A fast k-means implementation using coresets *

Gereon Frahling
Heinz Nixdorf Institute and
Department of Computer Science
University of Paderborn
D-33102 Paderborn, Germany
E-mail: frahling@uni-paderborn.de

Christian Sohler
Heinz Nixdorf Institute and
Department of Computer Science
University of Paderborn
D-33102 Paderborn, Germany
E-mail: csohler@uni-paderborn.de

December 5, 2005

Abstract

In this paper we develop an efficient implementation for a k-means clustering algorithm. The novel feature of our algorithm is that it uses *coresets* to speed up the algorithm. A coreset is a small weighted set of points that approximates the original point set with respect to the considered problem. The main strength of the algorithm is that it can quickly determine clusterings of the same point set for many values of k. This is necessary in many applications, since, typically, one does not know a good value for k in advance. Once we have clusterings for many different values of k we can determine a good choice of k using a quality measure of clusterings that is independent of k, for example the *average silhouette coefficient*. The average silhouette coefficient can be approximated using coresets.

To evaluate the performance of our algorithm we compare it with algorithm KMHybrid [28] on typical 3D data sets for an image compression application and on artificially created instances. Our data sets consist of $300,000$ to 4.9 million points. We show that our algorithm significantly outperforms KMHybrid on most of these input instances. Additionally, the quality of the solutions computed by our algorithm deviates less than that of KMHybrid.

We also computed clusterings and approximate average silhouette coefficient for $k = 1, \ldots, 100$ for our input instances and discuss the performance of our algorithm in detail.

1 Introduction

Clustering is the computational task to partition a given input into subsets of equal characteristics. These subsets are usually called clusters and ideally consist of similar objects that are dissimilar to objects in other clusters. This way one can use clusters as a coarse representation of the data. We loose the accuracy of the original data set but we achieve simplification.

Clustering has many applications in different areas of computer sciences such as computational biology, machine learning, data mining and pattern recognition. Since the quality of a partition is rather problem dependent, there is no general clustering algorithm. Consequently, over the years many different clustering algorithms have been developed. These algorithms can be characterized as *hierarchical algorithms* or *partitioning algorithms*.

Hierarchical algorithms build a hierarchy of clusters, i.e. every clusters is subdivided into child clusters, which form a partition of their parent cluster. Depending how the hierarchy is built we distinguish between *agglomerative* (bottom-up) and *divisible* (top-down) clustering algorithms.

Partitioning algorithms try to compute a clustering directly. For example, they try to compute a clustering by iteratively swapping objects or groups of objects between the clusters or they try to identify dense areas containing many points.

The most prominent and widely used clustering algorithm is *Lloyd's algorithm* sometimes also referred to as *the* k-*means algorithm*. This algorithm requires the input set to be a set of points in the d-dimensional Euclidean space. Its goal is to find k cluster centers and a partitioning of the points such that the sum of squared distances to the nearest center is minimized. The algorithm is a heuristic that converges to a local optimum. The main benefit of Lloyd's algorithm is its simplicity and its foundation on analysis of variances. Also, it is relatively efficient. The drawbacks are that the user must specify the number of clusters in advance, the algorithm has difficulties to deal with outliers and clusters that differ significantly in size, density, and shape.

Our Contribution. In this paper we develop an efficient k-means clustering algorithm called CoreMeans. The main

*Research is partially supported by DFG grant Me 872-8-2 and by the EU within the 6th Framework Programme under contract 001907 "Dynamically Evolving, Large Scale Information Systems" (DELIS).

new idea is that the algorithm uses coresets [11] to speed up the computation of the clustering. Such a coreset is a small weighted set of points that for every set of k centers has roughly the same cost as the original (unweighted) point set [5, 15, 11, 14]. Our algorithm first computes a small coreset of the input points and then runs a combination of Lloyd's algorithm and random swaps, which is somewhat similar to the algorithm Hybrid presented in [28, 20]. Then the algorithm doubles the size of the coreset and runs for a few steps on this coreset. This process is done until the coreset coincides with the whole point set. The coreset computation is supported by a quadtree (or higher dimensional equivalent) based data structure. This data structure can also be used to speed up nearest neighbor queries.

We compare our algorithm with algorithm KMHybrid [28, 20], which uses a combination of Lloyd's algorithm with random swaps and simulated annealing. On most of the input instances our algorithm significantly outperforms KMHybrid, especially for low dimensional instances. For high dimensional instances our algorithm finds good solutions faster but KMHybrid's solution after a few seconds is slightly better. If we want to compute a clustering for one value of k the running time of both algorithms is often dominated by the setup time to compute auxiliary data structures. In this case CoreMeans benefits from its smaller setup time.

However, in many applications we do not know the right value of k in advance. In such a case one has to compute clusterings for many different values of k. Then one can use a quality measure independent of k to find out the best clustering. A prominent quality measure for such a scenario is the *average silhouette coefficient*. Unfortunately, computing the *average silhouette coefficient* for one clustering takes quadratic time, which is not feasible for point sets of medium and large size. However, we would like to compute the sihouette coefficient for many values of k. In this situation we can see the real strength of coresets. Using coresets it is possible to find clusterings and compute their average silhouette coefficient for large point sets and many values of k. For example, we computed clusterings for $k = 1, \ldots, 100$ and approximated their average silhouette coefficient for a set of more than 4.9 million points in 3D consisting of the RGB values of an image in a few seconds on one core of an Intel Pentium D dual core processor with 3 GHz core frequency. In higher dimensions we did the same computations for an (artificially created) point set of $300,000$ points in 20 dimensions for all values of k between 1 and 100 in less than 8 minutes. Without coresets, the computation of the sihouette coefficient for one value of k takes several hours.

