Faster k-Medoids Clustering: Improving the PAM, CLARA, and CLARANS Algorithms

Erich Schubert[1](✉) and Peter J. Rousseeuw[2]

[1] Technische Universität Dortmund, Dortmund, Germany
erich.schubert@tu-dortmund.de

[2] Department of Mathematics, KU Leuven, Leuven, Belgium
peter@rousseeuw.net

Abstract. Clustering non-Euclidean data is difficult, and one of the most used algorithms besides hierarchical clustering is the popular algorithm Partitioning Around Medoids (PAM), also simply referred to as k-medoids.

In Euclidean geometry the mean—as used in k-means—is a good estimator for the cluster center, but this does not exist for arbitrary dissimilarities. PAM uses the medoid instead, the object with the smallest dissimilarity to all others in the cluster. This notion of centrality can be used with any (dis-)similarity, and thus is of high relevance to many domains and applications.

A key issue with PAM is its high run time cost. We propose modifications to the PAM algorithm that achieve an $O(k)$-fold speedup in the second ("SWAP") phase of the algorithm, but will still find the same results as the original PAM algorithm. If we slightly relax the choice of swaps performed (while retaining comparable quality), we can further accelerate the algorithm by performing up to k swaps in each iteration. With the substantially faster SWAP, we can now explore faster intialization strategies. We also show how the CLARA and CLARANS algorithms benefit from the proposed modifications.

Keywords: Cluster analysis · k-Medoids · PAM · CLARA · CLARANS

1 Introduction

Clustering is a common unsupervised machine learning task, in which the data set has to be automatically partitioned into "clusters", such that objects within the same cluster are more similar, while objects in different clusters are more different. There is not (and likely never will be) a generally accepted definition of a cluster, because "clusters are, in large part, in the eye of the beholder" [7], meaning that every user may have different enough needs and intentions to want a different algorithm and notion of cluster. And therefore, over many years of research, hundreds of clustering algorithms and evaluation measures have been proposed, each with their merits and drawbacks. Nevertheless, a few seminal methods such as hierarchical clustering, k-means, PAM [9], and DBSCAN [6] have received repeated and widespread use. One may be tempted to think that these *classic* methods have all been well researched and understood, but there are still many scientific publications trying to explain these algorithms better (e.g., [19]), trying to parallelize and scale them to larger data sets, trying to better understand similarities and relationships among the published methods (e.g., [18]), or proposing further improvements – and so does this paper for the widely used PAM algorithm.

A classic method taught in textbooks is k-means (for an overview of the complicated history of k-means, refer to [3]), where the data is modeled using k cluster means, that are iteratively refined by assigning all objects to the nearest mean, then recomputing the mean of each cluster. This converges to a local optimum because the mean is the least squares estimator of location, and both steps reduce the same quantity, a measure known as sum-of-squared errors:

$$SSQ := \sum_{i=1}^{k} \sum_{x_j \in C_i} ||x_j - \mu_i||_2^2 \,. \tag{1}$$

In k-medoids, the data is modeled similarly, using k representative objects m_i called medoids (chosen from the data set; defined below) that serve as "prototypes" for the clusters in order to allow using arbitrary other dissimilarities and arbitrary input domains, using the absolute error criterion ("total deviation", TD) as objective:

$$TD := \sum_{i=1}^{k} \sum_{x_j \in C_i} d(x_j, m_i) \,, \tag{2}$$

which is the sum of dissimilarities of each point $x_j \in C_i$ to the medoid m_i of its cluster. If we use squared Euclidean as distance function (i.e., $d(x,m) = ||x-m||_2^2$), we almost obtain the usual SSQ objective used by k-means, except that k-means is free to choose any $\mu_i \in \mathbb{R}^d$, whereas in k-medoids $m_i \in C_i$ must be one of the original data points. For *squared* Euclidean distances and Bregman divergences, the arithmetic mean is the optimal choice for μ. For L_1 distance (i.e, $\sum |x_i - y_i|$), also called Manhattan distance, the component-wise median is a better choice in $\mathbb{R}^d$ [4]. For unsquared Euclidean distances, we get the much harder Weber problem [14], which has no closed-form solution [4]. For other distance functions, finding a closed form to compute the best m_i would require a separate mathematical analysis. Furthermore, our input domain is not necessarily a $\mathbb{R}^d$ vector space. In k-medoids clustering, we therefore constrain m_i to be one of our data samples. The medoid of a set C is defined as the object with the smallest sum of dissimilarities (or, equivalently, smallest average) to all other objects in the set:

$$\text{medoid}(C) := \arg\min_{x_i \in C} \sum_{x_j \in C} d(x_i, x_j) \,.$$

This definition does not require the dissimilarity to be a metric, and by using arg max it can also be applied to similarities. The algorithms discussed below all can trivially be modified to maximize similarities rather than minimizing distances, and none assumes the triangular inequality. Partitioning Around Medoids (PAM, [9]) is the most widely known algorithm to find a good partitioning using medoids, with respect to TD (Eq. 2).

2 Partitioning Around Medoids (PAM) and Its Variants

The "Program PAM" [9] consists of two algorithms, BUILD to choose an initial clustering, and SWAP to improve the clustering towards a local optimum (finding the global optimum of the k-medoids problem is, unfortunately, NP-hard). The algorithms require a dissimilarity matrix, which requires $O(n^2)$ memory and typically $O(n^2d)$ time to compute (but much more for expensive distances such as earth movers distance).

In order to find a good initial clustering, BUILD chooses k times the point which yields the smallest distance sum TD (this means first choosing the point with the smallest distance to all others; afterwards always adding the point that reduces TD most). The motivation here was to find a good starting point, in order to require fewer iterations of the refinement procedure. The second part, SWAP, improves the clustering by considering all possible simple changes to the set of k medoids, which effectively means replacing (swapping) some medoid with some non-medoid, which gives $k(n-k)$ candidate swaps. If it reduces TD, the best such change is then applied, in the spirit of a greedy steepest-descent method, and this process is repeated until no further improvements are found.

