PAULO GUSTAVO LOPES CANDIDO

AGRUPAMENTO ESCALÁVEL DE FLUXOS CONTÍNUOS DE DADOS COM ESTIMATIVA DO NÚMERO DE GRUPOS

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS – BRASIL
2018

Ficha catalográfica preparada pela Biblioteca Central da Universidade Federal de Viçosa - Câmpus Viçosa

T
C217a
2018

Candido, Paulo Gustavo Lopes, 1989-
Agrupamento escalável de fluxos contínuos de dados com estimativa do número de grupos / Paulo Gustavo Lopes Candido. – Viçosa, MG, 2018.
xiii, 39 f. : il. (algumas color.) ; 29 cm.

Orientador: Murilo Coelho Naldi.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f. 36-39.

1. Aprendizado do computador. 2. Análise por agrupamento. 3. Fluxo de dados (Computadores). 4. Big data. 5. Computação evolutiva. 6. Mineração de dados (Computação). I. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. II. Título.

CDD 22. ed. 006.31

PAULO GUSTAVO LOPES CANDIDO

AGRUPAMENTO ESCALÁVEL DE FLUXOS CONTÍNUOS DE DADOS COM ESTIMATIVA DO NÚMERO DE GRUPOS

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

APROVADA: 21 de dezembro de 2018.

Giovanni Ventorim Comarela

Ricardo Cerri

Murilo Coelho Naldi
(Orientador)

A Deus e minha família.

"Sábio é aquele que conhece os limites da própria ignorância."
(Sócrates)

Agradecimentos

Agradeço primeiramente a Deus por me acompanhar a cada passo.

Agradeço à minha esposa, que me apoiou incessantemente, principalmente nos momentos mais difíceis, por todo o amor, paciência e carinho durante essa jornada e principalmente, por acreditar no meu sonho.

Aos meus pais pelo amor e incentivo a cada etapa da minha vida, por serem o meu porto seguro enquanto navego por águas turbulentas.

Aos meus amigos César Gennaro, Betinho Côrrea, Bruna Martins e Rubens Oliveira pela fraternidade e carinho durante esses três anos de caminhada.

Ao meu orientador Murilo Naldi, pelos ricos ensinamentos, pela ajuda e solicitude em tempo integral. Por acreditar na nossa pesquisa, mesmo nos momentos de dificuldade.

Aos meus coorientadores Elaine Faria e Jonathan Silva por compartilharem e enriquecerem esse trabalho com seus conhecimentos.

Meu respeito e admiração a todos os professores e colegas que compartilharam junto comigo as alegrias e percalços na busca desse objetivo em comum.

Agradeço ao financiamento fornecido pela Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), ao qual foi essencial para que pudesse me dedicar à essa pesquisa.

Agradeço a Universidade Federal de Viçosa e ao Departamento de Informática pela oportunidade de galgar mais um grau na escada do conhecimento.

E finalmente, a todos aqueles que, direta ou indiretamente, contribuíram para a execução deste trabalho, os meus sinceros agradecimentos.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ARI	*Adjusted Rand Index*
BkM	*Bisecting kMeans*
D-Stream	*Discretized Stream*
F-EAC	*Fast Evolutionary Algorithm for Clustering*
ISESC	*Incremental Scalable Evolutionary Stream Clustering*
ISESC-CD	*Incremental Scalable Evolutionary Stream Clustering with Change Detection*
ISS	*Incremental Simplified Silhouette*
k	Número de grupos
KC99	*KDDCup99 (10% version) data set*
PPA	*PaperAbstracts data set*
RDD	*Resilient Distributed Dataset*
SESC	*Scalable Evolutionary Stream Clustering*
SS	*Simplified Silhouette Index*
YTG	*YouTube Games data set*

Resumo

CÂNDIDO, Paulo Gustavo Lopes, M.Sc., Universidade Federal de Viçosa, dezembro de 2018. **Agrupamento Escalável de Fluxos Contínuos de Dados com Estimativa do Número de Grupos**. Orientador: Murilo Coelho Naldi. Coorientadores: Elaine Ribeiro de Faria Paiva e Jonathan de Andrade Silva.

Avanços da tecnologia têm mudado a forma como dados são coletados, armazenados e analisados. Novas abordagens têm sido utilizadas para agrupamento não supervisionado de dados, tais como agrupamento de dados gerados em tempo real (fluxos contínuos de dados), e agrupamento escalável de dados. Ambas as abordagens descartam o armazenamento do conjunto completo de dados em memória principal devido a restrições físicas, utilizando técnicas como leitura linear e computação distribuída, respectivamente, para lidar com grande volume de dados. O agrupamento de fluxos contínuos de dados precisa lidar com características específicas desse formato: são virtualmente e potencialmente ilimitados e possuem uma distribuição não-estacionária. Apesar de pouco exploradas, técnicas para estimação dinâmica do número de grupos mostraram ser eficazes na manutenção dos modelos de agrupamento, uma vez que grupos podem surgir e desaparecer ao longo do tempo. Considerando ainda um cenário de aumento exponencial na quantidade de dados gerados em tempo real, surge a necessidade de algoritmos escaláveis (capazes de distribuir o processamento) para agrupamento de fluxos de dados, capazes de estimar o número de grupos, a fim de manter um alto nível de qualidade. Neste trabalho são apresentados cinco algoritmos com essa finalidade, dos quais quatro são baseados na computação evolutiva. O modelo funcional *MapReduce* é utilizado para prover escalabilidade por meio de um sistema distribuído, garantindo confiabilidade,

resiliência e tolerância a falhas. Os algoritmos foram experimentados, analisados e comparados estatisticamente a fim de verificar sua qualidade e desempenho. Os resultados mostram que os algoritmos propostos são capazes de obter modelos de alta qualidade para fluxos de dados de alta velocidade que precisem ser escalados, mesmo com variações de distribuição e número de grupos.

Abstract

CÂNDIDO, Paulo Gustavo Lopes, M.Sc., Universidade Federal de Viçosa, December, 2018. **Scalable Data Stream Clustering with Estimation of the Number of Clusters**. Advisor: Murilo Coelho Naldi. Co-advisors: Elaine Ribeiro de Faria Paiva and Jonathan de Andrade Silva.

Advances in technology have been changing the way data is collected, stored and analyzed. New approaches have been used for unsupervised data clustering, such as real-time data clustering (data stream), and scalable data clustering. Both approaches discard the storage of the whole dataset in main memory due to physical constraints, using techniques such as linear reading and distributed computing, respectively, to handle large volumes of data. Data stream clustering algorithms must deal with specific characteristics of this format: they are virtually and potentially unbounded and have a non-stationary distribution. Although little explored, techniques for dynamic estimation of the number of clusters have been shown to be effective in maintaining clustering models, since clusters may arise and disappear over time. Considering also a scenario with an exponential increase of the real-time generated data, there is a need for scalable algorithms to cluster data streams, able to estimate the number of clusters, in order to maintain a high level of quality. In this work, five algorithms are presented for this purpose, of which four are based on evolutionary computation. The MapReduce functional model is used to provide scalability across a distributed system, ensuring reliability, resilience, and fault-tolerance. The algorithms were experimented, analyzed and compared statistically in order to verify their quality and performance. The results show that the proposed algorithms are able to obtain high-quality models for high-speed data streams that

need to be scaled, even with variations of distribution and number of clusters.

Capítulo 1

INTRODUÇÃO

O avanço da tecnologia aliado à sua massificação tem contribuído para o aumento exponencial da quantidade de dados gerados diariamente pelo mundo. O Facebook em 2014 hospedava em média cerca de 233 terabytes de fotos por dia [Tom White, 2015]. Em 2015, satélites, observatórios, radares e outros sensores meteorológicos somavam uma coleta de 20 terabytes por dia enquanto hospitais e clinicas médicas geravam um volume de cerca de 665 terabytes de dados médicos diariamente [The Software Alliance, 2015]. Além do alto volume de dados gerados, observa-se uma taxa exponencial de crescimento. Uma estimativa de 2015 [The Software Alliance, 2015] mostra que em apenas dois anos (2014-2015) foram gerados 90% de todos os dados gerados até então. Esse grande volume de dados e suas projeções de crescimento têm mudado a forma como os dados são analisados.

Há algumas décadas, o agrupamento de dados, assim como a mineração de dados de modo geral, era focada em análise de pequenos conjuntos de dados estáticos [Gama, 2010], contudo, os avanços na tecnologia e quantidades cada vez maiores de dados a serem analisados, exigiram algoritmos com novas habilidades, como agrupamento de conjuntos escaláveis de dados e agrupamento de fluxos contínuos de dados.

Fluxos contínuos de dados são sequências de objetos virtualmente ilimitadas com uma distribuição não estacionária [Silva et al., 2013]. O processo de aprendizagem desse formato de dados envolve lidar com algumas características inerentes a ele:

- Fluxos de dados são potencialmente e virtualmente ilimitados e os objetos são recebidos ao longo do tempo de forma contínua [Aggarwal, 2015], o que implica que em momento algum, o conjunto completo de dados estará disponível para análise, tampouco é possível armazená-lo por completo em memória principal, na qual acessos aleatórios aos dados são aceitáveis.

- Fluxos de dados evoluem com o decorrer do tempo e por não possuírem uma distribuição estacionária, estão sujeitos a uma localidade temporal [Aggarwal, 2007], ou seja, os dados são distribuídos não só espacialmente, mas também temporalmente [Gama, 2010]. Esse comportamento, comumente chamado de

mudança de conceito ou *concept-drift*, implica que o que foi aprendido até determinado momento pode mudar com o tempo.

Uma abordagem vastamente utilizada [Guha et al., 2000; Silva et al., 2013; Aggarwal, 2015] para lidar com o tamanho ilimitado de um fluxo de dados é reduzir ao máximo o número de leituras em cada objeto. Normalmente utiliza-se uma única leitura para então liberar o objeto da memória principal. Muitos algoritmos [Zhang et al., 1997; Aggarwal et al., 2003; Zhou et al., 2008; Ackermann et al., 2012] utilizam essa leitura única (*single-scan*) para criar e atualizar sumários estatísticos, de forma incremental, que possam representar esses objetos no futuro. A sumarização normalmente ocorre à medida que os objetos são recebidos (componente *online*), e quando requisitado, os dados sumarizados são agrupados por um algoritmo de agrupamento (componente *offline*). Essa abordagem permite que uma quantidade indeterminada de objetos seja representada por um único sumário estatístico, ocupando poucos bytes de memória.

Para lidar com a mudança de conceito, o modelo de janelas temporais é tipicamente utilizado [Zhu & Shasha, 2002]. Essa abordagem garante que com o passar do tempo, os objetos antigos percam relevância na análise, enquanto os mais novos, que representam as tendências do fluxo de dados, se destacam. Os modelos mais comuns de janela são:

- Janelas deslisantes (*slide windows*): apenas as informações mais recentes do fluxo são armazenadas, seguindo o princípio de filas (*first in, first out*) de tamanho fixo ou variável. Quando um novo objeto é recebido, o menos relevante (mais antigo) é descartado.

- Janelas amortizadas (*dumped windows*): neste modelo, os objetos do fluxo têm diferentes pesos. Quando são recebidos, o peso máximo é atribuído, e a medida que novos objetos são recebidos, os objetos mais antigos têm seu peso diminuído de acordo com uma função de decaimento.

- Janelas por ponto de referência (*landmark windows*): o ponto de referência deste modelo define a partir de quando os dados são relevantes. Apenas os dados recebidos após este ponto são armazenados. Pontos de referência podem ser definidos de acordo com o próprio conjunto de dados, ou fatores externos, como por exemplo, fatores temporais (dias, semanas, etc).

Por outro lado, a análise de conjuntos escaláveis de dados, por meio da computação distribuída, tem ganhado foco no campo da mineração de dados sob termos como *big data* e *data science*. Assim como aplicações em fluxos de dados, aplicações escaláveis também lidam com grandes conjuntos de dados que não podem ser indexados na memória principal de uma única máquina [Aggarwal, 2015], mas diferentemente, normalmente consideram um conjunto estático de dados, previamente armazenados de forma distribuída em um conjunto de máquinas [Dean & Ghemawat, 2008].

O modelo de programação funcional *MapReduce* [Dean & Ghemawat, 2008] foi introduzido para auxiliar na manipulação de dados escaláveis. Se trata de um

modelo para uniformizar o acesso aos dados distribuídos na forma de duas operações principais:

- O método ***map*** é uma operação de transformação distribuída, aplicada a cada objeto do dataset pelo nó em que está armazenado, produzindo pares de chave-valor.
- O método ***reduce*** é utilizada para se agregar, de forma distribuída, todos os valores que possuem a mesma chave, obtendo assim informações relevantes.

Muitas aplicações do mundo real podem ser expressas pelo modelo *MapReduce* [Dean & Ghemawat, 2008] e contam com uma série de garantias oferecidas pelos *frameworks* que o implementam, como particionamento dos dados, agendamento/execução de tarefas, recuperação a falhas e outras comunicações necessárias para sincronização dos nós do sistema distribuído.

Durante muitos anos, abordagens para análise de fluxos de dados e análise escalável de dados estáticos evoluíram de forma isolada, mas o crescente volume/velocidade de dados gerados em tempo real tem contribuído para o problema central abordado nessa pesquisa: analisar fluxos de dados de alta velocidade em que o processamento centralizado seja inviável. A união dessas duas abordagens é um potencial solução para este problema, como mostrado na análise dos experimental apresentada neste trabalho.

