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

Erich Schubert
Technische Universität Dortmund, Dortmund, Germany
erich.schubert@tu-dortmund.de

Peter J. Rousseeuw
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 PAM, partitioning around medoids, also known 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 hold 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 such as biology that require the use of Jaccard, Gower, or even more complex distances.

A key issue with PAM is, however, its high run time cost. In this paper, we propose modifications to the PAM algorithm where at the cost of storing $O(k)$ additional values, we can 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. We also show how the CLARA and CLARANS algorithms benefit from this modification.

In experiments on real data with k=100, we observed a 200× speedup compared to the original PAM SWAP algorithm, making PAM applicable to larger data sets, and in particular to higher k.

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" [6], meaning that every user may have different enough needs and intentions to want a different algorithm and notion of cluster. And therefore, hundreds of clustering algorithms and evaluation measures have been proposed over the years, each with their merits and drawbacks. Nevertheless, a few seminal methods such as hierarchical clustering, k-means, PAM [9, 11], and DBSCAN [5] have received repeated and widespread use. One may be tempted to think that after 60 to 20 years these methods have all been well researched and understood, but there are still many scientific publications trying to explain them better (e.g., [22]), trying to understand similarities and relationships among those methods (e.g., [20]), or proposing further improvements – and so does this paper for the widely used PAM algorithm, also known as k-medoids.

In hierarchical agglomerative clustering (HAC), each object is initially its own cluster. The two closest clusters are then merged repeatedly to build a cluster tree called dendrogram. HAC is a very flexible method: it can be used with any distance or (dis-)similarity, and it allows for different rules of aggregating the object distances into cluster distances, such as the minimum ("single linkage"), average, or maximum ("complete linkage"). Minimum linkage corresponds to the minimum spanning tree of the distance graph. While the dendrogram is a powerful visualization for small data sets, extracting partitions from hierarchical clustering is not trivial, and thus users often turn to simpler methods.

Another classic method taught in textbooks is k-means (for the complicated history of k-means, refer to Bock [2]), 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 optimization converges because the mean is the least squares estimator of location, and both steps optimize the same quantity, a measure known as sum-of-squared errors (SSQ, also called SSE, and equivalent to WCSS):

$$SSQ := \sum_{i=1}^{k} \sum_{x_j \in C_i} ||x_j - \mu_i||_2^2 \ . \tag{1.1}$$

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

$$TD := \sum_{i=1}^{k} \sum_{x_j \in C_i} d(x_j, m_i) \ , \tag{1.2}$$

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 and a fixed cluster assignment. For L_1 distance (i.e, $\sum |x_i - y_i|$), also called Manhattan distance, the component-wise median is a better choice [3, k-medians]. For unsquared Euclidean distances,[1] we get the much harder Weber problem [16], which has no closed-form solution [3] (for a recent survey of algorithms for the Weber point see [7]). For other distance functions, finding a closed form to compute the best m_i would require non-trivial mathematical analysis of each distance function separately. Furthermore, our input data does not necessarily come from 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:

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

This definition does not require the dissimilarity to be a metric, and by using $\arg\max$ instead of the $\arg\min$ it can also be applied to similarities. The algorithms discussed below all can be trivially be modified to maximize similarities rather than minimizing distances, none assumes the triangular inequality. Partitioning Around Medoids (PAM, [9, 11]) is the most widely known algorithm to find a good partitioning using medoids, with respect to TD (Equation 1.2).

2 Partitioning Around Medoids (PAM, [9, 11])

The "Program PAM" [11] consists of two algorithms, BUILD to choose an initial clustering, and the SWAP stage to further optimize 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 for many popular distance functions in d dimensional data $O(n^2 d)$ time to compute (but potentially much more for expensive distances such as earth movers distance).

2.1 BUILD Initialization Algorithm In order to find a good initial clustering (rather than relying on a random sampling strategy as commonly used with k-means), we choose k times the point which yields the smallest distance sum TD (in the first iteration, this means choosing the point with the smallest distance to all others; afterwards adding the point as next medoid, that reduces TD most). We give a pseudocode in Algorithm 1, where we use ΔTD as symbol for the change in TD (which should be negative to be beneficial), and * for the best values found so far. If we cache the nearest medoid in line 11, then this initialization needs $O(n^2 k)$ time, so it already is a fairly expensive algorithm. The motivation here was to find a good starting point, in order to require fewer iterations of the refinement procedure. In the experiments, we will also study whether a clever sampling-based approach similar to k-means++ [1] that needs only $O(nk)$ time is an interesting alternative.

