Organizing Graph Structure for Statistical Neighborhood Search Optimization

Dalibor Krleža
University of Zagreb

Anamari Nakić
University of Zagreb

Boris Vrdoljak
University of Zagreb

Abstract

Various modern computing solutions and algorithms, such as statistical databases and clustering algorithms, use statistical inference to classify input data. Statistical classification by utilizing statistical distance is computationally complex and requires optimized search methods to reduce the number of classification candidates in the learned statistical model. The existing indexing data structures, such as B-trees, R-trees, and M-trees, are designed for organizing classes into containing subspaces, which cannot be used to organize statistical distributions. When using statistical distance, the relationship between any statistical distribution and an input data variable is relative, which poses an issue when trying to organize statistical distributions into an indexing structure. In this paper, we propose a statistical organizing graph structure named Σ-index. The Σ-index can be used for organizing statistical distributions to reduce the time needed for the statistical classification. It organizes statistical distributions based on the relative statistical distance and relationship among them, which consequently reduces the number of tested distributions needed to classify the input data variable. The Σ-index was applied on an existing statistical clustering algorithm, to demonstrate the computation cost reduction between the proposed Σ-index and naïve sequential approach.

Keywords: Indices, Statistical classification, Clustering, Databases, Navigable Small World.

1. Introduction

In the statistical machine learning context (Alpaydin 2020), statistical classification is a mapping between an input random variable and a set of statistical distributions, i.e., a statistical model. It can be taken that the input variable is a random vector in an n-dimensional space

$$X = [x_1, x_2, ..., x_n]^\top \in \Theta \subset \mathbb{R}^n \tag{1}$$

where Θ is underlying observed process parameter space. The outcome of the statistical classification is in set of multivariate statistical distributions

$$F_X = \{f_{X_1}, f_{X_2}, ..., f_{X_k}\} \tag{2}$$

The mapping function for the statistical classification could be built using the *maximal likelihood estimation* (MLE) principle (Rossi 2018). Another method is using the statistical

(Mahalanobis) distance (Mahalanobis 1936), correlated to the MLE principle. By using multivariate normal distributions

$$f_{X_i} = \mathcal{N}_n^i(\mu_i, \Sigma_i) \in F_X \tag{3}$$

the statistical distance of a random n-dimensional vector X can be calculated as

$$d_M(X, f_{X_i}) = \sqrt{[X - \mu_i]^\top \Sigma_i^{-1} [X - \mu_i]} \tag{4}$$

The statistical mapping function, i.e., *a classifier* in the machine learning terminology, is an optimization function that selects the most fitting outcome from the set of statistical distributions F_X.

$$f_{X_C} = \underset{f_{X_i} \in F_X}{arg\ min}\ d_M(X, f_{X_i}) \tag{5}$$

Calculating (5) is computationally expensive due to the complexity of the statistical distance (4) calculation. Each calculation of $Q = \Sigma^{-1}$ using Cholesky decomposition is $O(n^3)$ complex (Krishnamoorthy and Menon 2013). Matrix and vector products must be added to the computational expensiveness of (4), especially as the number of dimensions n rises.

In an naïve attempt to implement the optimization function (5), we could *sequentially scan* all statistical distributions in F_X. The time needed to solve (5) rises with number of statistical distributions in F_X. Statistical Hierarchical Clustering (SHC) algorithm (Krleža, Vrdoljak, and Brčić 2020) is an example of the statistical classifier, which shows that time needed to solve (5) with the naïve sequential scan attempt is significantly greater than competitive non-statistical clustering algorithms. This affects the preferred choice of the classification algorithms. It must be noted that many non-statistical clustering algorithms use some kind of spatial organizing structures, such as R-trees (Guttman 1984) or M-trees (Ciaccia, Patella, and Zezula 1997), to reduce the number of computations needed to search for the optimal resulting class. This is possible since most of the non-statistical classifiers work in the absolute Euclidean space, where the relationships between classes, including the input variable, can be described with Euclidean distance.

Applying statistical distance (4) of the set F_X will result in a set of statistical spaces

$$\mathcal{M} = \{M_i : 1 \leq i \leq |F_x|, M_i = \mathbb{R}^1, c(M_i) = \mu_i(f_{X_i})\} \tag{6}$$

where each statistical space is a one-dimensional real space $\mathbb{R}^1$ with centroid $c(M_i) = \mu_i(f_{X_i})$ in the center of the statistical distribution f_{X_i} population. A value in the statistical space $d_1 = d_M(\cdot, f_{X_i}) \in M_i, c(M_i) = \mu_i$ transformed back to the Euclidean space forms a level hyper-surface made of points that are d_1 standard deviations distant from the centroid $c(M_i)$. It is expect the following to hold

$$\forall f_{X_i}, f_{X_j} \in F_X : i \neq j, \mu_i \neq \mu_j, \Sigma_i \neq \Sigma_j \Rightarrow d_M(\mu_i, f_{X_j}) \neq d_M(\mu_j, f_{X_i}) \tag{7}$$

The previous inequality is the consequence of the difference between centroids μ_i, μ_j and precision matrices $Q_i = \Sigma_i^{-1}, Q_j = \Sigma_j^{-1}$ in the transformation done by (4). The conclusion is that the set of statistical spaces $\mathcal{M}$ comprises disjunct unrelated statistical spaces. Hence,

statistical distributions F_X cannot be organized in B-trees (Bayer and McCreight 1970), R-trees or M-trees, since these structures require that child nodes represent subspaces of the parent node space, which cannot be found in $\mathcal{M}$.

An interesting concept of navigable "small world" (NSW) graphs was introduced by Kleinberg (Kleinberg 2000). Kleinberg was working with social graphs, in which he described social interactions between people. The term "small world" comes from an idea that we can connect the whole world through creating a graph made of people acquaintances. This theory resulted in *navigable networks*, which are characterized by their structure and navigation algorithm. Based on this principle, many indexing structures and algorithms were developed, e.g., for searching nearest neighbors in clustering (Malkov, Ponomarenko, Logvinov, and Krylov 2012).

Using NSW principle, we propose a novel probabilistic-statistical structure suitable to organize statistical distributions in F_X, called the Σ-index. The purpose of the Σ-index is to reduce the computational cost for (5), i.e., to enable efficient search for a statistical distribution in F_X that is the optimal fit for the input variable X. The Σ-index is constructed by combining *statistical distribution neighborhood* and population size. Creating, updating, and querying the Σ-index are outlined in several algorithms given throughout the paper.

We cannot assume that the underlying observed processes are static (Gama 2010) and do not change over time. Statistical distribution evolution affects how the proposed Σ-index is updated and used. Modern statistical clustering algorithms capture such evolution changes as a population drift (Gama 2010) or even a population split (Krleža *et al.* 2020). Such statistical evolutionary changes of the underlying observed processes are discussed and taken into consideration in the updating and querying algorithms of the Σ-index.

The experimental verification and assessment of the Σ-index is done in the SHC implementation, which allows the conclusion about the achieved computation cost reduction of the proposed Σ-index in comparison to the sequential scan approach. The comparative testing was performed on various synthetic and real-life examples, giving insight into the average usability of the Σ-index. The results are then compared to the theoretical Σ-index complexity assessment.

This paper is organized as follows: the next Section is covering *statistical neighborhood* definition, Σ-index construction rules, and statistical distribution evolution. In Section 3 we elaborate algorithms for creating, updating, and querying the Σ-index. In the same Section we give definitions for the computational cost reduction calculation. Experimental evaluation and usage of the proposed Σ-index is done in Section 4. Section 5 covers additional theoretical assessment of the Σ-index advantages and usability over the sequential scan approach. The conclusion is given in Section 6.

2. Statistical neighborhood and structure definition

The term of the statistical neighborhood is discussed in (Krleža *et al.* 2020) as part of the outcome calculation, where we calculate the neighborhood of the input observation X. Once the optimal fitting statistical distribution f_{X_i} is known, it was taken that the f_{X_i} neighborhood is the same as the X neighborhood, which is an imprecise approximation used to avoid statistical distance re-calculation.

Definition 2.1 ***Statistical neighbor****: The statistical distribution f_{X_i} is a neighbor of the statistical distribution f_{X_j} if*

$$d_M(\mu_i, f_{X_j}) \leq \theta_n \tag{8}$$

where θ_n is the neighborhood threshold.

Such approach allows us to determine the complete neighborhood of a statistical distribution f_{X_i}, which is a mapping multifunction

$$\begin{aligned} f_N : F_X &\to P(F_X) \\ f_N(f_{X_i}) &= \{f_{X_j} \in P(F_X) : 0 < d_M(\mu_j, f_{X_i}) \leq \theta_n\} \end{aligned} \tag{9}$$

The observed underlying process sample defined in (1) can be split into populations forming the statistical distributions in F_X

$$\begin{aligned} \mathcal{P} : F_X &\to P(\Theta) \\ \mathcal{P}(f_{X_i}) &= \{X_j \in P(\Theta) : f_{X_i} = \underset{f \in F_X}{arg\ min}\ d_M(X_j, f)\} \end{aligned} \tag{10}$$

and the remainder of the *unprocessed* observations

$$X \in \Theta' = \Theta \setminus \bigcup_{f_{X_i} \in F_X} \mathcal{P}(f_{X_i}) \tag{11}$$

The observed sample Θ probabilistic distribution over F_X can be used to reduce the computational cost of (5). The most probable outcome of the input variable X are statistic distributions having the biggest populations. The probability that the input variable X outcome is the statistical distribution f_{X_i} can be expressed as

$$p(X \in f_{X_i}) = \frac{|\mathcal{P}(f_{X_i})|}{|\Theta| - |\Theta'|} \tag{12}$$

Using (9), (10), and (12) we can create a directed acyclic graph (DAG) (Harris, Hirst, and Mossinghoff 2008)

Definition 2.2 *Σ-index is a DAG G_Σ defined as*

$$\begin{aligned} G_\Sigma &= (N_\Sigma, E_\Sigma) \\ N_\Sigma &= F_X \cup \{root\} \\ E_\Sigma &\subseteq N_\Sigma \times N_\Sigma \end{aligned} \tag{13}$$

where the nodes of the graph are the statistic distributions from F_X.

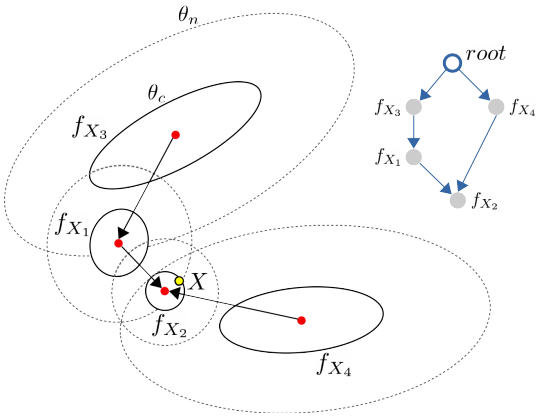


Figure 1: An example of the Σ-index.

Directed edges of the graph G_Σ are defined as follows

$$\forall(f_{X_i}, f_{X_j}) \in E_\Sigma : f_{X_j} \in f_N(f_{X_i}), |\mathcal{P}(f_{X_i})| \geq |\mathcal{P}(f_{X_j})| \tag{14}$$

which means that an edge is created from a statistical distribution f_{X_i} to all statistical populations in the neighborhood that are equally or less probable to be the outcome of the input variable X. Slightly differently taken

$$\forall f_{X_j} \in f_N(f_{X_i}) \exists (f_{X_i}, f_{X_j}) \in E_\Sigma : |\mathcal{P}(f_{X_i})| \geq |\mathcal{P}(f_{X_j})| \tag{15}$$

Based on (14), there is a possibility to create a cycle in the graph G_Σ, which must be prevented

$$\forall(f_{X_i}, f_{X_j}) \in E_\Sigma \nexists (f_{X_j}, f_{X_i}) \in E_\Sigma : |\mathcal{P}(f_{X_i})| = |\mathcal{P}(f_{X_j})| \tag{16}$$

In case population sizes of the neighbor statistical distributions are equal, only one directed edge connecting them in the graph G_Σ is allowed.

