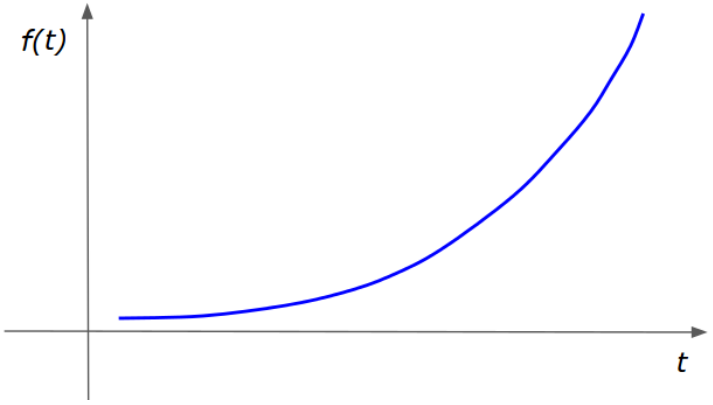


FIGURE 5.4: Plot of an exponential function.

The next step is to add a new edge that connects the two closest nodes (Figure 5.5).
The last step is to remove each link exceeding a maximum age, since these links are no longer useful because they were replaced by younger and shorter edges that were created during the graph refinement in steps 14-16, in Algorithm 19.

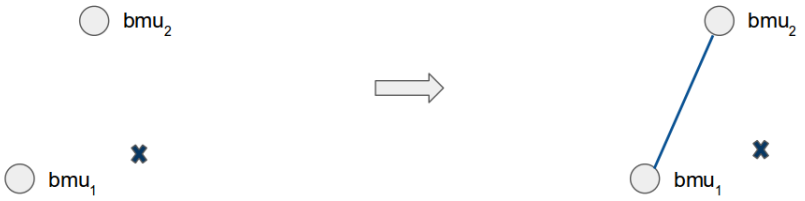


FIGURE 5.5: Edge insertion between the two nearest nodes.

Algorithm 17: Edge Management

1 Increment the age of all edges emanating from bmu_1 and weight them using Equation (5.3);
2 **if** *bmu_1 and bmu_2 are connected by an edge* **then**
3 set the age of this edge to zero
4 **else**
5 create an edge between bmu_1 and bmu_2, and mark its time stamp;
6 Remove edges whose age is greater than age_{max} (a user parameter);

5.2.2.3 Node insertion

Periodically, a node is inserted into the graph between the nodes with the largest error values according to Equation (5.4). It is also called the refinement step.

$$\mathbf{w}_r = 0.5(\mathbf{w}_q + \mathbf{w}_f) \tag{5.4}$$

Algorithm 18 with Figure 5.2 illustrate the node insertion procedure.

Algorithm 18: Node Insertion

1 Find node q with the maximum accumulated error;
2 Find the neighbor f of q with the largest accumulated error;
3 Add the new node, r, half-way between nodes q and f according to Equation (5.4);
4 Insert edges connecting the new node r with nodes q and f, and remove the original edge between q and f;

5.2.2.4 Reservoir management

The aim of using the reservoir is to hold, temporarily, the distant data points. As mentioned before, each node c has a threshold distance, δ_c. The first batch of data is assigned to their nearest nodes without comparing distances thresholds. The

distance threshold of each node is learned by taking the maximum distance of the node to the farthest point that it has been assigned. When the reservoir is full, its data is re-passed for learning (note that for those data points, we do not apply the distance threshold test, and this is to ensure that these data points will not revisit the reservoir). They are placed in the heap of the data stream, $\mathcal{X}$, to be dealt with first and the distance thresholds of nodes are updated accordingly.

5.2.2.5 Model update

In the model's update step, the nearest node, bmu_1, and its topological neighbors are moved towards the current observation $\mathbf{x}_i$. The nearest node is updated according to Equation (5.5)

$$\mathbf{w}_{bmu_1} = \mathbf{w}_{bmu_1} + \alpha_1.(\mathbf{x}_i - \mathbf{w}_{bmu_1}) \tag{5.5}$$

The topological neighbors of the nearest node are updated according to Equation (5.6)

$$\mathbf{w}_c = \mathbf{w}_c + \alpha_2.(\mathbf{x}_i - \mathbf{w}_c) \tag{5.6}$$

for all direct neighbors of the nearest node, bmu_1.

The error variable of the nearest node is updated according to Equation (5.7)

$$error(bmu_1) = error(bmu_1) + \|\mathbf{x}_i - bmu_1\|^2. \tag{5.7}$$

Table 5.1 overviews the list of parameters used in the G-Stream algorithm.

5.2.2.6 Computational complexity

The most consuming operations in Algorithm 19 are steps 4, 16, 17, and 18 with $O(k)$ time complexity each, where k is the number of nodes in the graph. Node insertion phase (step 16) is repeated $\frac{3.n}{\beta}$ times, where n is the number of data points of the data stream. Seeking the nearest node (step 4), fading function (step 16), and adjusting the error variable (step 18) phases are repeated whenever a new data point is available, i.e. n times. The other steps have a constant time complexity.

Algorithm 19: G-Stream

Data: $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$, π_{min}, λ_{age}, η, age_{max}, d
Result: set of nodes $\mathcal{C} = \{c_1, c_2, ...\}$ and their prototypes $\mathbf{W} = \{\mathbf{w}_{c_1}, \mathbf{w}_{c_2}, ...\}$

Initialize node set $\mathcal{C}$ to contain two nodes, c_1 and c_2: $\mathcal{C} = \{c_1, c_2\}$;
while *there is a data point to proceed* **do**
  Get the next data point in the data stream, $\mathbf{x}_i$;
  Find the nearest node bmu_1 and the second nearest node bmu_2;
  if $\|\mathbf{x}_i - bmu_1\| > \delta_{bmu_1}$ **then**
    put $\mathbf{x}_i$ in the reservoir;
    if *the reservoir is full* **then**
      Treat reservoir: return its data to the head of the data stream, $\mathcal{X}$, to be dealt with first;
  else
    Increment the number of points assigned to bmu_1 and mark the time stamp of $\mathbf{x}_i, t_i$;
    Add the squared distance to a local error counter variable according to Equation (5.7);
    Move bmu_1 and its topological neighbors towards $\mathbf{x}_i$ according to Equations 5.5 and 5.6 respectively;
    Update edges as described in Algorithm 17;
    if *the number of data points passed is an integer multiple of a parameter* β **then**
      for *i=1 to* η **do**
        `// creation of` η `nodes`
        Insert a new node as described in Algorithm 18;
    Apply *fading* according to Equation (5.1), delete outdated and isolated nodes;
    Finally, decrease the error of all nodes by multiplying them with a constant d;

Symbole	Description
π_{min}	the minimum weight of a node. If bellow, this node is deleted
λ_{age}	the rate of growth of the edges' age
β	the cycle interval between node insertions
η	the number of nodes to add at each iteration
age_{max}	the maximum edges's age
d	a constant for adjusting the error variable of nodes, such as $0 < d < 1$

TABLE 5.1: Parameters used in the G-Stream algorithm

Therefore, G-Stream has a complexity given by $n.(3.O(m)) + \frac{3.n}{\beta}.O(k) = n.(3 + \frac{3}{\beta}).O(k) \approx O(kn)$.

5.3 Experimental evaluations

In this section, we present an experimental evaluation of the G-Stream algorithm. We compared our algorithm with the GNG algorithm and several well-known and relevant data stream clustering algorithms, including StreamKM++, DenStream, ClusTree, and GWR. Our experiments were performed on the MATLAB platform using real-world and synthetic datasets. All the experiments are conducted on a PC with Core(TM)i7-4800MQ with two 2.70 GHz processors, and 8GB of RAM, which runs Windows 7 Professional operating system.

5.3.1 Datasets

To evaluate the clustering quality and scalability of the G-Stream algorithm both real and synthetic datasets are used. The two synthetic datasets used are DS1 and letter4. All the others are real-world publicly available datasets. Table 5.2 overviews all the datasets used.

Datasets	#observations	#features	#classes
DS1	9,153	2	14
DS2	5,458	2	13
letter4	9,344	2	7
Sea	60,000	3	2
HyperPlan	100,000	10	5
KddCup99	494,021	41	23
CoverType	581,012	54	7
Sensor	2,219,803	5	54

TABLE 5.2: Overview of all datasets.

- DS1 and DS2 are generated by http://impca.curtin.edu.au/local/software/synthetic-data-sets.tar.bz2.
- The letter4 dataset is generated by a Java code https://github.com/feldob/Token-Cluster-Generator.
- The Sea dataset was taken from http://www.liaad.up.pt/kdus/products/datasets-for-concept-drift.
- The HyperPlan dataset was taken from [Zhu, 2010].

- The real-world databases were taken from the UCI repository [Bache and Lichman, 2013], which are the KDD-CUP'99 Network Intrusion Detection stream dataset (KddCup99) and the Forest CoverType dataset (CoverType) respectively.

The algorithms are evaluated using three performance measures: accuracy (purity), Normalized Mutual Information (NMI) and Rand index [Strehl and Ghosh, 2002] (please refer to Appendix A for more details). The value of each measure lies between 0 and 1. A higher value indicates better clustering results.

5.3.2 Tuning parameter settings

As shown in Table 5.1 and Algorithms 16 and 19, the GNG-online and G-Stream algorithms require some tuning parameters. We slightly varied these parameters and have empirically chosen those giving the best values. The parameters $age_{max} = 250$ (the maximum age of edges), $\pi_{min} = 2$ (the minimum weight of nodes), $\eta = 3$ (the number of nodes inserted at a time) are fixed for all datasets.

Table 5.3 gathers the values for the other parameters α_1: the winning node adaptation factor; α_2: the winning node, neighbor adaptation factor; β: the cycle interval between node insertions; λ_1: the decay factor in the fading function; λ_2: the strength factor in weighting edges; $|window|$: the size of the sliding window; $|reservoir|$: the reservoir size.

Datasets	α_1	α_2	β	λ_1	λ_2	$\|window\|$	$\|reservoir\|$
DS1	0.01	0.0005	250	0.4	0.4	600	300
DS2	0.01	0.0005	250	0.4	0.4	600	300
letter4	0.01	0.0005	250	0.4	0.4	600	300
Sea	0.01	0.001	400	0.2	0.2	1000	300
HyperPlan	0.01	0.001	300	0.2	0.2	600	550
KddCup99	0.1	0.01	300	0.2	0.2	1000	400
CoverType	0.1	0.01	250	0.2	0.2	1000	400
Sensor	0.1	0.01	300	0.2	0.2	800	400

TABLE 5.3: Tuning parameter settings.

5.3.3 Evaluation and performance comparison

This section aims to evaluate the clustering quality of the G-Stream and compare it to well-known data stream clustering algorithms, as well as the GNG-online

algorithm. As explained in section 5.2, the GNG and G-Stream algorithms start with two nodes. We used an online version of GNG but without the parameters that we added expressly to show the interest and contribution of these parameters in G-Stream. Therefore, we carried out experiments by initializing two nodes randomly among the first 20 points and we repeated this 10 times.

For comparison purposes, we used DenStream [Cao et al., 2006] and ClusTree [Kranen et al., 2011] from the **stream** R package [Bolanos et al., 2014]. Comparison is also performed with StreamKM++ [Ackermann et al., 2012] (this latter algorithm was coded in the C language) and GWR [Marsland et al., 2002] (the MATLAB code of the algorithm is provided in `http://seat.massey.ac.nz/personal/s.r.marsland/gwr.html`). StreamKM++ was evaluated by choosing randomly the seed node among the first 20 points. DenStream was evaluated by performing a variant of the DBSCAN algorithm in the offline step. ClusTree was evaluated by performing the k-means algorithm in the offline step by setting the k parameter to 10 (all these algorithms are presented in chapter 4).

All experiments were repeated 10 times and the results (the average value with its standard deviation) are reported in Tables 5.4, 5.5, and 5.6, which report results in terms of accuracy, NMI, and Rand index respectively.

In Table 5.4, it is noticeable that G-Stream's accuracies are higher for all datasets as compared to StreamKM++, DenStream, CluStree and GWR, except for GWR for the letter4, Sea, and HyperPlan datasets. We recall that GWR makes several iterations on data while all other algorithms (including G-Stream) make just one pass over the data.

In Table 5.5, we can see that the NMI values of G-Stream are higher than the other algorithms except for DenStream for the Sea dataset, GWR for the HyperPlan dataset, and ClusTree for the sensor dataset.

Table 5.6 shows that the Rand index values of G-Stream are higher than the other algorithms except for StreamKM++ for the Sea dataset and GWR for the HyperPlan dataset. We recall that G-Stream proceeds in a single phase whereas StreamKM++, DenStream and ClusTree proceed in two phases (online and offline phase), and GWR proceeds in several iterations.

Figure 5.6 compares the G-Stream algorithm (red line with circle) with the GNG-online algorithm (blue line with cross) with respect to accuracy for the DS1, DS2, letter4, HyperPlan, and Sea datasets. The data points are passed based on the sliding windows principle (Figure 5.16). We recall that the number of the

Datasets	**G-Stream**	StreamKM++	DenStream	ClusTree	GWR
DS1	**0.9809 ±0.0061**	0.6754 ±0.0183	0.7740 ±0.0000	0.6864 ±0.0275	0.5720 ±0.0350
DS2	**0.8632 ±0.0075**	0.6261 ±0.0360	0.7190 ±0.0000	0.6220 ±0.0000	0.6664 ±0.0216
letter4	0.9832 ±0.0050	0.6871 ±0.0263	0.8110 ±0.0000	0.8110 ±0.0000	**0.9856 ±0.0050**
Sea	0.8386 ± 0.0021	0.7886 ±0.0091	0.8240 ±0.0001	0.8224 ±0.0065	**0.8467 ±0.0009**
HyperPlan	0.4238 ±0.0021	0.3966 ±0.0055	0.4250 ±0.0000	0.4380 ±0.0089	**0.4402 ±0.0006**
KddCup99	**0.9805 ±0.0050**	0.6922 ±0.1140	0.9544 ±0.0031	0.8182 ±0.1304	0.9161 ±0.0017
CoverType	**0.6085 ±0.0087**	0.5266 ±0.0074	0.5850 ±0.0011	0.5850 ±0.0000	0.6030 ±0.0000
Sensor	**0.0834 ±0.0002**	0.0561 ±0.0014	0.0660 ±0.0000	0.0790 ±0.0000	0.0726 ±0.0000

TABLE 5.4: Comparing G-Stream with different algorithms in terms of accuracy.

Datasets	**G-Stream**	StreamKM++	DenStream	ClusTree	GWR
DS1	**0.7289 ±0.0113**	0.7021 ±0.0209	0.6973 ±0.0000	0.7064 ±0.0168	0.5697 ±0.0285
DS2	**0.6700 ±0.0054**	0.6242 ±0.0182	0.6228 ±0.0000	0.6231 ±0.0000	0.5481 ±0.0211
letter4	**0.6265 ±0.0064**	0.5532 ±0.0219	0.1637 ±0.0000	0.2425 ±0.0000	0.5767 ±0.0029
Sea	0.1380 ±0.0009	0.1463 ±0.0042	**0.1646 ±0.0000**	0.1583 ±0.0095	0.1331 ±0.0006
HyperPlan	0.0186 ±0.0009	0.0103 ±0.0023	0.0208 ±0.0000	0.0170 ±0.0042	**0.0256 ±0.0002**
KddCup99	**0.6670 ±0.0089**	0.3926 ±0.2815	0.6290 ±0.0300	0.5724 ±0.2974	0.6315 ±0.0003
CoverType	**0.1403 ±0.0029**	0.0874 ±0.0086	0.0475 ±0.0201	0.0362 ±0.0042	0.1411 ±0.0000
Sensor	0.1154 ±0.0012	0.0795 ±0.0038	0.3087 ±0.0000	**0.3238 ±0.0000**	0.0942 ±0.0000

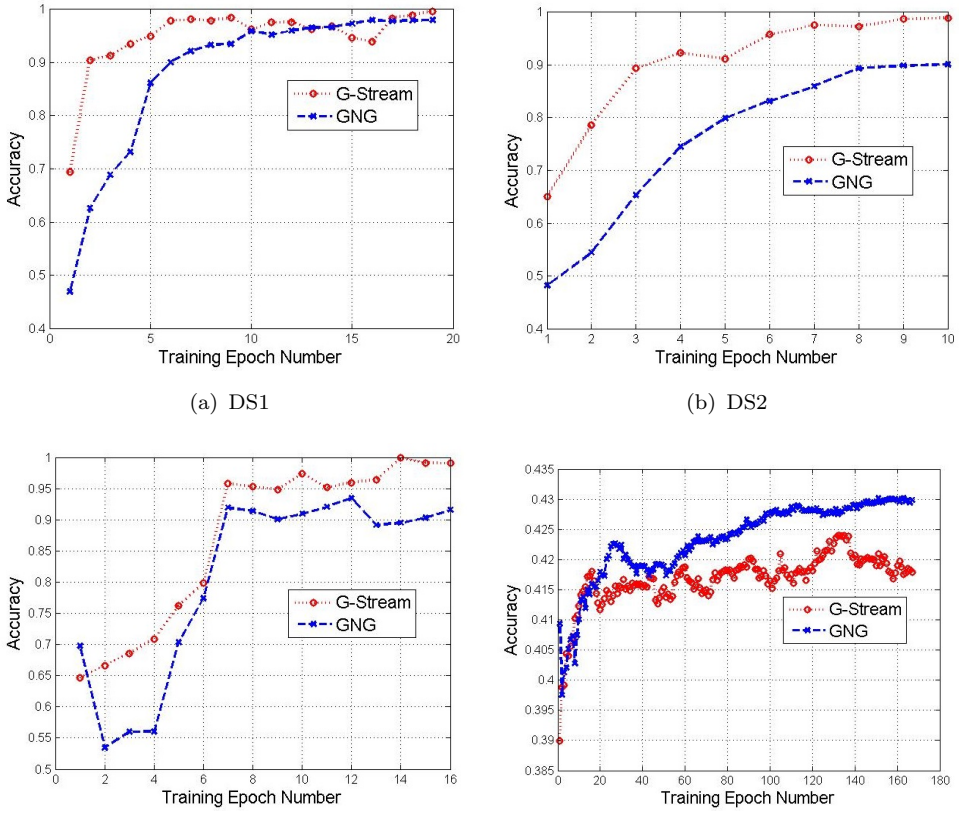
TABLE 5.5: Comparing G-Stream with different algorithms in terms of NMI.

window to which a data point belongs is used as the time-stamp of the concerned data point. After passing each window, we calculate the accuracy of the concerned algorithm (G-Stream or GNG-online).

For almost all cases, the accuracy value of G-Stream is higher than for GNG-online. Indeed, for DS2 and letter4 datasets, the accuracy values of G-Stream are

Datasets	**G-Stream**	StreamKM++	DenStream	ClusTree	GWR
DS1	**0.8530 ±0.0024**	0.8443 ±0.0048	0.8491 ±0.0000	0.8442 ±0.0066	0.8050 ±0.0137
DS2	**0.8698 ±0.0007**	0.8533 ±0.0074	0.8607 ±0.0000	0.8505 ±0.0000	0.8431 ±0.0093
letter4	**0.8156 ±0.0015**	0.7941 ±0.0145	0.5019 ±0.0000	0.5514 ±0.0000	0.8086 ±0.0007
Sea	0.4707 ±0.0001	**0.5072 ±0.0016**	0.4700 ±0.006	0.4917 ±0.0034	0.4689 ±0.0001
HyperPlan	0.7042 ±0.0008	0.6674 ±0.0004	0.6038 ±0.0000	0.6529 ±0.0016	**0.7061 ±0.0001**
KddCup99	**0.8380 ±0.0036**	0.6339 ±0.2316	0.8164 ±0.0106	0.8289 ±0.1798	0.8305 ±0.0006
CoverType	**0.6231 ±0.0008**	0.6106 ±0.0018	0.4604 ±0.0070	0.5080 ±0.0005	0.6216 ±0.0000
Sensor	**0.9592 ±0.0010**	0.9143 ±0.0076	0.3481 ±0.0000	0.3082 ±0.0000	0.8373 ±0.0000

TABLE 5.6: Comparing G-Stream with different algorithms in terms of Rand index.



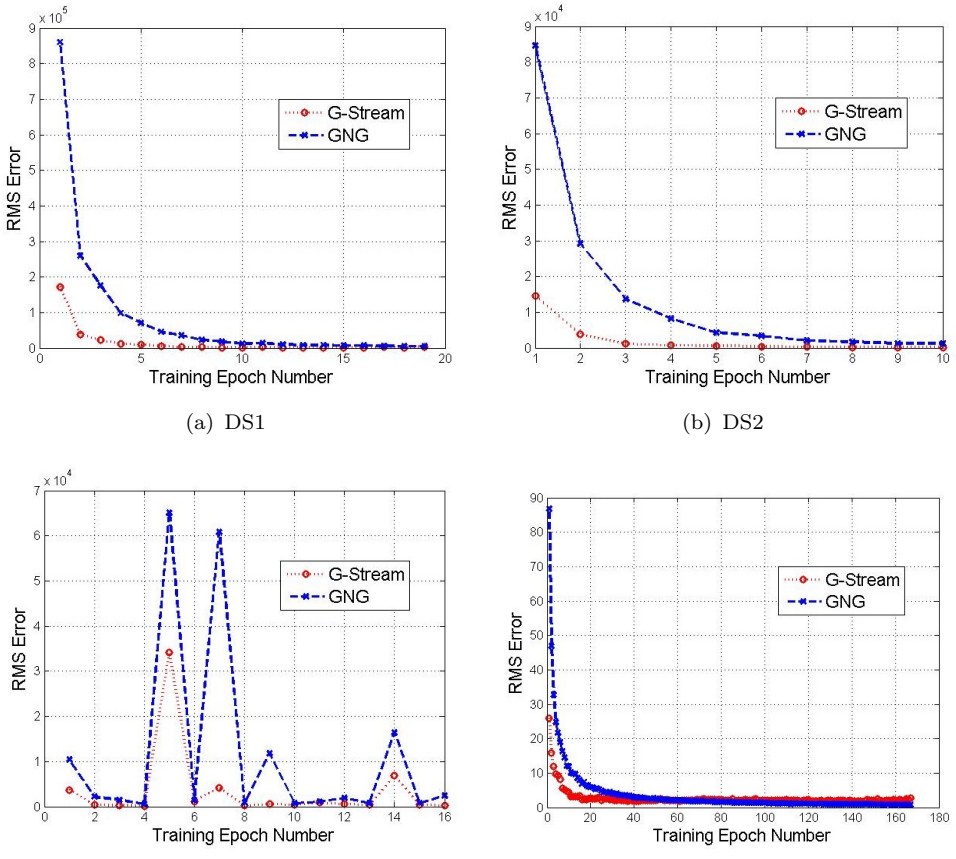
(a) DS1 (b) DS2

(c) letter4 (d) HyperPlan

FIGURE 5.6: Accuracy for G-Stream and GNG-online.

higher than the ones of GNG-online for all windows. Also, the G-Stream accuracy values are higher than for GNG-online on DS1 and Sea datasets except for some windows. For the HyperPlan dataset, we can see that the GNG-online algorithm exceeds the G-Stream algorithm in terms of accuracy. This can be explained by the number of nodes created by the GNG-online algorithm which is higher than the one of the G-Stream algorithm for the last times. Note that the number of nodes created by the G-Stream algorithm could be adjusted by the decay factor (λ_1) and the interval nodes insertion (β) parameters.

An important and widely used measure of resolution, the quantization error [Kohonen et al., 2001], is computed. Figure 5.7 compares the two algorithms in terms of RMS error (red line with circle for the G-Stream algorithm, blue line with cross for the GNG-online algorithm). For all windows, the G-Stream algorithm has lower values of the RMS error than those of the GNG-online algorithm on all datasets.

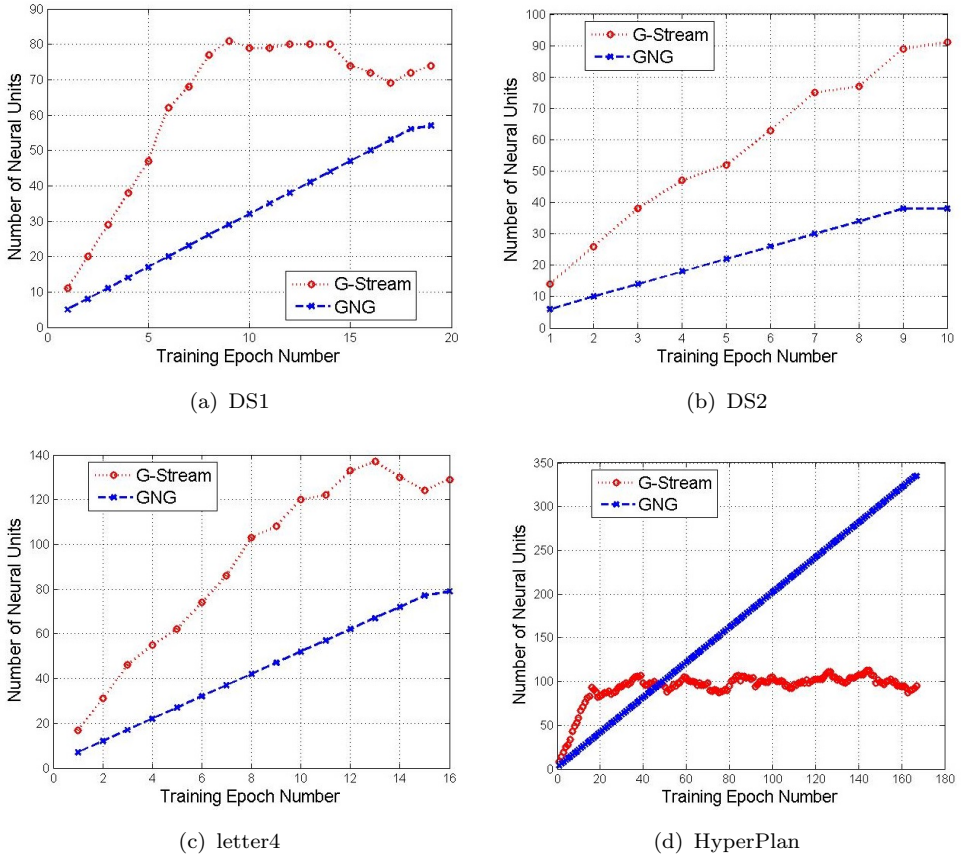


(a) DS1 (b) DS2

(c) letter4 (d) HyperPlan

FIGURE 5.7: RMS error for G-Stream and GNG-online.

Figure 5.8 compares the two algorithms (G-stream and GNG-online) in terms of the number of nodes creating the graph (red line with circle for the G-Stream algorithm, blue line with cross for the GNG-online algorithm). Despite that we create several nodes at each iteration (against a single node for GNG-online), the number of nodes created by G-Stream becomes steady (against a continuous increase for GNG-online) due to the application of the fading function.