1.1 O problema e sua importância

O agrupamento de dados é uma das principais técnicas de aprendizagem exploratória. Sua aplicação em fluxos contínuos de dados possibilitou que conjuntos reais de dados, que são naturalmente distribuídos temporalmente fossem analisados, como por exemplo [Silva et al., 2013]: rede de sensores, dados meteorológicos, mercado de ações, monitoramento de tráfego de rede, etc.

Devido ao crescente aumento nos volumes de dados gerados em tempo real e às limitações físicas (*hardware*) de um sistema centralizado, algoritmos tradicionais para agrupamento de fluxos de dados podem não ser capazes de processar fluxos de alta velocidade (*high speed data stream* ou *big data stream*), ou seja, que geram um grande volume de dados em um curto espaço de tempo. Para que a análise seja sustentável, é necessário que o processamento dos dados seja mais rápido que sua taxa de chegada, e em sistemas centralizados, os próprios limites físicos impossibilitam essa tarefa para fluxos realmente rápidos.

Existem na literatura vários trabalhos para processamento distribuído de fluxo de dados [Neumeyer et al., 2010; Morales & Bifet, 2015; Bifet et al., 2015; Shahrivari & Jalili, 2016]. Essa abordagem permite que vários nós, trabalhando em conjunto, sejam capazes de processar fluxos de alta velocidade. Em um sistema escalável, a quantidade de nós pode ser redimensionada a fim de adequar a uma necessidade de poder computacional. Contudo, a manipulação e manutenção de sistemas distribuídos é demasiadamente complexa, considerando todos os fatores que influenciam a velocidade, sincronicidade, tolerância a falhas, acesso distribuído/compartilhado,

entre outros. O modelo *MapReduce* simplifica e padroniza as operações distribuídas, abstraindo a maior parte dos problemas, mas poucas pesquisas para agrupamento em fluxo de dados exploraram essa alternativa.

Além do problema de escalabilidade para fluxos de dados de alta velocidade, um dos principais problemas em agrupamentos de dados de forma geral é a definição do número de grupos. O algoritmo k-means é um dos 10 algoritmos mais influentes no campo da mineração e dados [Wu & Kumar, 2009], e inspirou grande parte dos algoritmos para agrupamento de dados da atualidade. Uma das características do k-means, que foi herdada por muitos de seus sucessores foi a necessidade de se informar um número de grupos *a-priori*. Em alguns casos, isso realmente é necessário, mas em muitos casos reais, a quantidade de grupos é desconhecida por parte do usuário. Para fluxos de dados, esse problema é ainda maior. Como os dados evoluem, a quantidade de grupos pode variar ao longo do tempo. Um valor estático informado *a-priori* pode não ser adequado a todo o fluxo de dados. Apesar de existirem trabalhos voltadas para estimativa dinâmica do número de grupos para fluxos de dados [Silva et al., 2017], a maior parte dos algoritmos ignora esse problema [Masud et al., 2011], especialmente, no contexto de fluxos de dados em alta velocidade, no qual não existem trabalhos relacionados de acordo com a pesquisa realizada.

1.2 Objetivos

Nesta seção são apresentados os objetivos geral e específicos que foram desenvolvidos ao longo deste trabalho.

1.2.1 Objetivo Geral

A Figura 1.1 apresenta três sub-campos de pesquisa em agrupamento de dados e suas interseções: *a*) agrupamento escalável de dados por meio do modelo *MapReduce*, *b*) agrupamento de fluxos contínuos de dados, e *c*) Agrupamento de dados com estimativa do número de grupos (k). O objetivo deste trabalho reside na área x: agrupar de forma particional e escalável, fluxos de dados de alta velocidade, estimando dinamicamente o número de grupos.

1.2.2 Objetivos Específicos

Neste trabalho foram considerados os seguintes objetivos específicos:

- Mostrar a equivalência de qualidade entre as abordagens centralizada e distribuída (interseções e e x da Figura 1.1).
- Explorar novas formas de lidar com fluxos de dados de forma distribuída dentro do contexto do problema.

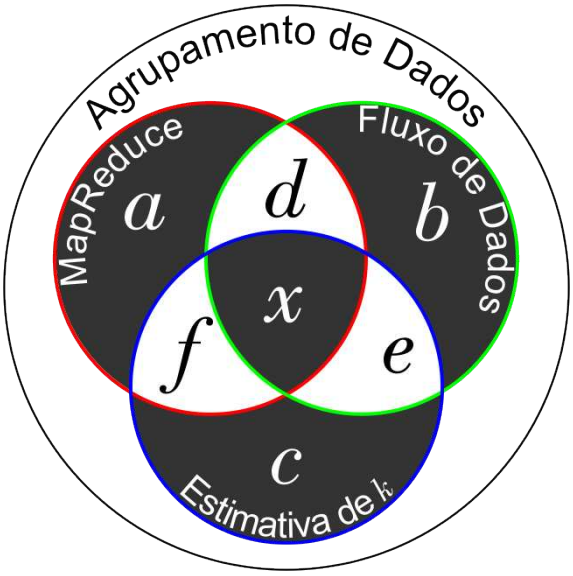

Figura 1.1: Áreas do Agrupamento de Dados. Adaptado de Cândido et al. [2017]

1.3 Organização da dissertação

A dissertação está organizada como se segue: o Capítulo 2 consiste em uma versão modificada do artigo Cândido et al. [2017], com a apresentação e experimentação de dois algoritmos para o problema em questão e o Capítulo 3 consiste em uma versão modificada do artigo Candido et al. [2018], onde são apresentados e experimentados três novos algoritmos com o mesmo objetivo. O Capítulo 4 apresenta as conclusões gerais do trabalho, obtidas por meio da análise dos resultados, e sugestões para trabalhos futuros.

A descrição dos artigos que compõem esta dissertação é apresentada a seguir:

- Cândido, P. L.; Naldi, M. C.; Silva, J. A. & Faria, E. R. (2017). **Scalable Data Stream Clustering with k Estimation**. In *2017 Brazilian Conference on Intelligent Systems (BRACIS)*, pp. 336–341.
- Candido, P. L.; Silva, J. A.; Faria, E. R. & Naldi, M. C. (2018). **Scalable Batch Stream Clustering with k Estimation**. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, Rio de Janeiro. IEEE.

Chapter 2

Scalable Batch Stream Clustering with k Estimation

Paulo L. Cândido, Murilo C. Naldi, Jonathan A. Silva, Elaine R. Faria
In *2017 Brazilian Conference on Intelligent Systems (BRACIS)*, 2017, Uberlândia. p. 336–341.

Abstract

The constant increasing of the generated data in real time has been creating new challenges for machine learning tasks and for one of their main branches: data clustering. The scenario of big data stream (high-speed data stream) has become reality. In order to deal with this scenario, new approaches are required. In this work, we present new techniques based on three clustering fields: Data Stream, MapReduce and automatic estimation of k from data. The goal is to cluster a high-speed data stream with varying number of clusters. Two scalable algorithms are proposed, based on centralized data stream algorithms. The first is based on the StreamKM++ and the second based on the F-EAC, an evolutionary algorithm used for batch clustering purpose. Results achieved the same high quality as the original centralized versions of the algorithms. The proposed techniques can be used instead the centralized ones when the velocity/volume is so high to fit in a centralized system.

2.1 Introduction

The machine learning techniques has been supporting the knowledge discovery over data in the last four decades [Gama, 2010]. The first techniques focused mainly in small static datasets (usually processed in batches). However, as technology evolved, new challenges arose and, therefore, the requirement for solutions that could overcome them. Among the main clustering fields, there are two of increasing interest over the last years: big data and data stream. Both of these fields deal with huge (virtually infinity) amounts of data, i.e. the whole dataset does not

fit in main memory. To work around it, big data techniques usually use a set of machines (clusters) and distributed computation. A parallel programming model, called MapReduce, was introduced in Dean & Ghemawat [2008] in order to deal with big data. Its main characteristic is the use of the functional methods *map* and *reduce* to perform operations parallely over the data. On the other hand, data stream techniques are based on few scans over data (a single scan in most applications) [Guha et al., 2000], once some data stream might be potentially unlimited. Another challenge of data stream is the concept drift [Aggarwal, 2007], i.e. the data may evolve over time, including changes in the number of clusters (k), known as concept evolution [Moulton et al., 2019]. Clustering algorithms must estimate this number (k) in order to achieve better quality, since the model also evolve with the evolving data [Silva, 2015; Masud et al., 2011].

With the increasing volume and velocity of real time data generation [The Software Alliance, 2015], novel clustering algorithms for high-speed data stream are required. This work aims to provide techniques to cluster a scalable amount of evolving data streams and dynamically estimate the number of clusters using MapReduce model. The goal is not to improve accuracy or performance; it is to enable the clustering of high-speed data stream with estimation of k when centralized systems cannot deal with the velocity/volume of data stream. Figure 2.1 shows different areas of data clustering and their intercections. Big data clustering using MapReduce model is represented by a; b represents data stream clustering; c represents general clustering with estimation of k; data stream clustering scaled by MapReduce is represented by d; e represents the data stream clustering with estimation of number of clusters; f represents big data clustering by MapReduce with k estimation; and x represents big data stream clustering with estimation of k. All the individual areas (a, b, c) and pairs (d, e, f) have been already explored, but our goal and major contribution is the intercection x that has not yet been explored.

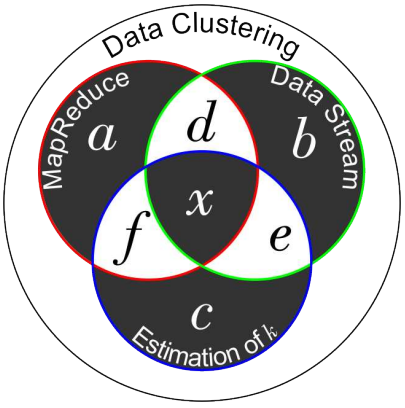

Figure 2.1: Data Clustering Areas

2.2 Related Work

Following the Figure 2.1, besides the notable work of the individual fields (a, b, c), the intersection between them (d, e, f) are relatively close to the purpose of this work, considering, to the best of our knowledge, there is no other work with the same goal as ours (which resides in the x of Figure 2.1).

In Bifet et al. [2015] is presented an open source machine learning library for scalable processing of data stream using MapReduce (d). There are also presented two traditional algorithms adapted to be scalable: CluStream [Aggarwal et al., 2004] and StreamKM++ [Ackermann et al., 2012]. However, as the original ones, the user must provide the number of clusters, which may be unknown or variable for some stream datasets. Inappropriate or static values may affect directly the quality of the model depending on the stream dataset.

For big data clustering using MapReduce with k estimation (f), an adaptation of F-EAC [Alves et al., 2006] is presented in Oliveira & Naldi [2015b], an evolutionary heuristic for clustering that is able to process scalable amounts of static datasets (batches) without requiring that the number of the clusters be provided. However, it is not designed to deal with data stream and concept drift.

Several algorithms focused on data stream clustering with estimation of k (e) were presented in Silva [2015]; Silva et al. [2017]. Although the algorithms are able to deal with large amount of data, they are not scalable, i.e. they are subject to physical limits of the centralized computing. In the scenario we presented, the high velocity/volume of data stream can make the centralized computing inappropriate. However, all the presented algorithms achieved good results with regard to quality and processing time. The estimation of k was able to keep the model close to reference clusters even with the variation of number of clusters due the concept drift.

Due to the high quality and low processing time achieved by Silva [2015], the best algorithms were selected to be adapted to MapReduce model in order to make them scalable. The selection was made by Pareto optimality analysis with two objectives: quality and processing time average. Among the dominants, we used the dominance count (the amount of dominated algorithms) to select two of them for this work: SKM-IBkM and FEACS-ISS.

The SKM-IBkM algorithm [Silva, 2015] is based on StreamKM++ [Ackermann et al., 2010], Bisecting kMeans (BkM) [Steinbach et al., 2000] and Incremental Simplified Silhouette (ISS) [Silva & Hruschka, 2016]. The algorithm keeps a set of buckets where the new instances are stored. As the buckets are filled, the objects are relocated and, when it is necessary, a merge-and-reduce process is performed, using the coreset tree structure [Ackermann et al., 2010], to summarize the data. The result is used in the *offline* component to: i) estimate the number of clusters using the BkM; and ii) create the clustering model using kMeans++. Once the model is obtained, the algorithm tries to keep it refreshed incrementally using the micro-clusters maintenance method of the CluStream algorithm to update a copy of the last coreset. The buckets continue to be updated on each new instance. While the incremental method is able to keep the model accurate, the *offline* component

is not performed. Only when a new cluster is detected by the incremental method, a signal is triggered to release the execution of the *offline* component.

The FEACS-ISS [Silva, 2015] is based on ISS [Silva & Hruschka, 2016] and on the evolutionary algorithm proposed in [Silva et al., 2017]. It does not have a summarization/abstraction step (which is common for several algorithms from the literature [Silva et al., 2013]). Nevertheless, the algorithm achieved good results on processing time due to its incremental method, which is able to update the model without necessity of performing the F-EAC all times. The incremental method updates the model, and when it detects significant changes, a variation of F-EAC [Alves et al., 2006; Silva et al., 2017] is performed, which is able to estimate the number of clusters and to induce the model.

2.3 Proposed Algorithms

To deal with big data stream and to achieve scalability, MapReduce was used to parallelize SKM-IBkM and FEACS-ISS [Silva, 2015]. In order to handle with data stream into MapReduce, we adopted discretized streams (D-Stream) [Zaharia et al., 2012b]. The main concept behind D-Stream is to treat the stream as a series of deterministic batches (micro-batches) retained for small time intervals and held reliably in a distributed system. For each micro-batch received, it is possible to perform transformations and actions through a structure called Resilient Distributed Dataset (RDD) [Zaharia et al., 2012a], and these operations are performed in parallel over distributed system, achieving scalability.