1.1 Related Work

It is beyond the scope of this section to give a comprehensive overview of the clustering literature. Instead we will give a brief overview of the most important developments with focus on partitioning algorithms. We will also briefly summarize the recent work on coresets. For a more comprehensive overview of the work in clustering we refer to the surveys/books [12, 18, 6]. An overview of coreset constructions is given in [2].

k-Means Clustering. The most popular algorithm for the k-means clustering problem is *Lloyd's algorithm* [10, 23, 25]. It is known that this algorithm converges against a local optimum [32]. Recently, a number of very efficient implementations of this algorithm have been developed [3, 20, 19, 29, 30, 31]. These algorithms reduce the time needed to compute the nearest neighbors in a Lloyd's iteration, which is the most time consuming step of the algorithm. In the Euclidean space there are many $(1+\epsilon)$-approximation algorithms for the k-means clustering problem [5, 9, 15, 17, 22, 24]. Also for the k-means problem in metric spaces efficient constant factor approximation algorithms are known [26, 20].

The quality of random sampling in metric spaces has been analyzed for some clustering problems including the metric and the Euclidean k-median [27, 8]. The analysis can be easily extended to the k-means clustering problem.

A testbed for k-means clustering algorithms has been given in [28]. We compare our implementation with algorithm KMHybrid, which is the fastest of 4 implementations described in [28].

Coresets. A coreset is a small set of points that approximates the original set with respect to some problem. In the context of clustering algorithms several coreset constructions have been developed for the k-median and k-means clustering problem [5, 15, 11, 14]. These coresets found applications in approximation algorithms [5, 14], data streaming [15, 11], and clustering of moving data [13]. Also for projective clustering, coresets have been developed [16]. Apart from clustering, coresets have found applications in basic problems in computational geometry, for example, to compute an approximation of the smallest enclosing ball of a point set [4] or to approximate extent measure of point sets [1, 7].

2 Preliminaries

We will use $\|p, q\|$ to denote the Euclidean distance between p and q and we generalize this definition to sets, i.e. $\|p, Q\| = \min_{q \in Q} \|p, q\|$ for any set Q of points in the $\mathbb{R}^d$. We use $\mathcal{C}(Q)$ to denote the centroid of a set Q, i.e. $\mathcal{C}(Q) = \frac{1}{|Q|} \sum_{q \in Q} q$.

We will deal with weighted and unweighted sets of points. We will always assume that the set P represents the input instance, which is an unweighted set of n points in the $\mathbb{R}^d$.

A weighted point set will usually be denoted by R. The weights of the points in R are given by $w(r)$ for every $r \in R$. We will only consider integer weights in this paper.

In the k-means clustering problem we want to find a set $C = \{c_1, \ldots, c_k\}$ of k cluster centers and a partition the set P into k clusters $C_1, \ldots, C_k$ such that

$$\sum_{i=1}^{k} \sum_{p \in C_i} \|p, c_i\|^2$$

is minimized. The goal of the weighted k-means problem is to find a set $C = \{c_1, \ldots, c_k\}$ of k cluster centers and a partition of a weighted set R into k clusters $C_1, \ldots, C_k$ such that

$$\sum_{i=1}^{k} \sum_{p \in C_i} w(p) \cdot \|p, c_i\|^2$$

is minimized. Hence in this case the weight is simply the multiplicity of a point.

Once the set C is known an optimal partition can be computed by assigning every point to its nearest cluster center breaking ties arbitrarily, i.e. $C_i = \{p \in P : \|p, c_i\| = \|p, C\|\}$ under the assumption that all distances are distinct. We therefore can write $\mathbf{means}(P, C)$ to denote the cost of an optimal partition of P with respect to the centers in C. In a similar way we write $\mathbf{means}(R, C)$ to denote the cost of an optimal partition of a weighted set R with respect to C. It is also known that the centroid $\mathcal{C}(Q)$ of a set Q is the point $c \in \mathbb{R}^d$ that minimizes $\sum_{p \in Q} \|p, c\|^2$.

2.1 The Basic k-Means Method

Based on the observations that it is easy to compute an optimal partition for a fixed set of centers and an optimal set of centers for a fixed partition, a simple and elegant clustering heuristic has been developed [10, 23, 25]. Nowadays, one often refers to this heuristic as *the* k*-means algorithm* or *Lloyd's algorithm*. This algorithm runs in iterations. At the beginning of an iteration the algorithm has a set of k centers $\{c_1, \ldots, c_k\}$. Every iteration consists of two steps.

1. For every $p \in P$ compute its nearest center in $\{c_1, \ldots, c_k\}$. Partition P into k sets $C_1, \ldots, C_k$ such that C_i contains all points whose nearest center is c_i.
2. For every cluster C_i compute its centroid $\mathcal{C}(C_i)$, i.e. the optimal center of that cluster. Then set $c_i = \mathcal{C}(C_i)$ for every $1 \leq i \leq k$.

Each iteration runs in $O(ndk)$ time. Typically, the algorithm runs for a fixed number of iterations (standard values are in the range from 50 to 500). It is well known that the algorithm only converges to a local optimum and that the quality of this solution depends strongly on the intial set of centers. Therefore, the algorithm is usually repeated several times with different sets of initial centers and the best discovered partition is returned.

2.2 Algorithm KMHybrid

In the experiments we compare our algorithm to an algorithm called KMHybrid [28, 20]. KMHybrid combines swapping of centers with Lloyd's algorithm and a variant of simulated annealing. The algorithm does one swap followed by a sequence of Lloyd's steps and an acceptance test. If the current solution passes the test, the algorithm continues with the current solution. Otherwise, it restores the old one. The acceptance test is based on a variant of simulated annealing. Additionally, the algorithm uses a kD-tree to speed up nearest neighbor search in the Lloyd's steps. For more details see [28, 20].

2.3 Coresets and their Construction

A coreset for an unweighted set of points P is a much smaller weighted set of points that approximates P with respect to the k-means problem.

Definition 2.1 *A weighted set of points* R *is an* ϵ-coreset *for* P *with respect to the* k*-means problem, if for every set* C *of* k *centers we have* $(1-\epsilon)\mathbf{means}(P, C) \leq \mathbf{means}(R, C) \leq (1+\epsilon)\mathbf{means}(P, C)$.