In (13) we added the *root* node, which is used for all nodes in the graph G_Σ that have no *incoming* edges.

$$\forall f_{X_i} \in F_X \nexists f_{X_j} \in F_X : (f_{X_j}, f_{X_i}) \in E_\Sigma \Rightarrow (root, f_{X_i}) \in E_\Sigma \tag{17}$$

In Figure 1, we can see an example of the Σ-index comprising four statistical distributions $F_X = \{f_{X_1}, ..., f_{X_4}\}$. The most populated statistical distributions are placed directly under the root node. These are statistical distributions that are the most probable outcome for the input variable X. As we descend down the Σ-index, we encounter less populated statistical distributions. Each transition from E_Σ means a step further into the statistical neighborhood. θ_c in Figure 1 is the *population bound* of each statistical distribution.

2.1. Evolutionary changes of the observed process

When the observed process sample gets big enough, we can face an evolutionary change in one of the statistical distributions. Such behavior have been noticed in data stream clustering

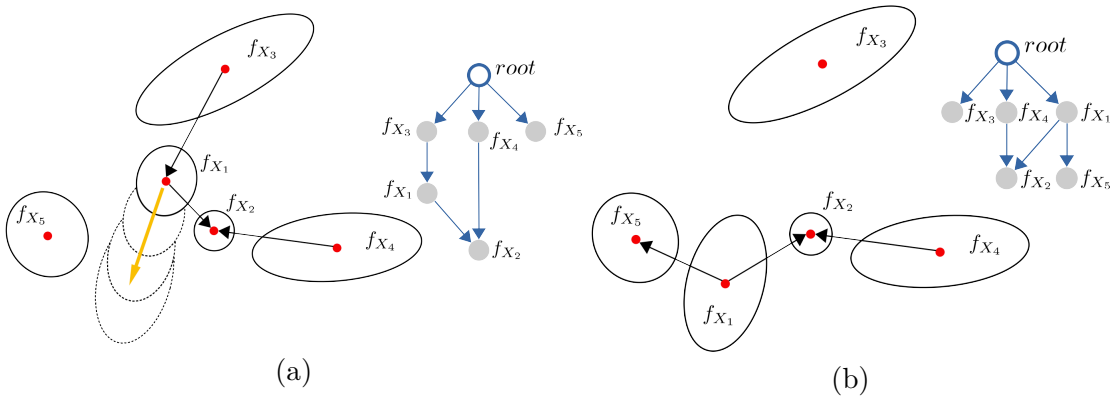


Figure 2: The observed process evolution.

algorithms where the observed sample becomes endless (Gama 2010). In the clustering algorithms terminology, such behaviour is known as statistical distribution drift and split (Krleža *et al.* 2020), and is completely covered by SHC.

Population rise

In some cases, a statistical distribution population can start rapidly rising, which can lead to the violation of the rule defined in (14). In case when a child node becomes bigger population-wise than one of its parents, we must change the direction of the violating edge in the Σ-index DAG.

$$\exists(f_{X_i}, f_{X_j}) \in E_\Sigma : |\mathcal{P}(f_{X_j})| > |\mathcal{P}(f_{X_i})| \Rightarrow E_\Sigma \setminus = (f_{X_i}, f_{X_j}), E_\Sigma \cup = (f_{X_j}, f_{X_i}) \tag{18}$$

Statistical distribution variance change or centroid move

One of the evolutionary changes is a change of a statistical distribution covariance Σ. This makes the statistical distribution neighborhood larger or smaller. Similarly, a statistical distribution centroid move can significantly change the neighborhood and relationships to neighbor statistical distributions, thus, it changes the Σ-index DAG structure.

In Figure 2 we can see an example of the observed process evolution. The population of the statistical distribution f_{X_1} starts to grow in size and variance. At the same time, the centroid $\mu(f_{X_1})$ moves away from the population of the statistical distribution f_{X_3}. The beginning of the evolution is seen in Figure 2a and the end is seen in Figure 2b.

The Σ-index DAG in such situations is updated by refreshing the structure surrounding the evolving statistical distribution.

2.2. Neighborhood search

Since there are no cycles in the Σ-index DAG, in our search for the neighborhood of the input variable X results in a set of walks $\mathcal{S}$ (Harris *et al.* 2008). An acyclic walk is a sequence of nodes

$$\mathcal{S} = \{w_i = (n_1, n_2, ..., n_e) : n_j \in N_\Sigma, n_j \neq n_k\} \tag{19}$$

where nodes cannot be repeated in the sequence. Each neighborhood search walk can be divided into two subwalks

$$\begin{aligned} w_i &= (w_p, w_s) \\ \forall n_j \in w_p : d_M(X, n_j) &> \theta_n \\ \forall n_j \in w_s : d_M(X, n_j) &\leq \theta_n \end{aligned} \tag{20}$$

Dividing the neighborhood search into two subwalks reflects two distinct phases of the search. In the first phase, i.e., doing the subwalk w_p, we traverse nodes that do not have the input variable X in their neighborhood, i.e., $d_M > \theta_n$, which we add to the subwalk w_p. This directly means wasted computations. As soon as we find the first node having X in the neighborhood, i.e., $d_M \leq \theta_n$, this node becomes the first node in the w_s subwalk. This represents the second phase of the neighborhood search, where we must find all nodes that have X in the neighborhood. The complete neighborhood is calculated as the result of the following multifunction

$$\begin{aligned} f_{N_*} &: \Theta' \rightarrow P(F_X) \\ f_{N_p}(X) = \bigcup_{w_i \in \mathcal{S}} w_p(w_i), f_{N_s}(X) &= \bigcup_{w_i \in \mathcal{S}} w_s(w_i) \end{aligned} \tag{21}$$

The main idea is to minimize all w_p subwalks, and maximize w_s subwalks. In case we do not have w_s subwalks in $\mathcal{S}$, this means that X does not have an outcome in the Σ-index DAG, hence X is an outlier. The outcome of the input variable X is a subset of the neighborhood

$$f_C(X) = \{n_i \in f_{N_s}(X) : d_M(X, n_i) \leq Q_c\} \subseteq f_{N_s}(X) \tag{22}$$

comprising only statistical distributions that have the input observation X directly in their population.

The analysis of the outcome subgraph

Let us define a vertex-induced subgraph $G_\Sigma[f_C(X)]$ of the Σ-index DAG, comprising only nodes from the outcome of the input variable $f_C(X) \subseteq N_\Sigma$. The subgraph $G_\Sigma[f_C(X)]$ could be composed of several disconnected subgraphs.

Theorem 2.1 ***The connection theorem****: The vertex-induced subgraph $G_\Sigma[f_C(X)]$ is connected for a sufficiently large neighborhood, having thresholds*

$$\theta_n \geq nm * \theta_c \tag{23}$$

where nm is a neighborhood multiplier. To achieve the connected vertex-induced subgraph $G_\Sigma[f_C(X)]$, the neighborhood multiplier must be $nm \geq 2$.

Proof: The minimal condition for $f_C(X)$ to be connected is to have one node that has all other nodes in the neighborhood, which means

$$\exists n_i \in f_C(X), \forall n_j \in f_C(X) : n_i \neq n_j, d_M(\mu(n_j), n_i) \leq \theta_n \tag{24}$$

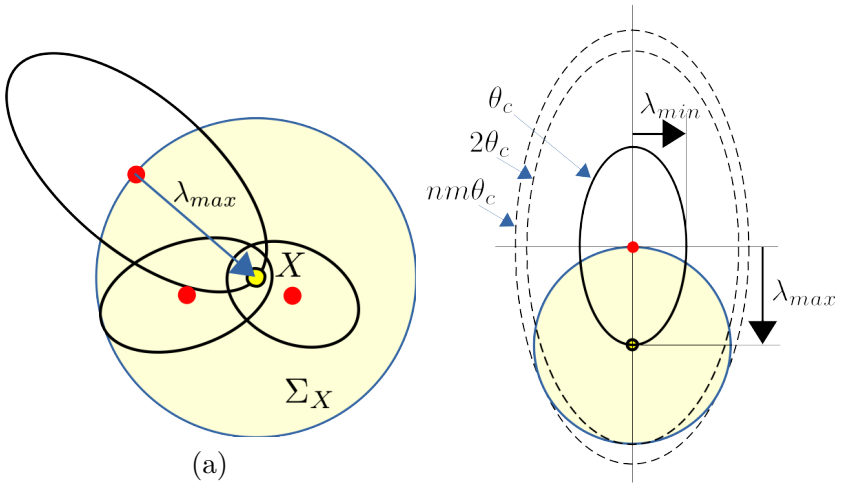


Figure 3: The outcome neighborhood.

We start by finding a hyper-sphere around the input variable X that contains all centroids from the outcome $f_C(X)$. We get the worst case scenario by performing the eigendecomposition on covariance matrices from $f_C(X)$ (Abdi and Williams 2010; Axler 2015). First, we find a node whose covariance matrix has the maximal eigenvalue in $f_C(X)$

$$n_b = \underset{n_i \in f_C(X)}{arg\ max}(max(\lambda(\Sigma(n_i)))) \tag{25}$$

where

$$\lambda_{min} = min(\lambda(\Sigma(n_b))), \lambda_{max} = max(\lambda(\Sigma(n_b))), \tag{26}$$

are the minimal and maximal eigenvalues of that node. We create a statistical distribution having X for the centroid and

$$diag(\Sigma_X) = \lambda_{max} \tag{27}$$

for the covariance matrix. For such distribution

$$\forall n_i \in f_C(X) : d_M(\mu(n_i), \mathcal{N}_n(X, \Sigma_X)) \leq \theta_c \tag{28}$$

all centroids are within the classifying statistical distance θ_c from the input variable X. This forms a hyper-sphere, similar to Figure 3a, which must be included in the neighborhood of n_b to form a connected vertex-induced subgraph $G_\Sigma[f_C(X)]$, as stated in (24). Figure 3b shows the projection of the n_b hyper-surface to the $\lambda_{min} - \lambda_{max}$ plane. To include the hyper-sphere $\mathcal{N}_n(X, \Sigma_X)$ bounded by θ_c into the n_b hyper-surface, we need to expand the n_b hyper-surface bound to be $> \theta_c$. Using the basic trigonometry, the expansion can be calculated as

$$\theta_n = \frac{2 * \lambda_{max}}{\lambda_{min}} * \theta_c, nm = \frac{2 * \lambda_{max}}{\lambda_{min}} \tag{29}$$

With $\lambda_{min} \leq \lambda_{max}$, the neighborhood multiplier is always $nm \geq 2$, which is similar to (Dasgupta 1999).

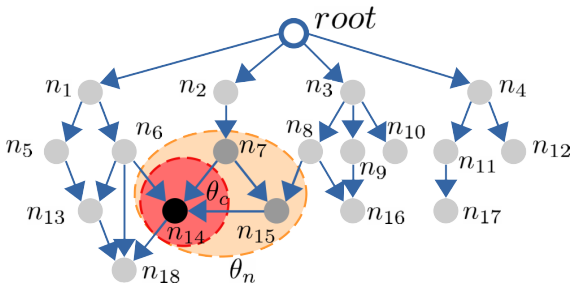


Figure 4: A neighborhood of the input variable X in the Σ-index DAG.

