EXAMENSARBETE INOM INFORMATIONS- OCH KOMMUNIKATIONSTEKNIK,
AVANCERAD NIVÅ, 30 HP
STOCKHOLM, SVERIGE 2017

Building Evolutionary Clustering Algorithms on Spark

XINYE FU

KTH
SKOLAN FÖR INFORMATIONS- OCH KOMMUNIKATIONSTEKNIK

Abstract

Evolutionary clustering (EC) is a kind of clustering algorithm to handle the noise of time-evolved data. It can track the truth drift of clustering across time by considering history. EC tries to make clustering result fit both current data and historical data/model well, so each EC algorithm defines snapshot cost (SC) and temporal cost (TC) to reflect both requests. EC algorithms minimize both SC and TC by different methods, and they have different ability to deal with a different number of cluster, adding/deleting nodes, etc.

Until now, there are more than 10 EC algorithms, but no survey about that. Therefore, a survey of EC is written in the thesis. The survey first introduces the application scenario of EC, the definition of EC, and the history of EC algorithms. Then two categories of EC algorithms - model-level algorithms and data-level algorithms are introduced one-by-one. What's more, each algorithm is compared with each other. Finally, performance prediction of algorithms is given. Algorithms which optimize the whole problem (i.e., optimize change parameter or don't use change parameter to control), accept a change of cluster number perform best in theory.

EC algorithm always processes large datasets and includes many iterative data-intensive computations, so they are suitable for implementing on Spark. Until now, there is no implementation of EC algorithm on Spark. Hence, four EC algorithms are implemented on Spark in the project. In the thesis, three aspects of the implementation are introduced. Firstly, algorithms which can parallelize well and have a wide application are selected to be implemented. Secondly, program design details for each algorithm have been described. Finally, implementations are verified by correctness and efficiency experiments.

Keywords: Evolutionary clustering; Spark; Survey; Data-intensive computing.

Abstrakt

Evolutionär clustering (EC) är en slags klustringsalgoritm för att hantera bruset av tidutvecklad data. Det kan spåra sanningshanteringen av klustring över tiden genom att beakta historien. EC försöker göra klustringsresultatet passar både aktuell data och historisk data / modell, så varje EC-algoritm definierar ögonblicks kostnad (SC) och tidsmässig kostnad (TC) för att reflektera båda förfrågningarna. EC-algoritmer minimerar både SC och TC med olika metoder, och de har olika möjligheter att hantera ett annat antal kluster, lägga till / radera noder etc.

Hittills finns det mer än 10 EC-algoritmer, men ingen undersökning om det. Därför skrivs en undersökning av EC i avhandlingen. Undersökningen introducerar först applikationsscenariot för EC, definitionen av EC och historien om EC-algoritmer. Därefter introduceras två kategorier av EC-algoritmer - algoritmer på algoritmer och algoritmer på datanivå en för en. Dessutom jämförs varje algoritm med varandra. Slutligen ges resultatprediktion av algoritmer. Algoritmer som optimerar hela problemet (det vill säga optimera förändringsparametern eller inte använda ändringsparametern för kontroll), acceptera en förändring av klusternummer som bäst utför i teorin.

EC-algoritmen bearbetar alltid stora dataset och innehåller många iterativa datintensiva beräkningar, så de är lämpliga för implementering på Spark. Hittills finns det ingen implementering av EG-algoritmen på Spark. Därför implementeras fyra EC-algoritmer på Spark i projektet. I avhandlingen införs tre aspekter av genomförandet. För det första är algoritmer som kan parallellisera väl och ha en bred tillämpning valda att implementeras. För det andra har programdesigndetaljer för varje algoritm beskrivits. Slutligen verifieras implementeringarna av korrekthet och effektivitetsexperiment.

Nyckelord: Evolutionär clustering; Spark; Undersökning; Datintensiv databehandling.

Acknowledgements

I'm grateful I have the opportunity to conduct my thesis at SICS and do research about Spark. Firstly, I'd like to express my thanks to Amir H. Payberah, my supervisor at SICS, for his kindly and patient guide for my project and thesis. I'd like to thank Ahmad Al-Shishtawy, a researcher at SICS. He helped me deal with many cluster configuration problems.

I also want to sincerely thank my examiner Sarunas Girdzijauskas and supervisor Vladimir Vlassov at KTH. They gave me much useful feedback during the project, especially feedback on my presentation so that I can write the thesis in a more clear logic.

Finally, I'd like to say thanks to my friend, Dalei Li. During the project, he answered my so many questions, helped me whenever I need. His experience and kind-heart helped a lot. I also want to thank my family and all other friends for their understanding and help during the project.

Contents

List of Figures

List of Tables

Abbreviations

AFFECT Adaptive Forgetting Factor for Evolutionary Clustering and Tracking. 15, 30

CP Change Parameter. 14

DYN-MOGA DYNamic MultiObjective Genetic Algorithms. 18

EC Evolutionary Clustering. 1, 12, 30

EFMs Exponential Family Mixture. 15, 19

EKM-MOEA/D Evolutionary k-means Clustering based on MOEA/D. 16

EMD Earth Mover Distance. 19

FIR Finite Impulse Response. 15

GMM Gaussian Mixture Model. 11, 19

HDD Historical Data Dependent. 19

HMD Historical Model Dependent. iii, 19

LDA Latent Dirichlet allocation. 11

MMM Multinomial Mixture Model. 19

NA Negated Average Association. 6

NC Normalized Cut. 6

NMI Normalized Mutual Information. 10, 18, 37

PCM Preserving Cluster Membership. 15

PCQ Preserving Cluster Quality. 15

RC Radio Cut. 6

RDD Resilient Distributed Dataset. ii, 1, 7

RDDs Resilient Distributed Datasets. 8

SC Snapshot Cost. 1, 10, 14, 38

SQ Snapshot Quality. 38

TC Temporal Cost. 1, 10, 14, 38

TQ Temporal Quality. 38

Chapter 1

Introduction

Clustering is a well-researched topic in data mining, but traditional clustering algorithms cannot handle time-evolved data well because of noise. Evolutionary Clustering (EC) solves the problem. Evolutionary Clustering inputs a sequence of time-evolved dataset $D_1, D_2, D_3, ..., D_t$, and outputs a sequence of clustering result $C_1, C_2, C_3, ..., C_t$. The current clustering result C_t should both fit current data D_t well, and fit previous data D_{t-1} or model C_{t-1} well. Since the history D_{t-1} or C_{t-1} is under consideration, the noise can be smoothed. To satisfy both criteria, snapshot cost (SC) is defined to measure how inaccurately current clustering represents current data, and temporal cost (TC) is defined to measure how inaccurately current clustering represents historical data or the distance between current and last clustering result. Different EC algorithms define different SC and TC and use different methods to minimize both of SC and TC simultaneously. The most common method is to define and minimize an overall cost $cost = (1 - CP) \times SC + CP \times TC$ which is controlled by a change parameter CP ranged from 0 to 1. Since the first EC algorithm reformulated by Chakrabarti et al. in 2006[3], there are already more than 10 EC algorithms.

Spark[34] is a data-intensive cluster computing tool which automatically provides job managing, locality-aware scheduling and fault tolerance similar to previous cluster computing model MapReduce[5]. Especially, Spark's abstraction resilient distributed dataset (RDD) can be cached in memory, which enables iterative algorithms running on clusters. Machine learning algorithms always include many iterative data-intensive computations and have to process a large number of data. Hence, many machine learning algorithms have been built on Spark to speed up computation.

1.1 Background

To understand the thesis, the reader should have the knowledge of basic clustering algorithms, such as k-means, spectral clustering, and hierarchical clustering[8]. It is because many EC algorithms are just variants of traditional clustering algorithms.

Since the EC algorithms will be built on Spark using Scala, and the relevant code will be explained in section 4, the reader should know the principle and concepts of Spark,

and understand the code example of Spark written in Scala[34].

In section 2, basic backgrounds of clustering algorithms and Spark will be given.

1.2 Problem

Currently, there are more than 10 EC algorithms, but no survey to arrange them in order. Every researcher who is interested in EC has to spend much time on collecting all papers and read them throughout. Even researchers read all papers; it is hard to recognize pros and cons of each algorithm, compare them with each other and select which to use. Hence, one of the problems of the thesis is

What's the theoretical framework of Evolutionary Clustering?

The problem can be divided into following sub-questions:

1. What's the criterion to categorize EC algorithms? How to categorize each of them?
2. What criteria to compare EC algorithms? What's the comparison result?
3. Which EC should perform better? Is it possible to give a performance prediction rank for each EC algorithm according to theory?

Moreover, EC is a kind of machine learning algorithm, and also always has many iterative data-intensive computations. Besides, EC algorithm handles time-evolved data, so EC should process a large number of data. Hence, EC algorithm is suitable to be implemented on Spark. However, until now, no such implementation exists, so the user of EC algorithms cannot process large dataset quicker with the help of Spark. Hence, the other problem the thesis solves is

How to implement Evolutionary Clustering algorithms on Spark?

The problem can be divided into following sub-questions:

1. What are suitable EC algorithms to implement on Spark?
2. How to implement them properly on Spark?
3. How much the implementations improve the efficiency of EC algorithms?

1.3 Purpose

To solve the first problem, a survey is written in the thesis. It covers categorization, comparison and performance prediction of each EC algorithms in theory.

To solve the second question, implementation details including selecting algorithms, designing program and experiments are covered in the thesis.

The project is carried out at SICS Swedish ICT. The main purpose of the project is to implement EC algorithms on Spark.

1.4 Goal

For the first problem, a written survey of EC algorithms will be delivered in in the thesis. For the second problem, four EC algorithms have been built on Spark. The implementations deal with batched dataset rather than data stream, and codes in Scala will be issued. Implementation details and experiment results will be given in the thesis.

1.5 Benefits, Ethics and Sustainability

EC has a wide application. For example, social network updates every day, EC algorithm is suitable for community analysis of the social network. For another instance, moving objects equipped with GPS sensors can be clustered continuously by EC algorithms. It can be extended to traffic jam prediction or animal migration analysis.

If there is a survey of EC algorithms, researchers can select a suitable algorithm with a quick glance at the survey. It saves much time. Moreover, if there are already EC implementations on Spark, it makes the large data processing easy.

In the project, only data collection is related to ethics. Since the main purpose of experiments is to test program efficiency, so only synthetic data is used. Therefore, there is nothing unethical in the project.

In the project, writing and implementation are carried out on the laptop, and the experiments are carried out on virtual machines. As a whole, the project is sustainable.

1.6 Methodology/Methods

This section is written according to [9]. In this section, philosophical assumptions, research methods, and research approaches will be discussed. The others will be discussed in chapter 3 and 4. The project solves two problems according to section 1.2 so that the methodology will be discussed for each of them.

The EC algorithm survey in section 3 solves the first problem. The survey is only in theoretical; it provides categorization, comparison and performance prediction of EC algorithms. Therefore, the survey is a qualitative research; it makes the conclusion based on existing papers rather than a large number of data. Similarly, its philosophical assumption is interpretivism which attempts to observe a phenomenon based on the meanings people assign to it rather than credible data and facts. Hence, the suitable research method is conceptual research which establishes concepts in an area of literature reviews. Since no large data set is used, the research approach is the inductive approach.[9]

For the second problem, it is a combination of the quantitative and qualitative research problem. At first, the suitable algorithms are selected based on 1) whether the EC algorithm is suitable for implementing on Spark; 2) whether the EC algorithm has a wide application. It is a qualitative research problem. The program design is also based on Spark design principles, so it is also a qualitative research problem. The corresponding

philosophical assumption is interpretivism. The research method is applied research because the theory is used to solve practical problems. Then the research approach is the inductive approach. However, prove the correctness and efficiency of implemented algorithms are quantitative research problems, and the related philosophical assumption is realism. Experimental research method and deductive research approach will be used to test whether implemented algorithms run faster on clusters than on one machine. If they run faster with the increment of worker number, the algorithms are implemented properly. On the other hand, a good experimental result reflects proper algorithm selection and program design.[9]

1.7 Delimitations

The following are delimitations of the thesis/project.

- Two papers [31] and [30] are not covered in the survey. It is because they are both much beyond the scope of knowledge. Just in case misunderstanding them and making wrong conclusions, they are not covered in the survey now. Including them in the survey can be regarded as the future work of the project/thesis.
- The survey is only in theoretical, and the performance prediction in the survey will not be tested in the thesis. It is because most of the time is spent in implementing EC algorithms on Spark, and there are too much EC algorithms to implement if performance prediction is needed to be tested. It is the future work.
- When testing the efficiency of implemented algorithms on Spark, only synthetic data is used. It will be the future work.

1.8 Outline

The thesis is structured as follows. Chapter 2 presents extend background of Spark and related work. Chapter 3 is a theoretical survey of Evolutionary Clustering algorithms. Chapter 4 explains implementation details of Evolutionary Clustering algorithms are given. Chapter 5 presents experiments and results. Finally the conclusion is given in chapter 6.

Chapter 2

Background & Related Work

In this chapter, background for understanding the following chapters are introduced concisely. Firstly, clustering concepts and popular clustering algorithms are presents. Then Spark concepts, code examples, Spark program design principles, and project related data structures are given. Finally, related work is presented.

2.1 Clustering Basics

In this section, clustering concepts and popular clustering algorithms are presents. Although there are many clustering algorithms, readers only need to know *k*-means, spectral clustering, DBSCAN, and agglomerative hierarchical clustering in order to understand the following sections.

2.1.1 What's Clustering

Clustering is a well-studied subject including data clustering and graph clustering. Data clustering allocates a batch of data points into several groups,[8] and graph clustering partitions a connected graph into several subgraphs (like Figure 2.1 shows).[18] Data points (nodes) in the same group (subgraph) are more similar to each other than Data points (nodes) in the different groups (subgraphs).

2.1.2 *k*-means

k-means[12] is a data clustering algorithm. It uses k centroids to represent k clusters. Data point belongs to the cluster with the nearest centroid. Therefore, *k*-means's objective is to minimize the following cost function

$$D(X, C) = \sum_{c=1}^{k} \sum_{i \in c} ||x_i - m_c||^2, \tag{2.1}$$

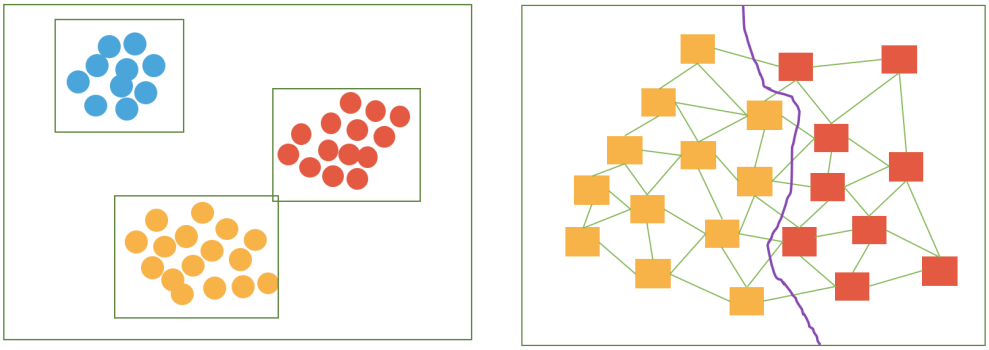

Figure 2.1: Data clustering (left) and graph clustering (right)

where X is a dataset, x_i is an observation, C is a clustering result, c is a cluster and m_c is the centroid of a cluster. The algorithm starts with initializing k centroids randomly or deliberately. Then iterate following two steps until certain iteration or getting a satisfied result. (i.e. the cost function 2.1 is low enough.):

- Allocate each observation to its closest centroid;
- Update each centroid as the expectation of observations belong to it.

2.1.3 Spectral Clustering

Spectral clustering is a graph clustering algorithm, the graph it processes presents as a proximity matrix in which each entry represents an edge value of two nodes. It can also deal with data clustering problem because observations can be transformed to similarity matrix first.[26]

The first step of the spectral clustering is to generate a positive definite similarity matrix W. (Gaussian similarity function, dot product, etc. get positive definite similarity.) Here the notations are defined. D is a diagonal matrix with elements corresponding to row sums of W. L is an unnormalized graph Laplacian matrix calculated by $L = D - W$. $\mathcal{L}$ is the normalized Laplacian matrix calculated by $\mathcal{L} = I - D^{-\frac{1}{2}}WD^{-\frac{1}{2}}$. X is the first k eigenvectors of W or Lor $\mathcal{L}$ associated with the top-k eigenvalues of the matrix; it can be regarded as n k-dimensional observations.[26]

Spectral clustering has three variants - optimizing negated average association (NA) or normalized cut (NC) or ratio cut (RC). All of them are NP-hard, so the algorithm turns to optimize the relaxed version problem. The three problems can be solved as follows:[26]

1. Optimize NA: Compute X of W. Then do k-means to X.
2. Optimize RC: Compute X of L. Then do k-means to X.
3. Optimize NC: Compute X of $\mathcal{L}$. Normalize n observations of X, then do k-means to normalized X.

2.1.4 DBSCAN

DBSACAN is a density-based clustering algorithm. It is not only robust to noise but also can handle any shape of the cluster without knowing the number of clusters previously. Data points can be divided into core nodes, boundary nodes, and noisy nodes. $MINPTS$ and EPS are two parameters should be set in advance. $MINPTS$ is a threshold of point number, and EPS is a radius threshold. If a node has no less than $MINPTS$ of the node within EPS, it is a core node. Core node links all its neighbors within EPS. Every node will be judged whether it is a core node. As a result, nodes directly or indirectly connected form a cluster, and the remaining single nodes are noisy nodes. Within a cluster, except for core nodes, the remaining nodes are boundary nodes. [6]

2.1.5 Aggromerative Hierarchical Clustering

Agglomerative Hierarchical Clustering[8] is also suitable for both data clustering and graph clustering. Similar to spectral clustering, the algorithm starts to generate a proximity matrix for all data points (nodes). Then it iterates the following steps:

1. Merge two points with the largest similarity;
2. Replace rows and columns of merged points in similarity matrix by a row and a column of the new point. The similarity value of the new point is the average or maximal value of the two merged points.

After several iterations, the data points will be generated into a bottom-up binary tree. Tree nodes represent a merge of two points. Then the tree is cut at a certain height to obtain a flat clustering result.

2.2 Spark Basics

In the thesis, four EC algorithms are implemented on Spark, so this section introduces RDDs & Spark concepts, Spark code examples, Spark program design principles, and project related data structures.