Constructing a coreset. In our algorithm we will use a coreset construction from [11], which has been used to derive a streaming algorithm (a small space dynamic data structure) for several problems including k-median and k-means clustering. We will briefly review the theoretical construction here. Later in Section 3 we will discuss our implementation.

For the theoretical part we assume that the point set has integer coordinates from $\{1, \ldots, \Delta\}$. For simplicity we will focus on the 2D-case. We assume that we know the cost *Opt* of an optimal set of centers. In our algorithm we replace this assumption by guessing the value of *Opt*. To construct the coreset we subdivide the input space using $\log \Delta$ nested square grids $\mathcal{G}^{(1)}, \ldots \mathcal{G}^{(\log \Delta)}$. The cells in grid $\mathcal{G}^{(i)}$ have side length 2^i. To compute the coreset we will first identify the *heavy cells* of each grid. A cell is called heavy, if it contains more points than specified by a certain threshold. This threshold depends quadratically on the inverse of the side length of the grid cell. Under the assumption that d is a constant we can parametrize the threshold by some value $\delta = \Theta(\epsilon^d/(k \log n))$ in the following way.

Definition 2.2 *[11] We call a cell of grid* $\mathcal{G}^{(i)}$ heavy*, if it contains at least* $\delta \cdot \frac{Opt}{4^i}$ *points of* P.

The process of constructing the coreset is somewhat similar to computing a quadtree decomposition of the (heavy) cells of the input space. We start with the coarsest grid $\mathcal{G}^{(\log \Delta)}$ and identify all heavy cells. Then we subdivide every heavy cell $\mathcal{H}$ into 4 subcells. We call $\mathcal{H}$ the parent cell

of its subcells. We mark $\mathcal{H}$, if none of its subcells is heavy. Otherwise, we do not mark $\mathcal{H}$ and recurse this process with all heavy subcells. By the definition of heavy cells it is clear that this recursion must eventually stop, since at some point a heavy cell is required to have more than n points (by our assumption of integer coordinates, we have $Opt > n$ for sufficiently large n). At the end of this recursion we choose one point from every marked cell and call it a representative point of that cell. The union of all representative points is the coreset. It remains to compute the weights of the coreset points. The weight of a coreset point is given by the number of points assigned to it in the following assignment. Every point in a marked heavy cell $\mathcal{H}$ is assigned to the representative point of $\mathcal{H}$. For every point p that is not contained in a heavy cell let $\mathcal{H}_1$ denote the smallest heavy cell that contains p. Since $\mathcal{H}_1$ is not marked it must contain at least one marked heavy cell $\mathcal{H}_2$ (which may be much smaller than $\mathcal{H}_1$). Then p is assigned to the representative point of $\mathcal{H}_2$. If $\mathcal{H}_1$ contains more than one marked heavy cell, p may be assigned to the representative point of any one of them.

Theorem 1 *[11] The coreset construction described above computes an ϵ-coreset of size $O(k \log n/\epsilon^{d+2})$.* □

2.4 The Silhouette Coefficient

Usually, the right number of clusters is not known in advance. Since the k-means objective function drops monotonically as k increases, one needs a different measure for the quality of a clustering that is independent of k. Such a measure is provided by the average *silhouette coefficient* [21] of the clustering. The silhouette coefficient of a point p_i is computed as follows. First compute the average distance of p_i to the points in the same cluster as p_i. Then for each cluster C that does not contain p_i compute the average distance from p_i to all points in C. Let b_i denote the minimum average distance to these clusters. Then the silhouette coefficient of p_i is defined as $(b_i - a_i)/\max(a_i, b_i)$.

The value of the silhouette coefficient of a point varies between -1 and 1. A value near -1 indicates that the point is clustered badly. A value near 1 indicates that the point is well-clustered. To evaluate the quality of a clustering one can compute the average silhouette coefficient of all points.

3 The Algorithm

We first provide a high level description of our algorithm and then we give some more details on the implementation. The algorithm starts to compute a coreset of size roughly $2k$ and chooses k points from this coreset as a starting solution. Then it repeats $\max\{40/\sqrt{k}, 2\}$ times the following two steps, which form the main loop of the algorithm.

- First it runs Lloyd's algorithm for d steps. After this, the current solution is compared to the previously best and the algorithm continues with the better of these solutions.
- In the second step, the algorithm chooses a number k_0 between 0 and k uniformly at random. Then it picks centers from the current set of centers according to the following probability distribution until k_0 different centers are chosen. The probability that the center of cluster C_j is replaced is $\frac{1}{|C_j| \cdot \sum_{i=1}^{k} \frac{1}{|C_i|}}$, where $C_1, \ldots, C_k$ denotes the current clustering. Finally, these k_0 random centers are replaced by points chosen uniformly at random from the coreset.

```
CoreMeans(P, k)
  m ← 2k
  while m ≤ |P| do
    Compute coreset of size m.
    if m = 2k then
      C ← k random points from coreset
      K ← C
    repeat max{40/√k, 2} times      (* Main loop *)
      Do d iterations of Lloyd's algorithm
      starting with C
      Let C denote the current solution
      if means(P, K) < means(P, C) then
        C ← K
      K ← C
      Choose k_0 at random from {0, ..., k}
      Swap k_0 centers from C with points
      chosen uniformly from the coreset
    m ← 2 · m
```

After the main loop is finished the algorithm doubles the size of the coreset and continues with the main loop. This is done until the coreset is the whole point set. The algorithm is given in Figure 3.

