Fränti *J Big Data (2018) 5:13*
https://doi.org/10.1186/s40537-018-0122-y

Journal of Big Data

RESEARCH Open Access

Efficiency of random swap clustering

Pasi Fränti*

*Correspondence:
franti@cs.uef.fi
Machine Learning Group,
School of Computing,
University of Eastern Finland,
P.O.Box 111, 80101 Joensuu,
Finland

Abstract

Random swap algorithm aims at solving clustering by a sequence of prototype swaps, and by fine-tuning their exact location by k-means. This randomized search strategy is simple to implement and efficient. It reaches good quality clustering relatively fast, and if iterated longer, it finds the correct clustering with high probability. In this paper, we analyze the expected number of iterations needed to find the correct clustering. Using this result, we derive the expected time complexity of the random swap algorithm. The main results are that the expected time complexity has (1) linear dependency on the number of data vectors, (2) quadratic dependency on the number of clusters, and (3) inverse dependency on the size of neighborhood. Experiments also show that the algorithm is clearly more efficient than k-means and almost never get stuck in inferior local minimum.

Keywords: Clustering, Random swap, K-means, Local search, Efficiency

Introduction

The aim of clustering is to group a set of N data vectors $\{x_i\}$ in D-dimensional space into k clusters by optimize a given objective function f. Each cluster is represented by its *prototype*, which is usually the *centroid* of the cluster. *K-means* performs the clustering by minimizing the distances of the vectors to their cluster prototype. This objective function is called *sum-of-squared errors* (SSE), which corresponds to minimizing within-cluster variances. The output of clustering is the set of cluster labels $\{p_i\}$ and the set of prototypes $\{c_i\}$.

K-means was originally defined for numerical data only. Since then, it has also been applied to other types of data. The key is to define the distance or similarity between the data vectors, and to be able to define the prototype (center). It is not trivial how to do it, but if properly solved, then k-means can be applied. In case of categorical data, several alternatives were compared including *k-medoids*, *k-modes*, and *k-entropies* [1].

Quality of clustering depends on several factors. The first step is to choose the attributes and the objective function according to the data. They have the biggest influence on the clustering result, and their choice is the most important challenge for practitioners. The next step is to deal with *missing attributes* and *noisy data*. If the number of missing attributes is small, we can simply exclude these data vectors from the process. Otherwise some *data imputation* technique should be used to predict the missing values; for some alternatives see [2].

Springer Open

© The Author(s) 2018. This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

Noise and outliers can also bias the clustering especially with the SSE objective function. Detection of outliers is typically considered as a separate pre-processing step. Another approach is to perform the clustering first, and then label points that did not fit into any cluster as outliers. Outlier removal can also be integrated in the clustering directly by modifying the objective function [3]. After the pre-processing steps, the main challenge is to optimize the clustering so that the objective function would be minimized. In this paper, we focus on this problem.

We use the *centroid index* (CI) as our primary measure of success [4]. It counts how many real clusters are missing a prototype, and how many have too many prototypes. The CI-value is the higher of these two numbers. It is demonstrated in Fig. 1 where four real clusters are missing a prototype. This value provides a clear intuition about the result. Specifically, if CI = 0, the result is correct clustering. Sometimes we normalize CI by the number of clusters, and report the relative CI-value (CI/k). If the ground truth is not available, the result can be compared with the global minimum (if available), or with the best available solution used as gold standard.

K-means is very good in fine-tuning the cluster boundaries locally but it is unable to solve the cluster locations globally. In the solution in Fig. 2 (current solution), most prototypes are in correct locations, except at the top there are two prototypes but only one would be needed; one prototype is also missing in the middle. This solution has value CI = 1. K-means is not able to fix it as the two regions are spatially separated, and there is one stable cluster between the two problematic ones. Gradual changes cannot therefore happen by k-means iterations. This kind of problems is typical especially when the data contains well-separated clusters.

On the other hand, the correct locations of the prototypes can be solved by a sequence of *prototype swaps*, and leaving the fine-tuning of their exact location to k-means. In Fig. 2, only one swap is needed to fix the solution. An important observation is that it is not even necessary to swap one of the redundant prototypes but simply removing any prototype in their immediate neighborhood is enough since k-means can fine-tune

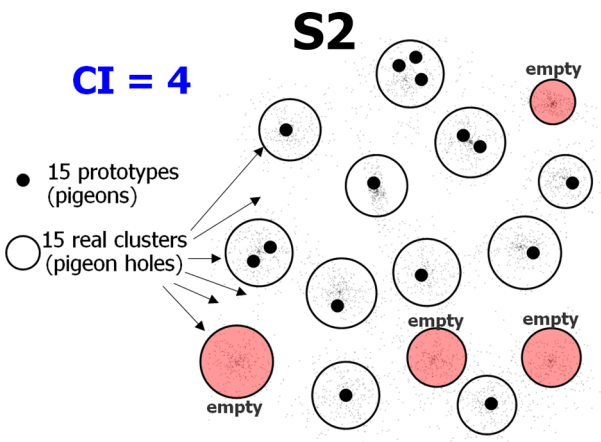

Fig. 1 Illustration of the clustering problem using a pigeon-hole principle with dataset S_2. The circles represent the location of the ground truth clusters (pigeon holes), and the black dots the prototypes. Centroid index is calculated as how many cluster-level errors (either empty slots or over-crowded slots) there are

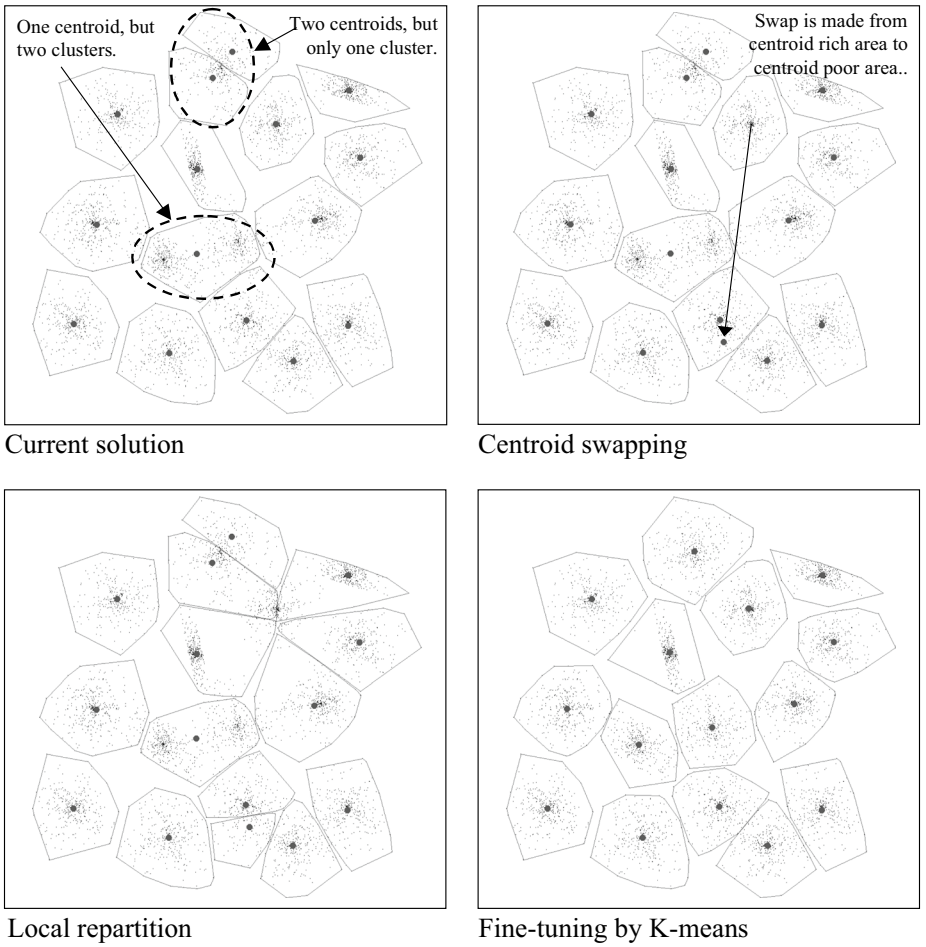

Fig. 2 Demonstration of the prototype swap for a sample initial solution for $N = 5000$ data vectors from $k = 15$ clusters (data set S_2)

their exact location locally. Also, the exact location where the prototype is relocated is not important, as long as it is in the immediate neighborhood where the prototype is needed.

Several *swap-based* clustering algorithms have been considered in literature. *Deterministic swap* selects the prototype to be swapped as the one that increases the objective function value f least [5–7], or by merging two existing clusters [8, 9] following the spirit of agglomerative clustering. The new location of the prototype can be chosen by considering all possible data vectors [7], splitting an existing cluster [7, 10], or by using some heuristic such as selecting the cluster with the largest variance [5]. The swap-based approach has also been used for solving *p-median problem* [11].

The main drawback of these methods is their computational complexity. Much simpler but effective approach is *random swap* strategy: select the prototype to be removed randomly and replace it to the location of a randomly selected data vector. This trial-and-error approach was first used in the *tabu search* algorithm presented in [12], and later simplified to a method called *randomized local search* (RLS) [13]. The main observation was that virtually the same clustering quality is reached independent of the initial

solution. The same conclusion was later confirmed in [14]. CLARANS is another variant of this technique using medoids instead of centroids [15].

The main reason why random swap approach is not so widely used is the lack of theoretical results of its properties. Our experiments, however, have shown that it works much better than k-means in practice, and in most cases, is also more efficient [16–20]. Its main limitation is that there is no clear rule how long the algorithm should be iterated and this parameter needs to be selected experimentally.

In this paper, we formulate the *random swap* (RS) as a probabilistic algorithm. We show that the expected time complexity to find the correct cluster allocation of the prototypes is polynomial. The processing time of the algorithm depends on how many iterations (trial swaps) are needed, and how much time each iteration takes. We will show that for a given probability of failure (q), the time complexity of the algorithm is upper bounded by a function that has linear O(N) dependency on the number of data vectors, quadratic O(k^2) dependency on the number of clusters, and inverse dependency on the size of neighborhood.

The main advantage of random swap clustering is that it is extremely simple to implement. If k-means can be implemented for the data, random swap is only a small extension. K-means consists of two steps (partition step and centroid step), and the random swap method has only one additional step: prototype swap. In most cases, this step is independent on the data and the objective function. It is also trivial to implement, which makes it highly useful for practical applications.

Besides the theoretical upper bound, we compare the efficiency experimentally against k-means, repeated k-means, k-means++, x-means, global k-means, agglomerative clustering and genetic algorithm. With our clustering benchmark data sets, we compare the results to the known ground truth and observe that random swap finds the correct cluster allocation every time. In case of image data, we use genetic algorithm (GA) as golden standard, as it is the most accurate algorithm known.

The rest of the paper is organized as follows. In "Random swap clustering" section, we recall k-means and random swap algorithms and analyze their time complexities. In "Number of iterations" section, we study the number of iterations and derive the expected time complexity of the algorithm. The optimality of the algorithm is also discussed. In "Neighborhood size" section, we define the concept of neighborhood. Experimental studies are given in "Experiments" section to demonstrate the results in practice. Conclusions are drawn in "Conclusions" section.