2.2.1 RDDs

The data-intensive application includes many data-intensive computations which process a large number of data independently by the same operations. Since data-intensive computations are independent of data, distributed computation speeds up the process. Cluster computing frameworks like MapReduce and Dryad have successfully implemented large-scale data-intensive applications on commodity clusters. They automatically provide locality-aware scheduling, fault tolerance, and load balancing. However, they cannot be applied to an important class of applications - data reuse applications (e.g., iterative machine learning). It is because they are built around an acyclic data flow model and store intermediate data into stable external storage, which causes inefficient data reuse due to data replication, disk I/O, and serialization. [34][33]

Spark solves the problem by introducing an abstraction called resilient distributed datasets (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. RDDs are lazy, so old RDDs will be discarded if memory is limited. However, if user caches an RDD in memory across machines, it can be reused again and again and won't be discarded. Hence, Spark can deal with data reuse applications.[34][33]

RDD can only be created by transformation operations from stable storage data or other RDDs; RDDs can be used by actions to return a value or export data to a storage system. Since RDDs is lazy, only an action is performed on an RDD; the RDD is generated by previous transformations. Spark also provide scalability and fault tolerance automatically. The computation process of a program will be drawn as a lineage, so lose RDD can be rebuilt according to the lineage.[34][33]

The dependencies between RDDs includes narrow and wide dependencies. Narrow dependencies mean each partition of the parent RDD is used by at most one partition of the child RDD, it allows for pipelined execution on one cluster node. Recovery after a node failure is more efficient with a narrow dependency, as only the lost parent partitions need to be recomputed. Wide dependencies mean multiple child partitions may depend on the RDD. Recovery after a node failure requires a complete re-execution for wide dependencies. Hence, if an RDD has wide dependencies with other RDDs, it is better to cache it.[34][33]

2.2.2 Spark

Spark is a language-integrated API for RDDs in an unmodified version of Scala. The system runs over the Mesos cluster manager and can read data from any Hadoop input source using Hadoop's existing input plugin APIs.[34][33]

Spark consists of a driver program and a cluster of workers. The driver program that implements the high-level control flow of their application, and the workers can store RDD partitions in RAM across operations.[34][33]

Except for RDDs, another two abstractions of Spark for parallel programming are parallel operations and share variables. Parallel operations will be introduced in section 2.2.3 with code examples. There are two types of shared variables - broadcast variables and accumulators. The broadcast variables are immutable and distributed on workers. If a user wants to share a fixed variable with all workers, (different from local variables on a worker,) the broadcast variable is a good choice. Accumulator allows update of a shared variable, but only addition is allowed.[34][33]

2.2.3 Spark Code Examples

Spark common used parallel operations and their example codes are listed here.[24]

- **map:** *map* is a transformation, it transforms each element of an RDD by a function. For example,

$$val\ rdd2 = rdd1.map(\_ * 0.05), \tag{2.2}$$

each element of $rdd1$ multiplies 0.05 and saves in $rdd2$.

- **foreach:** *foreach* is an action, it passes each element through a user provided function. For example,
$$rdd1.foreach(println(\_)), \tag{2.3}$$
each element of $rdd1$ is printed out. map and $foreach$ are different, the map result of each element will be saved in a new rdd, but $foreach$ just uses each element of rdd without saving anything.

- **reduce:** *reduce* is an action, it combines dataset elements using an associative function to produce a result at the driver program. For example,
$$val\ sum = rdd1.reduce(\_ + \_) \tag{2.4}$$
calculates the sum of elements in rdd1. It's easy to express as
$$val\ sum = rdd1.sum(). \tag{2.5}$$
If each element of an RDD is a pair, the first element of the pair is *key* and the second element is $value$. Then
$$val\ keySum = rdd1.reduceByKey(\_ + \_) \tag{2.6}$$
means elements with the same key will be added together, and a list of pairs like (key, sum) will be returned.

- **groupByKey:** $groupByKey$ is a transformation. Similar to $reduceByKey$, $groupByKey$ groups values with the same key, the grouped values are stored in a Iterable[V]. For example,
$$val\ keyGroup = rdd1.groupByKey() \tag{2.7}$$
returns a list of pairs like $(key, group)$.

- **collect:** *collect* is a action, it sends all elements of the dataset to the driver program. For example,
$$val\ keyGroup = rdd1.collect() \tag{2.8}$$
sends rdd1 to driver program.

2.2.4 Spark Program Design Principles

Based on the concepts of RDDs and Spark, the following program design principles are concluded.

- Parallelize computation as much as possible.
- Cache RDDs if they will be reused for several times, or it's a global parameter of the last iteration which can't be calculated again.

- Unpersist cached RDDs if it won't be used again.
- Use broadcast variables and accumulators properly.

2.2.5 Project Related Data Structure

Spark mllib provides three Distributed Matrices which distributively store matrix content in workers. Some machine learning algorithms use similarity matrix, so Distributed Matrices are useful.

- **CoordinateMatrix[22]:** CoordinateMatrix is a sparse matrix, and every entry in the matrix is a MatrixEntry, like $MatrixEntry(rowIndex, columnIndex, value)$. MatrixEntries are stored distributively and randomly. Since MatrixEntry is a small unit data structure, CoordinateMatrix doesn't give too much pressure to each worker. Also, map and reduce are easy to be conducted for CoordinateMatrix.
- **IndexedRowMatrix (or RowMatrix)[23]:** IndexedRowMatrix (or RowMatrix) stores matrix as a RDD of IndexedRow (or Row). IndexedRow has a row index, but Row doesn't. RowMatrix and IndexedRowMatrix provide functions like PCA and SVD. However, if a similarity matrix is stored in IndexedRowMatrix or RowMatrix, a Row(IndexedRow) may be too large for the memory. Hence, if it doesn't need to use a specific function like SVD or PCA, IndexedRowMatrix should be avoided for large dataset.
- **BlockMatrix[20]:** BlockMatrix divides a matrix in several blocks, and stores blocks distributively. BlockMatrix can do matrix multiplication, addition, and subtraction.

2.3 Evaluation of Evolutionary Clustering

For EC algorithm, the goal is to minimize SC and TC simultaneously. However, algorithms have different SC and TC, so it's difficult to compare their results. Moreover, the ultimate aim of EC is to track the drift of true clustering, low SC and TC do not necessarily translate into a good quality in tracking the drift of true clustering. Therefore, the external criterion is needed to directly evaluate the application of interest. [14]

In the experiment section, the synthetic data will provide the true classes of observations (label), so external criterion can be used to compare clustering result of different algorithms.

Normalized mutual information(NMI) is one of the external criterion, which measures difference between clustering result and the label. The clustering result is expressed as $\Omega = \{\omega_1, \omega_2, ...\omega_j\}$, and the classes (label) are expressed as $C = \{c_1, c_2, ...c_j\}$.

NMI doesn't request the same number of class number and cluster number, and each partition in classes compares with each partition in clusters. The formula of NMI is [14]. NMI is ranged between 0 to 1. The higher NMI, the higher clustering quality.

$$
\begin{aligned}
&NMI(\Omega, C) = \frac{I(\Omega, C)}{[H(\Omega) + H(C)]/2'} \\
&where, I(\Omega, C) = \sum_k \sum_j \frac{|\omega_j \cap c_j|}{N} log \frac{N|\omega_k \cap c_j|}{|\omega_k||c_j|}, \\
&where, H(\Omega) = -\sum_k \frac{|\omega_k|}{N} log \frac{|\omega_k|}{N}
\end{aligned}
\tag{2.9}
$$

2.4 Related Work

Until now, as far as we know, there is no survey of EC so that no related survey will be mentioned here. However, the related work of EC will be discussed in Section 3.7, because they will be compared with EC there.

Spark has provided clustering implementations *k*-means, Latent Dirichlet allocation (LDA), Bisecting k-means, and Gaussian Mixture Model (GMM) on its official website.[21] However, there is no implementation of EC on Spark as well, so no related work will be mentioned here.

Chapter 3

A Survey of Evolutionary Clustering

In this section, the first problem mentioned in section 1.2 will be solved. The survey of Evolutionary Clustering EC explains the theoretical framework of EC in details.

Before answering the sub-questions, basic concepts of EC is talked about. Section 3.1 talks about the application scenario which can't be solved by traditional algorithms and but is suitable for EC. Then section 3.2 defines EC, gives general workflow of EC algorithms, and gives the categorization criteria. After that, section 3.3 introduce history of EC.

Then section 3.4 and 3.5 introduce all EC in their categories. Moreover, the comparison criteria and result are listed in section 3.6. Also, the performance prediction of EC algorithms is given in section 3.8. Besides, the related work of EC is introduced in section 3.7.

3.1 Application Scenario

In the real application, data and graph for clustering aren't always static. Data points may move, and graph nodes may change connections with each other. Hence, a sequence of new clustering result should be generated continuously. However, traditional clustering methods can't handle the case, just like the following two examples.

Figure 3.1 shows a time-evolved data clustering example called traffic jam prediction. Moving objects equipped with GPS sensors are to be clustered to predict traffic jam. A red group and a yellow group move from A to B by the same route at a different time. The signal of one red member' sensor is bad, so he's still located in A when other members are in B. According to the graph, this member will be clustered into the yellow group. Obviously, it's wrong. If the GPS works, the member will be clustered into the red group. Hence, we need a clustering algorithm which can cluster the noise into the correct cluster.

Figure 3.2 shows a time-evolved graph clustering example called communities of dynamic networks. Communities are clustering result of social networks. Eliza is interested in beauty makeup and follows many makeup bloggers on her social networks. She

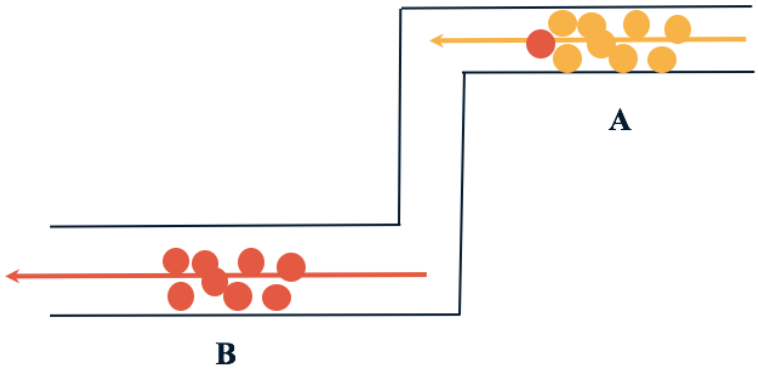

Figure 3.1: Application Scenario: Traffic jam prediction

is clustered into makeup community in 2015. Her interest changes slowly from makeup to fashion in two years, so her community changes from makeup to fashion until 2017. One day in 2016, a major political event happens in Eliza's country. She commented on some posts about the event and concerned with how it is going. If using traditional clustering algorithms, Eliza is going to be clustered into a political community that day, but it's not the truth obviously.

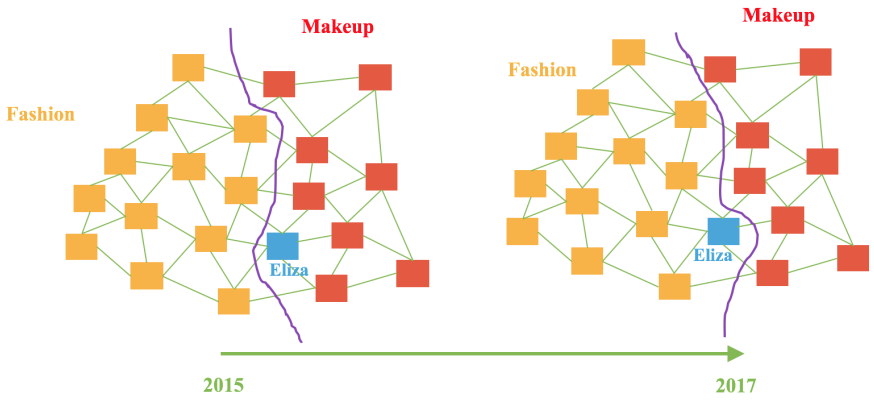

Figure 3.2: Application Scenario: Communities of dynamic networks

In both examples, the traditional clustering algorithms can't handle noise well. Therefore, new clustering algorithms to deal with time-evolved clustering problem are expected.

3.2 Evolutionary Clustering

Based on the two examples, data points (nodes) to be clustered change slowly, and behave anomaly sometimes. As a result, the clustering drifts in the long term, and also have short-term variation. If an algorithm can reflect the long-term drift of clustering, it can deal with time-evolved clustering problem.[4]

Evolutionary clustering (EC) algorithms are such algorithms to solve the time-evolved

clustering problem. They reflect long-term drift of clustering so that they are robust to short-term clustering variations. [4]

Evolutionary clustering algorithms work as follows. A sequence of time-stamped data is inputted, EC algorithms output a sequence of a clustering result. The current clustering result not only fits the current data well but also don't deviate too dramatically from the recent history.[3] Notice, the main objective of EC is still to fit current clustering result well with the current data. EC never only depends on history.

Let's see whether EC solves the two examples above. For traffic jam prediction, EC algorithms which remember historical distance among members can handle the case. For communities of dynamic networks, EC will cluster Eliza into makeup or fashion community than a political community since EC considers Eliza's previous actions.

There is one thing to notice. All algorithms talked about here are in the online setting. The online setting means the algorithm can't see following batches of data when clustering current batch of data. On the contrary, off-line setting means all batches of data are available when clustering.

As stated above, clustering result of EC algorithm should both accurately reflect current data, and similar to the clustering at the previous timestep. Hence, each EC algorithm defines a snapshot cost (SC) and a temporal cost (TC) to reflect both constraints respectively.[3]

Snapshot cost (SC) measures how inaccurately current clustering represents current data. A higher SC means a worse clustering result for current data. [3]

Temporal cost (TC) measures the goodness-of-fit of the current clustering result concerning either historical data features or historical clustering results. A higher TC means a worse temporal smoothness. If TC measures the distance between current and last clustering models, the algorithm can be categorized into **model-level** algorithm. If TC measures how inaccurately the current clustering represents the historical data, the algorithm can be categorized into **data-level** algorithm. The categorization result will be explained in 3.4 and 3.5.[3][4].

After defining its own SC and TC, an EC algorithm tries to minimize both of them simultaneously. Most of EC algorithms combine SC and TC into an overall cost function:

$$Cost = CP * TC + (1 - CP)SC, \tag{3.1}$$

where CP is change parameter which controls how much history affects clustering result. If $CP = 0$, it's traditional clustering algorithm. If $CP = 1$, only history will be taken into consideration. [3]

3.3 History of EC

In 2006, Chakrabarti et al. published the first paper about EC. The paper defines the general framework of Evolutionary Clustering as stated in section 3.2. It also put forward two greedy heuristic EC algorithms - Evolutionary k-means and Evolutionary agglomerative hierarchical clustering. Both algorithms are model-level algorithms. They don't try

to minimize their defined SC and TC, but just take historical model into consideration when updating current model.[3]

Inspired by the first paper, Yun Chi et al. extended EC to Spectral clustering algorithms in 2007. The paper put forward one data-level EC algorithm (PCQ) and one model-level EC algorithm (PCM). Both algorithms minimize overall cost like formula 3.1, so they are in a more rigorous framework than Evolutionary k-means and Evolutionary agglomerative hierarchical clustering.[4]

In the following years, many papers extended traditional clustering algorithms to EC frameworks. For example, Ravi Shankar et al. extended frequent itemsets into a model-level EC algorithm in 2010.[19] Yuchao Zhang et al. extended DBSCAN into a model-level EC algorithm in 2013.[36]

Algorithms mentioned above only extend specific clustering algorithms into EC framework, and select change parameter manually. Kevin S Xu et al. formulated AFFECT in two papers in 2010 and 2014.[28][29] AFFECT is a data-level algorithm, it adjusts data similarity by history first, then do clustering. AFFECT optimizes change parameter automatically and can extend to all clustering algorithms with proximity matrix as input. Min-Soo Kim et al. formulated a similar data-level algorithm to AFFECT in 2009, but the algorithm can't calculate CP automatically and is only designed for dynamic networks clustering.[10] What's more, James Rosswog et al. put forward another similar algorithm to AFFECT called FIR in 2008.[17] The difference is AFFECT uses only the history of the last timestep, but FIR uses a longer history.

Not only traditional clustering algorithms can be extended to EC framework, whatever methods minimize SC and TC simultaneously can be regarded as EC algorithms. Multi-objective optimization methods are suitable to minimize SC and TC at the same time. Hence, Jingjing Ma et al. and Francesco Folino et al. put forward two EC algorithms respectively based on multi-objective optimization in 2010 and 2011.[7][13] Statistical methods are also used to create new EC algorithm. For example, Jianwen Zhang et al. put forwarded an EC algorithm in 2009 which formulates EC as density estimation of Exponential Family Mixture (EFMs).[35]

Just as we mentioned in section 3.2, only online EC algorithms are talked about. Offline algorithms like [27] and [1] aren't mentioned in the thesis.

3.4 Model-level EC Algorithms

In this section, all existing EC algorithms will be introduced briefly by the category of data-level or model-level algorithms. Since space is limited, details of algorithms won't be covered here.

3.4.1 Evolutionary *k*-means

The Evolutionary *k*-means formulated by *Chakrabarti* is a greedy approximation algorithm which changes only the second iterated step in standard *k*-means introduced in section 2.1.2. Updating centroids consider both expectations of current observations and

the position of the closest centroid of the last timestep. Although Evolutionary k-means is a greedy algorithm, it used SC and TC to measure clustering quality. SQ (which is inverse proportional to SC) of Evolutionary k-means is [3]

$$sq(C, U) = \sum_{x \in U} (1 - min_{c \in C} ||c - x||), \tag{3.2}$$

and TC is

$$hc(C, C') = min_{f:[k] \to [k]} ||c_i - c'_{f(i)}||. \tag{3.3}$$

3.4.2 EKM-MOEA/D

EKM-MOEA/D is Evolutionary k-means Clustering based on MOEA/D. The algorithm defined the same SQ and HC as Evolutionary k-means Clustering introduced in section 3.4.1. However, EKM-MOEA/D optimizes SQ and HC instead of a greedy heuristic. MOEA/D is a multi-objective optimization method, so SQ and HC can be optimized simultaneously by MOEA/D. The details of MOEA/D is beyond the scope of the thesis. The experiment of EKM-MOEA/D demonstrate the result outperforms EKM.[13]

3.4.3 Evolutionary Agglomerative Hierarchical Clustering

Evolutionary Agglomerative Hierarchical Clustering first defines the SC and TC. Node similarity means similarity of two children of the node. Snapshot quality (be inversely proportional to SC) is defined as the sum of tree node similarity. The distance between two points means nodes number along with the route between the two points on the tree. The difference between two points' distance expresses two clusterings' difference. Therefore, TC is defined as the difference expectation of pairs of objects' distance between last timestep and current timestep.[3]

Although the paper defines SC and TC, the algorithm doesn't minimize them. The paper formulates four greedy heuristics, each of heuristics changes the points merge conditions as follows:[3]

1. **Squared:** Merge two points with the highest $SQ - CP * HC$ at each timestep
2. **Linear-Internal:** Merge two points with the highest $SQ - CP * (HC + penalty)$ at each timestep. The penalty performs when the merge for nodes that are still too close compared to the last timestep.
3. **Linear-External:** Merge two points with the highest $SQ - CP * (HC + penalty)$ at each timestep. The penalty performs when the merge for nodes will cause an additional cost to merge the third node.
4. **Linear-Both:** Merge two points with the highest $SQ - CP * (HC + penalty)$ at each timestep. The penalty combines penalties of Linear-Internal and Linear External.

3.4.4 Evolutionary Spectral Clustering - PCM

Evolutionary Spectral Clustering provides relaxed NA and NC optimization solutions for a data-level EC algorithm (PCQ) and model-level EC algorithm (PCM).

PCM SC is NA (NC) of current clustering result and current data. TC for PCM is the difference of clusters from two timesteps as

$$tc(X_t, X_{t-1}) = \frac{1}{2}||X_t X_t^T - X_{t-1} X_{t-1}^T||^2, \tag{3.4}$$

which doesn't request the same number of cluster across time. In PCM, the only step different from Spectral clustering is to calculate the first k eigenvectors of $CP * X_{t-1} X_{t-1}^T + (1 - CP) * W_t$ for NA and $CP * X_{t-1} X_{t-1}^T + (1 - CP) * D_t^{-\frac{1}{2}} W_t D_t^{-\frac{1}{2}}$ for NC.[4]

3.4.5 Evolutionary Clustering with DBSCAN

Evolutionary clustering with DBSCAN remember the neighbor vector of core nodes at each timestep $\overrightarrow{C_t}$, the vector is after temporal smoothness by a computation of M_t original vector at t and C_{t-1}. SC is defined as $\sum_i (c_t^i - m_t^i)^2$, and HC is defined as $\sum_i (c_t^i - c_{t-1}^i)^2$. The overall cost like formula 3.1 is optimized, so the temporal smoothness is like:[36]

$$\overrightarrow{C_t} = (1 - cp)\overrightarrow{M_t} + cp\overrightarrow{C_{t-1}} \tag{3.5}$$

For example, at $t = 1$, there are 5 core nodes with 4, 5, 3, 4, 3 neighbors respectively, EC with DBSCAN remembers a vector $C_1 = [4, 5, 3, 4, 3]$. At $t = 2$, there are still 5 nodes with 3, 4, 5, 4, 3 neighbors respectively, the vector before temporal smoothness is $M_2 = [2, 2, 5, 4, 3]$. If $cp = 0.5$, EC with DBSCAN remembers $C_2 = [3, 3.5, 4, 4, 3]$. If $MINPTS = 3$, the first and second node are still core nodes after temporal smoothness, which is different from original clustering result.[36]

EC with DBSCAN has a shortcoming that the number of core nodes across time should be the same; it's nearly impossible in the real application.