(a) DS1 (b) DS2

(c) letter4 (d) HyperPlan

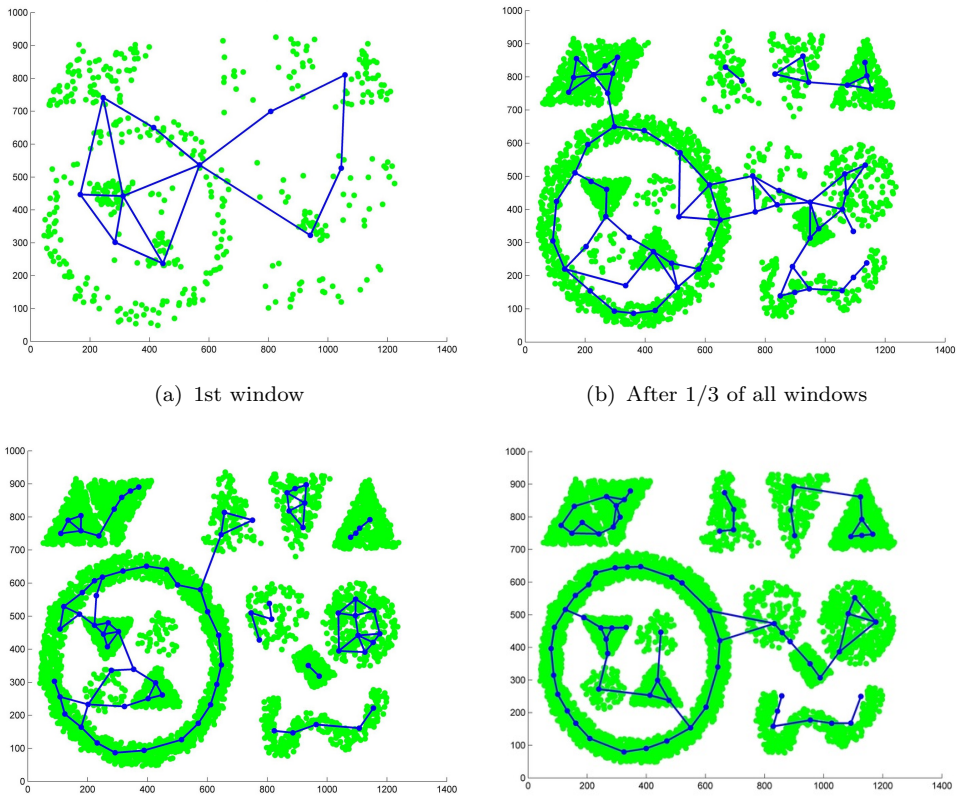
FIGURE 5.8: Number of nodes for G-Stream and GNG-online.

5.3.4 Visualization

Figures 5.9, 5.10, 5.11 show the evolution of the node creation by applying G-Stream on the DS1, DS2, and letter4 datasets respectively (green points represent data points of the data stream and blue points are nodes of the graph with edges in blue lines).

The first sub-figure (of Figures 5.9, 5.10, 5.11) represents the intermediate graph after seeing the first window's data points. The second (resp. third) sub-figure represents the intermediate graph after seeing 1/3 (resp. 2/3) of all windows.

The last sub-figure represents the final graph. These figures illustrate that the G-Stream algorithm manages to recognize the structures of the data stream and can separate these structures with an optimal visualization.



(a) 1st window (b) After 1/3 of all windows

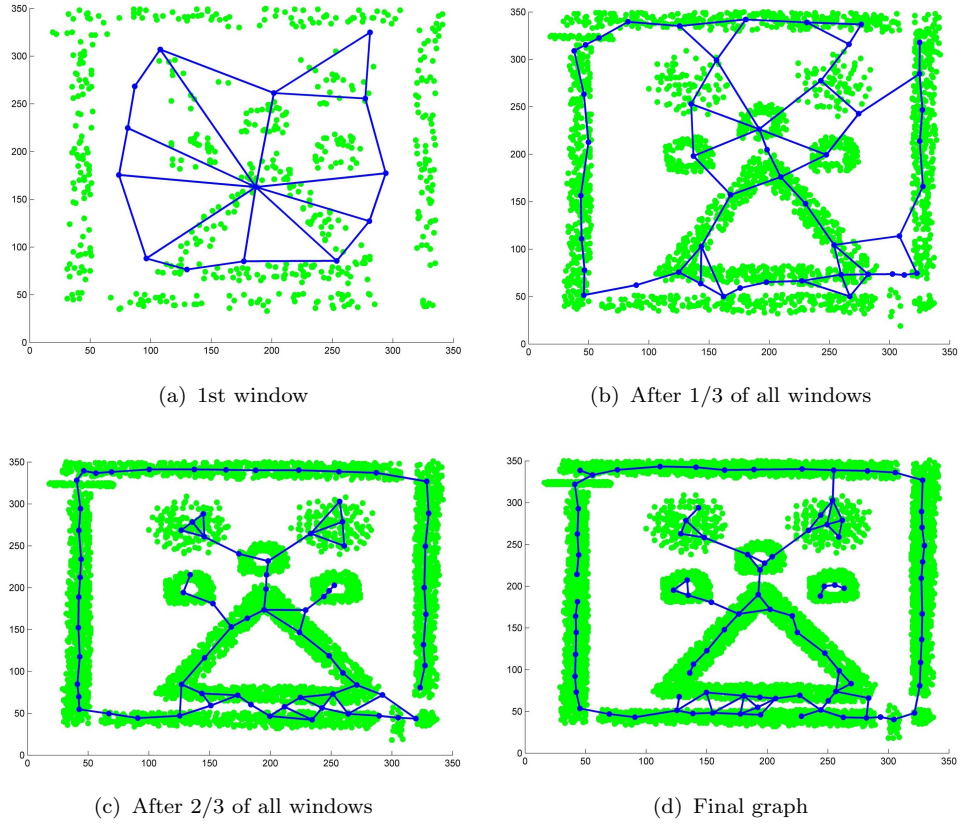
(c) After 2/3 of all windows (d) Final graph

FIGURE 5.9: Evolution of graph creation of G-Stream on DS1 (dataset and topological result). The intermediate graph after seeing the first window's data points; the 1/3 of all windows; the 2/3 of all windows; and the final graph.

Figure 5.12 compares the G-Stream algorithm with the GNG-online algorithm on the 2-dimensional (DS1, DS2, and letter4) datasets, in terms of visual results (i.e., the final graph created by the G-Stream/GNG-online algorithm for each dataset). As illustrated in these figures, we can see that the GNG-online algorithm maintains many spurious edges connecting nodes belonging to different clusters, while the G-Stream algorithm mainains fewer of these inter-cluster edges.

5.3.5 Evolving data streams

In this subsection, we perform the G-Stream algorithm on different data streams ordered by class labels to demonstrate its effectiveness in clustering evolving data



(a) 1st window (b) After 1/3 of all windows

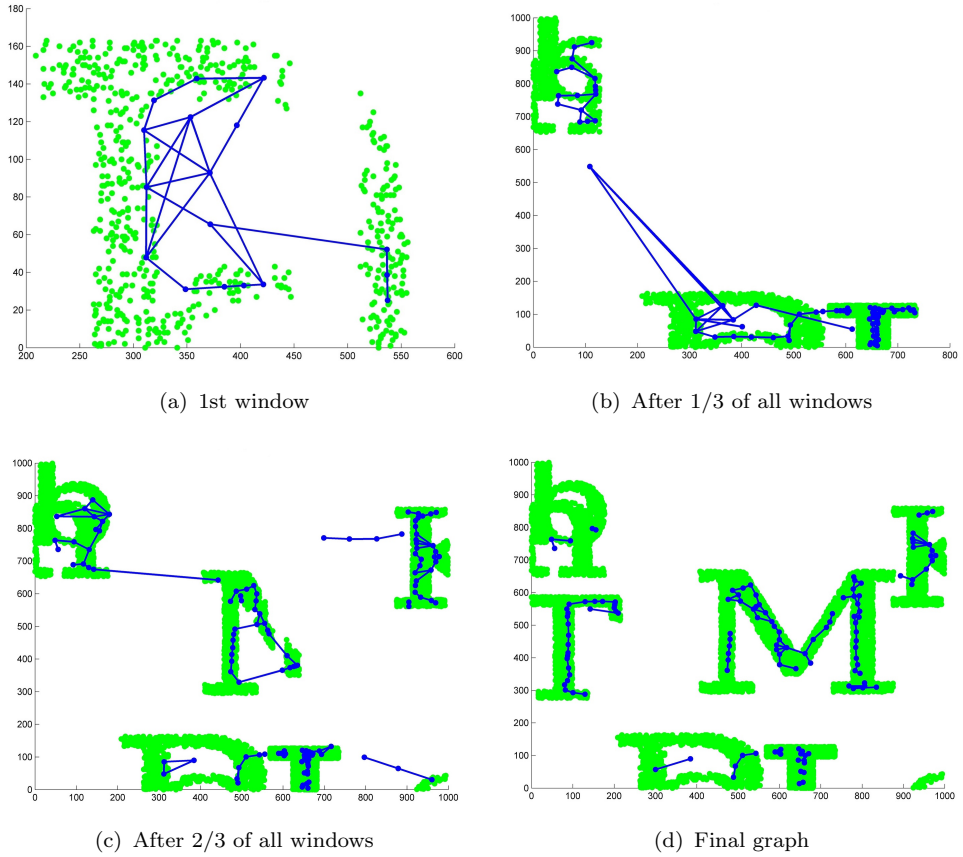
(c) After 2/3 of all windows (d) Final graph

FIGURE 5.10: Evolution of graph creation of G-Stream on DS2 (dataset and topological result). The intermediate graph after seeing the first window's data points; the 1/3 of all windows; the 2/3 of all windows; and the final graph.

streams (i.e., data points of the first class arrive in first, then the ones of the second, third, etc. class). In this case, old concepts (class labels) disappear due to the use of fading function. In the same time, new concepts (class labels) appear as new data points arrive. Note that the class labels are not known to the clustering algorithm. We use the same experimental protocol as described in section 5.3.3, i.e., we did experiments by initializing two nodes randomly among the first 20 points, we repeated this 10 times, and we report the average value with its standard deviation in Figures 5.13, 5.14, 5.15.

Figure 5.13 compares the G-Stream algorithm, in terms of the accuracy, with and without ordering of class labels. It shows that the G-Stream algorithm with ordering of classes can find clusters with accuracy values as comparable to those without ordering of classes.

Figure 5.14 compares the G-Stream algorithm, in terms of the NMI, with and without ordering of class labels. It shows that the G-Stream algorithm with ordering of classes can find clusters with NMI values as comparable to those without



(a) 1st window (b) After 1/3 of all windows

(c) After 2/3 of all windows (d) Final graph

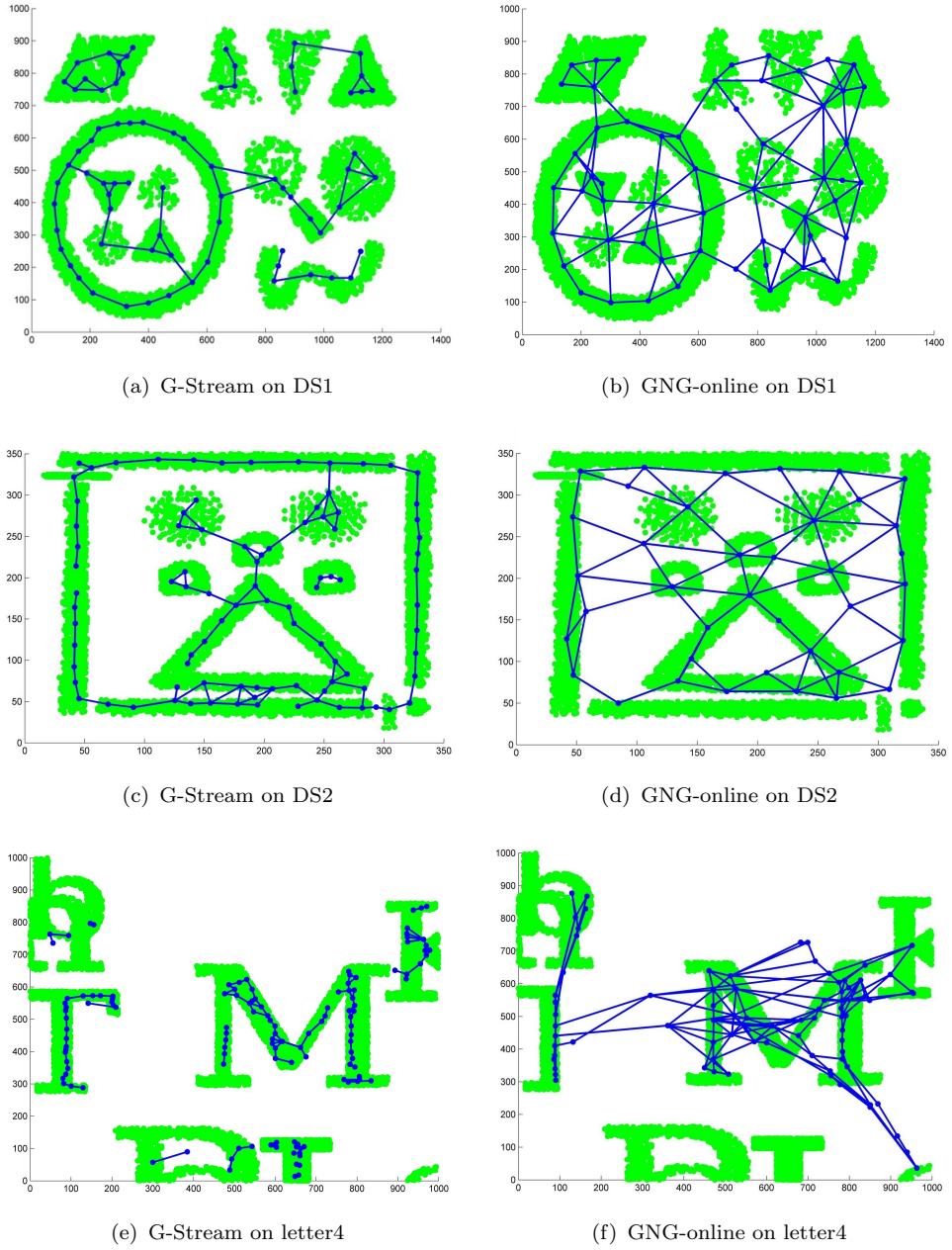
FIGURE 5.11: Evolution of graph creation of G-Stream on letter4 (dataset and topological result). The intermediate graph after seeing the first window's data points; the 1/3 of all windows; the 2/3 of all windows; and the final graph.

ordering of classes for most datasets. Although, we can see that the values of NMI bend down in the case where we sort the data points based on their class labels, for the DS1, the DS2 and the KddCup99 datasets.

Figure 5.15 compares the G-Stream algorithm, in terms of the Rand index, with (i.e., we sort the data points based on their class labels) and without ordering of class labels. Except for the KddCup99 dataset where the Rand index value decreases in the case where the data points are sorted, this Figure shows that the G-Stream algorithm with ordering of classes can find clusters with Rand index values as comparable to those without ordering of classes for most datasets.

5.3.6 Clustering over sliding windows

In many applications, the most recent N observations are considered to be more critical and preferable [Zhou et al., 2008]. Therefore, clustering data streams over



(a) G-Stream on DS1 (b) GNG-online on DS1

(c) G-Stream on DS2 (d) GNG-online on DS2

(e) G-Stream on letter4 (f) GNG-online on letter4

FIGURE 5.12: Visual result comparison of G-Stream with GNG-online (dataset and topological result). The final graph created by the G-Stream/GNG-online algorithm.

sliding windows is a natural choice and becomes one of the most popular models.

The idea behind the sliding window model is to perform detailed analysis (the clustering process) over both the most recent data points and the model obtained from the old ones [Amini et al., 2014]. In the sliding window model, data points arrive continually, and at the period (or window) t, we consider N recent data points that contain some data points of window $t-1$. Figure 5.16 illustrates the principle of the sliding windows model.

FIGURE 5.13: Accuracy of G-Stream with and without ordering of classes.

FIGURE 5.14: NMI of G-Stream with and without ordering of classes.

The data in a window can formally be written as $(\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N)$, where a single observation $\mathbf{x}_i$ is d-dimensional. In our sliding windows model, the evolution of data items can be presented as follows:

$$Data^{window_t} = \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_M, \mathbf{x}_{M+1}, \mathbf{x}_{M+2}, ..., \mathbf{x}_{N-1}, \mathbf{x}_N$$

$$Data^{window_{t+1}} = \mathbf{x}_M, \mathbf{x}_{M+1}, \mathbf{x}_{M+2}, ..., \mathbf{x}_{N-1}, \mathbf{x}_N, \mathbf{x}_{N+1}, ..., \mathbf{x}_{N+M-1}, \mathbf{x}_{N+M}$$

This means that the M oldest data points are removed and M new data points are appended. Hence, the percentage of overlap between two windows is defined as $(M/N)\%$. To assess the G-Stream algorithm while adopting sliding windows, we vary the percentage of overlap between two windows, i.e., $(M/N)\% =$

FIGURE 5.15: Rand index of G-Stream with and without ordering of classes.

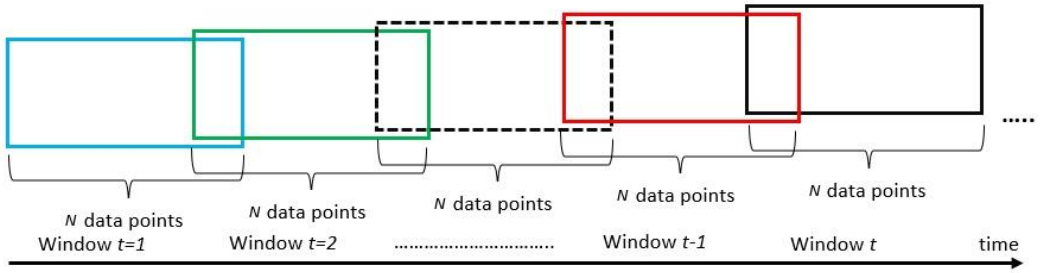


FIGURE 5.16: Analysis on the sliding windows model

$0\%, 25\%, 50\%$, and 75% and we report the results in Tables 5.7, 5.8, and 5.9. For this experiment, we set the values of the parameters as follows: $\alpha_1 = 0.001$, $\alpha_2 = 0.00001$, $\lambda_1 = \lambda_2 = 0.4$, $\beta = 300$, $N = 600$, $|reservoir| = 400$.

In Table 5.7, we can see that the accuracy grows as the percentage of overlap between windows increases. This is especially evident for the relatively small and medium sized datasets. For very large datasets, we can see a stability in the values of the accuracy (this is due to the use of the fading function allows to remove outdated nodes). The same result can be shown in the case of the NMI values. However, the Rand index values do not significantly grow while increasing the percentage of overlap between two consecutive windows.

5.3.7 Execution time

The efficiency of algorithms is measured by their execution time. Referring to the computational complexity we calculated in section 5.2.2, the execution time

Datasets	0%	25%	50%	75%
DS1	0.7013	0.7498	0.8390	0.9069
DS2	0.6136	0.6486	0.6926	0.7466
letter4	0.9007	0.9379	0.9634	0.9781
Sea	0.8286	0.8224	0.8215	0.8291
HyperPlan	0.4274	0.4096	0.4196	0.4207
KddCup99	0.9605	0.9810	0.9803	0.9819
CoverType	0.5343	0.5411	0.5386	0.5301
Sensor	0.0787	0.0796	0.0796	0.0799

TABLE 5.7: Accuracy of G-Stream while changing the overlap percentage of sliding windows.

Datasets	0%	25%	50%	75%
DS1	0.6440	0.6633	0.6898	0.7124
DS2	0.5354	0.5300	0.5630	0.5906
letter4	0.6224	0.6720	0.6681	0.6380
Sea	0.1409	0.1377	0.1394	0.1422
HyperPlan	0.0208	0.0131	0.0153	0.0186
KddCup99	0.6258	0.7237	0.7079	0.6824
CoverType	0.0896	0.1028	0.0865	0.0835
Sensor	0.1070	0.1114	0.1067	0.1145

TABLE 5.8: NMI of G-Stream while changing the overlap percentage of sliding windows.

Datasets	0%	25%	50%	75%
DS1	0.8352	0.8416	0.8511	0.8594
DS2	0.8366	0.8467	0.8496	0.8592
letter4	0.8194	0.8436	0.8422	0.8223
Sea	0.4764	0.4746	0.4765	0.4749
HyperPlan	0.7017	0.7012	0.7015	0.7011
KddCup99	0.8187	0.8614	0.8597	0.8399
CoverType	0.6197	0.6195	0.6199	0.6185
Sensor	0.9596	0.9607	0.9612	0.9616

TABLE 5.9: Rand index of G-Stream while changing the overlap percentage of sliding windows.

strongly depends on the number of nodes comprising the graph and the size of the data stream. We recall that G-Stream is implemented in MATLAB, and SVStream is the only MATLAB program that we have (the other algorithms are implemented in Java, R, or C languages).

Figure 5.17 shows the execution time of G-Stream and that of SVStream. We can see that both the execution time of G-Stream and SVStream grow as the size of the data stream grows, and G-Stream is more efficient than SVStream. Note

that we could not obtain the execution time of the SVStream algorithm for the Sensor dataset since it did not stop. Whereas the execution time of the G-Stream algorithm for the Sensor dataset was 21926 seconds, taking the parameters as follows: $\alpha_1 = 0.01$, $\alpha_2 = 0.001$, $\lambda_1 = \lambda_2 = 0.4$, $\beta = 300$, $|window| = 600$, $|reservoir| = 400$.

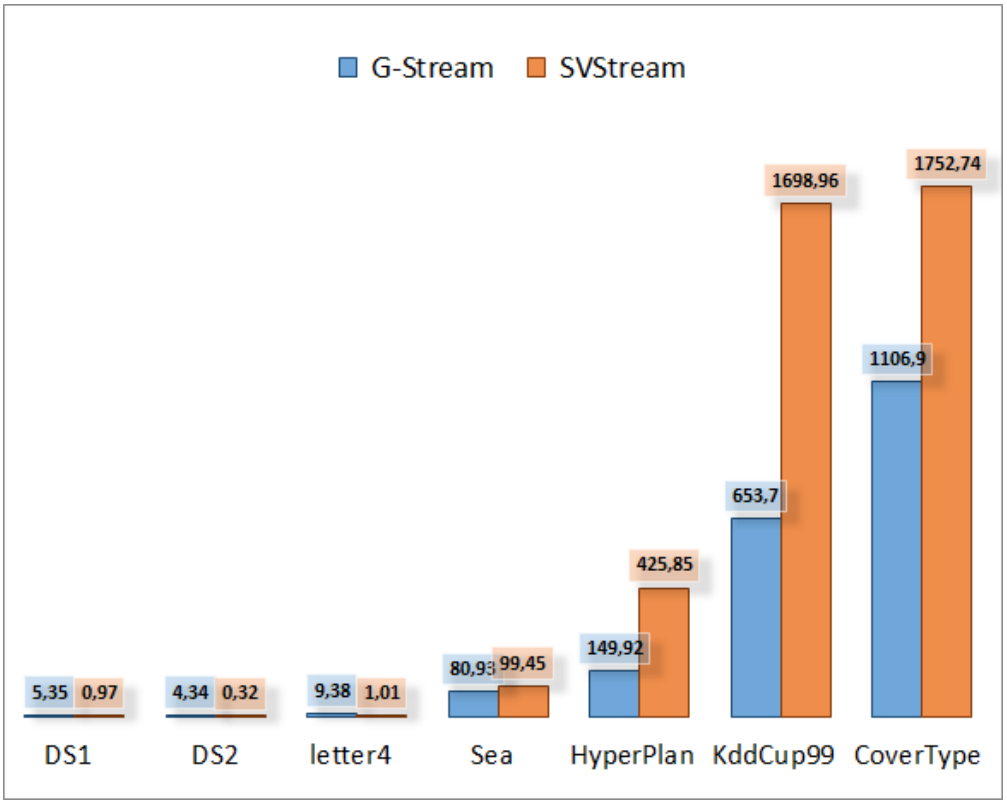


FIGURE 5.17: Execution time (in seconds)

5.4 Conclusion

In this chapter, we have proposed G-Stream, an efficient method for topological clustering an evolving data stream in an online manner, which allows on-the-fly cluster creation. It can be used as an online component in the on/offline framework takes into account the topology of the data to find the final clusters.

In G-Stream, the nodes are weighted using a fading function, and the edges using an exponential function. Starting with two nodes, G-Stream compares the arriving data points to the current prototypes, storing the very distant ones in a reservoir, learns the threshold distances automatically, and many nodes are created at the same time.

Experimental evaluation for a number of real and synthetic datasets demonstrates the effectiveness and efficiency of G-Stream in discovering clusters of arbitrary shape. Our experiments show that G-Stream outperforms the GNG-online algorithm in terms of visual results and clustering quality criteria such as accuracy, the Rand index and NMI. Its performance as compared to three relevant data stream algorithms are promising.

The work presented in this chapter has resulted in the following publications:

- Mohammed Ghesmoune, Mustapha Lebbah, and Hanane Azzag. A new growing neural gas for clustering data streams. Neural Networks, 78:36–50, 2016. ISSN 0893-6080. doi: http://dx.doi.org/10.1016/j.neunet.2016.02.003. URL http://www.sciencedirect.com/science/article/pii/S0893608016000289. Special Issue on "Neural Network Learning in Big Data".

- Mohammed Ghesmoune, Mustapha Lebbah, and Hanane Azzag. Clustering over data streams based on growing neural gas. In Advances in Knowledge Discovery and Data Mining - 19th Pacific-Asia Conference, PAKDD 2015, Ho Chi Minh City, Vietnam, May 19-22, 2015, Proceedings, Part II, pages 134–145, 2015c. doi: 10.1007/978-3-319-18032-8 11. URL http://dx.doi.org/10.1007/978-3-319-18032-8_11.

- Mohammed Ghesmoune, Hanane Azzag, and Mustapha Lebbah. G-Stream: Growing neural gas over data stream. In Neural Information Processing - 21st International Conference, ICONIP 2014, Kuching, Malaysia, November 3-6, 2014. Proceedings, Part I, pages 207–214, 2014. doi: 10.1007/978-3-319-12637-1 26. URL http://dx.doi.org/10.1007/978-3-319-12637-1_26.

- Mohammed Ghesmoune, Mustapha Lebbah and Hanane Azzag. G-Stream: une approche incrémentale pour le clustering de flux de données. In SFC 2015, 09-11 Septembre 2015, Nantes.

- Mohammed Ghesmoune, Hanane Azzag and Mustapha Lebbah. Une nouvelle méthode topologique pour le clustering de flux de données. In COSI 2015, Coloque sur l'optimisation et les systèmes d'information, Oran, 01-03 Juin 2015.

- Mohammed Ghesmoune, Mustapha Lebbah, and Hanane Azzag. Clustering topologique pour le flux de données. In 15èmes Journées Francophones Extraction et Gestion des Connaissances, EGC 2015, 27-30 Janvier 2015, Luxembourg, pages 137–142, 2015a. URL http://editions-rnti.fr/?inprocid=1002072.