To support efficient computation of coresets and to speed up nearest neighbor queries in Lloyd's algorithm we use a quadtree or its higher dimensional equivalent. Our approach is the analog to the kD-tree algorithm from [20]. The root of the quadtree corresponds to a bounding box of the point set. With each cell B associated with a node of the quadtree we store the number of points contained in the cell, the sum of the point vectors $\sum_{p \in B} p$, and the sum of the squared ℓ_2-norm of the point vectors $\sum_{p \in B} \|p\|_2^2$. This information can be used to quickly compute the *exact* cost of the partition of P that corresponds to a given partition of the coreset. Since all points from one cell are assigned to the same coreset point and hence to the same cluster, we can compute the cost of a cluster C by first computing the mean c of the cluster from the sum of the vectors devided by the number of points in the cluster. Then we have to compute $\sum_{p \in C} d(p, c)^2 = \sum_{p \in C} \|p - c\|_2^2 = \sum_{p \in C} \|p\|_2^2 - \|c\|_2^2$,

which can be done efficiently by using the information stored with each box.

In our implementation we fixed the depth of the quadtree to 11. To build the tree we proceeded bottom-up. We identify the non-empty cells in the grid corresponding to the 11-th level of the tree. The non-empty cells are stored in a hash table with cell coordinates as keys. After we have computed the non-empty cells in the finest grid we iterate over these cells and compute all non-empty cells in the next coarser grid together with the corresponding point statistics.

To compute a coreset of around m points we first have to identify a good guess for $\gamma := \delta \cdot Opt$. We do it by setting γ to a large value and dividing it iteratively by 1.3. After each iteration we compute the size of the coreset from the cell statistics (without computing the actual coreset points). For high values of γ this is done very fast since the coreset size can be computed using a few large cells. Alltogether the time to compute the coreset *size* is negligible compared to the coreset computation time.

The coreset for a given value of γ is computed using a recursive depth first search function on the quadtree cells. For the root cell we call a function COMPUTECORESETPOINTS. This function has a cell as input parameter as well as statistics about points to be moved into that cell. COMPUTECORESETPOINTS adds the input points to the cell statistics. Then for each subcell it checks heaviness. If there is at least one heavy subcell, it calls COMPUTECORESETPOINTS for all heavy subcells. The points given as function parameter and the points in all light subcells are then moved into a heavy subcell by adding up their statistics and giving these statistics to one call of COMPUTECORESETPOINTS as function parameters. If a cell has no heavy subcell, a coreset point is introduced from the statistics about cell points and the statistics given as function input.

To speed up of the later k-means algorithm we store the following statistics: For each cell occupied by coreset points we store a pointer to a corresponding coreset point (if there is one) and pointers to all subcells containing coreset points. During the construction of coreset R we additionally compute for each cell B the sum of the coreset vectors $\sum_{r\in R\cap B} w(r)\cdot r$, the number $\sum_{r\in R\cap B} w(r)$ of points assigned to the coreset points in the cell and the sum of squared ℓ_2-norm of the coreset vectors $\sum_{r\in R\cap B} w(r)\cdot \|r\|_2^2$.

One iteration of the k-means algorithm is done as follows: Instead of searching the nearest center for each coreset point separately we use an approach analogous to the kd-tree approach as in [20]. We start with a list $\mathcal{L}$ of all centers as possible candidates for nearest centers and do a depth first walk on those quadtree cells which contain a coreset point. For each cell B in the quadtree we check if we can rule out that some centers c in $\mathcal{L}$ are the nearest center to any point in B. This is done by first computing the point p from $\mathcal{L}$ that is nearest to the center of B. Then we check for each point $q \in \mathcal{L}$, whether B lies completely on the same side as p of the bisector between p and q. All centers which cannot be nearest centers for coreset points in B are evicted from $\mathcal{L}$ and the algorithm proceeds to the children of cell B. If $|\mathcal{L}| = 1$ we know the nearest center for all coreset points within the cell. Since we hold statistics for all coreset points within each cell we can then assign all coreset points in one step to the center $c \in \mathcal{L}$ and stop. If $|R\cap B| = 1$, we compute the distances of the coreset point to all $c \in \mathcal{L}$ directly, assign the coreset point and stop the depth first walk.

```
SILHOUETTE(K)
  m ← 5
  while m ≤K
    Compute coreset of size m.
    for each k ∈ {1,...,100} do
      Use main loop of CoreMeans
      to compute clustering
      Compute average silhouette coefficients
      for the current coreset and centers
    m ← 2 · m
```

Computation of silhouette coefficients. The computation of silhouette coefficients for each point p_i is speeded up in the following way: We first compute the average distance a_i to all points in the same cluster. To compute b_i, the minimum over average distances $b_{i,j}$ to points in other clusters C_j, we identify the second nearest cluster center c_l and compute the average distance $b_{i,l}$ to all points in C_l. In most cases $b_{i,l}$ is the minimum of all $b_{i,j}$ for other clusters. To get a certificate for this, we use the lower bound $d(p_i, c_j) \le b_{i,j}$. We check for all other clusters if $d(p_i, c_j) \ge b_{i,l}$. If this inequality holds then $b_{i,l} \le b_{i,j}$ and $b_{i,j}$ cannot be the minimal one. In that case we save the computations of all distances to points in cluster C_j.

4 Experiments

We implemented our algorithm using C++. The code was compiled using gcc version 3.4.4 using optimization level 2 (-O2). We compare our algorithm to KMHybrid [20]. KMHybrid was compiled using the same compiler and also with optimization level 2. We ran it using the standard settings given by the developers.

We ran our experiments on an Intel Pentium D dual core processor with two cores. The algorithm used one core with core frequency 3 GHz. The computer has 2 GByte RAM.

4.1 Data Sets

We performed our experiments on two different types of instances. The first type of instance consists of images and we want to cluster the RGB values of the pixels. Thus the input

points lie in 3D and the i-th input point corresponds to the RGB-values of the i-th pixel of the image. Such a clustering has applications in lossy data compression, since one can reduce the palette of colors used in the picture to the colors corresponding to the cluster centers.