Random swap clustering

The clustering problem is defined as follows. Given a set of N data vectors x in D-dimensional space, partition the vectors into k clusters so that *sum of squared error* SSE (intra cluster variance) is minimized. For consistency to our previous works, we normalize the sum per vector and per dimension. In this way, the function represents how much error the clustering causes per single attribute (dimension). In image processing context, it is also natural to normalize the value per pixel (N vectors in image, D pixels in vector). It is calculated as follows:

$$\text{normalized MSE} = \frac{SSE}{N \cdot D} \text{where } SSE = \sum_{i=1}^{N} \left\| x_i - c_{p_i} \right\|^2 \quad (1)$$

K-means algorithm starts with a set of randomly selected vectors as the initial prototypes. It then improves this it initial solution iteratively by the following two steps. In the first step, optimal partition is solved in respect to the given set of prototypes by mapping each data vector to its nearest prototype:

$$p_i = \arg\min_{1 \le j \le k} \left\| x_i - c_j \right\|^2 \forall i \in [1, N] \tag{2}$$

In the second step, a new set of prototypes is calculated based on the new partition:

$$c_j = \sum_{p_i = j} x_i \Big/ \sum_{p_i = j} 1 \quad \forall j \in [1, k] \tag{3}$$

These steps are iteratively performed for a fixed number of iterations, or until convergence.

Random swap algorithm

Random swap removes one existing cluster and creates a new one to a different part of the data space. This is done by selecting a randomly chosen prototype C_s, and replacing it by a randomly selected data vector x_i:

$$c_s \leftarrow x_i \quad \Big| \quad s = \text{rand}(1, k), \quad i = \text{rand}(1, N) \tag{4}$$

This generates a global change in the clustering structure. An alternative implementation of the swap would be to create the new cluster by first choosing an existing cluster randomly, and then by selecting a random data vector within this cluster. We use the first approach for simplicity but the second approach is useful for the analysis of the swap.

After the swap, local repartition is performed to update the partition. This local repartition is not obligatory as the solution will anyway be tuned by k-means afterwards, but it merely speed-ups the process. First, vectors of the removed cluster are re-partitioned to their nearby clusters. This is done by comparing the distances to all other prototypes (including the new cluster) and selecting the nearest:

$$p_i \leftarrow \arg\min_{1 \le j \le k} \left\| x_i, c_j \right\|^2 \quad \forall \quad p_i = s \tag{5}$$

Second, the new cluster is created by attracting vectors from nearby clusters by calculating the distance of all vectors to the new prototype. If the distance is smaller than the distance to the prototype of the current cluster, the vector will join the new cluster:

$$p_i \leftarrow \arg\min_{j = s \vee j = p_i} \left\| x_i, c_j \right\|^2 \quad \forall \quad i \in [1, N] \tag{6}$$

The new solution is then modified by two iterations of k-means to adjust the partition borders locally. The overall process is a trial-and-error approach: a new solution is accepted only if it improves the objective function (1).

Pseudo code of the algorithm is sketched in Fig. 3. In brief, random swap is a simpler wrapper, in which any existing k-means library can be used. One can also leave out the local re-partition step as k-means will anyway fix the partition. The purpose

Random Swap(X) $\longrightarrow$ C, P

$C \leftarrow$ Select random representatives(X);
$P \leftarrow$ Optimal partition(X, C);
REPEAT T times
  (C^{new},j) $\leftarrow$ Random swap(X, C);
  $P^{new} \leftarrow$ Local repartition(X, C^{new}, P, j);
  C^{new}, $P^{new} \leftarrow$ *Kmeans*(X, C^{new}, P^{new});
  IF $f(C^{new}, P^{new}) < f(C, P)$ THEN
    (C, P) $\leftarrow$ C^{new}, P^{new};
RETURN (C, P);

Fig. 3 Pseudo code of the *Random Swap* (RS) clustering. Local repartition is optional and can be left out especially if k-means is iterated 2 or more times

Iteration	nMSE	Time	
0	5312689297	0.006186	
1	2687180682	0.017132	CI=2
2	2275082565	0.027188	
3	1704970391	0.037289	CI=1
9	1700185570	0.097908	
16	1328087775	0.161352	CI=0
28	1327946264	0.265631	
58	1327923352	0.516983	
121	1327910949	1.027286	

Fig. 4 Demonstration of the process for S_2

of the local re-partition is merely to speed-up the process as it basically implements a half iteration of k-means but faster. Otherwise, it has no significance in the algorithm. To sum up, the only additions random swap have to k-means are the swap and the comparison (IF–THEN). They are both trivial to implement. Therefore, if k-means can be applied to the data, so can random swap.

The process of the algorithm is demonstrated in Fig. 4 with $T = 5000$ trial swaps. Eight of the trial swaps improves the sum of squared error (SSE) and are thus accepted. Among the accepted swaps, three reduces the CI-value (iterations 1, 3 and 16). The rest of the accepted swaps provide minor improvement in SSE via local fine-tuning (iterations 2, 9, 28, 58, 121). After that, no further improvement is found. In this example, 16 iterations were needed to solve the correct clustering (CI = 0), and 121 iterations to complete the local fine-tuning. However, the algorithm does not know when to stop, and we therefore need to estimate how many iterations (trial swaps) should be applied.

Implementations are available in our web pages in C, Matlab, Java, Javascript, R and Python:

http://www.uef.fi/web/machine-learning/software
http://cs.uef.fi/sipu/animator/
http://cs.uef.fi/sipu/clusterator/
http://cs.uef.fi/pages/franti/cluster/

There are also video lecture (*youtube*), presentation material (*ppt*), flash animation (*animator*) and web page (*Clusterator*) where anyone can upload data in text format and obtain quick clustering result in just a 5 s, or alternatively, use longer 5 min option for higher quality. It is currently limited to numerical data only but we plan to extend it to other data types in future.

Time complexity of a swap

Time complexity of a single iteration depends on the implementation of the following steps:

1. Swap of the prototype.
2. Removal of the old cluster.
3. Creation of the new cluster.
4. Updating affected prototypes.
5. K-means iterations.

Step 1 consists of two random number generations and one copy operation, which take O(1) time. For simplicity, we assume here that the dimensionality is constant $d = O(1)$. In case of very high dimensions, the complexities should be multiplied by d due to the distance calculations.

In step 2, a new partition is found for every vector in the removed cluster. The time complexity depends on the size of the removed cluster. In total, there are N data vectors divided into k clusters. Since the cluster is selected randomly, its expected size is N/k. Processing of a vector requires k distance calculations and k comparisons. This multiplies to $2k \cdot N/k = 2N$. Note that the *expected* time complexity is independent on the size of the cluster.

In step 3, distance of every data vector to the newly created prototype is calculated. There are N vectors to be processed, each requiring 2 distance calculations. Thus, the time complexity of this step sums up to $2N$, which is also independent on the size of the cluster.

In step 4, new prototypes are generated by calculating cumulative sums of the vectors in each cluster. To simplify the implementation, the cumulative sums are calculated already during the steps 2 and 3. One addition and one subtraction are needed per each vector that changes its cluster. The sums of the affected clusters (the removed, the new and their neighbors) are then divided by the size of the cluster. There are N/k vectors both in the removed and in the new cluster, on average. Thus, the number of calculations sums up to $2N/k + 2N/k + 2\alpha = O(N/k)$ where α denotes to the neighborhood size (see "Neighborhood size" section).

The time complexity of the k-means iterations is less trivial to analyze but the rule of thumb is that only local changes appear due to the swap. In a straightforward implementation, $O(Nk)$ time would be required for every k-means iteration. However, we use the reduced search variant [21], where full search is needed only for the vectors in the affected clusters. For the rest of the vectors, it is enough to compute distances only to the prototypes of the affected clusters. We estimate that the number of the affected clusters equals to the size of neighborhood of the removed and

Table 1 Time complexity estimations of the steps of the random swap algorithm, and the observed numbers as the averages over the first 500 iterations for data set *Bridge* (N = 4096, k = 256, N/k = 16, α ≈ 8)

Step	Time complexity	Observed number of steps at iteration		
		50	100	500
Prototype swap	2	2	2	2
Cluster removal	$2N$	7526	8448	10,137
Cluster addition	$2N$	8192	8192	8192
Prototype update	$4N/k+2\alpha$	53	61	60
K-means iterations	$\leq 4\alpha N$	300,901	285,555	197,327
Total	$O(\alpha N)$	316,674	302,258	215,718

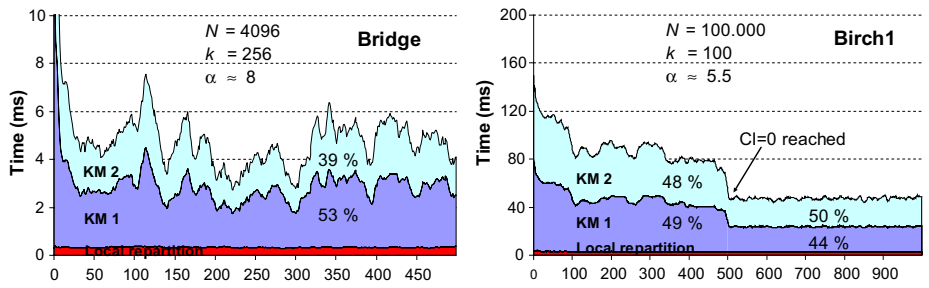

Fig. 5 Processing time profile with the random swap iterations. Time taken by the local repartition (steps 1–4) remains rather stable during the iterations whereas the time taken by k-means varies more

the added clusters, which is 2α. The expected number of vectors in those clusters is $2\alpha \cdot (N/k)$. The time complexity of one k-means iteration is therefore $2\alpha \cdot (N/k) \cdot k$ for the full searches, and $(N - (2\alpha \cdot (N/k))) \cdot 2\alpha \leq N \cdot 2\alpha$ for the rest. These sum up roughly to $4\alpha N = O(\alpha N)$ for two k-means iterations.

Table 1 summarizes the time complexity and shows also real observed numbers for the *Bridge* data set (see "Experiments" section). These numbers are reasonably close to the expected time complexities; only k-means iterations take twice more than what expected. The reason is that we apply only two iterations whereas the fast k-means variant in [21] does not become fully effective during the first two iterations. It is the more effective the less the prototypes are moving. This can be seen in Fig. 5; 1st iteration takes 53% share of the total processing time, but 2nd iteration only 39%. Nevertheless, the theoretical estimates are still well within the order of the magnitude bounds.

Figure 5 shows the distribution of the processing times between the steps 1–4 (swap + local repartition) and the step 5 (k-means). The number of processing time required by k-means is somewhat higher in the early iterations. The reason is the same as above: there are more prototypes moving in the early stage but the movements soon reduce to the expected level. The time complexity function predicts that k-means step would take $4\alpha N/(2N + 2N + 4\alpha N) = 89\%$ proportion of the total processing time with *Bridge*. The observed number of the steps gives 197,327/215,718 = 91% at the 500 iterations, and the actual measured processing times of k-means takes

$0.53 + 0.39 = 92\%$. For $BIRCH_1$, the time complexity predicts 85% proportion whereas the actual is 97% but it drops to 94% around 500 iterations after all the prototypes have found their correct location.