The algorithm CLARA [10]) repeatedly applies PAM on a subsample with $n' \ll n$ objects, with the suggested value $n' = 40+2k$. Afterwards, the remaining objects are assigned to their closest medoid. The run with the least TD (on the entire data) is returned. If the sample size is chosen $n' \in O(k)$ as suggested, the run time reduces to $O(k^3)$, which explains why the approach is typically used only with small k [12].

Lucasius et al. [12] propose a genetic algorithm for k-medoids, by performing a randomized exploration based on "mutation" of the best solutions found so far. The algorithm CLARANS [13] interprets the search space as a high-dimensional hypergraph, where each edge corresponds to swapping a medoid and non-medoid. On this graph it performs a randomized greedy exploration, where the first edge that reduces the loss TD is followed until no edge can be found with $p = 1.25\% \cdot k(n-k)$ attempts. Other proposals include optimizations for Euclidean space and tabu search heuristics [8].

Reynolds et al. [16] discuss an interesting trick to speed up PAM. They show how to decompose the change in the loss function into two components, where the first depends only on the medoid removed, the second part only on the new point. This decomposition forms the base for our approach, and we will thus discuss it in Sect. 3 in more detail.

Park and Jun [15] propose a "k-means like" algorithm for k-medoids (actually already considered by [16] before), where in each iteration the medoid is chosen to be the object with the smallest distance sum to other members of the cluster, then each point is assigned to the nearest medoid until TD no longer decreases. This is, unfortunately, not very effective at improving the clustering: new medoids are only chosen from within the cluster, and *have* to cover the entire current cluster. This misses many improvements where cluster members can be reassigned to *other* clusters with little cost; such improvements are considered by SWAP. Furthermore, the discrete nature of medoids makes this much more likely to get stuck in a local optimum. In our experiments this approach produced much worse results than PAM, as previously observed by [16].

3 Finding the Best Swap

The algorithm SWAP evaluates every swap of each medoid m_i with any non-medoid x_j. Recomputing the resulting TD using Eq. 2 every time requires finding the nearest medoid for every point, which causes many redundant computations. Instead, PAM only computes the *change* in TD for each object x_o if we swap m_i with x_j:

$$\Delta TD = \textstyle\sum_{x_o} \Delta(x_o, m_i, x_j) \tag{3}$$

In the function $\Delta(x_o, m_i, x_j)$ we can often detect when a point remains assigned to its current medoid (if $c_k \neq c_i$, and this distance is also smaller than the distance to x_j), and then immediately return 0. Because of space restrictions, we do not repeat the original "if" statements used in [9], but instead condense them into the equation:

$$\Delta(x_o, m_i, x_j) = \begin{cases} \min\{d(x_o, x_j), d_s(o)\} - d_n(o) & \text{if } i = \mathit{nearest}(o) \\ \min\{d(x_o, x_j) - d_n(o), 0\} & \text{otherwise} \end{cases}, \tag{4}$$

where $d_n(o)$ is the distance to the nearest medoid of o, and $d_s(o)$ is the distance to the second nearest medoid. Computing them on the fly increases the runtime by a factor of $O(k)$, but we can cache these values, and only update them when performing a swap.

Reynolds et al. [16] note that we can decompose ΔTD into: (i) the loss of removing medoid m_i, and assigning all of its members to the next best alternative, which can be computed as $\sum_{o \in C_i} d_s(o) - d_n(o)$ (ii) the (negative) loss of adding the replacement medoid x_j, and reassigning all objects closest to this new medoid. Since (i) does not depend on the choice of x_j, we can make the loop over all medoids m_i outermost, reassign all its points to the second nearest medoid (cache the distance to the now nearest neighbor), and compute the resulting loss. We then iterate over all non-medoids and compute the benefit of using them as the missing medoid instead. In the Δ function, we no longer have to consider the second nearest now (we virtually removed the old medoid already). The authors observed roughly a two-fold speedup using this approach.

Our approach is based on a similar idea of exploiting redundancy in these computations (by caching shared computations), but we instead move the loops over the medoids m_i into the *innermost* for loop. The reason for this is to further remove redundant computations. This becomes apparent when we realize that in Eq. 4, the second case does not depend on the current medoid i. If we transform the second case into an if statement, we can often avoid to iterate over all k medoids.

3.1 Making PAM SWAP Faster: FastPAM1

Algorithm 1 shows the improved SWAP algorithm. In lines 4–5 we compute the benefit of making x_j a medoid. As we do *not yet* decide which medoid to remove, we use an array of ΔTDs for each possible medoid to replace. We can now for each point compute the benefit when removing its current medoid (line 9), or the benefit if the new medoid is closer than the current medoid (line 10), which corresponds to the two cases in Eq. 4. Because the second case does not depend on i, we can replace the min statement with an if conditional *outside* of the loop in lines 10–12. After iterating over all points, we choose the best medoid, and remember the overall best swap. If we always prefer the smaller index i on ties, we choose *exactly the same* swap as the original PAM algorithm.

Assuming that the new medoid is closest in $O(1/k)$ cases on average, we can compute the change for all k medoids with $O(k \cdot 1/k) = O(1)$ effort, by saving on the innermost loop. Therefore, we expect a typical speedup on the order of $O(k)$ compared to the original PAM SWAP (but it may be hard to guarantee this for any *useful* assumption on the data distribution; the worst case supposedly remains unaffected) at the slight cost of storing one ΔTD for each medoid m_i (compared to the cost of the distance matrix and the distances to the nearest and second nearest medoids, the cost of this is negligible).