Algorithm 1: PAM BUILD: Find initial cluster centers.

```
1  (TD, m_1) ← (∞, ⊥);
2  foreach x_j do                                   // First medoid
3  |  TD_j ← 0 ;
4  |  foreach x_o ≠ x_j do  TD_j ← TD_j + d(x_o, x_j);
5  |  if TD_j < TD then  (TD, m_1) ← (TD_j, x_j);
6  for i = 1 ... k − 1 do                           // Other medoids
7  |  (ΔTD*, x*) ← (∞, ⊥);
8  |  foreach x_j ∉ {m_1, ..., m_i} do
9  |  |  ΔTD ← 0 ;
10 |  |  foreach x_o ∉ {m_1, ..., m_i, x_j} do
11 |  |  |  δ ← d(x_o, x_j) − min_{o∈m_1,...,m_i} d(x_o, o);
12 |  |  |  if δ < 0 then ΔTD ← ΔTD + δ;
13 |  |  if ΔTD < ΔTD* then
14 |  |  |  (ΔTD*, x*) ← (ΔTD, x_j);
15 |  (TD, m_{i+1}) ← (TD + ΔTD*, x*);
16 return TD, {m_1, ..., m_k};
```

Algorithm 2: PAM SWAP: Iterative improvement.

```
1  repeat
2  |  (ΔTD*, m*, x*) ← (0, ⊥, ⊥) ;
3  |  foreach m_i ∈ {m_1, ..., m_k} do
4  |  |  foreach x_j ∉ {m_1, ..., m_k} do
5  |  |  |  ΔTD ← 0 ;
6  |  |  |  foreach x_o ∉ {m_1, ..., m_k} \ m_i do
7  |  |  |  |  ΔTD ← ΔTD + Δ(x_o, m_i, x_j)
8  |  |  |  if ΔTD < ΔTD* then
9  |  |  |  |  (ΔTD*, m*, x*) ← (ΔTD, m_i, x_j)
10 |  break loop if ΔTD* ≥ 0 ;
11 |  swap roles of medoid m* and non-medoid x* ;
12 |  TD ← TD + ΔTD* ;
13 return TD, M, C;
```

[1] It is a common misconception that k-means would minimize Euclidean distances. It optimizes the sum of *squared* Euclidean distances, and even then the textbook algorithm may end up slightly off a local optimum, because always assigning a point to its nearest center *can* increase the variance by moving the centers away from other points [8].

changes to the set of k medoids, which effectively means replacing (swapping) some medoid with some other 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 steepest-descent method, and this process is repeated until no further improvements are found. We give a pseudocode of this in Algorithm 2. If we again cache the necessary data to compute the $\Delta(x_o, m_i, x_j)$ function (Equation 3.4, explained in Section 3) in line 7 efficiently, then the run time of this algorithm is $O(k(n-k)^2)$ for each iteration. While the authors of PAM assumed that only few iterations will be needed (if the algorithm is already initialized well, using the BUILD algorithm above), we do see an increasing number of iterations with increasing amounts of data (but usually we will have fewer than k iterations).

Both the pseudocode for BUILD and SWAP given here omit the details of managing the cached distances. For each object, we need to store the index of the nearest medoid $\mathit{nearest}(o)$ and its distance $d_{\mathit{nearest}}(o)$, and also the distance to the second nearest medoid $d_{\mathit{second}}(o)$ (if we also store the index of the second nearest center, we may be able to avoid some more distance computations). In particular in line 11, when executing the best swap, we need to carefully update the cached values.

2.3 Variants of PAM The algorithm CLARA [12, 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 [14]. Because CLARA uses PAM internally, it will directly benefit from the improvements proposed in this article.

Lucasius, Dane, and Kateman [14] propose a genetic algorithm to find the best k-medoids partitioning, which will perform a randomized exploration of the search space based on the best solutions found so far. Crossover mutations correspond to taking some medoids from both "parents", whereas mutations replace medoids with random objects. It is not obvious that this will efficiently provide a sufficient coverage of the enormous search space (there are $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ possible sets of medoids) for a large k. In order to benefit from the proposed improvements, a more systematic mutation strategy would need to be adopted, making the method similar to CLARANS below. Wei, Lee, and Hsu [23] found the genetic methods to work only for small data sets, small k, and well separated symmetric clusters, and it was usually outperformed by CLARANS.

The algorithm CLARANS [15] 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 cost TD is followed until no edge can be found with $p = 1.25\% \cdot k(n-k)$ attempts. In Section 3.4 we will outline how our approach can be used to explore k edges at a time efficiently; this will allow exploring a larger part of the search space in similar time, but we expect the savings to be relatively small compared to PAM.