Further speed-up of k-means could be obtained by using the activity-based approach jointly with kd-tree [22], or by exploiting the activity information together with triangular inequality rule for eliminating candidates in the nearest neighbor search [23]. This kind of speed-up works well for data where the vectors are concentrated along the diagonal but generalizes poorly when the data is distributed uniformly [21, 24]. It is also possible to eliminate those prototypes from the full search whose activity is smaller than a given threshold and provide further speed-up at the cost of decreased quality [25, 26].

Number of iterations

We define three different types of swap:

- Trial.
- Accepted.
- Successful.

One *trial swap* is made in every iteration but only a swap that improves the objective function is called *accepted swap*. However, in the following analysis we are interested not all accepted swaps but only those that also reduces CI-value. In other words, swaps that correct one error in the global prototype allocation; minor fine-tunings do not count in this analysis. We therefore define a swap as *successful* if it reduces the CI-value.

In the example in Fig. 4, among the 5000 thousand trial swaps, 8 are accepted of which 3 we consider successful. Another example is shown in Fig. 6 where the swap in the middle improves objective function and is therefore accepted. However, it does not fix any problem in the prototype allocation, and is therefore not considered as a successful swap. In the rightmost case, the prototype is moved elsewhere and it fixes one more cluster location (CI changes from 2 to 1). This swap shows also significant reduction in the objective function (from 20.12 to 15.87). Larger scale examples are shown in Fig. 7; one with $CI = 4$ and another one with $CI = 9$.

Successful swaps

To achieve a successful swap, the algorithm must succeed in three independent tasks:

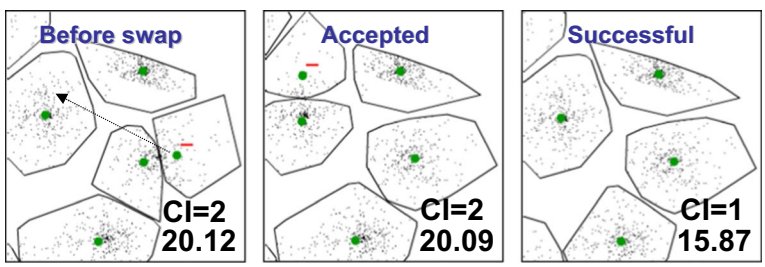

Fig. 6 Example of an accepted swap (middle) that improves SSE, and a successful swap (right) that also reduces the value of the centroid index (CI). Part of the dataset S_2 is shown

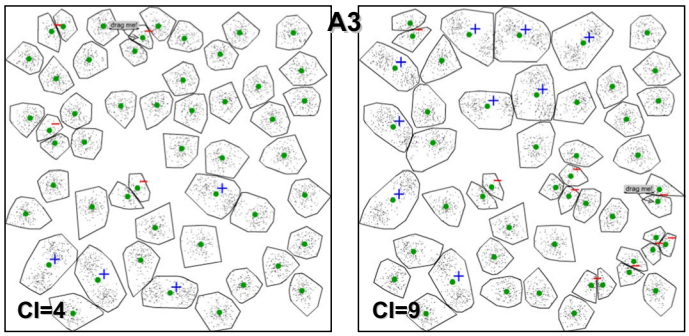

Fig. 7 Number of wrongly allocated prototypes in dataset A_3, as measured by centroid index. Prototypes mapped more than once are marked by '+', and unmapped prototypes by '−'

- Select a proper prototype to be removed.
- Select a proper location for the prototype.
- Perform local fine-tuning successfully.

The first two are more important, but we will show that the fine-tuning can also play a role in finding a successful swap. We analyze next the expected number of iterations to fix one prototype location, and then generalize the result to the case of multiple swaps.

To make successful swap to happen, we must remove one prototype from an over-partitioned region and relocate it to an under-partitioned region, and the fine-tuning must relocate the prototype so that it fills in one real cluster (*pigeon-hole*). All of this must happen during the same trial swap.

Assume that CI = 1. It means that there is one real cluster missing a prototype, and another cluster overcrowded by having two prototypes. We therefore have two favorable prototypes to be relocated. The probability for selecting one of these prototypes by a random choice is $2/k$ as there are k prototypes to choose from. To select the new location, we have N data vectors to choose from and the desired location is within the real cluster lacking prototype. Assume that all the clusters are of the same sizes, and that the mass of the desirable cluster is twice that of the others (it covers two real clusters). With this assumption, the probability that a randomly selected vector belongs the desired cluster is $2(N/k)/N = 2/k$. The exact choice within the cluster is not important because k-means will tune the prototype locations locally within the cluster.

At first sight, the probability for a successful swap appears to be $2/k \cdot 2/k = O(1/k^2)$. However, the fine-tuning capability of k-means is not limited within the cluster but it can also move prototypes across neighboring clusters. We define here that two clusters are *k-means neighbors* if k-means can move prototypes from a cluster to its neighbor. In order this to happen, the clusters must be both spatial neighbors and also in the vicinity of each other. This concept of neighborhood will be discussed more detailed in "Neighborhood size" section.

Note that it is also possible that the swap solves one allocation problem but creates another one elsewhere. However, this not considered a successful swap because it does not change the CI-value. It is even possible that CI-value occasionally increases even when objective function decreases but this is also not important for the analysis. In fact,

by accepting any swap that decreases the objective function, we guarantee that the algorithm will eventually converge; even if the algorithm itself does not know when it happens. We will study this more detailed in "Experiments" section.

Probability of successful swap

To keep the analysis simple, we introduce a data-dependent variable α to represent the size of the k-means neighborhood (including the vector itself). Although it is difficult to calculate exactly, it provides a useful abstraction that helps to analyze the behavior of the algorithm. Mainly α depends on the dimensionality (d) and structure of the data, but also on the size (N) and number of clusters (k). The worst case is when all clusters are isolated ($\alpha = 1$). An obvious upper limit is $\alpha \leq k$.

By following the intuition that any k-means neighbor of the desired cluster is good enough (both for the removal and for the addition) we estimate the probability of a successful swap as:

$$p = (a/k) \cdot (a/k) = \mathrm{O}(a/k)^2 \tag{7}$$

In total, there are $\mathrm{O}(\alpha)$ clusters to choose from, but both the removal and addition must be made within the neighborhood. This probability becomes lower when the number of clusters (k) increases, but higher when the dimensionality (d) increases. The exact dependency on dimensionality is not trivial to analyze. Results from literature imply that the number of spatial neighbors increases exponentially with the dimensionality: $\alpha = \mathrm{O}(2^d)$ [27]. However, the data is expected to be clustered and has some structure; it usually has lower intrinsic dimensionality than its actual dimensionality.

Analysis for the number of iterations

We study next the probability that the algorithm can fix one cluster in T iterations. We refer the probability of *success* as p, and the probability of *failure* as $q = 1 - p$. If only one swap is needed, the probability of failure equals to the probability of selecting T unsuccessful swaps in a row:

$$q = \left(1 - \frac{\alpha^2}{k^2}\right)^T$$

We can estimate the number of iterations (trial swaps) needed to find the successful swap with the probability of q as follows:

$$\log q = T \cdot \log\left(1 - \frac{\alpha^2}{k^2}\right) \Leftrightarrow T = \frac{\log(q)}{\log\left(1 - \frac{\alpha^2}{k^2}\right)} \tag{8}$$

For example, we have visually estimated that the size of neighborhood in Fig. 2 is 4, on average. This estimate leads to the probability of a favorable swap as $(\alpha/k)^2 = (4/15)^2 \approx 7\%$. The dependency of T on p and q is demonstrated in Fig. 8.

Bounds for the number of iterations

We derive tight bounds for the required number of iterations for (8) as follows.

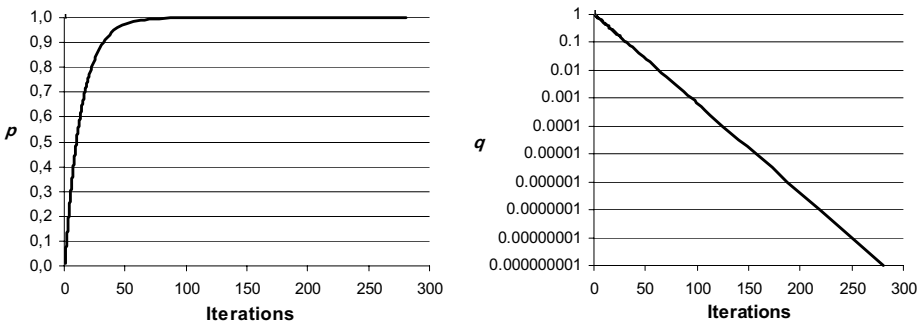

Fig. 8 Theoretical calculations for the probability of success (p) and failure (q) as a function of the iterations for S_2

Theorem

$$T = \Theta\left(-\ln q \cdot \frac{k^2}{\alpha^2}\right) \tag{9}$$

Proof for upper limit
According to [28, p. 54]:

$$\frac{x}{1+x} \leq \ln(1+x) \leq x, \quad \forall \quad x > -1 \tag{10}$$

This inequality is strict when $x=0$. From the second part of (10), we can derive inequality:

$$\begin{aligned} &\ln(1+x) \leq x \\ &\Rightarrow \ln(1-x) \leq -x \,\forall\, x < 1 \\ &\Rightarrow \frac{1}{\ln(1-x)} \geq -\frac{1}{x} \\ &\Rightarrow \frac{-1}{\ln(1-x)} \leq \frac{1}{x} \end{aligned}$$

Applying this result to (8) with $x=(\alpha/k)^2$, we derive an upper bound as:

$$T = \frac{\ln q}{\ln\left(1-\alpha^2/M^2\right)} \leq \frac{-\ln q}{\alpha^2/M^2} = -\ln(q) \cdot \frac{k^2}{\alpha^2} \tag{11}$$

Proof for lower limit
In a similar manner, we can derive another inequality from the first part of (10):

$$\begin{aligned} &\frac{x}{1+x} \leq \ln(1+x) \\ &\Rightarrow \frac{-1}{\ln(1-x)} \geq \frac{1-x}{x} \end{aligned}$$

Applying this to (8), we derive a lower bound as:

$$T = \frac{\ln q}{\ln\left(1-\alpha^2/k^2\right)} \geq -\ln(q)\frac{1-\alpha^2/k^2}{\alpha^2/k^2} = -\ln(q)\cdot\frac{k^2}{\alpha^2} \tag{12}$$

Since the same function is both the upper (11) and lower bound (12), the theorem is proven. □

Multiple swaps

The above analysis was made only if one prototype is incorrectly located. In case of the S_{1-4} datasets, 1 swap is needed in 60% cases of a random initialization, and 2 swaps in 38% cases. Only very rarely (< 2%) three or more swaps are required.