Algorithm 1. FastPAM1: Improved SWAP algorithm

```
1  repeat
2      $(\Delta TD^*, m^*, x^*) \leftarrow (0, \text{null}, \text{null})$ ;        // Empty best candidate storage
3      foreach $x_j \notin \{m_1, \ldots, m_k\}$ do
4          $d_j \leftarrow d_{\text{nearest}}(x_j)$ ;                           // Distance to current medoid
5          $\Delta TD \leftarrow (-d_j, -d_j, \ldots, -d_j)$ ;            // Change if making j a medoid
6          foreach $x_o \neq x_j$ do
7              $d_{oj} \leftarrow d(x_o, x_j)$ ;                         // Distance to new medoid
8              $(n, d_n, ds) \leftarrow (\text{nearest}(o), d_{\text{nearest}}(o), d_{\text{second}}(o))$ ;  // Cached values
9              $\Delta TD_n \leftarrow \Delta TD_n + \min\{d_{oj}, d_s\} - d_n$ ;          // Loss change
10             if $d_{oj} < d_n$ then                               // Reassignment check
11                 foreach $m_i \in \{m_1, \ldots, m_k\} \setminus m_n$ do
12                     $\Delta TD_i \leftarrow \Delta TD_i + d_{oj} - d_n$;          // Update loss change
13         $i \leftarrow \arg\min \Delta TD_i$ ;                           // Choose best medoid i
14         if $\Delta TD_i < \Delta TD^*$ then $(\Delta TD^*, m^*, x^*) \leftarrow (\Delta TD_i, m_i, x_j)$ ;   // Store
15     break loop if $\Delta TD^* \geq 0$;
16     swap roles of medoid $m^*$ and non-medoid $x^*$ ;
17     $TD \leftarrow TD + \Delta TD^*$ ;
18 return $TD, M, C$;
```

3.2 Swapping Multiple Medoids: FastPAM2

A second technique to make SWAP faster is based on the following observation: PAM will always identify the *single* best swap, then restart search; whereas the classic k-means updates all means in each iteration. Choosing the best swap has the benefit that this makes the algorithm independent of the data order [9] as long as there are no ties, and it means we need to execute fewer swaps than if we would greedily perform any swap that yields an improvement (where we may end up replacing the same medoid several times). But on the other hand, in particular for large k, we can assume that many clusters will be independent, and we could therefore update the medoids of these clusters in the same iteration, hence reduce the number of iterations by up to a factor of k.

Based on this observation, we propose to consider the best swap for *each* medoid, i.e., perform up to k swaps. This is a fairly simple modification shown in Algorithm 2, as we can use an array of swap candidates $(\Delta TD^*{}_i, x^*{}_i)$ in line 3, storing the best candidate for each current medoid m_i, and update these in line 15. After evaluating all possible swaps, we find the best swap within these up to k candidates (if we did not find a candidate, the algorithm has converged). We perform the best of these swaps in line 18. Then we recompute in line 22 for each remaining swap candidate if it still improves the clustering, otherwise this additional swap is not performed.

Algorithm 2. FastPAM2: SWAP with multiple candidates

```
1  repeat
2      foreach $x_o$ do compute nearest$(o)$, $d_{\text{nearest}}(o)$, $d_{\text{second}}(o)$; ;
3      $\Delta TD^*, x^* \leftarrow [0,\dots,0], [\text{null},\dots,\text{null}]$ ;          // Empty best candidates array
4      foreach $x_j \notin \{m_1,\dots,m_k\}$ do
5          $d_j \leftarrow d_{\text{nearest}}(x_j)$ ;                                   // Distance to current medoid
6          $\Delta TD \leftarrow (-d_j, -d_j, \dots, -d_j)$ ;                          // Change for making j a medoid
7          foreach $x_o \neq x_j$ do
8              $d_{oj} \leftarrow d(x_o, x_j)$ ;                                         // Distance to new medoid
9              $(n, d_n, ds) \leftarrow (\text{nearest}(o), d_{\text{nearest}}(o), d_{\text{second}}(o))$ ;   // Cached
10             $\Delta TD_n \leftarrow \Delta TD_n + \min\{d_{oj}, d_s\} - d_n$ ;                 // Loss change for $x_o$
11             if $d_{oj} < d_n$ then                                             // Reassignment check
12                 foreach $m_i \in \{m_1,\dots,m_k\} \setminus m_n$ do
13                     $\Delta TD_i \leftarrow \Delta TD_i + d_{oj} - d_n$;                       // Update loss change
14         foreach $i$ where $\Delta TD_i < \Delta TD^*{}_i$ do
15             $(\Delta TD^*{}_i, x^*{}_i) \leftarrow (\Delta TD_i, x_j)$ ;                      // Remember the best swap for i
16     break loop if $\min \Delta TD^* \geq 0$ ;                         // Stop if no improvements were found
17     while $i \leftarrow \arg\min \Delta TD^*$ and $\Delta TD^*{}_i < 0$ do           // Execute all improvements
18         swap roles of medoid $m_i$ and non-medoid $x^*{}_i$ ;
19         $TD \leftarrow TD + \Delta TD^*{}_i$ ;
20         $\Delta TD^*{}_i \leftarrow 0$ ;                                     // Disable the swap just performed
21         foreach $j$ where $\Delta TD^*{}_j < 0$ do                           // For remaining swap candidates
22             $\Delta TD^*{}_j \leftarrow \sum_{x_o \notin \{m_1,\dots,m_k\} \setminus m_j} \Delta(x_o, m_j, x^*{}_j)$ ;          // Recompute $TD$
23 return $TD, M, C$;
```

The improvements of this strategy are, unsurprisingly, much smaller than those of FastPAM1. In early iterations we see multiple swaps being executed, but in the later iterations it is common that only few medoids change. Nevertheless, this simple modification yields another measurable performance improvement. However—in contrast to the first improvement—this no longer guarantees to yield the same result. From a theoretical point of view, both the original PAM, and FastPAM2 perform a steepest descent optimization strategy; where PAM only permits descends consisting of a single swap, whereas FastPAM2 can perform multiple swaps at once as long as they use different medoids. Therefore, both are able to find results of equivalent quality. In our experiments, FastPAM2 would often find *marginally better* results than PAM, and faster.