3.4.6 FacetNet

FacetNet is a dynamic networks EC algorithm. It doesn't extend traditional clustering algorithms, but formulates clustering problem as a non-negative matrix factorization problem. For each graph clustering algorithm, the input is a proximity matrix W. FacetNet assume every similarity entry in the matrix is a combined effect due to all m communities. Hence, FaceNet approximate similarity entry $w_{ij} \approx \sum_{k=1}^{m} p_k \cdot p_{k \to i} \cdot p_{k \to j}$, where p_k is the probability that w_{ij} is due to the k-th community, $p_{k \to i}$ and $p_{k \to j}$4 are the probabilities that community k involves node v_i and v_j respectively. Written in a matrix form, $W \approx X \Lambda X^T$, where X is a $n \times m$ non-negative matrix with $x_{ik} = p_{k \to i}$, and

Λ is an $m \times m$ non-negative diagonal matrix with $\lambda_k = p_k$. $X\Lambda$ fully characterize the community structure (clustering result) in the mixture model.[11]

In EC setting, FacetNet is a model-level EC algorithm. It defined SC as KL-divergence between true proximity matrix W and approximated matrix W_a, and also defined the TC as the KL-divergence between the current clustering result from $X_t\Lambda_t$ and clustering result of the last timestep $X_{t-1}\Lambda_{t-1}$. FaceNet tries to minimize the overall cost in formula 3.1 using an iterative algorithm.[11]

3.4.7 Incremental frequent itemset mining

Frequent itemsets is a document clustering algorithm. The algorithm believes documents with similar keyword set are similar and should be clustered into a cluster.

One of the frequent itemsets algorithms works as follows. Each document has a doc-space which includes keywords for the document. Some keywords are frequently combined in several documents; they form frequent itemsets. For a batch of the document, there is a list of frequent itemsets. Each frequent itemset has a doc-list which includes all documents with these keywords. As a result, documents are clustered automatically by frequent itemsets. However, a document may be allocated to several clusters (i.e., the clusters may be overlapped.) To reduce the overlaps between clusters, a document can only be allocated to a cluster with the highest following score. (The score is proposed by Yu, et al[32])[2]

$$Score(d, T) = \sum_{t \in T} (d \times t)/length(T) \tag{3.6}$$

where $d \times t$ denotes the tf-idf score of each word t in T , the frequent itemset in document d.

Now Frequent itemsets mining is extended to EC setting. After clustering a batch of the document, frequent itemsets are remembered. The remembered frequent itemsets are the initialization of next frequent itemsets. It's called incremental frequent itemset mining.

Incremental frequent itemset mining defines Snapshot Quality (be inversely proportional to SC) and HC. SQ is clustering quality of current documents by general measures like F-Score[37], NMI[25], etc. To define HC, the algorithm first matches every clustering at t_i to the most similar clustering at t_{i+1}. For each pair of clusters, the documents included only in one of them are selected into a Set $S'$to measure the difference. $\sum_{s \in S'} Score(s, T)$ measures difference of a pair of clusters, so $HC = \sum_{\forall clusterpair} \sum_{s \in S'} Score(s, T)$. Since HC compares clustering results of two timesteps, incremental frequent itemset mining is a model-level EC algorithm. What's more, incremental frequent itemset mining restricts cluster number because of match of different timestep clusters. [19]

3.4.8 DYN-MOGA

DYN-MOGA (DYNamic MultiObjective Genetic Algorithms) is an EC algorithm designed for Dynamic Networks. It uses multi-objective optimization methods to minimize SC and TC simultaneously, which is similar to EKM-MOEA/D. [7]

DYN-MOGA defines SC as the community score introduced in [16] which effectively maximize the number of connections inside each community and minimizes the number of links between the communities. TC is defined as NMI of two clustering results; it measures the similarity of community structures from the last timestep to now. Due to its TC, the algorithm is a model-level clustering algorithm. [7]

DYN-MOGA formulates EC problem as follows. The population Ω means a pool of possible clustering result $CR^t_m, m = 1, 2, ...$for current network. The objective of EC is to minimize both $SC(CR^t)$ and $TC(CR^t)$ simultaneously by selecting nondominated individuals from a population. Nondominated individuals perform no worse than all other individuals when minimizing SC and/or TC. In other words, a solution performs well on at least one constraint. Hence, there is not one unique solution to the problem, but a set of solutions are found. These solutions are called *Pareto-optimal*. [7]

DYN-MOGA represent communities based on locus-based adjacency representation. Every node in network call gene and every gene has its allele value in the range of $\{1, ..., No.node\}$. The allele value j of a node i means i and j are linked to the network. Linked nodes (directly or indirectly) form a community. Therefore, locus-based adjacency representation can represent different community structures in a network. [7]

Based on above community structure, the algorithm works as follows. When $t = 1$, optimize SC only. From $t = 2$, a population of randomly generated individuals is created, and individuals will be ranked according to Pareto dominance. Then individuals with the lower rank will be selected and changed to another individual by applying variation operators. (e.g., uniform crossover, mutation.) The new individuals will be added to the population and ranked with old individuals. The variation and rank step will iterate until getting a satisfied set of nondominated individuals. Finally, modularity is used to select the highest score community structure. [7]

3.4.9 On-line Evolutionary Exponential Family Mixture - HMD

The exponential family is a probability distribution set which can be uniquely expressed using Bregman divergence. Exponential Family Mixtures (EFMs) is a mixture of several exponential families. e.g., GMM, multinomial mixture model (MMM). [35]

The clustering problem can be formulated as an EFM estimation problem. The dataset to be clustered is an unknown true distribution, clustering via EFM uses an EFM to approximate the unknown true distribution. The approximation procedure is to minimize variational convex upper bound of the KL-divergence between unknown true distribution and EFM model by EM procedure. Many clustering algorithms are EFM estimation. (e.g., k-means and GMM clustering.)[35]

Online Evolutionary EFM provides a data-level algorithm - historical model dependent (HMD), and a model-level algorithm - historical data dependent (HDD). For HMD, it defines SC as the KL-divergence between unknown current data distribution and current estimated EFM model and defines TC as Earth Mover Distance (EMD) between current estimated EFM model and last estimated EFM model. The objective is to minimize the overall cost like formula 3.1. It minimizes the variational convex upper bound of loss

function by w-step, q-step or ≡-step. Evolutionary k-means introduced in 3.4.1 is a special case of HMD with approximate computing of dead in w-step. [35]

The data-level model will be introduced in section 3.5.5. [35]

Most of EC algorithms track the same object from time to time, so data to be clustered at different time epochs should be identical. Online Evolutionary EFM is different; it can be applied to the scenario that data of different epochs are arbitrary I.I.D. samples from different underlying distributions. Hence, it obviously deals with the variation of data size. What's more, it doesn't limit the cluster number. Besides, using different specific exponential families, both HMD and HDD can produce a large family of evolutionary clustering algorithms. [35]

3.5 Data-level EC algorithms

3.5.1 A Particle-and-Density Based Evolutionary Clustering Method

The algorithm is designed for dynamic network clustering (time-evolved network). It's a data-level model, so it remembers similarity matrix at each timestep. It uses cost embedding method to smooth similarity between nodes first then clustering. For a pair of node v and w, SC is defined as one-dimensional Euclidean distance measure between $d_O(v,w)$ and $d_t(v,w)$, where d_O means original distance and d_t means distance after temporal smoothness. TC is defined as $d_{t-1}(v,w)$ and $d_t(v,w)$. The cost embedding method minimizes overall cost like formula 3.1, and the solution asks to smooth similarity of node pair as follows. [10]

$$d_t(v,w) = (1 - CP) \times d_{t-1}(v,w) + CP \times d_O(v,w) \tag{3.7}$$

After cost embedding, clustering is applied to adjusted similarity matrix. Cost embedding has two advantages, independent of both the similarity measure and the clustering algorithm. Although cost embedding doesn't limit clustering algorithm, DBSCAN (density-based clustering algorithm) is recommended, because of its advantages of an arbitrary number of clusters, handling noises, and being fast. As stated in section 3.4.5, $MINPTS$ and EPS are parameters needed to be set. Clustering result is sensitive to EPS but isn't much sensitive to $MINPTS$, so the algorithm also determines EPS automatically by maximizing modularity. Since modularity maximizing is NP-complete, so a heuristic algorithm is used. [10]

The algorithm can easily identify the stage of each community among the three stages: evolving, forming, and dissolving by using a community structure - nano-communities. If one node v at last timestep and another node w at current timestep (no need the same node) have a non-zero score for a similarity function, they form a nano-community $NC(v,w)$. v and w have a link which is different from an edge between two nodes at the same timestep. The whole dynamic network is modeled as a collection of lots of nano-communities. For a dynamic network, if links among the subset of nano-communities are dense, these nodes form a cluster. A cluster changes across time can be represented by (quasi) l-clique-by-clique(shortly, l-KK). Figure 3.3 shows a 4-kk example. Nodes in

an oval are in the same community at the same timestep, and the lines are links for nano-communities. From left to right, there are four timesteps. If the number of nodes in a community change, it's shown on l-kk. Hence, community's evolving, forming, and dissolving can be detected by l-kk. The detailed method to detect community change won't be explained here. If interested, turn to section 5 of [10].

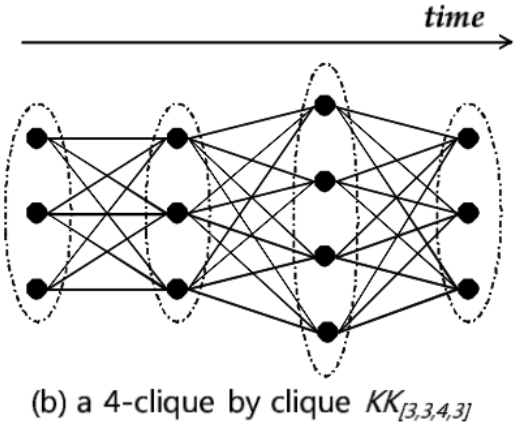

Figure 3.3: Example of 4-clique-by-clique(4-KK)[10]

In fact, although the algorithm is designed for dynamic network clustering, the similarity adjustment then clustering steps are suitable for all clustering problems.

3.5.2 AFFECT

AFFECT is similar to A Particle-and-Density Based Evolutionary Clustering Method. It's a data-level EC algorithm. However, AFFECT is more general, it's suitable for both data and graph clustering problems, and can be extended to all traditional clustering algorithms with similarity matrix as input. What's more, CP is automatically optimized by AFFECT adaptively, so that temporally smoothed similarity matrix approximates truth similarity matrix. [29]

AFFECT assumes cluster change is a mixed result of the long-term drift of clusters and noise. Hence, in a data-level setting, the truly proximity matrix (represent long-term drift of clusters) assumes to be a linear combination of a true proximity matrix and a zero-mean noise matrix as [29]

$$W^t = \Psi^t + N^t, t = 0, 1, 2, ... \tag{3.8}$$

where W_t is a matrix calculated from the dataset, Ψ^t is the unknown true proximity matrix which is the goal for the algorithm to accurately estimate at each time step. N^t reflects short-term variations due to noise, N^t at different time step are mutually independent.[29]

A estimate of Ψ^t is smoothed proximity matrix:

$$\hat{\Psi}^t = CP^t \times \Psi^{t-1} + (1 - CP^t) \times W^t \tag{3.9}$$

Shrinkage estimators are used for estimating an optimal CP at each timestep, so that estimated $\hat{\Psi}^t$ has a minimized mean squared error (MSE) with the true proximities Ψ^t. After estimating true proximity matrix, any clustering algorithms with proximity matrix as input can follow directly. [29]

Hence, the procedure of AFFECT repeats:

1. Adaptively estimates the optimal smoothing parameter CP using shrinkage estimation;
2. Accurately tracking the time-varying proximities between objects: $\hat{\Psi}^t = CP^t \times \Psi^{t-1} + (1 - CP^t) \times W^t$;
3. Followed by static clustering.

Actually, $\hat{\Psi}^t$ incorporates proximities not only from time t-1, but potentially from all previous timesteps because $\hat{\Psi}^{t-1}$ covers a longer history. $\hat{\Psi}^t$ can be unfolded like[29]

$$\begin{aligned}\hat{\Psi}^t =&(1 - CP^t)W^t + CP^t(1 - CP^{t-1})W^{t-1} + CP^tCP^{t-1}(1 - CP^{t-2})W^{t-2} + ... \\ &+ CP^tCP^{t-1}...CP^2(1 - CP^1)W^1 + CP^tCP^{t-1}...CP^2CP^1W^0\end{aligned} \tag{3.10}$$

The details of shrinkage estimators aren't covered in this section. Since AFFECT are implemented in the project, the details will be covered in implementation section.[29]

3.5.3 FIR

FIR (Finite Impulse Response) is a data-level EC algorithm. Although it doesn't follow general framework of EC by minimizing SC and TC, it solves the time-evolve clustering problem similar to AFFECT. [17]

As stated in section 3.5.2, the smoothed proximity matrix can be unfolded as formula 3.10, and $CP^t, t = 0, 1, 2...$ are automatically optimized by shrinkage estimators. FIR also formulate smoothed proximity matrix as [17]

$$\hat{\Psi}^t = b^tW^t + b^{t-1}W^{t-1} + b^{t-2}W^{t-2} + ... + b^1W^1 + b^0W^0, \tag{3.11}$$

But the coefficients $b^t, t = 0, 1, 2...$ don't have as good optimizer as shrinkage estimators. The paper considers to set all coefficients to 1 using the flat filter, and also tries the linear decreasing filter and quadratic decreasing filters. No one coefficient setting outperformed the others during all time periods. Finally, they use Adaptive History Filtering which gives more weight to the time periods when the clusters are far apart. Adaptive History Filtering separates overlapped clusters well, but can't detect a natural merge of two clusters. As a result, cluster number should be fixed. What's more, even if AFFECT uses a long history, but the older matrix will get smaller coefficients (coefficients for older matrix will be smaller and smaller by repeatedly multiplying a coefficient less than 1.) FIR has a risk to give too large weight to long ago history. [17]

3.5.4 Evolutionary Spectral Clustering - PCQ

PCQ is a data-level algorithm, SC and TC are defined similarly. SC is NA (NC) of current clustering result and current data. TC is NA (NC) of current clustering result and last timestep data. In PCQ, the only step different from Spectral clustering is to calculate the first k eigenvectors of $CP * W_{t-1} + (1 - CP) * W_t$ for NA and $CP * D_{t-1}^{-\frac{1}{2}} W_{t-1} D_{t-1}^{-\frac{1}{2}} + (1 - CP) * D_t^{-\frac{1}{2}} W_t D_t^{-\frac{1}{2}}$ for NC.[4]

3.5.5 On-line Evolutionary Exponential Family Mixture - HDD

The concepts of Online Evolutionary EFM and model-level algorithm (HMD) are already introduced in section 3.4.9. The data-level algorithm historical data dependent (HDD) is going to be introduced here. HDD defines SC the same as HMD and defines TC as KL-divergence of last unknown data distribution and current estimated EFM model. The objective is to minimize the overall cost like formula 3.1. It minimizes the variational convex upper bound of loss function by EM procedure. [35]

3.6 EC Algorithms Comparison

After introducing algorithms one by one in section 3.4 and in section 3.5, algorithms are compared by specified criteria here. The comparison of model-level algorithms are listed in table 3.1 and table 3.2. The comparison of data-level algorithms are listed in table 3.3. The comparison for each criteria are listed as follows.

- **Application Scenario:** If an algorithm is suitable for graph clustering, it must be suitable for data clustering. It's because data points can be transformed to proximity matrix. However, some algorithms are only suitable for data clustering problem. They process data points one-by-one, but graph can't be transformed to observations easily.
- **Adding/Deleting nodes:** Algorithms which need to process proximity matrix (especially algorithms designed for dynamic networks) request points should be identical across time because matrix addition is always executed. These algorithms can deal with adding/deleting a small part of points. They add an empty row and an empty column in the last matrix if a new point is added now. They remove related rows and columns if some old points are deleted. As a whole, these algorithms request most of the points should be identical across time. On the contrary, algorithms which process data points, one-by-one accept data points arbitrary I.I.D samples from distributions.
- **Changing cluster number:** Data-level algorithms always accept cluster number change, because they don't need to compare two clustering results. Model-level algorithms need to compare two clustering results. If they use NMI or chi-square to calculate the similarity of two clustering results, clustering number doesn't need to be fixed. If they match the closest cluster across time first, the calculate pair distance, the cluster number has to be fixed because of the one-by-one match.

Table 3.1: Model-level Algorithms Comparison 1

	Application Scenario	**Adding/Deleting nodes**	**Changing cluster number**	**How to compare two clustering results**
Evolutionary k-means	data clustering	yes, arbitrary I.I.D samples from distributions	no	match clusters across time one-by-one
EKM-MOEA/D	data clustering	yes, arbitrary I.I.D samples from distributions	no	match clusters across time one-by-one
Evolutionary Agglomerative Hierarchical Clustering	data&graph clustering	yes, but most of points should be identical across time	yes, by change tree cutting criteria manually.	compare distance of each point pair in clustering result
PCM	data&graph clustering	yes, but most of points should be identical across time	yes, manually setting.	chi-square statistics.
EC with DBSCAN	data&graph clustering	yes, but the core points should be identical across time	yes, automatically detected	difference of neighbour number of core nodes
FacetNet	data&graph clustering	yes, but most of points should be identical across time	yes, automatically detected	KL-divergence of clustering results
Incremental frequent itemset mining	document clustering	yes, arbitrary I.I.D samples from distributions	yes, automatically detected	match clusters across time one-by-one
DYN-MOGA	data&graph clustering	yes, but most of points should be identical across time	yes, automatically detected	NMI of two clustering results
HMD	data clustering	yes, arbitrary I.I.D samples from distributions	yes, manually setting.	Earth Mover Distance (EMD) between two estimated EFM models

Table 3.2: Model-level Algorithms Comparison 2

	Memory	**Optimization Method**	**Extension ability**	**optimize CP automatically**	**Others**
Evolutionary k-means	centroids of last timestep	heuristic	no	no	-
EKM-MOEA/D	centroids of last timestep	multi-objective optimization	no	no	-
Evolutionary Agglomerative Hierarchical Clustering	bottom-up binary tree of last timestep	heuristic	no	no	-
PCM	k-eigenvector matrix of last timestep	relaxed version optimization	no	no	-
EC with DBSCAN	vector of neighbor number of core nodes	the first-order derivative of the cost function equals to zero	no	no	core nodes should be identical across time
FacetNet	dot product of matrix $X\Lambda$	an iterative algorithm	no	no	-
Incremental frequent itemset mining	frequent itemsets of last timestep	heuristic	no	no	-
DYN-MOGA	last community	multi-objective optimization	no	no	-
HMD	last estimated EFM model	minimize the variational convex upper bound of loss function by w-step, q-step or $\equiv$-step	yes	no	-

Table 3.3: Data-level Algorithms Comparison

	A particle-and Density based EC	**AFFECT**	**FIR**	**PCQ**	**HDD**
Application Scenario	data&graph clustering	data&graph clustering	data&graph clustering	data&graph clustering	data clustering
Adding/Deleting nodes	yes, but most of points should be identical across time	yes, but most of points should be identical across time	yes, but most of points should be identical across time	yes, but most of points should be identical across time	yes, arbitrary I.I.D samples from distributions
Changing cluster number	yes, automatically	yes, manually set	yes, manually set	yes, manually set	yes, manually set
Memory	similarity matrix				data distribution
Optimization Method	optimize good parameters by modularity optimization	shrinkage estimation	Adaptive History Filtering	relaxed version optimization	EM procedure
Extension ability	yes, only extends to clustering algorithms with similarity matrix as input	yes, only extends to clustering algorithms with similarity matrix as input	yes, only extends to clustering algorithms with similarity matrix as input	no	yes, all EFM model
optimize CP automatically	no	yes	yes	no	no

For both data-level and model-level algorithm, whether an algorithm can detect suitable cluster number automatically is determined by the clustering algorithm it extends. For example, DBSCAN detects cluster number automatically, so EC algorithm extends DBSCAN can do it too.