Although D-Stream treats the data stream as micro-batches, each instance is labeled with arrival timestamp to preserve the original data stream order. After arrival, each micro-batch is submitted to the proposed algorithms in order to update the clustering model, either incrementally or by performing the full clustering. This process repeats indefinitely, while the stream is active.

2.3.1 SKM-IBkM adaptation

The SKM-IBkM adaptation was divided into two components: *online*, with stages: *i*) coreset construction, *ii*) coreset update and *iii*) model update; and *offline*, with stages: *iv*) estimation of k and *v*) full clustering. Figure 2.2 shows an overview of the algorithm that is going to be introduced in this section. When a micro-batch arrives, it is submitted to two stages: i and ii. The stage i updates the coreset over the buckets. This coreset is only used when the *offline* component (stages iv and v) is performed, but it must be always updated, once it may be necessary at any time. There are two main operations on the stage i: the relocation of data in the buckets and the merge-and-reduce operation. This stage keeps L buckets of size m available to store data, where m is a user parameter, L is calculated by $\lceil log_2 \frac{n}{m} + 2 \rceil$ and the slide window size was used as n. The instances of a new micro-batch are put in the first bucket, in the arrival timestamp order, and when this bucket is full, the data is moved to second one; if the receiving bucket is already full, the merge-and-reduce operation must be performed.

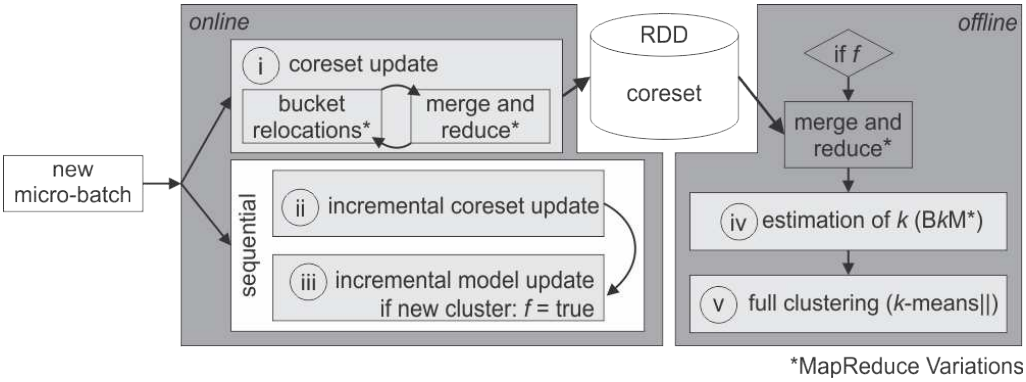

Figure 2.2: SKM-IBkM adaptation overview

The merge-and-reduce operation, on the stage i, uses a parallel version of coreset tree [Ackermann et al., 2010] to reduce $2m$ objects in m. The key idea to scale the coreset tree building is to process entire levels of the tree at once when it is necessary. Considering that the final tree must have m leaves, m split operations would be performed in the original algorithm. Our adapted version for MapReduce performs in the worst case (*a completely unbalanced tree*) m split operations and $log_2 m$ in the best case (*a completely balanced tree*), once our split operation is done by level instead of by node. Although the split of many nodes are calculated at once, they are not really splitted until they are selected to be. The selection of nodes to be splitted is made by the master node, that walks from the root through the tree (top-down), selecting the next step according to the node cost (the sum of squared errors of objects into a node was used as cost in this work) until reach a leaf. Figure 2.3 represents a coreset tree during a merge-and-reduce operation. Nodes marked with 'f' are leaves, thick nodes are the selected nodes to be splitted and dark nodes are closed nodes, i.e. nodes that were already calculated, but do not belong to the tree yet. Figure 2.3.a shows the walk (thick path) of driver program to select a leaf to be splitted. If the next level of the selected leaf (thick node) was not already calculated, thus all the next level is calculated at once, as shown in Figure 2.3.b, but only the children of the selected node are opened, i.e. become part of the tree. The selected node to be splitted in the Figure 2.3.b has the next level already calculated, thus no extra level calculation is required, their children are only opened and the tree cost is updated as shown in the Figure 2.3.c.

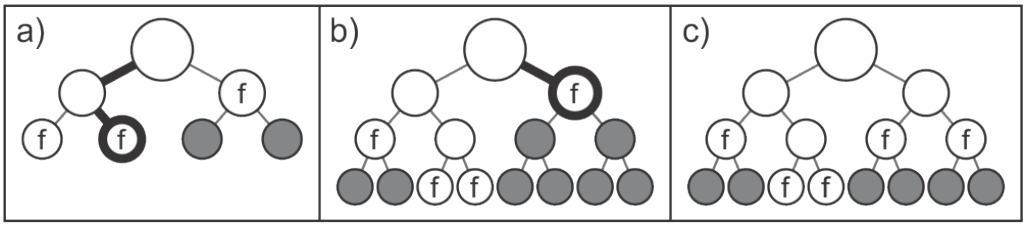

Figure 2.3: Coreset tree

In order to split the nodes of each level and control the access of the objects, it was used a transformation that groups all objects by their node index. It allows

applying the original logic in each node parallely through the distributed system. When m open leaves are reached, the objects of the closed nodes move up until reach an open node and, finally, the data of each node is summarized into a weighted point. The set of all weighted points over the buckets is the coreset. It is reliably stored over distributed system and stay available to *offline* component.

The stage *ii* (coreset incremental update) updates a copy of the coreset obtained in last run of *offline* component, using a variation of micro-cluster update method from CluStream [Silva, 2015]. The updated coreset is used to update the model on stage *iii* (model incremental update). The incremental stages (*ii* and *iii*) of the original algorithm are sequential, i.e. each step depends on the previous one. The coreset and model update caused by a new instance arrival may influence the result of the next instances. Because of this, and in order to preserve the order of the data stream, the sequential (incremental) processing was kept from the original version. To do so, as the instances arrive in micro-batches, they are sorted by timestamp and serialized to be incrementally processed by the driver program. Although it may be a bottleneck of the process, due the high sum of processing time of sequential tasks, the full clustering can be performed at any time if the incremental processing fails or reaches a time out. Nevertheless, in any case the processing time is negatively affected by the sequential process. After the incremental process, the updated model is evaluated and returned to user. Eventually, when a new cluster is detected, a signal f is triggered and the *offline* component is allowed to run.

The *offline* component (stages *iv* and *v*) is called periodically, and only executed if a signal f was triggered by the incremental method. When the *offline* component starts, the objects of the non-empty buckets of the stage i undergo again the merge-and-reduce process in order to reduce all points stored in buckets in a coreset C with only m objects. In the stage *iv* (estimation of k), the coreset C are submitted to a parallel variation of BkM we developed. The key-idea is the same of the coreset tree, the hierarchical tree is made level by level, on demand. The same structure of opened and closed nodes is used; the selection of the next node to split is made centralized and the entire level calculation is made parallel by node through the distributed system, applying the original algorithm [Steinbach et al., 2000] in each node to split them. Each new split represents a partition with $k + 1$ clusters and each partition is evaluated by a parallel version of simplified silhouette index (SS) [Oliveira & Naldi, 2015b]. Finally, the k of the best model is returned.

A variation of kMeans++, called kMeans|| [Bahmani et al., 2012], uses the estimated k and the coreset C in the full clustering stage (*v*). This algorithm is able to seed and run the kMeans over MapReduce. Once the original logic of the k-means++ seeding is sequential (each prototype is sampled according the already selected ones), the authors proposed an oversampling technique in order to parallelize the sampling step. In each iteration of sampling, several points are sampled parallely and independently, considering only the selected points of past iterations. After some iterations, it is expected to have more points than the required k (estimated by BkM), but the amount is low enough to fit into a centralized RAM. In order to find k prototypes, the traditional sampling method of k-means++ is applied: each prototype is sampled from selected points with probability proportional

to distance of the already selected prototypes. The scalable version of the k-means is the same used by Oliveira & Naldi [2015a]. For each iteration, a map method is used to find and assign each instance to a centroid and a reduce method is used to update the centroids. The result of the *offline* component is a clustering model valid to that moment.

2.3.2 FEACS-ISS adaptation

The second adapted algorithm was FEACS-ISS, with two stages: i) the incremental update of model; and ii) the full clustering using a parallel version of F-EAC. When a new instance arrives, the algorithm updates the model on stage i. If the model is updated with success, the incremental process continues.

In the original algorithm presented by Silva [2015], when the incremental processing of one element fails, the full clustering is allowed to run. A failure in this context means the partition has significant changes (i.e. a new cluster arises from the stream) and it is necessary to rebuild the model. In this adaptation, when a incremental update fails, it is aborted for the rest of the micro-batch and the full clustering is performed with all data of micro-batch. As the clustering model is needed only after the whole micro-batch is processed, there is no need for intermediary models. In the original version, the full clustering stage is called periodically (according to a user parameter) and it is only executed if a signal was raised by the incremental stage. In this version, the full clustering do not run in fixed periods of time. Instead, we only prevent consecutive executions of full clustering, imposing a minimum number of incremental micro-batch processing (user parameter) before a new execution of full clustering. While this number of iterations is not reached, the incremental method is forced to continue trying to update the model, as in the original algorithm. It allows the algorithm to adapt faster, once the full clustering is immediately executed (if it is allowed) when significant changes are observed.

Such as in the SKM-IBkM, the incremental update (stage i) is sequential (thus cannot be fully parallelized). Therefore, the instances of a micro-batch are sorted by timestamp and inserted in the model, one-by-one, following the original algorithm. The incremental processing of instances consists in comparing the individual SS index of new instance with the worst SS index of the closest cluster. If the SS index of the new instance is high, the instance is inserted, causing changes in selected cluster. If not, two executions of kMeans are performed with two different partitions: one considering that the instance was inserted in the closest cluster, even making the cluster's SS worse; and other considering the new instance as the centroid of a new cluster. Although the incremental processing is sequential, the executions of kMeans are performed in parallel through the distributed system. The best result among the two partitions (measured by SS) defines if a new clustering model must be made (when a new cluster is created) or not (when the the instance is inserted in a previously existing cluster or if it is not sufficient to define a new one).

In the stage ii, the SF-EAC [Oliveira & Naldi, 2015b] was used as a scalable variation of F-EAC. There are two main components in F-EAC which were adapted in SF-EAC to become scalable: the intensification component, k-means, and the

evaluator component, SS. The F-EAC is an evolutionary algorithm for clustering that, due the diversification of the initial population and the mutation operations, is able to estimate the number of clusters by comparing partitions with different values for k with SS index. The final result is the best solution (measured by SS) found through the evolutionary search. It is returned to the user as the model valid to the moment of the last micro-batch received from stream.

2.4 Experiments

In order to experiment the proposed algorithms, we used two artificial and two real dataset. The artificial datasets were obtained by MOA [Bifet et al., 2010] data stream generator. The artificial dataset 1 (A_10) has 10 dimensions while the artificial dataset 2 (A_100) has 100 dimensions. Both have 5 ± 3 clusters and 2×10^5 objects, that were divided into 200 micro-batches of 10^3 objects for the experiments. The third dataset is the well-known KDDCup99 dataset[1], 10% version, containing 4.94021×10^5 connection records with 22 types of intrusion and the normal connection. In its preprocessing step, the nominal attributes, including the label, were removed and the numeric ones were normalized. This dataset was divided into 99 micro-batches of 4941 objects plus one micro-batch of 4862 objects. The PaperAbstracts dataset, built by us, is a set of 6.65×10^4 abstracts of scientific papers equally distributed in 7 different themes/clusters of different knowledge areas: *Concrete*, *Hyperactivity*, *Investment*, *Neural Network*, *Photosynthesis*, *Proton* and *Tectonic Plates*. In order to preprocess the text dataset, *bag-of-words* and *TF-IDF* [Gupta & Gurpreet, 2009] were used to transform it into a feature vector and the techniques *stop-words*, *stemming* [Gupta & Gurpreet, 2009], *Luhm's cut* [Luhn, 1958] and *Var-TFIDF* [Nourashrafeddin, 2014] were used to reduce dimensionality from 1.52 $\times 10^5$ to 5871 attributes (stems). In order to consolidate the data as a stream, the objects were sorted ascending according to their distance of the center of their cluster, favoring the gradual evolution of the data. Moreover, clusters start and stop to arrive at different moments. This dataset were divided into 100 micro-batches of 665 objects.

In order to perform the experiments, a 6-machines cluster (1 master and 5 slaves) was launched on Amazon AWS[2]. The instance type `r4.large` was used with two cores of the Intel Xeon® E5-2686v4 with 15.25 GB of memory RAM. The algorithms was implemented in Java 8 and the framework Apache Spark 2.0.2[3] based on D-Stream was used.

For SKM-IBkM adaptation we used buckets of size 1000 for artificial datasets and 500 for real datasets. 5 attempts per split were used in the BkM. For FEACS-ISS adaptation, we used 10 as population size and 10 generations, which is more than sufficient to achieve good results according to Naldi et al. [2011]. All k-means executions were performed with 5 iterations which according empirical evidences

[1] http://kdd.ics.uci.edu/databases/kddcup99/
[2] https://aws.amazon.com/pt/ec2/instance-types/
[3] http://spark.apache.org/

[Anderberg, 1973] is enough to achieve quality results. For each algorithm, 10 independent repetitions were performed.