3. Algorithms and computational cost reduction

3.1. Query algorithm

The usual approach to find the first node in the Σ-index DAG that has X is the statistical neighborhood can use one of the well established algorithms, such as depth first search (DFS) or breadth first search (BFS) (Harris *et al.* 2008). The most sensible approach is using DFS. Randomly picking adjacent nodes in DFS gives the worst case scenario identical to the sequential scan (SS) approach.

$$T(SS) = |F_X| \tag{30}$$

Using DFS on the example in Figure 4 we can get the minimal walk set $\mathcal{S}$

$$\begin{aligned} \mathcal{S} = \{&w_1 = ((root, n_2), (n_7, n_{14})), \\ &w_2 = ((root, n_2), (n_7, n_{15}, n_{14}))\} \end{aligned} \tag{31}$$

having the neighborhood

$$f_{N_s}(X) = \{n_7, n_{14}, n_{15}\} \tag{32}$$

Even with randomly picking an adjacent node for the recursive call in DFS, having sufficiently big observation sample, we can achieve $T(\Sigma) \leq T(SS)$. However, by introducing probability correlated structure in the Σ-index DAG (12)(14), we expect that the most of the observed sample data will have statistical neighborhood close to the *root* node, leaving the bottom of the Σ-index DAG unexplored. To reduce randomness of DFS, we modified it to include proximity condition when picking the adjacent node for the recursive call, i.e., recursive calls to the adjacent nodes are not random but from closest to the most statistically distant nodes. Such approach requires pre-calculation of statistical distances for all adjacent nodes, which seems to be inefficient and costly.

The connection theorem 2.1 ensures that statistical distributions surrounding the observation X are connected into a single subgraph $G_\Sigma[f_{N_s}](X)$ for a sufficiently big θ_n, which can be used to create the modified DFS algorithm. Once we enter the subgraph $G_\Sigma[f_{N_s}](X)$, we need to find its bounds, which leaves us minimizing the walk w_p (20).

The modified DFS is given in Algorithm 1. The query function DFS_QUERY receives X for the input variable, n for the current node, v for the set of already visited nodes, r for

Algorithm 1 The Σ-index query function

```
1: function DFS_QUERY(X, n, v, r, c, G_Σ)
2:     l_1 ← {}, l_2 ← {}
3:     for (n, n_c) ∈ E_Σ(G_Σ) do
4:         if (n_c, ·) ∉ v then
5:             d = d_M(X, n_c)
6:             v_1 ← (n_c, d), v ← v ∪ {n_c}
7:             if d ≤ θ_n then
8:                 l_1 ← l_1 ∪ {v_1}, r ← r ∪ {n_c}
9:                 if d ≤ θ_c then
10:                    c ← c + 1
11:            else
12:                l_2 ← l_2 ∪ {v_1}
13:    for (n_p, n) ∈ E_Σ(G_Σ) do
14:        if (n_p, ·) ∉ v then
15:            d = d_M(X, n_p)
16:            v_1 ← (n_p, d), v ← v ∪ {n_p}
17:            if d ≤ θ_n then
18:                l_1 ← l_1 ∪ {v_1}, r ← r ∪ {n_p}
19:                if d ≤ θ_c then
20:                    c ← c + 1
21:    if l_1 ≠ ∅ then
22:        l_3 ← l_1
23:    else
24:        if r ≠ ∅ then return (r, v, c)
25:        l_3 ← l_2
26:    done ← false
27:    while ¬done do
28:        l_3 ← sort ascending(l_3)
29:        for (n_DFS, d_DFS) ∈ l_3 do
30:            (r, v, c) ←DFS_QUERY(X, n_DFS, v, r, c, G_Σ)
31:            if (l_1 = ∅ ∨ n = root) ∧ r ≠ ∅ then return (r, v, c)
32:        if r = ∅ ∧ l_3 = l_2 then
33:            l_3 ← l_2
34:        else
35:            done ← true
       return (r, v, c)
```

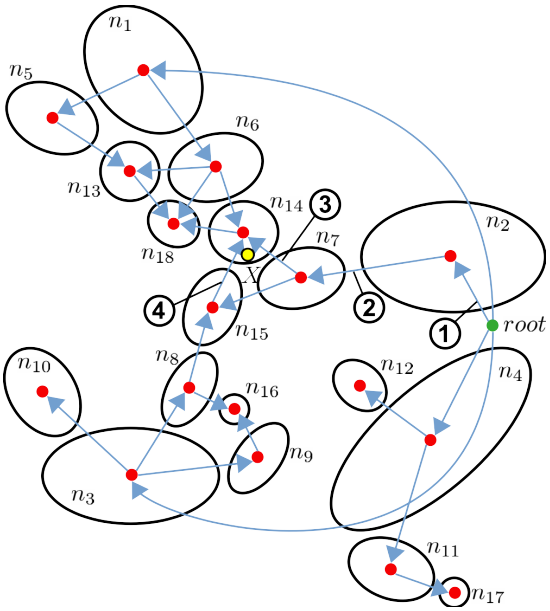


Figure 5: Statistical distributions in space for the Σ-index DAG in Figure 4.

the current set of neighborhood nodes $r \subseteq f_{N_s}(X)$, c for the number of already discovered outcome nodes in $f_C(X)$, and G_Σ. The query function maintains two sets

$$l_1, l_2 \subseteq N_\Sigma(n) \times \mathbb{R} \tag{33}$$

comprising ordered pairs made of a Σ-index DAG node and a statistical distance of the input variable X to the centroid of the node statistical distribution. l_1 is used to store nodes that have X in the statistical neighborhood, and l_2 nodes that do NOT have X in the statistical neighborhood. We process all adjacent nodes of $N_\Sigma(n) \setminus v$ that were not visited before. All child nodes $(n, n_c) \in E_\Sigma$ can be placed in both l_1 and l_2 sets. Parent nodes $(n_p, n) \in E_\Sigma$ can be placed only in l_1 if they have X in the statistical neighborhood, which is done to establish a bridge between two vertical paths. This allows us to explore the whole $G_\Sigma(f_{N_s}(X))$ as suggested by the connection theorem 2.1. Once we processed all non-visited adjacent nodes, we prioritize using l_1 over l_2 for the recursive calls. Before initiating recursive calls, we sort nodes ascending by statistical distance. This way we start recursive calls with the statistically closest node. If we did not found a set of neighborhood nodes using l_1, i.e. $r = \emptyset$, we retry recursive calls by using nodes in l_2. Finally, when we return back to the node that had $l_1 = \emptyset$ or back to the root node, and we found some nodes having X is the neighborhood, i.e., $r \neq \emptyset$, our query is completed and we can exit the modified DFS algorithm.

Having a connected subgraph $G_\Sigma[f_{N_s}(X)]$ is a prerequisite for maintaining the same precision when using Algorithm 1, comparing to the *sequential scan* approach. Having multiple disconnected subgraphs in $G_\Sigma[f_{N_s}(X)]$ (22) might significantly diminish Algorithm 1 computational cost reduction or precision. Either we spend more time to find all the disconnected subgraphs from $G_\Sigma[f_{N_s}(X)]$ and loose some computation cost reduction, or we find only one subgraph and loose precision.

Figure 5 shows statistical distributions from the Σ-index DAG in Figure 4 transformed back to the two-dimensional Euclidean space. Statistical distribution centroids in Figure 5 are con-

nected equally as the Σ-index DAG in Figure 4, to demonstrate the statistical neighborhood concept. The processing of the example in Figures 4 and 5 by Algorithm 1 is done in four steps marked in Figure 5. In the *root* node we calculate statistical distances of the input variable X to all top nodes $\{n_1, n_2, n_3, n_4\}$. The node n_2 is selected first, as the closest one, and the first recursive jump is made to the node n_2. In node n_2 we have a single child node n_7 belonging to the set l_1, leaving us no choice but to make the second recursive call to the node n_7. In node n_7 we calculate statistical distances to the child nodes $\{n_{14}, n_{15}\}$. Both child nodes are in the set l_1. The node n_{14} is picked as the nearest for the third recursive call. Processing the adjacent child node $\{n_{18}\}$ and parent nodes $\{n_6, n_{15}\}$ for the node n_{14} results in $l_1 = \{(n_{15}, d_{15})\}$. The final, fourth recursive call is done to the parent node n_{15}. After processing all adjacent nodes for n_{15}, we have $l_1 = \emptyset, l_2 \neq \emptyset$. Since we have found nodes $r = f_{N_s}(X)$ that have X in their neighborhood, due to restrictions in lines 24 and 31 of Algorithm 1 we cascadingly return to the *root* node. The final result of the processing is $r = \{n_7, n_{14}, n_{15}\}$.

3.2. Computational cost reduction

For the Σ-index DAG, we can divide the statistical distance calculations into two distinct categories, for nodes that do not have X in the statistical neighborhood $f_{N_p}(X)$ and for nodes that are having X in the statistical neighborhood $f_{N_s}(X)$. To decide the next node in a walk $w_i \in \mathcal{S}$, according to Algorithm 1, we need to calculate statistical distances for all adjacent nodes. All nodes in $f_{N_p}(X)$ and their adjacent nodes that are not in $f_{N_s}(X)$ are contributing to the wasted computations. Also there could be adjacent nodes to the nodes in $f_{N_s}(X)$ that are not in the X statistical neighborhood for which calculating were necessary to determine the bound of the subgraph $G_\Sigma[f_{N_s}(X)]$. The total number of wasted calculations can be expressed as

$$T_p^\Sigma(X) = |(f_{N_p}(X) \cup \bigcup_{n \in (f_{N_p}(X) \cup f_{N_s}(X))} N_{G_\Sigma}(n)) \setminus (f_{N_s}(X) \cup \{root\})| \tag{34}$$

where we did not count the *root* node. The total number of useful calculations is

$$T_s^\Sigma(X) = |f_{N_s}(X)| \tag{35}$$

The total number of calculations is

$$T^\Sigma(X) = T_p^\Sigma(X) + T_s^\Sigma(X) \leq T(SS) \tag{36}$$

The goal of the Σ-index neighborhood search is to achieve

$$T_p^\Sigma(X) < T(SS) - T_s^\Sigma(X) \tag{37}$$

The total computational cost reduction is

$$R^\Sigma(X) = \frac{T(SS) - T^\Sigma(X)}{T(SS)} \tag{38}$$

For the example in Figures 4 and 5 we have

$$\begin{aligned} T_p^\Sigma(X) = |\{root, n_2, n_7, n_{14}, n_6, n_{18}, n_{15}, n_8\} \setminus \{root, n_7, n_{14}, n_{15}\}| = 4 \\ T_s^\Sigma(X) = |\{n_7, n_{14}, n_{15}\}| = 3 \\ T^\Sigma(X) = 7 < T(SS) = 18 \end{aligned} \tag{39}$$

The total cost reduction is $R^{\Sigma}(X) = \frac{18-7}{18} = 61\%$, i.e., we have reduced the computational cost for 61% from the sequential scan approach. Finally, using (38) we can calculate a computational cost reduction distribution

$$\hat{f_{\Sigma}}(t) = \sum_{X_i \in (\Theta \setminus \Theta')} \mathbb{1}\{t = \lceil R^{\Sigma}(X_i) \rceil\} \tag{40}$$

which is a valuable information about computational cost reduction properties of the specific Σ-index DAG instance. If

$$\hat{f_{\Sigma}}(t > 0) > 0 \tag{41}$$

we have a Σ-index DAG capable of computational cost reduction for at least one statistical distribution in F_X, which is a minimal condition to call the Σ-index better than the sequential approach.