In the next chapter, we will present in details the second contribution which is a micro-batching GNG-based data stream clustering algorithm. This algorithm is implemented using Spark Streaming.

Chapter 6

Micro-Batching Growing Neural Gas for Clustering Data Streams

In this chapter, we will introduce our second contribution about the "batchStream" algorithm for streaming data clustering. In contrast to the G-Stream introduced in chapter 5 which is a sequential GNG algorithm, batchStream is a scalable, distributed algorithm. We start by defining a new cost function taking into account the subsets of observations arriving in batches. After that, we propose a model for scalability. This model consists of decomposing the data stream clustering problem into the elementary functions, Map and Reduce. Its implementation is assured in the Spark Streaming platform.

6.1 Introduction

As in the previous chapter, we consider in the following, clustering multi-dimensional data in the form of a stream, i.e., a sequence of potentially infinite, non-stationary data arriving continuously where random access to data is not feasible and storing all arriving data is impractical. When applying data mining techniques, or more specifically clustering algorithms, to data streams, restrictions in execution time and memory have to be considered carefully. To deal with time and memory restrictions, many of existing data stream clustering algorithms use the two-phase framework proposed in Aggarwal et al. [2003].

Velocity, which refers to the rate that Big Data are generated at high speed (speed of data in and out), is an important dimension (or concept) of the Big Data domain Demchenko et al. [2013]. Currently, Spark Streaming Zaharia et al. [2013] and Apache Flink Schelter et al. [2013] may be considered as the most widely used streaming platforms. These distributed streaming systems, as presented in chapter 2, are based on two processing models, *record-at-a-time* and *micro-batching*. In a *record-at-a-time* processing model, long-running stateful operators process records as they arrive, update the internal state, and send out new records. On the other hand, the *micro-batching* processing model runs each streaming computation as a series of deterministic batch computations on small time intervals. Among the available frameworks that implements the *micro-batching* processing model, we can find Spark Streaming. It is an extension of the core Spark API[1] that enables high-throughput, reliable processing of live data streams.

In the previous chapter 5, G-Stream was presented as a data stream clustering approach based on the Growing Neural Gas algorithm. G-Stream uses a stochastic approach to update the prototypes, and it was implemented on a "centralized" platform. In this chapter, we propose **batchStream**, a novel distributed algorithm for discovering clusters of arbitrary shape in an evolving data stream. The batchStream algorithm is implemented on a distributed streaming platform based on the *micro-batching* processing model, i.e., the Spark Streaming API[2]. In the proposed algorithm, the topological structure is represented by a graph wherein each node represents a cluster, which is a set of "close" data points and neighboring nodes (clusters) are connected by edges. Starting with only two nodes, the graph size is not fixed but may also evolve as several nodes (clusters) are created in each iteration. We use an exponential fading function to reduce the impact of old data whose relevance diminishes over time. For the same reason, links between nodes are also weighted by an exponential function.

The data received in each interval is stored reliably across the cluster to form an input dataset for that interval. Once the time interval is completed, this dataset is processed via deterministic parallel and distributed operations, such as *Map* and *Reduce* to produce new datasets representing either program outputs or intermediate states Zaharia et al. [2013]. The input data is split and the master assigns the splits to the Map workers. Each worker processes the corresponding input split, generates key/value pairs and writes them to intermediate files (on disk

[1] http://spark.apache.org/
[2] http://spark.apache.org/streaming/

or in memory). The Reduce function is responsible for aggregating information received from the Map functions.

6.2 Micro-batching clustering

In this section we introduce Micro-Batching Growing Neural Gas for Clustering Data Streams (batchStream) and highlight some of its novel features. The batchStream algorithm is based on Growing Neural Gas (GNG), which is, as presented in chapter 3, an incremental self-organizing approach that belongs to the family of topological maps such as Self-Organizing Maps (SOM) Kohonen et al. [2001] or Neural Gas (NG) Martinetz and Schulten [1991]. It is an unsupervised algorithm capable of representing a high dimensional input space in a low dimensional feature map. Typically, it is used for finding topological structures that closely reflect the structure of the input distribution.

We assume that the data stream consists of a sequence $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$ of n (potentially infinite) elements of a data stream arriving at times $t_1, t_2, ..., t_n$, where $\mathbf{x}_i = (x_i^1, x_i^2, ..., x_i^d)$ is a vector in $\mathbb{R}^d$. We denote by $\mathbf{X}_1 = \{\mathbf{x}_1, ..., \mathbf{x}_p\}$ where p is the size of the window, thus $\mathcal{X} = \{\mathbf{X}_1, \mathbf{X}_2, ..., \mathbf{X}_L\}$. At each time, batchStream is represented by a graph $\mathcal{C}$ where each node represents a cluster. Each node $c \in \mathcal{C}$ has

- a prototype $\mathbf{w}_c = (w_c^1, w_c^2, ..., w_c^d)$ representing its position;
- π_c representing the weight of this node;
- $error(c)$ an error variable representing the sum of distances between this node and the data-points assigned to it.

When data arrive in a stream, we may want to estimate clusters dynamically, updating them as new data arrive. An implementation of a Growing Neural Gas algorithm over Data Stream on a "centralized" platform would be as follows Ghesmoune et al. [2014, 2015b]: Starting with two nodes, and as a new data point is reached, the nearest and the second-nearest nodes are identified, linked by an edge, and the nearest node with its topological neighbors are moved toward the data point. Each node has an accumulated error variable and a weight which varies over time using a fading function.

Using an edge management procedure, one, two or three nodes are inserted into the graph between the nodes with the largest error values. A new node r is inserted into the graph between the nodes with the largest error values, q and f, according to Equation (6.1). It is also called the refinement step.

$$\mathbf{w}_r = 0.5(\mathbf{w}_q + \mathbf{w}_f) \tag{6.1}$$

Nodes can also be removed if they are identified as being superfluous.

However, the "naive" design of a distributed version of G-Stream (presented in the previous chapter 5) would raise difficulties, which are resolved by batchStream. It operates with parameters to control the decay (or "forgetfulness") of the estimates. The algorithm uses a generalization of the mini-batch GNG update rule. In the adaptation step of the GNG algorithm, the nearest node and all of its neighbors are moved in the direction of the data point.

To incorporate the scheme of mini-batch learning, we first define the objective (or cost) function for online clustering for a fixed topology as follows:

$$\mathcal{J}^{(t+1)}_{batchStream}(\phi, \mathcal{W}) = \sum_{\mathbf{x}_i \in \mathcal{X}^{(t+1)}} \sum_{c_j \in C} \mathcal{K}^T\big(\delta(c_j, \phi(\mathbf{x}_i))\big) \|\mathbf{x}_i - \mathbf{w}^{(t+1)}_{c_j}\|^2 \tag{6.2}$$

where $\mathcal{X}^{(t+1)} = \{\mathbf{X}_1, \mathbf{X}_2, ..., \mathbf{X}_{t+1}\}$ and $\phi(\mathbf{x}_i)$ is the assignment function which returns the network node to which $\mathbf{x}_i$ is assigned:

$$\phi(\mathbf{x}_i) = \arg\min_{c_j} \|\mathbf{x}_i - \mathbf{w}_{c_j}\|^2, \tag{6.3}$$

and $\mathcal{K}^T(\delta(c_r, c_s))$ is the mutual influence between nodes c_r and c_s (usually, a Gaussian function is a common choice for $\mathcal{K}^T(.)$ that will shrink with time). The notion of neighborhood is introduced by the function $\mathcal{K}^T(.)$, which is called the *neighborhood function*, defined in Equation (6.4)

$$\mathcal{K}^T(\delta(c_r, c_s)) = e^{\frac{-\delta(c_r, c_s)}{T}} \tag{6.4}$$

where T represents the temperature function that controls the size of the neighborhood and the algorithm convergence, and $\delta(c_r, c_s)$ is the length of the shortest path between the nodes c_r and c_s. For our experiments, we took only the direct neighborhood (see Figure 6.1). Thus, $\mathcal{K}^T(.)$ is set to 1.

The next step is to decrease $\mathcal{R}_{batchStream}(\phi, \mathcal{W})$ according to the set of referents

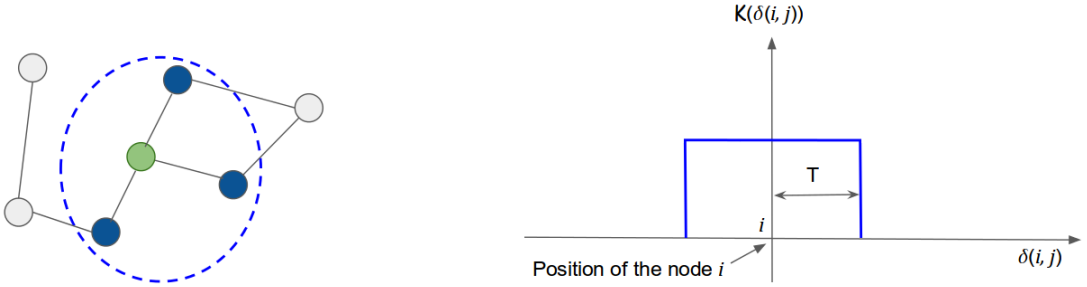


FIGURE 6.1: In left: the direct neighborhood of a node. In right: the neighborhood function. The nodes of the direct neighborhood have the same influence, outside, they have none.

$\mathcal{W}$. It is assumed in this case that ϕ is fixed at the current value. Thus, the prototypes $\mathbf{w}_c$ are calculated using the following equation:

$$\mathbf{w}_c^{(t)} = \frac{\sum_{\mathbf{x}_i \in \mathcal{X}^{(t)}} \mathcal{K}^T\big(\delta(c, \phi(\mathbf{x}_i))\big)\mathbf{x}_i}{\sum_{\mathbf{x}_i \in \mathcal{X}^{(t)}} \mathcal{K}^T\big(\delta(c, \phi(\mathbf{x}_i))\big)} \tag{6.5}$$

$$= \frac{\sum_{\mathbf{x}_i \in \mathcal{X}^{(t-1)}} \mathcal{K}^T\big(\delta(c, \phi(\mathbf{x}_i))\big)\mathbf{x}_i + \sum_{\mathbf{x}_i \in \mathbf{X}_t} \mathcal{K}^T\big(\delta(c, \phi(\mathbf{x}_i))\big)\mathbf{x}_i}{\sum_{\mathbf{x}_i \in \mathcal{X}^{(t-1)}} \mathcal{K}^T\big(\delta(c, \phi(\mathbf{x}_i))\big) + \sum_{\mathbf{x}_i \in \mathbf{X}_t} \mathcal{K}^T\big(\delta(c, \phi(\mathbf{x}_i))\big)}. \tag{6.6}$$

Rather than scannimg all the data, we will scan them block by block (where $Pr = \{\mathbf{x}_i : \phi(\mathbf{x}_i) = r\}$, i.e., the set of observations $\mathbf{x}_i$ assigned to the cluster r):

$$\mathbf{w}_c^{(t)} = \frac{\sum_{r \in C} \sum_{\mathbf{x}_i \in Pr^{(t-1)}} \mathcal{K}^T(\delta(c, r))\mathbf{x}_i + \sum_{r \in C} \sum_{\mathbf{x}_i \in Pr^{(t)}} \mathcal{K}^T(\delta(c, r))\mathbf{x}_i}{\sum_{r \in C} \sum_{\mathbf{x}_i \in Pr^{(t-1)}} \mathcal{K}^T(\delta(c, r)) + \sum_{r \in C} \sum_{\mathbf{x}_i \in Pr^{(t)}} \mathcal{K}^T(\delta(c, r))} \tag{6.7}$$

$$= \frac{\sum_{r \in C} \mathcal{K}^T(\delta(c, r)) \sum_{\mathbf{x}_i \in Pr^{(t-1)}} \mathbf{x}_i + \sum_{r \in C} \mathcal{K}^T(\delta(c, r)) \sum_{\mathbf{x}_i \in Pr^{(t)}} \mathbf{x}_i}{\sum_{r \in C} \sum_{\mathbf{x}_i \in Pr^{(t-1)}} \mathcal{K}^T(\delta(c, r)) + \sum_{r \in C} \sum_{\mathbf{x}_i \in Pr^{(t)}} \mathcal{K}^T(\delta(c, r))} \tag{6.8}$$

Multiplying by the factors $\frac{n_r^{(t-1)}}{n_r^{(t-1)}}$ and $\frac{m_r^{(t)}}{m_r^{(t)}}$, where $n_c^{(t-1)}$ is the number of points assigned to the cluster c thus far, and $m_r^{(t)}$ is the number of points assigned to the

cluster in the current batch, we get:

$$= \frac{\sum_{r \in C} \mathcal{K}^T(\delta(c,r)) n_r^{(t-1)} \frac{\sum_{\mathbf{x}_i \in Pr^{(t-1)}} \mathbf{x}_i}{n_r^{(t-1)}}}{\sum_{r \in C} \sum_{\mathbf{x}_i \in Pr^{(t-1)}} \mathcal{K}^T(\delta(c,r)) + \sum_{r \in C} \sum_{\mathbf{x}_i \in Pr^{(t)}} \mathcal{K}^T(\delta(c,r))} \tag{6.9}$$

$$+ \frac{\sum_{r \in C} \mathcal{K}^T(\delta(c,r)) m_r^{(t)} \frac{\sum_{\mathbf{x}_i \in Pr^{(t)}} \mathbf{x}_i}{m_r^{(t)}}}{\sum_{r \in C} \sum_{\mathbf{x}_i \in Pr^{(t-1)}} \mathcal{K}^T(\delta(c,r)) + \sum_{r \in C} \sum_{\mathbf{x}_i \in Pr^{(t)}} \mathcal{K}^T(\delta(c,r))}. \tag{6.10}$$

Let $\mathbf{w}_c^{(t-1)}$ be the previous center for the cluster c, and $\mathbf{z}_r^{(t)}$ is the new cluster center from the current batch as defined in Equation (6.11).

$$\mathbf{z}_r^{(t)} = \frac{\sum_{\mathbf{x}_i \in Pr^{(t)} | \phi(\mathbf{x}_i) = r} \mathbf{x}_i}{m_r^{(t)}} \tag{6.11}$$

$$\mathbf{w}_c^{(t)} = \frac{\sum_{r \in C} \mathcal{K}^T(\delta(c,r)) \mathbf{w}_c^{(t-1)} n_r^{(t-1)} + \sum_{r \in C} \mathcal{K}^T(\delta(c,r)) \mathbf{z}_r^{(t)} m_r^{(t)}}{\sum_{r \in C} \sum_{\mathbf{x}_i \in Pr^{(t-1)}} \mathcal{K}^T(\delta(c,r)) + \sum_{r \in C} \sum_{\mathbf{x}_i \in Pr^{(t)}} \mathcal{K}^T(\delta(c,r))} \tag{6.12}$$

$$= \frac{\sum_{r \in C} \mathcal{K}^T(\delta(c,r)) \mathbf{w}_c^{(t-1)} n_r^{(t-1)} + \sum_{r \in C} \mathcal{K}^T(\delta(c,r)) \mathbf{z}_r^{(t)} m_r^{(t)}}{\sum_{r \in C} \mathcal{K}^T(\delta(c,r)) n_r^{(t-1)} + \sum_{r \in C} \mathcal{K}(c,r) m_r^{(t)}}. \tag{6.13}$$

However, in batchStream (see Algorithm 20 for details), for each batch of data $\mathbf{X}_p$, we assign all points $\mathbf{x}_i \in \mathbf{X}_p$ to their best match unit, compute new cluster centers, then update each cluster. The update rule, i.e., the adaptation step, in a mini-batch version without taking into account the neighbors of the referent would be as described in Equation (6.14):

$$\mathbf{w}_c^{(t+1)} = \frac{\mathbf{w}_c^{(t)} n_c^{(t)} \alpha + \mathbf{z}_c^{(t)} m_c^{(t)}}{n_c^{(t)} \alpha + m_c^{(t)}}, \tag{6.14}$$

where α is a decay factor parameter $0 < \alpha < 1$, $\mathbf{w}_c^{(t)}$ is the previous center for the cluster, $n_c^{(t)}$ is the number of points assigned to the cluster thus far, $\mathbf{z}_c^{(t)}$ is the new cluster center from the current batch, and $m_c^{(t)}$ is the number of points assigned to the cluster c in the current batch.

Equation (6.15) updates the number of points assigned to the cluster.

$$n_c^{(t+1)} = n_c^{(t)} + m_c^{(t)} \tag{6.15}$$

The error variable of the nearest node is calculated according to Equation (6.16)

$$error(bmu_1) = \|\mathbf{x}_i - bmu_1\|^2 \tag{6.16}$$

In most data stream scenarios, more recent data can reflect the emergence of new trends or changes in the data distribution de Andrade Silva et al. [2013]. There are three window models commonly studied in data streams: landmark, sliding and damped (as presented in chapter 4).

We consider the damped window model, in which the weight of each data point decreases exponentially with time via a *fading* function. The weight of each node decreases exponentially with time t via a decay factor parameter $0 < \alpha < 1$, i.e.,

$$\pi_c^{(t+1)} = \pi_c^{(t)} \alpha \tag{6.17}$$

If the weight of a node is less than a threshold value then this node is considered as outdated and then deleted (with its links). The decay factor can be used to ignore the past: with $\alpha = 1$ all data will be used from the beginning; with $\alpha = 0$ only the most recent data will be used. This is analogous to the *fading* function de Andrade Silva et al. [2013] which is defined as follows :

$$f(t) = 2^{-\lambda t} \tag{6.18}$$

where $\lambda > 0$. In a general case, when the referent moves toward a data point, it also moves its neighborhood toward this point Kohonen et al. [2001].

By including the neighborhood function (Equation 6.4) with the fading function (Equation 6.17) in Equation (6.14), in our model, we use Equation (6.19) to carry out the adaptation step for micro-batch streams:

$$\mathbf{w}_c^{(t+1)} = \frac{\mathbf{w}_c^{(t)} n_c^{(t)} \alpha + \sum_{r \in \mathcal{C}} \mathcal{K}^T(\delta(r,c)) \mathbf{z}_r^{(t)} m_r^{(t)}}{n_c^{(t)} \alpha + \sum_{r \in \mathcal{C}} \mathcal{K}^T(\delta(r,c)) m_r^{(t)}} \tag{6.19}$$

where $\mathbf{z}_r^{(t)}$ is the previous center for the cluster r (which is a neighbor of the considered referent node), $\mathcal{K}^T(.)$ is the *neighborhood function* defined in Equation (6.4).

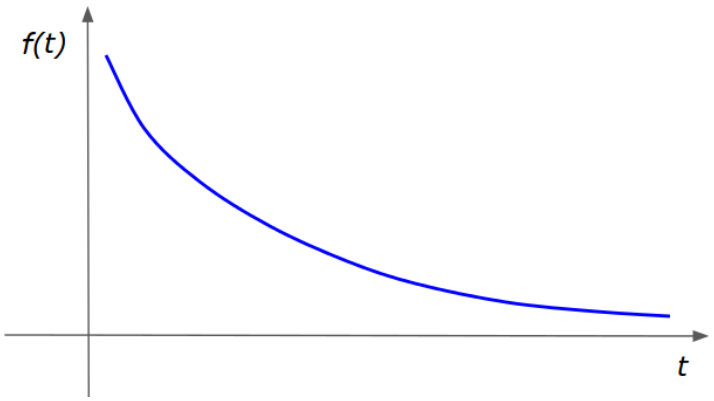


FIGURE 6.2: Plot of a fading function.

We are now ready to outline batchStream in Algorithm 20.

Algorithm 20: batchStream

Input: $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$, α, π_{min}, λ_{age}, η, age_{max}, d
Output: set of nodes $\mathcal{C} = \{c_1, c_2, ...\}$ and their prototypes $\mathbf{W} = \{\mathbf{w}_{c_1}, \mathbf{w}_{c_2}, ...\}$

1 Initialize of the model by creating a graph of two nodes (the first 2 data-points);
2 **while** *there is a micro-batch to proceed* **do**
3 $\quad \mathbf{X}_t \leftarrow$ get the micro-batch of data points arrived at time interval t;
4 $\quad$ Apply the *mapping* step in Function map ;
5 $\quad$ Apply the *reduce* step in Function reduce;
6 $\quad$ Apply the adaptation step: updateRule(pointStats, α, λ_{age}, age_{max});
7 $\quad$ Update the variable error of each node;
8 $\quad$ Apply *fading*, delete isolated nodes;
9 $\quad$ Add η new nodes in Function addNewNodes ;
10 $\quad$ Decrease the error of all units by multiplying them with a constant d ;

Table 6.1 overviews the list of parameters used in the batchStream algorithm.

Symbole	Description
α	the decay factor parameter
π_{min}	the minimum weight of a node. If bellow, this node is deleted
λ_{age}	the rate of growth of the edges' age
η	the number of nodes to add at each iteration
age_{max}	the maximum edges's age
d	a constant for adjusting the error variable of nodes, such as $0 < d < 1$

TABLE 6.1: Parameters used in the batchStream algorithm

6.3 Modeling using MapReduce

The input data is split and the master assigns splits to the Map workers. In the Map step, each worker processes the corresponding input split, generates *key/value* pairs and writes them to intermediate files (on disk or in memory). The *key* corresponds to the *best match unit* (bmu_1), called also *the nearest node*, whereas its *value* represents a tuple of (bmu_2, *error*, $\mathbf{x}_i$, 1), where bmu_2 represents the second nearest node.

Then the master will launch the *Reduce* tasks that take as input both the results of the Maps and the results of the previous interval's Reduces. The Reduce function is responsible for aggregating information received from Map functions. For each *key*, the Reduce function works on the list of values, *closest*. Figure 6.3 illustrates the sequences of Map and Reduce tasks triggered automatically by the framework (the Spark engine in our case).

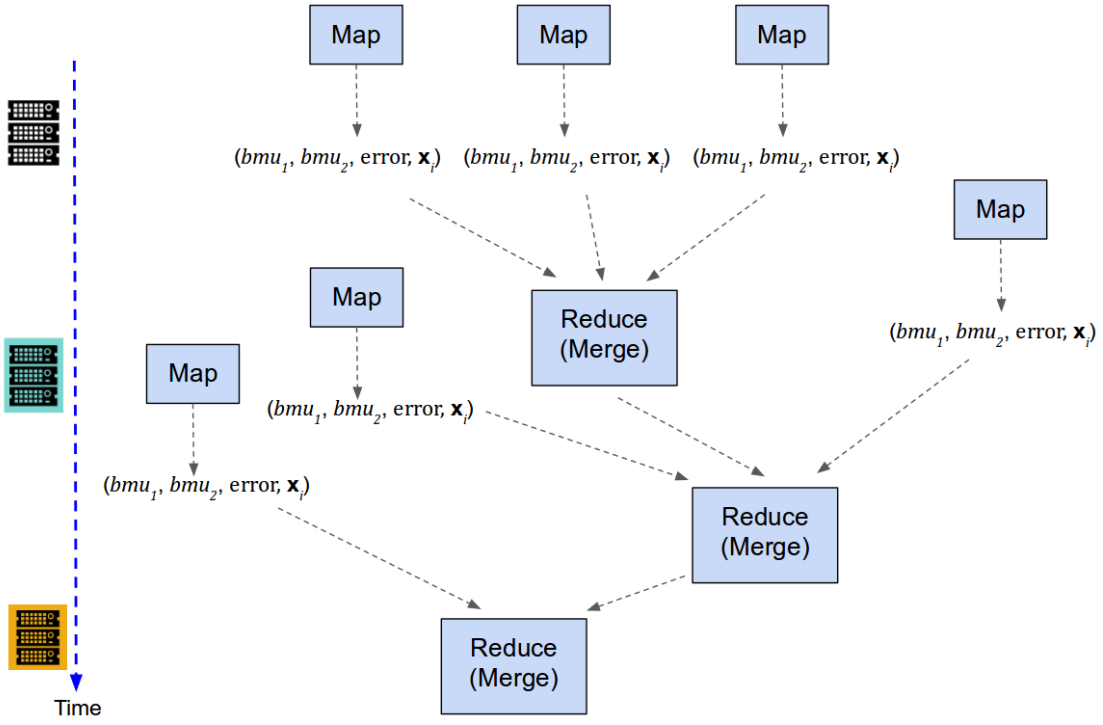


FIGURE 6.3: Overview of the Map and Reduce tasks in batchStream.

To compute the prototype of each node, the Reduce function groups by bmu_1 and sums the values received in the *closest* list. The final output is the list *pointStats*. Each element of *pointStats* contains a bmu_1 (a prototype), as *key* and the second nearest node bmu_2, the sum of errors $error_t$, the sum sum_t and the count of points assigned to each node $count_t$, as the *value*.

The function updateRule performs the operations related to updating graph edges. The way to increase the age of edges is inspired by the fading function in the sense that the creation time of a link is taken into account. Contrary to the *fading* function, the age of the links will be strengthened by the exponential function:

$$g(t) = 2^{\lambda_{age}(t-t_0)} \tag{6.20}$$

where $\lambda_{age} > 0$, defines the rate of growth of the age over time, t denotes the current time and t_0 is the creation time of the edge.

The next step is to add a new edge that connects the two closest nodes. The last step is to remove each link exceeding a maximum age, since these links are no longer useful because they were replaced by younger and shorter edges that were created during the graph refinement in step 9.

Function map($\mathbf{X}_t$: the t-th micro-batch of data points)

1 **foreach** $\mathbf{x}_{ti} \in \mathbf{X}_t$ **do**
2 Key $\leftarrow bmu_1$, the nearest node;
3 Value $\leftarrow (bmu_2, error, \mathbf{x}_{ti}, 1)$ such as: bmu_2 is the second nearest node, and calclute the error according to Equation (6.16);
4 Emit (Key, Value);

Function reduce(key_t, List $closest$)

Output: $prototype_t$: the prototype of the t-th micro-batch, bmu_2: the second nearest node, sum_t: the sum of errors, and $count_t$: the number of data points in the t-th micro-batch

1 $bmu_2 \leftarrow 0$; $error_t \leftarrow 0$; $sum_t \leftarrow 0$; $count_t \leftarrow 0$; $pointStats \leftarrow List()$;
2 **foreach** $value_t \in closest$ **do**
`// where` $value_t$ `is the corresponding value of the pair (`key_t`, Value)`
3 $bmu_2 \leftarrow bmu_2$ + the 1-st value of tuple $value_t$;
4 $error_t \leftarrow error_t$ + the 2-nd value of tuple $value_t$;
5 $sum_t \leftarrow sum_t$ + the 3-rd value of tuple $value_t$;
6 $count_t \leftarrow count_t$ + the 4-th value of tuple $value_t$;
7 $prototype_t \leftarrow sum_t / count_t$;
8 Add $(prototype_t, bmu_2, sum_t, count_t)$ to the list $pointStats$;

Function updateRule(List $pointStats$, α, λ_{age}, age_{max})

```
// Decrease the weight of nodes
```

1 **foreach** $c \in \mathcal{C}$ **do** Update $\pi_c^{(t)}$ according to Equation (6.17) ;
2 **foreach** $ps \in pointStats$ **do**
    `// ps is a tuple: (bmu, (bmu2, error, sum, count))`
3 Calculate the new prototype according to Equation (6.19)
4 Increment the age of all edges emanating from bmu and weight them ;
5 **if** *bmu and bmu_2 are connected by an edge* **then** set the age of this edge to zero;
6 **else** create an edge between bmu and bmu_2, and mark its time stamp;
7 Remove the edges whose age is greater than age_{max}. If this results in nodes having no emanating edges, remove them as well;

Function addNewNodes(η : number of nodes to add)

1 **for** $j \leftarrow 1$ **to** η **do**
2 Find the node with the largest error;
3 Find the neighbor f with the largest accumulated error;
4 Add the new node r half-way between nodes q and f according to Equation (6.1);
5 Insert edges connecting the new unit r with units q and f, and remove the original edge between q and f. Remove the original edge between q and f;
6 Initialize the weight of r and the age of edges emanating from r to zero;
7 Decrease the error variables of q and f by multiplying them with a constant ϵ where: $0 < \epsilon < 1$;
8 Initialize the error variable of r with the new value of the error variable of q;

6.4 Experimental evaluations

In this section, we present an experimental evaluation of the batchStream algorithm. We compared our algorithm with several well-known and relevant data stream clustering algorithms, including ClusTree, DenStream, and the MLlib[3] implementation of Streaming-KMeans.