Once no other published work was found with the same goal of this, the results from centralized version of the algorithms were obtained to compare the quality of the adapted algorithms for the artificial datasets (A_10 and A_100). Table 2.1 shows a comparison of Adjusted Rand Index (ARI) average and standard deviation among the proposed and centralized versions of both algorithms for A_10 and A_100. In the artificial datasets, all algorithms achieved high quality (ARI average > 0.99) and low deviation (< 0.035). A Scott-Knott analysis [Scott & Knott, 1974] with 95% of confidence level based on the artificial datasets over ARI was used in order to verify the statistical resemblance/difference among the algorithms. It showed that there is no significant statistical difference among the centralized and scalable version of the FEACS-ISS algorithm. The difference between the versions of SKM-IBkM is too small according to the averages, but it is significant according to the Scoot-Knott test.

		A_10	A_100
SKM-IBkM	Centralized	0.9953 (0.027)	**0.9997 (0.003)**
	Scalable	**0.9962 (0.023)**	0.9973 (0.012)
FEACS-ISS	Centralized	**0.9923 (0.034)**	**0.9958 (0.027)**
	Scalable	**0.9950 (0.029)**	**0.9994 (0.005)**

Table 2.1: ARI Comparison - A_10 and A_100. [μ (σ)]

The charts in Figures 2.4 and 2.5 show a comparison of ARI on KDDCup99 and PaperAbstracts datasets, respectively, among the two proposed algorithms. The vertical axis represents the quality measured by ARI and the horizontal axis represents the evolution through the micro-batches. Besides the main lines representing the evaluation average, there are also bars showing the standard deviation. The ARI of KDDCup99 dataset was calculated based on attack classes [Olusola et al., 2010] instead of the dataset label (attack type). There are 4 attack classes plus the normal connection. The ARI results were higher than using the labels, showing the clustering is closer to the attacks classes.

The ARI decline observed in all experiments, visible in the charts of Figures 2.4 and 2.5, is due the appearance of new clusters (concept evolution). While the new instances do not represent significant changes on partition, justifying by SS the creation of a new cluster on the model, it is normal that the ARI suffers a decrease, once the data distribution on the clusters will not match with the reference clusters. In cases when the new cluster appears close to another one, the SS index may judge that is better keep them together in a unique cluster, affecting also the ARI. When new instances reinforce the appearing cluster or the cluster drifts apart from its closest neighbor, the new cluster is detected by the clustering model and the ARI increases. The PaperAbstracts dataset is more challenging for clustering tasks than the others in this work due to its low cohesion, i.e., the data features do not establish well defined clusters. It can be evidenced by the low SS index ($\mu = 0.1875$, $\sigma = 0.0748$). As can be seen in the Figure 2.5, this feature also negatively affects the ARI and increase its variation, once objects are not always related with the closest reference cluster.

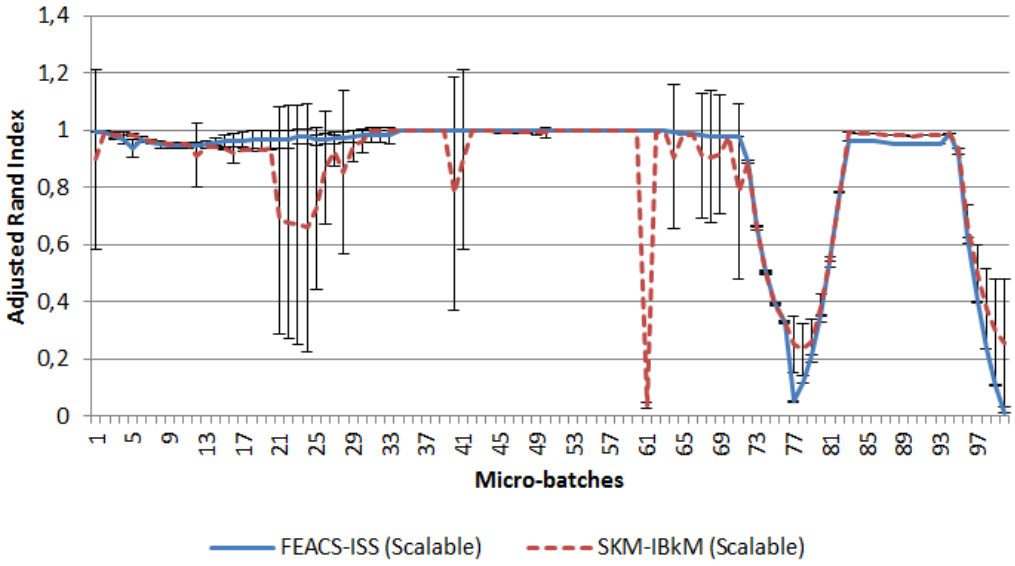

Figure 2.4: ARI Comparison - KDDCup99

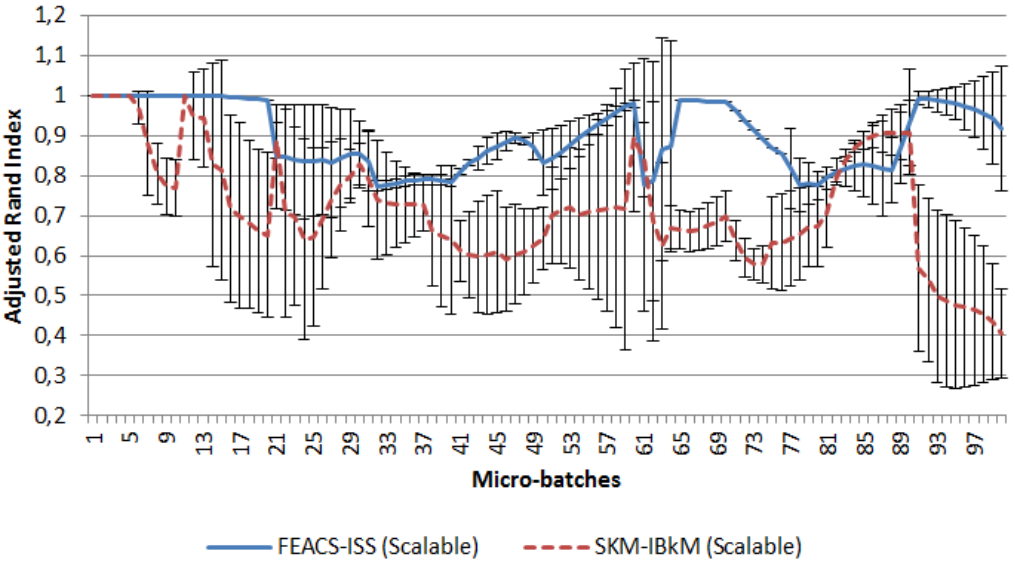

Figure 2.5: ARI Comparison - PaperAbstracts

In spite of concept drift and low cohesion (PaperAbstracts dataset), the proposed algorithms achieved high quality measured by ARI. The general average of ARI for real datasets are 0.8149 (σ=0.2442) for SKM-IBkM adaptation and 0.8892 (σ=0.2063) for FEACS-ISS adaptation. A Scott-Knott analysis with 95% of confidence level based on the real datasets over ARI showed that the FEACS-ISS adaptation is significantly superior to SKM-IBkM adaptation.

In the Table 2.2 it is possible to see the average and standard deviation of the processing time by micro-batch for all datasets in the scalable versions. As expected, the FEACS-ISS achieved lower processing time than SKM-IBkM, once the first uses an evolutionary algorithm for full clustering, which is well known by competitive processing time; and the incremental method is computationally simpler than SKM-IBkM's one. The centralized versions are much faster, as expected, due the several extra operations that must be performed by a distributed system, besides the network latency. The processing time average in seconds was 0.1994 ($\sigma = 0.1194$) and 0.2921 ($\sigma = 0.3068$) for SKM-IBkM and FEACS-ISS respectively in A_10 and A_100. Note that the centralized version will be the most suitable version when it can deal with the amount of data stream due the low processing time. However, in the presented scenario, centralized processing is not an option due the scalable amounts of data. In this scenario, the scalable versions are the most suitable, because they can deal with the scalable amounts of data through a machine cluster.

	SKM-IBkM - Escalável			FEACS-ISS - Escalável		
	μ	σ	μ/n	μ	σ	μ/n
A_10	34.657	86.710	0.173	9.730	8.867	0.049
A_100	40.344	88.681	0.202	15.577	11.845	0.078
KDDCup99	249.973	316.969	0.051	23.772	12.878	0.003
PaperAbstracts	103.807	140.024	0.156	46.554	79.256	0.070

Table 2.2: Processing time by micro-batch in seconds

2.5 Conclusion

In this work we presented two novel algorithms based on the best ones presented by Silva [2015] whose objective is to cluster scalable amounts of data stream adopting big data processing techniques and estimating the number of clusters. Our main contribution is to provide a way to process data stream through a distributed system in order to dealing with big data stream while the estimation of k ensures a high quality, once the variations of the number of clusters can be caught by the model.

Our experiments show that the presented algorithms achieve high quality results as the original ones. These algorithms are not designed for cases where centralized technique can be applied. For datasets that can be held by a single machine, the centralized version is the most suitable. These algorithms are designed for datasets where the centralized version cannot be applied. Although there are other algorithms that can deal with big data stream, our solution implements the estimation of k in order to reach better quality level.

Among both presented algorithms, the FEACS-ISS adaptation achieved better results in both perspectives, quality and processing time. The evolutionary component ensures competitive processing time with high quality while the incremental processing, not based in data abstraction, keeps the model updated using the dataset instances, which also aid the high quality achievement. The distributed system provides a propitious environment to keep and process scalable amounts of data, which is useful, once the data is not summarized.

The main limitation of the presented algorithms was the sequential/incremental processing that cannot be completely parallelized. Future work will focus on full parallelism. To do so, the instances in the same micro-batch should not have different time stamps, i.e. they should not be ordered among themselves. Therefore, the high-speed stream would be considered a sequence of sets of instances instead a sequence of instances. We believe that the micro-batch approach is fitted for real world applications with high velocity/volume, for which data instances arrive so quickly that a generalized analysis over a set may give better and faster results than considering the instances individually.

Chapter 3

Scalable Batch Stream Clustering with k Estimation

Paulo L. Cândido, Jonathan A. Silva, Elaine R. Faria, Murilo C. Naldi In *2018 IEEE Congress on Evolutionary Computation (CEC)*, 2018, Rio de Janeiro.

Abstract

Approaches that combine streaming algorithms and distributed computing have potential to deal with voluminous and high-speed data streams. Considering the data stream clustering task, also an important issue needs to be addressed, estimate the number of clusters dynamically, since it may vary due to concept drift. This work proposes three evolutionary-based algorithms to overcome these requirements. They are based in the discretized stream model, where the sequential batches of objects are distributed and processed in a parallel way using the MapReduce model. The proposed algorithms achieve superior experimental results, either in quality and processing time, overcoming the state-of-the-art.

3.1 Introduction

The increasing volume and velocity of data continuously generated impose several challenges to machine learning algorithms. These data streams are ordered sequence of objects with non-stationary data distribution [Gama, 2010]. A useful exploratory technique to analyze data streams involves clustering [Aggarwal et al., 2003; Silva et al., 2017].

Despite the development of several approaches for clustering, an important limitation of most of them is the prior definition of the number of clusters, k. The automatic estimation of k has been investigated using different strategies to *steady-state* data sets [Naldi et al., 2009]. The clustering problem is a well-known type of NP-hard optimization problem [Falkenauer, 1998] for $k > 1$, even for constant k and static data set. Evolutionary algorithms, such as the *Fast Evolutionary Algorithm for Clustering* (F-EAC) [Alves et al., 2006; Naldi et al., 2011], designed to

evolve clustering partitions during an evolutionary search guided by a validity index. Improved along the years, the F-EAC algorithm is the base for the algorithms proposed in this work due to its success in parallel and distributed computing [Naldi & Campello, 2015], data streams [Silva et al., 2017], and MapReduce [Oliveira et al., 2017].

Considering the data stream clustering task, several challenges need to be addressed [Silva et al., 2013]: i) data continuously generated, ii) unbounded size, iii) evolving data, represented by phenomena such as, changes in clusters (concept drift) and emergence of new clusters (concept evolution [Moulton et al., 2019]), and iv) variable number and size of clusters, that means clusters can appear, disappear, merge, and split. In order to deal with such challenges and estimate the k value, an extension of the F-EAC algorithm, named FEAC-Stream, was proposed for data streams in Silva et al. [2017], which assumes that changes in data distribution (concept-drift) can reflect in the data partition. The FEAC-Stream algorithm essentially keeps the data clusters incrementally updated and when a concept-drift has been detected, the evolutionary search starts to estimate the k value.

Although traditional data streams algorithms were designed to deal with data continuously generated, most of them are not able to handle with high volume and velocity streams. In this scenarios, parallelization may come in handy to increase the computational performance of the involved algorithms. In order to reach this goal, programming models must be used to allow large volumes of data to be analyzed in a timely manner, while guaranteeing safety against failures and scalability as the demand for data analysis as well as the database sizes grow. The MapReduce model [Dean & Ghemawat, 2008] is an abstraction that allows for simplified distributed and scalable programming.

The work Cândido et al. [2017] proposes two scalable data stream clustering algorithms, which use the MapReduce model, with automatically estimation of k from data. The first algorithm, SKM-IBkM adaptation, creates data summaries using distributed and parallel computation and estimates k by the Bisecting k-means algorithm [Steinbach et al., 2000]. The second, FEACS-ISS adaptation, uses a scalable version of F-EAC to cluster the data and estimate the number of clusters. Although they can handle with big data streams, they process the objects one-by-one, while preserving their arrival order. However, processing each object individually is not feasible for MapReduce model [Dean & Ghemawat, 2008], which spend similar effort to process a single object or a full batch of objects. Thus, the later is preferred and the discretized stream model was introduced in Zaharia et al. [2012b]. In this model, the data stream is presented as a sequence of batches of objects, instead of single objects. Such model vastly improves the performance of MapReduce frameworks.