3.3. Adding new statistical distribution

In an evolving scenario we never know how much statistical distributions we have in the observed sample, or when are they going to appear during the sample processing. When we add a new statistical distribution to F_X, we need to add it to the Σ-index DAG structure as well. At the moment of addition, we are not aware of the neighborhood surrounding the newly added statistical distribution, therefore, we can consider it for an expensive operation, as we need to check its statistical relationship to all other pre-existing statistical distributions.

Algorithm 2 The Σ-index add procedure

```
1:  procedure ADD(n_new, G_Σ)
2:      (r, v, c) ←DFS_QUERY(μ(n_new), root, {}, {}, 0, G_Σ)
3:      for n ∈ r do
4:          if |P(n_new)| < |P(n)| then
5:              E_Σ ← E_Σ ∪ (n, n_new)
6:          else
7:              E_Σ ← E_Σ ∪ (n_new, n)
8:      for n ∈ N_Σ \ r do
9:          d = d_M(μ(n), n_new)
10:         if d ≤ θ_n then
11:             if |P(n_new)| < |P(n)| then
12:                 E_Σ ← E_Σ ∪ (n, n_new)
13:             else
14:                 E_Σ ← E_Σ ∪ (n_new, n)
15:     if ∄(·, n_new) ∈ E_Σ then
16:         E_Σ ← E_Σ ∪ (root, n_new)
```

In Algorithm 2 we use the existing query algorithm in Algorithm 1 to determine nodes that have n_{new} in their statistical neighborhood. We connect n_{new} to the neighbor statistical distributions based on the rules in (14) and (16). We also need to take in consideration that

adding new edges in E_Σ does not create a cycle, as defined in Section 2. In the second step, we additionally check all nodes that were not returned by the query algorithm, since there still could be a statistical neighborhood whose centroid is in the statistical neighborhood of the new statistical distribution n_{new}. Finally, if there is no statistical distribution in the neighborhood that has bigger (or equal) population, we place n_{new} directly under the *root* node.

3.4. Complete update for significant evolutionary changes

Evolutionary changes described in Section 2.1 require re-checking and update of the Σ-index DAG structure. Once we detect a significant statistical distribution variance change, or a centroid move, we need to refresh this statistical distribution. The magnitude of the *significant* statistical distribution changes is arbitrary, which is the reason why we can perform a complete update of the Σ-index DAG structure. The complete update is an expensive operation, as we cannot rely in the query algorithm to re-check the statistical distribution neighborhood, since the local DAG structure around the refreshed node might be inaccurate due to the evolutionary change that happened.

Algorithm 3 The Σ-index complete update procedure

procedure COMPLETE_UPDATE(n_{ch}, G_Σ)
  $l_1 \leftarrow \{\}$
  for $n \in N_\Sigma \setminus \{N_\Sigma(n_{ch}) \cup root\}$ **do**
    ▷ We check only nodes that are not adjacent
    $d_1 \leftarrow d_M(\mu(n), n_{ch}), d_2 \leftarrow d_M(\mu(n_{ch}), n)$
    if $d_1 \leq \theta_n \vee d_2 \leq \theta_n$ **then**
      if $|\mathcal{P}(n_{ch})| < |\mathcal{P}(n)|$ **then**
        $E_\Sigma \leftarrow E_\Sigma \cup (n, n_{ch})$
      else
        $E_\Sigma \leftarrow E_\Sigma \cup (n_{ch}, n)$
        ▷ The node n moved into the neighborhood
      $l_1 \leftarrow l_1 \cup \{n\}$
  for $n \in N_\Sigma(n_{ch}) \setminus l_1$ **do** ▷ We check old adjacent nodes
    $d_1 \leftarrow d_M(\mu(n), n_{ch}), d_2 \leftarrow d_M(\mu(n_{ch}), n)$
    if $d_1 > \theta_n \wedge d_2 > \theta_n$ **then**
      $E_\Sigma \leftarrow E_\Sigma \setminus \{(n, n_{ch}), (n_{ch}, n)\}$
      ▷ The node n moved out of the neighborhood
    else
      if $|\mathcal{P}(n_{ch})| < |\mathcal{P}(n)|$ **then**
        $E_\Sigma \leftarrow E_\Sigma \setminus (n_{ch}, n), E_\Sigma \leftarrow E_\Sigma \cup (n, n_{ch})$
      else
        $E_\Sigma \leftarrow E_\Sigma \setminus (n, n_{ch}), E_\Sigma \leftarrow E_\Sigma \cup (n_{ch}, n)$
  if $\nexists(\cdot, n_{ch}) \in E_\Sigma$ **then**
    $E_\Sigma \leftarrow E_\Sigma \cup (root, n_{ch})$

Algorithm 3 is divided into two loops. In the first loop we check whether the nodes that are currently NOT adjacent to the refreshed node n_{ch} moved into its statistical neighborhood. We add edges based on the population size for all nodes that have moved into the n_{ch} sta-

tistical neighborhood. We also maintain a set l_1 of such nodes, to avoid duplicate statistical distance calculations in the second loop. The second loop is checking whether the existing adjacent nodes, without those added in the first loop, have moved out of the n_{ch} statistical neighborhood, and removes all edges for such nodes. The second loop also checks whether the population sizes of adjacent nodes are correct w.r.t. edge directions, and corrects edge directions in case (14) and (16) are not respected. Finally, if there is no statistical distribution in the neighborhood of n_{ch} that has bigger (or equal) population, we place n_{ch} directly under the *root* node.

3.5. Incremental update

Since the complete update described in the previous Section is an expensive operation, i.e., its complexity is the same as the *sequential scan* querying for a single input observation, the main motivation is to reduce time needed to update the Σ-index DAG for each input observation. In all statistical classification applications, it is customary to perform a statistical classification first and then to update the underlying model. In our case, we perform a query that returns the whole neighborhood of the input observation X. Once we choose a classifying statistical distribution n_{ch} from the neighborhood returned by the query, the rest of the neighbor statistical distributions can be used for an update. Since we already know the neighborhood, we can omit any additional calculations in the update. This can work only when there are no *significant* changes in the classifying statistical distribution, i.e., only for a small number of input observations, preferably for each individual input observation. This represents is an incremental update.

Algorithm 4 The Σ-index incremental update procedure

1: **procedure** INC_UPDATE($n_{ch}, G_\Sigma, f_{N_s}(X)$)
2: **for** $n \in f_{N_s}(X) \setminus n_{ch}$ **do** ▷ We check the neighbor nodes returned by the query
3: **if** $|\mathcal{P}(n_{ch})| < |\mathcal{P}(n)|$ **then**
4: $E_\Sigma \leftarrow E_\Sigma \setminus (n_{ch}, n), E_\Sigma \leftarrow E_\Sigma \cup (n, n_{ch})$
5: **else**
6: $E_\Sigma \leftarrow E_\Sigma \setminus (n, n_{ch}), E_\Sigma \leftarrow E_\Sigma \cup (n_{ch}, n)$
7: **for** $n \in N_\Sigma(n_{ch}) \setminus f_{N_s}(X)$ **do** ▷ We check nodes that ceased to be the neighbor nodes
8: $E_\Sigma \leftarrow E_\Sigma \setminus \{(n, n_{ch}), (n_{ch}, n)\}$
9: **if** $\nexists(\cdot, n_{ch}) \in E_\Sigma$ **then**
10: $E_\Sigma \leftarrow E_\Sigma \cup (root, n_{ch})$

Algorithm 4 uses the set of neighbor nodes $f_{N_s}(X)$ (21) as an input parameter. First, we iterate through the neighbor nodes returned by the query $f_{N_s}(X)$, to ensure that the Σ-index DAG has all the necessary edges for this neighborhood. After that, we iterate through old neighborhood nodes $N_\Sigma(n_{ch}) \setminus f_{N_s}(X)$ and remove them from the Σ-index DAG, as these statistical distributions are not in the n_{ch} neighborhood anymore. Finally, if there is no statistical distribution in the neighborhood of n_{ch} that has bigger (or equal) population, we place n_{ch} directly under the *root* node. The advantage of such approach is a very low cost for each incremental update. However, as we update the Σ-index DAG only locally, there is a possibility that incremental updates lead to a disconnected neighborhood subgraph $G_\Sigma(f_{N_s}(X))$, which can lower Σ-index query precision.

3.6. Removing statistical distribution

In some cases, such as decay and removal in data stream clustering algorithms (Gama 2010; Silva, Faria, Barros, Hruschka, de Carvalho, and Gama 2013), we want to remove an existing statistical distribution as obsolete.

Algorithm 5 The Σ-index removal procedure

1: **procedure** REMOVE(n_{rem}, G_Σ)
2: $E_\Sigma \leftarrow E_\Sigma \setminus (\cdot, n_{rem})$ ▷ Remove parents
3: **for** $n_c \in N_\Sigma(n_{rem})$ **do**
4: $E_\Sigma \leftarrow E_\Sigma \setminus (n_{rem}, n_c)$ ▷ Remove child n_c
5: **if** $\nexists(\cdot, n_c) \in E_\Sigma$ **then**
6: $E_\Sigma \leftarrow E_\Sigma \cup (root, n_c)$ ▷ If child n_c has no parent, add it to the root
7: $N_\Sigma \leftarrow N_\Sigma \setminus \{n_{rem}\}$

In Algorithm 5 we first remove all edges to parents of the removed node n_{rem}. Next, in a loop we iterate through child nodes of n_{rem} to check whether there will be a child node n_c that looses all parents after the edge (n_{rem}, n_c) removal, which needs to be placed under the *root* node. Finally, we remove the node n_{rem} from the Σ-index DAG.

4. Usage and experimental verification

Algorithms 1 to 5 were implemented in the C++ programming language and added to SHC, which was originally implemented to use the sequential scan approach. The Σ-index and SHC are interfaced to R using **Rcpp** package (Eddelbuettel and Francois 2011). For the stream clustering infrastructure, we used the **stream** package (version >=1.4-0) (Hahsler, Bolaños, and Forrest 2017). Finally, we created **SHClus** R package that comprises both SHC and Σ-index implementations.

4.1. Manual usage

First, we create an empty S3 Σ-index object with initial thresholds

```
R> si <- SigmaIndex(theta = 3.2, neighborhood = 9.6)
```

where $\theta_c = 3.2$ and $\theta_n = 9.6$. The initialized object has now only the ROOT node. Next, we manually add some statistical distributions. The covariance matrix of the added statistical distribution must be invertible, as the Σ-index is inverting the covariance matrix before use.

```
R> covariance1 <- matrix(data=c(0.8, 0, 0, 0.8), nrow=2, ncol=2)
R> covariance1

     [,1] [,2]
[1,]  0.8  0.0
[2,]  0.0  0.8

R> addPopulation(si, "1", c(0.5,0.5), covariance1, 800)
R> covariance2 <- matrix(data=c(0.5, 0, 0, 0.5), nrow=2, ncol=2)
```

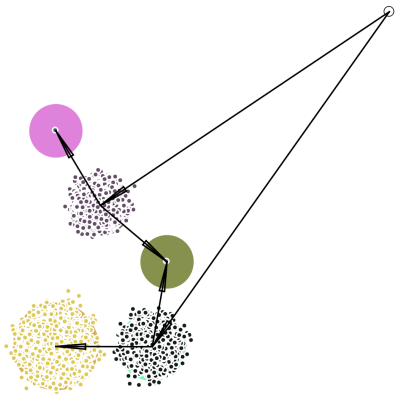


Figure 6: Σ-index DAG as the result of the manual creation.

```
R> addPopulation(si, "2", c(7,0.5), covariance2, 500)
R> covariance3 <- matrix(data=c(0.4, 0, 0, 0.4), nrow=2, ncol=2)
R> addPopulation(si, "3", c(3.5,10.0), covariance3, 400)
```

and we add some outliers

```
R> out_covariance <- matrix(data=c(0.3, 0, 0, 0.3), nrow=2, ncol=2)
R> addPopulation(si, "4", c(8.0,6.2), out_covariance, 1)
R> addPopulation(si, "5", c(0.5,15.0), out_covariance, 1)
```