Our test images consist of three large images (Tower, Bridge, and PaSCo) and three medium size images (Monarch, Frymire, and Clegg). The latter images are commonly used to evaluate the performance of image compression algorithms. The exact sizes of the test images can be found in Table 4.1. The images are available at http://homepages.uni-paderborn.de/frahling/coremeans.html

4.1.1 Artificially created instances.

The second type of instance is artificially created. Instance ArtificialxD consists of $300,000$ points in x dimensions. The instance is generated by taking a random point from one of 20 Gaussian distributed clusters, whose center are picked uniformly at random from the unit cube. The standard deviation of the Gaussian distribution is $0.02 \cdot \sqrt{d}$, i.e. it is the product of the one dimensional Gaussian distribution with standard deviation 0.02. An example of a sample of points from instance Artificial2D is given in Figure 5.

4.2 Comparison of CoreMeans and KMHybrid

To evaluate the performance of CoreMeans we compare our algorithm to KMHybrid. We first compare the setup times for both algorithms, i.e. the time to construct the auxiliary data structures. If one wants to compute a clustering for fixed value of k then the setup times often dominate the running time of the algorithm. If a good value of k is not known, then one often wants to compute a clustering for multiple values of k. In this case, it is more interesting to compare the running time of both algorithms without setup time (however, the time to extract the coresets from the data structure is contained in the given running times). This is done in Sections 4.2.2 to 4.2.4. In Section 4.2.2 we compare both algorithm for different input sizes. In Section 4.2.3 we focus on the performance with increasing dimension, and in Section 4.2.4 we investigate into the dependence on the number of clusters.

Instance	Size	Setup Times KMHybrid	Setup Times CoreMeans
Tower	4,915,200	28.59	4.77
Bridge	3,145,728	18.13	2.95
PaSCO	3,145,728	19.41	4.29
Frymire	1,234,390	4.71	0.65
Clegg	716320	2.76	1.05
Monarch	393,216	1.43	0.63
Artificial5D	300,000	2.27	1.49
Artificial10D	300,000	3.71	2.17
Artificial15D	300,000	4.71	2.70
Artificial20D	300,000	6.09	3.87

Table 1: Data sets and setup time.

4.2.1 Setup time

The times to compute the auxiliary data structure are given in Table 4.1. The time to build these structures does not depend on k. The setup time for KMHybrid is between 1.5 to 7 times higher than that of CoreMeans. There is a tendency that the gap becomes larger for larger instances. However, there seems to be also an dependence on the distribution of points as the largest factor was achieved for the medium size instance Frymire.

If one computes a clustering for one value of k then the setup time is typically larger than the computation time. Even for the larger instances both algorithms obtain a good clustering in a few seconds (see also Section 4.2.2).

4.2.2 Dependence on input size

To evaluate the dependence on the input size we run both algorithms on instance Monarch, Clegg, Frymire, PaSCo, Bridge, and Tower. We used paramter $k = 50$. In general, CoreMeans performs better for smaller k (see Section 4.2.4) and tends to perform similar to KMHybrid as k increases. The results are shown in Figures 7 to 12. The plots give the average performance of 10 runs. The vertical bars indicate the best and worst solution found within these runs. The relative performance of CoreMeans increases slightly with the size of n. We would like to emphasize that the difference between the best and worst solution found during the 10 runs is much smaller for CoreMeans. Therefore, to guarantee a good solution we have to run KMHybrid more often than CoreMeans. Another interesting observation is that CoreMeans achieves slightly better approximations for larger instances.

4.2.3 Dependence on the dimension

Next we are interested in the dependence on the dimension. To evaluate this dependence, we compare the average performance of 10 runs of KMHybrid and CoreMeans for $k = 20$ on the instances ArtificialxD for $x = 5, 10, 15$, and 20. The graphs are shown in Figure 1 and 2. CoreMeans performs better on all instances. The most significant difference in performance can be found in the 5D instance, where CoreMeans performs a factor $10 - 30$ better. The higher the dimension the smaller is the advantage of CoreMeans. In these experiments the deviation of KMHybrid

was much bigger than that of CoreMeans. Although CoreMeans shows the much better average performance, the best solution found by KMHybrid was better than the best solution found by CoreMeans. Overall, the performance of the algorithm for medium dimensions was much better than theory predicts with an exponential dependence on the dimension.

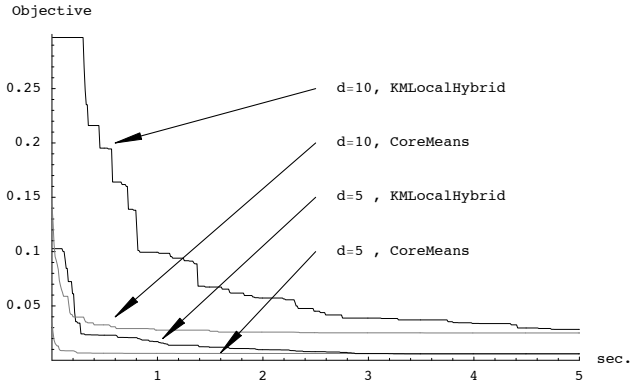

Figure 1: Performance for ArtificialxD with $x = 5$ and $x = 10$ excluding setup time.

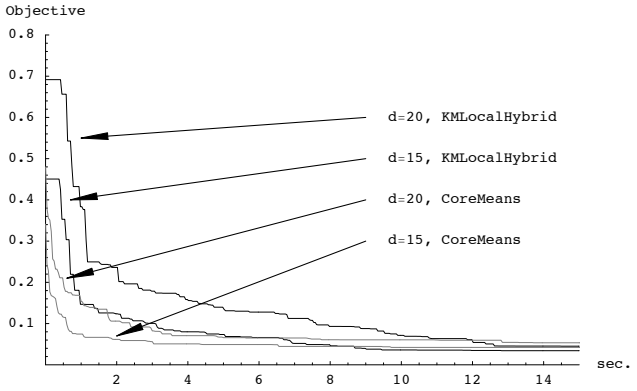

Figure 2: Performance for ArtificialxD with $x = 15$ and $x = 20$ excluding setup time.