In the sense of taking full advantage of MapReduce frameworks, this paper proposes three novels evolutionary algorithms for clustering streams discretized into micro-batches of objects, which are able to automatically estimate the number of clusters from data and detect concept drifts and novelty. The first one, SESC (Section 3.3.2), is based on two-component widely used for the data streams clustering algorithms: parallel and distributed data summarization, which allows the

adoption of fast centralized evolutionary clustering, based on the F-EAC algorithm [Alves et al., 2006]. The second algorithm, ISESC (Section 3.3.3), uses the full data from a landmark window [Silva et al., 2013] (without abstractions) to obtain a more detailed clustering model, which may be incrementally updated after each micro-batch. Dealing with raw (not abstracted) data requires higher computational power, which is granted by MapReduce inherent parallelism and distribution. The third proposed algorithm, ISESC-CD (Section 3.3.4), extends the ISESC with a new change-detection component that improves the performance of the algorithm reducing the frequency of full clustering applications to a minimum. Through an empirical evaluation in four artificial data sets and three real ones widely used in the literature, the proposed algorithms are compared to a state-of-the-art algorithm, the FEACS-ISS algorithm [Cândido et al., 2017], to demonstrate their efficiency and accuracy.

3.2 Related Work

Several data stream clustering algorithms have been proposed and most of them can be summarized into two steps [Silva et al., 2013]: data abstraction (online component) and clustering (offline component). The abstraction step keeps an online summary of the data using a particular data structure, such as feature vectors, micro-clusters, coreset tree, grids, etc. The clustering step obtains a data partition using traditional algorithms, like k-means. However, the choice of the best number of clusters is a challenging task for batch and stream scenarios.

Several approaches, such as Steinbach et al. [2000]; Pelleg & Moore [2000], estimate the number of clusters in batch scenarios by systematically obtaining data partitions with different numbers of clusters, using the k-means algorithm, and comparing them using internal validation indices, such as Simplified Silhouette Index (SS) [Hruschka et al., 2014]. However, such approaches could not achieve good quality results or/and spent too much time when compared with some evolutionary approaches [Naldi et al., 2009]. On the other hand, evolutionary algorithms, such as F-EAC [Alves et al., 2006; Naldi et al., 2011] evolve data partitions with variable k, and have shown more effective than approaches based on multiple executions of k-means, being successfully adapted to distributed computing [Naldi & Campello, 2015] and data streams [Silva et al., 2017].

SF-EAC [Oliveira et al., 2017] (Section 3.3.1) is a scalable version of F-EAC. It was built on top of the MapReduce model [Dean & Ghemawat, 2008] in order to distribute the processing through a machine cluster (distributed system). The authors showed that SF-EAC is able to find clusterings with quality equivalent or superior to other scalable algorithms [Garcia & Naldi, 2014; Debatty et al., 2014] but in less computational time.

Another algorithm based on F-EAC, the FEAC-Stream algorithm [Silva et al., 2017], was designed to deal with data streams. The first $InitSize$ objects are stored in a buffer and the F-EAC algorithm find a data partition. Then, an incremental method inspired on Aggarwal et al. [2003] is responsible to keep the clusters updated when subsequent objects arrive. Specifically, each object is absorbed by the closest

centroid of the actual data partition and the outdated clusters (according to their age) are removed. The distance of the arriving object and its closest cluster center is monitored by the Page–Hinkley test [Mouss et al., 2004], which can trigger warning and alarm states (according to threshold values). When a warning state is reached, the algorithm begins to store the arriving objects in a buffer, which are candidates to represent the new clusters that are arriving and may cause the decrease of the current data partition quality. Finally, when an alarm state is reached, it means that significant changes have been detected, which may evidence that the quality of the actual data partition is so poor that a new one must be built as soon as possible. Then, the full buffer is used into F-EAC in order to estimate k value. The incremental component of FEAC-Stream allows to perform low cost k estimation, once F-EAC does not need to be executed for each object that arrives or at fixed time intervals. The k estimation property enables FEAC-Stream to handle with concept-drift. However, the FEAC-Stream algorithm was proposed in a centralized way and cannot deal with scalable amounts of data. Their incremental method and the use of F-EAC for k estimation are the basis for our three proposals: SESC (Section 3.3.2), ISESC (Section 3.3.3) and ISESC-CD (Section 3.3.4).

FEACS-ISS Adaptation is presented in Cândido et al. [2017], which aims at clustering data streams with automatic estimation of the number of clusters. The algorithm was designed for MapReduce and inherently scalable. FEACS-ISS has two stages: i) the incremental model update; and ii) full clustering. After a model is built with the $InitSize$ objects by using SF-EAC, the arriving objects are processed one-by-one: their SS values are compared to the worst SS of their closest cluster; if it is higher, the objected is absorbed by that cluster and the process continues; if an object has an SS value which is less than the worst SS of the cluster, full clustering is performed (by the SF-EAC). The incremental update fails (and force the full-clustering) when a new cluster arrives or an existent cluster is split (the cluster disappearance is detected by outdated timestamps), but it was not designed to detect when two existent clusters merge. In addition, in order to keep the data stream order, its incremental processing is sequential. This characteristic makes it incompatible with the distributed computation, because the incremental processing is sequential and cannot be parallelized. In this work, we aim to overcome these limitations of the FEACS-ISS Adaptation.

3.3 Scalable Fast Evolutionary Algorithms for Clustering Streams

Big data frameworks can be used to grant scalability, reliability and performance increase for clustering. MapReduce [Dean & Ghemawat, 2008] is an efficient and fault-tolerant programming model that manages big data into a distributed and parallel system. Its key concept is to apply the functional methods, map and reduce, in parallel on each node of a machine cluster, instead of applying them sequentially. Due to the incompatibility of sequential and parallel approaches, MapReduce applications cannot deal directly with instance-based data stream algorithms, which

requires complex and costly bookkeeping protocols to perform per-object processing [Zaharia et al., 2012b]. In order to take advantage of the distributed and parallel systems, the discretized stream model was introduced in Zaharia et al. [2012b]. Its main idea is the discretization of the data stream into buckets called micro-batches, for which their objects are processed in parallel, saving resources and increasing performance of MapReduce frameworks.

3.3.1 Background approaches: F-EAC and SF-EAC

F-EAC was successful on clustering with the estimation of k in different scenarios [Naldi & Campello, 2015; Silva et al., 2017; Oliveira et al., 2017]. The main steps of F-EAC are presented in the Algorithm 1. The initial population is generated (step 1) by randomly drawing both the number of clusters and their prototypes for each individual (partition). During each generation, the population is refined by k-means algorithm (step 3) and evaluated by the fitness function (step 4), usually the SS. The selection of individuals is made in step 6 based on their fitness value, preserving the fittest individual by elitism. The selected individuals are mutated (step 7) by two operators designed to modify its number of clusters to promote diversity. The first operator (MO_1) removes clusters and assign the orphan objects to the closest remaining ones. The second operator (MO_2) splits clusters by selecting, for each cluster, a random object and the farthest object from it. Then, the two selected objects are used as seeds of each new clusters generated, moving the objects from the original cluster to the cluster with the most similar seed. Both operators are guided by fitness value, i.e., the probability of mutation for each cluster is inversely proportional to its fitness value. The choice of the mutation operator (between MO_1 and MO_2) depends on their previous results: if the use of one operator generates a new data partition with fitness value greater than its predecessor, this operator will be chosen again for this data partition in the next generation. Otherwise, the other operator will be chosen. If the individual in question belongs to the initial population or was selected through elitism, any of the operators can be chosen with the same odds. The best individual (elitism) and the ones resulted from mutation are used to composing the next generation (step 8). When the stop criterion (step 10) is reached, the best individual found during the evolutionary search is returned. This criterion may be enforced by imposing a limit on the number of generations and/or a minimum threshold for diversity.

SF-EAC [Oliveira et al., 2017] is the scalable version of F-EAC built over the MapReduce, in order to distribute and parallelize data processing. There are three operations of SF-EAC that depends on data: k-means execution, fitness calculation, and mutation operators. The MapReduce version of k-means calculates, in the *map* function, the distances between each object x_i and centroids $c_i \in C$, finds the closest centroid for each object, and returns a set of pairs $<$*centroid_ id, object_ position*$>$. The *reduce* function, calculates the new position of each centroid c_i by the average of *object_ positions* assigned to their *centroid_ id*. After that, the *map* function of the fitness method calculates the individual values of fitness (e.g. SS) for each object x_i based on centroids C, and returns a set of pairs $<$*centroid_ id, fitness_ value*$>$.

Algorithm 1: F-EAC

```
1: initialize a population of individuals (partitions)
2: repeat
3:    apply the k-means algorithm to each individual
4:    evaluate individuals (fitness function)
5:    if stop criterion is not reached then
6:       select individuals
7:       mutate individuals
8:       set the new population
9:    end if
10: until stop criterion is reached
```

During the *reduce* function of fitness method, the averages by *centroid_id* are used to calculate the fitness for clusters and partitions.

In order to reduce the data access of SF-EAC, the MO_1 does not make the object reassignments after removing clusters, which will be done during k-means. Additionally, MO_2 only duplicate the cluster centroid, building two new empty clusters with the same original centroid. The next k-means execution will randomly assign each object to its closest cluster, as the similarity of their centroids to objects are equal. The remaining F-EAC operations follow the original algorithm since they are processed in the master node and do not access the data.

3.3.2 Proposed algorithm 1: SESC

Scalable Evolutionary Stream Clustering (SESC) is composed of two phases: abstraction and clustering. The abstraction phase aims at summarizing the data in a parallel and distributed way, while clustering phase aims to sequentially build a clustering model based on these data summaries. An overview of SESC is presented in Figure 3.1. Assuming the discretized stream model [Zaharia et al., 2012b], the stream of objects arrives in a sequence of micro-batches $X_1, ..., X_n$. Each micro-batch X_t has its objects distributed to the machine cluster. Independently, each node performs the abstraction phase in order to keep its micro-clusters updated. Each micro-cluster has four components: N (the number of data objects), LS (the linear sum of the data objects), SS (the sum of squared data objects), and TS (the most recent timestamp of the data objects). The update has the following steps [Aggarwal et al., 2003]: first, q micro-clusters are created via the k-means algorithm. Usually, the number of micro-clusters q is larger than the number of (macro)clusters k, limited by the size of the data stream [Aggarwal et al., 2003]. Then, for all nodes in parallel, the micro-clusters are updated, as proposed by CluStream [Aggarwal et al., 2003], based on the arrival sequence of the objects. During the update, the objects are inserted into the closest micro-cluster or, if the distance of the object to the closest micro-cluster exceeds a maximum boundary, a new micro-cluster is created with the assigned object. As the number of micro-cluster is constant [Aggarwal et al., 2003], the creation of a micro-cluster requires the removal of another.

Specifically, if there are outdated micro-clusters, the oldest micro-cluster is removed. Otherwise, the two closest micro-clusters are merged.

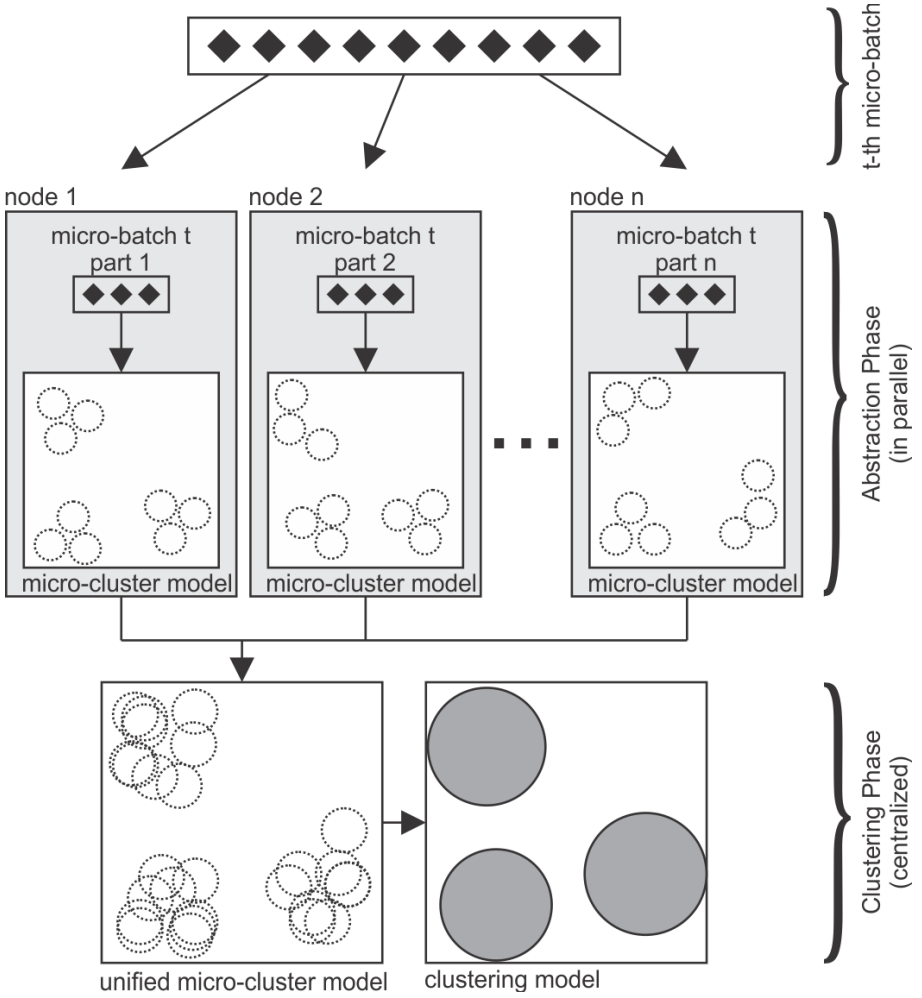

Figure 3.1: SESC algorithm

During the clustering phase, after all micro-clusters are updated, a unified micro-cluster model Z_u is generated by the union of micro-clusters from all nodes ($Z_u = Z_1 \cup Z_2 \cup ... \cup Z_m$). This unified model, containing abstract information of the stream until that point, is used as input for F-EAC (Section 3.3.1) adapted for weighted objects. The F-EAC evolves partitions of the unified model, with considerably less than the amount of data than the whole stream of objects, allowing a fast local execution of the algorithm. However, the use of data abstractions causes inevitable information loss from the original data. As the micro-clusters are processed independently, the unified model may contain micro-clusters with overlaps or different densities. That may not have a significant impact over F-EAC, as the algorithm ponders the micro-clusters centroid with the number of objects they represent. In the end, the fittest clustering is returned.