Reynolds et al. [18] discuss an interesting trick to speed up PAM. They show how to decompose the change in the cost 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 Section 3 in more detail.

Park and Jun [17] propose a "k-means like" algorithm for k-medoids (which actually was already considered by Reynolds et al. [18] 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 the distances no longer decrease. While this works reasonably well on idealized data, it relies very much on a good initialization, because it is not very effective at further 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, however, found by PAM. Furthermore, the means used in k-means change with every point we move to a different cluster, but the medoids will very often remain the same, they are too coarse for this optimization strategy. In our experiments this approach produced noticeably worse results than PAM, in line with the earlier observations by Reynolds et al. [18]. The paper also contributes an $O(n^2)$ initialization, that unfortunately tends to choose all initial medoids close to the center of the data set. Choosing cluster medoids with a k-means like strategy will take $O(n^2)$ time because we have to assume the clusters to be unbalanced, and contain up to $O(n)$ objects; nevertheless this is k times faster than PAM. But because it does *not* consider reassigning points to other clusters when choosing the new medoid, this approach will miss many improvements found by PAM. This reduces the number of iterations, but also produces worse results.

3 Finding the Best Swap

The algorithm SWAP evaluates every swap of any medoid m_i with any non-medoid x_j. Recomputing the resulting TD using Equation 1.2 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 separately:

$$\Delta TD = \sum\nolimits_{x_o} \Delta(x_o, m_i, x_j) \tag{3.3}$$

Algorithm 3: PAM+: Improved SWAP algorithm

```
1  repeat
2      (ΔTD*, m*, x*) ← (0, ⊥, ⊥) ;                          // Empty best candidate storage
3      foreach x_j ∉ {m_1, ..., m_k} do
4          d_j ← d_nearest(x_j) ;                             // Distance to current medoid
5          ΔTD ← (−d_j, −d_j, ..., −d_j) ;                    // Cost change for making j a medoid
6          foreach x_o ≠ x_j do
7              d_oj ← d(x_o, x_j) ;                           // Distance to new medoid
8              (n, d_n, ds) ← (nearest(o), d_nearest(o), d_second(o)) ;   // Cached
9              ΔTD_n ← ΔTD_n + min{d_oj, d_s} − d_n ;         // Cost change for current
10             if d_oj < d_n then                             // Reassignment check
11                 foreach m_i ∈ {m_1, ..., m_k} \ m_n do
12                     ΔTD_i ← ΔTD_i + d_oj − d_n;            // Update cost change
13         i ← arg min ΔTD_i ;                                // Choose best medoid i
14         if ΔTD_i < ΔTD* then (ΔTD*, m*, x*) ← (ΔTD_i, m_i, x_j) ;   // Remember the best swap
15     break loop if ΔTD* ≥ 0;
16     swap roles of medoid m* and non-medoid x* ;
17     TD ← TD + ΔTD* ;
18 return TD, M, C;
```

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 [11], but instead condense them directly into the following equation:

$$\Delta(x_o, m_i, x_j) = \begin{cases} \min\{d(x_o, x_j), d_s(o)\} - d_n(o) \\ \qquad \text{if } i = \textit{nearest}(o) \\ \min\{d(x_o, x_j) - d_n(o), 0\} \\ \qquad \text{otherwise} \end{cases} \tag{3.4}$$

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 would increase the cost of SWAP by a factor of $O(k)$, but we can cache these values, and only update them when performing a swap.

Reynolds et al. [18] note that we can decompose ΔTD into: (i) the cost of removing medoid m_i, and assigning all of its cluster members to the next best alternative, (ii) the (negative) cost of adding the replacement medoid x_j, and reassigning all objects closest to this new medoid. Since (i) does not depend on our 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 cost of doing so. We then iterate over all non-medoids that can fill the now-empty slot, and compute the benefit of doing so. 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, and so do we in our experiments.

Our approach is based on a similar idea of exploiting redundancy in these 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 Equation 3.4, 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 medoids.

3.1 Making PAM SWAP faster: PAM+

Algorithm 3 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 need an array of ΔTD 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 Equation 3.4. The interesting property is now since 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.