4.2.4 Dependence on the number of clusters

To investigate in the dependence on the number of cluster centers, we ran a number of experiments on different inputs. Due to space limitations we only present results for $k = 10, 50, 100$, and 200 for instance Bridge. These results are typical for the performance we encountered. As before, the Figures 3 and 4 show the average performance of 10 runs excluding the setup times. Typically, our algorithm performs significantly better for small values of k. For example, for $k = 10$ CoreMeans often performs a factor $10 - 100$ better. Additionally, the quality of the solutions computed by KMHybrid varies significantly. CoreMeans is less sensitive to the random choices of the algorithm. As a consequence one must perform more runs of KMHybrid to obtain a good solution with high probabilty. As k grows larger the performance gap between the two algorithms decreases. The reason for this is that the quality of the coreset decreases as k grows.

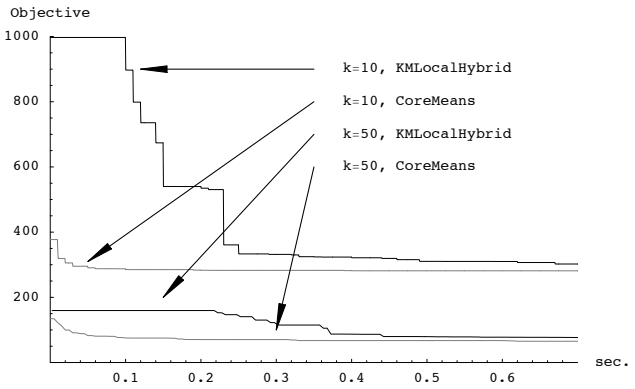

Figure 3: Performance for Bridge with $k = 10$ and $k = 50$ excluding setup time.

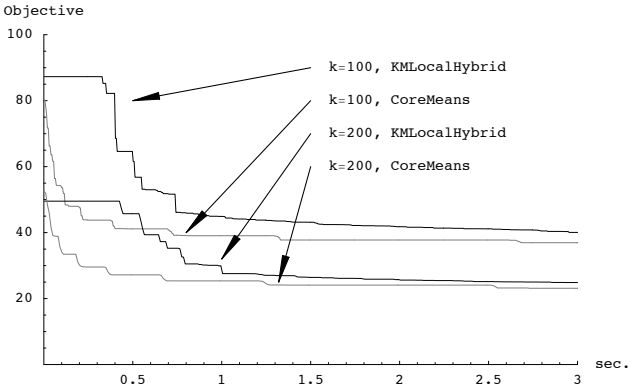

Figure 4: Performance for Bridge with $k = 100$ and $k = 200$ excluding setup time.

4.3 Computing the silhouette coefficient

We computed the approximate average silhouette coefficient for $1 \leq k \leq 100$ for instances Tower, Clegg, Monarch, and ArtificialxD with $x = 2, 10, 20$ using coresets of different sizes. Table 4.3 summarizes the running times of our tests. The second column gives the overall running time for the computation and the third column states the time spend to compute the silhouette coefficients. Since the time to compute the silhouette coefficient is quadratic in the coreset size,

Instance	Coreset	Time	Silhouette
Tower	404	7.99	0.84
	1607	19.24	6.43
Clegg	423	4.69	0.8
	1720	15.07	6.58
Monarch	428	4.80	0.77
	1626	15.37	6.11
Artificial2D	427	2.52	0.62
	1616	7.73	4.3
	6431	51.89	45.57
Artificial10D	400	43.34	1.88
	1711	123.38	17.68
Artificial20D	408	139.58	4.18
	1778	442.5	40.62

Table 2: Time to compute clusterings and approximate average silhouette coefficients. The second column contains the overall running time (including setup). The third column gives the time required to compute the approximate average silhouette coefficient.

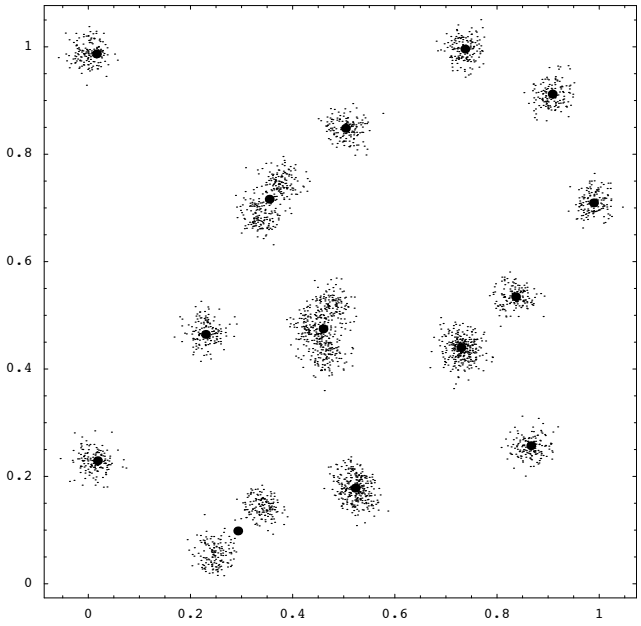

Figure 5: A sample of points from instance Artificial2D. The bold points are the centers that achieve the best average silhouette coefficient.

the fraction of time spent for this computation increases significantly with increasing coreset size.

To show the effectivity of our method we focus on instance Artificial2D. A sample of points from this instance is shown in figure 5. The average silhouette coefficent for this instance and coreset sizes 427, 1616, and 6431 is given in Figure 6. We see that even the smallest coreset suffices to approximate the coefficient quite well. The only problem is that the silhouette coefficient increases slightly with k. A reason for this may be that the number of centers is already relatively large compared to the number of coreset points. If some centers contain only one point, then they have silhouette coefficient exactly 1 and this may lead to slightly increasing coefficient, if k is large compared to the coreset size. For our applications a coreset of size roughly 1600 will definitly suffice. There is almost no difference to one with more than 6000 points.