3.3.3 Proposed algorithm 2: ISESC

Our second algorithm, the *Incremental Scalable Evolutionary Stream Clustering* (ISESC), does not use data abstractions. Dealing with raw data requires more memory and processing time, but provides higher accuracy. The algorithm adopts a landmark window [Silva et al., 2013] to remove expired clusters from the model. ISESC relies on the high storage capacity of the cluster to store data, scalable thanks to MapReduce. Of course, this capacity is not infinite as the stream could be, but we believe it is high enough to store at least the objects of the landmark window since the oldest objects are discarded when expired.

An overview of ISESC is presented in Figure 3.2. After receiving the first micro-batch, the algorithm builds the first clustering model using the SF-EAC algorithm (Section 3.3.1). After that, the algorithm receives a sequence of micro-batches and have their objects distributed and submitted to an incremental update component.

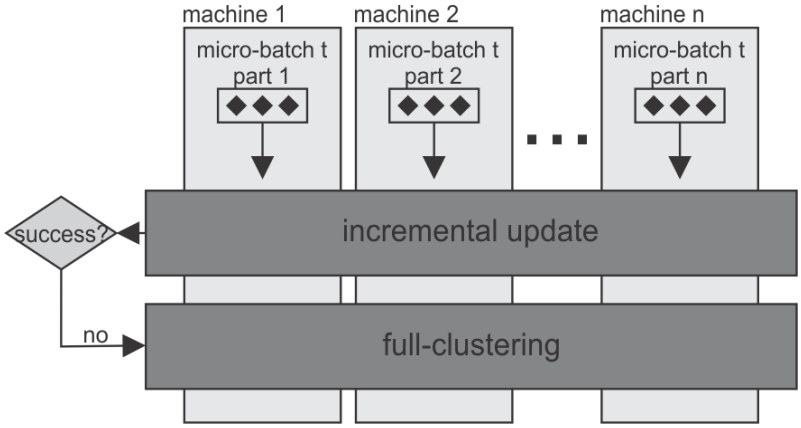

Figure 3.2: ISESC algorithm

Inspired by Aggarwal et al. [2003]; Cândido et al. [2017], but differently from SESC, the incremental update of ISESC considers the clustering model itself, not micro-clusters. Specifically, each cluster store two features: i) the farthest distance (fd) from an object in that cluster to its centroid, and ii) the lowest SS value from an object inside the cluster (ws). When a new micro-batch arrives, its objects are distributed by the MapReduce Framework. Then, the clustering model is mapped, distributed and compared with each object of the new micro-batch. If the distance of an object to the closest centroid is smaller than the fd of its respective cluster and the SS value of the object is higher than the clusters' ws value, the object is considered to fit the current model and is labeled as *success*. Otherwise, it is labeled as *fail*. If the entire micro-batch is labeled as *success*, i.e., no significant change was detected, then a fine tune on the model is made using MapReduce k-means and expired clusters are removed. Differently, if any object was labeled as *fail*, a significant change may be occurring. In this case, the incremental update fails and full-clustering is applied.

After being triggered, the full-clustering component applies SF-EAC (Section 3.3.1) over all non-expired stored objects. However, the initial population is not

randomly built. Instead, its composed by derivations of the current model. Given two initial boundaries k_{min} and k_{max}, a copy of the current model is adapted for each number of clusters $k' \in [k_{min}, k_{max}]$. For $k-1$ to k_{min} clusters, an iterative process of merging clusters is repeated until the desired number of clusters is obtained. This process consists of merging the closest pair of clusters, i.e., the clusters with the lowest value resulted from the distance between their centroids minus their radii (fd). For $k+1$ to k_{max} clusters, an iterative process of creating new clusters is repeated until the desired number of clusters is obtained. A new cluster is created using as seed the non-absorbed object (labeled as *fail*) which has the higher distance value in relation to the closest cluster centroid. The initial population is created with the original clustering model plus its derivations and SF-EAC is applied to evolve theses partitions. We believe that this methodology generates a population of clustering solutions with higher potential, as we believe that the stream would not suffer abrupt changes in most scenarios. If so, SF-EAC will need fewer generations to evolve solutions. It is also important to note that the SF-EAC does not limit its evolutionary search to solutions with $k \in k_{min}, k_{max}$, i.e., it is capable of finding the fittest solution out of these boundaries.

3.3.4 Proposed algorithm 3: ISESC-CD

The third proposed algorithm, *Incremental Scalable Evolutionary Stream Clustering with Change Detection* (ISESC-CD), is an extension of ISESC. It aims at reducing the processing time trying to avoid unnecessary executions of full-clustering, based on the relaxation of the incremental component of ISESC. Although the ISESC conservative evaluation may be useful for data sets with little or no concept-drift (which means that the clusters rarely changes), moderate concept-drift may often trigger the full-clustering unnecessarily. The ISESC-CD has an additional component to detect changes in the clusters being updated. Such clusters changes can be considered as the displacement of clusters centroids, the appearance of a new cluster, division and merging. The disappearing of clusters occurs when outdated clusters are removed. As shown in Figure 3.3, the change-detection component is located between the incremental and full-clustering. If a displacement is detected, an execution of k-means is enough to update the model, otherwise, the full-clustering is executed. Note that is not necessary to identify what exactly was the change neither how many clusters changed, as any of them will trigger the full-clustering.

The change-detection component consists of two steps: first, the split and merge operations of ISESC (presented in Section 3.3.3) are used to generate partitions with $k+1$ and $k-1$ clusters, respectively. After that, a parallel k-means (MapReduce version) is applied on the three partitions (original and derivations) and evaluated by SS. If the original partition does not have the highest SS value, then change is detected and full-clustering applied using SF-EAC as described in Section 3.3.3. This method may result in computational savings, as the processing time of the change-detection component is substantially lower than the full-clustering component. However, if changes are often detected, the change detection component becomes a burden and the original ISESC is preferred.

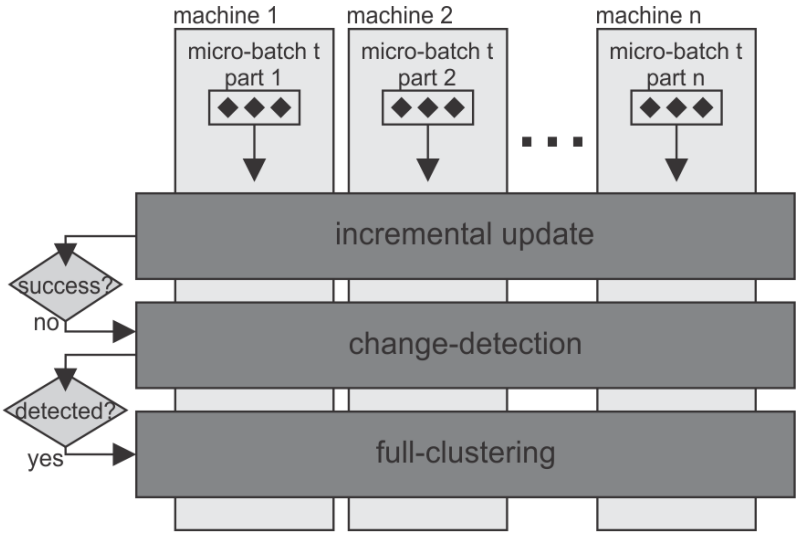

Figure 3.3: ISESC-CD algorithm

3.4 Experiments

In this section, we experimentally compare the algorithms proposed in this work with FEACS-ISS adapted for micro-batches. The FEACS-ISS is the dominant algorithm compared in Cândido et al. [2017] and is considered to be state-of-the-art mainly because, as far as the author's known, there is not in the literature a better clustering algorithm for data streams that has the advantages of being implemented in MapReduce (scalability, reliability, flexibility, parallelism) and is capable to automatically define and adapt the number of clusters during the stream. The algorithms were evaluated using both the Adjusted Rand Index (ARI) [Hubert & Arabie, 1985] to measure the quality of the data partitions and computational time to measure the performance.

The proposed algorithms were implemented in Scala. Apache Spark[1] was chosen for execute the MapReduce jobs, and Spark Streaming[2] was used as Discretized-Stream implementation. A cluster with ten commodity computers was used to execute the experiments; altogether the clusters provided 60 processing cores (3.5GHz) and 73GB (35GB usable by JVM) of RAM memory. It is important to note that the scalability inherited from MapReduce allows the replication of the experiments using larger and more sophisticated systems.

The characteristics of the seven data sets used in our experiments are presented in Table 3.1. The artificial data sets were built using the data stream generator RandomRBFGenerator [Bifet et al., 2010]. STAB is a stable data set with no displacement of clusters, a favorable scenario for the incremental component of ISESC and ISESC-CD. LAHI, HALI and HAHI data sets were obtained with different amounts of data, attributes, and clusters for generic experiments. For the YouTube Games data set (YTG) [Madani et al., 2013], the `text_game_lda_1000`

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

version was used and the noise class 31 (random games) was removed. The ARI of KDDCup99 (10% version) data set (KC99) [Lichman, 2013] was calculated based on attack classes (Normal, DOS, PROBE, R2L and U2R) instead of the data set label. The cosine dissimilarity was used for PaperAbstracts (PPA) and YTG data sets. The real data sets do not have some type of natural data sequence that could characterize them as a data stream. Thus, in order to consolidate them as data streams, we have chosen to sort the data objects according to their similarity, favoring the gradual evolution of the data with the creation and disappearance of clusters over time. All the data sets were divided into 100 micro-batches.

Data set	Type	Attributes	Instances	# of Clusters	Window size
STAB	Artificial	8	5×10^5	3..7	2.5×10^4
LAHI	Artificial	2	10^7	3..9	5×10^3
HALI	Artificial	64	10^5	3..9	5×10^5
HAHI	Artificial	64	10^7	10..20	5×10^5
KC99 [Lichman, 2013]	Real	34	494021	2..6	24705
PPA [Cândido et al., 2017]	Real	5871	6.65×10^4	2..6	6650
YTG [Madani et al., 2013]	Real	1000	8.79×10^4	2..7	2637

Table 3.1: Characteristics of artificial and real data sets. The columns show: the name and type of each data set, their number of attributes and instances, the range of the number of clusters, and the size of the landmark window.

In all the experiments, k-means was executed for 5 iterations to fine tune the data partitions, which is enough to achieve quality results according to studies in Anderberg [1973]. The F-EAC based algorithms were executed for 3 generations with a population of 10 individuals. For each data set, the first micro-batch was used by all compared algorithms to build the initial clustering model. In the incremental-update component of SESC, were kept 120 micro-clusters equally distributed on the machine cluster. All the experiments were repeated 10 times.

ARI comparisons among the four algorithms over the data sets are presented in the charts of Figure 3.4. The vertical axis presents the ARI score and the horizontal axis measures the micro-batch position in the stream from 1 to 100. For each algorithm, the main lines indicate the average of ARI and the vertical bars are their standard deviations. Concept drift in the streams causes quality variations, reflected by the ARI scores. The results allow concluding that there are no absolute best or the worst algorithm. However, for the majority of the evaluations, ISESC variations had the best quality results and SESC had the best performance.

SESC had the worst quality for PPA (Figure 3.4.f), with a quality loss caused by the natural information loss of its data abstraction. Nevertheless, it was the best algorithm for YTG data set (Figure 3.4.g), especially from micro-batches 43 to 63: during these micro-batches, there is an overlap among clusters, which made the other algorithms to merge them into a single cluster; but the data abstraction of SESC lessens the overlap effect, which improved the accuracy of the algorithm.