3.2 Benefits and Costs If we assume that the new medoid is closest in $O(1/k)$ cases on average (this assumes a somewhat balanced cluster size distribution), then we can compute the change for all k medoids with $O(k \cdot 1/k) = O(1)$ effort, by saving the innermost loop in line 12. Therefore, we expect a typical speedup on the order of $O(k)$ compared to the original PAM SWAP (but it may be impossible to guar-

Algorithm 4: PAM++: SWAP with multiple candidates

```
1  repeat
2      foreach x_o do compute nearest(o), d_nearest(o), d_second(o) ;
3      ΔTD*, x* ← [0,...,0], [⊥,...,⊥] ;                    // Empty best candidates array
4      foreach x_j ∉ {m_1,...,m_k} do
5          d_j ← d_nearest(x_j) ;                            // Distance to current medoid
6          ΔTD ← (−d_j, −d_j, ..., −d_j) ;                   // Cost change for making j a medoid
7          foreach x_o ≠ x_j do
8              d_oj ← d(x_o, x_j) ;                          // Distance to new medoid
9              (n, d_n, ds) ← (nearest(o), d_nearest(o), d_second(o)) ;   // Cached
10             ΔTD_n ← ΔTD_n + min{d_oj, d_s} − d_n ;        // Cost change for current
11             if d_oj < d_n then                            // Reassignment check
12                 foreach m_i ∈ {m_1,...,m_k} \ m_n do
13                     ΔTD_i ← ΔTD_i + d_oj − d_n;           // Update cost change
14         foreach i where ΔTD_i < ΔTD*_i do
15             (ΔTD*_i, x*_i) ← (ΔTD_i, x_j) ;               // Remember the best swap for each i
16     break loop if min ΔTD* ≥ 0 ;                          // At least one improvement was found
17     while i ← arg min ΔTD* and ΔTD*_i ≥ 0 do               // Execute all improvements
18         swap roles of medoid m_i and non-medoid x*_i ;
19         TD ← TD + ΔTD*_i ;
20         ΔTD*_i ← 0 ;                                      // Prevent further processing
21         foreach j where ΔTD*_j < 0 do                     // Recompute TD for remaining improvements
22             ΔTD ← 0 ;
23             foreach x_o ∉ {m_1,...,m_k} \ m_j do ΔTD ← ΔTD + Δ(x_o, m_j, x*_j) ;
24             if ΔTD ≤ τ · ΔTD*_j then ΔTD*_j ← ΔTD ;       // Tolerance check
25             else ΔTD*_j ← 0;                              // Skip otherwise
26 return TD, M, C;
```

antee this for any useful assumption on the data distribution; the worst case supposedly remains unaffected) at the slight cost of temporarily storing one ΔTD for each medoid m_i (compared to the cost of storing the distance matrix and the distances to the nearest and second nearest medoids, the cost of this is negligible).

3.3 Swapping Multiple Medoids: PAM++

A second technique to make this second stage of PAM faster is based on the following observation: PAM will always identify the single best swap to perform, then restart search; whereas the classic k-means reassigns all points, and updates all means in each iteration. Choosing the best swap has the benefit that this makes the algorithm independent of the data order [11] (as long as there are no ties), and it also means we need to execute much fewer swaps than if we would greedily perform any swap that yields an improvement (where we may end up repeated 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. Naively assuming that each medoid would be swapped in each iteration, this would allow us to reduce the number of iterations by k.

Based on this observation, we propose to consider the best swap for each medoid, and not only the single best swap, i.e., perform up to k swaps. This is a fairly simple modification shown in Algorithm 4, as we can simply store an array of swap candidates $(\Delta TD^*_i, x^*_i)$ in line 3, storing one 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 cannot find any, the algorithm has converged). We perform the best of these swaps in line 18, mark it as done. Then we have to recompute in line 23 for each remaining swap candidate if it still improves the clustering, otherwise the swap is not performed. At this point, in line 24, we explore two alternatives: (a) "strict" ($\tau = 1$): only swaps are executed that appear to be independent of the previous swaps (because their ΔTD has not changed) and (b) "greedy" ($\tau = 0$): execute any swap that still yields a benefit.

The benefits of this strategy are, unsurprisingly, much smaller than the first improvement. In early iterations we see multiple swaps being executed, but in the later iterations it is common that only few medoids change at all. Nevertheless, this simple modification does yield another measurable performance improvement. However—in contrast to the first improvement—this no longer guarantees to yield the exact same result (because the additional swaps may occasionally

be worse than the swaps found when rescanning, or may be executed in a slightly different order). Interestingly, in our experiments, even the "greedy" strategy would often find slightly better results, and faster.