- **Memory:** Data-level algorithms remember similarity matrix or data distribution, and model-level algorithms remember last clustering result which is different from different algorithms.
- **Optimization Method:** There are three categories of optimization methods. Firstly, only the earliest algorithms just provide heuristics to simply involve current data and last model in updating clustering result. In the following, most of the algorithms optimize the overall cost function in formula 3.1, and get their specific solution by their defined SC and TC. Besides, there are two algorithms EKM-MOEA/D and DYN-MOGA which optimize SC and TC separately using multi-objective optimization methods.
- **Optimize CP automatically:** Except for EKM-MOEA/D and DYN-MOGA, other algorithms minimize overall cost controlled by CP or use overall cost with CP to measure the clustering result. Among all algorithms need a CP, only AFFECT and FIR optimize CP automatically. According to experiments of both paper, AFFECT's optimization is outperformed FIR's.
- **Extension ability:** Model-level algorithms can't extend to other methods because they define TC by their clustering method. On the contrary, data-level algorithms have a better extensibility, because they always adjust data similarity in advance, then follows traditional clustering method. All traditional clustering method with

similarity matrix as input can follow all data-level model. On the other hand, data-level algorithms are too similar to each other.

3.7 Related Work of Evolutionary Clustering

3.7.1 Incremental Clustering

The objective of incremental clustering is to update clustering result quickly when new data arrive. There is two main difference between incremental clustering and evolutionary clustering. The first is illustrated in figure 3.4. For incremental clustering, the objective of is to get one clustering result for all data, while evolutionary clustering will get a sequence of clustering results. (i.e. a clustering result for a batch of data).

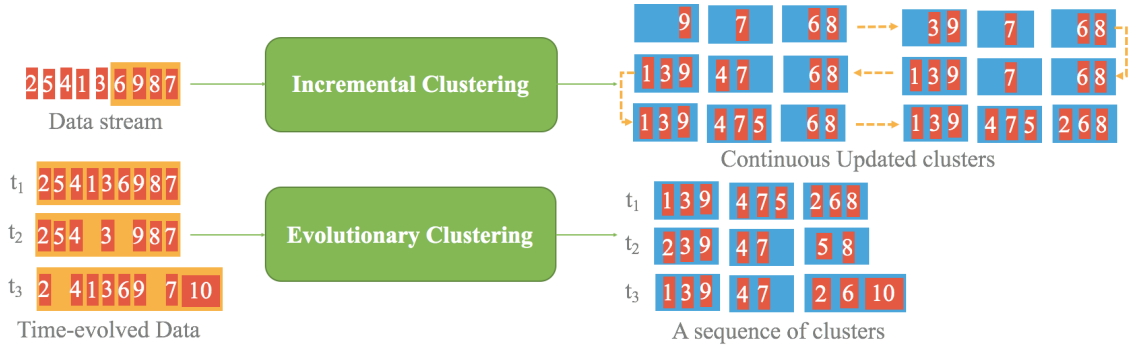

Figure 3.4: Difference between incremental clustering and evolutionary clustering.

Secondly, incremental clustering update clustering results quickly primarily to avoid the cost of storing all historical similarities, the focus of incremental clustering is at low computational cost at the expense of clustering quality. It's an obvious different objective against evolutionary clustering which wants to remember more historical similarities. [29] However, Some incremental clustering algorithms targets similar problem setting as evolutionary clustering, like [15], but their focus is still on low computational cost.

However, incremental clustering is suitable for initialization model at each timestep. For example, in Evolutionary *k*-means, centroids should be initialized at each timestep. If using centroids of the last timestep as initialization, the algorithm will convergence quickly and be more stable. [29]

3.7.2 Clustering data streams

A large amount of data that arrive at high rate make it impractical to store all the data in memory or to scan them multiple times, so data stream clustering tries to cluster all data with one-pass scanning and process data scalability (by using limited memory). Although EC's application scenarios are always a large volume of data, the objective of EC are obviously different from data stream clustering. [29]

3.7.3 Constrained clustering

Constrained clustering algorithms optimize some goodness-of-fit objective subject to a set of constraints. They can be classified as hard constrained and soft constrained. For clustering problems, hard constrained may specify that two objects must or must not be in the same cluster, but soft constrained may bias clustering results based on additional information. Evolutionary clustering can be regarded as a soft constrained when historical data or historical clustering results are be regarded as additional information. However, current constrained clustering algorithms don't have the same objective with EC. E.g., divide the time series into segments that differ significantly from one another. Therefore, constrained clustering is different from EC currently.[29]

3.8 Performance Prediction in Theory

In this section, the performance prediction of EC algorithms will be listed here.

EC algorithms are compared with each other. Since most of the algorithms track identical objects and don't have extension ability, these two criteria aren't taken into consideration. For other criteria,

- Data-level algorithms as a whole should perform better than model-level algorithms because data-level algorithms remember more history and adjust model at a fine granularity level. However, it can not be said that all data-level algorithms outperform all model-level algorithms since other criteria should be considered as well.
- Algorithms accept different cluster number across time should perform better than fixed cluster number algorithms. It's because cluster merge or split is natural. No matter whether the cluster number change automatically or manually, if the determined cluster number is appropriate, the performance should be good.
- Algorithms optimizing the whole problem should perform better than algorithms optimizing of defined SC and TC but not CP. Then any optimization should perform better than heuristics.

In the following, data-level algorithms and model-level algorithms are compared separately, because no experiment or theory supports comparison of all of them.

Data-level algorithms: AFFECT and HDD optimize the whole problem, and AFFECT support cluster number change automatically while HDD manually. Hence, both of them should be the best. PCQ and particle-and-density based EC select CP manually, they are on the second level. Although FIR optimizes the whole problem, their optimization performs badly in experiments because considering too more historical information. Hence, it should be the worst. In summary, the performance rank should be: (The larger, the better.)

AFFECT, HDD > particle-and-density based EC, PCQ > FIR

Model-level algorithms: HMD, DYN-MOGA optimize the whole problem and accept a change of cluster number, they should perform best. PCM manually sets CP and

take a change of cluster number; it should perform worse than HMD and DYN-MOGA. EKM-MOEA/D optimizes the whole problem, but restricts the cluster number; it should perform worse than PCM. Evolutionary Agglomerative Hierarchical Clustering and Incremental frequent itemset mining are heuristics accept a change of cluster number, and Evolutionary *k*-means is heuristic using fixed cluster number. Although EC with DBSCAN doesn't restrict clustering number, it limits core nodes number which is unrealistic in the real application. In summary, the performance rank should be: (The larger, the better.)

HMD, DYN-MOGA > PCM > EKM-MOEA/D
> EC with Agglomerative Hierarchical, Incremental frequent itemset mining
> Evolutionary *k*-means > EC with DBSCAN

Chapter 4

Building Evolutionary Clustering Algorithms on Spark

In this section, first two sub-problems of the second problem mentioned in section 1.2 will be solved. Section 4.1 selects suitable EC algorithms for Spark, and section 4.2, 4.3, and 4.4 explain how to implement four algorithms properly on Spark. It should be noted that the implementations process batched dataset rather than data stream.

4.1 Selecting algorithms

In this project, four evolutionary clustering (EC) algorithms are built on Spark. Since EC algorithms are categorized as data-level and model-level algorithms, two of each category are selected to be implemented on Spark. The algorithms are determined based on 1) whether the EC algorithm has a wide application; 2) whether the EC algorithm is suitable for implementing on Spark.

The first determined algorithm is AFFECT. The reasons are: 1) If AFFECT is implemented, another clustering algorithm is easy to be extended based on AFFECT. Hence, AFFECT has a wide application; 2) AFFECT is designed to track the true drift of clustering. (i.e., it optimizes CP.) Hence, AFFECT should perform better compared to other algorithms. 3) Similarity calculation & CP optimization can be parallelized well.

In theory, all clustering algorithms with similarity matrix as input can extend AFFECT directly. However, clustering algorithms which are easy to be parallelized are suitable for extending AFFECT on Spark. Agglomerative Hierarchical Clustering merges binary trees continuously, which doesn't parallelize well. Density-based clustering algorithms, like DBSCAN, can parallelize well. However, if the input is similarity matrix, current distributed DBSCAN algorithm doesn't work. Spectral clustering is easy to be parallelized and obviously accepts matrix input. k-means processes observations piece-by-piece so that it can parallelize well. However, it can't calculate centroid-point distance if the input is similarity matrix. Fortunately, paper [29] provides a version of k-means which uses similarity matrix as input. Therefore, k-means and spectral clustering are selected to extend AFFECT according to paper [29].

After selecting two data-level algorithms, two model-level algorithms are needed. Since *k*-means and spectral clustering parallelize well and they are popular clustering algorithms, Evolutionary *k*-means in paper [3] and PCM in paper [4] are going to implemented.

In the following, implementation of four algorithms will be introduced one-by-one. We start from theory to program design details. Spark program design principles are considered when designing programs.

4.2 Evolutionary *k*-means

4.2.1 Theory

In chapter 3, Evolutionary *k*-means has been briefly introduced. More details about Evolutionary *k*-means will be given here.

The algorithm requests the data point as unit vectors in Euclidean Space and works only upon cosine similarity. Evolutionary *k*-means also starts from centroids initialization, but traditional initialization methods are only used for batch 1 data. From batch 2 data, initialized centroids are centroids from the last timestep. (Like incremental clustering) After initialization, iterate following steps until certain iteration or getting a satisfied result.

1. Allocate data points to the closest centroid;
2. pairing each centroid at t to its nearest peer at t-1 by a function f;
3. update each centroid by considering both expectation of surrounding points and the closest centroid of last timestep according to

$$c_j^t \leftarrow \gamma(1-CP)E^t_{X\in closest(j)}(x) + CP(1-\gamma)c^{t-1}_{f(j)}, \tag{4.1}$$

   where $E^t_{X\in closest(j)}(x)$ is mean of current surrounding point of centroid j; $c^{t-1}_{f(j)}$ and c_j^t are pairing up centroids by f; $\gamma = \frac{n_j^t}{n_j^t + n^{t-1}_{f(j)}}$ measures relative sizes of these two clusters; $n_j^t = |closest(j)|$ be the number of points belonging to cluster j at time t. In addition, CP measures how much to remember;
4. Normalize centroids.

4.2.2 Implementation

The pseudocode of implemented algorithm is shown in Algorithm 1. In the following, last centroids mean centroids of the last batch of data, and current centroids mean centroids of the current batch of data. Line 3 to 10 makes certain observations in each batch have the same dimension. Line 11 normalizes each observation using Spark built-in normalization transformation. Line 12 caches normalized dataset because it will be transformed for many times. Line 13 to 18 initializes last and current centroids. For

batch one, spark built-in KMeans (run one iteration) is used to initialize current centroids; the last centroids should be the same as current centroids. For the others batches, current centroids are initialized as last centroids, which is similar to incremental clustering. Line 19 counts surrounding point number of each last centroid and return an array of tuple $(lastCentroid, count)$. Line 21 counts surrounding point number and calculates surrounding points mean of each current centroid, it returns an array of triple $(currentCentroid, count, mean)$. Line 22 matches each last centroid to one current centroid using a greedy algorithm - algorithm 14. Line 23-24 updates each current centroid according to formula 4.1. Line 26 and 27 calculates snapshot quality and historical cost according to formula 3.2 and formula 3.3 respectively.

Algorithm 14 is a greedy algorithm which matches each last centroid to the nearest current centroid. Line 2 initializes an array of tuple $(lastCentroid, currentCentroid)$. Line 4 finds the closest current centroid for each last centroid, and record distances for the closest pairs. Line 5 sort the centroid pairs by distances from small to large. Line 6 to line 12 traverses each centroid pair. If both current and last centroids are still in mutable sets, match them, and delete them from $lastModel$ and $currentModel$. If at least one of current and last centroids have been deleted from mutable sets, ignore the pair. For remaining centroids in $lastModel$ and $currentModel$, repeat line 3 to10, until both of them are empty.

In algorithm 1, parallelization is carried out at the normalizing dataset, counts surrounding point number of each last centroid, and counts surrounding point number and calculate surrounding points mean of each current centroid. The first is built-in Spark transformation, so the left two are discussed here.

Algorithm 3 calculates number of surrounding point for each last centroid. Line 2 store centroids into Spark Matrix. Line 3 let the matrix multiply each point, which is the same as calculating the dot product of the point with each centroid. Then a vector with dot product value of each centroid is got, the vector index with the largest value is returned as the closest centroid index. Line 3 returns an RDD with a centroid index in each Row. Line 4 maps an RDD with centroid indexes to a tuple of $(centroid, 1)$ then reduce by key, so that Line 4 get an RDD with a tuple $(centroid, count)$ in each Row. Algorithm 4 works similar to Algorithm 3, so it won't be explained more.

In the program, no broadcast variable and accumulator is used.

4.3 AFFECT

The principle and procedure of AFFECT have been introduced in section 3.5.2. In this section, implementation will be explained in details. In the following, implementation of AFFECT, especially CP optimization, will be discussed first. Then, implementation of clustering algorithms followed by AFFECT will be explained.

4.3.1 Implementing AFFECT

Pseudocode of AFFECT is shown in Algorithm 5. The algorithm inputs data in the format $(index : value_1, value_2, ...)$. The user can set AFFECT iteration number $evoIter$ and number of cluster k manually. In the thesis, k-means and spectral clustering are extended to AFFECT, and both of them needs a k value, so a k is inputted here. If DBSCAN is extended to AFFECT, k is of unneeded.

Line 2 saves all data index, and line 3 broadcast it. Line 4 maps each observation into a $(key, value)$ pair (i.e. data with index). In line 5, data with index Cartesians itself, so that $n \times n$ pair of observations are generated. (like $((key_1, value_1), (key_2, value_2))$.) Line 6 map each pair to a $MatrixEntry$, and line 7 creates a similarity matrix for the RDD of MatrixEntry. The similarity measures are different according to clustering algorithms. Line 8-13 initializes last similarity matrix and clustering result if it's the first batch of data.

Last similarity matrix should be adjusted before using. It's because some objects appear at last timestep may not generate new data now, and some new objects generate data at this timestep but not previously. AFFECT deal with the problem by adding/deleting data simply. If a new object is added, the line of the object in last similarity matrix will be the same as this timestep. If an old object disappears, the line of the object in last similarity matrix will be deleted directly. Therefore, last and current similarity matrix will be in the same shape. Line 14 adds or deletes the rows and columns as stated above. Line 15 catches lastMatrix in memory because it can't be recalculated if it's discarded.

Line 16 to 22 iterates the procedure introduced in section 3.5.2. CP is calculated on line 17. Each value of current matrix multiplies $(1 - CP)$, then transforms to a BlockMatrix in line 18. Similarly, each value of the last matrix multiplies CP, then transforms to a BlockMatrix in line 19. Line 20 adds current and last matrices, and get the evolutionary matrix. Line 21 does the static clustering with the evolutionary matrix.

After several iterations, TC and SC are calculated on line 23 and 25. Line 24 unpersist the last matrix, just in case to seize too much memory. Then evolutionary matrix will be stored as the last matrix in line 26 to adjust proximity matrix of next batch. Line 27 stores final clustering result which will be the model initialization of next batch.

4.3.2 Optimizing CP

In this section, CP optimization will be explained in details from theory to implementation.

Let's recall that the true proximity matrix at each timestep can be approximated as W^t and $\hat{\Psi}^{t-1}$)

$$\hat{\Psi}^t = cp^t \hat{\Psi}^{t-1} + (1 - cp^t) W^t, \ where \ \hat{\Psi}^0 = W^0 \tag{4.2}$$

Since $\hat{\Psi}^{t-1}$ has been temporal smoothed at last timestep, $\hat{\Psi}^{t-1}$ has a low variance. The drift of clustering causes it a high bias from $\hat{\Psi}^t$. W^t has a low bias from $\hat{\Psi}^t$ but a high variance because of noise. Hence, optimizing cp has become a bias-variance trade-off

problem. AFFECT calculates an optimized cp at each timestep by minimizing the MSE (squared Frobenius norm) between $\hat{\Psi}^t$ and Ψ^t. The optimized cp is according to the equation

$$(cp^t)^* = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n} var(n_{ij}^t)}{\sum_{i=1}^{n}\sum_{j=1}^{n}\{(\hat{\psi}_{ij}^{t-1} - \psi_{ij}^t)^2 + var(n_{ij}^t)\}}. \tag{4.3}$$

Since both Ψ^t and $var(N^t)$ are unknown, AFFECT uses the spatial sample mean and variance to replace ψ_{ij}^t and $var(n_{ij}^t)$. AFFECT assumes a block structure (Dynamic Gaussian mixture model (GMM) and dot product similarity matrix satisfy the assumption.)

Based on the assumption, ψ_{ij}^t and $var(N_{ij}^t)$ can be estimated based on the following equation.

1. For two distinct objects i and j both in cluster c,

$$\hat{E}[w_{ij}^t] = \frac{1}{|c|(|c|-1)}\sum_{l\in c}\sum_{m\in c, m\neq l} w_{lm}^t \tag{4.4}$$

$$\hat{var}[w_{ij}^t] = \frac{1}{|c|(|c|-1)-1}\sum_{l\in c}\sum_{m\in c, m\neq l}(w_{lm}^t - \hat{E}[w_{ij}^t])^2 \tag{4.5}$$

2. For the same object i,

$$\hat{E}[w_{ii}^t] = \frac{1}{|c|}\sum_{l\in c} w_{ll}^t \tag{4.6}$$

$$\hat{var}[w_{ii}^t] = \frac{1}{|c|-1}\sum_{l\in c}(w_{ll}^t - \hat{E}[w_{ii}^t])^2 \tag{4.7}$$

3. For distinct objects i in cluster c and j in cluster d with $c \neq d$,

$$\hat{E}[w_{ij}^t] = \frac{1}{|c||d|}\sum_{l\in c}\sum_{m\in d} w_{lm}^t \tag{4.8}$$

$$\hat{var}[w_{ij}^t] = \frac{1}{|c||d|-1}\sum_{l\in c}\sum_{m\in d}(w_{lm}^t - \hat{E}[w_{ij}^t])^2 \tag{4.9}$$

The pseudocode of calculating CP is shown in Algorithm 6. The algorithm inputs last matrix, current proximity matrix, and current clustering result. According to formula 4.4-4.9, ψ_{ij}^t and $var(N_{ij}^t)$ are estimated based on the block structure in three conditions. If there are k clusters, there are k, k, and $\frac{k(k-1)}{2}$ possible $\hat{E}[w_{ij}^t]$ and $\hat{var}[w_{ij}^t]$ estimations for the above three conditions respectively. Before optimizing CP, ψ_{ij}^t and $var(N_{ij}^t)$ are estimated according to different clusters in advance by line 4-6. Example in Fugure 4.1 illustrates the procedure. In the example, there are 5 observations in 2 clusters. Cluster 1 has observations 1, 7, 20, and Cluster 2 has observations 3, 50. For each condition, matrix is first filtered, then the mean and variance is estimated by traversing

the entries in the filtered matrix. As a result, for the first two condition, an array of triple $(clusterIndex, mean, variance)$ is returned. For the third condition, an array of quadruple $(clusterOneIndex, clusterTwoIndex, mean, variance)$ is returned.