Figure 6 represents the Σ-index for the previous manual example. We can retrieve the total number of nodes $|N_\Sigma|$ in the Σ-index DAG by

```
R> getTotalPopulations(si)

[1] 6
```

Additionally, we can obtain back details for a specific statistical distribution

```
R> pop <- getPopulations(si)
R> names(pop)

[1] "5" "4" "3" "2" "1"

R> outlier <- pop$"4"
R> outlier$mean

[1] 8.0 6.2

R> outlier$icovariance

         [,1]     [,2]
[1,] 3.333333 0.000000
[2,] 0.000000 3.333333
```

Querying

We define a small set of observations for statistical classification.

```
R> query_data <- data.frame(X=c(0.37,6.5,8.05),Y=c(0.505,0.4,6.3))
R> query_data

     X     Y
1 0.37 0.505
2 6.50 0.400
3 8.05 6.300
```

A query in the previously defined Σ-index object `si`, defined in Algorithm 1, can be done the following way

```
R> res <- queryDataPoints(si, query_data)
```

From the result `res`, we can retrieve the outcome set of nodes $f_C(X)$ for each input data point as

```
R> unlist(res[[1]]$outcome)

        1
0.1454519
```

and the remainder of the nodes $f_{N_s}(X) \setminus f_C(X)$ that have `query_data[[1]]` in the neighborhood as

```
R> unlist(res[[1]]$neighborhood)

       2
9.376239
```

The results are indicating the Mahalanobis distance from the centroid of the *outcome* or *neighborhood* nodes. One query can return multiple nodes in each category for each input data point. For example

```
R> res <- queryDataPoints(si, c(3.75,0.5))
R> unlist(res[[1]]$outcome)

NULL

R> unlist(res[[1]]$neighborhood)

       1        2
3.633610 4.596194
```

Statistics

The Σ-index object collects statistics for each query call. Before the query, we can reset statistical counters by using

```
R> resetStatistics(si)
```

Then we can initiate the query

```
R> res <- queryDataPoints(si, c(0.9,0.7))
R> unlist(res)

     outcome.1 neighborhood.2
      0.500000       8.631338
```

and obtain the statistical counters

```
R> unlist(getStatistics(si))

              totalCount          classifiedNodes
                     4.0                      2.0
            missedNodes          sequentialNodes
                     2.0                      5.0
computationCostReduction
                     0.2
```

We obtained only 20% computational cost reduction. The reason for this is a small, shallow and interconnected Σ-index DAG. If we move outlier 4 away from the population 2, we gain some additional computational cost reduction.

```
R> si <- SigmaIndex(theta = 3.2, neighborhood = 9.6)
R> covariance1 <- matrix(data=c(0.8, 0, 0, 0.8), nrow=2, ncol=2)
R> addPopulation(si, "1", c(0.5,0.5), covariance1, 800)
R> covariance2 <- matrix(data=c(0.5, 0, 0, 0.5), nrow=2, ncol=2)
R> addPopulation(si, "2", c(7,0.5), covariance2, 500)
R> covariance3 <- matrix(data=c(0.4, 0, 0, 0.4), nrow=2, ncol=2)
R> addPopulation(si, "3", c(3.5,10.0), covariance3, 400)
R> out_covariance <- matrix(data=c(0.3, 0, 0, 0.3), nrow=2, ncol=2)
R> addPopulation(si, "4", c(6.5,15.0), out_covariance, 1)
R> addPopulation(si, "5", c(0.5,15.0), out_covariance, 1)
R> res <- queryDataPoints(si, c(0.9,0.7))
R> unlist(getStatistics(si))

              totalCount          classifiedNodes
                     3.0                      2.0
            missedNodes          sequentialNodes
                     1.0                      5.0
computationCostReduction
                     0.4
```

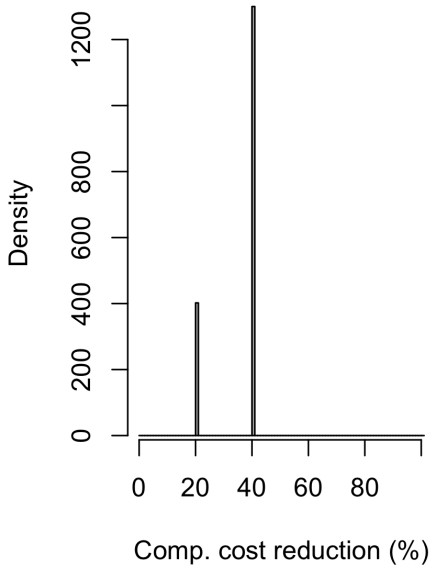


Figure 7: Σ-index computational cost reduction histogram.

Histogram

Finally, the Σ-index can statically calculate its ability to reduce computational costs in a from of histogram, as defined in (40). This is shared through a computational cost reduction histogram.

```
R> hist <- getHistogram(si)
R> hist
```

```
     0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19  20 21 22 23
[1,] 0 0 0 0 0 0 0 0 0 0  0  0  0  0  0  0  0  0  0  0 402  0  0  0
     24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39   40 41 42 43 44
[1,]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 1300  0  0  0  0
     45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65
[1,]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
     66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86
[1,]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
     87 88 89 90 91 92 93 94 95 96 97 98 99 100
[1,]  0  0  0  0  0  0  0  0  0  0  0  0  0   0
```

Figure 7 shows the distribution of the

```
R> sum(hist)
```

```
[1] 1702
```

population elements through the computational cost reduction. We can see that 402 elements belong to the 20% reduction bin and 1300 elements belong to the 40% reduction bin. We can expect that most of the future query hits will have 40% of the computational cost reduction.

4.2. Statistical distribution update

We can update registered statistical distributions by adding observations to them. Update by adding one observation is done on the inverse covariance matrix of the statistical distribution using Sherman-Woodbury-Morrison formula (Sherman and Morrison 1950; Woodbury 1950). Adding one observation would probably not change the updated population significantly to affect the Σ-index DAG structure, for which we can use the incremental update defined in Algorithm 4.

The complete update

We create 2 new observations as

```
R> new_observ <- data.frame(X=c(0.6,6.9),Y=c(0.4,0.55))
R> new_observ
```

```
    X    Y
1 0.6 0.40
2 6.9 0.55
```

and then we add them to statistical distributions `"1"` and `"3"`

```
R> ids <- c("1","3")
R> addDataPoints(si, ids, new_observ)
```

Calling the method `addDataPoints` does the complete update for each observation, as defined in Algorithm 3. This is quite costly, since the *sequential scan* is preformed for each added observation.

Incremental update

In a streamlined process, we usually perform queries before we decide to update some of the statistical distributions, or to create outliers. By performing the query, we already did the calculations for the input observations, so we know their neighborhood $f_{N_s}(\cdot)$. Using this information, we can update only portions of the Σ-index DAG, without any additional computation costs. For this, we need to perform the query first, and then to use these results to update the Σ-index DAG. The incremental update, as defined in Algorithm 4, takes the outcome statistical distribution to be the incrementally updated statistical distribution. When using the incremental update, it is best to do it observation by observation, so that updates affect the subsequent queries.

First, we create two observations for the updates.

```
R> new_observ <- data.frame(X=c(0.62,6.91),Y=c(0.41,0.551))
R> new_observ
```

```
     X     Y
1 0.62 0.410
2 6.91 0.551
```

Then we process these two observations by performing the query first, and then the incremental update using the query results.

```
R> for(r in 1:nrow(new_observ)) {
+  query_res <- queryDataPoints(si, new_observ[r,])
+  addDataPointsInc(si, new_observ[r,], query_res)
+  }
```

The advantage of such approach is that we spend only those computation needed to perform the query.

4.3. Data stream usage

We can use data stream generators from the **stream** package to generate a synthetic case that can be directly converted to the S3 Σ-index object. We can generate a bigger number of clusters and outliers to make the example more complex.

```
R> dsd <- DSD_Gaussians(k=50,outliers=50,separation_type="Mahalanobis",
+    space_limit=c(0,150),separation=4,variance_limit=8,
+    outlier_options=list(outlier_horizon=20000))
```

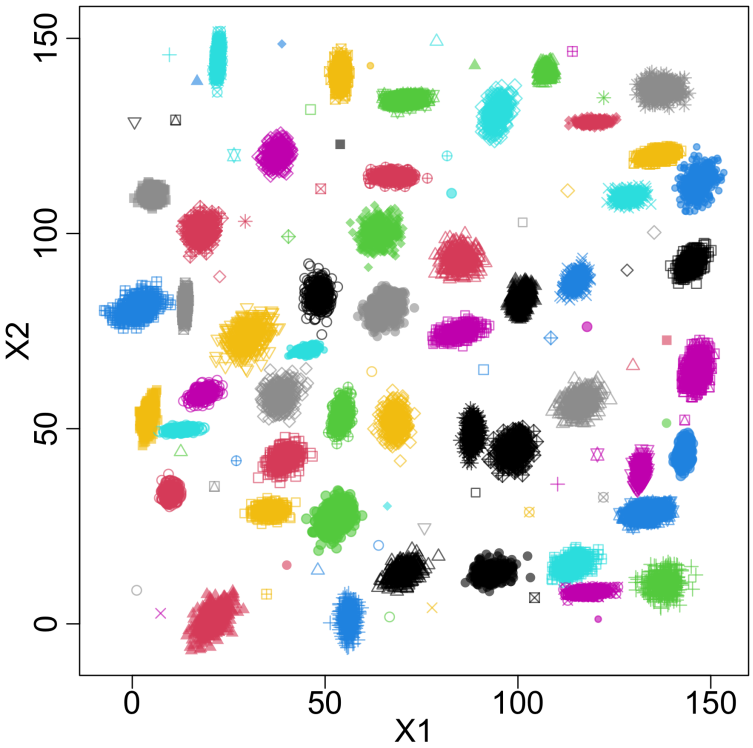
Figure 8a shows a synthetic Gaussian example generated by the `DSD_Gaussians` data stream generator. We can convert definitions from this data stream generator into an S3 Σ-index object

```
R> si <- convertFromDSD(dsd, total_elements = 20000, theta = 3.2,
+  neighborhood = 9.6)
```

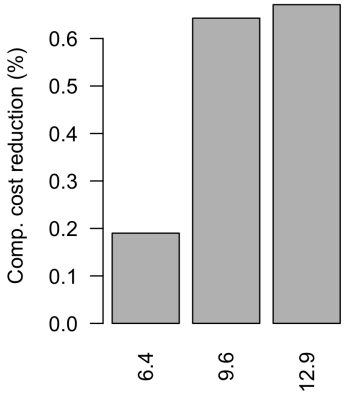
We start by generating and querying the starting 200000 observations and obtaining the statistics for the query

```
R> resetStatistics(si)
R> res1 <- queryDataPoints(si, get_points(dsd, 20000))
R> stats1 <- getStatistics(si)
R> unlist(stats1)
```

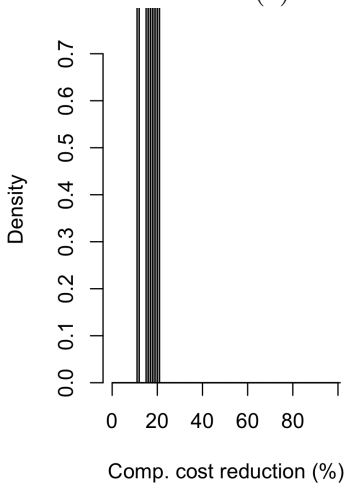
```
              totalCount          classifiedNodes
            714214.0000               43072.0000
            missedNodes          sequentialNodes
            658089.0000             2000000.0000
computationCostReduction
                  0.6429
```



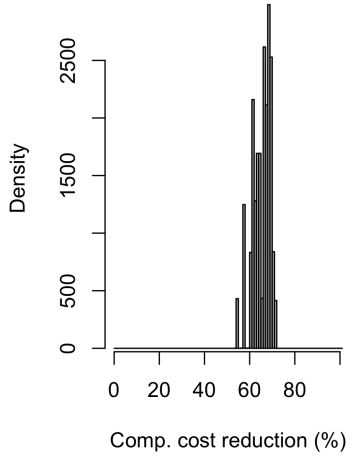
(a) Data stream plot



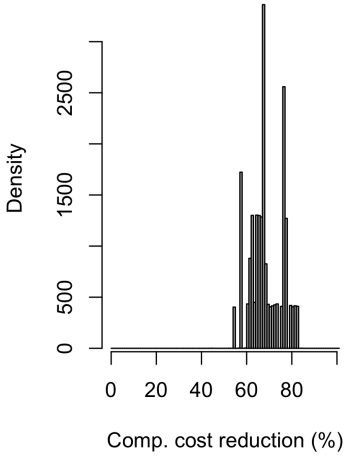
(b) Comp. cost reduction



(c) $\theta_n = 6.4$



(d) $\theta_n = 9.6$



(d) $\theta_n = 12.9$

Figure 8: Data stream plot, Σ-index computational cost reduction, and histograms for $\theta_n \in \{6.4, 9.6, 12.9\}$.