The result of "Analysis for the number of iterations" section can be generalized to multiple (w) swaps as follows. It is known that $T \gg w$ and $p \geq \alpha^2/k^2$ independent on the number of swaps. An upper bound for the probability for performing less than w successful swaps in T iterations can be calculated by the binomial probability:

$$q \leq \sum_{i=0}^{w-1} \binom{T}{i} \cdot \left(\frac{\alpha^2}{k^2}\right)^i \cdot \left(1-\frac{\alpha^2}{k^2}\right)^{T-i} \tag{13}$$

The idea is that there is a sequence of T swaps of which ($i < w$) are successful. The expected number of iterations for this would be:

$$\hat{T} = \left(\sum_{i=1}^{w}\frac{1}{i}\right)\cdot\frac{k^2}{\alpha^2} = \mathrm{O}\left(\log_2 w\cdot\frac{k^2}{\alpha^2}\right) \tag{14}$$

The only difference to the case of single swap is the logarithmic term ($\log_2 w$), which depends only on the number of swaps needed. Even this is a too pessimistic estimation since the probability of the first successful swap is up to w^2 times higher than that of the last swap. This is because there are potentially w times more choices for the successful removal and addition. Experimental observations show that 2.7 swaps are required with S_1–S_4, on average, and the number of iterations is multiplied roughly by a factor of 1.34, when compared to the case of a single swap. However, the main problem of using the Eq. (13) in practice is that the number of swaps (w) is unknown.

Overall time complexity

The total processing time is the number of iterations multiplied by the time required for a single iteration (αN). Based on the results in section combined with (11–13), it can be estimated as:

$$t(N,k) \leq -\ln q\cdot\log w\cdot\frac{k^2}{\alpha^2}\cdot\alpha N = \mathrm{O}\left(\frac{-\ln(q)\cdot\log w\cdot Nk^2}{\alpha}\right) \tag{15}$$

The *expected* processing time is derived accordingly multiplying (14) by αN:

$$\hat{t}(N,k) \leq \log\, w \cdot \frac{k^2}{\alpha^2} \cdot \alpha N = \mathrm{O}\left(\frac{-\log(w) \cdot Nk^2}{\alpha}\right) \tag{16}$$

From (16), we can make the following observations about the time complexity:

- Linear dependency on N.
- Quadratic dependency on k.
- Logarithmic dependency on w.
- Inverse dependency onα.

The main result is that the time complexity has only linear dependency on the size of data, but quadratic on the number of clusters (k). Although k is relatively small in typical clustering problems, the algorithm can become slow in case of large number of clusters.

The size of the neighborhood affects the algorithm in two ways. On one hand, it increases the time required by the k-means iterations. On the other hand, it also increases the probability for finding successful swaps, and in this way, it reduces the total number of iterations. Since the latter one dominates the time complexity, the final conclusion becomes somewhat surprising: the larger the neighborhood (α) the faster the algorithm.

Furthermore, since α increases with the dimensionality, the algorithm has inverse dependency on dimensionality. The higher the dimensionality implies the algorithm being faster. In this sense, data having low intrinsic dimensionality represent the worst case. For example, $BIRCH_2$ (see "Experiments" section) has one-dimensional structure ($D=1$) having small neighborhood size ($\alpha=3$).

Optimality of the random swap

So far we have assumed that the algorithm finds the correct cluster allocation every time (CI = 0). This can be argued by the *pigeonhole principle*: there are k pigeons and k pigeonholes. In a correct solution, exactly one pigeon (prototype) occupies one pigeon hole (cluster). The solution for this problem can be found by a sequence of swaps: by swapping pigeons from over-crowded holes to the empty ones. It is just a matter of how many trial swaps are needed to make it happen. We do not have formal proof that this would always happen but it is unlikely that the algorithm would get stuck in sub-optimal solution with wrong cluster allocation (CI > 0). With all our data having known ground truth and unique global minimum, random swap always finds it in our experiments.

A more relevant question is what happens with real world data that does not necessarily have clear clustering structure. Can the algorithm also find the global optimum minimizing SSE? According to our experiments, this is not the case. Higher dimensional image data can have—not exactly multiple optima—but multiple plateaus with virtually the same SSE-values. In such cases, random swap ends up to any of the alternative plateaus all having near-optimal SSE-values. Indirect evidence in [4] showed that two highly optimized solutions with the same cluster allocation (CI = 0) have also virtually the same objective function value (SSE = 0.1%) with significantly different allocations of the prototypes (5%).

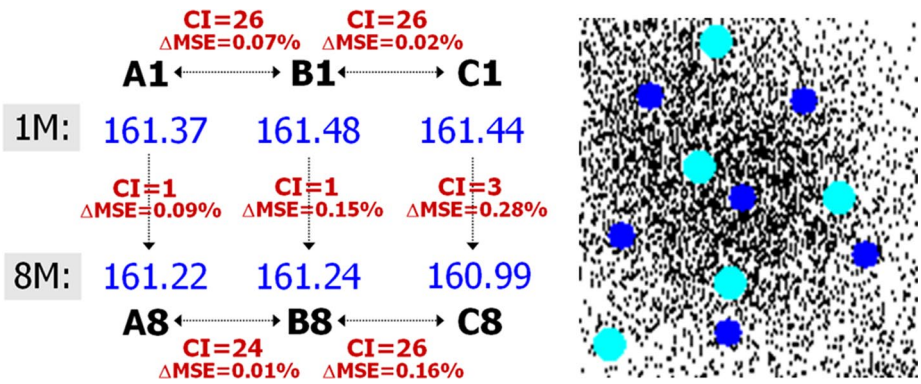

Fig. 9 With image dataset *Bridge* ($k=256$) random swap can find several different global allocations: CI = 26 (10%) but difference in SSE is negligible (< 0.2%). Visualization (right) shows the difference in practice

We constructed similar experience here using three different random initial solutions: A, B and C, see Fig. 9. We first performed one million trial swaps (A1, B1, C1) and then continued further to 8 million trial swaps, in total (A8, B8, C8). As expected, all solutions have very similar SSE-values (< 0.3% difference). Despite of this, their cluster level differences (CI-values) remain significantly high: A8 and B8 had 24/256 ≈ 9% differences. This indicates that the random swap algorithm will find one of the near-optimal solutions but not necessarily the one minimizing SSE.

To sum up: proof of optimality (or non-optimality) of the cluster level allocation (CI-value) remains an open problem. For minimizing SSE, the algorithm finds either the optimal solution (if unique) or one of the alternative near-optimal solutions all having virtually equal quality.

Neighborhood size

In the previous analysis, we introduced the concept of *k-means neighborhood*. The size of this neighborhood (including the vector itself) is denoted by α. It is defined as the average over the entire data set. In practice, it is not possible to calculate or even estimate α accurately without actually performing k-means. The value is bounded to the range $\alpha \in [1,k]$, and the worst case is $\alpha = 1$ when all clusters are isolated from each other. We next discuss how the size of this neighborhood could be analyzed in the context of multidimensional data.

Voronoi neighbors

One way to analyze whether clusters are neighbors is to calculate *Voronoi partition* of the vector space [29, 30] according to a given set of prototypes, and define two clusters as neighbors if they share a common *Voronoi facet*. An example of Voronoi partition is shown in Fig. 10 (left).

For 2-D data, an upper limit for the number of Voronoi surfaces has been shown to be $3k - 6$ [30]. Since every Voronoi surface separates two neighbor partitions, we can derive an upper limit for the average number of neighbors as $2 \cdot (3k - 6)/k = 6 - 12/k$, which approaches to 6 when k becomes large. In our 2-D data sets (S_1–S_4), there are 4 Voronoi neighbors, on average, varying from 1 to 10. For D-dimensional data, the

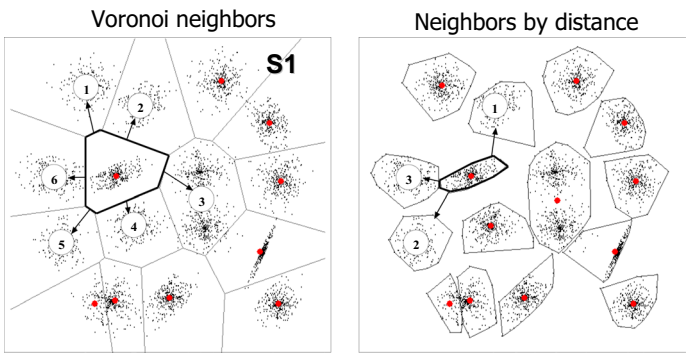

Fig. 10 Definition of neighborhood according to Voronoi partition (left), and according to their spatial connectivity and distance (right) for dataset S_1

number of Voronoi surfaces is upper bounded by $O(k^{\lceil D/2 \rceil})$ [30]. Respectively, an upper bound for the Voronoi neighbors is $O(2 \cdot k^{\lceil D/2 \rceil}/k) = O(2 \cdot k^{\lceil D/2 \rceil - 1})$.

However, such upper bounds estimates are far from reality. Firstly, there cannot be more neighbors than there are clusters ($\alpha \leq k$). Secondly, k-means cannot solve the local optimization if the distance between the clusters is greater than the longest distance within the clusters because no vector can then change partition by k-means iterations. So we have the following bounds derived both from theory and from data:

Theory for 2-dim: $\alpha \leq 6 - 12/k$
Theory for D-dim: $\alpha \leq O(2 \cdot k^{\lceil D/2 \rceil - 1}$
Data limit: $\alpha \leq k$

According to our experiments ("Estimating α and *T*" section), the theoretical bounds are reasonable for 2-*D* but not for higher dimensions. For example, our highest dimensional ($D=74$) dataset *KDD04-Bio* has only 33.3 neighborhood size whereas the theoretical upper bound is 10^{118}, and the data limit 2000. The *DIM-32* dataset is even more extreme; there are almost no neighbors because the clusters are well separated.

Data	Dim	*k*	Theory	Reality
S_2	2	15	5.2	4.5
Bridge	16	256	10^{17}	5.4
DIM-32	32	16	10^{19}	1.1
KDD04-Bio	74	2000	10^{118}	33.3

To sum up, the number of Voronoi neighbors is significantly higher than the size of the k-means neighborhood. A better estimator for the size of neighborhood is therefore needed.

Estimation algorithm

For the sake of analysis, we introduce an algorithm to estimate the average size of the k-means neighborhood. We use it merely to approximate the expected number of iterations (14) for a given data set in experiments in "Experiments" section. The idea is to first to find Voronoi neighbors and then analyze whether they are also k-means neighbors.

However, Voronoi can be calculated fast in O(k·log k) only in case of 2-dimensional data. In higher dimensions it takes O($k^{\lceil D/2 \rceil}$) time [31]. We therefore apply the following procedure.

First, we perform random swap clustering for a small number of iterations ($T=5$) to obtain an initial clustering. We then compare each pair of prototypes c_a and c_b to conclude whether the two clusters a and b are neighbors. We use the *XNN graph* introduced in [32]:

1. Calculate the half point of the prototypes: $hp \leftarrow (c_a + c_b)/2$.
2. Find the nearest prototype (nc) for hp.
3. If $nc = c_a$ or $nc = c_b$ then (a, b) are neighbors.

Every pair of prototypes that pass this test, are detected as spatial neighbors. We then calculate all vector distances across the two clusters. If any distance is smaller than the distance of the corresponding vector to its own cluster prototype, it is evidence that k-means has potential to operate between the clusters. Accordingly, we define the clusters as k-means neighbors. Although this does not give any guarantee, it is reasonable indicator for our purpose.