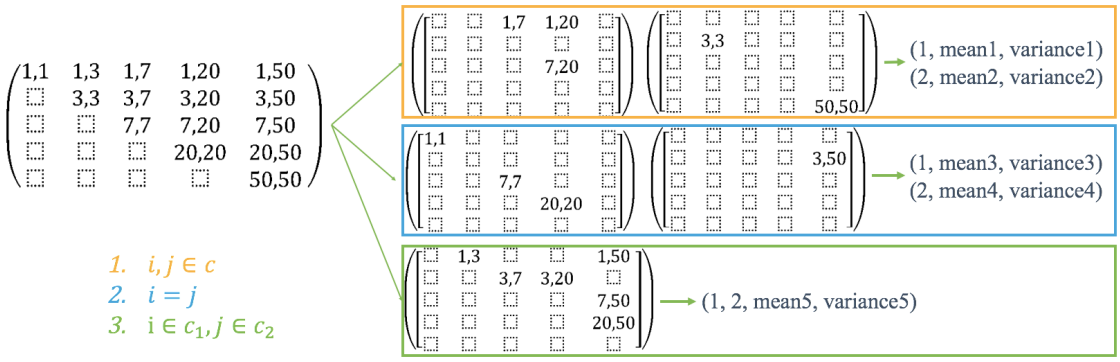

Figure 4.1: Divide Matrix when calculating CP

After estimating $\hat{E}[w_{ij}^t]$ and $\hat{var}[w_{ij}^t]$, CP is estimated by updating numerator and denominator of formula 4.3 continuously. Numerator and denominator are updated in parallel as shown in line 7.

4.3.3 Extend *k*-means for AFFECT

In this section, a *k*-means which extended to AFFECT is introduced. Traditional *k*-means uses coordinates values to represent centroid and point. AFFECT smooth similarity first then do static clustering, and the smoothed similarity can't be transformed back to coordinates values. Therefore, a similarity matrix based centroid-point distance formula is needed. Paper [29] provides such a formula. The squared Euclidean centroid-point distance is

$$||\mathbf{x}_i - \mathbf{m}_c||^2 = w_{ii} - \frac{2\sum_{j\in c} w_{ij}}{|c|} + \frac{\sum_{i,j\in c} w_{jl}}{|c|^2}, \tag{4.10}$$

where w_{ij} is dot product of observations i and j. Using the formula, it's innecessary to update centroid's coordinates. Updating cluster's membership directly is enough. After testing, the formula doesn't actually calculate squared Euclidean centroid-point distance, but if the dataset is suitable for cosine distance measure, the formula's result is proportional to cosine distance. Since centroid doesn't need to be updated, it's fine to use formula 4.10.

The pseudocode of adjusted *k*-means is shown in Algorithm 7. The algorithm inputs a proximity matrix and the previous clustering result. The algorithm repeats calculating minimal centroid-point distance for each point(line 4-6) and updating clustering result (line 8) until the clustering result is fixed.

The main difference between traditional *k*-means and adjusted *k*-means is how to calculate centroid-point distance. Adjusted *k*-means uses formula 4.10. The third term of formula 4.10 is the same for a cluster, so it's calculated and broadcasted in advance (line4-5). Then the centroid-point distance is calculated in parallel.

4.3.4 Extend Spectral Clustering for AFFECT

Spectral clustering has been introduced in section 2.1.3, it is easy to extend AFFECT in Spark. In the project, Spectral clustering optimizing NA is implemented. It has only two steps.

1. Do eigenvector decomposition to Adjusted proximity matrix, and get X which is the first k eigenvectors of W associated with the top-k eigenvalues of the matrix.
2. Do k-means to X.

In the project, eigenvector decomposition is done my Spark built-in SVD decomposition. Since Adjusted proximity matrix is symmetry, so SVD decomposition is the same as eigenvector decomposition. The k-means is also done by Spark built-in k-means.

4.4 PCM

Theory of PCM has been introduced in section 3.4.4. Implementation of PCM is very similar to AFFECT-Spectral (Algorithm 5). The differences are as follows. PCM doesn't need iterated calculation, so line 16 and 22 should be deleted. PCM set CP manually, so line 17 can be deleted. PCM remembers X_{t-1} which is the first k eigenvectors of W_{t-1} associated with the top-k eigenvalues of matrix. In PCM, last Matrix W_{t-1} is calculated by $X_{t-1}X_{t-1}^T$, it should be added in the algorithm.

Chapter 5

Experiments & Results

In this chapter, the third subproblem of problem two in section 1.2 will be answered by experiment. The main purpose of the experiment is to test whether the four implementations are implemented properly on Spark. (i.e., They improve efficiency with the increment of worker number.)

Of course, before testing efficiency, all of them will be tested whether they work correctly on time-evolved data. Labelled synthetic datasets will be used in the experiment. Since the datasets are labeled, (i.e. have ground truth,) NMI introduced in section 2.3 is a unified evaluation measurement for all algorithms. In addition to NMI, TC and SC defined by each algorithm will be used to evaluate each of them respectively.

5.1 DataSet

For each algorithm, Euclidean distance and dot product are implemented. In the experiment, we use dot product as the similarity measurement, so the datasets are also suitable for dot product measurement. The two datasets used in the experiment are synthetic datasets. The first one simulates time-evolved data without noise, and the second one simulates time-evolved data with noise.

5.1.1 Time-evolved Dataset without Noise

Spectral clustering and *k*-means are extended to Evolutionary Clustering Scenario in the thesis. Among them, spectral clustering first does dimensional reduction then clustering. Hence, we use 5-dimensional data. (i.e. 2-dimensional or 3-dimensional data are meaningless for Spectral Clustering.)

In order to simulate time-evolved data, the dataset has 7 batches (timestamps) from batch 0 to batch 6. In each batch of the non-noise dataset, there are 3 well-separated Gaussian distributions. The mean of each distribution is controlled by 5 coefficients (the norm from the original point and 4 angles between mean four axes.) In the dataset, the norm is 100, and the four angles are the same for each distribution for convenience. Angles for each batch and each distribution are listed in table 5.1. In addition, the

variance of the norm is 1, and the variance of each angle is $\frac{\pi}{180}$. Besides, each distribution has 1000 observations in correctness experiments.

Table 5.1: Angles of time-evolved dataset without noise

	distribution 1	distribution 2	distribution 3
batch 0	0.0	$\frac{2.0\pi}{3}$	$\frac{4.0\pi}{3}$
batch 1	$\frac{0.05\pi}{3}$	$\frac{2.05\pi}{3}$	$\frac{4.05\pi}{3}$
batch 2	$\frac{0.7\pi}{3}$	$\frac{2.7\pi}{3}$	$\frac{4.7\pi}{3}$
batch 3	$\frac{1.1\pi}{3}$	$\frac{3.1\pi}{3}$	$\frac{5.1\pi}{3}$
batch 4	$\frac{1.3\pi}{3}$	$\frac{3.3\pi}{3}$	$\frac{5.3\pi}{3}$
batch 5	$\frac{1.7\pi}{3}$	$\frac{3.7\pi}{3}$	$\frac{5.7\pi}{3}$
batch 6	$\frac{1.9\pi}{3}$	$\frac{3.9\pi}{3}$	$\frac{5.9\pi}{3}$

5.1.2 Time-evolved Dataset with Noise

Except for well-separated dataset, another dataset with noise is used in the experiment. The dataset is similar to the well-separated dataset, except for the change of angles in each Gaussian distribution. In the not well-separated dataset, distribution 2 and distribution 3 first move close to each other (from batch 0 to 3), then overlap (batch 4), then leave each other (from batch 5 to 8). 7 batches of data aren't enough to simulate the whole process, so we add 2 more batches. The detailed angles are listed in table 5.2.

Table 5.2: Angles of time-evolved dataset with noise

	distribution 1	distribution 2	distribution 3
batch 0	$\frac{0.0\pi}{3}$	$\frac{2.7\pi}{3}$	$\frac{3.5\pi}{3}$
batch 1	$\frac{0.05\pi}{3}$	$\frac{2.8\pi}{3}$	$\frac{3.4\pi}{3}$
batch 2	$\frac{0.10\pi}{3}$	$\frac{2.9\pi}{3}$	$\frac{3.3\pi}{3}$
batch 3	$\frac{0.15\pi}{3}$	$\frac{3.0\pi}{3}$	$\frac{3.2\pi}{3}$
batch 4	$\frac{0.20\pi}{3}$	$\frac{3.1\pi}{3}$	$\frac{3.1\pi}{3}$
batch 5	$\frac{0.25\pi}{3}$	$\frac{3.2\pi}{3}$	$\frac{3.0\pi}{3}$
batch 6	$\frac{0.30\pi}{3}$	$\frac{3.3\pi}{3}$	$\frac{2.9\pi}{3}$
batch 7	$\frac{0.35\pi}{3}$	$\frac{3.4\pi}{3}$	$\frac{2.8\pi}{3}$
batch 8	$\frac{0.40\pi}{3}$	$\frac{3.5\pi}{3}$	$\frac{2.7\pi}{3}$

5.2 Correctness Experiments

In this section, the correctness of four algorithms is tested. Each algorithm will run both datasets stated in section 5.1 with $CP = 0.0, 0.2, 0.4, 0.6, 0.8, 1.0$ respectively. Then their performance will be tested by NMI and its own defined snapshot quality(SQ)/snapshot cost(SC) and temporal cost(TC)/temporal quality(TQ).

5.2.1 PCM

For PCM, we implement Negated Average Association. Hence, the snapshot quality(SQ) is $NA = Tr(Z_t^T W_t Z_t)$, and a higher snapshot quality means a better performance on current batch data; the temporal quality(TQ) is $NA = Tr(Z_t^T W_{t-1} Z_t)$, and a higher temporal quality means a better performance on previous batch data.

We first run time-evolved dataset without noise. Since the distributions are well-separated, NMIs for all batches of data under all CP settings are 1.0. It means all CP settings can cluster the well-separated dataset perfectly. (i.e. Traditional clustering algorithm can handle time-evolved data without noise.) However, different CP settings have different TQ and SQ performances. The TQ and SQ testing result for non-noise data are shown in figure 5.1. For all CP settings, temporal quality is set to 0 for all batch 0 data, since there is no previous batch before batch 0. When $CP = 1.0$, temporal quality is obviously higher than other CP settings. However, the differences in TQ for other CP settings are too small to be distinguished in figure 5.1. SQ result is similar to the result of TQ, except the SQ when $CP = 1.0$ are much worse than other CP settings. (The detailed experiment result is appended in Appendix C.1 and C.3.)

In order to distinguish the difference of SQ and TQ among different CP settings, adjusted results are shown in figure 5.2. In figure 5.2, the same first digits of TQ/SQ of the same batch data are removed so that the differences are more clear. For example, if TQs for batch 0 data are 123123.2487812, 123123.2432412, 123123.2425313, 123123.2413414 under different CP settings, the same first digits 123123.24 will be removed. As a result, the adjusted TQ are 87812, 32412, 25313, 13414. Since we delete the same first digits of each value, the fluctuation in figure 5.2 doesn't mean something wrong in the implementation. The actual trend of TQ and SQ should be in accordance with the trend in figure 5.1, and figure 5.2 is only intended to show the difference of TQ and SQ under different CP settings. In figure 5.2, it's clear that a higher CP leads to a higher TQ and a lower SQ, which is in line with the theory. Therefore, the implementation of PCM performs correctly on the time-evolved dataset without noise. (The detailed experiment result is appended in Appendix C.2 and C.4.)

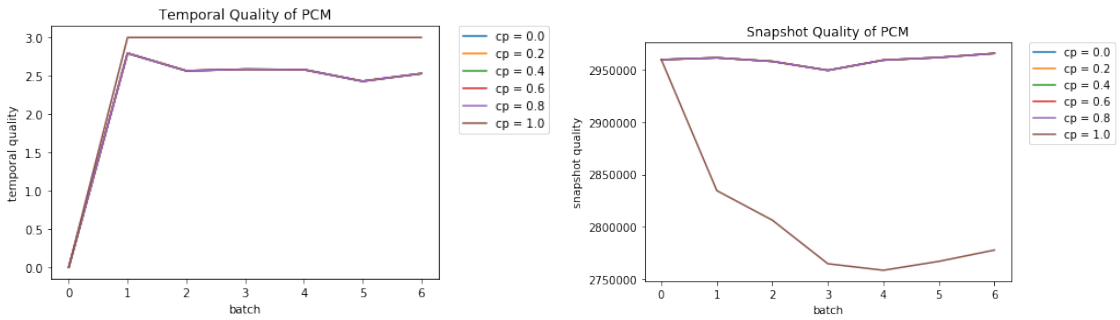

Figure 5.1: Temporal quality & Snapshot quality of PCM

Then we run time-evolved dataset with noise. The NMI result of noise data is shown in figure 5.3. When $CP = 1.0$, the NMI stays 1.0 all the time, since it only remembers the first batch's data. It's a special case, and $CP = 1.0$ isn't always the best CP choice for other cases. From batch 0 to batch 4, two distributions move closer and closer to each other, so whatever CP setting has a worse NMI result as time flows. From batch

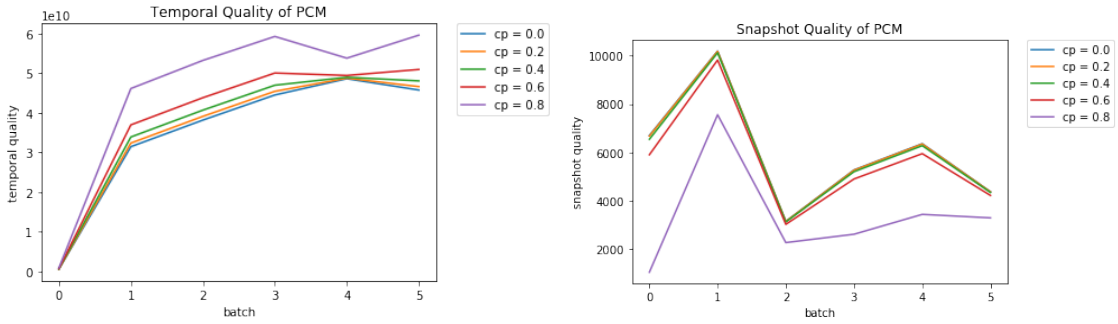

Figure 5.2: Temporal quality & Snapshot quality of PCM (adjusted)

5 to batch 8, two distributions leave each other, so whatever CP setting has a better NMI result as time flows. However, except for $CP = 1.0$, no one CP setting has a better performance than other CP settings all the time. (The detailed experiment result is appended in Appendix C.5.)

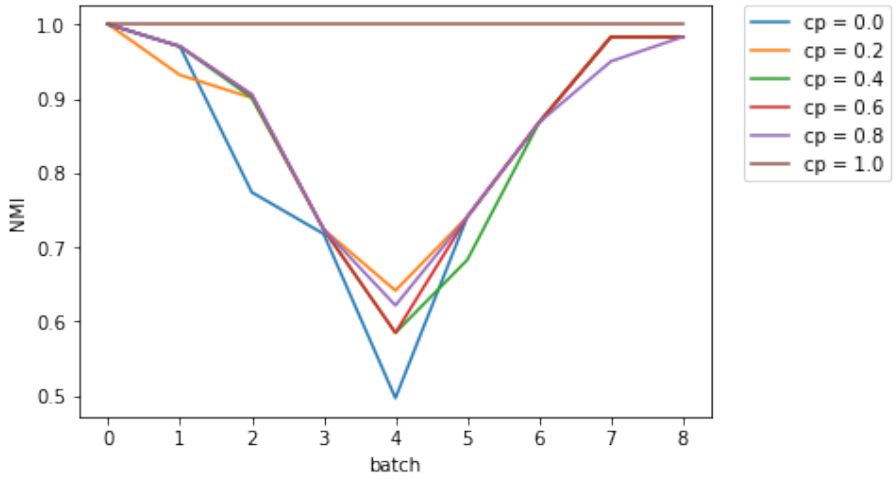

Figure 5.3: NMI result of PCM of noise data

Now, we analyze TQ and SQ testing result for noise data. The results are shown in figure 5.4. The result is similar to the result of non-noise data. When $CP = 1.0$, the TQ is much better than other CP settings, and the SQ is much worse than other CP settings. It's a special case as stated above. The differences of TQ and SQ for other CP settings are vague since the differences are too small. Similarly, the adjusted TQ and SQ testing result are shown in figure 5.5. Similar to figure 5.2, figure 5.5 is intended to show a clear differences of TQ and SQ result under different CP settings. Hence, the fluctuation in figure 5.5 doesn't mean the actual trend of the result. In figure 5.5, it's clear that a larger CP has a better TQ and a worse SQ, which is in line with the theory. (The detailed experiment result is appended in Appendix C.6, C.7, C.8 and C.9.)

Therefore, the implementation of PCM is correct according to both non-noise and noise data experiments.

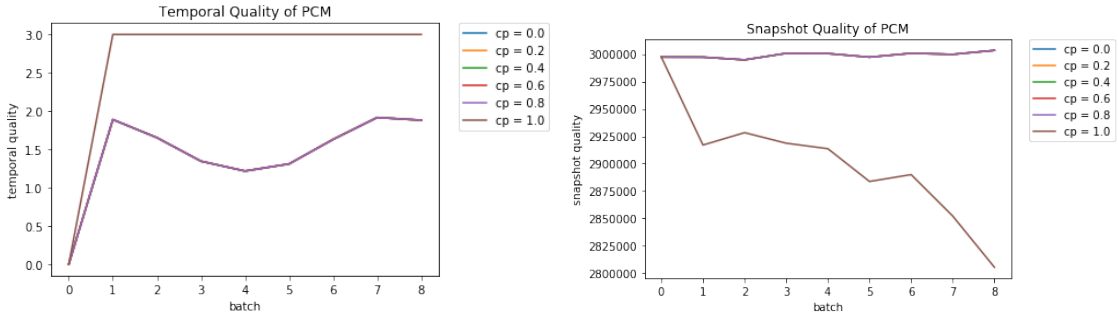

Figure 5.4: Temporal quality & Snapshot quality of PCM for noise data

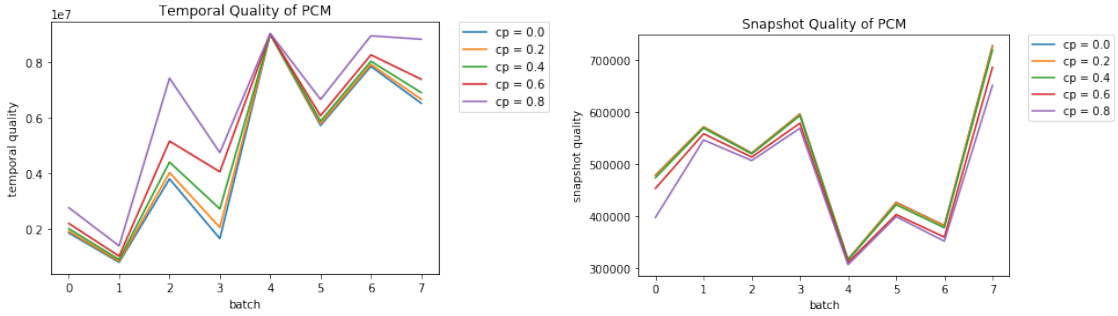

Figure 5.5: Temporal quality & Snapshot quality of PCM for noise data (adjusted)

5.2.2 AFFECT-Spectral clustering

For AFFECT-Spectral clustering, SQ and TQ are the same as PCM. Since the only difference between PCM and AFFECT-Spectral clustering is the optimized CP, we won't show TQ and SQ results under different CP settings for AFFECT-Spectral clustering. We need to show if the optimized CP provides a better clustering result than fixed CP. We first run the well-separated dataset. Since PCM under all CP settings can cluster the well-separated dataset well, optimized CP can do it too. (i.e. NMI for batches of data is 1.0.) Then we run the noise data. The optimized CP and NMI for all batches are listed in the table 5.3. Compared to figure 5.3, the NMI result of optimized CP performs better.