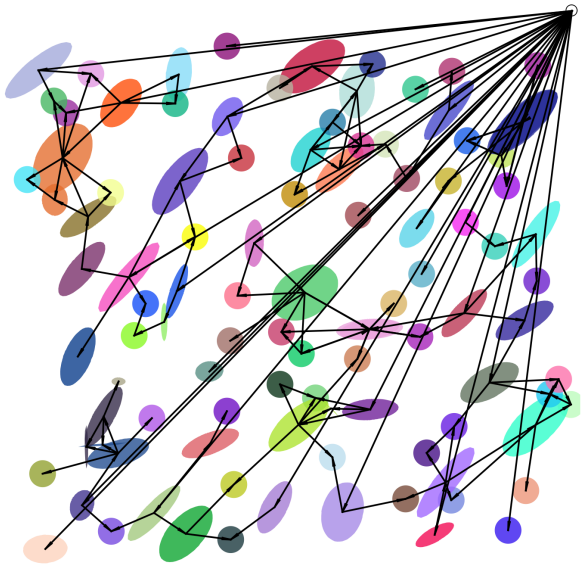


Figure 9: An example of SHC model and related Σ-index.

Using distinct neighborhood thresholds $\theta_n \in \{6.4, 9.6, 12.9\}$, we obtain different computational cost reductions. Figure 8b contains average comp. cost reduction for all three settings. Figures 8c-d contains histograms for $\theta_n = 6.4$, $\theta_n = 9.6$, and $\theta_n = 12.9$ respectively. We can draw a conclusion that the computational cost reduction gets higher with neighborhood threshold θ_n. Such claim must be cross-verified with processing times for each of the setting in Figure 8, as the overall processing time includes Σ-index updating times as well, which is also a limiting factor. Intuitively, the more interconnected Σ-index is, the higher update time should be.

4.4. Synthetic statistical test in clustering

In the previous test we used data stream definitions (`DSD_Gaussians`, **stream** package) to predefine statistical distribution in the Σ-index. In this test we plan to use the SHC algorithm between data stream definition and Σ-index. An example of the SHC model and related Σ-index DAG can be seen in Figure 9.

Test 1

For this test we synthetically generated 50 populations and 50 outliers in a sample made of 20000 observations.

```
R> dsd <- DSD_Gaussians(k=50,outliers=50,separation_type="Mahalanobis",
+                       separation=4,space_limit=c(0,150),variance_limit=8,
+                       outlier_options=list(outlier_horizon=20000))
```

To check for possible accuracy degradation, we consulted corrected Rand Index (CRI) (Desgraupes 2017) for both sequential scan and Σ-index approach. Next, we construct an SHC instance as

```
R> shc_c <- DSC_SHC.man(2, 3.5, 0.9, cbVarianceLimit = 0, cbNLimit = 0,
+  decaySpeed = 0, sharedAgglomerationThreshold = 100, sigmaIndex = T,
+  sigmaIndexNeighborhood = 3)
```

We can perform an evaluation of the SHC for the generated data stream as

```
R> evaluate(shc_c, dsd, n=20000, type="macro", measure=c("crand",
+  "outlierjaccard"), single_pass_update=T)

Evaluation results for macro-clusters.
Points were assigned to micro-clusters.

         cRand OutlierJaccard
        0.9990         0.8197
```

and then retrieve the Σ-index computational cost reduction through SHC for the previously executed queries.

```
R> shc_c$RObj$getComputationCostReduction()

[1] 0.7718
```

Figures 10a-c shows histograms for all three neighborhood multipliers $nm \in \{2, 3, 4\}$. Figures 10d-i shows the comparation between the *sequential scan* approach and Σ-index. In Figure 10d we can see the corrected Rand for all four settings. There is no precision degradation when using Σ-index. In Figure 10e we can see the number of statistical distributions for whose we needed to calculate the Mahalanobis distance. The worst case scenario when using the Σ-index requires significantly less calculations than the *sequential scan* approach. This pattern is inversely reflected in the computational cost reduction in Figure 10f. We can see that the setting `nm = 4` is slightly better than `nm = 3`. Query and processing times in Figures 10g and 10h follow the same pattern with some multiplier C_1, which depends on the clustering algorithm, SHC in this case. Finally, the update time in Figure 10i is obviously the highest in the setting `nm = 4`, since the Σ-index DAG is the most interconnected of all settings. The obvious conclusion is that times relate as $query + update = processing + C_2$. Figures 10a and 10f show a small difference since histograms are calculated in the invocation moment. Applied to SHC, Figure 10f shows the whole data stream evolution, while Figure 10a shows only the final result.

Test 2

Another test would be synthetically generated 50 populations and 50 outliers in a sample made of 20000 observations that are not well-separated, which could be challenging for any clustering algorithm.

```
R> dsd <- DSD_Gaussians(k=50,outliers=50,separation_type="Mahalanobis",
+      separation=2,space_limit=c(0,90),variance_limit=8,
+      outlier_options=list(outlier_horizon=20000,
+                           outlier_virtual_variance=0.3))
```

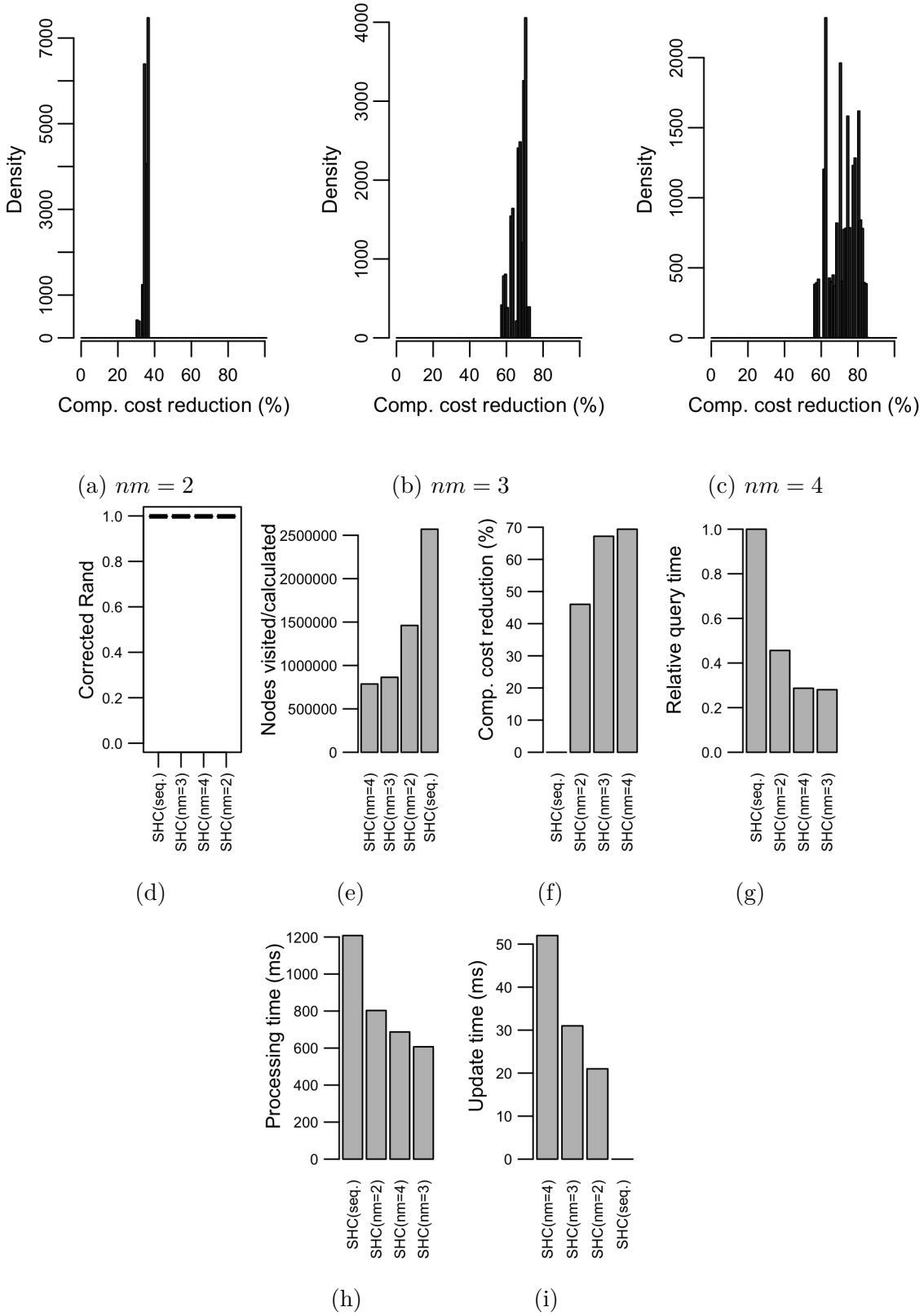


Figure 10: SHC comparative results between *sequential scan* and Σ-index for well-separated data stream.