Experiments

We next test the theory and the assumptions using the data sets summarized in Table 2 and visualized in Fig. 11. The vectors in the first set (*Bridge*) are 4×4 non-overlapping blocks taken from a gray-scale image, and in the second set (*Miss America*) 4×4 difference blocks of two subsequent frames in video sequence. The third data set (*House*) consists of color values of the *RGB* image. *Europe* consists of differential coordinates from a large vector map. The number of clusters in these is fixed to $k=256$. We also use several generated data sets such as *BIRCH* [33], the high-dimensional data sets from [24], and the datasets S_1–S_4 with varying overlap of the clusters. *KDD04-Bio* is a large-scale high-dimensional data set [34].

Ground truth clustering results exist for all the generated data. For the rest, we iterate random swap algorithm for 1 million iterations, and use the final result as the reference

Table 2 Summary of the Data Sets

Data set	Ref.	Type of data	Vectors (N)	Clusters (k)	Vectors per cluster	Dimension (d)
Bridge	[35]	Gray-scale image	4096	256	16	16
House[a]	[35]	*RGB* image	34,112	256	133	3
Miss America	[35]	Residual vectors	6480	256	25	16
Europe		Diff. coordinates	169,673	256	663	2
$BIRCH_1$–$BIRCH_3$	[33]	Artificial	100,000	100	1000	2
S_1–S_4	[6]	Artificial	5000	15	333	2
Unbalance	[42]	Artificial	6500	8	821	2
Dim16–Dim1024	[24]	Artificial	1024	16	64	16–1024
KDD04-Bio	[34]	DNA sequences	145,751	2000	73	74

For archive of the data sets: http://cs.uef.fi/sipu/datasets/

[a] Duplicate data vectors are combined and their frequency information is stored instead

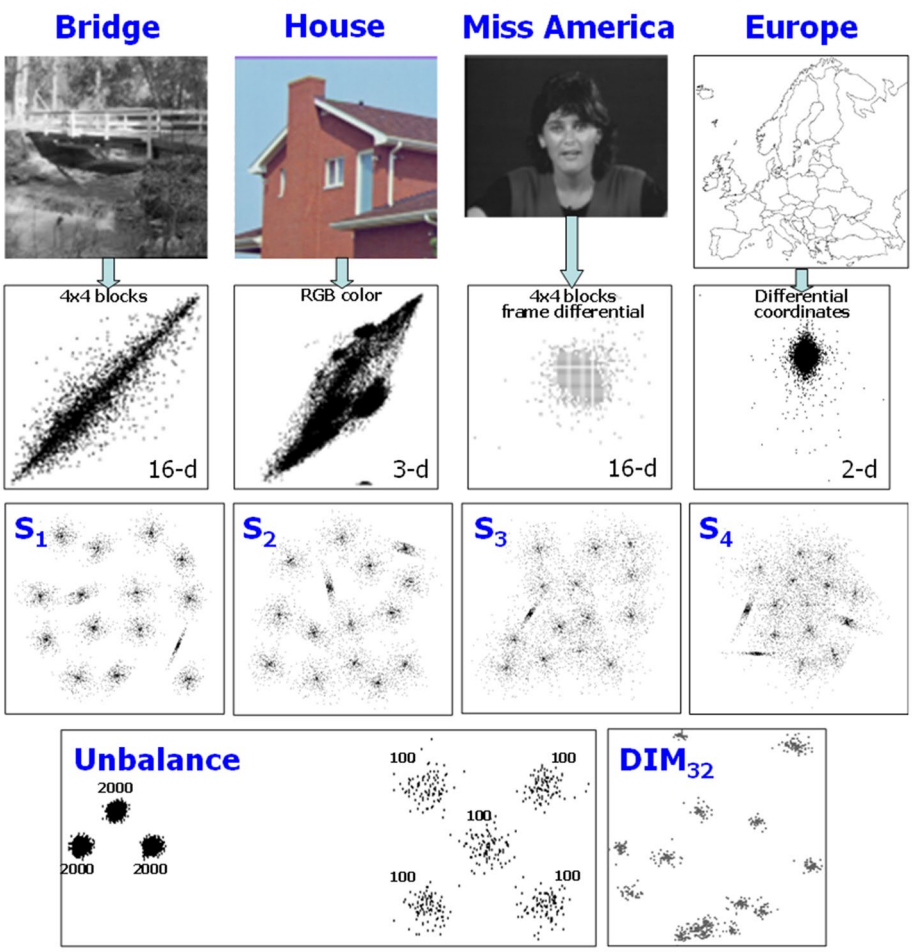

Fig. 11 Data sets (S_1–S_4), Unbalance and the first two dimensions of Dim_{32}

solution (golden standard) when applicable. To measure the goodness of the result, we calculate the centroid index (CI) against the ground truth (or reference) solution. Value CI = 0 indicates that the cluster level structure is correct.

All tests have been performed on a Dell PowerEdge R920 Server with four Xeon E7-4860 v2 processors having 1 TB memory and using RedHat Enterprise Linux 7 operating system.

Estimating α and *T*

Table 3 reports the estimated neighborhood sizes when calculated by the algorithm of "Estimation algorithm" section and that of the full data. For our analysis, it does not make a big difference at which stage ($T=0$, $T=5$, $T=5000$) the estimation is made. Even if the value was calculated from a random clustering ($T=0$) it still provides a useful estimate. In the following experiments, we use the values calculated at $T=5$.

We estimate the number of iterations (trial swaps) required to find the clustering for three given probabilities of failure (10, 1, 0.1%). The estimated number of iterations is small for data with few clusters (S_1–S_4, *Unbalance*). Even the highest confidence level ($q=0.1\%$) requires only 97–137 iterations for S_1–S_4, and 167 for *Unbalance*. The

Table 3 Neighborhood size (α) estimated from the data and from the clustering result after T Iterations

Dataset	Full data	From clustering			Estimated iterations (T)		
		Initial $T=0$	Early $T=5$	Final $T=5000$	$q=10\%$	$q=1\%$	$q=0.1\%$
Bridge	69.8	8.7	5.4	4.6	33,595	67,910	100,785
House	15.4	6.7	8.3	8.2	13,381	26,761	40,142
Miss America	346	34.2	17.1	11.9	3593	7078	10,617
Europe	(5.0)	4.8	6.3	6.3	26,699	53,398	80,098
$BIRCH_1$	5.0	4.5	5.8	5.6	2908	5815	8723
$BIRCH_2$	(4.7)	3.1	3.1	2.9	10,524	21,048	31,572
$BIRCH_3$	(4.9)	4.1	4.9	5.0	4508	9016	13,523
S_1	4.8	3.7	4.1	4.2	46	92	137
S_2	4.9	3.7	4.5	4.7	37	73	110
S_3	4.9	3.9	4.4	4.3	38	77	115
S_4	4.9	3.9	4.8	5.0	32	64	97
Unbalance	3.4	2.3	2.3	2.0	56	111	167
Dim-32	26.8	1.5	1.1	1.0	920	1839	2759
Dim-64	37.1	1.9	1.1	1.0	920	1839	2759
Dim-128	47.3	1.4	1.0	1.0	1135	2271	3406
KDD04-Bio	–	286.2	33.3	30.4	72,800	145,600	218,401

Estimated number of iterations (T) for selected values of q are calculated as $T = -\ln q \ln w (k/\alpha)^2$

clusters in the *Dim* dataset are isolated, which makes the size of the neighborhood very small (1.1). It therefore has larger estimates (about 1700). For the image datasets the estimates are significantly higher because of large number of clusters.

We next study how these predicted numbers compare to reality. For this, we run the algorithm for each dataset exactly the number of times estimated in Table 4. The runs are then repeated 10,000 times, and we record how many times the algorithm actually found the correct clustering (CI = 0). The results for Dim and S datasets in Fig. 12 show that the estimates are slightly over-optimistic for S_1–S_4. For example, for S_1 the predicted number of iterations are $T = (46, 92, 137)$ for the failure values $q = (10, 1, 0.1\%)$, whereas the algorithm actually failed 18, 4 and 0.6% times. For the Dim sets the results are much closer to the estimates.

The inaccuracies originate from two factors: the estimate of α is somewhat over-optimistic, and the fact that we apply only two iterations of k-means. To understand the significance of these we plot the distribution of the iterations (trial swaps) required in Fig. 13 as observed in the reality. In 90% of cases, the algorithm requires less than 70 (S_1) and 50 (S_4) iterations. The corresponding estimates ($q = 10\%$) are 46 (S_1) and 34 (S_4). Thus, the number of iterations needed is roughly 1.5 than what is estimated. This is well within the order of magnitude of the estimates of Eqs. (15) and (16).

The worst case behavior is when $\alpha = 1$, meaning that all clusters are isolated. If we left α out of the equations completely, this would lead to estimates of $T \approx 700$ for S_1–S_4 even with $q = 10\%$. With this number of iterations we always found the correct clustering and the maximum number of iterations ever observed was 322 (S_1) and 248 (S_4) at the end of the tail of the distribution. To sum up, the upper bounds hold very well in practice even if we ignored α.

Table 4 Expected number of iterations (Exp) for the last successful swap is estimated as $1/p = (k/\alpha)^2$, which is multiplied by log w to obtain the value for all swaps

Dataset	α	w	Last swap			All swaps		
			Exp	Real^2	Real^{10}	Exp	Real^2	Real^{10}
Bridge	5.4	90	2247	–	–	14,590	–	–
House	8.3	69	951	–	–	5811	–	–
Miss America	17.1	116	224	–	–	1537	–	–
Europe	6.3	130	1651	–	–	11,595	–	–
BIRCH$_1$	5.8	19	297	440	121	1263	637	197
BIRCH$_2$	3.1	21	1041	761	548	4571	1246	924
BIRCH$_3$	4.9	26	416	–	–	1958	–	–
S$_1$	4.1	2.8	13	26	18	20	33	23
S$_2$	4.5	2.7	11	19	9	16	25	12
S$_3$	4.4	2.7	12	17	7	17	22	10
S$_4$	4.8	2.7	10	19	9	14	25	11
Unbalance	2.3	4.0	12	58	54	24	122	110
Dim-32	1.1	3.7	212	52	52	399	73	76
Dim-64	1.1	3.7	212	59	64	399	83	88
Dim-128	1.0	3.8	256	56	65	493	91	98
KDD04-Bio	33.3	435	3607	–	–	31,617	–	–

The observed results are recorded using both two (Real^2) and ten (Real^{10}) k-means iterations. The number of swaps is calculated from the initial solution. Results are averages of 100 runs