3.3 Faster Initialization with Linear Approximative BUILD (LAB): FastPAM

With these optimizations to SWAP, reducing the time from $O(k(n-k)^2)$ to $O((n-k)^2)$, the bottleneck of PAM becomes the BUILD phase. In our experiments with large k, PAM would spend 99% of the run time in SWAP. With above optimizations this reduces to about 15%. About 16% is the time to compute the distance matrix, and 69% of the time is spent in BUILD. The complexity of BUILD is in $O(kn^2)$, so for large k this is expected to happen. Because we made SWAP much faster, we can afford to begin with slightly worse starting conditions, even if we need more iterations of SWAP afterwards.

Algorithm 3. FastPAM LAB: Linear Approximate BUILD initialization.

```
1  $(TD, m_1) \leftarrow (\infty, \text{null})$;
2  $S \leftarrow$ subsample of size $10 + \lceil\sqrt{n}\rceil$ from $X$ ;                    // Subsample
3  foreach $x_j \in S$ do                                                                  // First medoid
4      $TD_j \leftarrow 0$ ;
5      foreach $x_o \in S \wedge x_o \neq x_j$ do $TD_j \leftarrow TD_j + d(x_o, x_j)$;
6      if $TD_j < TD$ then $(TD, m_1) \leftarrow (TD_j, x_j)$;          // Smallest distance sum
7  for $i = 1 \ldots k-1$ do                                                               // Other medoids
8      $(\Delta TD^*, x^*) \leftarrow (\infty, \text{null})$;
9      $S \leftarrow$ subsample of size $10 + \lceil\sqrt{n}\rceil$ from $X \setminus \{m_1, \ldots, m_i\}$ ;   // Subsample
10     foreach $x_j \in S$ do
11         $\Delta TD \leftarrow 0$ ;
12         foreach $x_o \in S \wedge x_o \neq x_j$ do
13             $\delta \leftarrow d(x_o, x_j) - \min_{o \in m_1, \ldots, m_i} d(x_o, o)$;
14             if $\delta < 0$ then $\Delta TD \leftarrow \Delta TD + \delta$;
15         if $\Delta TD < \Delta TD^*$ then $(\Delta TD^*, x^*) \leftarrow (\Delta TD, x_j)$;   // best reduction
16     $(TD, m_{i+1}) \leftarrow (TD + \Delta TD^*, x^*)$;
17 return $TD, \{m_1, \ldots, m_k\}$;
```

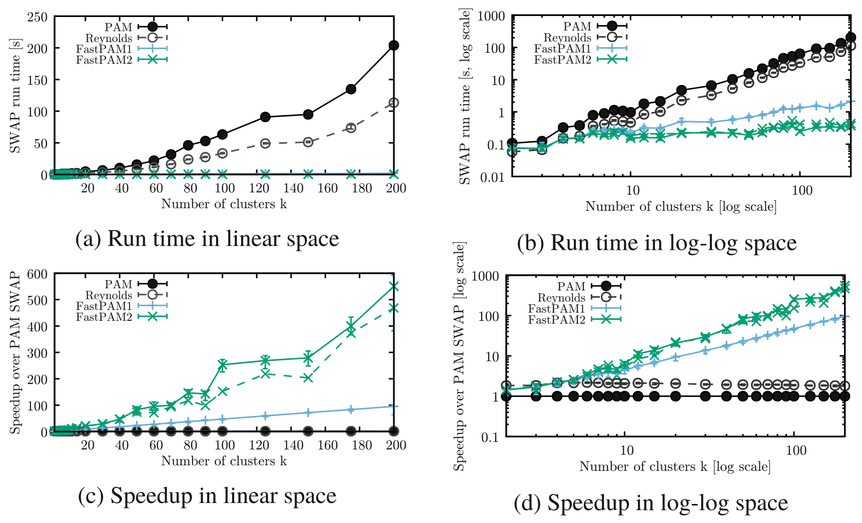
An elegant way of initializing k-means is k-means++ [2]. The beautiful idea of this approach is to choose seeds with the probability proportional to their squared distance to the nearest seed (the first seed is picked uniformly). If we assume there is a cluster of points and no seed nearby, the probability mass of this cluster is substantial, and we are likely to place a seed there; afterwards the probability mass of this cluster reduces. Furthermore, this initialization is (in expectation) $O(\log k)$ competitive to the optimal solution, so it will theoretically generate good starting conditions. But as seen in our experiments, this guarantee is pretty loose; and BUILD empirically produces much better starting conditions than k-means++ (we are not aware of a detailed theoretical analysis). The reason is that k-means++ picks *random* points (usually) from different clusters, but makes no effort to find good centers of the clusters (which is not that important for k-means, where the mean is in between of the data points). Therefore, with k-means++-style initialization we need around k additional swaps to pick the medoid of each cluster (and hence, k iterations of original PAM SWAP, although much fewer with FastPAM2). Because a single iteration of swap used to take as much time as BUILD, the k-means++ initialization only begins to shine if we use FastPAM1 to reduce the cost of iterating together with the FastPAM2 strategy of doing as many swaps as possible.

We experimented with k-means++, but eventually settled for a different strategy we call LAB (Linear Approximative BUILD). What we title "FastPAM" then is the combination of LAB with the optimizations of FastPAM2. As the name indicates, LAB is a linear approximation of the original PAM BUILD. In order to achieve linear runtime in n, we simply subsample the data set. Before choosing each medoid, we sample $10 + \lceil\sqrt{n}\rceil$ points from all non-medoid points. From this subsample we choose the one with the largest decrease ΔTD with respect to the current subsample only. Results were slightly better with sampling k times, and not just once; since each object has k chances of being in the sample, and if we draw a bad sample it only affects a single medoid. A pseudocode of LAB is given as Algorithm 3. Clearly, the complexity is down to $O(kn)$.

3.4 Integration: FastCLARA and FastCLARANS

Since CLARA [10] uses PAM as a subroutine, we can trivially use our improved FastPAM with CLARA. In the experiments we will denote this variant as FastCLARA.