Our experiments were performed on the Spark Streaming platform using public real-world and synthetic data sets. Experiments on the large datasets (the Sensor, the CoverType, and the KddCup99 datasets) are conducted on the *Teralab*[4] cluster which runs the Debian operating system with the following properties:

[3] http://spark.apache.org/docs/latest/mllib-guide.html
[4] https://www.teralab-datascience.fr/en/home

- 5 data-nodes: 50 Gb system disc, 20 VCPUs, 120Gb RAM, 4×200Gb data discs
- 1 edge-node: 4 VCPUs, 32Gb RAM, 100Gb hard disc
- 1 service-node: 4 VCPUs, 16Gb RAM, 60Gb hard disc
- 2 name-nodes: 30Gb system disc, 2 VCPUs, 4Gb RAM.

The experiments for the other datasets were conducted on a PC with Core(TM)i7-4800MQ with 2×2.70 GHz processors, and 8Gb RAM, which runs the Ubuntu 13.10 operating system.

6.4.1 Datasets

To evaluate the clustering quality and scalability of the batchStream algorithm both real and synthetic data sets are used. The synthetic data sets used are DS1 and letter4. All the others are real-world publicly available data sets. Table 6.2 overviews all the data sets used.

Datasets	#observations	#features	#classes
Sensor	2,219,803	5	54
CoverType	581,012	54	7
KddCup99	494,021	41	23
Sea	60,000	3	2
letter4	9,344	2	7
DS1	9,153	2	14

TABLE 6.2: Overview of all data sets.

- DS1 is generated by http://impca.curtin.edu.au/local/software/synthetic-data-sets.tar.bz2.
- The letter4 dataset is generated by a Java code https://github.com/feldob/Token-Cluster-Generator.
- The Sea dataset was taken from http://www.liaad.up.pt/kdus/products/datasets-for-concept-drift.

- The real-world databases were taken from the UCI repository [Bache and Lichman, 2013], which are the KDD-CUP'99 Network Intrusion Detection stream dataset (KddCup99) and the Forest CoverType dataset (CoverType) respectively.

The algorithms are evaluated using three performance measures: Accuracy (Purity), Normalized Mutual Information (NMI) and Rand index (please refer to Appendix A for more details). The value of each measure lies between 0 and 1. A higher value indicates better clustering results.

6.4.2 Evaluation and performance comparison

This section aims to evaluate the clustering quality of the batchStream and compare it to well-known data stream clustering algorithms. As explained in section 6.2, batchStream algorithms start with two nodes.

For comparison purposes, we used the MLlib implementation of Streaming-KMeans (this latter algorithm was also coded in the Spark Streaming platform)[5]. A comparison is also performed with DenStream Cao et al. [2006] and ClusTree Kranen et al. [2011] from the **stream** R package Bolanos et al. [2014]. Streaming-KMeans was evaluated by setting the k parameter to the known number of classes of each dataset. DenStream was evaluated by performing a variant of the DBSCAN algorithm in the offline step. ClusTree was evaluated by performing the k-means algorithm in the offline step by setting the k parameter to 10.

Table 6.3 reports the results in terms of accuracy, NMI, and Rand index. In this Table, it is noteworthy that batchStream's Accuracies (Acc) are higher for all data sets as compared to Streaming-KMeans, DenStream and CluStree, except for ClusTree for Streaming-KMeans for the KddCup99 data set. Its NMI values are higher than the other algorithms except for Streaming-KMeans for DS1 and KddCup99 data sets. Its Rand index values are higher than the other algorithms except for Streaming-KMeans for Sea and DS1 data sets. We recall that batchStream proceeds in one single phase whereas Streaming-KMeans, DenStream and ClusTree proceed in two phases (online and offline phase).

[5]https://spark.apache.org/docs/latest/mllib-clustering.html#streaming-k-means

Datasets		**batchStream**	Streaming KMeans	DenStream	ClusTree
DS1	Acc	**0.9773**	0.8067	0.7740	0.6864
	NMI	0.7019	**0.7274**	0.6973	0.7064
	Rand	0.8473	**0.8657**	0.8491	0.8442
letter4	Acc	**0.8566**	0.4848	0.8110	0.8110
	NMI	**0.6844**	0.4672	0.1637	0.2425
	Rand	**0.8542**	0.6915	0.5019	0.5514
Sea	Acc	**0.8374**	0.6269	0.8240	0.8224
	NMI	**0.1381**	0.0018	0.1646	0.1583
	Rand	0.4708	**0.5030**	0.4700	0.4917
KddCup99	Acc	0.9262	**0.9832**	0.9544	0.8182
	NMI	0.6622	**0.7035**	0.6290	0.5724
	Rand	0.8367	**0.8382**	0.8164	0.8289
CoverType	Acc	**0.6527**	0.4957	0.5850	0.5850
	NMI	**0.1653**	0.0727	0.0475	0.0362
	Rand	**0.6233**	0.5931	0.4604	0.5080
Sensor	Acc	0.1086	0.0690	0.5850	0.5850
	NMI	**0.1471**	0.0970	0.0475	0.0362
	Rand	**0.9738**	0.9555	0.4604	0.5080

TABLE 6.3: Comparing batchStream with other data stream clustering algorithms.

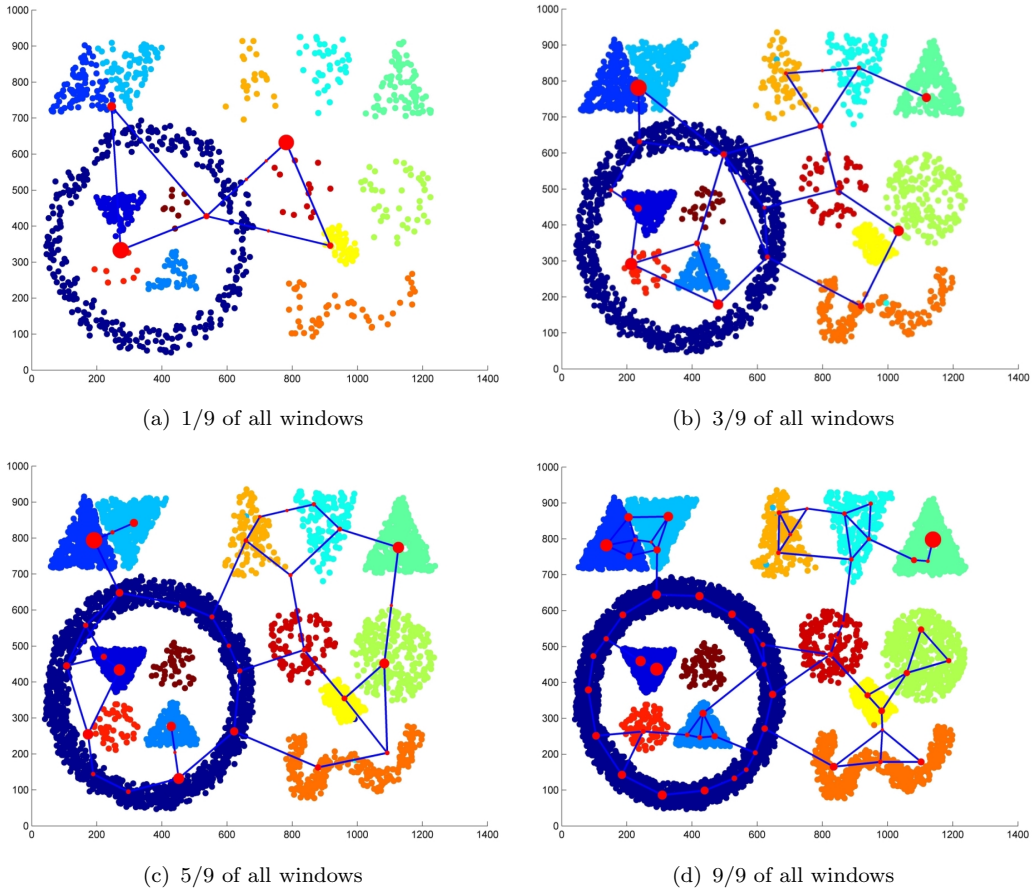
6.4.3 Visualization of graph creation evolution

6.4.3.1 Non-overlapping data streams

Figures 6.4 and 6.5 show the evolution of the node creation by applying batchStream on the DS1 and DS2 data sets (colored points represent data points of the data stream and red points are nodes of the graph with edges in blue lines; each color of the data points correspond to class of labels and the size of the nodes of the graph are proportional to their weight). It illustrates that batchStream manages to recognize the structures of the data stream and can separate these structures with the best visualization.

6.4.3.2 Overlapping data streams

In some situations, input data streams may overlap (i.e, some data points are located on the same space). Figure 6.6 shows the evolution of graph creation of batchStream on the lettersMR dataset where data points of the letter M arrive at first then those of R. The graph generated by batchStream can adapt with the



(a) 1/9 of all windows

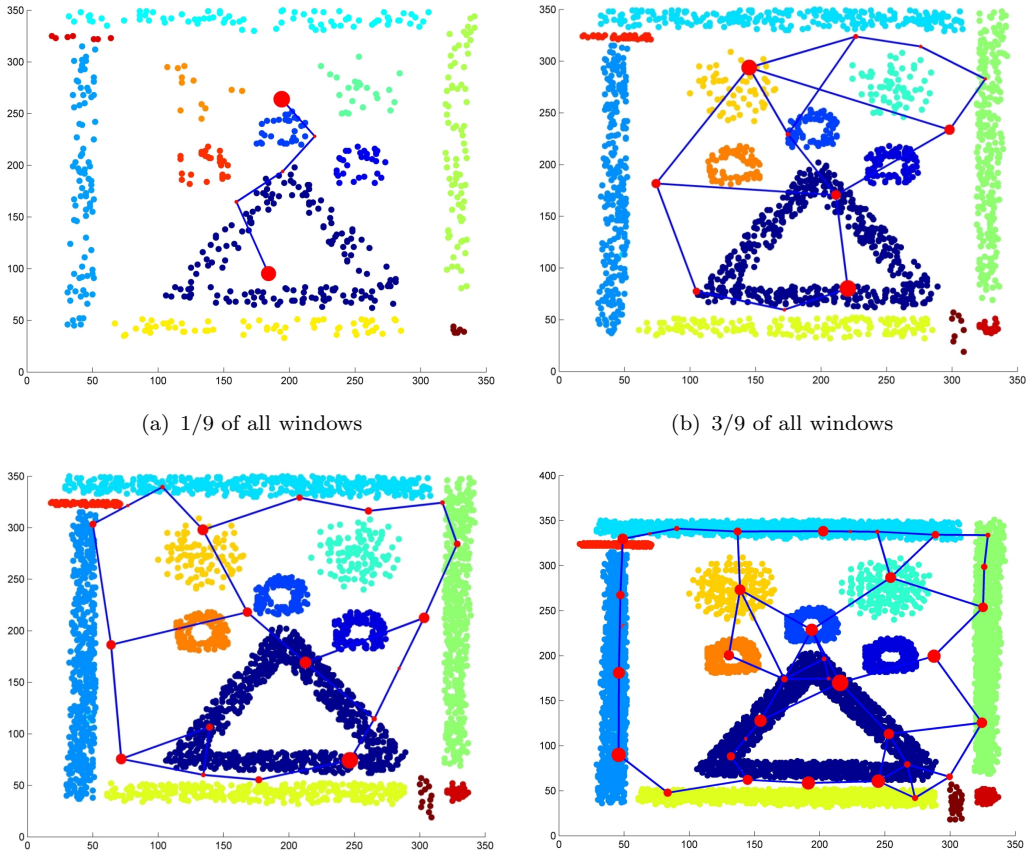
(b) 3/9 of all windows

(c) 5/9 of all windows

(d) 9/9 of all windows

FIGURE 6.4: Evolution of graph creation of batchStream on DS1 (data set and topological result). The intermediate graph after seeing the 1/9 of all windows; the 3/9 of all windows; the 5/9 of all windows; and the final graph (9/9 of all windows).

evolving overlapped data stream since it can "forget" the old letter M and learn the topological structure of the novel letter R (this is mainly due to the fading function).



(a) 1/9 of all windows

(b) 3/9 of all windows

(c) 5/9 of all windows

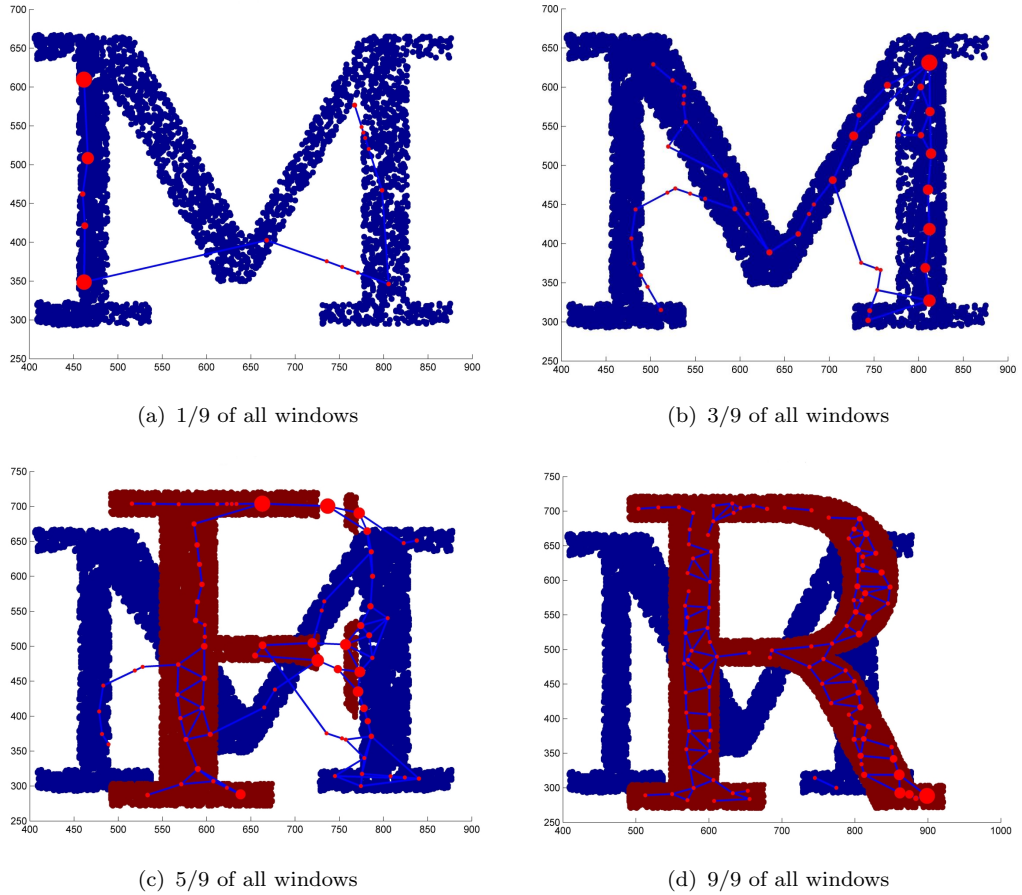
(d) 9/9 of all windows

FIGURE 6.5: Evolution of graph creation of batchStream on DS2 (data set and topological result). The intermediate graph after seeing the 1/9 of all windows; the 3/9 of all windows; the 5/9 of all windows; and the final graph (9/9 of all windows).

6.4.4 Evolving data streams

In this subsection, we perform the batchStream algorithm on different data streams ordered by class labels to demonstrate its effectiveness in clustering evolving data streams (i.e., data points of the first class arrive in first, then the ones of the second, third, etc. class). In this case, old concepts (class labels) disappear due to the use of fading function. In the same time, new concepts (class labels) appear as new data points arrive. Note that the class labels are not known to the clustering algorithm. We report the results in Figure. 6.7.

The top panel in Figure 6.7 compares the batchStream algorithm, in terms of the accuracy, with (i.e., we sort the data points based on their class labels) and without ordering of class labels. It shows that the batchStream algorithm with ordering of classes can find clusters with accuracy values as comparable to those



(a) 1/9 of all windows

(b) 3/9 of all windows

(c) 5/9 of all windows

(d) 9/9 of all windows

FIGURE 6.6: Evolution of graph creation of batchStream on lettersMR (data set and topological result). The intermediate graph after seeing the 1/9 of all windows; the 3/9 of all windows; the 5/9 of all windows; and the final graph (9/9 of all windows).

without ordering of classes.

The middle panel in Figure 6.7 compares the batchStream algorithm, in terms of the NMI. It shows that the batchStream algorithm with ordering of classes can find clusters with NMI values as comparable to those without ordering of classes for most datasets. Although, we observe that the values of NMI are lower in the case where we sort the data points based on their class labels, for the DS1 and the KddCup99 datasets.

The bottom panel in Figure 6.7 compares the batchStream algorithm, in terms of the Rand index. Except for the KddCup99 dataset where the Rand index value decreases in the case where the data points are sorted, this Figure shows that the batchStream algorithm with ordering of classes can find clusters with Rand index values as comparable to those without ordering of classes for most datasets.

(a) Accuracy

(b) NMI

(c) Rand index

FIGURE 6.7: Accuracy, NMI and Rand index for batchStream with and without ordering of classes.

6.4.5 Temporal performance vs batch interval

Spark Streaming uses the concept of micro-batch streaming, i.e., it aggregates data arriving within a *batch interval* and, at the end of the time interval, it applies the MapReduce operation on the batch data. MapReduce operations are parallel functions that run on distributed data. Thus, the wider the batch interval (the window length) is, the more distributed data to treat, the more parallelization is effective. However, in real-world applications, wider batch intervals may cause high latency.

Figure 6.8 shows the execution time of batchStream for the insurance dataset (this dataset is described in chapter 7) while varying the size of the batch interval. To do this, we simulated the insurance dataset as a data stream. The source generating the data stream takes the *batch-size*, as parameter and then ingests *batch-size* input data each time. The batch sizes used in this experiments are: 1, 5, and 10 million of input data. Figure 6.8 shows the overall time execution (including the delay time) of the last batches. It shows that the larger the batch size is, the lower the time execution is taken by the batchStream algorithm.

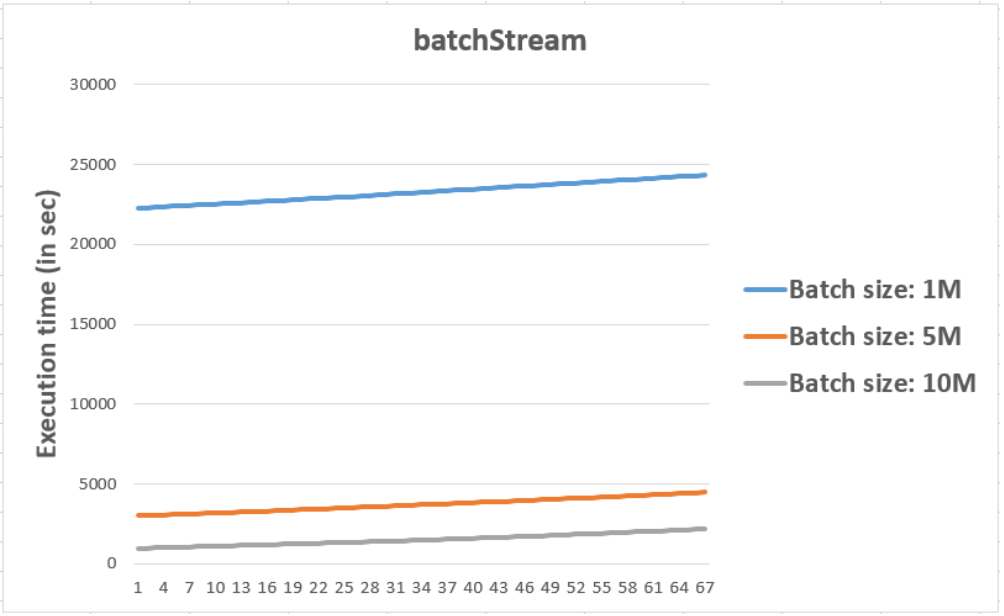


FIGURE 6.8: The overall execution time of batchStream as a function of window length (batch size).

6.5 Conclusion

In this chapter, we have presented batchStream, an efficient method for topological clustering an evolving data stream in an online manner. In batchStream, the nodes are weighted by a fading function and the edges by an exponential function. The batchStream algorithm is implemented on a distributed streaming platform based on the micro-batching processing model.

Experimental evaluation over a number of real and synthetic data sets demonstrates the effectiveness and efficiency of batchStream in discovering clusters of arbitrary shape. The performance of batchStream, in terms of clustering quality as compared to three relevant data stream algorithms are promising. We plan in the future to extend batchStream to deal with binary, categorical, and mixed data streams, and also to make our algorithm as autonomous as possible.

The work presented in this chapter has resulted in the following publication: Mohammed Ghesmoune, Mustapha Lebbah, and Hanane Azzag. Micro-batching growing neural gas for clustering data streams using spark streaming. In INNS Conference on Big Data 2015, San Francisco, CA, USA, 8-10 August 2015, pages 158–166, 2015b. doi: 10.1016/j.procs.2015.07.290. URL http://dx.doi.org/10.1016/j.procs.2015.07.290.

The next chapter presents our contribution in the Big Data project, called "Square Predict". In particular, we will illustrate the utility of the batchStream algorithm, presented in the previous chapter 6, as an unsupervised learning for an insurance Big Data.

Chapter 7

Application for Insurance Big Data

This chapter is devoted to explain our work carried in the context of the Big Data project, named *Square Predict*. At first, we present the architecture of the proposed Big Data framework. Then, we will illustrate the utility of the batchStream algorithm, presented in the chapter 6, as an unsupervised learning for an insurance Big Data.

7.1 Introduction

Organisations are increasingly relying on Big Data to provide the opportunities to discover correlations and patterns in data that would have previously remained hidden, and to subsequently use this new information to increase the quality of their business activities.

The "Square Predict" project is gathering 3 public research labs and 4 private companies including AXA Data Innovation Lab. This project aims to provide the insurance industry a platform for real-time predictive analytics that can analyze the information published on social networks coupled with the information available in Open Data, e.g. to assess the rapidly the severity of a natural disaster and its impact on housing insurance payouts.

In this chapter we present, briefly, a 'story' of Big Data from the initial data collection to the final visualization, passing by the data fusion, and the analysis and clustering tasks. For this, we present a complete work-flow on:

(a) how to represent the heterogeneous collected data using the high performance RDF language, how to perform the fusion of the Big Data in RDF by resolving the issue of entity disambiguity, and how to query those data to provide more relevant and complete knowledge. For consistency, we omit the details concerning the collection, the fusion, and the query of heterogenous data; the interested reader can be refer to [Benbernou et al., 2015] for more details.

(b) as the data are received in data streams, we apply the *batchStream* method, presented in the chapter 6, in order to estimate, in real time, the impact of damage caused by a major climatic event, combining our unsupervised method with a supervised model.

7.2 Architecture of the Big data framework

In this section, we describe the Big data platform developed from the end-to-end. The application domain we target is insurance. The framework is built upon the Teralab[1] distributed clusters platform.

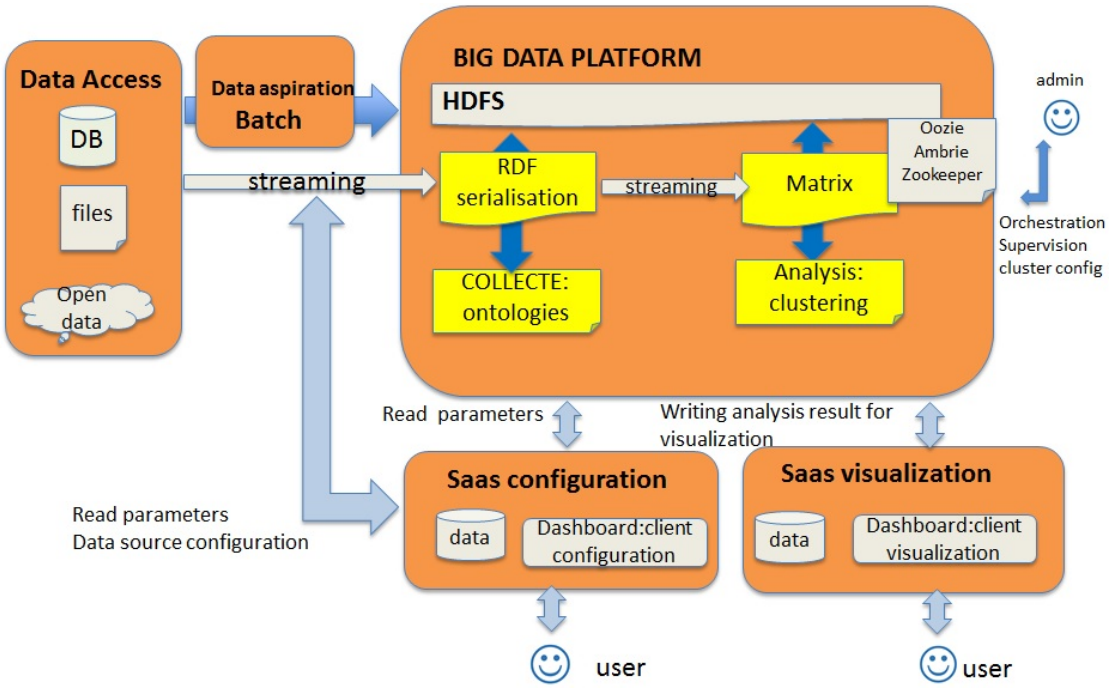


FIGURE 7.1: Big data platform

[1]https://www.teralab-datascience.fr/en/home

1. **Data sets.** The data in our platform are collected from different heterogeneous sources, including proprietary data sets (housing insurance contracts), and open data sets from organizations such as the French national institution of statistics INSEE[2] that contains information related to census household and housing surveys (i.e., type of heating, proportions of housing type in the local area etc), the ONDRP[3] which is a department of the National Institute of High Study of Security and Justice, which contains information related to crime and delinquency (i.e., home invasions, average of armed burglaries against individuals in their homes, etc.), as well as the well known open data base Dbpedia etc.