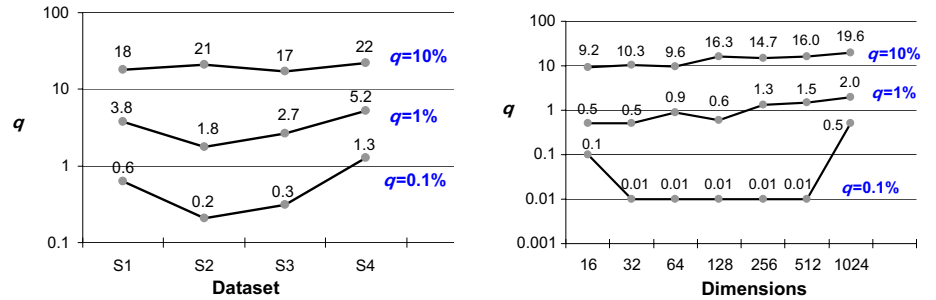

Fig. 12 Observed probability (%) of failure (q) for S_1–S_4 (left) and for the high dimensional data sets (right). The observed values are plotted, and the theoretical estimates are shown as blue

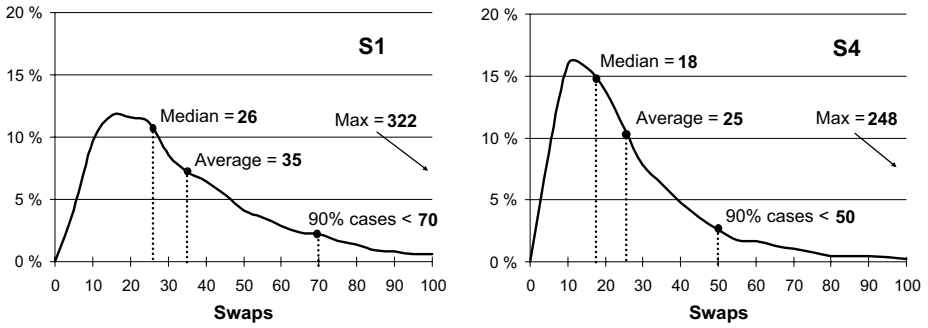

Fig. 13 Distribution of the number of trial swaps needed in reality for S_1 and S_4 (10,000 repeats)

Expected number of iterations

Next we study how well the Eq. (16) can predict the expected number of iterations. We use the same test setup and run the algorithm as long as needed to reach the correct result (CI = 0). Usually the last successful swap is the most time consuming, so we study separately how many iterations are needed from CI = 1 to 0 (*last swap*), and how many in total (*all swaps*). The estimated and the observed numbers are reported in Table 4. The number of successful swaps needed (w) is experimentally obtained from the data sets.

a. Estimated iterations

The expected results are again somewhat over-optimistic compared to the reality but still well within the order of magnitude of the time complexity results. For S sets, the algorithm is iterated about 50% longer [26, 29, 35] than estimated [18, 20, 21, 24]. For *BIRCH* data, the estimates for the *last* swap are slightly too pessimistic as it over-estimates the iterations by about 30%. The difference becomes bigger in case of all swaps. *Unbalance* has the biggest difference, almost 5 times, so let us focus on it a little bit more.

The creation of a new cluster assumes that all clusters are roughly of the same size. However, this assumption does not hold for the *Unbalance* data, which has three big clusters of size 2000 and five small ones of size 100. By random sampling, it is much more likely to allocate prototype in a bigger cluster. On average, random initialization allocates 7 prototypes within the three big clusters, and only one within the five small clusters. Starting from this initial solution, a successful swap must select a vector from any of the small clusters because the big clusters are too far and they are not k-means neighbors with the small ones. The probability for this is 500/6500 = 7.7% in the beginning (when $w = 4$ swaps needed), and 200/6500 = 3% for the last swap. The estimate is $1/k = 1/8 = 12.5\%$.

Despite this inaccuracy, the balance cluster assumption itself is usually fair because the time complexity result is still within the order of the magnitude. We could make even more relaxed assumption by considering the cluster sizes following arithmetic series cN, $2cN$,..., kcN, where c is a constant in range [0,1]. The analysis in [36] shows that the time complexity result holds both with the balance assumption and with the arithmetic case. The extreme case still exists though: cluster size distribution of (1, 1, 1,..., $N - k$) would lead to higher time complexity of $O(Nk^3)$ instead of $O(Nk^2)$. However, balance assumption is still fair because such tiny clusters are usually outliers.

b. K-means iterations

Another source of inaccuracy is that we apply only two k-means iterations. It is possible that k-means sometimes fails to make the necessary fine-tuning if the number of iterations is fixed too low. This can cause over-estimation of α. However, since the algorithm is iterated long, much more trial swaps can be tested within the same time. A few additional failures during the process is also compensated by the fact that k-means tuning also happens when a swap is accepted even if it is not considered as successful swap by the theory.

We tested the algorithm also using 10 k-means iterations to see whether it affects the estimates of T, see column Real[10] in Table 4. The estimates are closer to the reality with the S sets having cluster overlap. This allows k-means to operate better between the clusters. However, the estimates are not significantly more accurate, and especially with

datasets like *Dim* and *Unbalance* where clusters are well separated, the difference is negligible. The value of the k-means iterations is therefore not considered as important.

However, since k-means step is a bottleneck in terms of absolute running time, we studied its time-distortion efficiency with values 1, 2, 3, 4, 5, 10, 20 and 100. Earlier studies quite unanimously support that two iterations is the best choice [12, 13, 16], and our results here in Fig. 14 confirm this; two iterations is slightly better but the exact value is not critical. One k-means iteration is slightly inferior to two iterations. However, three or more iterations do not provide enough additional benefit to compensate the extra time spent. We observe the same trend with all data sets. Only exception is if we iterate the algorithm extremely long; several hours or even several days. Then applying 100 k-means iterations eventually provides the lower SSE-values with better time efficiency but this kind of result has no practical relevance.

c. Number of swaps

From Table 4, we can also observe that more time is spent for the last successful swap than to all previous ones together. It indicates that the logarithmic dependency on the number of swaps needed is too pessimistic estimation. We therefore study next how much more work is done by all the other swaps in addition to the last one. Results are summarized in Table 5 by measuring the factor between the number of iterations requires by the last swap relative to that of all swaps.

Overall trend is that the log w term is slightly too high estimate when compared to the reality. In case of S sets, the log w value indicates 1.5 total work, whereas the reality is between 1.27 and 1.34. For example, S_1 requires 33 swaps in total, of which 26 are spent for the last swap. In *BIRCH* datasets, the difference is much more visible. About 20 swaps are needed, which indicates extra work by a factor of about 4. In reality, only 50% more is required. The corresponding numbers for unbalance (2.0 vs. 2.09) and Dim sets (1.9 vs. 1.41–1.61) show also mild over-estimate.

Overall, our conclusion is that for datasets with clear clustering structure, the last swap is the bottleneck and the additional work for all the previous swaps at most doubles the total work load. We therefore conclude that knowing the exact number of swaps needed is not very important to have a good estimate for the required number of iterations.

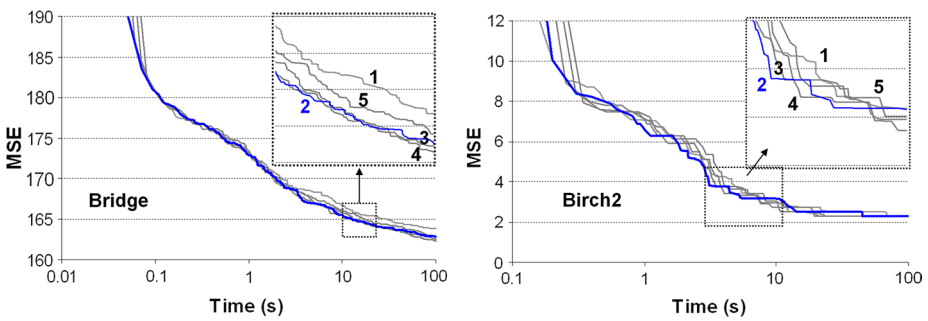

Fig. 14 Effect of the number of k-means iterations applied within RS. The value 2 is slightly better choice but the exact value is not critical. Only when iterated very long, high number of k-means iterations can become more beneficial with the image datasets. When running 45 min with Bridge, iterating k-means until convergence improved the SSE-value from 161.38 to 160.89 (improvement of 0.3%)

Table 5 Total number of iterations required to reach a solution with certain number of swaps remaining (0–4)

Dataset	Iterations to reach CI-value						Factor	log *w*
	Total	0	1	2	3	4		
$BIRCH_1$	637	440	78	44	24	14	1.45	4.2
$BIRCH_2$	1246	761	191	84	51	33	1.64	4.4
S_1	33	26	7	2	1	1	1.34	1.5
S_2	25	19	4	1	1	1	1.30	1.4
S_3	22	17	4	1	1	1	1.34	1.4
S_4	25	19	3	1	1	1	1.27	1.4
Unbalance	122	58	29	23	13	1	2.09	2.0
Dim-32	73	52	13	6	3	2	1.42	1.9
Dim-64	83	59	13	7	4	2	1.41	1.9
Dim-128	91	56	20	9	4	3	1.61	1.9

Results are averages of 100 runs. Most work is spent for finding the *last swap*. The factor indicates how many iterations are spent for all swaps (total) compared to the last swap (CI = 0). The estimated factor is given by the term log *w*

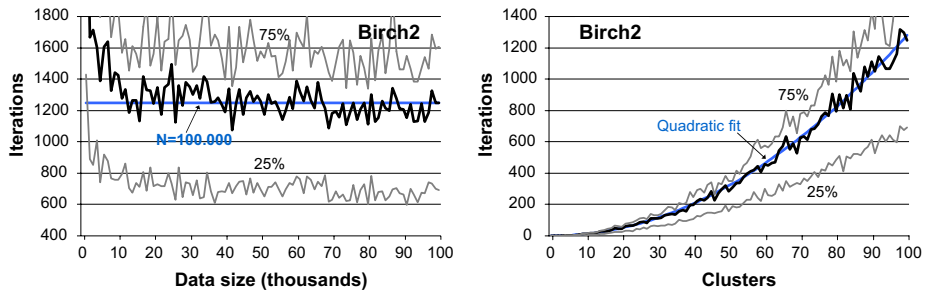

Fig. 15 Number of iterations needed to find the correct clustering for subsets of $BIRCH_2$ when varying the size of data (left), and number of clusters (right). The black line is the average of 100 runs, and the 25 and 75% quartiles show the level of variation between the individual runs. The blue lines are linear (left) and quadratic (right) fits to the data

d. Effect of N and k

We perform one more analysis by varying the size of the data (N) and the number of clusters (k). We reduce the size of $BIRCH_2$ in two alternative ways. First we generated random subsamples by eliminating 1000 random vectors at a time to create a series of subsets with $N = 1000–100{,}000$. Second we eliminated one cluster at a time to create a series of subsets with $k = 1–100$. For these subsets, we ran the algorithm until it found the correct clustering (CI = 0). Results are summarized in Fig. 15. We observe that the number of iterations needed has almost no dependency on the size of data but there it has strong quadratic correlation with k. These observations correspond directly to our theory in Eq. (14).