CLARANS [13] uses a randomized search instead of considering all possible swaps. For this, it chooses a random pair of a non-medoid object and a medoid, computes whether this improves the current loss, and then greedily performs this swap. Adapting the idea from FastPAM1 to the random exploration approach of CLARANS, we pick only the non-medoid object at random, but can consider all medoids at a similar cost to looking at a single medoid. This means we can either explore k times as many edges of the graph, or we can reduce the number of samples to draw by a factor of k. In our experiments we opted for the second choice, to make the results comparable to the original CLARANS in the number of edges considered; but as the edges chosen involve the same non-medoids, we



(a) Run time in linear space

(b) Run time in log-log space

(c) Speedup in linear space

(d) Speedup in log-log space

Fig. 1. Run time of PAM SWAP (SWAP only, without DAISY, without BUILD)

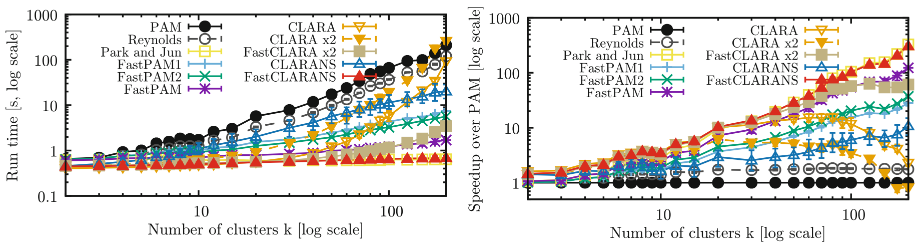
expect a slight loss in quality that should be easily countered by increasing the subsampling rate of non-medoids. By varying the subsampling rate, the user can control the tradeoff between computation time and exploration.

4 Experiments

Theoretical considerations show that we must expect an $O(k)$ speedup of FastPAM1 over the original PAM algorithm, so our experiments primarily need to verify that there is no trivial error (in contrast to much work published in recent years, the speedup is not just empirical). Nevertheless constant factors and implementation details can make a big difference [11], and we want to ensure that we do not pay big overheads for theoretical gains that would only manifest for infinite data.[1] For FastPAM2 the speedup is expected to be only a small factor due to the reduction in iterations. In contrast to FastPAM1, it does not guarantee the exact same results; therefore we also want to verify that they are of the expected equivalent quality. The worse starting conditions of LAB should not affect the final result, but will require additional iterations of SWAP. We observed increased runtimes when using k-means++ for PAM initialization, therefore it needs to be verified experimentally that LAB does not require excessive additional iterations.

We showcase results from the "one-hundred plant species leaves" data set (texture features only) from the well-known UCI repository [5], but we verified

[1] Clearly, our $O(k)$ fold speedup must be immediately measurable, not just asymptotically, because the constant overhead for maintaining the fixed array cache is small.



(a) Run time in log-log space

(b) Speedup in log-log space

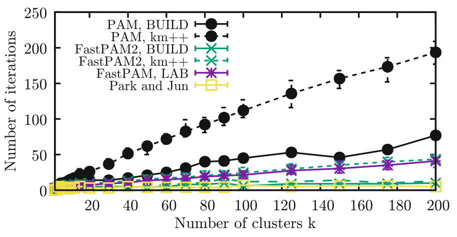
Fig. 2. Run time comparison of different variations and derived algorithms.

the results on additional data sets (not included because of space restrictions). We chose this data set because it has 100 classes, and 1600 instances, a fairly small size that PAM can still easily handle. Naively, one would expect that $k = 100$ is a good choice on this data set, but some leaf species are likely not distinguishable by unsupervised learning. We used the ELKI open-source data mining toolkit [20] in Java to develop our version. For comparison, we also ported FastPAM2 to the R `cluster` package, which is based on the original PAM source code and written in C. Experiments were run on an Intel i7-7700 at 3.6 GHz with turbo boost disabled. We perform 25 runs, and plot the average, minimum and maximum. Both implementations and all data sets show similar behavior, so we are confident that the results are not just due to implementation differences [11].

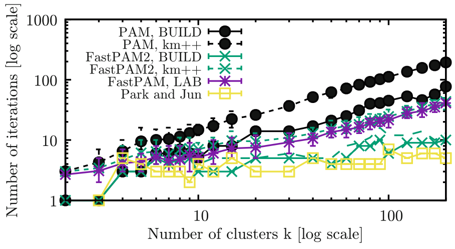
4.1 Run Time Speedup

In Fig. 1, we vary k from 2 to 200, and plot the run time of the PAM SWAP phase *only* (the cost of computing the distance matrix and the BUILD phase is not included), using the original PAM, the Reynolds version, and the proposed improvements. Figure 1a shows the run time in linear space, to visualize the drastic run time differences observed. Reynolds' was quite consistently two times faster than the original PAM; but our proposed methods were faster by a factor that grows approximately linearly with the number of clusters k. In log-log-space, Fig. 1b, we can differentiate the three variants studied.

In Fig. 1c we plot the speedup over PAM. Reynolds' SWAP clearly was about twice as fast as the original PAM. The FastPAM1 improvement gives an empirical speedup factor of about $\frac{1}{2}k$, while the second improvement contributed an additional speedup of about 2–2.5× by reducing the number of iterations. Because of the multiplicative effect of these savings, the linear plot in Fig. 1c gives the false impression that this second contribution yields the larger benefit. The logspace plot in Fig. 1d more accurately reflects the contribution of the two factors, resulting in a speedup of over 250 times at $k = 150$; while at $k = 2$ and $k = 3$ the speedup was just 1.4× resp. 1.75×, and less than our implementation of Reynolds (in R, our method is as fast as Reynolds for $k = 2$). In the most extreme case tested, a speedup of about 1000× at $k = 200$ is measured – but because the