3.4 Wider Exploration in CLARANS CLARANS [15] 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 cost, and then greedily performs this swap. Reusing the idea from PAM+, we can pick only the non-medoid object at random, but consider all medoids for swapping at a similar cost. 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 original CLARANS in the number of edges considered; but as the edges chosen involve the same non-medoids, we expect a slight loss in quality that should be easily countered by increasing the subsampling rate of non-medoids.

3.5 Faster Initialization With these optimizations to SWAP, that reduce the run time from $O(k(n-k)^2)$ to $O((n-k)^2)$, the main bottleneck of PAM suddenly becomes the first algorithm, BUILD. In the experiments below on the plant species data at $k = 200$, using the R implementation, PAM would spend 99% of the run time in SWAP, with all our optimizations this reduces to about 15%. About 16% is the time to compute the distance matrix, and 69% of the time is spend in BUILD. The run time of BUILD is in $O(kn^2)$, so for large k this is not unexpected to happen. But since we were able to make SWAP so much faster, we can easily afford to begin with slightly worse starting conditions. A very elegant way of choosing starting conditions in k-means is known as k-means++ [1]. The beautiful idea of this approach is to choose additional seeds with the probability proportional to their 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 the next seed there; afterwards the probability mass of this cluster reduces. At the same time, the algorithm is randomized; and we can run it multiple times and keep the best result. This helps if there is some local optimium that we might get caught in. We suggest to try this initialization approach also with PAM++ – it can be trivially adapted to sample proportional to the dissimilarity instead of the squared Euclidean distance.