We defined a bit differently configured SHC instance comparing to the test 1.

```
R> shc_c <- DSC_SHC.man(2, 3.2, 0.3, cbVarianceLimit = 0, cbNLimit = 0,
+  decaySpeed = 0, sharedAgglomerationThreshold = 700, sigmaIndex = T,
+  sigmaIndexNeighborhood = 3)
```

Evaluating CRI and Outlier Jaccard gives the results that are lower than in the test 1.

```
R> evaluate(shc_c, dsd, n=20000, type="macro", measure=c("crand",
+  "outlierjaccard"), single_pass_update=T)

Evaluation results for macro-clusters.
Points were assigned to micro-clusters.

         cRand OutlierJaccard
        0.8525         0.3146
```

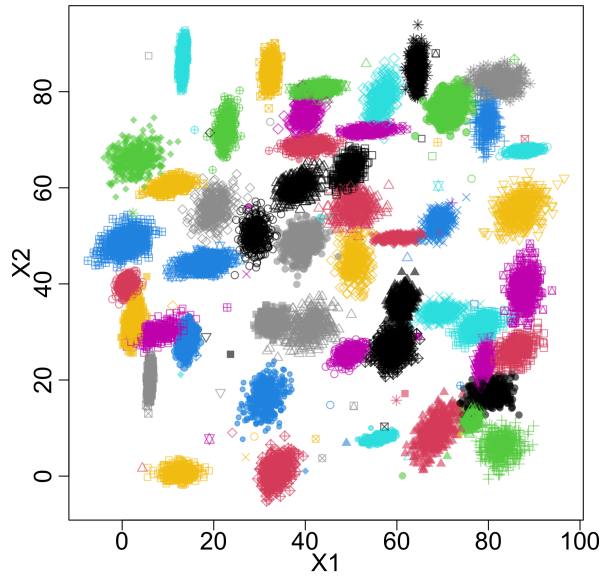
The results that can be seen in Figure 11 are similar to the results in Figure 10. Obviously, SHC has somewhat poorer results looking at the CRI and Outlier Jaccard indices, which is not a surprise since the generated data stream is not well-separated and concepts are significantly overlapping, as seen in Figure 11a. Figures 11b and 11c also indicate that there are no significant precision distinction between the *sequential scan* approach and using Σ-index. In Figures 11d-f we can see histograms for the Σ-indices using $nm \in \{2, 3, 4\}$. We can observe high computational cost reduction even for $nm = 2$, which is obviously due to the low separation distance and overlapping concepts. This is also reflected in Figures 11g-i, which show similar drop in query and processing times for all neighborhood multiplier settings.

4.5. Sensors dataset

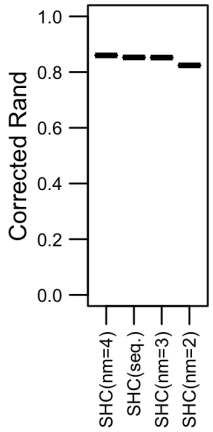
As the real-life testing data stream, we have used the Intel Berkeley Laboratory sensor data[1]. The test setting is the same as in (Krleža *et al.* 2020). We have chosen this data stream because of low dimensionality and statistically close sensor readings. Also, it is an example of a data streaming environment representing a constantly evolving endless stream of observations, which we explained in Sections 2.1, 3.3, 3.4, 3.5, and 3.6. The whole sensors data stream contains around 2 million sensor readings.

In this test we used only two statistical neighbor multipliers $nm \in \{3, 6\}$. Figure 12 shows the results of the sensors data stream processing. In Figure 12a we can see that there is no accuracy degradation between sequential scan and Σ-index settings. Figure 12b shows total processing times needed aggregated by 1000 sensor readings, which obviously drops when using Σ-index. There is an additional small processing time drop between $nm = 3$ and $nm = 6$. In Figure 12c we can see the number of calculations for all three SHC settings. Again, there is a visible drop when using Σ-index. Similar trend can be seen with query times in Figure 12d. Updating times and computation cost reduction were observed only for Σ-index SHC settings, and can be seen in Figures 12e and 12f. The updating time is expectedly higher for deeper Σ-index structure. Interestingly, computation cost reduction is lower but

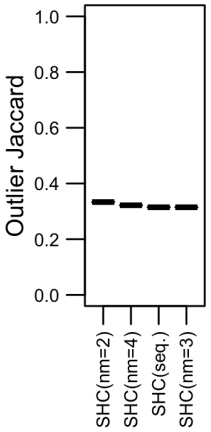
[1] http://db.csail.mit.edu/labdata/labdata.html



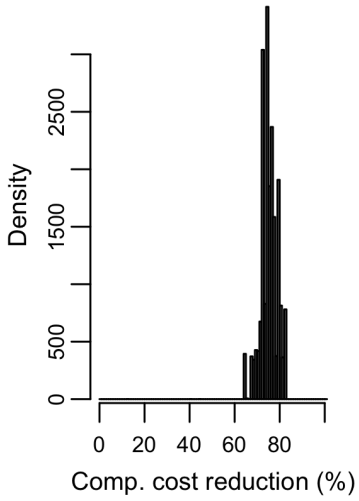
(a)



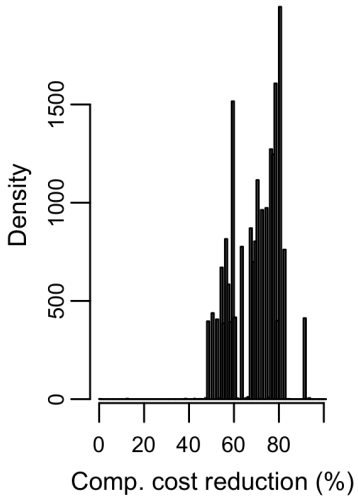
(b)



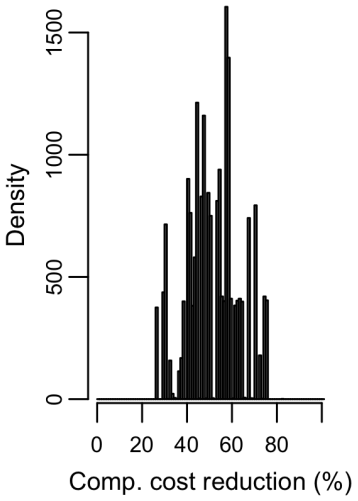
(c)



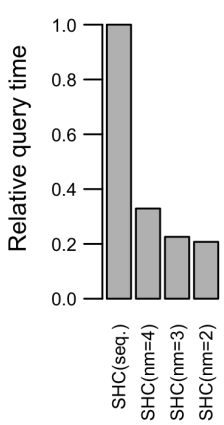
(d) $nm = 2$



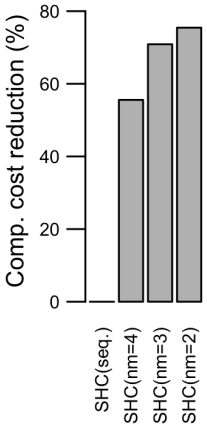
(e) $nm = 3$



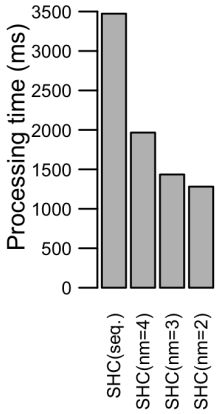
(f) $nm = 4$



(g)



(h)



(i)

Figure 11: Not well-separated SHC comparative results.

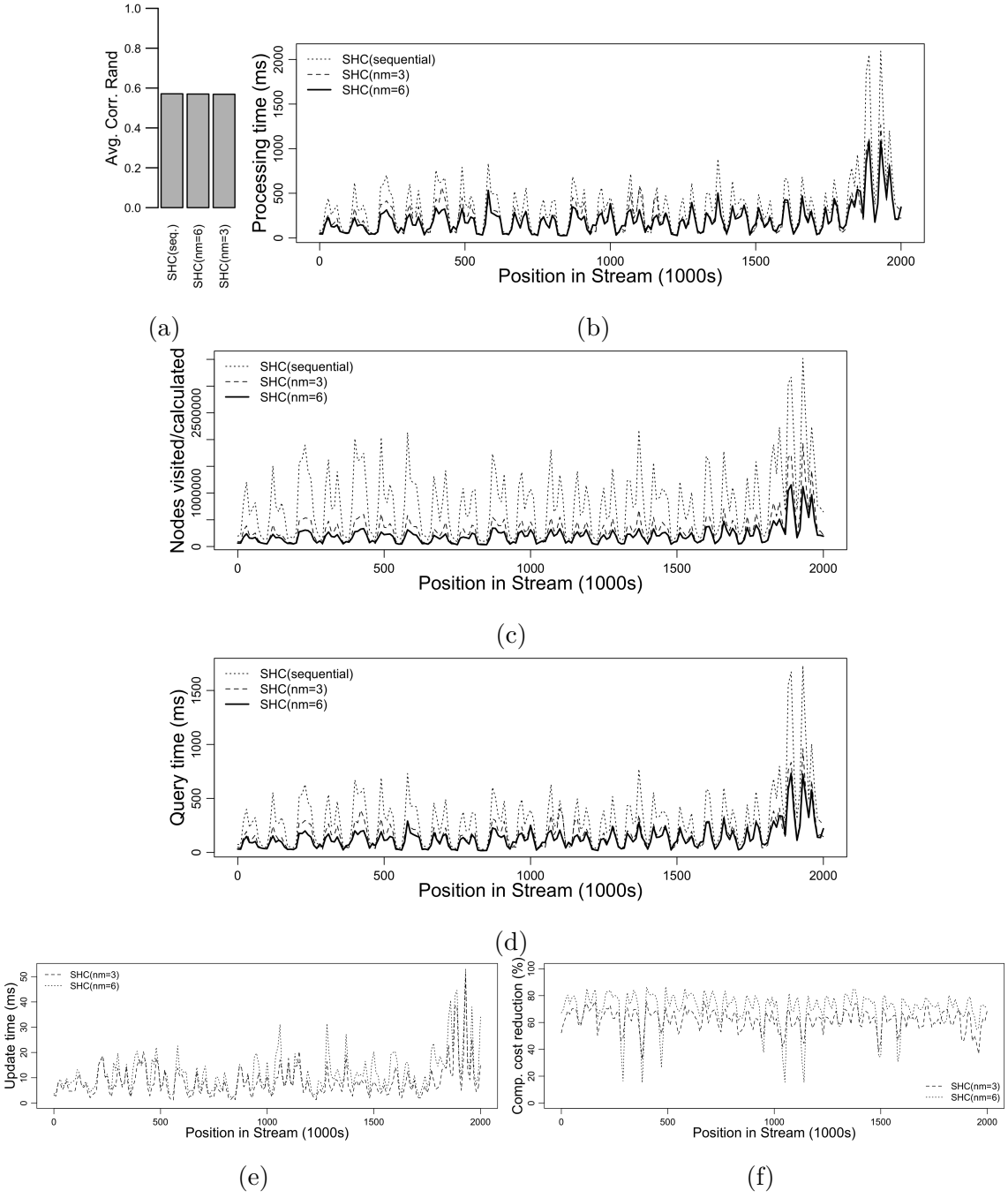


Figure 12: Comparative results of SHC processing the Sensor dataset using the sequential scan approach and Σ-index.

more stable for smaller neighborhood multiplier which we attribute to the high number of outliers that appear in daily CRI drops.

5. Discussion

In the previous Section we have experimentally assessed the computational cost reduction provided by the Σ-index, which still leaves the question of theoretical minimum for which the Σ-index offers a computational cost reduction in comparison to the sequential scan approach.

Theorem 5.1 *Any Σ-index DAG for $|F_X| < 3$ cannot achieve computational cost reduction $T^{\Sigma}(X) = T(SS)$.*

For $|F_X| = 1, |N_\Sigma| = 2$, the proof is trivial. There can be only $f_{N_p}(X) = \{root\}$ and $f_{N_s}(X) = N_\Sigma \setminus \{root\}$, which results in $T_p^{\Sigma}(X) = 0$ according to (34) and $T_s^{\Sigma}(X) = 1$ according to (35).