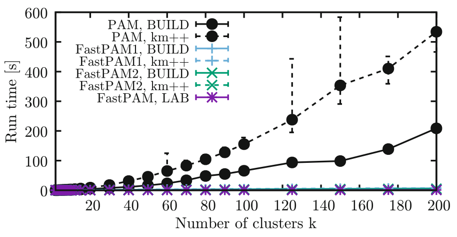


(a) Iterations in linear space

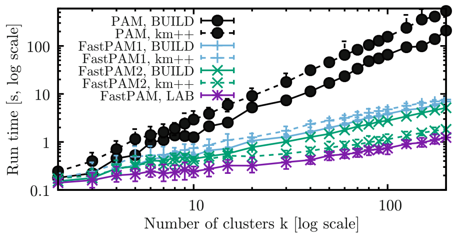


(b) Iterations in log-log space

Fig. 3. Number of iterations for PAM vs. FastPAM2 and BUILD vs. LAB initialization



(a) Initialization runtime [linear]



(b) Initialization runtime [log-log]

Fig. 4. Runtime impact of k-means++ and LAB initialization

speedup depends on $O(k)$, the exact values are meaningless, furthermore, we excluded the distance matrix computation and initialization in this experiment.

In Fig. 2, we study the run time of approximations to PAM (including the distance matrix computation and initialization time now). We only present the log-log space plots, because of the extreme differences. The run time of CLARA, as k increases, approaches the run time of PAM. This is expected, because the subsample size for CLARA is chosen as $40 + 2k$, and necessary because the subsample size needs to be sufficiently larger than k. For CLARA x2 we also evaluate doubling this value to $80 + 4k$, and we also double the number of restarts from 5 to 10. CLARA x2 is thus expected to take 8 times longer, but should give better results. FastCLARA is CLARA using our FastPAM approach, and performs much better, but for large k also eventually becomes slower than FastPAM. The run time of CLARANS on this data set (see later for CLARANS problems) is in between the original PAM and CLARA, and with our optimizations FastCLARANS becomes the fastest method tested (at similar quality to CLARANS, and with the same problems). Park and Jun's [15] approach is similarly fast to FastCLARANS for large k, but its quality is quite poor, as we will see and discuss in Sect. 4.3.

4.2 Number of Iterations

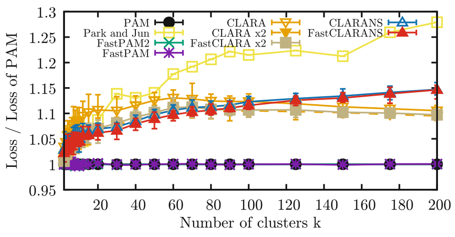
We are not aware of theoretical results on the number of iterations needed for PAM. Based on results for k-means, we must assume that the worst case is superpolynomial like k-means [1], albeit in practice a "few" iterations are usually enough. Because of this, we are also interested in studying the number of iterations.

Figure 3 shows the number of iterations needed with different methods, both in linear space and log space. In line with previous empirical results, only few iterations are necessary. Because PAM only performs the best swap in each iteration, a linear dependency on k is to be assumed; interestingly enough we usually observed much less than k iterations, so many medoids remain unchanged from their initial values (note that this may be due to the rather small data set size, too). The k-means++ initialization required roughly 2–4$\times$ as many iterations for PAM; with the original algorithm where each iteration would cost about as much as the BUILD initialization, this choice is detrimental even for small k. With the improvements of this paper, these additional iterations are cheaper than the rather slow BUILD initialization by a factor of $O(k)$ now, hence we can now begin with a worse but cheaper starting point. Furthermore, the FastPAM2 approach which performs up to k swaps in each iteration does reduce the number of iterations substantially. FastPAM2 with BUILD performed the second-lowest numbers of iterations. Our proposed LAB initialization of FastPAM saves a few extra iterations compared to the k-means++ strategy, at better initial quality, and hence is measurably faster in the end. Park and Jun [15] at first seems to perform very well in this figure, with slightly fewer iterations than FastPAM2 with BUILD. Unfortunately, this is because the "k-means style" algorithm misses many improvements to the clustering, and hence produces much worse results as we will observe next.

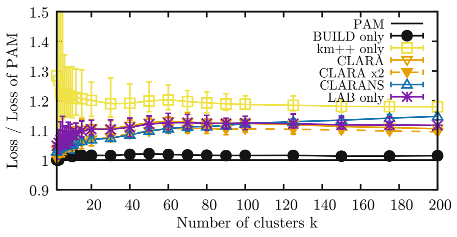
In Fig. 4 we revisit the runtime experiment, and focus on initialization. As we can see, the increased number of iterations hurts runtime with the original PAM algorithm as well as its Reynolds variant substantially (the reasons for this are explained in Sect. 3.3); for FastPAM1, the use of k-means++ only comes at a small performance penalty (while it still needs as many iterations as the original PAM, these have become $O(k)$ times faster, and the initialization cost begins to matter much more), and with FastPAM2's ability to perform multiple swaps per iteration, a linear-time initialization such as the proposed LAB clearly becomes the preferred initialization method, in particular for large k.

4.3 Quality

Any algorithmic change and optimization comes at the risk of breaking some things, or negatively affecting numerics (see, e.g., [17] on how common numerical issues are even with basic statistics such as variance in SQL databases). In order to check for such issues, we made sure that our implementations pass the same unit tests as the other algorithms in both ELKI and R. We do not expect numerical problems, and Reynolds' variant and FastPAM1 are supposed



(a) Loss (TD) of final clustering



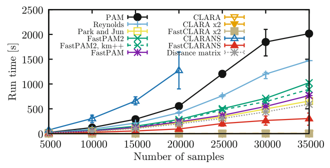
(b) Loss (TD) of BUILD, k-means++ and LAB

Fig. 5. Loss (TD) compared to PAM

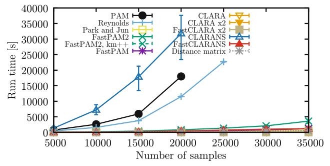
to give the same result (and do so in the experiments, so we exclude them from the plot). The FastPAM2 algorithm is greedy in performing swaps, and may therefore converge to a different solution, but of the same quality.