Table 5.3: NMI result of AFFECT-Spectral clustering for noise data

batch	CP	NMI
batch 0	0.0108317	0.9937967512863873
batch 1	0.1073556	0.9937967512863871
batch 2	0.1435724	0.9937967512863873
batch 3	0.2632847	0.8913647197575457
batch 4	0.2971376	0.7355696722827354
batch 5	0.2031391	0.6913295766384434
batch 6	0.1346736	0.8425014917559971
batch 7	0.1035679	0.8941037815430084
batch 8	0.1013767	0.9763153529148262

5.2.3 Evolutionary k-means

SQ and TC of Evolutionary k-means has been introduced in section 3.4.1. They are

$$sq(C, U) = \sum_{x \in U} (1 - min_{c \in C} ||c - x||), \tag{5.1}$$

and

$$tc(C, C') = min_{f:[k] \to [k]} ||c_i - c'_{f(i)}||, \tag{5.2}$$

where C is current clustering result, C' is last clustering result, and U is the dataset.

We first run the dataset without noise. Similar to PCM's result, NMI for all batches of data under all CP settings is 1.0. The TC and SQ testing result are shown in figure 5.6. On the left of figure 5.6, the temporal cost is higher if CP is smaller, which is in accordance with the theory. On the right of figure 5.6, snapshot quality is higher if CP is smaller, which is also in accordance with the theory. (The detailed experiment result is appended in Appendix C.10 and C.11.)

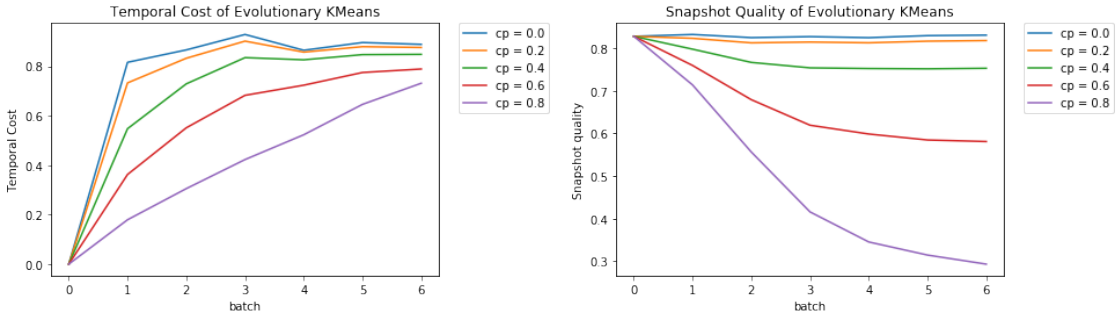

Figure 5.6: Temporal cost& Snapshot quality of Evolutionary k-means for non-noise data

Then we run time-evolved dataset with noise. The NMI result is shown in figure 5.7. Before the overlap of two distritbutions, a larger CP setting should perform better, and a smaller CP setting should perform better after distribution 2 and 3 overlap. In figure 5.7, from batch 0 to batch 3, a larger CP setting actually has a better NMI result; From from batch 6 to batch 8, a smaller CP setting actually has a better NMI result; From batch 4 to batch 5, there is no obvious trend.(The detailed experiment result is appended in Appendix C.12 and C.13.)

The TC and SQ result for noise data are shown in figure 5.8.

For the TC result, the TC from batch 0 to 3 and from batch 6 to 8 are higher than the TC from batch 4 to 5. It's because from batch 4 to 5, two distributions are overlapped, and TC of Evolutionary k-means is the expected absolute value of the difference between matched centroid pairs, so the centroids for two distributions are closer to each other than other timesteps. In addition, when distribution 2 and 3 move close to or leave each other, a larger CP setting has a lower TC, which is in line with the theory. However, when the two distributions are (near) overlapped, there is no clear trend which CP setting is better. If the TC isn't the absolute value, the TC trend from batch 4 to 5 should be the same as the trend from batch 0 to 3 and from batch 6 to 8. (The detailed experiment result is appended in Appendix C.14.)

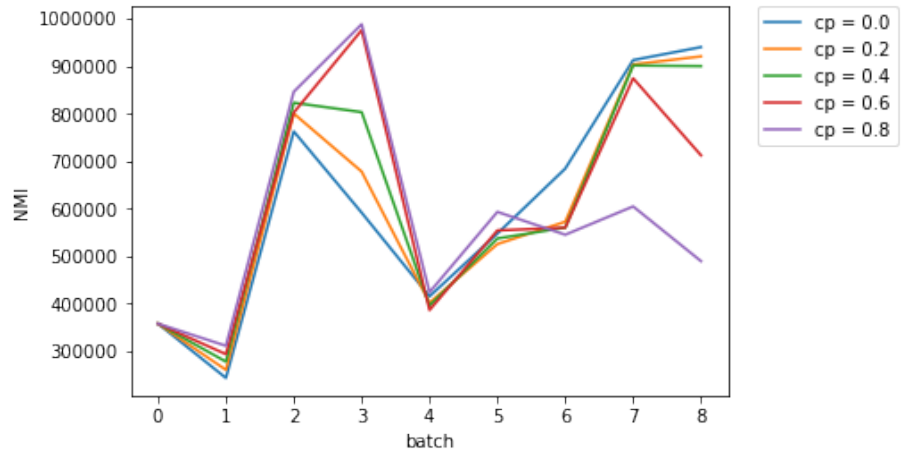

Figure 5.7: NMI result of Evolutionary *k*-means for noise data

For the SQ result, it's clear a lower *CP* setting has a larger SQ, which is in line with the theory. There is one difference between the SQ result of noise data and that of non-noise data. When distribution 2 and 3 overlap, the SQ of noise data under a higher *CP* setting is lower than that under a lower *CP* setting. It's because, when distribution 2 and 3 overlap, the centroids of the two distributions are very close to each other if *CP* is small. Whatever points are clustered into a correct cluster, the point-centroid distance is very small. On the other side, if *CP* is large, both centroids of distribution 2 and 3 are a little away from the center of the overlapped distributions. Hence, whatever points are clustered into a correct cluster, the point-centroid distance is large if *CP* is large. (The detailed experiment result is appended in Appendix C.15.)

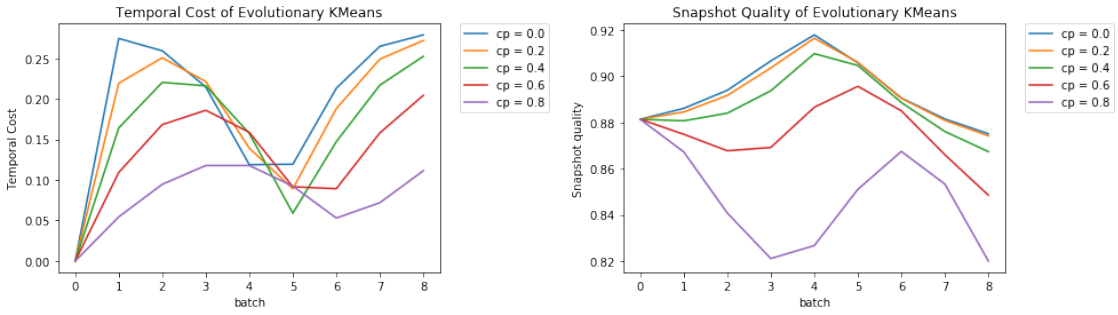

Figure 5.8: Temporal cost& Snapshot quality of Evolutionary *k*-means for noise data

Therefore, according to the NMI and SQ/TC evaluation, the implementation of Evolutionary *k*-means on Spark is correct.

5.2.4 AFFECT-*k*-means

The SC of AFFECT is the sum of centroid-point distance for current dataset calculated by formula 4.10. Similarly, TC of AFFECT is the sum of centroid-point distance for the last dataset calculated by formula 4.10. Since AFFECT-*k*-means uses a *k*-means algorithm with proximity matrix as input, we first show if the algorithm works correctly for both

non-noise and noise data under different *CP* settings. Then we will show whether the optimized *CP* provides a better performance than fixed *CP* settings.

We first run the well-separated dataset. Just like other algorithms, AFFECT-*k*-means clusters the dataset perfectly. (i.e. NMI is 1.0 for all batches of data under all *CP* settings.) The TC and SC results are shown in figure 5.9. The TC/SC for the same batch of data is always the same and doesn't show expected curves.

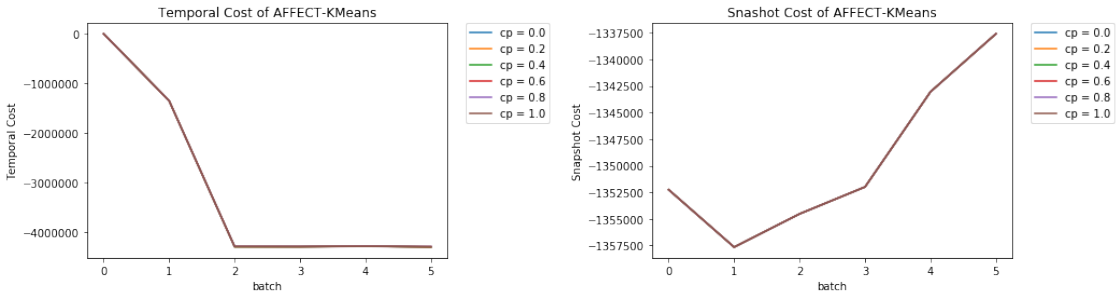

Figure 5.9: Temporal cost& Snapshot cost of AFFECT-*k*-means for non-noise data

It's because of the definition of SC and TC here. Just review the TC/SC formula as follows.

$$||\mathbf{x}_i - \mathbf{m}_c||^2 = w_{ii} - \frac{2\sum_{j\in c} w_{ij}}{|c|} + \frac{\sum_{i,j\in c} w_{jl}}{|c|^2}, \tag{5.3}$$

For a batch of data, the proximity matrix is fixed. Since NMI is 1.0 for the same batch of data under all *CP* settings, so the clustering results are the same. If we use the formula, the centroid-point distances for the same point under different *CP* settings are the same if the clustering result and the proximity matrix are the same. Therefore, in figure 5.9, the TC/SC for the same batch of data is always the same. From a certain perspective, the defined TC/SC are not good evaluation measurements.

Then we run the noise dataset. The NMI result is shown in figure 5.10, and the record is appended in Appendix C.16. The result of AFFECT *k*-means is similar to the result of Evolutionary *k*-means. Before distribution 2 and 3 overlapped, a higher *CP* setting performs better; after distribution 2 and 3 overlapp, a lower *CP* setting performs better gradually.

We are not going to analyze the TC and SC result for AFFECT *k*-means. It's because if the clustering method and proximity matrix are fixed, the TC and SC are fixed. It's not a good measurement.

Finally, we will run the noise data with the optimized *CP* setting, in order to see whether the optimized *CP* setting has a better NMI result. According to the result in table 5.4, AFFECT-*k*-means provides good result before and after distribution 2 and 3 overlapped. When the two distributions are (nearly) overlapped, the optimized *CP* doesn't perform better than other *CP* settings.

Based on the NMI experiment result, AFFECT *k*-means is implemented correctly.

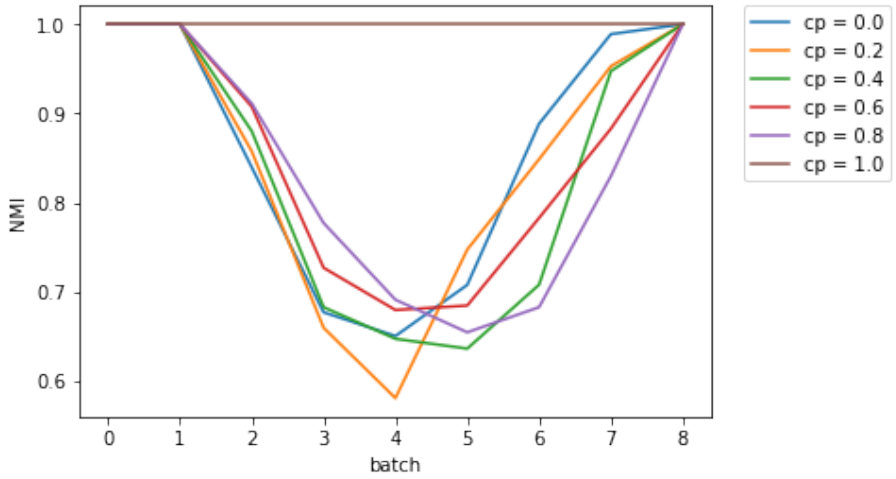

Figure 5.10: NMI result of AFFECT *k*-means for noise data

Table 5.4: NMI result of AFFECT-*k*-means for noise data

batch	*CP*	NMI
batch 0	0.0015256	1.0
batch 1	0.1052625	1.0
batch 2	0.2143257	1.0
batch 3	0.2508956	0.8167114929527717
batch 4	0.2759801	0.647661999743697
batch 5	0.2015299	0.7461175870548299
batch 6	0.1802462	0.7285453014338626
batch 7	0.0914781	0.9701911554558438
batch 8	0.0148692	0.9829929733076637

5.3 Efficiency Experiments

In the following, the efficiency of each algorithm is evaluated. The main objective is to test whether these four algorithms are parallelized well.

The cluster used in the experiment are OpenStack virtual machines provided by SICS Swedish ICT. Each virtual machine has 80GB disk and 8GB RAM, and the system is Ubuntu 16.10. If not mention, the executor memory is 1GB, and executor memory overhead is 0.1GB.

We test the efficiency by using the well-separated dataset. In the experiment, only two parameters change - data size and worker number. With a specific data size, execution time will be recorded for different worker number. After running one program, all workers will be restarted to create the same execution environment for each program. Also, each parameter setting will be run three times, and the average time will be recorded.

5.3.1 Evolutionary *k*-means

For evolutionary *k*-means, the parameters used are $CP = 0.2$, $k = 3$, $iterNum = 10$. The testing result is shown in figure 5.11, and the records are appended in the Appendix C.17. The well-separated dataset has 7 batches, the time we show in figure 5.11 and Appendix C.17 are average time (in second) of running a batch.

According to figure 5.11, if the data size is larger than 1000000, more workers run more efficiently than one worker. What's more, the larges data size we run, the better parallelization performance we get. However, if the data size is small, the memory of one machine is enough, distributive computation spends more time on job allocation, data shuffle, etc..

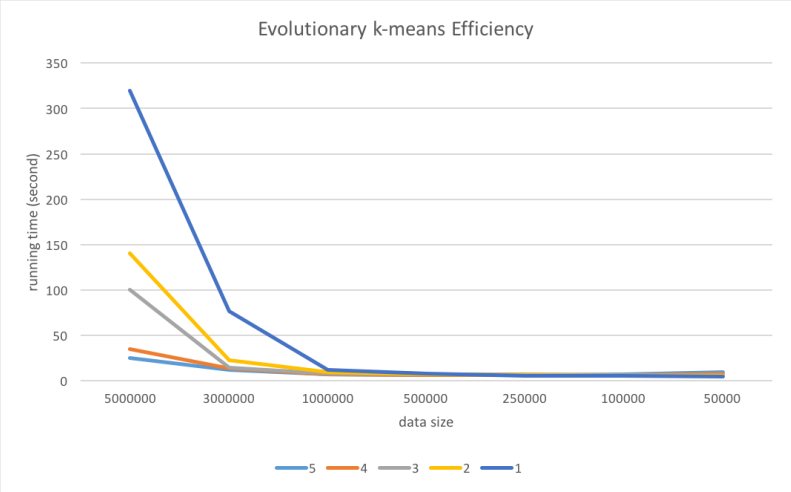

Figure 5.11: Evolutionary *k*-means Efficiency test (completed)

If only small data set testing result is examined, (shown in Figure 5.12) 250000 is the junction point. Therefore, data set with less than 250000 observations don't have to run

the program on clusters. As a whole, the implementation runs in a high efficiency and parallelizes well.

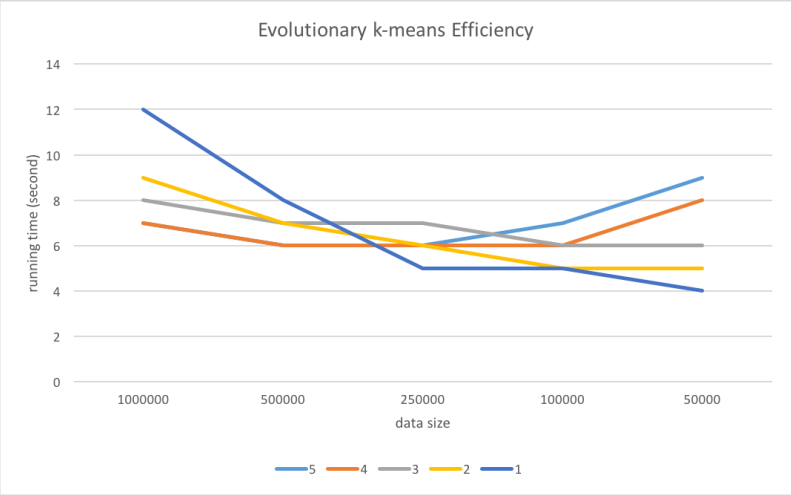

Figure 5.12: Evolutionary *k*-means Efficiency test (small size)

5.3.2 PCM

For PCM, the parameters used are $CP = 0.2$, $k = 3$, $kMeansIter = 10$. The testing result is shown in Figure 5.13, and the experiment records are appended in Appendix C.18. PCM is much slower than Evolutionary *k*-means when running the same size of data, since PCM process entries in proximity matrix rather than observations. If PCM processes 4000 data points, it means 16000000 data points for Evolutionary *k*-means. In addition, PCM uses Spark built-in SVD decomposition which is time-consuming. What's more, Evolutionary *k*-means is easier to parallelize than PCM, so Evolutionary *k*-means is much quicker than PCM.

The result of PCM is similar to the result of Evolutionary *k*-means. For large dataset, with the increment of the worker number, the running time decreases. A larger dataset, a larger difference between the running time of different worker number. It means the implementation parallelize well. 4000 data is the junction point. If the data size is less than 4000, more workers may lead to a longer running time because of job allocation, data shuffle, etc. If the dataset is larger than 4000, it's suitable to run on clusters.

5.3.3 AFFECT-Spectral clustering

For AFFECT Spectral clustering, the parameters used are $k = 3$, $kMeansIter = 10$, $evoIter = 3$.The testing result is shown in Figure 5.14, and the detailed record is appended in Appendix C.19. There are two differences between PCM and AFFECT-Spectral clustering. Firstly, AFFECT-Spec optimizes CP but PCM doesn't. Secondly, AFFECT-Spec runs 3 more iterations than PCM. Therefore, AFFECT-Spec is obviously slower than PCM. According to figure 5.14, AFFECT-Spec is actually slower than PCM. Similar to PCM, if a dataset is small, more workers can't guarantee a faster clustering. If a dataset is large enough, more worker leads to a faster clustering process.