Time-distortion efficiency in practice

We next compare time-distortion performance of the *random swap* (RS) algorithm to *repeated k-means* (RKM) in practice. It is well known that k-means often gets stuck into an inferior local minimum, and because of this, most practitioners repeat the algorithm. The number of repeats is similar parameter as the number of iterations in RS. We

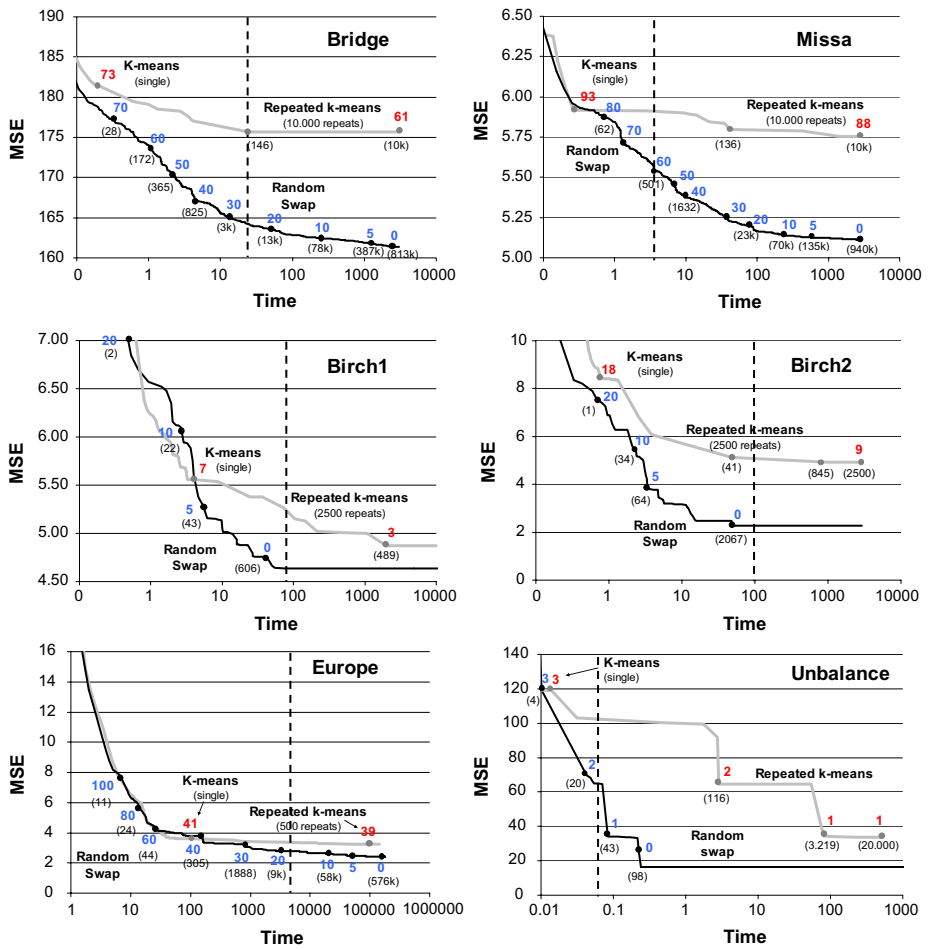

Fig. 16 Time-distortion comparison of random swap and repeated K-means. In parentheses there are number of trial swaps (random swap) or repeats (repeated K-means) applied. The blue and red values show the number of remaining swaps to reach the final solution. Time unit is seconds. The dashed vertical line shows the expected number of iterations

consider 10–100 repeats but it can be easily extended to much higher. Open questions are how many repeats should one use, and whether RKM will also find the correct global allocation.

We first let both RS and RKM run extremely long: RS 1 million iterations, and repeat RKM 500–25,000 times depending on the data. Time-distortion results are summarized in Fig. 16 showing also the number of trial swaps (RS) and repeats (RKM). The number of remaining successful swaps needed is shown by red and blue colors. In case of real data, we use the result of RS after 1 million iterations as reference. This is not necessarily the global optimum but merely as a point of comparison. The dashed line shows the point when the expected number of iterations is reached.

The results show that RS is significantly more efficient than RKM. It usually achieves the same quality equally fast as k-means, and outperforms it when kept iterating longer. In case of artificial data with clear clusters ($Birch_1$, $Birch_2$, *Unbalanced*), RS finds the correct clustering (CI = 0) in all cases. For the same data, k-means has

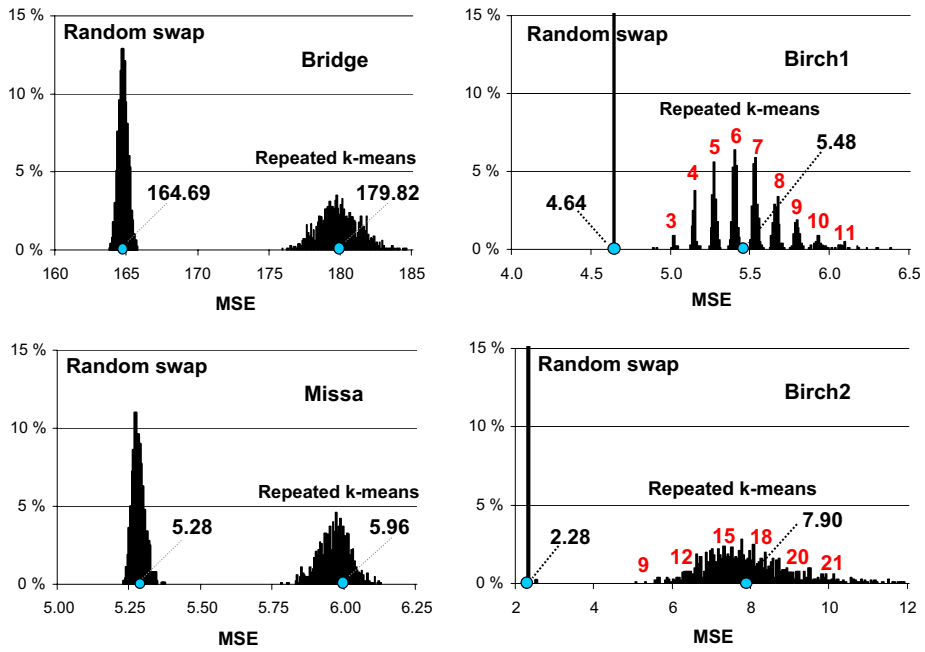

Fig. 17 Histograms of SSE-values for RS and RKM. CI-values are plotted by red font (when applicable) to approximate the number of incorrectly allocated prototypes (missing swaps)

(CI = 7, CI = 18, CI = 3) and repeated k-means (CI = 3, CI = 9, CI = 1). RS achieves the correct clustering in less than 1 min using (606, 2067, 98) trial swaps. Repeated k-means is successful only with the *Unbalance* dataset. It finds the correct result 60% of the trials if repeated 20,000 times. On average, it took 17,538 repeats to reach CI = 0 by RKM.

Multi-dimensional image data sets do not have clear clustering structure. The clustering can therefore end-up with significantly different cluster allocation (10% different clusters) despite having virtually the same SSE-value (see "Optimality of the random swap" section). The number of missing swaps is therefore less intuitive what it means in practice, but the superiority of RS in comparison to the repeated K-means is still clear. The toughest data set is *Europe*, for which RS outperforms RKM only after running several minutes (after about 2000 swaps).

The expected number of iterations (dashed vertical line) gives a reasonable estimate when the algorithm has succeeded in case of *S* and *BIRCH* datasets. For the image datasets, it is just like a line drawn into water without any specific meaning. Since they lack clear clustering structure, the algorithm keeps on searching—and also keeps on finding—better allocations of the prototypes. It seems not to reach any local or global minimum. Additional tests revealed that it seems to stabilize somewhere before 10 M iterations, which corresponds roughly about 3 weeks of processing—way beyond any practical significance.

As a second test, we run RS several times with T = 5000 iterations. Histograms of the resulting values are shown in Fig. 17. In case of *S* and *BIRCH* datasets, RS always found the correct clustering. In case of image data sets, there are some variations but even the worst run of RS is always better than the best run of k-means. Based on the

Table 6 Summary of the processing times and clustering quality

Dataset	KM	RKM	KM++	XM	AC	RS_{5000}	RS_x	GKM	GA
Processing time (s)									
$BIRCH_1$	3.7	374	3.2	107	141	276	420	–	297
$BIRCH_2$	1.1	114	1.0	12	144	92	537	–	256
S_1	< 1	1.1	< 1	< 1	< 1	5	< 1	74	3
S_2	< 1	1.4	< 1	< 1	< 1	6	< 1	95	2
S_3	< 1	1.8	< 1	< 1	< 1	6	< 1	109	3
S_4	< 1	2.5	< 1	< 1	< 1	7	< 1	117	3
Unbalance	< 1	2.6	< 1	< 1	< 1	12	< 1	152	2
Dim-32	< 1	< 1	< 1	< 1	< 1	3	1.5	6	1
Dim-64	< 1	< 1	< 1	< 1	< 1	5	3	11	2
Dim-128	< 1	< 1	< 1	< 1	< 1	8	5	19	3
Centroid index (CI)									
$BIRCH_1$	*6.6*	*2.9*	*4.0*	*1.6*	0	0	0	–	0
$BIRCH_2$	*16.9*	*10.5*	*7.6*	*1.7*	0	0	0	–	0
S_1	*1.8*	0.0	*1.1*	*0.3*	0	0	0	0	0
S_2	*1.5*	0.0	*1.0*	*0.2*	0	0	0	0	0
S_3	*1.1*	0.0	*0.9*	*0.3*	0	0	0	0	0
S_4	*0.8*	0.0	*0.9*	*0.4*	*1*	0	0	0	0
Unbalance	*3.9*	*2.0*	*0.5*	*1.7*	0	0	0	0	0
Dim-32	*3.8*	*1.1*	*0.5*	*2.7*	0	0	0	0	0
Dim-64	*3.7*	*1.1*	0.0	*4.0*	0	0	0	0	0
Dim-128	*4.0*	*1.4*	0.0	*4.2*	0	0	0	0	0

The results of KM, RKM, KM++ and XM are averages over 10 runs. When correct result (CI = 0) is not always found the result is italics

SSE-values, it is difficult to conclude how significant the difference would be for practical application. However, the CI-values with *BIRCH* demonstrate it better. Among the k = 100 clusters, k-means fails in case of 7% with $BIRCH_1$, and 18% with $BIRCH_2$. Repeated k-means reaches levels 3 and 9% but it requires 1000 repeats, which takes several minutes. Reaching CI = 0 by RKM seems unrealistic, indicating that it is not capable to solve the global cluster allocation in general.

Finally, we tested the optimality of the random swap using the datasets with known ground truth (*S*, *Birch*, *Dim*, *Unbalance*). We let the algorithm run until it reached the correct clustering (CI = 0), and then restarted it from scratch. We let it run 10,000 times for *Birch* sets and 1,000,000 times for *S*, *Dim* and *Unbalance* sets. The algorithm found CI = 0 result every time and never got stuck into a sub-optimal solution even once.