For $|F_X| = 2, |N_\Sigma| = 3$, we have a clique such that

$$\forall n \in N_\Sigma : N_\Sigma(n) \neq \emptyset \tag{42}$$

Minimal conditions are $|f_{N_s}(X)| = 1$ and $f_{N_p}(X) = \{root\}$, which according to (34) gives

$$\bigcup_{n \in (\{root\} \cup f_{N_s}(X))} N_\Sigma(n) = N_\Sigma \setminus (f_{N_s}(X) \cup \{root\}) \tag{43}$$

resulting in

$$T_p^{\Sigma}(X) = 1, T_s^{\Sigma}(X) = 1 \Rightarrow T_p^{\Sigma}(X) + T_s^{\Sigma}(X) = T(SS) \tag{44}$$

Theorem 5.2 *There is a minimal threshold θ_n for which any Σ-index having $|F_X| \geq 3$ has reduced computational cost comparing to the sequential scan approach.*

The worst case is to have a flat Σ-index DAG, such that for the minimal $f_{N_p}(X) = \{root\}$

$$\{root\} \cup N_\Sigma(root) = N_\Sigma \tag{45}$$

Such an example can be seen in Figure 13. This results in $T_p^{\Sigma}(X) = |N_\Sigma| - 1$. Generally speaking, Σ-index DAG having $d(G_\Sigma) = 1$ will not produce any computational cost reduction comparing to the sequential scan approach. According to (40), such Σ-index will produce

$$\hat{f_\Sigma}(t > 0) = 0 \tag{46}$$

The solution to this problem is to produce a Σ-index DAG that has the maximal depth $d(G_\Sigma) \geq 2$. For achieving this, we need to find a minimal θ_n that satisfies

$$\theta_n = \min_{n_i, n_j \in F_X : n_i \neq n_j} d_M(\mu(n_i), n_j) \tag{47}$$

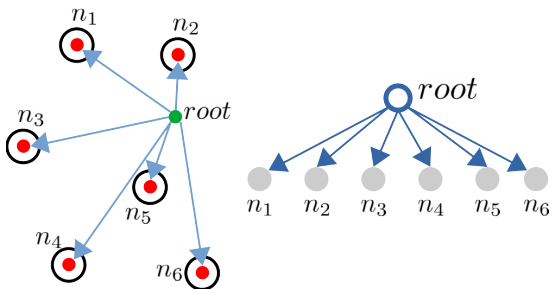


Figure 13: An example of a flat Σ-index scenario.

In such case we have

$$\exists n_i \in N_\Sigma(root) \exists n_j \in N_\Sigma(n_i) : (n_i, n_j) \in E_\Sigma \tag{48}$$

Regarding the minimal $f_{N_p}(X) = \{root\}$, we have at least one *root* adjacent node

$$\exists n_k \in N_\Sigma(root) : n_i \neq n_k \tag{49}$$

that produces

$$\begin{aligned} {w_p}' = (root), w_s = (n_k) \\ f_{N_p}(X) = \{root\}, f_{N_s}(X) = \{n_k\} \end{aligned} \tag{50}$$

since

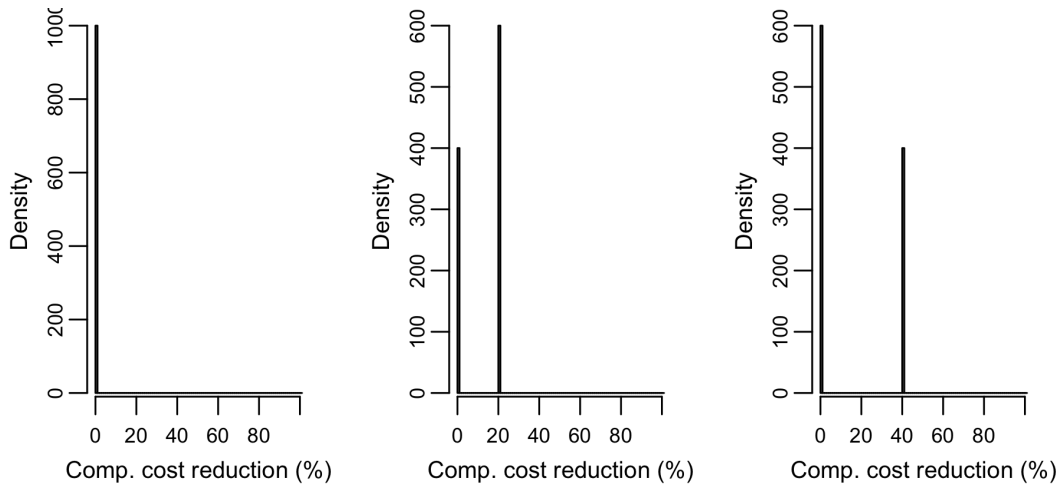
$$n_j \notin \bigcup_{n \in f_{N_p}(X)} N_\Sigma(n) \cup \bigcup_{n \in f_{N_s}(X)} N_\Sigma(n) \tag{51}$$

based on (34) we can conclude that $T^\Sigma(X) + 1 = T(SS)$, which proves the theorem.

The direct consequence of Theorem 5.2 is (41). To illustrate the proof of the Theorem 5.2, we generate a sample made of five populations having maximal variance 1. We spread centroids of these populations so that minimal statistical distance between them, according to (47), is $\theta_n = 7$. The generated sample is 1000 observations in total.

```
R> sigma <- matrix(data=c(1,0,0,1),nrow=2,ncol=2)
R> mu <- list(P1=c(20,20), P2=c(27,20), P3=c(20,30), P4=c(5,20), P5=c(20,5))
R> si1 <- SigmaIndex(theta = 3.2, neighborhood = 6.4)
R> for(mu_n in names(mu)) addPopulation(si1,mu_n,mu[[mu_n]],sigma,200)
R> si2 <- SigmaIndex(theta = 3.2, neighborhood = 9.6)
R> for(mu_n in names(mu)) addPopulation(si2,mu_n,mu[[mu_n]],sigma,200)
R> si3 <- SigmaIndex(theta = 3.2, neighborhood = 12.9)
R> for(mu_n in names(mu)) addPopulation(si3,mu_n,mu[[mu_n]],sigma,200)
```

Figure 14 shows the computational cost reduction distribution for 3 distinct θ_n settings. We can see in Figure 14a that $\theta_n \leq 6.4$ does not produce any computational cost reduction, hence, using Σ-index does not bring benefits compared to the sequential scan approach. However, as we go over the threshold $\theta_n \geq 6.4$, the Σ-index DAG starts having the maximal depth $d(G_\Sigma) > 1$, which brings benefits and computational cost reduction, as seen in Figures 14b and 14c.



(a) $\theta_n = 6.4$ (b) $\theta_n = 9.6$ (c) $\theta_n = 12.9$

Figure 14: Computational cost reduction histograms for the illustration of the proof of Theorem 5.2.

6. Conclusion

Many of current statistical machine learning algorithms use statistical distance to determine the observation outcome. Although popular, using statistical distance for machine learning comes with performance issues, mainly due to the high number of computations needed to scan a big model space. Using an organizing index to relax this problem would be beneficial. As we demonstrated in the paper, the current organizing indices are not compatible with statistical models.

To solve this, we proposed a new organizing structure, a DAG named Σ-index that can be used to organize a statistical model based on the statistical relations between distributions comprising the model. In the paper we define the DAG structure and building rules. We also defined a set of algorithms that can be used for querying and maintenance of the Σ-index. In the experimental verification, the C++ implementation of the Σ-index was tested with synthetic and real-life testing samples. In most of the tests, using Σ-index reduced computational cost by 60-80%.

Finally, we analyzed all borderline cases to prove that using Σ-index has advantages over the sequential scan approach in most of the cases. Theoretical analysis shows that using Σ-index has no benefits only when organizing less than three distinct populations. For all other cases, we need to have a sufficient θ_n to achieve good computational cost reduction and retain the same precision as with the *sequential scan* approach. The chosen θ_n depends on local statistical model density. In this paper we used globally chosen θ_n, which could diminish results in more dense areas. Future research of the proposed Σ-index could include a dynamic way of choosing local threshold θ_n, which could equalize the effect of the connection theorem across the whole space and all statistical distributions.

The proposed Σ-index can be used in variety of applications and products. We demonstrated usage in clustering algorithms. However, such organizing structure could be used in databases

or in-memory data hypercubes, which would open a possibility to define an algebra and languages for searching and manipulating data in more statistical way.

References

Abdi H, Williams LJ (2010). "Principal component analysis." *Wiley interdisciplinary reviews: computational statistics*, **2**(4), 433–459.

Alpaydin E (2020). *Introduction to Machine Learning*. MIT press.

Axler S (2015). *Linear algebra done right*. Springer-Verlag.

Bayer R, McCreight E (1970). "Organization and Maintenance of Large Ordered Indices." In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, p. 107–141. Association for Computing Machinery, New York, NY, USA. ISBN 9781450379410. doi:10.1145/1734663.1734671.

Ciaccia P, Patella M, Zezula P (1997). "M-tree: An Efficient Access Method for Similarity Search in Metric Spaces." In *Proceedings of the 23rd VLDB conference, Athens, Greece*, pp. 426–435. Citeseer.

Dasgupta S (1999). "Learning mixtures of Gaussians." In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, pp. 634–644. IEEE.

Desgraupes B (2017). "Clustering indices." *University of Paris Ouest-Lab Modal'X*, **1**, 34.

Eddelbuettel D, Francois R (2011). "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software, Articles*, **40**(8), 1–18. ISSN 1548-7660. doi:10.18637/jss.v040.i08. URL https://www.jstatsoft.org/v040/i08.

Gama J (2010). *Knowledge discovery from data streams*. CRC Press.

Guttman A (1984). "R-trees: A dynamic index structure for spatial searching." In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pp. 47–57.

Hahsler M, Bolaños M, Forrest J (2017). "Introduction to stream: An Extensible Framework for Data Stream Clustering Research with R." *Journal of Statistical Software, Articles*, **76**(14), 1–50. ISSN 1548-7660. doi:10.18637/jss.v076.i14. URL https://www.jstatsoft.org/v076/i14.

Harris JM, Hirst JL, Mossinghoff MJ (2008). *Combinatorics and graph theory*, volume 2. Springer-Verlag.

Kleinberg JM (2000). "Navigation in a small world." *Nature*, **406**(6798), 845–845.

Krishnamoorthy A, Menon D (2013). "Matrix inversion using Cholesky decomposition." In *2013 signal processing: Algorithms, architectures, arrangements, and applications (SPA)*, pp. 70–72. IEEE.

Krleža D, Vrdoljak B, Brčić M (2020). "Statistical hierarchical clustering algorithm for outlier detection in evolving data streams." *Machine Learning*. doi:10.1007/s10994-020-05905-4.

Mahalanobis PC (1936). "On the generalized distance in statistics." National Institute of Science of India.

Malkov Y, Ponomarenko A, Logvinov A, Krylov V (2012). "Scalable distributed algorithm for approximate nearest neighbor search problem in high dimensional general metric spaces." In *International Conference on Similarity Search and Applications*, pp. 132–147. Springer-Verlag.

Rossi RJ (2018). *Mathematical statistics: an introduction to likelihood based inference*. John Wiley & Sons.

Sherman J, Morrison WJ (1950). "Adjustment of an inverse matrix corresponding to a change in one element of a given matrix." *The Annals of Mathematical Statistics*, **21**(1), 124–127.

Silva JA, Faria ER, Barros RC, Hruschka ER, de Carvalho AC, Gama J (2013). "Data stream clustering: A survey." *ACM Computing Surveys (CSUR)*, **46**(1), 13.

Woodbury MA (1950). "Inverting modified matrices." *Memorandum report*, **42**(106), 336.

Affiliation:

Dalibor Krleža
Department of Applied Computing
Faculty of Electrical Engineering and Computing
University of Zagreb
Unska 3
10000 Zagreb, Croatia
E-mail: dalibor.krleza@fer.hr

Anamari Nakić
Department of Applied Mathematics
Faculty of Electrical Engineering and Computing
University of Zagreb
Unska 3
10000 Zagreb, Croatia
E-mail: anamari.nakic@fer.hr

Boris Vrdoljak
Department of Applied Computing
Faculty of Electrical Engineering and Computing
University of Zagreb
Unska 3
10000 Zagreb, Croatia
E-mail: boris.vrdoljak@fer.hr