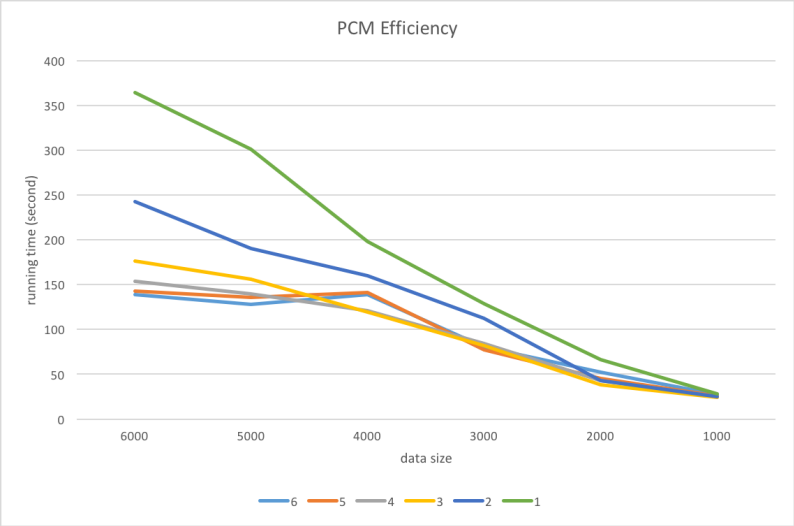

Figure 5.13: PCM Efficiency test

What's more, the larger dataset we run, the more specific difference between the running time of different worker number. The junction point of AFFECT-Spec is 4000. As a whole, AFFECT-Spec is parallelized well but is obviously worse than Evolutionary *k*-means.

The test result of AFFECT-Spectral clustering is similar to the result of AFFECT-*k*-means. With the increment of data size, the running time decreases. Also, a large dataset improves the efficiency more than a small dataset. However, we can't prove whether the improvement will be better if the data size is much larger.

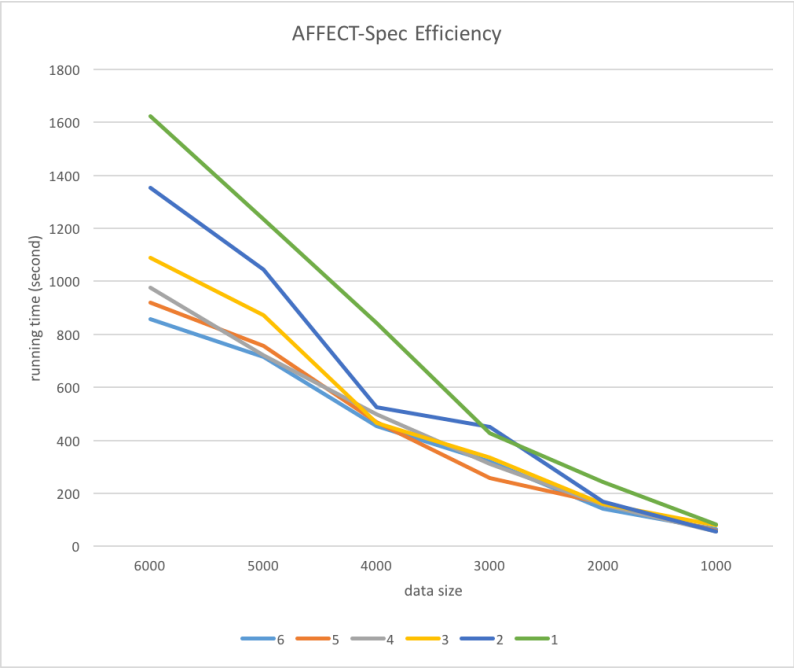

Figure 5.14: AFFECT Spectral Clustering Efficiency

5.3.4 AFFECT-*k*-means

For AFFECT *k*-means, the parameters used are $k = 3$, $evoIter = 3$. The testing result is shown in Figure 5.15, and the experiment records is appended in Appendix C.20. The result of AFFECT-*k*-means is really similar to the result of AFFECT-Spec, and the only difference is the clustering method after calculating evolutionary proximity matrix. AFFECT-*k*-means is also too slow to deal with real world large data. When the dataset is large, more workers perform better; When the dataset is small, more workers my lead to a worse result. The junction point is still 4000. The implementation is parallelized well, but obviously not as good as Evolutionary *k*-means.

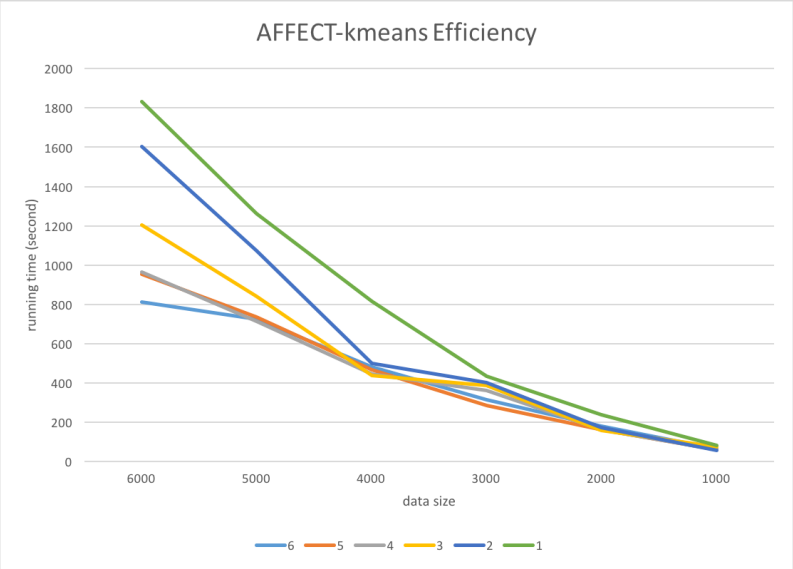

Figure 5.15: AFFECT k-means Efficiency

Chapter 6

Discussion and Conclusion

6.1 Discussion & Future Work

For the implementation part, four algorithms are implemented correctly on Spark. Four of them are parallelized well, so running them on clusters improve computing efficiency. In four of them, Evolutionary *k*-means is the best implementation, because it can deal with large dataset within an acceptable time. For three other algorithms, although they may have a more stable and better clustering result than Evolutionary *k*-means, they run slower than expected. (i.e. They can't deal with large data volume tasks in real life.) Therefore, if new EC algorithms are going to be implemented on Spark, we prefer a model-level algorithm. It's because no proximity matrix has to be stored and compute. For the future work, experiments of real data should be carried out at current four implementations.

For the survey part, the survey provides categorization, comparison, and performance prediction for EC algorithms. They are all useful for the researcher who is interested in EC. In future work, the performance prediction should be verified by experiments. Also, the two papers [31] and [30] which are not covered in the survey can be added to the survey in future work.

6.2 Conclusion

In this thesis, we solve two problems.

Firstly, the theoretical framework of evolutionary clustering is structured by a survey. The survey first introduces the application scenario, the definition, and the history of EC algorithms. Then two categories of EC algorithms - model-level algorithms and data-level algorithms are introduced one-by-one. What's more, each algorithm is compared with each other. Finally, performance prediction of algorithms is given. Algorithms which optimize change parameter or don't use change parameter to control should perform better than algorithms use a fixed change parameter to control. In addition, algorithms accept a change of cluster number should perform better than algorithms

don't accept. However, the performance prediction is in theory. In future work, the prediction should be verified by experiments.

Secondly, the thesis explains the whole process of building evolutionary clustering algorithms on Spark. Four EC algorithms are implemented on Spark in the project. Firstly, we state how to select suitable EC algorithms for Spark. Then program design details are given for each algorithm. The four program design principles are used to design programs. Finally, implementations are verified by correctness and efficiency experiments. All four algorithms are implemented correctly, and all of them are parallelizes well. However, only Evolutionary k-means improves efficiency dramatically when working on clusters. All other algorithms don't have such a dramatical result. In the real application, Evolutionary k-means should be a good choice then three of others.

Bibliography

[1] Amr Ahmed and Eric Xing. "Dynamic non-parametric mixture models and the recurrent chinese restaurant process: with applications to evolutionary clustering". In: *Proceedings of the 2008 SIAM International Conference on Data Mining*. SIAM. 2008, pp. 219–230.

[2] Florian Beil, Martin Ester, and Xiaowei Xu. "Frequent term-based text clustering". In: *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2002, pp. 436–442.

[3] Deepayan Chakrabarti, Ravi Kumar, and Andrew Tomkins. "Evolutionary clustering". In: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2006, pp. 554–560.

[4] Yun Chi et al. "Evolutionary spectral clustering by incorporating temporal smoothness". In: *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2007, pp. 153–162.

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

[6] Martin Ester et al. "A density-based algorithm for discovering clusters in large spatial databases with noise." In:

[7] Francesco Folino and Clara Pizzuti. "A multiobjective and evolutionary clustering method for dynamic networks". In: *Advances in Social Networks Analysis and Mining (ASONAM), 2010 International Conference on*. IEEE. 2010, pp. 256–263.

[8] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*. Vol. 1. Springer series in statistics New York, 2001.

[9] Anne Håkansson. "Portal of research methods and methodologies for research projects and degree projects". In: *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp). 2013, p. 1.

[10] Min-Soo Kim and Jiawei Han. "A particle-and-density based evolutionary clustering method for dynamic networks". In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 622–633.

[11] Yu-Ru Lin et al. "Facetnet: a framework for analyzing communities and their evolutions in dynamic networks". In: *Proceedings of the 17th international conference on World Wide Web*. ACM. 2008, pp. 685–694.

[12] Stuart Lloyd. "Least squares quantization in PCM". In: *IEEE transactions on information theory* 28.2 (1982), pp. 129–137.

[13] Jingjing Ma et al. "Spatio-temporal data evolutionary clustering based on MOEA/D". In: *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*. ACM. 2011, pp. 85–86.

[14] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. "An Introduction to Information Retrieval". In: ().

[15] Huazhong Ning et al. "Incremental spectral clustering with application to monitoring of evolving blog communities". In: *Proceedings of the 2007 SIAM International Conference on Data Mining*. SIAM. 2007, pp. 261–272.

[16] Clara Pizzuti. "GA-Net: A Genetic Algorithm for Community Detection in Social Networks." In: Springer.

[17] James Rosswog and Kanad Ghose. "Detecting and tracking spatio-temporal clusters with adaptive history filtering". In: *Data Mining Workshops, 2008. ICDMW'08. IEEE International Conference on*. IEEE. 2008, pp. 448–457.

[18] Satu Elisa Schaeffer. "Graph clustering". In: *Computer science review* 1.1 (2007), pp. 27–64.

[19] Ravi Shankar, GVR Kiran, and Vikram Pudi. "Evolutionary clustering using frequent itemsets". In: *Proceedings of the First International Workshop on Novel Data Stream Pattern Mining Techniques*. ACM. 2010, pp. 25–30.

[20] *Spark Blockmatrix*. URL: https://spark.apache.org/docs/latest/mllib-data-types.html#blockmatrix.

[21] *Spark Clustering*. URL: https://spark.apache.org/docs/latest/ml-clustering.html.

[22] *Spark CoordinateMatrix*. URL: https://spark.apache.org/docs/latest/mllib-data-types.html#coordinatematrix.

[23] *Spark Indexedrowmatrix*. URL: https://spark.apache.org/docs/latest/mllib-data-types.html#indexedrowmatrix.

[24] *Spark Transformations*. URL: https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations.

[25] Alexander Strehl and Joydeep Ghosh. "Cluster ensembles—a knowledge reuse framework for combining multiple partitions". In: *Journal of machine learning research* 3.Dec (2002), pp. 583–617.

[26] Ulrike Von Luxburg. "A tutorial on spectral clustering". In: *Statistics and computing* 17.4 (2007), pp. 395–416.

[27] Yi Wang et al. "Mining naturally smooth evolution of clusters from dynamic data". In: *Proceedings of the 2007 SIAM International Conference on Data Mining*. SIAM. 2007, pp. 125–134.

[28] Kevin S Xu, Mark Kliger, and Alfred O Hero. "Evolutionary spectral clustering with adaptive forgetting factor". In: *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*. IEEE. 2010, pp. 2174–2177.

[29] Kevin S Xu, Mark Kliger, and Alfred O Hero Iii. "Adaptive evolutionary clustering". In: *Data Mining and Knowledge Discovery* 28.2 (2014), pp. 304–336.

[30] Tianbing Xu et al. "Dirichlet process based evolutionary clustering". In: *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*. IEEE. 2008, pp. 648–657.

[31] Tianbing Xu et al. "Evolutionary clustering by hierarchical dirichlet process with hidden markov state". In: *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*. IEEE. 2008, pp. 658–667.

[32] Hwanjo Yu et al. "Scalable construction of topic directory with nonparametric closed termset mining". In: *Data Mining, 2004. ICDM'04. Fourth IEEE International Conference on*. IEEE. 2004, pp. 563–566.

[33] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 2–2.

[34] Matei Zaharia et al. "Spark: Cluster computing with working sets." In: *HotCloud* 10.10-10 (2010), p. 95.

[35] Jianwen Zhang et al. "On-line evolutionary exponential family mixture." In: *IJCAI*. 2009, pp. 1610–1615.

[36] Yuchao Zhang, Hongfu Liu, and Bo Deng. "Evolutionary clustering with DBSCAN". In: *Natural Computation (ICNC), 2013 Ninth International Conference on*. IEEE. 2013, pp. 923–928.

[37] Ying Zhao and George Karypis. "Evaluation of hierarchical clustering algorithms for document datasets". In: *Proceedings of the eleventh international conference on Information and knowledge management*. ACM. 2002, pp. 515–524.

Appendix A

Evolutionary *k*-means algorithms

A.1 Evolutionary *k*-means algorithm

Algorithm 1: Evolutionary *k*-means

```
1  function EvolutionaryKMeans(data, k, CP, iterNum, seed):model, hisCost, snapQuality
   Input  : data: A batch of dataset in libsvm format which can be transformed to
            sql.DataFrame directly. k: number of centroids, CP: Change Parameter,
            iterNum: Iteration Number of k-means.
   Output: model: an array of k-means centroids (vector). hisCost: Historical cost of
            current model. snapQuality: Snapshot Quality of current model.
2  data = toDataFrame(data);
3  newDim = data.first().size;
4  if isFirstBatch() then
5  |  dim = newDim
6  else
7  |  if newDim! = dim then
8  |  |  Error
9  |  end
10 end
11 normData = normalize(data)**Parallelization**;
12 normData.cache()
13 if isFirstBatch() then
14 |  currentModel = Initialize(normData) ;
15 |  lastModel = currentModel
16 else
17 |  currentModel = lastModel;
18 end
19 lastCentroidCount = getCentroidCount()**Parallelization**
```

```
20 for 1 to iterNum do
21     currentCentroidCountMean = getCentroidCountMean()**Parallelization**
22     centroidPair = matchCentorids(lastModel, currentModel)
23     currentModel =
         updateCentroid(centroidPair, lastCentroidCount, currentCentroidCountMean)
24     model = normalize(currentModel)
25 end
26 hisCost = calculateHisCost(model, lastModel)
27 snapQuality = calculateSnapQuality(model, normData)
28 return model, hisCost, snapQuality
```

Algorithm 2: Match Nearest Centroids

```
1  function MatchCentorids(currentModel, lastModel):centroidPairs
   Input  : currentModel: Centroids of current batch data, lastModel: Centroids of
            last batch data.
   Output: centroidPairs: an array of tuple (lastcentroid, currentcentroid)
2  centroidPairs = null
3  if !lastModel.isEmpty() then
4      nearestPairs = lastModel.findclosestCentroid(currentModel)
5      nearestPairs = nearestPairs.sortByDistance()
6      for each pair in nearestPairs do
7          if lastModel.contains(pair.last)&&currentModel.contains(pair.current) then
8              centroidPairs+ = (pair.last, pair.current)
9              lastModel.delete(pair.last)
10             currentModel.delete(pair.current)
11         end
12     end
13 end
14 return centroidPairs
```

A.2 Match Two Clustering Algorithm

A.3 Centroid-Number algorithm

Algorithm 3: Centroid-Number algorith

1 function getCentroidCount(*normData*, *lastModel*):*centroidCount*
Input : *normData*: normalized dataset, *lastModel*: Centroids of last batch data.
Output: *centroidCount*: a RDD of tuple $(lastcentroid, count)$
2 $lastModel = lastModel.toMatrix()$
3 $nearestCenter = normData.map(point => lastModel.multiply(point).argmax)$
****Parallelization****
4 $centroidCount = nearestCenter.map(center => (center, 1)).reduceByKey()$
****Parallelization****
5 **return** *centroidCount*

A.4 Centroid-Number-Mean algorithm

Algorithm 4: Centroid-Number-Mean algorith

1 function getCentroidCountMean(*normData*, *currentModel*):*centroidCountMean*
Input : *normData*: normalized dataset, *currentModel*: Centroids of current batch data.
Output: *centroidCountMean*: a RDD of triple $(currentCentroid, count, mean)$
2 $currentModel = currentModel.toMatrix()$
3 $nearestPointCenter = normData.map(point =>$
$(currentModel.multiply(point).argmax), point)$ ****Parallelization****
4 $centroidCountSum = nearestPointCenter.map((center, point) =>$
$(center, 1, point)).reduceByKey((number1, point1), (number2, point2) =>$
$number1 + number2, vectorSum(point1, point2))$ ****Parallelization****
5 $centroidCountMean = centroidCountSum.map((centroid, count, vectorSum) =>$
$(centroid, count, vectorSum/count))$ ****Parallelization****
6 **return** *centroidCountMean*

Appendix B

AFFECT

B.1 AFFECT

B.2 Optimizing CP

B.3 Ajusted k-means

Algorithm 5: AFFECT

```
1  function AFFECT(data,k,evoIter):clustering
   Input  : data: a RDD with observation in the format (index, value_0, value_1...),
            k:number of centroid, evoIter:Iteration number of AFFECT.
   Output: clustering: A map whose keys are clusters index and whose values are
            sets of point index of that cluster
2  objectSet = getIndex(data) **Parallelization**
3  broadcast(objectSet.collect())
4  dataWtihIndex = data.map((_.index, _.vector)) **Parallelization**
5  cartesianResult = dataWtihIndex.cartesian(dataWtihIndex)
6  proximity =
     cartesianResult.map(new MatrixEntry(_1.index, _2.index, eucliDis(_1.vector, _2.vector))
     **Parallelization**
7  SimilarityMatrix = newCoordinateMatrix(proximity)
8  if isFirstBatch() then
9  |  lastMatrix = SimilarityMatrix
10 |  currentModel = initRandom(objectSet)
11 else
12 |  currentModel = lastModel
13 end
14 lastMatrix = AddDeletePoint(lastMatrix)
15 lastMatrix.cache()
16 for i in evoIter do
17 |  CP = calculateCP(currentModel, SimilarityMatrix, lastMatrix)
18 |  CurrentWeight = SimilarityMatrix.entries.map(new MatrixEntry(i, j, value *
   |    (1 − CP))).toBlockMatrix() **Parallelization**
19 |  LastWeight = SimilarityMatrix.entries.map(new MatrixEntry(i, j, value *
   |    CP)).toBlockMatrix() **Parallelization**
20 |  EvolutionaryMatrix = CurrentWeight + LastWeight
21 |  clustering = staticClustering(EvolutionaryMatrix, currentModel)
22 end
23 hisCost = calculateHisCost(currentModel, lastModel)
24 lastMatrix.unpersist()
25 snapQuality = calculateSnapQuality(currentModel, normData)
26 lastMatrix = EvolutionMatrix
27 lastModel = currentModel
28 return clustering
```

Algorithm 6: Optimizing CP

function calculateCP($lastMatrix$,$currentMatrix$, $clustering$):CP
Input : $lastMatrix$: Adjusted proximity matrix of last timestep,$currentMatrix$: proximity matrix of current data, $clustering$: current clustering result.
Output: CP: change parameter.
$numerator = spark.accumulator("numerator")$
$denominator = spark.accumulator("denominator")$
$SameObjectEntry =$
$divideMatrixSameObject(clusterSet, lastMatrix, currentMatrix)$
$SameClusterEntry =$
$divideMatrixSameCluster(clusterSet, lastMatrix, currentMatrix)$
$DifClusterEntry = divideMatrixDifCluster(clusterSet, lastMatrix, currentMatrix)$
$lastMatrix.foreach\{entry => numerator.update(), denominator.update()\}$
****Parallelization****
return CP

Algorithm 7: Adjusted k-means

function adjustKMeans($Matrix$, $lastModel$):$newModel$
Input : $Matrix$: proximityMatrix, $lastModel$:A map whose index is cluster index, and value is the points belong to the cluster
Output: $currentModel$: new clustering result in a Map.
$currentModel = null$
while $lastModel != currentModel$ **do**
    $sameClusterMean = weightMeanSameCluster(lastModel, Matrix)$
    $broadcast(sameClusterMean)$
    $pointCenterPair =$
    $Matrix.toIndexedRowMatrix.rows.MinCenterPoint(sameClusterMean)$
    $lastModel = currentModel$
    $currentModel = pointCenterPair$
end
return $currentModel$

Appendix C

Experiment records

C.1 Temporal Quality of PCM for non-noise dataset

Table C.1: Temporal Quality of PCM for non-noise time-evolved data.