2. **SaaS Configuration.** This component is a software which provides a dashboard to assist the user to process a data configuration and transfer for serialization into the RDF format.

3. **RDF serialisation and reasoning mechanisms**. In order to provide a semantic reasoning by inferring new hidden data, all data are transformed into the RDF format (if they are not already so). The serialization format is a triplet *subject-predicate-object*. Moreover, the semantic links are built to connect RDF data of different sources with the concepts of an OWL ontology to process the data fusion. Those connections are used to identify and connect RDF data provided by multiple data sources that refers to the same real world entity. Those links are found through the query evaluation approach based on SPARQL language. Finally, once the proprietary data are cleaned and enriched with the newly inferred data, these data are ready to be the input of the analysis and clustering module. This process was done by LIPADE laboratory [Benbernou et al., 2015].

4. **Clustering and Analysis.** The aim of clustering, also known as unsupervised learning, is to separate the data set into a small number of groups where the members within a cluster are similar to each other, and members from different clusters are different to each other. The presence of clusters in a data set implies that there is the possibility of data reduction as all the members of a single cluster can be represented by a typical member known as the prototype. Furthermore, cluster membership is an

[2]http://www.insee.fr/fr/bases-de-donnees/default.asp?page=open-data/open-data-utilisation.htm

[3]http://www.inhesj.fr/fr/ondrp

important tool in analyzing and understanding the deep structure of the data set whenever the clusters correspond to groups of interest. Since we are merging heterogeneous data sets from different sources, clustering provides an analytical tool to quantify the new information created by this newly merged data set, with respect to the individual data sets. For this part of the project, we applied the batchStream (presented in chapter 6) as well as the SOM-MR algorithm (presented in chapter 3) for the insurance Big Data. The implementations of these algorithms are available at: https://github.com/Spark-clustering-notebook/coliseum.

5. **Visualization.** Graphical visualizations present the overall trends in the data, in contrast to their exact values in numerical representations. These over-arching patterns assist in providing a wider context for interpreting existing and new data. The fusion of many existing data sets, whilst providing a potentially unlimited source of new information, can also be potentially disorientating due to an information overload. Visualizations are effective in indicating the directions in which the analysis should proceed as they can present key aspects of the data set in a single graphical summary which would be not evident in a numerical form.

7.3 Application of batchStream for insurance big data

In this section, we demonstrate the utility of batchStream as a method for unsupervised learning for an insurance Big Data, consisting of 2 133 488 insurance contracts for damages claims made in continental France for the calendar year 2012. Five variables were supplied initially by the insurer, and an analysis based on solely on these in-house variables were inconclusive. These variables are: `NBPIECS`: the number of rooms in the dwelling, `CDQUALP`: the owner status (owner or tenant) `CDHABIT`: the dwelling code (apartment or house) `CDRESID`: the residence code (principal residence or second home), and `NB_SIN`: the number of claims. We thank the AXA company for providing us the large dataset that we used in the Square Predict project to validate our proposed algorithms.

We then proceeded to enrich these data with publicly available open data: 20 variables concerning the age of dwelling construction, the type of heating used,

and the age composition of the household members from population census and surveys conducted by the INSEE (the French official national statistical agency), and 13 variables concerning the rates of different types of crimes collected by the ONDRP (the French crime statistics agency). The procedures to process these open data so that they can be merged meaningfully with the insurance claims unit record data are lengthy, and their details are presented in [Benbernou et al., 2015].

To simplify the analysis, we focus on the fire damages. Most contracts are not subject to a claim (2 126 952 or 99.69%) whereas the remaining 6 536 contracts or 0.31% account for 89 410 763 € of damages paid out by the insurer. Further analysis of this highly inhomogeneous structure, in particular the added value of open Big Data, would be of interest to the insurer's business model.

The batchStream algorithm (see Algorithm 20 in chapter 6 for details) was applied to this merged data set, and 84 clusters of varying sizes, forms and locations were the result. Table 7.1 shows the five clusters which exceeded 4 million € in total claims per cluster: these 5 clusters account for 43.00% of the 89 million € of payouts and 35.76% of the 6 536 claims.

Cluster	Total claims	#contracts	#claims
1	10 327 077	16 0281	460
66	10 161 913	13 8769	709
55	8 480 123	10 9588	423
21	5 142 238	81 741	378
47	4 334 039	88 085	367
⋮	⋮	⋮	⋮
All	89 410 763	2 133 488	6 536

TABLE 7.1: Summary statistics for batchStream clusters for insurance data

The summary statistics in Table 7.1 indicate that the batchStream clusters contain important information of the insurance claims, though they are not sufficiently detailed. We carried out a post-hoc decision tree (Classification and Regression Tree or CART [Hastie et al., 2009]) analysis, computed by the **rpart** R package [Therneau et al., 2015]. Decision trees produce a set of interpretable decision rules used to construct to these clusters, as shown in Figure 7.2.

All trees are split at the root node using the in-house variable `nbsin_inc` (number of people assigned in the claim). For the entire data, there are no further splits, leading to a simple decision tree on the top left, indicating that the structure of

these data are not revealed at this aggregated level.

On the other hand, the decision trees for the batchStream clusters are highly structured, with the leaf nodes with an average claim of greater than 50 000 euros coloured in blue. The other in-house variables which appear in these decision trees are `dept` (2 digit postcode) and `nbpiecs` (number of rooms in the dwelling).

The INSEE housing variables, concerning the year of dwelling construction `const*`, the age composition of household members `nbPers*` and the type of heating `cmb*`, are frequently used in these decision rules, whereas the ONDRP variables appear less frequently.

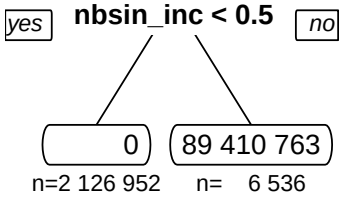
For Cluster #1, the important leaf nodes are created by decisions involving `constAvtProb` (proportion of dwellings constructed before 1949) and `cmbAutreProp` (proportion of dwellings using 'other' heating).

The tree for cluster #66 has the most number of levels of those displayed, and involves additionally `const7589Prob`, `const8903Prob` (proportion of dwellings constructed between 1975 and 1989, and 1989 and 2003), `ANEM_MOY` (average number of years since the last home improvement) and `nbPers4a6_MOY`, `nbPers19a24_MOY`, `nbPers19a24_MOY` (average number of persons between 4 and 6 years, 19 and 24 yers, and more than 75 years of age).

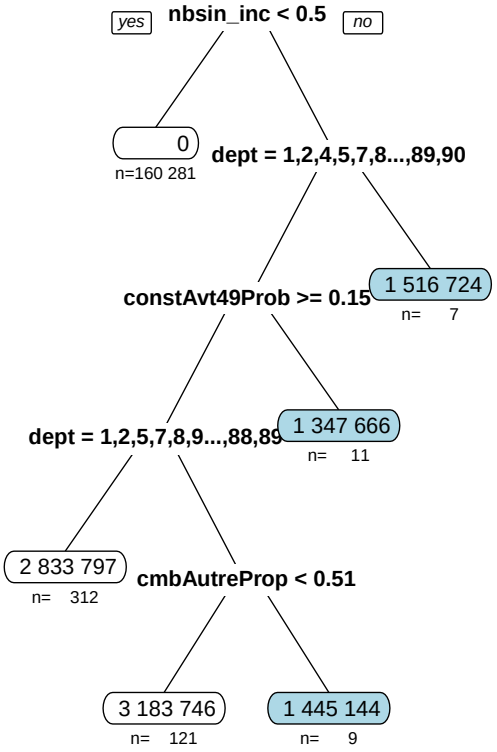
For Cluster #55, the other age composition variables `nbPers0a3_MOY`, `nbPers12a18_MOY`, `nbPers60a64_MOY` (average number of persons between 0 and 3 years, 12 and 18 yers, and 60 and 64 years of age) appear.

The final two clusters #21 and #47 are perhaps the most interesting from the point of view of added value of open data for describing fire damage insurance claims. The tree for cluster #21 involves `cmbGazBoutProp` (proportion of dwellings using bottled gas heating) and for cluster #47 `cmbElectProp` (proportion of dwellings using electric heating) and `DrgPub` (number of attacks against public property) and `MalEnfProp` (proportion of households with crimes committed against children). For these batchStream clusters, more detailed information relevant to insurance claims is provided by freely available open data.

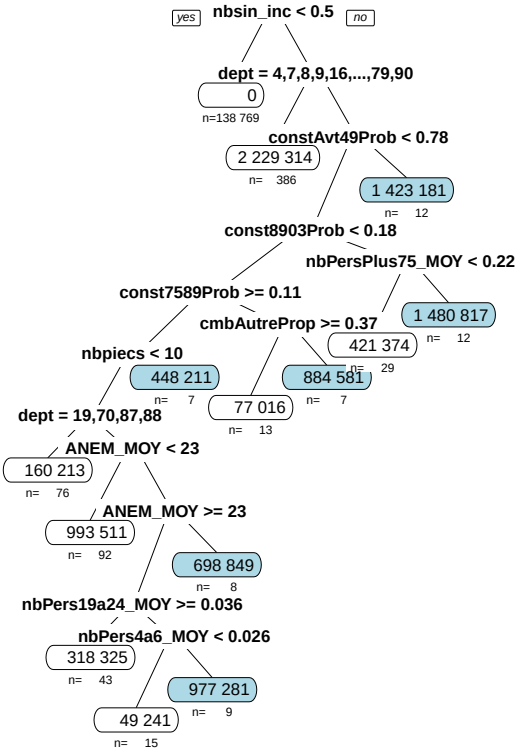
All



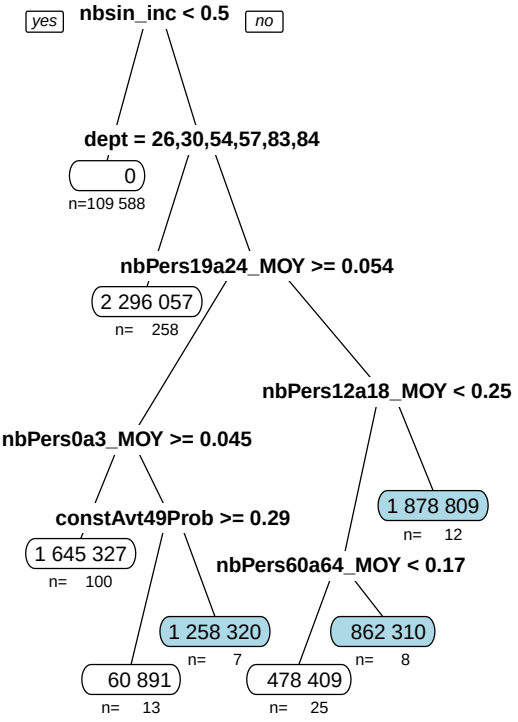
Cluster #1



Cluster #66



Cluster #55



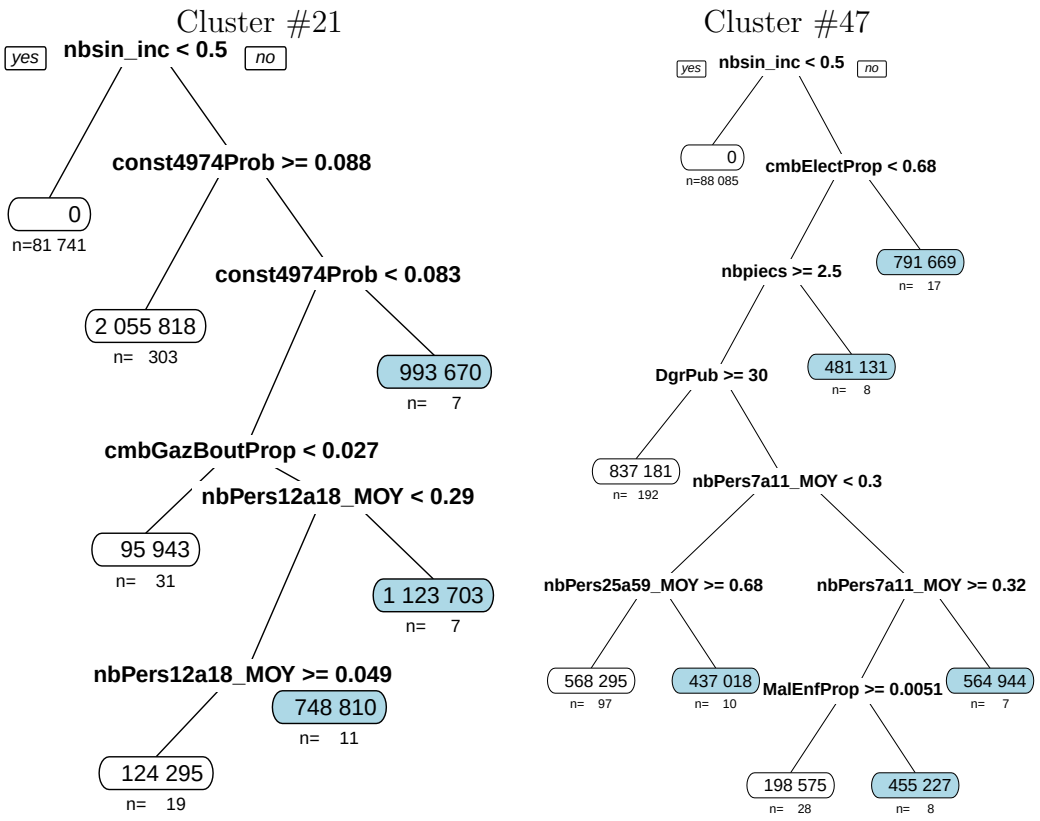


FIGURE 7.2: Decision trees for batchStream clusters of insurance data, for the total data and the 5 largest clusters by total cluster payouts. Leaf nodes with average claims of over 50 000 € are coloured in blue.

7.4 Analysis of the insurance big data using batchStream

To further analyze clusters, we use the following 3 indicators: rate of claims, payouts per contract, and loss per contract.

$$Rate\_of\_claims = \frac{Number\_of\_claims}{Number\_of\_contracts} \tag{7.1}$$

$$Payout\_per\_claim = \frac{Sum\_of\_claim\_amounts}{Number\_of\_claims} \tag{7.2}$$

$$Loss\_per\_contract = Rate\_of\_claims * Payout\_per\_claim \tag{7.3}$$

Regarding these indicators, especially the maximum and minimum values, the insurance company can focuse its analysis on the corresponding clusters. Thus, a

model based on the features of assigned data can be defined. Using this model, the insurance company can predict the payouts for a new customer within a cluster and so propose more personalised insurance contracts for its customers.

Table 7.2 summarizes the values of these indicators for clusters: 1, 66, 55, 21, and 47. We distinguish two types of claims: claim water damage (WAT) and claim fire damage (FIR).

Cluster	Rate_WAT	Rate_FIR	Payout_WAT	Payout_FIR	Loss_WAT	Loss_FIR
1	0.0126	0.0029	1706.4	22305	21.45	64.25
66	0.0235	0.0022	874.91	6249.2	20.58	13.71
55	0.0407	0.0016	886.3	4676.9	36.11	7.33
21	0.0277	0.0027	1276	12552	35.34	33.51
47	0.040	0.0029	1066.3	2658.9	42.64	7.80

TABLE 7.2: Rate of claims, Payout per claim, and Loss per contract for batchStream clusters for insurance data

Figues 7.3 and 7.4 show a visualization of contracts, assigned to clusers 21 and 55, on the map of France by department.

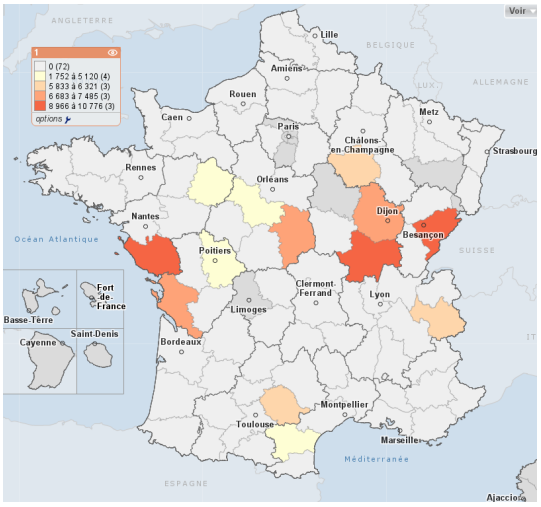
7.5 Conclusion

In this chapter, we presented our work carried in the context of the *Square Predict* project. In the first section, we have presented the architecture of the proposed Big Data framework. A demonstration of the utility of the batchStream algorithm (which is presented in the chapter 6) as an unsupervised learning for the insurance Big Data was also presented.

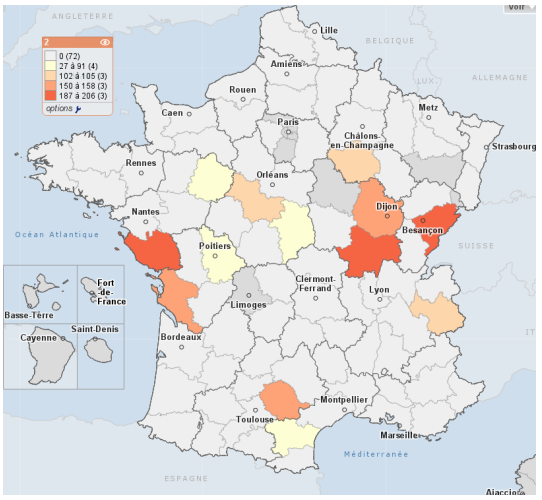
We plan in the future to extend batchStream to deal with binary, categorical, and mixed data streams, and also to make our algorithm as autonomous as possible.

The work presented in this chapter has resulted in the following publication: Hanane Azzag, Salima Benbernou, Tarn Duong, Mohammed Ghesmoune, Mustapha Lebbah, and Mourad Ouziri. Big Data: A Story from Collection to Visualization. Submitted to Machine Learning Journal: Special issue on Discovery Science, 2016.

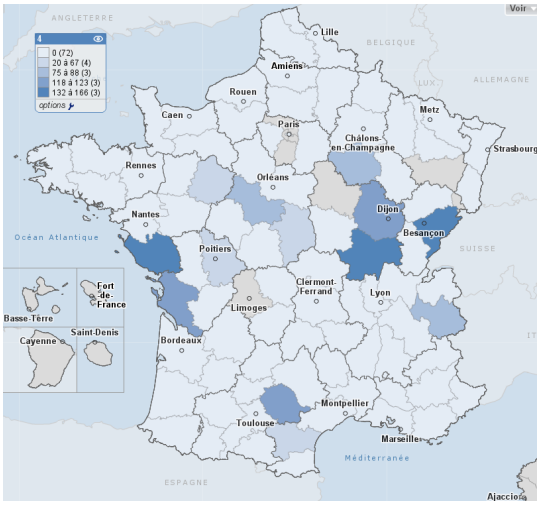
The next chapter is a perspective work which presents a hierarchical version of the GNG algorithm, called GH-Stream. The GH-Stream method is based on a topological and hierarchical structrure in order to deal with streaming data.



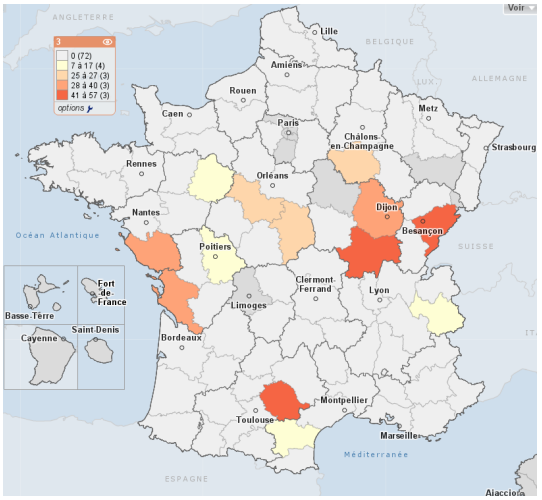
(a) All contrats



(b) Claimed contracts

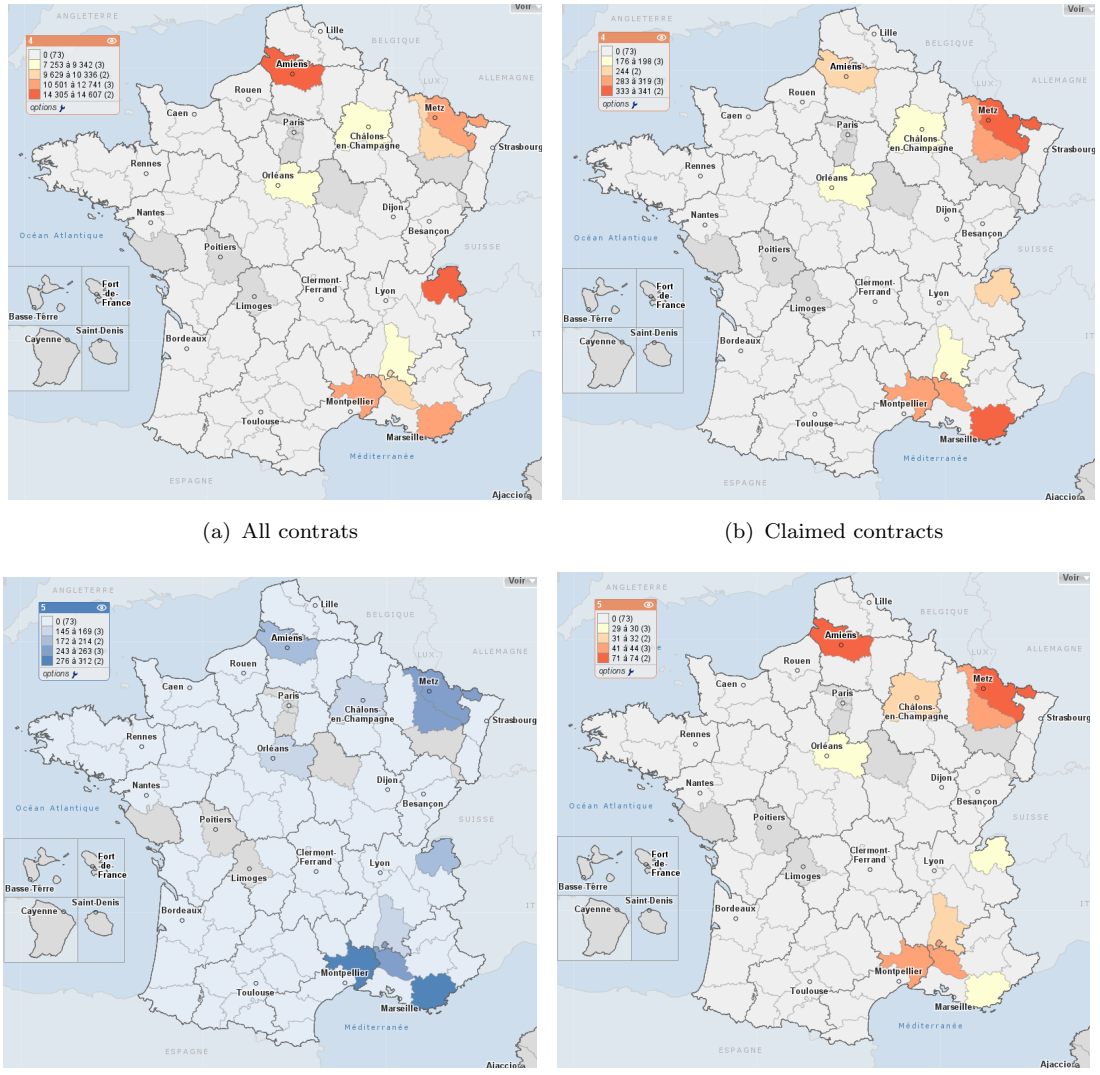


(c) Claimed contracts WAT



(d) Claimed contracts FIR

FIGURE 7.3: Visualtisation of contracts assigned to cluster #21



(a) All contrats

(b) Claimed contracts

(c) Claimed contracts WAT

(d) Claimed contracts FIR

FIGURE 7.4: Visualtization of contracts assigned to cluster #55

Chapter 8

Growing Hierarchical Trees for Data Stream Clustering and Visualization

In this chapter, we present our third contribution which is a new approach using a hierarchical and topological structure (or network) for both clustering and visualization. The topological network is represented by a graph in which each neuron represents a set of similar data points and neighbor neurons are connected by edges. The hierarchical component consists of multiple tree-like hierarchy of clusters which allows us to describe the evolution of a data stream, and then analyze it explicitly their similarity. This adaptive structure can be exploited by descending top-down from the topological level to any hierarchical level.

8.1 Introduction

Streaming algorithms have been introduced as a method to find patterns in continuous online data in real-time. Moreover, streaming algorithms must be capable of fast and incremental learning in order to overcome memory and time limitations. In the literature, many streaming algorithms have been adapted from clustering algorithms, e.g., the density-based method DBSCAN [Cao et al., 2006, Isaksson et al., 2012], the partitioning method k-means [Ackermann et al., 2012], the message passing-based method AP [Zhang et al., 2008], or the evolving algorithm

G-Stream [Ghesmoune et al., 2015b]. Please refer to chapter 4 for a comprehensive survey of data stream clustering methods.

Data stream clustering can be processed for the further analysis of dynamic patterns evolving over time, event tracking or future change trend detection. An attractive solution is to visualize data streams to reveal insight that may suggest further experiments to conduct. An interactive visualization should be able to express incremental information projected directly onto a low dimensional subspace. There are two main issues for visualizing streaming data concerning the total amount of data and newly arriving data.

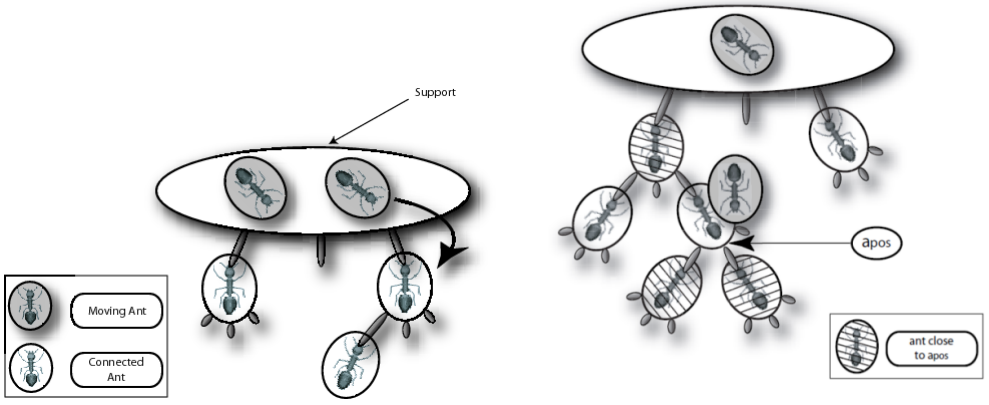
To address both data stream clustering and visualization at the same time, we propose the growing heuristic topological and hierarchical structure **GH-Stream** (Growing Hierarchical Trees over Data Stream), a variant of G-Stream which does not require the number of clusters to be specified beforehand. This type of structure consists of a topological network and multiple trees which can be exploited by descending from a general part to any particular part, i.e. from the topological level to any level of a hierarchical tree.