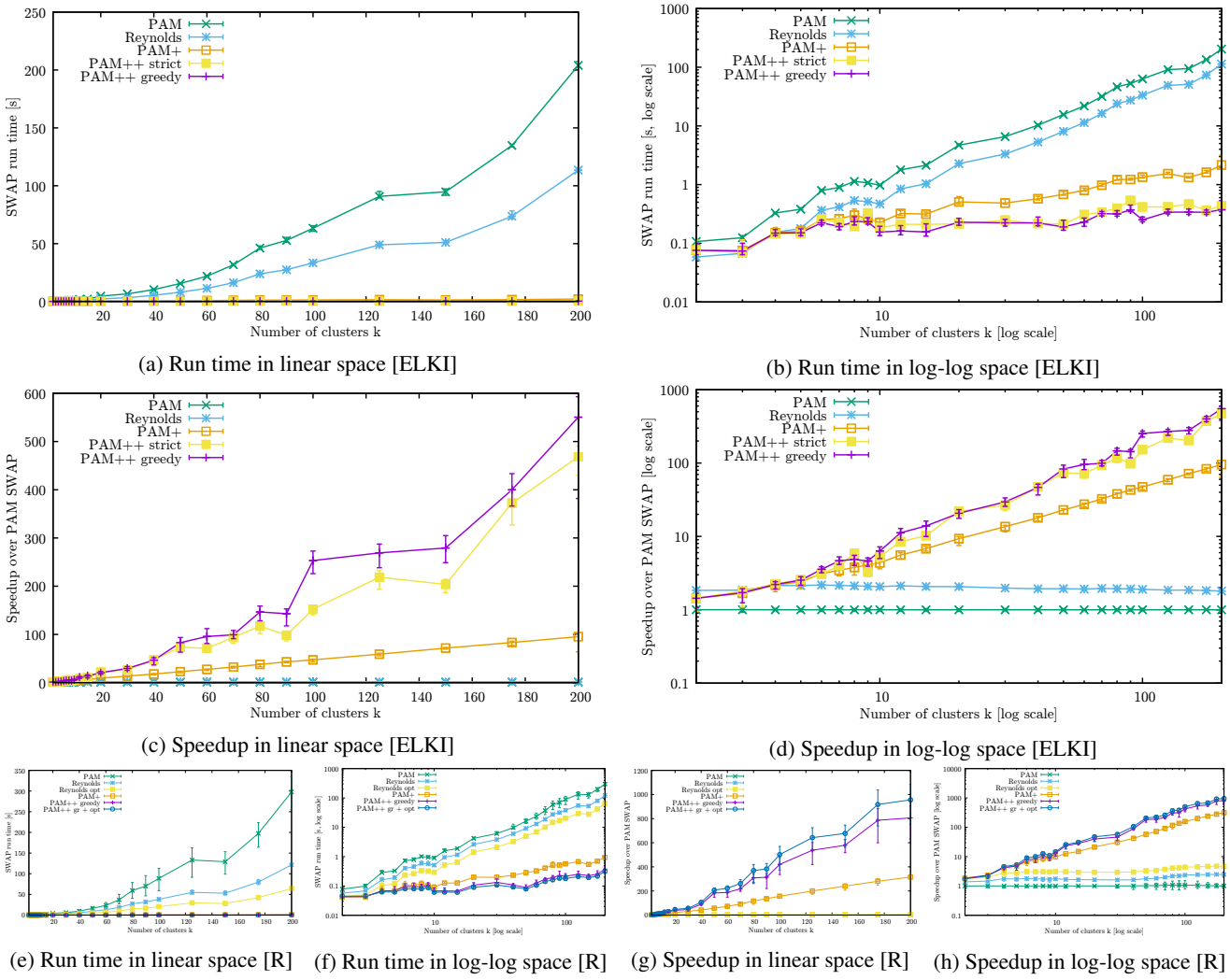
4 Experiments

For our first experiment, we use the "One-hundred plant species leaves" data set (texture features only) from the well-known UCI repository [4]. We chose this data set because it has 100 classes, and 1600 instances, a size that PAM can still reasonably 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 [21] in Java to develop our version. For comparison, we also ported it 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 report the average, minimum and maximum. Both implementations show similar behavior, so we are confident that the results are not just due to implementation differences [13].

4.1 Run Time Speedup In Figure 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, Reynolds version, as well as 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 linear with the number of clusters k. In log-log-space, Figure 1b we can differentiate the three variants studied. While the "greedy" variant is slightly faster than the "strict" variant of PAM++, the difference between these two is not very large compared to the main contribution of this paper.

In Figure 1c we plot the speedup over PAM. Reynolds SWAP clearly was about twice as fast as the original PAM. The first improvement suggested gave a 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 Figure 1c gives the false impression that this second contribution gave the most benefit. The logspace plot in Figure 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 $1.4\times$ respectively $1.75\times$, and less than our implementation of Reynolds (in R, as seen in Figure 1h, the difference at $k = 2, 3$ is negligible; so this is probably only an implementation difference). In Figure 1e to Figure 1h we provide the results using the R implementation, which clearly exhibit similar behavior. We only implemented the "greedy" $\tau = 0$ version in R; but we additionally include versions of Reynolds (`pamonce=2` of the existing package) and our approach that optimizes traversal of the distance matrix. In the most extreme case tested, a speedup of about $1000\times$ at $k = 200$ is measured – but since the speedup is expected to be $O(k)$, the exact values are meaningless.

In Figure 2, we study the run time of approximations to PAM (including the initialization algorithm run time here), including the well known CLARA and CLARANS algo-



(a) Run time in linear space [ELKI]

(b) Run time in log-log space [ELKI]

(c) Speedup in linear space [ELKI]

(d) Speedup in log-log space [ELKI]

(e) Run time in linear space [R]

(f) Run time in log-log space [R]

(g) Speedup in linear space [R]

(h) Speedup in log-log space [R]

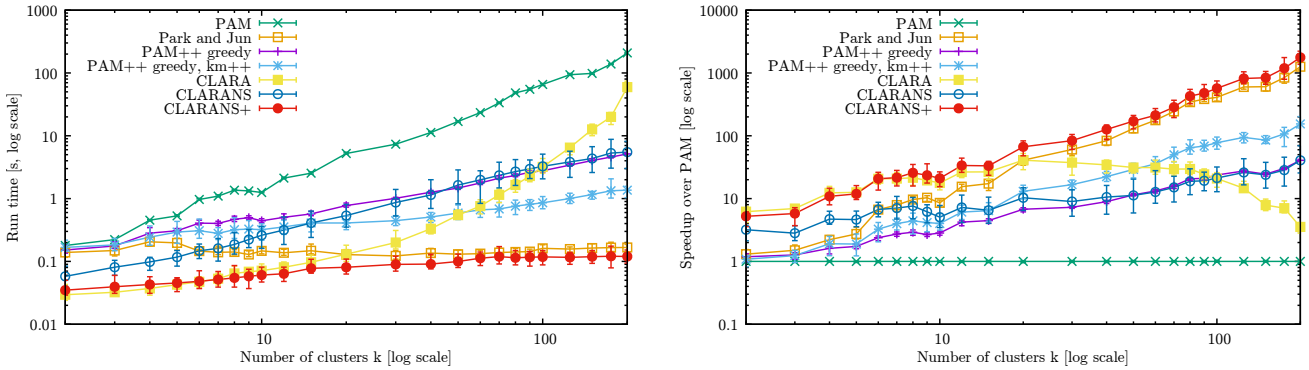
Figure 1: Run time comparison of PAM SWAP (only, without DAISY, without BUILD)

rithms. 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 must necessarily be increased with larger k. Park and Jun's [17] approach performs quite well, but its quality is rather low, as we will see below. CLARANS+ is the CLARANS approach enhanced with our optimization from Section 3.4. Clearly, this is the most scalable approach. The PAM++ version fares unexpectedly well, for large k comparable to the original CLARANS (but giving much higher quality, as we will see next). With k-means++ initialization (briefly mentioned in Section 3.5), because of the reduced initialization cost, it becomes even faster.