CP	batch 0	batch 1	batch 2	batch 3	batch 4	batch 5	batch 6
0.0	0.0	2.79418618538138	2.564631441578113	2.5849381445473765	2.580444471320224	2.428485707664763	2.528945717015586
0.2	0.0	2.7941863623881598	2.564632357612078	2.584939083991315	2.580445392948348	2.428487154293394	2.528946580950538
0.4	0.0	2.7941866573989818	2.564633884332414	2.5849406497273253	2.580446928992034	2.4284893386742166	2.5289480208425568
0.6	0.0	2.794187247418819	2.56463693776208	2.5849437811848017	2.58045000106745	2.4284938674349923	2.52895090062822
0.8	0.0	2.7941890174639346	2.5646460979630055	2.584953175440823	2.580459217198058	2.4285374537104014	2.52895953999821
1.0	0.0	3.0	3.0000000000000013	3.0000000000000036	3.0000000000000018	2.9999999999999996	2.9999999999999996

C.2 Adjusted Temporal Quality of PCM for non-noise dataset

Table C.2: Adjusted Temporal Quality of PCM for non-noise time-evolved data.

CP	batch 1	batch 2	batch 3	batch 4	batch 5	batch 6
0.0	618538138	31441578113	38144547376	44471320224	48570766476	45717015586
0.2	636238815	32357612078	39083991315	45392948348	48715429339	46580950538
0.4	665739898	33884332414	40649727325	46928992034	48933867421	48020842556
0.6	724741881	36937762080	43781184801	50001067450	49386743499	50900628220
0.8	901746393	46097963005	53175440823	59217198058	53745371040	59539998210

C.3 Snapshot Quality of PCM for non-noise dataset

Table C.3: Snapshot Quality of PCM for non-noise time-evolved data.

CP	batch 0	batch 1	batch 2	batch 3	batch 4	batch 5	batch 6
0.0	2959645.2518285275	2961418.333846696	2957963.939101965	2949440.692331399	2959181.0564527093	2961696.9257635702	2965633.512843652
0.2	2959645.251828525	2961418.333846675	2957963.9391018646	2949440.692331361	2959181.056452605	2961696.9257634557	2965633.51284361
0.4	2959645.251828526	2961418.33384654	2957963.9391012304	2949440.6923311497	2959181.0564519716	2961696.925762758	2965633.5128433523
0.6	2959645.251828527	2961418.3338458994	2957963.939098252	2949440.6923301583	2959181.056448963	2961696.9257594533	2965633.5128421322
0.8	2959645.251828528	2961418.3338410323	2957963.939075576	2949440.6923225923	2959181.0564260683	2961696.9257342913	2965633.5128328432
1.0	2959645.2518285275	2834475.173251886	2806232.433293705	2764605.3700496056	2758403.9011363573	2766899.9708003392	2777680.115364826

C.4 Adjusted Snapshot Quality of PCM for non-noise dataset

Table C.4: Adjusted Snapshot Quality of PCM for non-noise time-evolved data.

CP	batch 1	batch 2	batch 3	batch 4	batch 5	batch 6
0.0	6696	10196	3139	5270	6357	4365
0.2	6675	10186	3136	5260	6345	4361
0.4	6540	10123	3114	5197	6275	4335
0.6	5899	9825	3015	4896	5945	4213
0.8	1032	7557	2259	2606	3429	3284

C.5 NMI of PCM for noise dataset

Table C.5: NMI of PCM for noise dataset.

CP	batch 0	batch 1	batch 2	batch 3	batch 4	batch 5	batch 6	batch 7	batch 8
0.0	1.0	0.9701911554558438	0.7739862721886919	0.7180275258245575	0.4971340263759154	0.7409194869433314	0.867865487335697	0.9829929733076637	0.9829929733076637
0.2	1.0	0.9317131161633849	0.9011219347136158	0.7242284135114484	0.6417529618267943	0.7409194869433314	0.8678654873356971	0.9829929733076637	0.9829929733076637
0.4	1.0	0.9701911554558438	0.9011219347136158	0.7242284135114484	0.584917911865751	0.682983686559486	0.8678654873356971	0.9829929733076637	0.9829929733076637
0.6	1.0	0.9701911554558438	0.9054528902493049	0.7242284135114484	0.584917911865751	0.7409194869433312	0.867865487335697	0.9829929733076637	0.9829929733076637
0.8	1.0	0.9701911554558436	0.9054528902493049	0.7242284135114484	0.6218105432816295	0.7409194869433314	0.867865487335697	0.9501535823395036	0.9829929733076637
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

C.6 Temporal Quality of PCM for noise dataset

Table C.6: Temporal Quality of PCM for noise time-evolved data.

CP	batch 0	batch 1	batch 2	batch 3	batch 4	batch 5	batch 6	batch 7	batch 8
0.0	0.0	1.8881846113840206	1.6520790027784977	1.3445378870458455	1.2158164417024182	1.308989593000706	1.6295708458002574	1.9157834154399633	1.8802650739740205
0.2	0.0	1.888190279444579	1.6520826833609616	1.344540149197961	1.215820439820447	1.3089915381823871	1.6295767009078368	1.9157902747331519	1.8802665113658867
0.4	0.0	1.8881997266405075	1.6520888178698487	1.3445439197067088	1.2158271044952544	1.3089947804696362	1.629586460174546	1.9158017075271707	1.8802689071350005
0.6	0.0	1.8882186226387616	1.652101087655727	1.3445514616815704	1.2158404381394232	1.3090012662362023	1.6296059815359392	1.9158245755104324	1.8802736991082112
0.8	0.0	1.8882753234870135	1.6521379031588086	1.3445740952698235	1.2158804734617676	1.3090207330818775	1.629664568253464	1.9158931986261438	1.880288078508725
1.0	0.0	3.0000000000000147	3.0000000000000013	3.000000000000002	3.0000000000000013	3.000000000000002	2.9999999999999987	2.999999999999999	3.0000000000000027

C.7 Adjusted Temporal Quality of PCM for noise dataset

Table C.7: Adjusted Temporal Quality of PCM for noise time-evolved data.

CP	batch 1	batch 2	batch 3	batch 4	batch 5	batch 6	batch 7	batch 8
0.0	1846113	790027	3788704	1644170	8989593	5708458	7834154	6507397
0.2	1902794	826833	4014919	2043982	8991538	5767009	7902747	6651136
0.4	1997266	888178	4391970	2710449	8994780	5864601	8017075	6890713
0.6	2186226	1010876	5146168	4043813	9001266	6059815	8245755	7369910
0.8	2753234	1379031	7409526	4734617	9020733	6645682	8931986	8807850

Table C.8: Snapshot Quality of PCM for noise time-evolved data.

CP	batch 0	batch 1	batch 2	batch 3	batch 4	batch 5	batch 6	batch 7	batch 8
0.0	2997286.2601840053	2997233.7514784402	2994756.1335715812	3000732.623520988	3000504.465596417	2997242.145316064	3000753.012426059	2999788.7523815953	3003411.5397727704
0.2	2997286.2601840524	2997233.7514776993	2994756.133571206	3000732.623520768	3000504.4655959182	2997242.145315908	3000753.012425401	2999788.752380965	3003411.539772654
0.4	2997286.2601840515	2997233.7514733663	2994756.1335689256	3000732.6235194453	3000504.465592872	2997242.145314947	3000753.0124214035	2999788.752377143	3003411.5397719443
0.6	2997286.2601840557	2997233.7514528967	2994756.133558163	3000732.623513194	3000504.4655784722	2997242.1453104126	3000753.012402493	2999788.7523590485	3003411.539768604
0.8	2997286.2601840557	2997233.7513969673	2994756.133546174	3000732.625065531	3000504.4655687257	2997242.1453058648	3000753.0123984146	2999788.75235121	3003411.539765146
1.0	2997286.2601840533	2916980.6008960716	2928275.2447140515	2918651.003846713	2913624.5332332626	2883679.2614975446	2889981.1066282596	2852088.9930265415	2805363.2063691844

Table C.9: Adjusted Snapshot Quality of PCM for noise time-evolved data.

CP	batch 1	batch 2	batch 3	batch 4	batch 5	batch 6	batch 7	batch 8
0.0	478402	571581	520988	596417	316064	426059	381595	727704
0.2	477699	571206	520768	595918	315908	425401	380965	726540
0.4	473366	568925	519445	592872	314947	421403	377143	719443
0.6	452896	558163	513194	578472	310412	402493	359048	686040
0.8	396967	546174	506553	568725	305864	398414	351210	651460

C.8 Snapshot Quality of PCM for noise dataset

C.9 Adjusted Snapshot Quality of PCM for noise dataset

C.10 Temporal Cost of Evolutionary *k*-Means for non-noise dataset

Table C.10: Temporal Cost of Evolutionary *k*-Means for non-noise time-evolved data.

CP	batch 0	batch 1	batch 2	batch 3	batch 4	batch 5	batch 6
0.0	0.0	0.8157691522977811	0.8659157600915951	0.9283515595592866	0.8644783303235639	0.8957323852170261	0.8884068541374655
0.2	0.0	0.7325465817523916	0.832196483784308	0.9014480263546701	0.8569428669667812	0.8790682260169895	0.8757383626963002
0.4	0.0	0.5474462350796749	0.728706147612146	0.834701859229267	0.8258971111108274	0.8468207560027983	0.847935142215625
0.6	0.0	0.36240315201706624	0.5510496538063343	0.6823041340374237	0.7233301744309403	0.7746344191636392	0.7886867398900091
0.8	0.0	0.17933199174442152	0.3051975723909362	0.4226621996587821	0.5229920625902855	0.6457969544645339	0.7312802950869099

C.11 Snapshot Quality of Evolutionary *k*-means for non-noise dataset

Table C.11: Snapshot Quality of Evolutionary *k*-means for non-noise time-evolved data.

CP	batch 0	batch 1	batch 2	batch 3	batch 4	batch 5	batch 6
0.0	0.8283708852067486	0.8327354081749027	0.8253040381092734	0.8276198560186273	0.8251857553337235	0.8300057427343421	0.8311601685678628
0.2	0.8283708852067486	0.8236671627414754	0.8130380839534908	0.8149037966652238	0.8131664454976615	0.8169490684152143	0.8183800397620151
0.4	0.8283708852067491	0.7981809119333384	0.7670801936926255	0.7542722038004684	0.7526832819851206	0.7519728301248954	0.7532781547570846
0.6	0.8283708852067486	0.7599389129575106	0.6794963078710272	0.6194147136334596	0.5987615290466337	0.584704101833293	0.5810772820926002
0.8	0.8283708852067488	0.7141476090870659	0.5565192497880586	0.4156973186725811	0.34503486021656843	0.31426416110721171	0.29293825493864415

C.12 NMI of Evolutionary *k*-means for noise dataset

Table C.12: NMI of Evolutionary *k*-means for noise dataset.

CP	batch 0	batch 1	batch 2	batch 3	batch 4	batch 5	batch 6	batch 7	batch 8
0.0	0.996357926510910 4	0.9724379883396751	0.8776249195454948	0.68759162349734	0.5794154878593103	0.6854758188560143	0.8668439759912974	0.9709126581431078	0.9969400177317583
0.2	0.996357926510910 4	0.9726082399251037	0.8780014725129598	0.6876780546150545	0.5794020091353757	0.6852553394891804	0.865725609912811 7	0.970903528653236	0.9969204235245254
0.4	0.996357926510910 4	0.9727808845747723	0.8782293316847862	0.6878031503322383	0.5793960000490327	0.6853731699048665	0.8656013403000266	0.970901359756586	0.9969000134134143
0.6	0.996357926510910 4	0.9729381755592637	0.8780230297869933	0.6879756612739988	0.579386625750811	0.6855421581778604	0.8656013403000266	0.9708743902532503	0.9967122676086605
0.8	0.996357926510910 4	0.973114395315707	0.8784611086180283	0.6879881575408359	0.5794240573735349	0.685932397485606 3	0.8654520005465765	0.970604720353678	0.9964895980389522

C.13 Adjusted NMI of Evolutionary *k*-means for noise dataset

Table C.13: Adjusted NMI of Evolutionary *k*-means for noise dataset.

CP	batch 0	batch 1	batch 2	batch 3	batch 4	batch 5	batch 6	batch 7	batch 8
0.0	357926	243798	762491	591623	415487	547581	684397	912658	940017
0.2	357926	260823	800147	678054	402009	525533	572560	903528	920423
0.4	357926	278088	822933	803150	396000	537316	560134	901359	900013
0.6	357926	293817	802302	975661	386625	554215	560134	874390	712267
0.8	357926	311439	846110	988157	424057	593239	545200	604720	489598

C.14 Temporal Cost of Evolutionary *k*-means for noise dataset

Table C.14: Temporal Cost of Evolutionary *k*-means for noise time-evolved data.

CP	batch 0	batch 1	batch 2	batch 3	batch 4	batch 5	batch 6	batch 7	batch 8
0.0	0.0	0.2744531130863489	0.2592477697031659	0.2145766241421455	0.11880485193477844	0.11932620996603133	0.2130728597707409	0.264768430601621	0.27885586827990205
0.2	0.0	0.21922557880980562	0.25060543061071233	0.22129589625671828	0.1391699319027682	0.0890418262897660 4	0.1879652259018278 2	0.24892704652243508	0.2719383455671389
0.4	0.0	0.16411549434464123	0.2202898870366507	0.21603400266845335	0.15737721606326188	0.05891881657119968 4	0.1472033274593832	0.21685021838126495	0.2524144624026989
0.6	0.0	0.10917398411223493	0.16822312512156665	0.18585958803868557	0.15872143364290575	0.0915366560721461 2	0.0891468284365063	0.15795096401870837	0.20427435557619488
0.8	0.0	0.05445203292446815	0.09459853975282177	0.11774900225691627	0.1177681363487246	0.0924027995462935 2	0.0529629297307131 76	0.07186091336944392	0.11132249604340796

C.15 Snapshot Quality of Evolutionary *k*-means for noise dataset

Table C.15: Snapshot Quality of Evolutionary *k*-means for noise time-evolved data.

CP	batch 0	batch 1	batch 2	batch 3	batch 4	batch 5	batch 6	batch 7	batch 8
0.0	0.8813489085250211	0.8860705542482572	0.8938772260871086	0.9065795241612601	0.9178623996303693	0.9057677842391284	0.8905374188555276	0.881506301520095	0.8750847559455854
0.2	0.8813489085250211	0.8844802873070233	0.8916526328615005	0.9035428839415529	0.9164244470472871	0.9059865148597118	0.890443539774675	0.8808718269961499	0.8742146407429779
0.4	0.8813489085250199	0.8806789153198805	0.8839855024754855	0.8936726854153328	0.9097573060921301	0.9046174101116 2	0.8886099361404785	0.8760641589210413	0.8673638057383362
0.6	0.8813489085250211	0.8748597937932547	0.8677307666137014	0.8691527120928025	0.8865534390296472	0.8956094133008714	0.8850173471146269	0.8660187052976717	0.8485194190023789
0.8	0.8813489085250211	0.8672425875618369	0.8408148516137072	0.8211117823600184	0.8266966765643387	0.8510046373474721	0.8674494966619815	0.8533416034237001	0.8200381394399169

C.16 NMI of AFFECT *k*-means for noise dataset

Table C.16: NMI of AFFECT *k*-means for noise dataset.

CP	batch 0	batch 1	batch 2	batch 3	batch 4	batch 5	batch 6	batch 7	batch 8
0.0	1.0	1.0	0.839671062716	0.677302576416	0.6506724208907063	0.707960539346	0.8885453014338626	0.9889753057921956	1.0
0.2	1.0	1.0	0.857552088874	0.659941657685	0.581152182519942 2	0.747486195464	0.8487945710972422	0.9529929733076637	1.0
0.4	1.0	1.0	0.880172957310	0.682978953134	0.647552088874601	0.6365829144738	0.7081444495253784	0.9474861954642688	1.0
0.6	1.0	1.0	0.907602763882	0.727140312459	0.6798420407796374	0.684825652452	0.7829929733076637	0.8829929733076637	1.0
0.8	1.0	1.0	0.911215276941	0.777318042745	0.6913454251454545	0.654822028705	0.6829929733076637	0.8299425222457374	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Table C.17: Efficiency Experiment record of Evolutionary *k*-means.

data size	5000000	3000000	1000000	500000	250000	100000	50000
5 workers	25s	12s	7.5s	6.67s	6.5s	7.83s	9.16s
4 workers	35s	13s	7.5s	6.67s	6s	6.83s	8s
3 workers	100s	14s	8.5s	7.33s	7s	6s	6.83s
2 workers	140s	22s	9.3s	7.67s	6s	5.83s	5.33s
1 worker	320s	76s	12s	8.67s	5.67s	5.16s	4.83s

Table C.18: Efficiency Experiment record of PCM.

data size	6000	5000	4000	3000	2000	1000
6 workers	139s	128s	139s	80s	52s	27s
5 workers	143s	136s	141s	77s	45s	26s
4 workers	154s	140s	121s	84s	43s	25s
3 workers	176s	156s	119s	82s	38s	24s
2 workers	243s	190s	160s	112s	43s	25s
1 worker	364s	301s	198s	129s	66s	28s

Table C.19: Efficiency Experiment record of AFFECT-Spec.

data size	6000	5000	4000	3000	2000	1000
6 workers	856s	714s	453s	321s	143s	65s
5 workers	921s	756s	470s	259s	164s	59s
4 workers	975s	722s	498s	311s	158s	56s
3 workers	1088s	873s	466s	335s	160s	79s
2 workers	1353s	1043s	525s	451s	169s	55s
1 worker	1623s	1233s	842s	426s	244s	84s

C.17 Efficiency Experiment record of Evolutionary k-means.

C.18 Efficiency Experiment record of PCM.

C.19 Efficiency Experiment record of AFFECT-Spec.

C.20 Efficiency Experiment record of AFFECT-kmeans.

Table C.20: Efficiency Experiment record of AFFECT-kmeans.

data size	6000	5000	4000	3000	2000	1000
6 workers	812	726	480	312	179	62
5 workers	953	734	467	285	162	61
4 workers	964	712	445	361	163	59
3 workers	1203	839	437	385	159	74
2 workers	1602	1072	498	402	171	57
1 workers	1832	1262	814	435	238	80

TRITA TRITA-ICT-EX-2017:201