The highest silhouette coefficient value was achieved for 14 clusters (using the larger coreset) by the cluster centers shown in figure 5. The reason why only 14 clusters were found (although we had 20 cluster centers) can be explained by the fact that some of the clusters were very close to eachother and so the clustering coefficient is higher when one assign only one center to these clusters.

4.4 Summary

Summarizing we can say that CoreMeans performs very well especially for small values of k. When we compare the computation time of CoreMeans with KMHybrid we see that for one clustering the running time of both algorithms is typically dominated by the setup time. However, KMHybrid is much more volatile and might require a number of runs to guarantee a good solution. The main strength of CoreMeans is that it can very quickly compute relatively good solutions and that one can approximate the average silhouette coefficient from the coreset.

5 Conclusions

In this paper we presented an efficient implementation of a k-means clustering algorithm using coresets. Our algorithm performs very well compared to KMHybrid [28] for small dimension and small to medium k. The quality of the solutions varies less than that of KMHybrid, which implies that we need fewer runs to guarantee a good solution. The main strength of our algorithm is to quickly find relatively good approximations for many values of k, for example when a good value for k is not known in advance. In this case, we can also use the coresets to compute the average clustering coefficient and thus to find a good choice of k.

References

[1] P. Agarwal, S. Har-Peled, and R. Varadarajan. Approximating Extent Measures of Point. *Journal of the ACM*, 51(4): 606–635, 2004.

[2] P. Agarwal, S. Har-Peled, and K. Varadarajan. Geometric Approximation via Coresets. Survey available at http://valis.cs.uiuc.edu/ sariel/research/papers/04/survey/survey.pdf

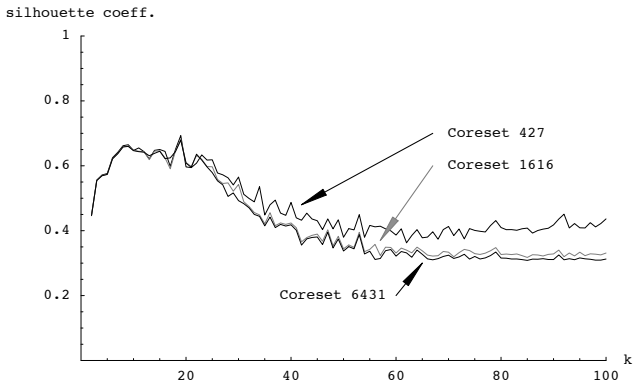

Figure 6: The average silhouette coefficient of Artificial2D.

[3] K. Alsabti, S. Ranka, and V. Singh. An Efficient k-Means Clustering Algorithm. *Proceedings of the first Workshop on High Performance Data Mining*, 1998.

[4] M. Badoiu and K. Clarkson. Smaller Core-Sets for Balls. *Proceedings of the 14th Symposium on Discrete Algorithms* (SODA'03), pp. 801–802, 2003.

[5] M. Badoiu, S. Har-Peled, and P. Indyk. Approximate Clustering via Coresets. *Proceedings of the 34th Annual ACM Symposium on Theory of Computing* (STOC'02), pp. 250–257, 2002.

[6] P. Berkhin. Survey of Clustering Data Mining Techniques. Available at ..., 2002.

[7] T.Chan. Faster Core-Set Constructions and Data-Stream Algorithms in Fixed Dimensions. *Proceedings of the 20th Annual ACM Symposium on Computational Geometry* (SoCG'04), pp. 246–258, 2004.

[8] A.Czumaj and C. Sohler. Sublinear-Time Approximation for Clustering via Random Sampling. *Proceedings of the 31st Annual International Colloquium on Autonate, Languages and Programming* (ICALP'04), pp. 396–407, 2004.

[9] W. Fernandez de la Vega, M. Karpinski, C. Kenyon, and Y. Rabani. Approximation schemes for clustering problems. *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 50–58, 2003.

[10] E. Forgey. Cluster Analysis of Multivariate Data: Efficiency vs. Interpretability of Classification. *Biometrics*, 21:768, 1965.

[11] G. Frahling and C. Sohler. Coresets in Dynamic Geometric Data Streams. *Proceedings of the 37th Annual ACM Symposium on Theory of Computing* (STOC'05), pp. 209–217, 2005.

[12] J. Hartigan. Clustering Algorithms. *John Wiley & Sons*, New York, 1975.

[13] S. Har-Peled. Clustering motion. *Discrete & Computational Geometry*, 31 (4): 545–565, 2004.

[14] S. Har-Peled and A. Kushal. Smaller Coresets for k-Median and k-Means Clustering. *Proceedings 21st Annual ACM Symposium on Computational Geometry* (SoCG'05), pp. 126–134, 2005.

[15] S. Har-Peled and S. Mazumdar. Coresets for k-Means and k-Median Clustering and their Applications. *Proceedings of the 36th Annual ACM Symposium on Theory of Computing* (STOC'04), pp.291–300, 2004.

[16] S. Har-Peled and K. Varadarajan. Projective Clustering in High Dimensions using Coresets. *Proceedings 18th Annual ACM Symposium on Computational Geometry* (SoCG'02), pp. 312–318, 2002.

[17] M. Inaba, N. Katoh, and H. Imai. Applications of weighted Voronoi diagrams and randomization to variance-based k-clustering. *Proceedings of the 10th Annual ACM Symposium on Computational Geometry*, pp. 332–339, 1994.

[18] A. Jain and R. Dubes. Algorithms for Clustering Data. *Prentice Hall*, New Jersey, 1988.

[19] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu. An Efficient k-Means Clustering Algorithm: Analysis and Implementation. IEEE Trans. Pattern Anal. Mach. Intell. 24(7): 881-892, 2002.