4.2 Quality Any algorithmic change and optimization comes at the risk of breaking some things, or negatively affecting numerics (see, e.g., [19] 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 except for the PAM++ algorithm, all are supposed to give the same result (and do so in the experiments). The PAM++ algorithm is more greedy in performing swaps (and not strictly doing a steepest descent), and may therefore converge to a different solution, but that should be of comparable quality.

In Figure 3 we visualize the cost, i.e., the objective TD of Equation 1.2, 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 [17] is over 25% worse, for the reasons discussed before (worse initialization, misses many improvements). If we used PAM BUILD for



(a) Run time in log-log space [ELKI]

(b) Speedup in log-log space [ELKI]

Figure 2: Run time comparison of different variations and derived algorithms

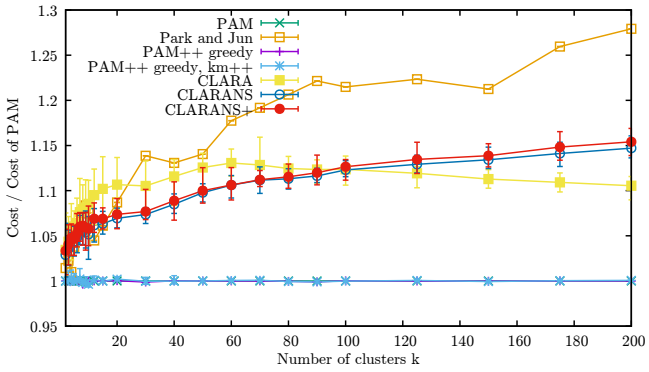
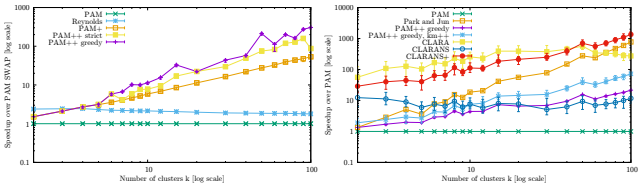


Figure 3: Cost (TD) compared to PAM



(a) SWAP Speedup [log-log]

(b) Overall speedup [log-log]

Figure 4: Results on Optical Digits data using ELKI

Park, the quality becomes only about 1% worse than PAM, but the runtime also increases substantially. Our strategy PAM++ even with the "greedy" approach gives results comparable to PAM as expected; the cheaper k-means++ initialization does not cause a noticeable loss in quality. CLARA (which only uses a subsample of the data) finds considerably worse results, but because the subsample size increases with k, it eventually approximates PAM as the subsample is much of the data set. CLARANS results become worse with 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). The quality of our CLARANS+ is slightly worse than CLARANS, because we let it subsample only k times fewer non-medoids, since it will explore all k medoids of each. Given the good run time of this approach, we encourage users to increase the sampling rate to 5% or more for large k.

4.3 Second Dataset We also verified our results on the "Optical Recognition of Handwritten Digits" data set from UCI [4] with $n = 5620$ instances, $d = 64$ variables, and 10 natural classes. The results are shown in Fig.4 and also show an $O(k)$ speedup compared to PAM. Clearly, the benefits on this data set are similar, and support our theoretical analysis.

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 [13]. 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 surprisingly large speedups obtained with this approach enable the use of this classic clustering method on much larger data, in particular with large k. Even such seemingly minor changes in such an algorithm can make a big difference. It is hard to devise such things on the drawing board – such solutions more naturally arise when trying to low-level optimize the code, such as when and when not to allocate memory for buffers, and trying to avoid recomputing the same values repeatedly. Today's compilers are reasonably good at performing local optimization, but will not introduce an additional array to cache such values.

The proposed method will be published as open-source contribution to the ELKI [21] framework, and a patch will be submitted to the R `cluster` package, to make it easy for others to benefit from these improvements. With these improvements, available 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] D. Arthur and S. Vassilvitskii. "k-means++: the advantages of careful seeding". In: *ACM-SIAM SODA*. 2007, pp. 1027–1035.