When new data arrive, nodes are removed or added (neurons in the topological level or tree nodes in the hierarchical level). This facilitates the visual task and adapts to the data change trends. Thus, the main contribution of this work is to present an incrementally hierarchical and topological structure that can be used to analyze data streams at any particular step.

8.2 AntTree

AntTree [Azzag et al., 2007] provides the hierarchical structure where each tree node represents one observation. The main principles are the following (Figure 8.1(a)): Initially, all observations are placed on the which corresponds to the tree roof. An observation will connect to the support or a connected observation in order to connect itself to a convenient location in the tree structure. The way to connect an observation to another depends on a similarity test (Figure 8.1(b)). Once all the observations are connected in the tree, the tree structure can be interpreted in many ways.

Considering the clustering problem, during the assembly of the structure, each observation $\mathbf{x}_i$ will be either:



(a) General principle of AntTree for tree building with self-assembly rules (an observation is represented by an ant). [Azzag et al., 2007]

(b) Connecting rules to find the nearest ant

FIGURE 8.1: AntTree principles

- moving on the tree: $\mathbf{x}_i$ moving over the support or over an other observation denoted by $\mathbf{x}_{pos}$, but $\mathbf{x}_i$ is not connected to the structure. It is thus completely free to move on the support or toward another observation within its neighborhood. If $\mathbf{x}_{pos}$ denotes the observation where $\mathbf{x}_i$ is located on, then $\mathbf{x}_i$ will move randomly to any immediate neighbors of $\mathbf{x}_{pos}$ in the tree.
- connected to the tree: $\mathbf{x}_i$ can no longer move anymore from the structure. Each observation has only one connection with other ants.

8.3 Growing Hierarchical Trees for Data Stream

The implementations of GH-Stream are strongly influenced by clustering tasks and visualization objectives. GH-Stream is developed using several rules from AntTree [Azzag et al., 2007] to add a new hierarchical dimension in G-Stream (this latter algorithm is presented in chapter 5).

In terms of human perception, a hierarchical tree is an efficient and optimal representation of a data structure. We are interested in particularly in AntTree to model artificial ants to build automatically complex structures. Due to the self-assembly rules defined by AntTree, this approach can be adapted to the self-organizing models.

As an online clustering algorithm, GH-Stream is able to find data patterns in large datasets evolving over time. Furthermore, as a visualization framework, GH-Stream provides a solution to data stream abstraction and changes necessarily

over time to reflect the stream evolution. With a dynamic two-level structure, we present an evolving visualization of a continuous data stream. Such hierarchical and topological structures have been studied for visualization in [Doan et al., 2012, 2013]. In the sequel, we will show how to benefit from this structure for visualizing evolving data streams as we are able to create various views for different time intervals. Here, the new view is modified from the old one ensuring that the user is able to perceive the differences between the two.

8.3.1 Dynamic multi-level structure for clustering

The proposed structure is illustrated in Figure 8.2. Several elements can be found in this structure:

- *Network* describes the topological space where data will be mapped discretely. Note that this network will extend if new data points arrive.
- *Neuron* or network node (square node) represents a cluster in the topological space. Each neuron is associated with a prototype (or a weight vector) to which input data are assigned, and with a hierarchical *tree*. This neuron is also the tree root to which the first tree nodes connect. Here the tree structure will evolve if new data points arrive.
- *Topological link* is created between a pair of neurons if they are considered as neighbors due to a given neighborhood function. A variable exists to control this type of link.
- *Tree node* (circle or triangle node) corresponds to a data point in the projected space. Data in old streams are represented by circle nodes and newly arriving data by triangle nodes.
- *Hierarchical link* is created between a pair of tree nodes if it satisfies a similarity test.

The proposed structure allows for effective visualization of tree nodes which represent input data. A test is applied in order to verify the similarity between a pair of data. A hierarchical link is formed if and only if these two are similar; otherwise a new subtree is created and is considered as a new sub-cluster. Thus

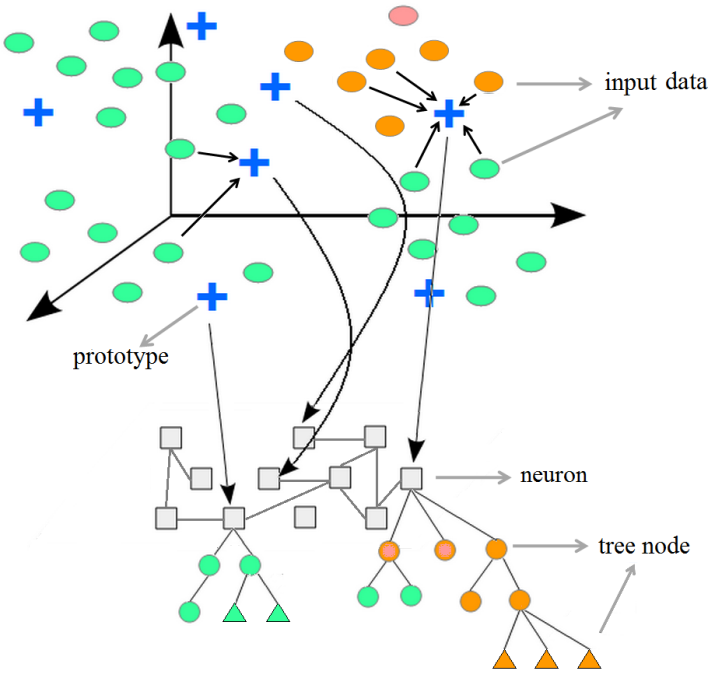


FIGURE 8.2: Hierarchical and topological structure.

data from new streams which are continuously and recursively grouped in a sub-tree are close to those from old streams. GH-Stream is able to detect clusters and represent these clusters in a topological and hierarchical structure. The confidence in each cluster may be easily observed because of hierarchical relations between the data.

8.3.2 GH-Stream

In this section, we give the algorithmic details of GH-Stream. Suppose that a data stream is denoted by $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$ of n (potentially infinite) data streams arriving in times $t_1, t_2, ..., t_n$, where $\mathbf{x}_i = (x_i^1, x_i^2, ...x_i^d)$ is a vector in $\mathbb{R}^d$. The proposed network $\mathcal{C}$ consists of neurons, each neuron $c \in \mathcal{C}$ is associated with a tree $tree_c$ and with a prototype $\mathbf{w}_c \in \mathbb{R}^d$.

In Algorithm 21, GH-Stream is divided into four main steps: 1) initialization, 2) assignment, 3) tree construction, and 4) adaptation.

Algorithm 21: GH-Stream

Data: $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$
Result: $tree, \mathcal{C}$

```
Initialize the network C associated with two trees and create an empty
  reservoir R (a list contains disconnected tree nodes) ;
while there is a data point to proceed do
    x_i ← the next point in the current data stream ;
    find c_0 using Equation 8.1; then find tree_0, the tree associated to c_0;
    constructTree(tree_0, x_i, R) ;
    adaptation(tree, x_i) ;
    remove outdated and isolated neurons (trees), and put all disconnected
      tree nodes into R ;
    if R is full then
        constructTree(tree_0, x_i, R) ;
```

8.3.2.1 Initialization step

At the beginning, GH-Stream is randomly initialized with only a 2-tree network in which these neurons are connected by a topological link. During the learning process, the network evolves and adapts to cover the data patterns. Thus, the GH-Stream network is more flexible and able to overcome the sensitivity to the topology.

8.3.2.2 Assignment step

As a new data point is reached, the nearest and the second-nearest neurons are identified, linked by an edge, and the nearest neuron and its topological neighbors are moved toward the data point. This assures that the quantization error of the current stream is minimized with respects to the data assuming that the prototype vectors are constant. Equation (8.1) is used to find the nearest node.

$$c_0 = \arg \min_{j=1,...,k} \|\mathbf{x}_i - \mathbf{w}_{c_j}\|^2 \tag{8.1}$$

where k is the current number of trees in the network.

Algorithm 22: constructTree

Data: $tree_0, \mathbf{x}_i$
Result: $tree_0$

if *less than 2 tree nodes are connected to the root of* $tree_0$ **then**
  connect $\mathbf{x_i}$ to the root of $tree_0$;
else
  $T_{root} = max(d(\mathbf{x}_i, \mathbf{x}_j))$; // where $\mathbf{x}_i$ and $\mathbf{x}_j$ are any pair of data connected to the root of $tree_k$; $d(\mathbf{x}_i, \mathbf{x}_j) = ||\mathbf{x}_i - \mathbf{x}_j||^2$, $\mathbf{x}_i, \mathbf{x}_j$ are normalized
  $\mathbf{x}^+ = \arg\min_r d(\mathbf{x}_i - \mathbf{x}_r)$; // $\forall \mathbf{x}_r$ is connected to the root of $tree_0$
  if $d(\mathbf{x}_i, \mathbf{x}^+) > T_{root}$ **then**
    disconnect $\mathbf{x}^+$ from the root ; // disconnect recursively $subtree_{\mathbf{x}^+}$
    put $subtree_{\mathbf{x}^+}$ into $\mathcal{R}$;
    if $\mathbf{x}_i$ *is disconnected* **then**
      $subtree_{\mathbf{x}_i} \leftarrow$ all nodes recursively connected to $\mathbf{x}_i$ before its disconnection ;
    connect $\mathbf{x}_i$ and $subtree_{\mathbf{x}_i}$ to $\mathbf{x}^{pos}$; // The subtree structure is kept as it was before the disconnection
  $connectRecursive(tree_0, \mathbf{x}_i, \mathbf{x}^+)$;

Algorithm 23: connectRecursive

Data: $tree_0, \mathbf{x}_i, \mathbf{x}^{pos}$
Result: $tree_0$

if *no tree nodes connected to* $\mathbf{x}^{pos}$ **then**
  $tree_0 = \text{connectSubTree}(tree_0, \mathbf{x}_i, \mathbf{x}^{pos})$;
$\mathbf{x}^+ = \arg\min_{\mathbf{x}_r} d(\mathbf{x}_i, \mathbf{x}_r)$; // $\forall \mathbf{x}_r$ is connected to $\mathbf{x}^{pos}$
if $d(\mathbf{x}_i, \mathbf{x}^{pos}) > d(\mathbf{x}_i, \mathbf{x}^+)$ **then**
  $connectRecursive(tree_0, \mathbf{x}_i, \mathbf{x}^+)$;
if $\mathbf{x}_i$ *is disconnected* **then**
  $subtree_{\mathbf{x}_i} \leftarrow$ all nodes recursively connected to $\mathbf{x}_i$ before its disconnection ;
connect $\mathbf{x}_i$ and $subtree_{\mathbf{x}_i}$ to $\mathbf{x}^{pos}$; // The subtree structure is kept as it was before the disconnection

8.3.2.3 Tree construction step

Here we show how to adapt the self-assembly rules inspired by AntTree. During the learning process, the status of a tree node can be varied due to the connecting or disconnecting rules. Therefore, we define three possibilities for the tree node status:

1. *Initial* : when a new data point arrives, its initial status is the default;
2. *Connected* : A tree node is currently connected to another node;
3. *Disconnected* : A tree node which was connected at least once but now is disconnected. We denote a resevoir $+\mathcal{R}$ to contain all disconnected nodes.

Once the data has been assigned to the nearest tree, they will take part in building trees. Data have to pass through the similarity test in Algorithms 22 and 23. At first, the tree is empty and since the similarity test can only be computed with at least two tree nodes, then the first two tree nodes are automatically connected to a tree as in the first test (Line 1 in Algorithm 22).

The second test (Line 6 in Algorithm 22) is used to find the best position in the hierarchical structure for each data point. It can be either to create a new subtree at the current tree node or pass top-down to become a leaf node.

During the learning process, there is a chance that objects could be *disconnected*. Concerning the disconnection, there are two distinct cases:

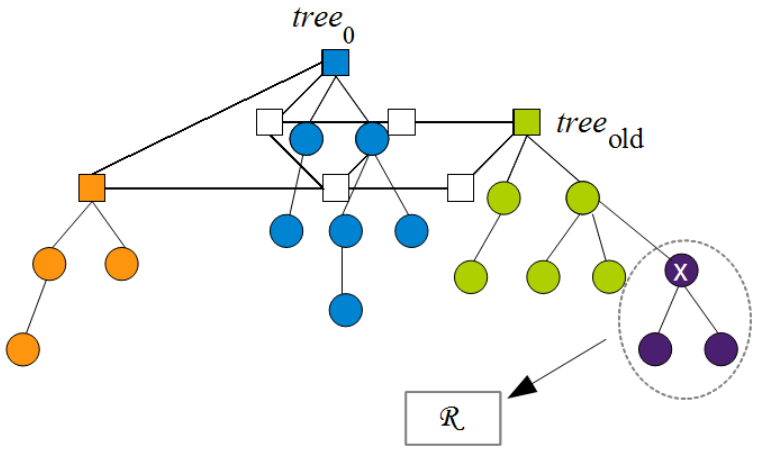
1. remove a tree node (Line 7 in Algorithm 21),
2. disconnect tree node(s) (Line 7 in Algorithm 22).

Whenever a tree node is disconnected from a tree, we have to check whether other child nodes exist in $subtree_{\mathbf{x}_i}$. If this is the case, we disconnect all of them from the specific $subtree_{\mathbf{x}_i}$. A simple example of disconnection for a group of nodes (or sub-tree) is depicted in Figure 8.3(a).

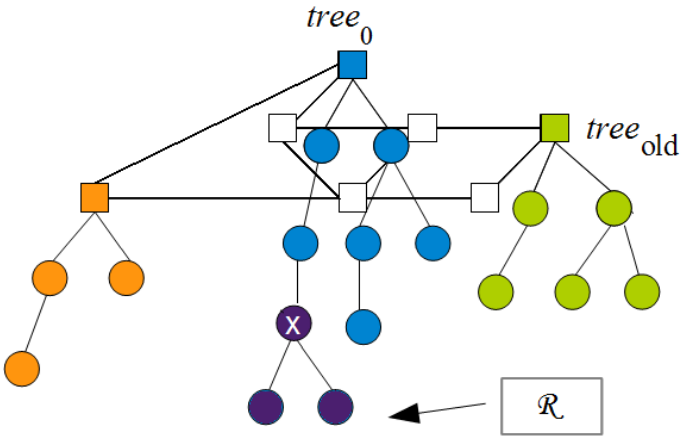
Given $tree_{old}$ as in this example, the tree node $\mathbf{x}$ consisting of three violet nodes is disconnected from this tree. All the nodes connected to $\mathbf{x}$ must be recursively disconnected too (Line 8 in Algorithm 21 or Line 7 in Algorithm 22); it applies to two child nodes of $\mathbf{x}$. Therefore $subtree_{\mathbf{x}}$ has *disconnected* status and is immediately put onto the list $\mathcal{R}$.

Suppose that a disconnected $\mathbf{x}_i$ becomes *connected* at a moment, we will keep this subtree structure by re-connecting these child nodes together (Line 12 in Algorithm 22 or Line 11 in Algorithm 23); hence this method can accelerate the learning process.

For example, let us take again the example in Figure 8.3(a). After obtaining the new assignment, $\mathbf{x}$ connects to $tree_0$. This implies that the child nodes of $\mathbf{x}$



(a) Disconnect $subtree_{\mathbf{x}}$ from $tree_{old}$ and put it into $\mathcal{R}$



(b) Re-connect $subtree_{\mathbf{x}}$ to $tree_0$

FIGURE 8.3: Rules to build a hierarchical structure. Neuron is colored according to a majority vote of data gathered within this neuron.

have $tree_0$ as their best match tree too. We systematically connect this subtree to $tree_0$ and the result is shown in Figure 8.3(b). Recall that this subtree is not kept till the end of learning as the nodes in the subtree may be disconnected in next iterations.

8.3.2.4 Adaptation step

Once a data point has been assigned to a prototype, this prototype and its neighbors are adjusted and moved toward the assigned object according to the "winner take most" rule [Fritzke, 1991].

In most data stream scenarios, more recent data can reflect the emergence of new trends or changes in the data distribution [de Andrade Silva et al., 2013].

Algorithm 24: adaptation: network adaptation

Data: $tree, subtree_{\mathbf{x}_i}, \mathcal{C}$
Result: $tree, \mathcal{C}$

add the squared distance to a local error counter variable, $error(tree_0)$, according to Equation (8.4);
move $\mathbf{w}_0$ and its topological neighbors towards $\mathbf{x}_i$:

$$\Delta_0 = \sum_{j \in subtree_{\mathbf{x}_i}} \epsilon_b(\mathbf{x}_j - \mathbf{w}_0)$$

$$\Delta_r = \sum_{j \in subtree_{\mathbf{x}_i}} \epsilon_r(\mathbf{x}_j - \mathbf{w}_r)$$

$\forall r$ is neighbor to c_0

find the second nearest tree $tree_1$ of $\mathbf{x}_i$;
if *$tree_0$ and $tree_1$ are connected by an edge* **then**
    set the age of that edge to 0 ;
create a new edge between them ;
remove the edges with an age larger than Max_{age} ;
decrease the error of all neurons ;
find two neurons with the largest accumulated error ;
insert new neurons in the half-way between these two ;
update the edges connecting to these two and decrease their error ;

There are three window models commonly studied in data streams: landmark, sliding and damped (as presented in chapter 4).

We consider, like many others, the damped window model, in which the weight of each data point decreases exponentially with time t via a fading function

$$f(t) = 2^{-\lambda_1(t-t_0)} \tag{8.2}$$

where $\lambda_1 > 0$, defines the rate of decay of the weight over time, t denotes the current time and t_0 is the timestamp of the data point. Note that data points are passed according to the sliding windows principle. We use the number of the window to mark the timestamps of data points belonging to this window. The weight of a neuron is based on data points associated with it:

$$\pi_c = \sum_{i=1}^{n_c} 2^{-\lambda_1(t-t_{i_0})} \tag{8.3}$$

where n_c is the number of points assigned to the node c at the current time t. If the weight of a neuron is less than a threshold value then this node is considered

as outdated and then deleted (with its links). This task is assured by Line 7 in Algorithm 21.

The error variable of the nearest node is updated according to Equation (8.4)

$$error(tree_0) = error(tree_0) + \|\mathbf{x}_i - \mathbf{w}_0\|^2. \tag{8.4}$$

8.3.3 Complexity

Algorithm 21 is repeated n times (n data points) to complete the learning process. For each time, there are three operations: assignment, tree construction, adaptation. The assignment and adaptation processes require one operation for each data point, but the tree construction requires $\log n$ operations. To summarize, GH-Stream has the complexity of $O(n \log n)$.

8.4 Experimental evaluations

This section is devoted to the experiments to illustrate the proposed model for data stream clustering and visualization. Our experiments were performed on the MATLAB platform using real-world and synthetic datasets.

8.4.1 Datasets

Datasets	#observations	#features	#classes
COIL100	7,200	1,024	100
DS1	9,153	2	14
Hyperplane	100,000	10	5
Letter4	9,344	2	7
Sea	60,000	3	2

TABLE 8.1: Data features

The experiments are performed using real-world and synthetic datasets. Table 8.1 overviews all the dataset features.

- COIL100 is available in http://www.cs.columbia.edu/CAVE/software/softlib/coil-100.php. This dataset contains images of 100 different objects with 72 images per object.

- DS1 is a synthetic dataset found in http://impca.curtin.edu.au/local/software/synthetic-data-sets.tar.bz2.
- Hyperplane and Sea are datasets with concept drift. These two are available in http://www.win.tue.nl/~mpechen/data/DriftSets/.
- Letter4 is generated by a Java code https://github.com/feldob/Token-Cluster-Generator.

8.4.2 Evaluation and performance comparison

In this subsection, we performed extensive experiments to evaluate the GH-Stream performance for data stream clustering. The experimental parameters for all the datasets are 600 data per stream, epoch = 300 (after 300 iterations, new neurons are added into the network), and MaxAge = 250. Due to the nature of different algorithms, they output different number of clusters at the end of learning process.

Datasets	GNG	G-Stream	StreamKM++	CluStream	GH-Stream
COIL100	0.323	0.233	**0.427**	0.373	0.374
COIL100	± 0.009	±0.009	± **0.015**	± 0.034	± 0.005
DS1	0.511	**0.993**	0.675	0.701	0.970
DS1	± 0.251	± **0.006**	± 0.018	± 0.028	± 0.010
HyperPlan	0.423	0.396	0.425	**0.438**	0.427
HyperPlan	± 0.002	± 0.005	± 0.000	± **0.008**	± 0.003
Letter4	0.577	0.991	0.687	0.934	**0.997**
Letter4	± 0.201	± 0.001	± 0.026	± 0.026	± **0.003**
Sea	0.838	0.788	0.824	0.822	**0.839**
Sea	± 0.002	± 0.009	± 0.001	± 0.006	± **0.002**

TABLE 8.2: Competitive performance of different approaches in terms of Accuracy

The GH-Stream efficiency is evaluated with different algorithms using three quality criteria: Accuracy, Normalized Mutual Information (NMI), and Rand Index. Each criterion should be maximized. Each method is run 10 times with random initializations and the Tables 8.2, 8.3, and 8.4 show the average and standard deviation of quality criteria over these 10 runs.

For the selected datasets, we notice that our GH-Stream provides good clustering results comparing to other methods. GH-Stream generally outperformed the others in term of quality criteria NMI and Rand index in most of cases such

Datasets	GNG	G-Stream	StreamKM++	CluStream	GH-Stream
COIL100	0.655	0.577	0.606	0.671	**0.687**
COIL100	± 0.004	±0.007	±0.0231	± 0.011	± **0.007**
DS1	0.491	0.712	0.702	0.723	**0.730**
DS1	± 0.132	± 0.004	± 0.021	± 0.022	± **0.007**
HyperPlan	0.018	0.010	**0.020**	0.017	0.019
HyperPlan	± 0.001	± 0.002	± **0.000**	± 0.004	± 0.001
Letter4	0.529	0.607	0.553	0.264	**0.657**
Letter4	± 0.074	± 0.002	± 0.022	± 0.034	± **0.006**
Sea	0.138	0.146	**0.164**	0.158	0.148
Sea	± 0.001	± 0.004	± **0.000**	± 0.009	± 0.001

TABLE 8.3: Competitive performance of different approaches in terms of NMI

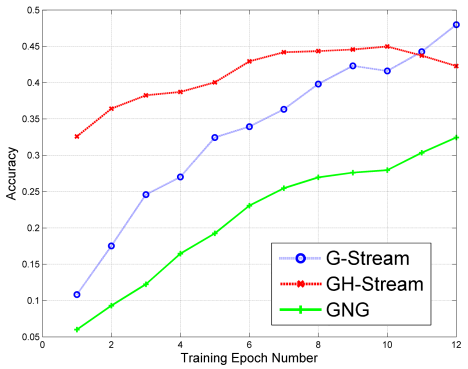
Datasets	GNG	G-Stream	StreamKM++	CluStream	GH-Stream
COIL100	0.973	0.921	0.883	0.977	**0.979**
COIL100	± 0.008	±0.012	±0.003	± 0.001	± **0.001**
DS1	0.621	0.846	0.844	0.845	**0.854**
DS1	± 0.122	± 0.001	± 0.004	± 0.007	± **0.001**
HyperPlan	0.704	0.667	0.603	0.652	**0.705**
HyperPlan	± 0.000	± 0.000	± 0.000	± 0.001	± **0.000**
Letter4	0.686	0.812	0.794	0.341	**0.818**
Letter4	± 0.084	± 0.001	± 0.014	± 0.004	± **0.002**
Sea	0.470	**0.507**	0.470	0.491	0.471
Sea	± 0.001	± **0.001**	± 0.006	± 0.003	± 0.000

TABLE 8.4: Competitive performance of different approaches in terms of Rand index

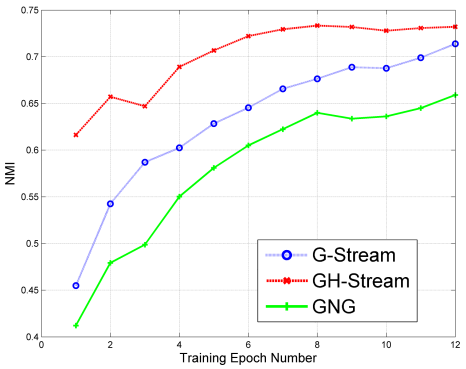
as COIL100, DS1, Letter4 datasets. On the other hand, GH-Stream gives comparable accuracy results.

In the direct comparison with G-Stream, GH-Stream uses less input parameters. For more specific, GH-Stream does not require the distance threshold for the similarity test. According to Tables 8.2, 8.3, and 8.4, GH-Stream is better in many cases.

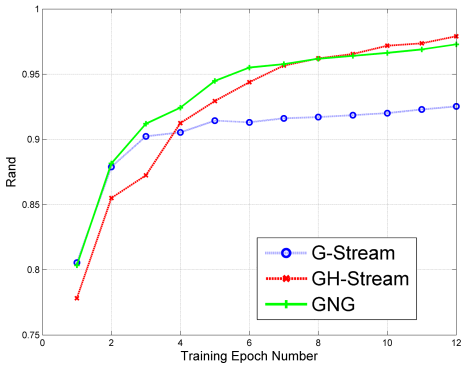
Some other experiments are carried out to analyze the clustering evolution during the learning process. In Figure 8.4 and 8.5, we show the changes in the quality criteria over time for two datasets COIL100 and Hyperplane. The GH-Stream performance dominates over G-Stream and GNG with respects to Accuracy and NMI in the beginning of learning. Moreover, GH-Stream also provides further information on data visualization which will be studied thoroughly in the next subsection.