Comparison to other algorithms

Since random swap is not the only good clustering algorithm around, we next put the results in wider perspective. Table 6 summarizes the quality and the processing times of a few selected algorithms including the best known and some popular ones: *k-means* (KM), *repeated k-means* (RKM), *k-means++* [37], *x-means* [38], *agglomerative clustering* (AC) [35], *random swap* (RS) [13], *global k-means* [7], and *genetic algorithm* [39]. For all k-means variants, we use the fast variant presented in [21]. For detailed description

and more comprehensive quality comparison, we refer to [6]. For RS we use both 5000 iterations (RS_{5000}) and the number of iterations estimated for failure probability $q=0.1\%$ (RS_x) from Table 3. RKM is repeated 100 times. KM, KM++ and XM are results of individual runs. The clustering quality is measured by the Centroid Index (CI).

The results show that all k-means variants (KM, RKM, KM++, XM) fail to find the correct result in more than 50% of the cases. K-means++ and x-means work better than k-means but not significantly better than RKM. Better algorithms (AC, RS, GKM, GA) are successful in all cases with the exception of AC, which makes one error with S_4. RS is simplest to implement. AC is also relatively simple but requires the fast variant from [35] since a straightforward school book or Matlab implementations would be an order of magnitude slower. GA [39] is composed of the same AC and k-means components.

The down side of using the better algorithms is their slower running time. Three algorithms (AC, RS, GA) work in reasonable time for data sets of size $N=5000$ but require already several minutes for the *Birch* sets of size $N=100{,}000$. GKM is slow in all cases.

Conclusions

Random swap is an efficient algorithm for solving clustering, and unlike k-means, it does not converge to an inferior local minimum [40, 41]. In this paper, we have analyzed the number of iterations (trial swaps) needed to solve the global allocation of the clusters. Our main results are that the expected processing time, $O(Nk^2 \log w/\alpha)$, depends on the following factors:

- Linear $O(N)$ dependency on the size of data.
- Quadratic $O(k^2)$ dependency on the number of clusters.
- Inverse $O(1/\alpha)$ dependency on the neighborhood size.
- Logarithmic $O(\log w)$ dependency on the number successful swaps needed.

The main limitation is that no practical stopping criterion can be reliably derived from the theory. Previously a fixed number of iterations has been used, such as $T=5000$, or another rule of thumb such as $T=N$. Here we used $w=1$ and estimated α using the preliminary clustering after $T=5$ iterations. This works ok in practice, but a better estimation would still be desired. Nevertheless, if the quality of the clustering is the main concern, one can simply iterate the algorithm as long as there is time.

The worst case of the algorithm is when the clusters are isolated ($\alpha=1$), which leads to $O(Nk^2)$ time for one swap. The number of successful swaps needed (w) is unknown but it has at most logarithmic additional cost. Empirical results indicate that the last swap is the most time-consuming. Theoretical upper limit would be $O(Nk^2 \log k)$ assuming that $w=k$ swaps were needed. Due to the quadratic dependency on k, a faster algorithm such as deterministic swap [20] or divisive clustering [10] might be more useful if the number of clusters is very high.

Implementation of random swap is publicly available in C, Matlab, Java, Javascript, R and Python. We also have web page (http://cs.uef.fi/sipu/clusterator/) where anyone can upload data in text format and obtain quick clustering result in a 5 s, or alternatively use longer 5 min option. As future work, we plan to extend the *Clusterator* to solve the number of clusters, and also to support non-numerical data. Importing random swap to

other machine learning platforms like Spark MLlib will be considered, including a parallel variant to provide better support for big data.

Authors' contributions
The author is solely responsible of the entire work. The author read and approved the final manuscript.

Authors' information
Pasi Fränti received the M.Sc. and Ph.D. degrees from the University of Turku, 1991 and 1994 in Computer Science. Since 2000, he has been a professor of Computer Science at the University of Eastern Finland. He has published 72 journal and 159 conference papers, including 14 IEEE transaction papers. His current research interests are in clustering and location-aware applications.

Acknowledgements
The author thanks Dr. Olli Virmajoki for his efforts on the earlier drafts: let's shed more sweat and tears in future!

Competing interests
The author declares there is no competing interests.

Data
All data is publicly available here: http://cs.uef.fi/sipu/datasets/.

Duplication
The content of the manuscript has not been published, or submitted for publication elsewhere.

Ethics approval and consent to participate
Not required.

Funding
No funding to report.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 11 January 2018 Accepted: 13 March 2018
Published online: 21 March 2018

References

1. Hautamäki V, Pöllänen A, Kinnunen T, Lee KA, Li H, Fränti P. A comparison of categorical attribute data clustering. In: Joint international workshop on structural, syntactic, and statistical pattern recognition (S+SSPR 2014), LNCS 8621, Joensuu, Finland; 2014. p 55–64.
2. Tuikkala J, Elo LL, Nevalainen OS, Aittokallio T. Missing value imputation improves clustering and interpretation of gene expression microarray data. BMC Inf. 2008;9(1):202.
3. Ott L, Pang L, Ramos F, Chawla S. On integrated clustering and outlier detection. In: Advances in neural information processing systems (NIPS). 2014. p 1359–67.
4. Fränti P, Rezaei M, Zhao Q. Centroid index: cluster level similarity measure. Pattern Recognit. 2014;47(9):3034–45.
5. Fritzke B. The LBG-U method for vector quantization—an improvement over LBG inspired from neural networks. Neural Process Lett. 1997;5(1):35–45.
6. Fränti P, Virmajoki O. Iterative shrinking method for clustering problems. Pattern Recognit. 2006;39(5):761–5.
7. Likas A, Vlassis N, Verbeek JJ. The global k-means clustering algorithm. Pattern Recognit. 2003;36:451–61.
8. Kaukoranta T, Fränti P, Nevalainen O. Iterative split-and-merge algorithm for VQ codebook generation. Opt Eng. 1998;37(10):2726–32.
9. Frigui H, Krishnapuram R. Clustering by competitive agglomeration. Pattern Recognit. 1997;30(7):1109–19.
10. Fränti P, Kaukoranta T, Nevalainen O. On the splitting method for vector quantization codebook generation. Opt Eng. 1997;36(11):3043–51.
11. Resende MGC, Werneck RF. A fast swap-based local search procedure for location problems. Ann Oper Res. 2007;150(1):205–30.
12. Fränti P, Kivijärvi J, Nevalainen O. Tabu search algorithm for codebook generation in VQ. Pattern Recognit. 1998;31(8):1139–48.
13. Fränti P, Kivijärvi J. Randomised local search algorithm for the clustering problem. Pattern Anal Appl. 2000;3(4):358–69.
14. Kanungo T, Mount DM, Netanyahu N, Piatko C, Silverman R, Wu AY. A local search approximation algorithm for k-means clustering. Comput Geometry. 2004;28(1):89–112.
15. Ng RT, Han J. CLARANS: a method for clustering objects for spatial data mining. IEEE Trans Knowl Data Eng. 2002;14(5):1003–16.
16. Fränti P, Gyllenberg HH, Gyllenberg M, Kivijärvi J, Koski T, Lund T, Nevalainen O. Minimizing stochastic complexity using GLA and local search with applications to classification of bacteria. Biosystems. 2000;57(1):37–48.
17. Rus C, Astola J. Legend based elevation layers extraction from a color-coded relief scanned map. In: IEEE international conference image processing (ICIP), vol. 2. Genova, Italy; 2005. p 237–40.

18. Güngörand Z, Ünle A. *K*-harmonic means data clustering with simulated annealing heuristic. Appl Math Comput. 2007;184(2):199–209.
19. Nosovskiya GV, Liub D, Sourina O. Automatic clustering and boundary detection algorithm based on adaptive influence function. Pattern Recognit. 2008;41(9):2757–76.
20. Fränti P, Tuononen M, Virmajoki O. Deterministic and randomized local search algorithms for clustering. In: IEEE international conference on multimedia and expo, (ICME'08). Hannover, Germany; 2008. p 837–40.
21. Kaukoranta T, Fränti P, Nevalainen O. A fast exact GLA based on code vector activity detection. IEEE Trans Image Process. 2000;9(8):1337–42.
22. Lai JZC, Liaw Y-C. Improvement of the k-means clustering filtering algorithm. Pat. Rec. 2008;41(12):3677–81.
23. Elkan C. Using the triangle inequality to accelerate k-means. In: International conference on machine learning (ICML'03), Washington, DC, USA; 2003. p 147–53.
24. Fränti P, Virmajoki O, Hautamäki V. Fast agglomerative clustering using a k-nearest neighbor graph. IEEE TransPattern Anal Mach Intell. 2006;28(11):1875–81.
25. Jin X, Kim S, Han J, Cao L, Yin Z. A general framework for efficient clustering of large datasets based on activity detection. Statist Anal Data Mining. 2011;4:11–29.
26. Lai JZC, Liaw Y-C, Liu J. A fast VQ codebook generation algorithm using codeword displacement. Pattern Recognit. 2008;41:315–9.
27. Pestov V. On the geometry of similarity search: dimensionality curse and concentration of measure. Inf Process Lett. 2000;73:4751.
28. Cormen T, Leiserson C, Rivest R. Introduction to algorithms. 2nd ed. New York: MIT Press; 1990. p. 54.
29. Aurenhammer F. Voronoi diagrams-a survey of a fundamental geometric data structure. ACM Comput Surv. 1991;23(3):345–405.
30. Preparata F, Shamos M. Computational geometry. New York: Springer; 1985.
31. Barber CB, Dobkin DP, Huhdanpää HT. The quickhull algorithm for convex hulls. ACM Trans Math Softw. 1996;22(4):469–83.
32. Fränti P, Mariescu-Istodor R, Zhong C. XNN graph. In: Joint international workshop on structural, syntactic, and statistical pattern recognition (S+SSPR 2016), Merida, Mexico, LNCS 10029; 2016. p 207–17.
33. Zhang T, Ramakrishnan R, Livny M. BIRCH: a new data clustering algorithm and its applications. Data Min Knowl Disc. 1997;1(2):141–82.
34. KDD Cup 2004. bio_train.dat, training data for the protein homology task. 2004. http://osmot.cs.cornell.edu/kddcup.
35. Fränti P, Kaukoranta T, Shen D-F, Chang K-S. Fast and memory efficient implementation of the exact PNN. IEEE Trans Image Process. 2000;9(5):773–7.
36. Zhong C, Malinen MI, Miao D, Fränti P. A fast minimum spanning tree algorithm based on K-means. Inf Sci. 2015;295:1–17.
37. Arthur D, Vassilvitskii S. K-means++: the advantages of careful seeding. In: ACM-SIAM symposium on discrete algorithms (SODA'07), New Orleans, LA. 2007. p 1027–35.
38. Pelleg D, Moore A. X-means: extending k-means with efficient estimation of the number of clusters. In: International conference on machine learning, (ICML'00), Stanford, CA, USA. 2000.
39. Fränti P. Genetic algorithm with deterministic crossover for vector quantization. Pattern Recognit Lett. 2000;21(1):61–8.
40. Wu X. On convergence of Lloyd's method I. IEEE Trans Inf Theory. 1992;38(1):171–4.
41. Selim SZ, Ismail MA. K-means-type algorithms: a generalized convergence theorem and characterization of local optimality. IEEE Trans Pattern Anal Mach Intell. 1984;6(1):81–7.
42. Rezaei M, Fränti P. Set matching measures for external cluster validity. IEEE Trans Knowl Data Eng. 2016;28(8):2173–86.

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com