[2] H. Bock. "Clustering Methods: A History of k-Means Algorithms". In: *Selected Contributions in Data Analysis and Classification*. Ed. by P. Brito, G. Cucumel, P. Bertrand, and F. Carvalho. 2007, pp. 161–172.

[3] P. S. Bradley, O. L. Mangasarian, and W. N. Street. "Clustering via Concave Minimization". In: *NIPS*. 1996, pp. 368–374.

[4] D. Dheeru and E. Karra Taniskidou. *UCI Machine Learning Repository*. 2017. URL: `http://archive.ics.uci.edu/ml`.

[5] M. Ester, H. Kriegel, J. Sander, and X. Xu. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *KDD*. 1996, pp. 226–231.

[6] V. Estivill-Castro. "Why so many clustering algorithms: a position paper". In: *SIGKDD Explorations* 4.1 (2002), pp. 65–75.

[7] H. Fritz, P. Filzmoser, and C. Croux. "A comparison of algorithms for the multivariate L_1-median". In: *Computational Statistics* 27.3 (Sept. 2012), pp. 393–410.

[8] J. A. Hartigan and M. A. Wong. "Algorithm AS 136: A K-Means Clustering Algorithm". In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28.1 (1979), pp. 100–108.

[9] L. Kaufman and P. J. Rousseeuw. "Clustering by means of medoids". In: *Statistical Data Analysis Based on the L_1 Norm and Related Methods*. Ed. by Y. Dodge. North-Holland, 1987, pp. 405–416. ISBN: 0444702733.

[10] L. Kaufman and P. J. Rousseeuw. "Clustering Large Applications (Program CLARA)". In: *Finding Groups in Data*. John Wiley&Sons, 1990. Chap. 3, pp. 126–163. ISBN: 9780471878766.

[11] L. Kaufman and P. J. Rousseeuw. "Partitioning Around Medoids (Program PAM)". In: *Finding Groups in Data*. John Wiley&Sons, 1990. Chap. 2, pp. 68–125. ISBN: 9780471878766.

[12] L. Kaufman and P. J. Rousseeuw. "Clustering Large Data Sets". In: *Pattern Recognition in Practice*. Elsevier, 1986, pp. 425–437.

[13] H. Kriegel, E. Schubert, and A. Zimek. "The (black) art of runtime evaluation: Are we comparing algorithms or implementations?" In: *Knowl. Inf. Syst.* 52.2 (2017), pp. 341–378.

[14] C. Lucasius, A. Dane, and G. Kateman. "On k-medoid clustering of large data sets with the aid of a genetic algorithm: background, feasiblity and comparison". In: *Analytica Chimica Acta* 282.3 (1993), pp. 647–669. ISSN: 0003-2670.

[15] R. T. Ng and J. Han. "CLARANS: A method for clustering objects for spatial data mining". In: *IEEE TKDE* 14.5 (2002), pp. 1003–1016.

[16] M. L. Overton. "A quadratically convergent method for minimizing a sum of euclidean norms". In: *Math. Program.* 27.1 (1983), pp. 34–63.

[17] H. Park and C. Jun. "A simple and fast algorithm for K-medoids clustering". In: *Expert Syst. Appl.* 36.2 (2009), pp. 3336–3341.

[18] A. P. Reynolds, G. Richards, B. de la Iglesia, and V. J. Rayward-Smith. "Clustering Rules: A Comparison of Partitioning and Hierarchical Clustering Algorithms". In: *J. Math. Model. Algorithms* 5.4 (2006), pp. 475–504.

[19] E. Schubert and M. Gertz. "Numerically stable parallel computation of (co-)variance". In: *SSDBM*. 2018, 10:1–10:12.

[20] E. Schubert, S. Hess, and K. Morik. "The Relationship of DBSCAN to Matrix Factorization and Spectral Clustering". In: *LWDA*. Vol. 2191. CEUR Workshop Proceedings. 2018, pp. 330–334.

[21] E. Schubert, A. Koos, T. Emrich, A. Züfle, K. A. Schmid, and A. Zimek. "A Framework for Clustering Uncertain Data". In: *PVLDB* 8.12 (2015), pp. 1976–1979.

[22] E. Schubert, J. Sander, M. Ester, H. Kriegel, and X. Xu. "DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN". In: *ACM Trans. Database Syst.* 42.3 (2017), 19:1–19:21.

[23] C. Wei, Y. Lee, and C. Hsu. "Empirical comparison of fast partitioning-based clustering algorithms for large data sets". In: *Expert Syst. Appl.* 24.4 (2003), pp. 351–363.