For the majority of the evaluations, ISESC and ISESC-CD were similar among themselves, which is expected since the only difference among them is the change detection component. ISESC-CD updates the clustering model only after change has been detected, which may take a few micro-batches of evaluations. This may

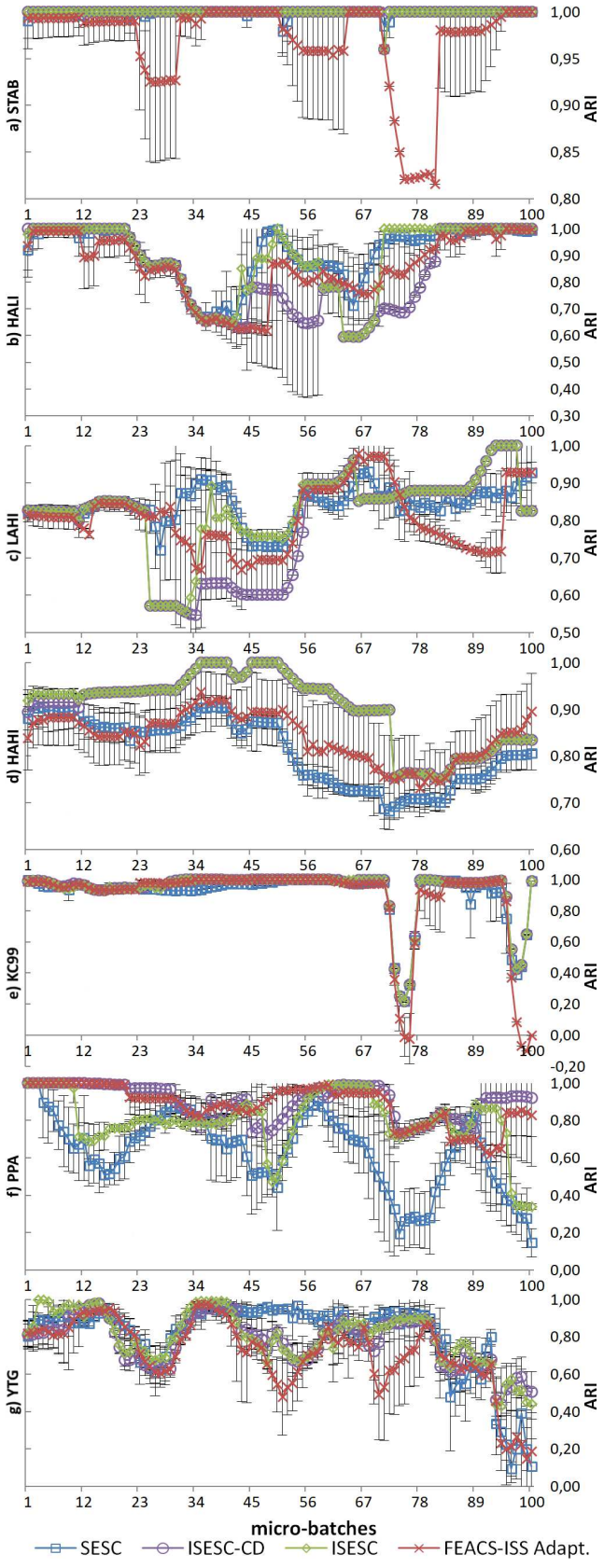

Figure 3.4: ARI Comparison

cause punctual quality variations, as observed from micro-batches 33 to 55 of LAHI data set (Figure 3.4.c) which the component fails to immediately detect change. Also, for PPA data set from micro-batch 10 to 33 (Figure 3.4.f) the ARI value of obtained data partition did not increase due to the cluster centers might be close enough to each other which according to SS fitness function is more suitable merging them into one.

Between the micro-batches 75 and 94 of LAHI data set (Figure 3.4.c), a new cluster appeared far from the most similar existent one (C), but with higher SS when compared to C's ws. Thus, the incremental component of the FEACS-ISS was not able to detect the change and these two clusters were merged into one. This did not happen with ISESC-CD, which detected and created a new cluster. On the other hand, an interesting behavior occurred between the micro-batches 66 and 74 of the LAHI data set: two clusters slowly overlapped each other, which triggered the ISESC-CD change-detection component, applied the full-clustering component and merged the clusters into a single one. Although the decision is correct according to SS index values, the reference partition still considered theses clusters independently, resulting in a higher ARI value for FEACS-ISS adaptation.

In order to quantitatively compare the quality of the algorithms, the ten ARI results of each algorithm for every micro-batch were analyzed using statistical tests. As the Shapiro-Wilk test [Shapiro & Wilk, 1965] rejects normality hypothesis for most of the reported results, we adopted the non-parametric statistical test of Friedman with 95% significance, followed by Fisher's least significant difference (LSD) for multi-comparisons [Conover, 1999]. After that, we counted the number of micro-batches each algorithm had the best result or equivalent, according to the applied tests. The resulted scores are displayed in Table 3.2, where the highest scores are in bold. In the artificial (well-behaved) data sets, ISESC had the best scores. However, the same cannot be said for the real data sets, where ISESC-CD was better for KC99 and PPA. ISESC and ISESC-CD had the best scores for STAB and HAHI artificial data sets. The only data set where SESC had the best score was in YTG data set, for the reasons above mentioned (overlapping clusters). FEACS-ISS adaptation had average scores. In general, in terms of ARI scores, the algorithms can be ranked: ISESC, ISESC-CD, FEACS-ISS adaptation and SESC.

	SESC	ISESC	ISESC-CD	FEACS-ISS Adapt.
STAB	98	**99**	**99**	66
HALI	76	**88**	72	51
LAHI	50	**74**	58	54
HAHI	1	**100**	**100**	24
KC99	51	72	**77**	52
PPA	15	46	**82**	80
YTG	**68**	66	51	34
Total	359	**545**	539	361

Table 3.2: Number of micro-batches with the best ARI result or a result with no statistical difference to the best, for each algorithm

For the sake of simplicity, we considered the total execution time of the com-

	SESC		ISESC		ISESC-CD		FEACS-ISS	
STAB	**18.0 (0.5)**	1	699.0 (7.3)	3	287.4 (1.3)	2	774.3 (9.3)	4
LAHI	**39.6 (1.1)**	1	2163.2 (134.6)	3	533.0 (11.9)	2	12197.5 (164.9)	4
HALI	**15.8 (0.3)**	1	854.0 (3.1)	4	300.8 (2.0)	2	417.3 (9.7)	3
HAHI	**36.9 (0.8)**	1	6681.8 (205.3)	3	781.5 (29.4)	2	12242.6 (207.0)	4
KC99	**121.9 (3.1)**	1	2652.6 (293.0)	4	822.4 (55.6)	2	993.8 (51.6)	3
PPA	5267.4 (450.5)	2	25663.2 (481.0)	4	5171.4 (260.2)	2	**2543.9 (560.8)**	1
YTG	**1249.1 (38.9)**	1	5178.2 (266.0)	4	2681.6 (263.4)	3	**1192.2 (125.8)**	2
Rank sum		8		25		15		20

Table 3.3: Means and standard deviation of the overall computational time in seconds, followed by the rank of the algorithm. The lower the rank is, the faster is the algorithm.

pared algorithms instead of the time for processing individual micro-batches. Table 3.3 shows a comparison of average and standard deviation (in brackets) of the processing time in seconds for each data set and its rank position. A Friedman Test with 95% of confidence level, followed by Fisher's least significant difference (LSD) for multi-comparisons, showed all algorithms are statistically different among themselves for each data set, except SESC and FEACS-ISS adaptation for YTG data set and SESC and ISESC-CD for PPA data set. The SESC was the fastest algorithm for most data sets (STAB, LAHI, HALI, HAHI and KC99, YTG), because of its data abstraction that allows increasing performance, it reached the result, for HAHI data set, up to 197 times faster than ISESC, 23 times than ISESC-CD, and 353 times than FEACS-ISS. In PPA and YTG data sets, FEACS-ISS Adaptation was the fastest algorithm, being up to 1.3 times faster than SESC and 7 times faster than ISESC in PPA data set; and twice faster than ISESC-CD in YTG data set. This was caused by the small number of objects and high dimensionality of these data sets, which is advantageous for the incremental processing of FEACS-ISS. Specifically, on the PPA data set, each micro-batch was processed by SECS in 94 seconds (on average, including the data abstraction and clustering phases) while FEACS-ISS adaptation could incrementally process 94% of micro-batches in 11 seconds (on average). Such behavior was not observed on larger data sets, where the incremental component of FEACS-ISS adaptation was stressed and became a bottleneck. However, SESC and FEACS-ISS do not have the best quality results for the data sets where their performance was considered the best. Finally, the ISESC was, in general, the slowest algorithm due to its very conservative incremental component that often triggered the full clustering, increasing its computational time. The ISESC-CD, designed to overcome this drawback, was faster than its original version for all data sets. Note that for STAB data set, the incremental component of ISESC could prevent 29% of the full-clustering executions, requiring one-quarter of processing time (on average) of the full-clustering. The Rank-sum (Table 3.3) shows that SESC was, in general, the fastest algorithm, followed by ISESC-CD, FEACS-ISS, and ISESC.

Figure 3.5 shows a processing time comparison over LAHI data set. This chart illustrates clearly the processing time behavior of each algorithm. SESC had a low processing time with a low variation. ISESC-CD had also a low processing time for the most micro-batches, because of its change-detection component. Only when this component detected some significant change, the full-clustering component was trig-

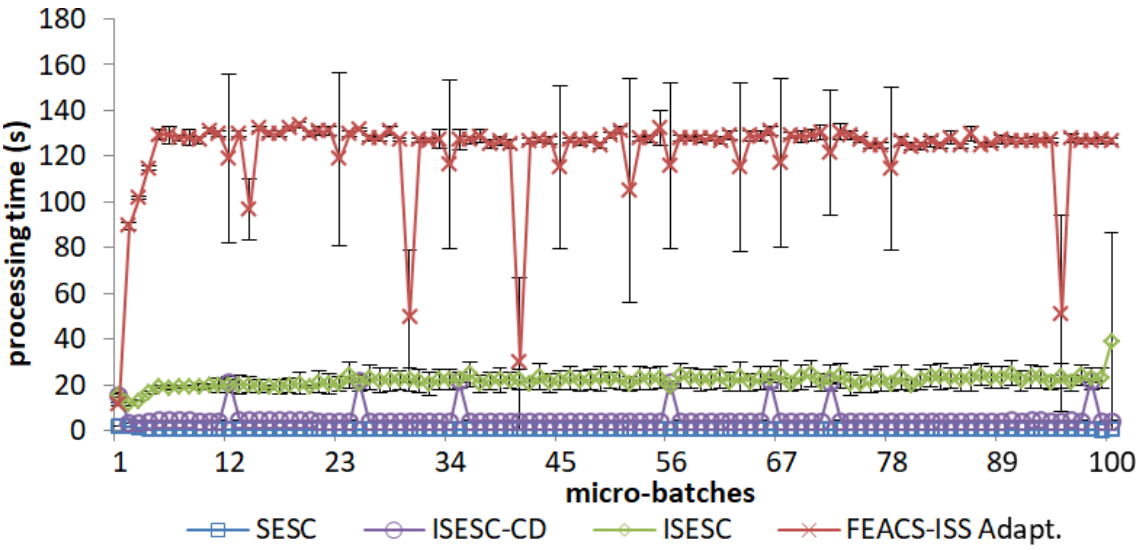

Figure 3.5: LAHI data set Processing Time Comparison

SESC	$O((n_{mb} * d * mc)/m)$
ISESC	$O((n_w * d * k_m * i * g * \|P\|)/m)$
ISESC-CD	$O((n_w * d * k_m * i * g * \|P\|)/m)$
FEACS-ISS	$O(n_{mb}^2 * d * k_m * i)$

Table 3.4: Asymptotic Computational Complexity Analysis

gered, increasing the processing time (which can be noted as small peaks in Figure 3.5). ISESC had a stable processing time through the micro-batches, once its conservative incremental component could not prevent the execution of full-clustering component for any micro-batch of this data set. FEACS-ISS adaptation had the highest processing time for this data set due to its incremental component. The observed valleys occurred when the incremental component failed early and the full-clustering was triggered. In this data set the incremental processing of FEACS-ISS adaptation was 3.2 times on average slower than its full-clustering component, which is presented as the valleys of Figure 3.5, showing that the incremental component is infeasible for such amount of data.

An asymptotic computational complexity analysis of a single micro-batch processing in the worst case is shown in Table 3.4, where n_{mb} is the size of micro-batch, d is the dimensionality of the data, mc is the number of micro-clusters, m is the number of worker nodes of distributed system, n_w is the size of landmark window, k_m is the maximum number of clusters found during the process, i is the maximum iterations of k-means, g is the maximum generations of F-EAC based algorithms, and $|P|$ is the population size. In the worst case, ISESC and ISESC-CD have the same asymptotic complexity, and SESC is the best, assuming that $mc < \frac{n_w}{n_{mb}} * k_m * i * g * |P|$. Although FEACS-ISS is a quadratic complexity in relation to n_{mb} in the worst case (a bottleneck was observed in Cândido et al. [2017]), the skip of the incremental phase when the first fail is observed avoids this case.

3.5 Conclusion

Three novel scalable batch stream evolutionary clustering algorithms that are able to improve partitions and estimate the number of clusters are proposed in this work and experimentally compared with a state-of-the-art algorithm. The results allow to concluding that, although each algorithm have had different results on each dataset, ISESC was the algorithm with the best quality results and SESC is the fastest one for the majority of the evaluations. Direct access and assessment of full batches of distributed data objects (including the arriving batch) allowed ISESC to reach the best quality results for most evaluations, with parallelization of the most burdensome components. On the order hand, the distributed abstraction component of SESC allowed to shrink large amounts of data into the main memory of a single processing node, which permitted the centralized fast application of the F-EAC full-clustering, resulting in the fastest executions for most data sets, especially the larger ones. It is important to note that ISESC-CD, which was designed to reduce the ISESC computational time with a new change-detection component, reached its goal. ISESC-CD had almost the same best quality results of ISESC, with a minor quality loss, but reduced the overall processing time of ISESC four folds on average.

The use of the discretized stream approach instead of processing objects one-by-one showed to be advantageous for MapReduce algorithms. The capability of assessing the whole batch of arriving objects at the same time allowed the design of improved incremental and change detection components. In general, FEACS-ISS has worse quality than ISESC and ISESC-CD, mainly due to the new insertion criterion of the incremental component and had worse computational time when compared to SESC, due to the abstraction approach, and ISESC-CD due to its detection component designed to avoid unnecessary full-clustering. It is possible to conclude that, for most scenarios, ISESC-CD dominated FEACS-ISS considering the quality of results and processing time. Thus, for the above-mentioned reasons, ISESC-CD showed to be the trade-off between quality and processing time among the compared algorithms.