(a) Accuracy



(b) NMI



(c) Rand index

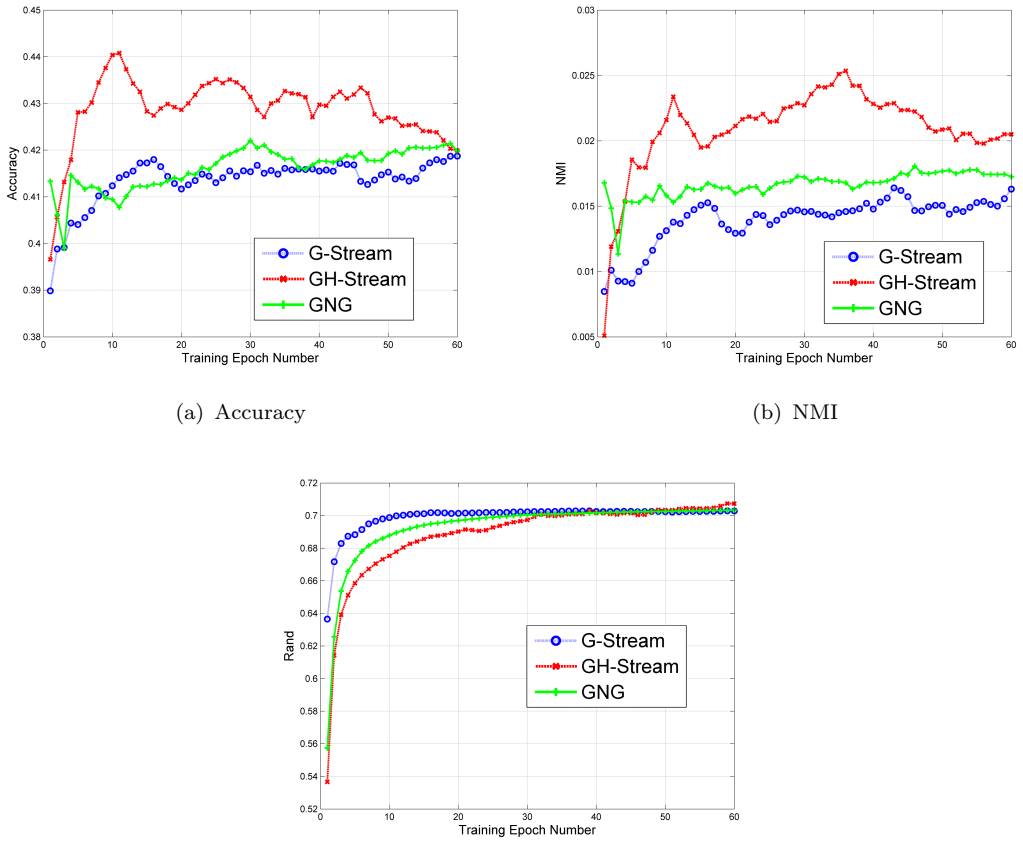
FIGURE 8.4: Performance of difference methods vs number of epoch over time during the learning process for COIL100

8.4.3 Visualization of tree evolution

GH-Stream does not only provide an informative clustering but also is a useful tool for data stream visualization based on the Tulip graph visualization [Auber, 2003]. Using the GEM layout, we provide multiple views to describe the changes in data stream clustering.

At t_i, there are data from old streams and newly arriving data from the current stream. By using different symbols (square for data from old streams and triangle for data from the current stream), it is quite easy to anticipate the differences among the visualizations provided by GH-Stream.

Take DS1 in Figure 8.6 as a visual example. The data in the reservoir $\mathcal{R}$ can be clearly seen as the isolated nodes (bottom left in this figure). Due to the self-assembly rules, new data arrive and connect to those in the same class. A good classification is indicated by the hierarchical trees or subtrees with the same color



(a) Accuracy

(b) NMI

(c) Rand index

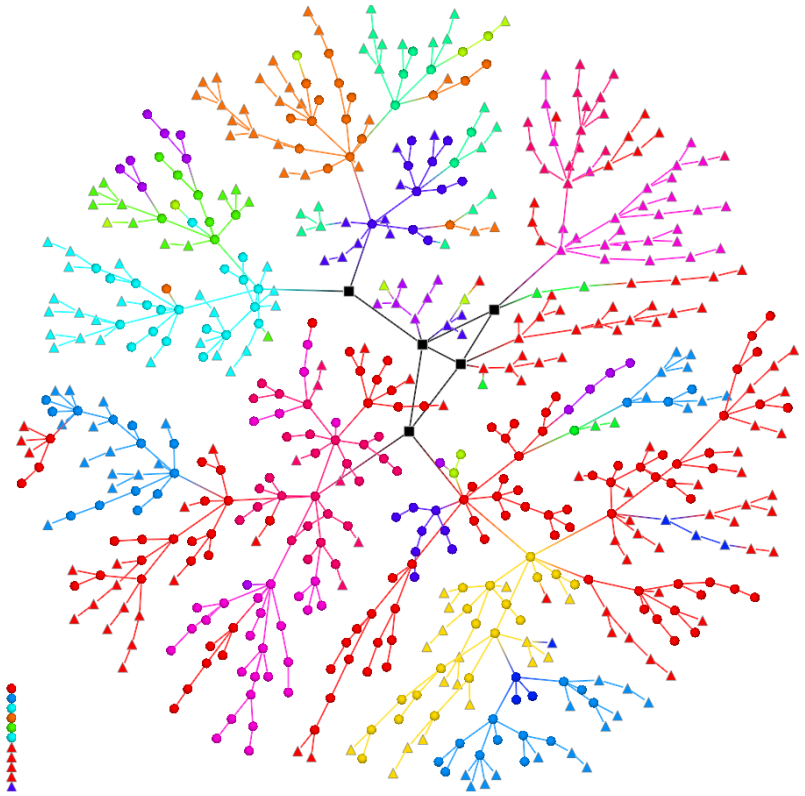
FIGURE 8.5: Performance of different clustering methods vs number of epochs during the learning process for Hyperplane

found in the proposed structure.

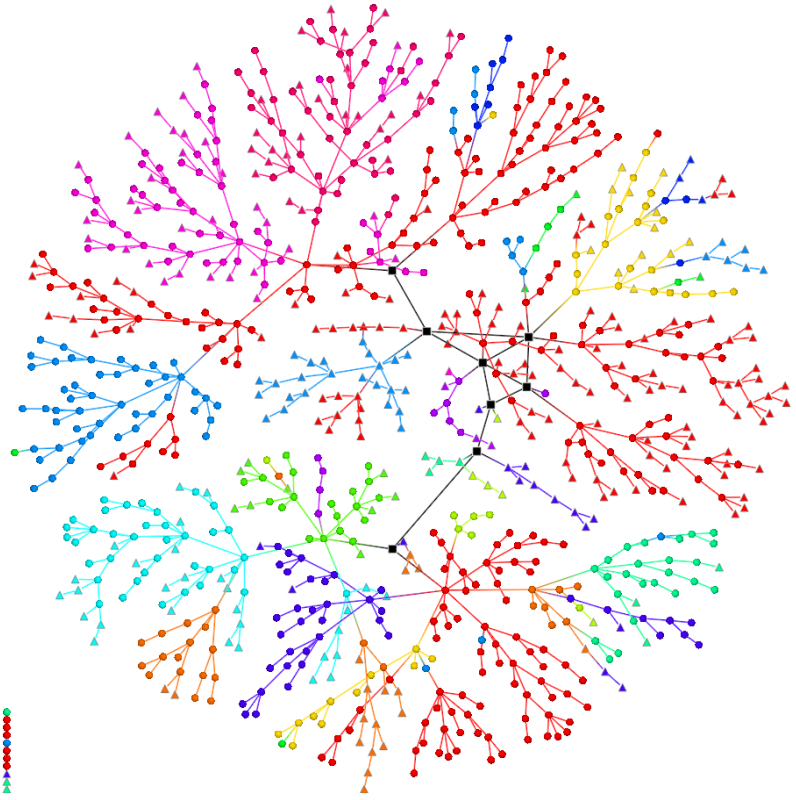
However, some regions are colored with different colors due to the fact that in the streaming algorithm, data are assigned only once so it is not possible to correct misclassifications, which requires further investigation. In practise, with an interactive visual tool, users are able to interpret and/or correct misclassified data points by moving their respective subtree and creating new clusters.

Figure 8.7 shows the visual result after learning all 7200 images from the dataset COIL100. Many regions with a single color can be observed again as in the previous example. In this case, each data point corresponds to an image. When we zoom as in Figure 8.7(b) to visualize in depth the similarity in the hierarchical structure, we see that images containing a cup belong to one class; in addition, all 72 images of this class are found in the same tree (the same group).

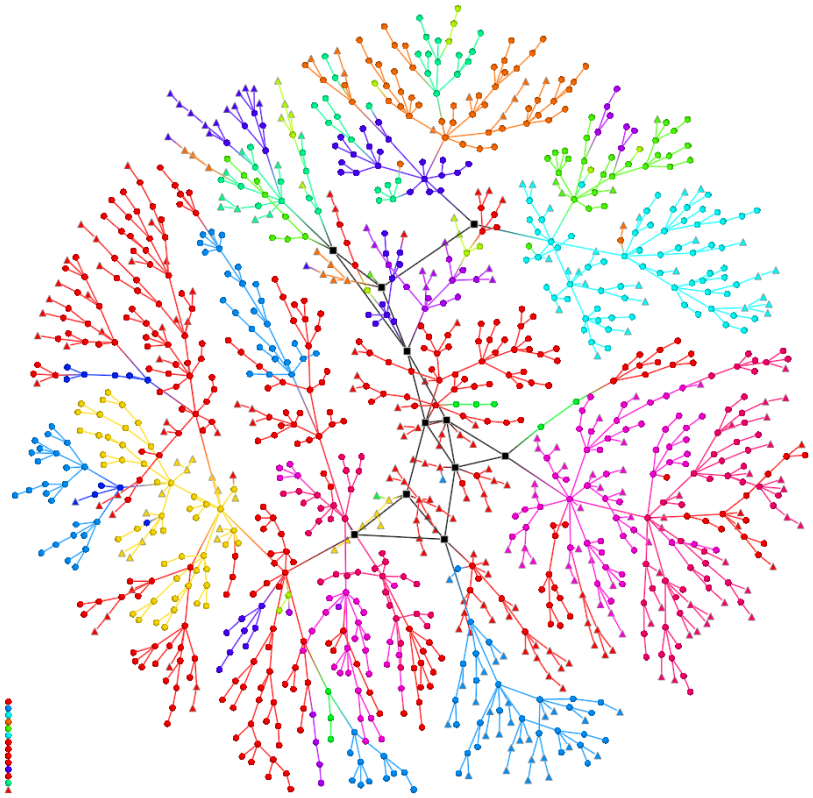
Thus, a couple of questions are raised: what are the neighbors of this tree? Are images found in these neighbor trees similar? To answer this question, another



(a) Visualization at t_2 (300 tree nodes in a 5-tree network)



(b) Visualization at t_3 (600 tree nodes in a 8-tree network)



(c) Visualization at t_4 (900 tree nodes in a 11-tree network)

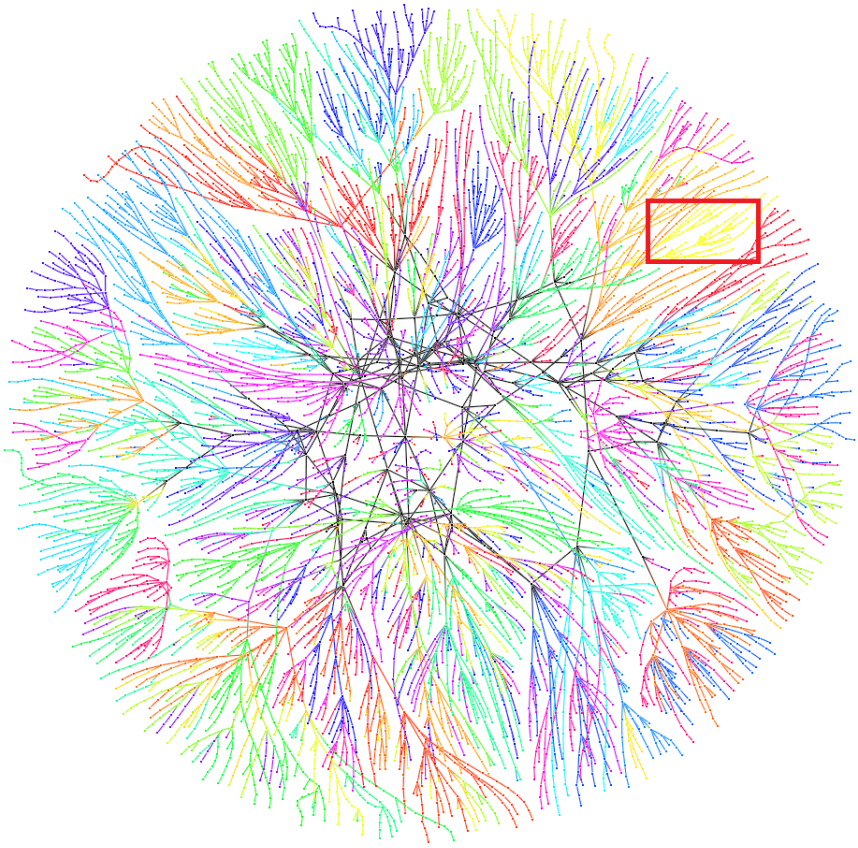
FIGURE 8.6: Visualization of the DS1 dataset. Each class is represented by a single color.

zoom taken from Figure 8.7(a) is shown in Figure 8.8. In this figure, images from different objects are put in the same groups but it is noteworthy that they have a similar shape/form such as a cup or a box. To summarize, GH-Stream is a convenient tool for visual tasks for data stream mining.

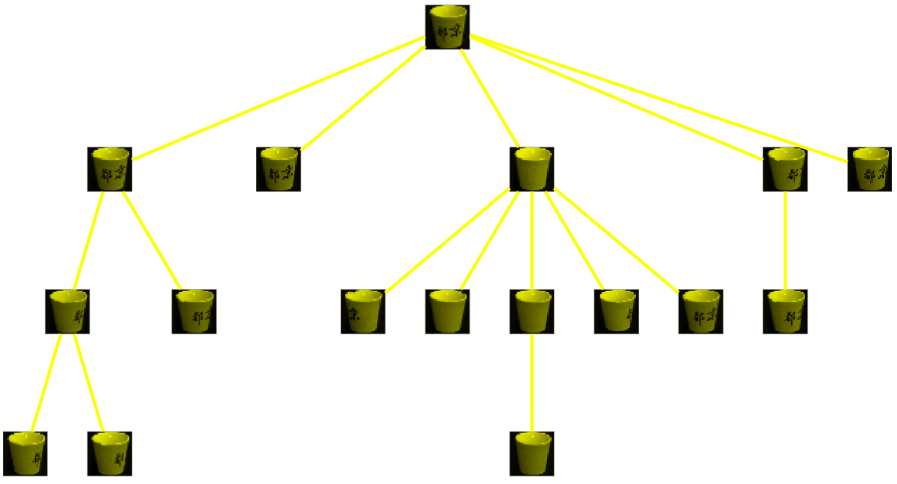
8.5 Conclusion

In this chapter, we have proposed a new clustering approach with the objective of a data stream visualization which adapts a neural network to a hierarchical and topological space. GH-Stream offers a good clustering performance as well as an efficient visualization to deal with the data that arrive over time in streams. GH-Stream is able to detect new classes and output satisfactory results. We also studied thoroughly the visual results and showed how to exploit the proposed structure.

In the future, GH-Stream will be improved to be more automatic which is an



(a) Complete visualization after the learning process (7200 tree nodes in a 127-tree network)



(b) Zoom sample extracted from Figure 8.7(a). These images are in the same class "cup"

FIGURE 8.7: Visualization of the COIL100 dataset. Each class is represented by a unique color.

important goal of data stream mining. Another perspective is to enhance the visualization component for Big Data so that GH-Stream can provide more degrees of freedom. One promising direction is to employ TreeMap in order to overcome

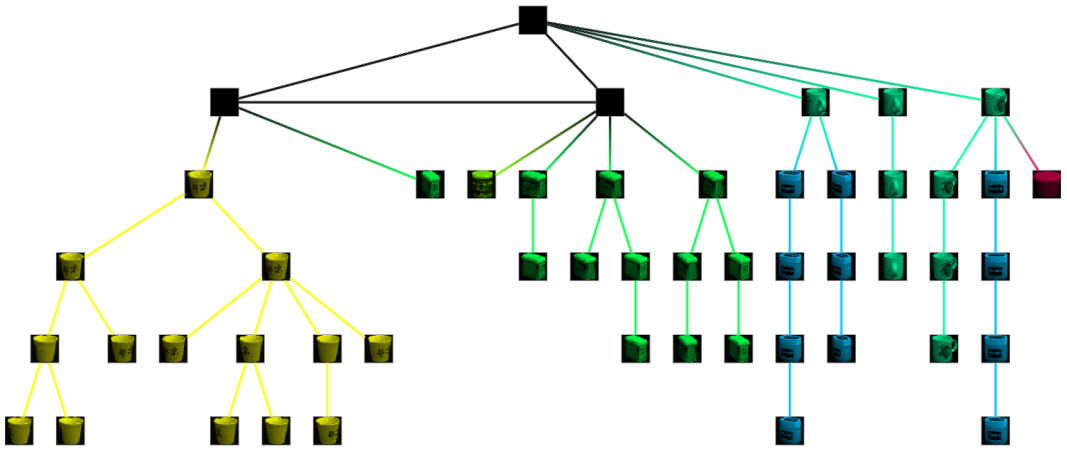


FIGURE 8.8: Zoom sample extracted from Figure 8.7(a). A 3-tree network shows both hierarchical and topological relations.

the complexity and redundancy problems in visualization.

The work presented in this chapter has resulted in the following publications: Nhat-Quang Doan, Mohammed Ghesmoune, Hanane Azzag, and Mustapha Lebbah. Growing hierarchical trees for data stream clustering and visualization. In 2015 International Joint Conference on Neural Networks, IJCNN 2015, Killarney, Ireland, July 12-17, 2015, pages 1–8, 2015. doi: 10.1109/IJCNN.2015. 7280397. URL http://dx.doi.org/10.1109/IJCNN.2015.7280397.

Chapter 9

Conclusion and perspectives

After summarizing the key issues touched upon this work, the next section discusses the main research avenues open for further work.

Summary

The first chapters were devoted to giving an introduction to the Big Data ecosystem and the state-of-the-art on both clustering and scalable methods using the MapReduce paradigm and clustering data streams.

Our first contribution is concerned with extending the GNG approach to deal with streaming data. The one-pass streaming clustering algorithm titled G-Stream (Growing Neural Gas over Data Streams) is presented. G-Stream, as a "sequential" clustering method, allows us to discover clusters of arbitrary shapes without any assumptions on the number of clusters. In G-Stream, an exponential fading function is used to reduce the impact of old data whose relevance diminishes over time. For the same reason, links between nodes are also weighted by an exponential function.

Afterwards, in the second contribution, we presented the "batchStream" distributed algorithm for scalable clustering data streams. We defined a new cost function taking into account the subsets of observations arriving in batches. After that, we proposed a model for scalability. This model consists of decomposing the data stream clustering problem into the elementary functions, Map and Reduce. Its implementation is assured in the Spark Streaming platform.

Then, we presented our work carried in the context of the Big Data project,

named *Square Predict*. First, we presented the architecture of the proposed Big Data framework. After that, we illustrated the utility of the batchStream algorithm as an unsupervised learning for an insurance Big Data.

In the previous chapter, we presented our third contribution which is a new approach using a hierarchical and topological structure for both clustering and visualization. The topological network is represented by a graph in which each node represents a set of similar data points and neighbor nodes are connected by edges. The hierarchical component consists of a multiple tree-like hierarchy of clusters which allow to describe the evolution of a data stream, and then to analyze explicitly their similarity.

Perspectives

Clustering binary data streams

We start by surveying some relevant algorithms proposed in the literature to deal with the problem of clustering binary data streams; then we present our scalable model for clustering binary data streams.

[Ordonez, 2003] proposed several improvements for k-means to cluster binary data streams. They showed that sufficient statistics are simpler for binary data. Distance computation is optimized for sparse binary vectors. A summary table with best cluster dimensions and outliers is maintained on-line.

[Babcock et al., 2003] presented an extension of the k-medians algorithm, based on the *exponential histogram* data structure, for data stream clustering under the sliding window model. This maintains statistics or information for the most recent N observations that is growing in real time, while operating with memory that is asymptotically smaller than the window size.

[Charikar et al., 2003] proposed a randomized algorithm for the k-medians problem which produces a constant factor approximation in one pass using poly-logarithmic space. HUE-Stream Meesuksabai et al. [2011] is designed to address uncertainty in heterogeneous data streams, i.e., including numerical and categorical attributes simultaneously.

We aim to extend the batchStream algorithm, presented in chapter 6, to deal with binary data streams. Thus, we wish to extend this scalable algorithm implemented in MapReduce to address binary data streams. As explained in chapter 2,

designing a MapReduce algorithm requires defining the Map and Reduce elementary functions.

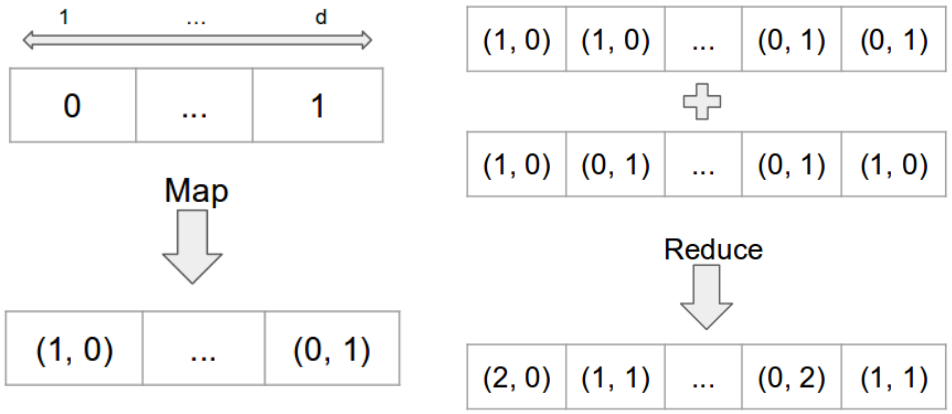
Our proposition is based on the works presented in [Lebbah, 2003, Govaert, 2009] for clustering binary data, as follows. Given a data stream consisting of a sequence $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$ of n (potentially infinite) observations arriving at times $t_1, t_2, ..., t_n$, where $\mathbf{x}_i = (x_i^1, x_i^2, ..., x_i^d)$ is a vector in $\{0,1\}^d$. We denote by $\mathbf{X}_1 = \{\mathbf{x}_1, ..., \mathbf{x}_p\}$ where p is the size of the window, thus $\mathcal{X} = \{\mathbf{X}_1, \mathbf{X}_2, ..., \mathbf{X}_L\}$. At each time, batchStream is represented by a graph $\mathcal{C}$ where each node represents a cluster. Each node $c \in \mathcal{C}$ has:

- a prototype $\mathbf{w}_c = (w_c^1, w_c^2, ..., w_c^d)$ which is a vector in $\{0,1\}^d$, representing its position;
- π_c representing the weight of this node;
- $error(c)$ an error variable representing the sum of distances between this node and the data-points assigned to it.

Our proposed main idea is that:

- At each time t, the Map function receives a micro-batch, $\mathbf{X}_t$, of observations then for each observation, $\mathbf{x}_{ti}$, it generates a *key/value* pair. The nearest node, bmu_1, is saved as a *key*. The corresponding *value* is the tuple $(bmu_2, \mathbf{x}_{ti}, vectPair, 1)$ where bmu_2 and $\mathbf{x}_{ti}$ are vectors in $\{0,1\}^d$. The variable $vectPair$ is used to generate statistics about the $\mathbf{x}_{ti}$'s: for each x_{ti}^j if $(x_{ti}^j == 1)$ return $(1,0)$ else return $(0,1)$, where $j = 1, ..., d$. In other words, the couple $(1,0)$ says that there is one zero, while the couple $(0,1)$ says that there is one one. The 1 value is used in counting.
- The Reduce function groups by bmu_1 and sums the corresponding values.

The Map and Reduce functions for clustering binary data stream are illustrated in Figure 9.1. When updating the model, for each couple $(v_1, v_2)^j$ resulting from the Reduce step, if $v_1 > v_2$ the we return 1 else we return 0; so that the result is also a binary vector in $\{0,1\}^d$, where $j = 1, ..., d$.



(a) The Map function (b) The Reduce function

FIGURE 9.1: The Map and Reduce functions for clustering binary data streams.

Open challenges in data stream clustering

In today's applications, evolving data streams are ubiquitous. Mining, knowledge discovery, or more specifically clustering streaming data is a recent domain compared to the offline (or batch) model. Thus, many of the challenges, issues and problems remain to be addressed in the streaming model. This section is devoted to discuss some challenging, outstanding issues and further directions from the viewpoints of both academic research and industrial applications [Khalilian and Mustapha, 2010, de Andrade Silva et al., 2013, Krempl et al., 2014, Gama, 2012, 2010].

Protecting privacy and confidentiality. Data streams present new challenges and opportunities with respect to protecting privacy and confidentiality in data mining. The main objective is to develop data mining techniques that would not uncover information or patterns which compromise confidentiality and privacy obligations. Privacy-by-design seems to be a promising paradigm to use.

Handling incomplete information. The problem of missing values, which corresponds to the incompleteness of features, has been discussed extensively for the offline, static settings. However, only few works address data streams, and especially evolving data streams.

Uncertain data. In most applications we do not have sufficient data for statistical operations so new methods are needed to manage uncertain data stream in an accurate and fast manner.

Variety of data. Data type diversity in a given stream (text, video, audio, static image, etc.) as well as differences in data processability (structured, semi-structured, unstructured data). Clustering these diverse types of data together, coming in a streaming form, is very challenging. Another interesting future application of data stream clustering is social network analysis. The activities of social network members can be regarded as a data stream, and a clustering algorithm can be used to show similarities among members, and how these similar profiles (clusters) evolve over time.

Synopsis, sketches and summaries. A synopsis is compact data structures that summarize data for further queries. Samples, Histograms, Wavelets, Sketches describe basic principles and recent developments in building approximate synopses (that is, lossy, compressed representations) of massive data [Cormode et al., 2012]. Data sketching via random projections is a tool for dimensionality reduction. Although this technique is extremely efficient, its main drawback is that it may ignore relevant features.

Distributed streams. Data streams are distributed by nature. For learning from distributed data, we need efficient methods in minimizing the communication overheads between nodes. Most importantly, in applications like monitoring, centralized solutions introduce delays in event detection and reaction, that can make mining systems inefficient. Many data clustering techniques are not trivial to parallelize. To develop distributed versions of some methods, much research is needed with practical and theoretical analysis to provide new methods.

Evaluation of data stream algorithms. Although in the field of static classification such tools exist, they are insufficient in data stream environments due to such problems as: concept drift, limited processing time, verification latency, multiple stream structures, evolving class skew, censored data, and changing misclassification costs. Indeed, in the streaming context, we are interested in how the evaluation metric evolves over time [Krempl et al., 2014].

Autonomous and self-diagnosis. Knowledge discovery from data streams requires the ability for predictive self-diagnosis. A significant and useful intelligence characteristic is diagnostics, not only after failure has occurred, but also predictive (before failure) and advisory (providing maintenance instructions). The development of such self-configuring, self-optimizing, and self-repairing systems is a major scientific and engineering challenge. All these aspects require monitoring

the evolution of the learning process, taking into account the available resources, and the ability to reason and learn about it [Gama, 2012, 2010].

Combining offline and online models. Online (or real-time) and offline (or batch) learning are mostly considered as mutually exclusive, but it is their combination that might enhance the value of data the most. Lambda Architecture [Marz and Warren, 2015] is a useful framework for designing big data applications where we can combine these two models in a same plateform. Figure 9.2 is a diagram of the Lambda Architecture.