In Fig. 5a we visualize the loss, i.e., the objective TD, of different approximations compared to the solution found by the original PAM approach (which is not necessarily the global minimum). For large k, the solution found by the approach of Park and Jun [15] is over 25% worse here, for the reasons discussed before. Our strategy FastPAM2 gives results comparable to PAM as expected (sometimes slightly better, sometimes slightly worse). The cheaper LAB initialization (full FastPAM) does not cause a noticeable loss in quality either, but further improves the total run time. CLARA (which only uses a subsample of the data) finds considerably worse results. By doubling the subsample size to $80 + 4k$ and doing twice as many restarts (CLARA x2) the results only improve slightly for large k (but much more for small k). CLARA x2 is until about $k = 70$ as good as CLARANS here, but faster; for larger k it becomes even better than CLARANS, but also slower. FastCLARA has the same quality as CLARA x2 (we use the x2 parameters, too), but it was much faster. FastCLARANS is slightly better than CLARANS, and was considerably faster. All the CLARANS results degrade with increasing k, so it may become necessary to increase the subsample size there, which will increase the run time (it is up to the user to choose his preferences, quality or runtime). In conclusion, all our "Fast" approaches perform as well as their older counterparts, but are $O(k)$ times faster.

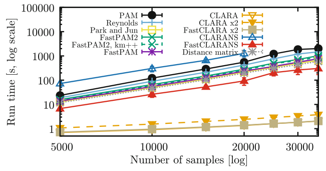
In Fig. 5b, we evaluate the quality of LAB, k-means++, and BUILD initialization compared to the converged PAM result. As seen in the previous experiments, all three initializations will yield similar results after PAM, but we can compare the quality of the initial medoids to the full PAM result. As we can see, the BUILD approach produces the best initial results (and as noted by [9], the BUILD result may be usable without further refinement). While k-means++ offers some theoretical advantages (c.f., Sect. 3.3), the initial result is quite bad as this strategy only attempts to pick a random point from each cluster, and not the medoids. Our proposed LAB initialization is in between k-means++ and BUILD, and by itself performs similar to CLARA. As it only considers a subset of the data, its medoids will be worse than BUILD; but because it chooses the



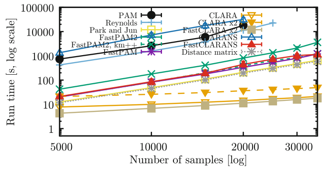
(a) Run time with $k = 10$



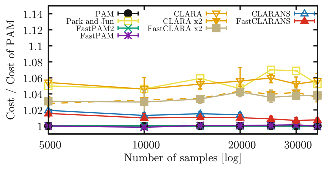
(d) Run time with $k = 100$



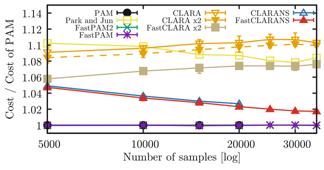
(b) Run time [log-log]



(e) Run time [log-log]



(c) Quality compared to PAM



(f) Quality compared to PAM

Fig. 6. Results on MNIST data with $k = 10$ (left) and $k = 100$ (right)

best medoid of the sample it performs better than k-means++. As it reduces the runtime for $O(n^2k)$ to $O(nk)$ it is the preferred choice for FastPAM nevertheless.

4.4 Scalability Experiments

Just as PAM, our method also requires the entire distance matrix to be precomputed. This will require $O(n^2)$ time and memory, making the method as-is unsuitable for big data (for real big data problems, it will however often be enough to cluster a subsample that fits into memory). Our improvements focus on reducing the dependency on k, but we nevertheless experimented with scalability in n, too (and we already included FastCLARA and FastCLARANS in the previous experiments). The behavior of the PAM variants is as expected $O(n^2)$, but we see nevertheless quite big differences between PAM, FastPAM, and sampling-based approaches. In this experiment, we use the well-known MNIST data set from the UCI repository [5], which has 784 variables (each corresponding to a pixel in a 28 × 28 grid) and 60000 instances. We used the first $n = 5000, 10000, \ldots, 35000$ instances with a time limit of 6 h and compare $k = 10$ and $k = 100$. The high number of variables makes this data set expensive for CLARANS.

The problem of quadratic runtime is best seen in the linear scale plots Fig. 6a and d. As a reference, we give the time needed for computing the distance matrix as dotted line, which is also quadratic. Except for CLARA and FastCLARANS, the runtime is dominated by computing the distance matrix (and hence CLARA, which only uses a constant-size sample, shines for large n). The original CLARANS suffers from excessive distance re-computations. The authors assumed that distances are cheap to compute, and noted that it may be necessary to cache the distances. FastCLARANS reduces the number of distance computations of CLARANS by a factor of $O(k)$, and is still cheaper than the full distance matrix here. For more expensive distances such as dynamic time warping, FastPAM will outperform FastCLARANS, and it will almost always give better results. For $k = 10$, only CLARANS, PAM and Reynolds' variant are problematic at this data size, but at $k = 100$ the benefits of our improvements become very noticeable. The CLARA methods are squeezed to the axis in the linear plot, and hence we also provide log-log plots in Fig. 6b and e. For $k = 10$, the lines of CLARA and FastCLARA x2 almost coincide by chance (note that FastCLARA x2 produces a result comparable to the slower CLARA x2 method; expected to be 8 times slower), but at $k = 100$ it is faster than CLARA demonstrating that our improvements also accelerate CLARA.

While the scalability in n is quadratic, we observe that if you can afford to compute the pairwise distance matrix, then you will *now* also be able to run FastPAM. For $k = 10$, the additional runtime of FastPAM was about 30% the runtime of computing the distance matrix computation, and at $k = 100$ FastPAM took about as much time as the distance matrix. Hence, if you can compute the distance matrix, you can also run FastPAM for reasonable values of $k \ll n$, and the main scalability problem is the memory consumption of the distance matrix.