Capítulo 4

CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho foram apresentados cinco algoritmos para agrupamento escalável de fluxos de dados contínuos com estimativa do número de grupos com o uso do modelo *MapReduce*. Todos foram experimentados, analisados e comparados para apresentação dos resultados. Conjuntos de dados reais e artificiais de variados tamanhos foram utilizados assim como testes estatísticos nas comparações entre algoritmos.

Com base nos experimentos apresentados, pode-se concluir que, por meio dos algoritmos propostos, é possível agrupar fluxos de dados de forma escalável sem perca de qualidade (medida por ARI). Os melhores resultados foram obtidos com a aplicação de algoritmos evolutivos sobre os dados brutos do fluxo, especificamente, os algoritmos ISESC e ISESC-CD. Ambos superaram em qualidade os algoritmos SESC e FEACS-ISS Adapt., que por sua vez supera o algoritmo SKM-IBkM Adapt. A utilização dos dados brutos reduz os possíveis impactos causados pela perca natural de informações devido a sumarização de dados, enquanto o modelo de busca evolutiva intercala diversificação e intensificação de uma população de soluções a fim de encontrar a que melhor se adéqua aos dados. Diferente do Bisecting kMeans, uma grande diversidade de soluções com variados valores para k são avaliados e mutados num processo evolutivo guiado.

O tempo computacional entre versões centralizadas e distribuídas são incomparáveis, mas dentre os algoritmos escaláveis, foram observadas grandes variações nesse aspecto. Os melhores resultados de acordo com o tempo computacional foram obtidos com abordagens de sumarização de dados mescladas à computação evolutiva. Sumarização de fluxos de dados é uma técnica tipicamente utilizada na literatura, mas nesse trabalho, foi aplicada de forma distribuída e escalável. Os dados já sumarizados são submetidos então a uma heurística evolutiva, que também contribui com sua velocidade. É importante ressaltar que o tempo de processamento em sistemas escaláveis está em relação inversa à capacidade computacional do sistema distribuído. Em aplicações reais, muitas vezes é possível reduzir o tempo de processamento aumentando a capacidade computacional, mas por rigor científico, cada artigo apresentado manteve os recursos fixos em todos os seus experimentos.

O processamento objeto-a-objeto, tradicionalmente utilizado por algoritmos clássicos, foi um limitador para escalabilidade dos algoritmos SKM-IBkM e FEACS-ISS. Apesar de usarem o poder computacional do sistema distribuído para realizar a maior parte das operações com acesso à dados, a condição de se manter os objetos do fluxo ordenado teve um alto custo no desempenho em conjuntos de dados maiores. A abordagem de micro-lotes em grandes conjuntos de dados, permitiu que os recursos computacionais fossem usados de forma mais adequada, obtendo-se um tempo computacional menor. Em contra-partida, o fluxo não é mais descrito como uma sequência ordenada de objetos, mas uma sequência ordenada de micro-lotes.

Apesar da melhor adequação de determinados algoritmos em determinados conjuntos de dados, o algoritmo ISESC-CD foi, de forma geral, o melhor avaliado dentre todos os algoritmos propostos, criando um balanceamento entre qualidade e tempo de execução. Sua qualidade é equiparada ao melhor avaliado, ISESC, enquanto seu tempo computacional foi em média o segundo melhor, ficando atrás apenas do SESC. O grande diferencial do ISESC-CD é o seu componente de detecção de mudanças. Ele foi responsável por uma redução de cerca de 75% do tempo computacional em relação ao ISESC enquanto manteve uma qualidade equivalente.

Trabalhos futuros podem seguir na direção de investigar o impacto de diferentes algoritmos de agrupamento nos componentes *full-clustering*, substituindo o F-EAC e SF-EAC. Diferentes técnicas de detecção de mudança também podem ser experimentados no ISESC-CD. E ainda, novos experimentos podem ser conduzidos utilizando mais conjuntos de dados, especialmente reais.

Referências Bibliográficas

Ackermann, M. R.; Märtens, M.; Raupach, C.; Swierkot, K.; Lammersen, C. & Sohler, C. (2012). StreamKM++: A clustering algorithm for data streams. *Journal of Experimental Algorithmics*, 17.

Ackermann, M. R.; Raupach, C.; Lammersen, C.; Sohler, C.; Märtens, M. & Swierkot, K. (2010). StreamKM++: A clustering algorithm for data streams. *Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments, ALENEX*.

Aggarwal, C. C. (2007). *Data Streams - Models and Algorithms*. Springer.

Aggarwal, C. C. (2015). *Data Mining*. Springer International Publishing, New York, NY, USA.

Aggarwal, C. C.; Han, J.; Wang, J. & Yu, P. S. (2003). A Framework for Clustering Evolving Data Streams. *Proc. of the 29th int. conf. on Very large data bases*, 29:81--92.

Aggarwal, C. C.; Han, J.; Wang, J. & Yu, P. S. (2004). A framework for projected clustering of high dimensional data streams. *Proceeding in VLDB '04 Proceedings of the Thirtieth international conference on Very large data bases*, 30:852--863.

Alves, V.; Campello, R. J. G. B. & Hruschka, E. R. (2006). Towards a Fast Evolutionary Algorithm for Clustering. *IEEE Congress on Evolutionary Computation*, pp. 1776--1783.

Anderberg, M. (1973). *Cluster Analysis for Applications*. Academic Press.

Bahmani, B.; Moseley, B.; Vattani, A.; Kumar, R. & Vassilvitskii, S. (2012). Scalable K-Means++. *Proceedings of the VLDB Endowment*, 5:622--633.

Bifet, A.; Holmes, G.; Kirkby, R. & Pfahringer, B. (2010). MOA Massive Online Analysis. *Journal of Machine Learning Research*, 11:1601--1604.

Bifet, A.; Maniu, S.; Qian, J.; Tian, G.; He, C. & Fan, W. (2015). StreamDM: Advanced Data Mining in Spark Streaming. *IEEE International Conference on Data Mining Workshop*.

Cândido, P.; Naldi, M. C.; Silva, J. A. & Faria, E. R. (2017). Scalable Data Stream Clustering with k Estimation. In *2017 Brazilian Conference on Intelligent Systems (BRACIS)*, pp. 336--341.

Candido, P. L.; Silva, J. A.; Faria, E. R. & Naldi, M. C. (2018). Scalable Batch Stream Clustering with k Estimation. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, Rio de Janeiro. IEEE.

Conover, W. (1999). *Practical nonparametric statistics*. Wiley.

Dean, J. & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107--113.

Debatty, T.; Michiardi, P.; Mees, W. & Thonnard, O. (2014). Determining the k in k-means with MapReduce. *CEUR Workshop Proceedings*, 1133:19--28.

Falkenauer, E. (1998). *Genetic Algorithms and Grouping Problems*. John Wiley & Sons, Inc., New York, NY, USA.

Gama, J. (2010). *Knowledge Discovery from Data Streams*. CRC Press, 1 edição.

Garcia, K. D. & Naldi, M. C. (2014). Multiple Parallel MapReduce k-means Clustering with Validation and Selection. *Brazilian Conference on Intelligent Systems*.

Guha, S.; Mishra, N.; Motwani, R. & O'Callaghan, L. (2000). Clustering Data Streams. *41st Annual Symposium on Foundations of Computer Science*.

Gupta, V. & Gurpreet, S. L. (2009). A Survey of Text Mining Techniques and Applications. *Journal of Emerging Technologies in Web Intelligence*, 1(1):60--76.

Hruschka, E. R.; de Castro, L. N. & Campello, R. J. G. B. (2014). Evolutionary algorithms for clustering gene-expression data. *Fourth IEEE International Conference on Data Mining*, pp. 403--406.

Hubert, L. & Arabie, P. (1985). Comparing partitions. *Journal of Classification*, 2(1):193--218.

Lichman, M. (2013). UCI machine learning repository. http://archive.ics.uci.edu/ml.

Luhn, H. P. (1958). The automatic creation of literature abstracts. *IBM Journal of Research and Development*, pp. 159--165.

Madani, O.; Georg, M. & Ross, D. A. (2013). On using nearly-independent feature families for high precision and confidence. *Machine Learning*, 92:457–477.

Masud, M. M.; Gao, J.; Khan, L.; Han, J. & Thuraisingham, B. (2011). Classification and Novel Class Detection in Concept-Drifting Data Streams under Time Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 23:859--874.

Morales, G. D. F. & Bifet, A. (2015). SAMOA: Scalable Advanced Massive Online Analysis. *Journal of Machine Learning Research*, 16:149--153.

Moulton, R. H.; Viktor, H. L.; Japkowicz, N. & Gama, J. (2019). Clustering in the presence of concept drift. In Berlingerio, M.; Bonchi, F.; Gärtner, T.; Hurley, N. & Ifrim, G., editores, *Machine Learning and Knowledge Discovery in Databases*, pp. 339--355, Cham. Springer International Publishing.

Mouss, H.; Mouss, D.; Mouss, N. & Sefouhi, L. (2004). Test of Page-Hinckley, an approach for fault detection in an agro-alimentary production system. In *2004 5th Asian Control Conference*, pp. 815--818, Melbourne.

Naldi, M. & Campello, R. (2015). Comparison of distributed evolutionary k-means clustering algorithms. *Neurocomputing*, 163:78 – 93.

Naldi, M. C.; Campello, R. J. G. B.; Hruschka, E. R. & Carvalho, A. (2011). Efficiency issues of evolutionary k-means. *Applied Soft Computing*, 11:1938--1952.

Naldi, M. C.; Fontana, A. & Campello, R. J. G. B. (2009). Comparison Among Methods for k Estimation in k-means. *Ninth International Conference on Intelligent Systems Design and Applications*, pp. 1006--1013.

Neumeyer, L.; Robbins, B.; Nair, A. & Kesari, A. (2010). S4: Distributed Stream Computing Platform. *IEEE International Conference on Data Mining Workshops*.

Nourashrafeddin, S. (2014). *Interactive term Supervised Text Document Clustering*. PhD thesis, Dalhousie University, Halifax, Nova Scotia.

Oliveira, G. V.; Coutinho, F. P.; Campello, R. J. & Naldi, M. C. (2017). Improving k-means through distributed scalable metaheuristics. *Neurocomputing*.

Oliveira, G. V. & Naldi, M. C. (2015a). *Estudo e Desenvolvimento de Metaheurísticas Evolutivas Escaláveis para Agrupamento de Dados*. PhD thesis, Universidade Federal de Viçosa.

Oliveira, G. V. & Naldi, M. C. (2015b). Scalable Fast Evolutionary k-Means Clustering. *2015 Brazilian Conference on Intelligent Systems (BRACIS)*.

Olusola, A. A.; Oladele, A. S. & Abosede, D. O. (2010). Analysis of KDD '99 Intrusion Detection Dataset for Selection of Relevance Features. *Proceedings of the World Congress on Engineering and Computer Science*, I.

Pelleg, D. & Moore, A. W. (2000). X-means: Extending K-means with Efficient Estimation of the Number of Clusters. *Proceedings of the Seventeenth International Conference on Machine Learning*, pp. 727--734.

Scott, A. J. & Knott, M. (1974). A cluster analysis method for grouping means in the analysis of variance. *Biometrics*, 30(3):507--512.

Shahrivari, S. & Jalili, S. (2016). Single-pass and linear-time k-means clustering based on MapReduce. *Information Systems*, 60:1--12.

Shapiro, S. S. & Wilk, M. B. (1965). An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611.

Silva, J. A. (2015). *Agrupamento de dados em fluxos contínuos com estimativa automática do número de grupos*. PhD thesis, Universidade de São Paulo, São Carlos.

Silva, J. A.; Faria, E. R.; Barros, R. C.; Hruschka, E. R.; Carvalho, A. C. & Gama, J. (2013). Data stream clustering. *ACM Computing Surveys*, 46(1):1--31.

Silva, J. A. & Hruschka, E. R. (2016). A Support System for Clustering Data Streams with a Variable Number of Clusters. *ACM Trans. Auton. Adapt. Syst.*, 11(2):11:1----11:26.

Silva, J. A.; Hruschka, E. R. & Gama, J. (2017). An evolutionary algorithm for clustering data streams with a variable number of clusters. *Expert Systems with Applications*, 67:228--238.

Steinbach, M.; Karypis, G. & Kumar, V. (2000). A Comparison of Document Clustering Techniques. Technical report, University of Minnesota.

The Software Alliance (2015). What's the Big Deal With Data? Technical report, The Software Alliance.

Tom White (2015). *Hadoop - The Definitive Guide*. O'Reilly Media, 4 edição.

Wu, X. & Kumar, V. (2009). *The Top Ten Algorithms in Data Mining*. CRC Press.

Zaharia, M.; Chowdhury, M.; Das, T. & Dave, A. (2012a). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *Symposium on Networked Systems Design and Implementation*.

Zaharia, M.; Das, T.; Li, H.; Shenker, S. & Stoica, I. (2012b). Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. *Proceeding in HotCloud'12 Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*.

Zhang, T.; Ramakrishnan, R. & Livny, M. (1997). BIRCH: A new data clustering algorithm and its applications. *Data Mining and Knowledge Discovery*, pp. 141--182.

Zhou, A.; Cao, F.; Qian, W. & Jin, C. (2008). Tracking clusters in evolving data streams over sliding windows. *Knowledge and Information Systems*, 15:181--214.

Zhu, Y. & Shasha, D. (2002). StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *Proceedings of the 28th International Conference on Very Large Databases*, pp. 358--369, Hong Kong.