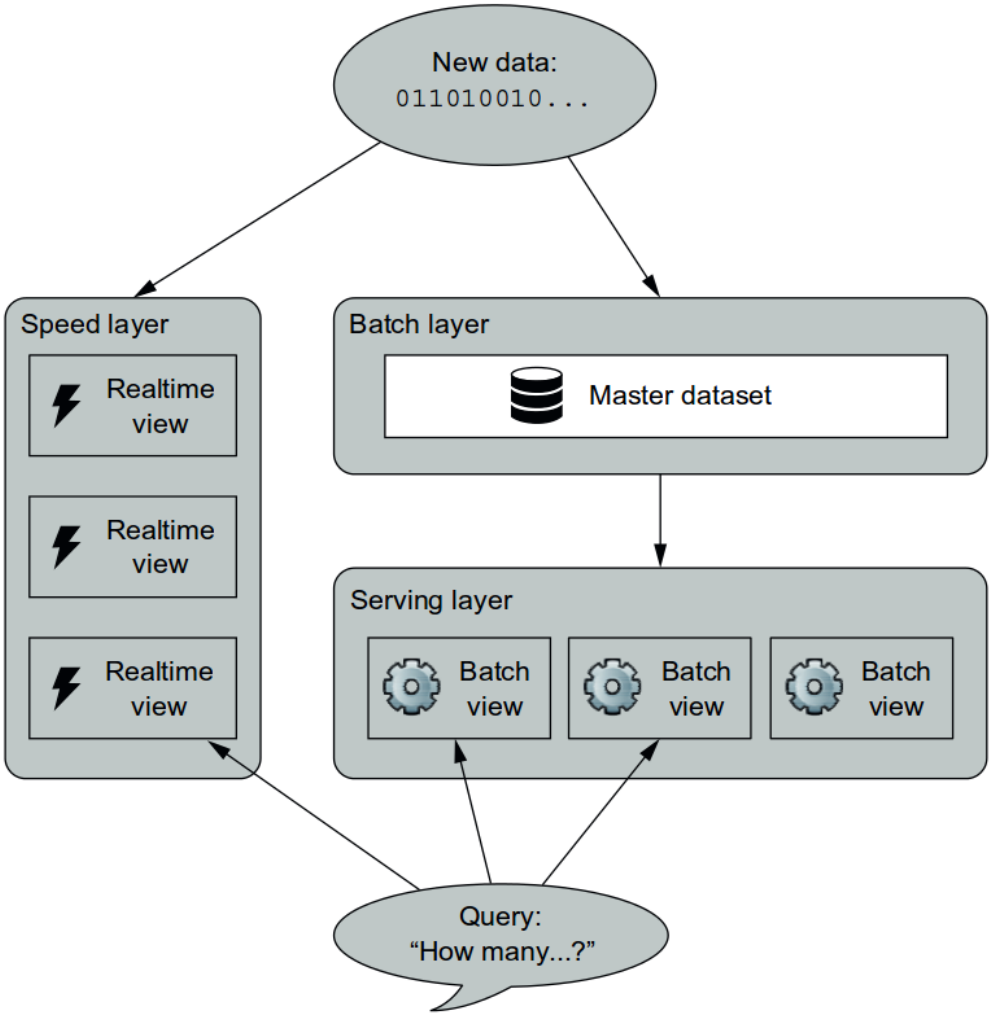


FIGURE 9.2: Lambda Architecture diagram [Marz and Warren, 2015]

Essentially, the Lambda Architecture comprises the following components, processes, and responsibilities:

- New Data: All data entering the system is dispatched to both the batch layer and the speed layer for processing.
- Batch layer: This layer has two functions: (i) managing the master dataset, an immutable, append-only set of raw data, and (ii) to pre-compute arbitrary query functions from scratch, called batch views.
- Serving layer: This layer indexes the batch views so that they can be queried ad hoc with low latency.
- Speed layer: This layer compensates for the high latency of updates to the serving layer, due to the batch layer. Using fast and incremental algorithms, the speed layer deals with recent data only.
- Queries: Any incoming query can be answered by merging results from both batch views and real-time views.

Designing data stream clustering methods in a Lambda Architecture where we can benefit from the high accuracy of the batch model will be interesting and challenging.

Appendix A

Quality criteria

The algorithms are evaluated using three performance measures: accuracy (purity), Normalized Mutual Information (NMI) and Rand index [Strehl and Ghosh, 2002]. The value of each measure lies between 0 and 1. A higher value indicates better clustering results. The accuracy (purity) averages the fraction of items belonging to the majority class of in each cluster.

$$Acc = \frac{\sum_{i=1}^{K} \frac{|N_i^d|}{|N_i|}}{K} \times 100\%, \tag{A.1}$$

where K denotes the number of clusters, N_i^d denotes the number of points with the dominant class label in cluster i, and N_i denotes the number of points in cluster i. Intuitively, the accuracy (purity) measures the purity of the clusters with respect to the true cluster (class) labels that are known for our datasets [Cao et al., 2006].

Normalized mutual information provides a measure that is independent of the number of clusters as compared to purity. It reaches its maximum value of 1 only when the two sets of labels have a perfect one-to-one correspondence [Strehl and Ghosh, 2002]. Given the true clustering $A = \{A_1, ..., A_k\}$ and the grouping $B = \{B_1, ..., B_h\}$ obtained by a clustering method, let C be the confusion matrix whose element C_{ij} is the number of records of cluster i of A that are also in the cluster j of B. The normalized mutual information $NMI(A, B)$ is defined as [Forestiero et al., 2013]:

$$NMI(A,B) = \frac{-2\sum_{i=1}^{C_A}\sum_{j=1}^{C_B} C_{ij} log(C_{ij}N/C_{i.}C_{.j})}{\sum_{i=1}^{C_A} C_{i.} log(C_{i.}/N) + \sum_{j=1}^{C_B} C_{.j} log(C_{.j}/N)}, \tag{A.2}$$

where C_A (resp. C_B) is the number of groups in the partition A (resp. B), $C_{i.}$ (resp. $C_{.j}$) is the sum of elements of C in row i (resp. column j), and N is the number of points. If $A = B$, $NMI(A, B) = 1$. If A and B are completely different, $NMI(A, B) = 0$.

The Rand index measures how accurately a classifier can classify data elements by comparing cluster labels with the underlying class labels. Given N data points, there are a total of $\binom{N}{2}$ distinct pairs of data points which can be categorized into four categories: (a) pairs having the same cluster label and the same class label (their number denoted as N^{11}); (b) pairs having different cluster labels and different class labels (their number denoted as N^{00}); (c) pairs having the same cluster label but different class labels (their number denoted as N^{10}); (d) pairs having different cluster labels but the same class label (their number denoted as N^{01}). The Rand index is defined as [Rand, 1971]:

$$Rand = (N^{11} + N^{00}) / \binom{N}{2}. \tag{A.3}$$

Bibliography

Yuri Demchenko, Paola Grosso, Cees De Laat, and Peter Membrey. Addressing big data issues in scientific data infrastructure. In *Collaboration Technologies and Systems (CTS), 2013 International Conference on*, pages 48–55. IEEE, 2013.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28, 2012a.

Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*, pages 226–231, 1996.

Charu C. Aggarwal and Chandan K. Reddy. *Data Clustering: Algorithms and Applications*. CRC Press, 2014.

Komkrit Udommanetanakit, Thanawin Rakthanmanon, and Kitsana Waiyamai. E-stream: Evolution-based technique for stream clustering. In *ADMA*, pages 605–615, 2007.

Xiangliang Zhang, Cyril Furtlehner, and Michèle Sebag. Data streaming with affinity propagation. In *ECML/PKDD (2)*, pages 628–643, 2008.

Nathan Marz and James Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.

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

Xiangliang Zhang. *Contributions to Large Scale Data Clustering and Streaming with Affinity Propagation. Application to Autonomic Grids.* PhD thesis, Université Paris-Sud, France, 2010.

John R. Mashey. Big data and the next wave of infrastress problems, solutions, opportunities. 1998.

Wei Fan and Albert Bifet. Mining big data: current status, and forecast to the future. *ACM sIGKDD Explorations Newsletter*, 14(2):1–5, 2013.

Douglas Laney. 3D data management: Controlling data volume, velocity, and variety. Technical report, META Group, February 2001.

John Gantz and David Reinsel. Extracting value from chaos. *IDC iview*, 1142: 1–12, 2011.

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006.

Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007):21, 2007.

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

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

Xiufeng Liu, Nadeem Iftikhar, and Xike Xie. Survey of real-time processing systems for big data. In *18th International Database Engineering & Applications Symposium, IDEAS 2014, Porto, Portugal, July 7-9, 2014*, pages 356–361, 2014.

Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics*

in Cloud Ccomputing, HotCloud'12, pages 10–10, Berkeley, CA, USA, 2012b. USENIX Association.

Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 423–438, 2013.

Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.*, 33(1), 2008.

Jeong-Hyon Hwang, Magdalena Balazinska, Alex Rasin, Ugur Çetintemel, Michael Stonebraker, and Stanley B. Zdonik. High-availability algorithms for distributed stream processing. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, pages 779–790, 2005.

Mohammed Ghesmoune, Mustapha Lebbah, and Hanene Azzag. Micro-batching growing neural gas for clustering data streams using spark streaming. In *INNS Conference on Big Data 2015, San Francisco, CA, USA, 8-10 August 2015*, pages 158–166, 2015a.

Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *CoRR*, abs/1505.06807, 2015.

Charu C. Aggarwal, T. J. Watson, Resch Ctr, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *In VLDB*, pages 81–92, 2003.

Marcel R. Ackermann, Marcus Märtens, Christoph Raupach, Kamil Swierkot, Christiane Lammersen, and Christian Sohler. StreamKM++: A clustering algorithm for data streams. *ACM Journal of Experimental Algorithmics*, 17(1), 2012.

Sebastian Schelter, Stephan Ewen, Kostas Tzoumas, and Volker Markl. "all roads lead to rome": optimistic recovery for distributed iterative data processing. In

22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013, pages 1919–1928, 2013.

Albert Bifet, Geoff Holmes, Bernhard Pfahringer, Philipp Kranen, Hardy Kremer, Timm Jansen, and Thomas Seidl. MOA: massive online analysis, a framework for stream classification and clustering. In *Proceedings of the First Workshop on Applications of Pattern Analysis, WAPA 2010, Cumberland Lodge, Windsor, UK, September 1-3, 2010*, pages 44–50, 2010.

Philipp Kranen, Ira Assent, Corinna Baldauf, and Thomas Seidl. The ClusTree: indexing micro-clusters for anytime stream mining. *Knowledge and information systems*, 29(2):249–272, 2011.

Feng Cao, Martin Ester, Weining Qian, and Aoying Zhou. Density-based clustering over an evolving data stream with noise. In *SDM*, pages 328–339, 2006.

Yixin Chen and Li Tu. Density-based clustering for real-time stream data. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Jose, California, USA, August 12-15, 2007*, pages 133–142, 2007.

Gianmarco De Francisci Morales and Albert Bifet. SAMOA: scalable advanced massive online analysis. *Journal of Machine Learning Research*, 16:149–153, 2015.

Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988. ISBN 0-13-022278-X.

David Arthur and Sergei Vassilvitskii. k-means++: the advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 1027–1035, 2007.

Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. Scalable k-means++. *Proceedings of the VLDB Endowment*, 5(7): 622–633, 2012.

Samuel Kaski, Jari Kangas, and Teuv Kohonen. Bibliography of self-organizing map (som) papers: 1981-1997. *Neural computing surveys*, 1:102–350, 1998.

Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998. ISBN 0132733501.

T. Kohonen, M. R. Schroeder, and T. S. Huang, editors. *Self-Organizing Maps*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 3rd edition, 2001. ISBN 3540679219.

T. Martinetz and K. Schulten. A "Neural-Gas" Network Learns Topologies. *Artificial Neural Networks*, I:397–402, 1991.

Bernd Fritzke. Unsupervised clustering with growing cell structures. In *In Proceedings of the International Joint Conference on Neural Networks*, pages 531–536. IEEE, 1991.

Bernd Fritzke. A growing neural gas network learns topologies. In *NIPS*, pages 625–632, 1994.

Oliver Beyer and Philipp Cimiano. Online semi-supervised growing neural gas. *Int. J. Neural Syst.*, 22(5), 2012.

Brendan J Frey and Delbert Dueck. Clustering by passing messages between data points. *science*, 315(5814):972–976, 2007.

Amineh Amini, Ying Wah Teh, and Hadi Saboohi. On density-based data streams clustering algorithms: A survey. *J. Comput. Sci. Technol.*, 29(1):116–141, 2014.

Pavel Berkhin. A survey of clustering data mining techniques. In *Grouping multidimensional data*, pages 25–71. Springer, 2006.

Chris Fraley and Adrian E Raftery. How many clusters? which clustering method? answers via model-based cluster analysis. *The computer journal*, 41(8):578–588, 1998.

Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)*, pages 1–38, 1977.

Geoffrey McLachlan and Thriyambakam Krishnan. *The EM algorithm and extensions*, volume 382. John Wiley & Sons, 2007.

Ali El Attar, Antoine Pigeau, and Marc Gelgon. Robust estimation of a global gaussian mixture by decentralized aggregations of local models. *Web Intelligence*

and Agent Systems, 11(3):245–262, 2013. doi: 10.3233/WIA-130273. URL http://dx.doi.org/10.3233/WIA-130273.

Ali El Attar. *Estimation robuste des modèles de mélange sur des données distribuées*. Theses, Université de Nantes, July 2012. URL https://tel.archives-ouvertes.fr/tel-00746118.

Ralf Lämmel. Google's mapreduce programming model—revisited. *Science of computer programming*, 70(1):1–30, 2008.

Eshref Januzaj, Hans-Peter Kriegel, and Martin Pfeifle. Dbdc: Density based distributed clustering. In *Advances in Database Technology-EDBT 2004*, pages 88–105. Springer, 2004.

Tugdual Sarazin, Hanane Azzag, and Mustapha Lebbah. SOM clustering using spark-mapreduce. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, May 19-23, 2014*, pages 1727–1734, 2014.

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

Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

Yaobin He, Haoyu Tan, Wuman Luo, Shengzhong Feng, and Jianping Fan. Mr-dbscan: a scalable mapreduce-based dbscan algorithm for heavily skewed data. *Frontiers of Computer Science*, 8(1):83–99, 2014.

Ali Seyed Shirkhorshidi, Saeed Aghabozorgi, Teh Ying Wah, and Tutut Herawan. Big data clustering: a review. In *Computational Science and Its Applications–ICCSA 2014*, pages 707–720. Springer, 2014.

Abhinandan S Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th international conference on World Wide Web*, pages 271–280. ACM, 2007.

Henggang Cui, Jinliang Wei, and Wei Dai. Parallel implementation of expectation-maximization for fast convergence.

Aniruddha Basak, Irina Brinster, and Ole J Mengshoel. Mapreduce for bayesian network parameter learning using the em algorithm. *Proc. of Big Learning: Algorithms, Systems and Tools*, 2012.

Alina Ene, Sungjin Im, and Benjamin Moseley. Fast clustering using mapreduce. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 681–689. ACM, 2011.

Charu C. Aggarwal. A survey of stream clustering algorithms. In *Data Clustering: Algorithms and Applications*, pages 231–258. 2013.

Hai-Long Nguyen, Yew-Kwong Woon, and Wee Keong Ng. A survey on data stream clustering and classification. *Knowl. Inf. Syst.*, 45(3):535–569, 2015.

Madjid Khalilian and Norwati Mustapha. Data stream clustering: Challenges and issues. *CoRR*, abs/1006.5261, 2010.

Yogita and D. Toshniwal. Clustering techniques for streaming data-a survey. In *Advance Computing Conference (IACC), 2013 IEEE 3rd International*, pages 951–956, 2013.

Maryam Mousavi, Azuraliza Abu Bakar, and Mohammadmahdi Vakilian. Data stream clustering algorithms: A review. *International Journal of Advances in Soft Computing & Its Applications*, 7(3), 2015.

Jonathan de Andrade Silva, Elaine R. Faria, Rodrigo C. Barros, Eduardo R. Hruschka, André Carlos Ponce Leon Ferreira de Carvalho, and João Gama. Data stream clustering: A survey. *ACM Comput. Surv.*, 46(1):13, 2013.

Lukasz Golab and M Tamer Özsu. Issues in data stream management. *ACM Sigmod Record*, 32(2):5–14, 2003.

Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining data streams: a review. *ACM Sigmod Record*, 34(2):18–26, 2005.

Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 358–369. VLDB Endowment, 2002.

Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Duplicate detection in click streams. In *Proceedings of the 14th international conference on World Wide Web*, pages 12–21. ACM, 2005.

Charu C Aggarwal. *Data streams: models and algorithms*, volume 31. Springer Science & Business Media, 2007.

Charu C Aggarwal. A framework for diagnosing changes in evolving data streams. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 575–586. ACM, 2003.

Daniel Kifer, Shai Ben-David, and Johannes Gehrke. Detecting change in data streams. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 180–191. VLDB Endowment, 2004.

Donko Donjerkovic, Yannis E Ioannidis, and Raghu Ramakrishnan. Dynamic histograms: Capturing evolving data sets. In *Proceedings of the international conference on data engineering*, pages 86–86. IEEE Computer Society Press; 1998, 2000.

Venkatesh Ganti, Johannes Gehrke, and Raghu Ramakrishnan. Mining data streams under block evolution. *ACM SIGKDD Explorations Newsletter*, 3(2): 1–10, 2002.

Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: An efficient data clustering method for very large databases. In *SIGMOD Conference*, pages 103–114, 1996.

Wicha Meesuksabai, Thanapat Kangkachit, and Kitsana Waiyamai. Hue-stream: Evolution-based clustering technique for heterogeneous data streams with uncertainty. In *Advanced Data Mining and Applications - 7th International Conference, ADMA 2011, Beijing, China, December 17-19, 2011, Proceedings, Part II*, pages 27–40, 2011.

Charu C. Aggarwal and Philip S. Yu. A framework for clustering uncertain data streams. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 150–159, 2008.

Chunyu Yang and Jie Zhou. Hclustream: A novel approach for clustering evolving heterogeneous data stream. In *Workshops Proceedings of the 6th IEEE International Conference on Data Mining (ICDM 2006), 18-22 December 2006, Hong Kong, China*, pages 682–688, 2006.

Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57, 1984.

Jan Peter Patist, Wojtek Kowalczyk, and Elena Marchiori. Maintaining gaussian mixture models of data streams under block evolution. In *International Conference on Computational Science*, pages 1071–1074. Springer, 2006.

Allou Samé and Hani El Assaad. A state-space approach to modeling functional time series application to rail supervision. In *22nd European Signal Processing Conference, EUSIPCO 2014, Lisbon, Portugal, September 1-5, 2014*, pages 1402–1406, 2014.

Hani EL ASSAAD. *Dynamic classification and modeling of non-stationary temporal data*. Theses, Université Paris-Est, December 2014. URL https://tel.archives-ouvertes.fr/tel-01143904.

Charlie Isaksson, Margaret H. Dunham, and Michael Hahsler. SOStream: Self organizing density-based clustering over data stream. In *MLDM*, pages 264–278, 2012.

Chang-Dong Wang, Jian-Huang Lai, Dong Huang, and Wei-Shi Zheng. SVStream: A support vector-based algorithm for clustering data streams. *IEEE Trans. Knowl. Data Eng.*, 25(6):1410–1424, 2013.

Asa Ben-Hur, David Horn, Hava T. Siegelmann, and Vladimir Vapnik. Support vector clustering. *Journal of Machine Learning Research*, 2:125–137, 2001.

David M. J. Tax and Robert P. W. Duin. Support vector domain description. *Pattern Recognition Letters*, 20(11-13):1191–1199, 1999.

Isaac J. Sledge and James M. Keller. Growing neural gas for temporal clustering. In *19th International Conference on Pattern Recognition (ICPR 2008), December 8-11, 2008, Tampa, Florida, USA*, pages 1–4, 2008.

Carlos Augusto Teixeira Mendes, Marcelo Gattass, and Hélio Lopes. Fgng: A fast multi-dimensional growing neural gas implementation. *Neurocomputing*, 128: 328–340, 2014.

SV Mitsyn and GA Ososkov. The growing neural gas and clustering of large amounts of data. *Optical Memory and Neural Networks*, 20(4):260–270, 2011.

Bernd Fritzke. A self-organizing network that can follow non-stationary distributions. In *Artificial Neural Networks - ICANN '97, 7th International Conference, Lausanne, Switzerland, October 8-10, 1997, Proceedings*, pages 613–618, 1997.

Stephen Marsland, Jonathan Shapiro, and Ulrich Nehmzow. A self-organising network that grows when required. *Neural Networks*, 15(8-9):1041–1058, 2002.

Stephen Marsland, Ulrich Nehmzow, and Jonathan Shapiro. On-line novelty detection for autonomous mobile robots. *Robotics and Autonomous Systems*, 51 (2):191–206, 2005.

Yann Prudent and Abdellatif Ennaji. An incremental growing neural gas learns topologies. In *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, volume 2, pages 1211–1216. IEEE, 2005.

Hatem Hamza, Yolande Belaïd, Abdel Belaïd, and Bidyut B Chaudhuri. Incremental classification of invoice documents. In *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, pages 1–4. IEEE, 2008.

Jean-Charles Lamirel, Zied Boulila, Maha Ghribi, and Pascal Cuxac. A new incremental growing neural gas algorithm based on clusters labeling maximization: application to clustering of heterogeneous textual data. In *Trends in Applied Intelligent Systems*, pages 139–148. Springer, 2010.

José GarcíA-RodríGuez, Anastassia Angelopoulou, Juan Manuel García-Chamizo, Alexandra Psarrou, Sergio Orts Escolano, and Vicente Morell GiméNez. Autonomous growing neural gas for applications with time constraint: optimal parameter estimation. *Neural Networks*, 32:196–208, 2012.

Marco AF Pimentel, David A Clifton, Lei Clifton, and Lionel Tarassenko. A review of novelty detection. *Signal Processing*, 99:215–249, 2014.

Mohamed-Rafik Bouguelia, Yolande Belaïd, and Abdel Belaïd. An adaptive incremental clustering method based on the growing neural gas algorithm. In *ICPRAM*, pages 42–49, 2013.

Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams: Theory and practice. *Knowledge and Data Engineering, IEEE Transactions on*, 15(3):515–528, 2003.

Xingquan (Hill) Zhu. Stream data mining repository (web site), 2010. URL http://www.cse.fau.edu/~xqzhu/stream.html.

K. Bache and M. Lichman. UCI machine learning repository, 2013.

Alexander Strehl and Joydeep Ghosh. Cluster ensembles — a knowledge reuse framework for combining multiple partitions. *Journal of Machine Learning Research*, 3:583–617, 2002.

Matthew Bolanos, John Forrest, and Michael Hahsler. *stream: Infrastructure for Data Stream Mining*, 2014. URL http://CRAN.R-project.org/package=stream. R package version 0.2-0.

Aoying Zhou, Feng Cao, Weining Qian, and Cheqing Jin. Tracking clusters in evolving data streams over sliding windows. *Knowl. Inf. Syst.*, 15(2):181–214, 2008.

Mohammed Ghesmoune, Hanene Azzag, and Mustapha Lebbah. G-stream: Growing neural gas over data stream. In *Neural Information Processing - 21st International Conference, ICONIP 2014, Kuching, Malaysia, November 3-6, 2014. Proceedings, Part I*, pages 207–214, 2014.

Mohammed Ghesmoune, Mustapha Lebbah, and Hanene Azzag. Clustering over data streams based on growing neural gas. In *Advances in Knowledge Discovery and Data Mining - 19th Pacific-Asia Conference, PAKDD 2015, Ho Chi Minh City, Vietnam, May 19-22, 2015, Proceedings, Part II*, pages 134–145, 2015b.

Salima Benbernou, Xin Huang, and Mourad Ouziri. Fusion of big RDF data: A semantic entity resolution and query rewriting-based inference approach. In *Web Information Systems Engineering - WISE 2015 - 16th International Conference, Miami, FL, USA, November 1-3, 2015, Proceedings, Part II*, pages 300–307, 2015.

T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York, second edition, 2009.

Terry Therneau, Beth Atkinson, and Brian Ripley. *rpart: Recursive Partitioning and Regression Trees*, 2015. URL https://CRAN.R-project.org/package=rpart. R package version 4.1-10.

Hanene Azzag, Gilles Venturini, Antoine Oliver, and Christiane Guinot. A hierarchical ant based clustering algorithm and its use in three real-world applications. *European Journal of Operational Research*, 179(3):906–922, 2007.

Nhat-Quang Doan, Hanane Azzag, and Mustapha Lebbah. Growing self-organizing trees for knowledge discovery from data. In *The 2012 International Joint Conference on Neural Networks (IJCNN), Brisbane, Australia, June 10-15, 2012*, pages 1–8, 2012.

Nhat-Quang Doan, Hanane Azzag, Mustapha Lebbah, and Guillaume Santini. Self-organizing trees for visualizing protein dataset. In *The 2013 International Joint Conference on Neural Networks, IJCNN 2013, Dallas, TX, USA, August 4-9, 2013*, pages 1–8, 2013.

D. Auber. Tulip : A huge graph visualisation framework. In P. Mutzel and M. Jünger, editors, *Graph Drawing Softwares*, Mathematics and Visualization, pages 105–126. Springer-Verlag, 2003.

Carlos Ordonez. Clustering binary data streams with k-means. In *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pages 12–19. ACM, 2003.

Brian Babcock, Mayur Datar, Rajeev Motwani, and Liadan O'Callaghan. Maintaining variance and k-medians over data stream windows. In *Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 9-12, 2003, San Diego, CA, USA*, pages 234–243, 2003.

Moses Charikar, Liadan O'Callaghan, and Rina Panigrahy. Better streaming algorithms for clustering problems. In *Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing*, pages 30–39, 2003.

Mustapha Lebbah. Carte topologique pour données qualitatives: application à la reconnaissance automatique de la densité du trafic routier. Master's thesis, Université de Versailles-Saint Quentin en Yvelines, 2003 2003. URL http://tel.archives-ouvertes.fr/tel-00161698.

Gérard Govaert. *Data Analysis*. ISTE-Wiley, 2009. URL https://hal.archives-ouvertes.fr/hal-00447855.

Georg Krempl, Indre Žliobaite, Dariusz Brzeziński, Eyke Hüllermeier, Mark Last, Vincent Lemaire, Tino Noack, Ammar Shaker, Sonja Sievi, Myra Spiliopoulou, et al. Open challenges for data stream mining research. *ACM SIGKDD explorations newsletter*, 16(1):1–10, 2014.

João Gama. A survey on learning from data streams: current and future trends. *Progress in AI*, 1(1):45–55, 2012.

João Gama. *Knowledge Discovery from Data Streams*. Chapman and Hall / CRC Data Mining and Knowledge Discovery Series. CRC Press, 2010.

Graham Cormode, Minos N. Garofalakis, Peter J. Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012.

Agostino Forestiero, Clara Pizzuti, and Giandomenico Spezzano. A single pass algorithm for clustering evolving data streams based on swarm intelligence. *Data Min. Knowl. Discov.*, 26(1):1–26, 2013.

W.M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(336):846–850, 1971.