If computing the distance matrix is prohibitive, it may still be possible to use FastCLARA, which is $O(k)$ times faster than CLARA, and will scale linearly in n. But as seen in Fig. 6c and f, CLARA will usually give worse results (about 10% in our experiments). For many users this difference will be acceptable, as a clustering result is never "perfect". For large data sets, FastCLARANS will usually give better results, unless the sample size of CLARA is increased considerably. But on the other hand, FastCLARANS is only advisable for inexpensive distance functions such as (low-dimensional) Euclidean distance, and requires a non-trivial distance cache otherwise.

5 Conclusions

In this article we proposed a modification of the popular PAM algorithm that typically yields an $O(k)$ fold speedup, by clever caching of partial results in order to avoid recomputation. This caching was enabled by changing the nesting order of the loops in the algorithm, showing once more how much seemingly minor looking implementation details can matter [11]. As a second improvement, we propose to find the best swap for each medoid, and execute as many as possible in each iteration, which reduces the number of iterations needed for convergence without loss of quality.

The major speedups obtained enable the use of this classic clustering method on much larger data, in particular with large k. With the faster refinement procedure, it now pays off to use cheaper initialization methods with PAM. For this, we propose LAB initialization, a linear-time approximation of the original PAM BUILD algorithm.

Methods based on PAM, such as CLARA, CLARANS, and the many parallel and distributed variants of these algorithms for big data, all benefit from this improvement, as they either use PAM as a subroutine (CLARA), or employ a similar swapping method (CLARANS) that can be modified accordingly as seen in Sect. 3.4.

The proposed methods are included in the open-source framework ELKI 0.7.5 [20], and FastPAM2 (but not yet LAB, FastCLARA, nor FastCLARANS) is included in the R `cluster` package 2.0.9, to make it easy for others to benefit from these improvements. With the availability in two major clustering tools, we hope that many users will find using PAM possible on much larger data sets with higher k than before.

References

1. Arthur, D., Vassilvitskii, S.: How slow is the k-means method? In: ACM Symposium on Computational Geometry (2006). https://doi.org/10.1145/1137856.1137880
2. Arthur, D., Vassilvitskii, S.: k-means++: the advantages of careful seeding. In: ACM-SIAM SODA (2007)
3. Bock, H.: Clustering methods: a history of k-means algorithms. In: Brito, P., Cucumel, G., Bertrand, P., de Carvalho, F. (eds.) Selected Contributions in Data Analysis and Classification. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73560-1_15
4. Bradley, P.S., Mangasarian, O.L., Street, W.N.: Clustering via concave minimization. In: NIPS (1996)
5. Dheeru, D., Karra Taniskidou, E.: UCI machine learning repository (2017). http://archive.ics.uci.edu/ml
6. Ester, M., Kriegel, H., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: KDD (1996)
7. Estivill-Castro, V.: Why so many clustering algorithms: a position paper. SIGKDD Explor. **4**(1), 65–75 (2002). https://doi.org/10.1145/568574.568575
8. Estivill-Castro, V., Houle, M.E.: Robust distance-based clustering with applications to spatial data mining. Algorithmica **30**(2), 216–242 (2001). https://doi.org/10.1007/s00453-001-0010-1

9. Kaufman, L., Rousseeuw, P.J.: Clustering by means of medoids. In: Dodge, Y. (ed.) Statistical Data Analysis Based on the L_1 Norm and Related Methods (1987). ISBN 0444702733
10. Kaufman, L., Rousseeuw, P.J.: Clustering large data sets. In: Pattern Recognition in Practice. Elsevier (1986). https://doi.org/10.1016/b978-0-444-87877-9.50039-x
11. Kriegel, H., Schubert, E., Zimek, A.: The (black) art of runtime evaluation: are we comparing algorithms or implementations? Knowl. Inf. Syst. **52**(2), 341–378 (2017). https://doi.org/10.1007/s10115-016-1004-2
12. Lucasius, C., Dane, A., Kateman, G.: On k-medoid clustering of large data sets with the aid of a genetic algorithm. Anal. Chim. Acta **282**(3), 647–669 (1993). https://doi.org/10.1016/0003-2670(93)80130-D
13. Ng, R.T., Han, J.: CLARANS: a method for clustering objects for spatial data mining. IEEE TKDE **14**(5), 1003–1016 (2002). https://doi.org/10.1109/TKDE.2002.1033770
14. Overton, M.L.: A quadratically convergent method for minimizing a sum of euclidean norms. Math. Program. **27**(1), 34–63 (1983). https://doi.org/10.1007/BF02591963
15. Park, H., Jun, C.: A simple and fast algorithm for k-medoids clustering. Expert Syst. Appl. **36**(2), 3336–3341 (2009). https://doi.org/10.1016/j.eswa.2008.01.039
16. Reynolds, A.P., Richards, G., de la Iglesia, B., Rayward-Smith, V.J.: Clustering rules: a comparison of partitioning and hierarchical clustering algorithms. J. Math. Model. Algorithms **5**(4), 475–504 (2006). https://doi.org/10.1007/s10852-005-9022-1
17. Schubert, E., Gertz, M.: Numerically stable parallel computation of (co-)variance. In: SSDBM (2018). https://doi.org/10.1145/3221269.3223036
18. Schubert, E., Hess, S., Morik, K.: The relationship of DBSCAN to matrix factorization and spectral clustering. In: LWDA. CEUR Workshop Proceedings, vol. 2191 (2018)
19. Schubert, E., Sander, J., Ester, M., Kriegel, H., Xu, X.: DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. ACM Trans. Database Syst. **42**(3), 19 (2017). https://doi.org/10.1145/3068335
20. Schubert, E., Zimek, A.: ELKI: a large open-source library for data analysis - ELKI release 0.7.5 “Heidelberg”. arXiv:1902.03616 (2019)