[20] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu. A Local Search Approximation Algorithm for k-Means Clustering. *Proceedings of the 18th Annual Symposium on Computational Geometry* (SoCG'02), pp. 10–18, 2002.

[21] D. Knuth. The Art of Computer Programming: Sorting and Searching, Vol. 3, Addison-Wesley, 1973.

[22] A. Kumar, Y. Sabharwal, and S. Sen. A simple linear time $(1+\varepsilon)$-approximation algorithm for k-means clustering in any dimensions. *Proceedings of the 45th IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 454–462, 2004.

[23] S. Lloyd. Least Squares Quantization in PCM. *IEEE Transactions on Information Theory*, 28: 129–137, 1982.

[24] J. Matoušek. On approximate geometric k-clustering. *Discrete & Computational Geometry*, 24(1): 61–84, 2000.

[25] J. MacQueen. Some Methods for Classification and Analysis of Multivariate Observations. *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pp. 281–296, 1967.

[26] R. Mettu and G. Plaxton. Optimal Time Bounds for Approximate Clustering. *Machine Learning*, 56(1-3):35–60, 2004.

[27] N. Mishra, D. Oblinger, and L. Pitt. Sublinear time approximate clustering. *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 439–447, 2001.

[28] D. Mount. KMlocal: A Testbed for k-means Clustering Algorithms. Available at http://www.cs.umd.edu/ mount/Projects/KMeans/kmlocal-doc.pdf

[29] D. Pelleg and A. Moore. Accelerating Exact k-Means Algorithms with Geometric Reasoning. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 277–281, 1999.

[30] D. Pelleg and A. Moore. x-Means: Extending k-means with efficient estimation of the number of clusters. *Proceedings of the 17th International Conference on Machine Learning*, 2000.

[31] S. Phillips. Acceleration of k-Means and Related Clustering Problems. *Proceedings of Algorithms Engineering and Experiments* (ALENEX'02), 2002.

[32] S. Selim and M. Ismail. k-Means-Type Algorithms: A Generalized Convergence Theorem and Characterizations of Local Optimality. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6::81–87, 1984.

[33] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: A new data clustering algorithm and its applications. *Journal of Data Mining and Knowledge Discovery*,1(2),pp. 141-182, 1997.

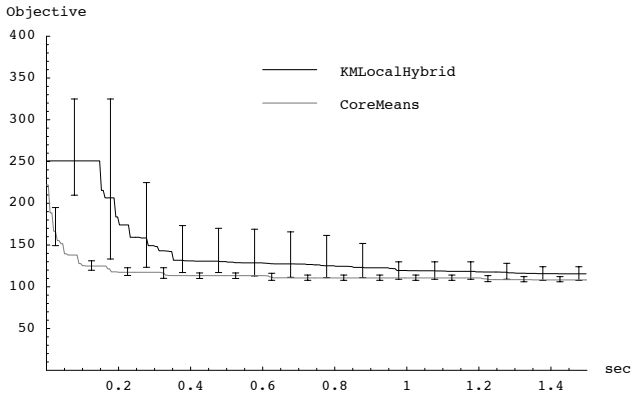

Figure 7: Data set Monarch; k = 50; excluding setup time.

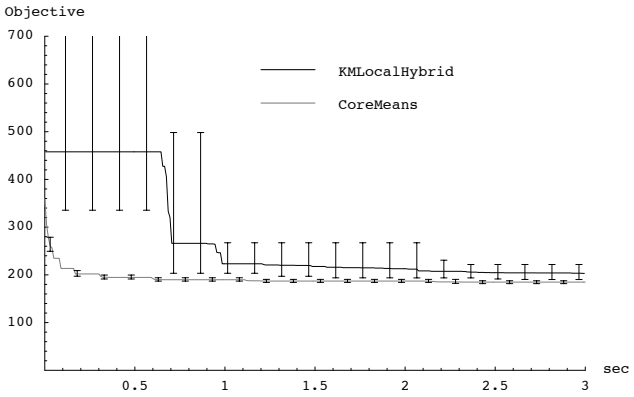

Figure 10: Data set PaSCo; k = 50; excluding setup time.

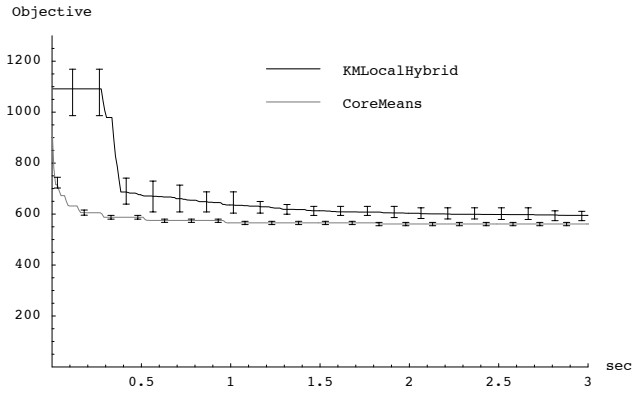

Figure 8: Data set Clegg; k = 50; excluding setup time.

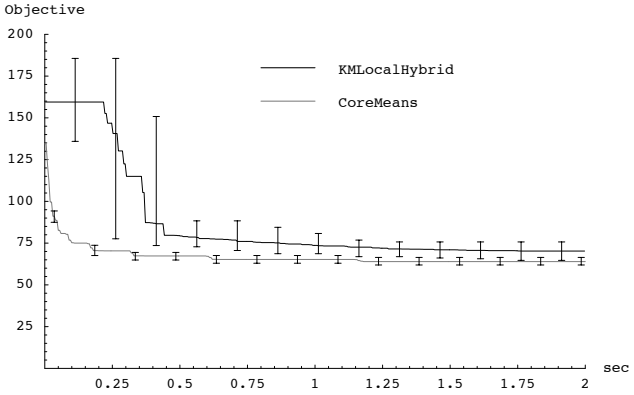

Figure 11: Data set Bridge; k = 50; excluding setup time.

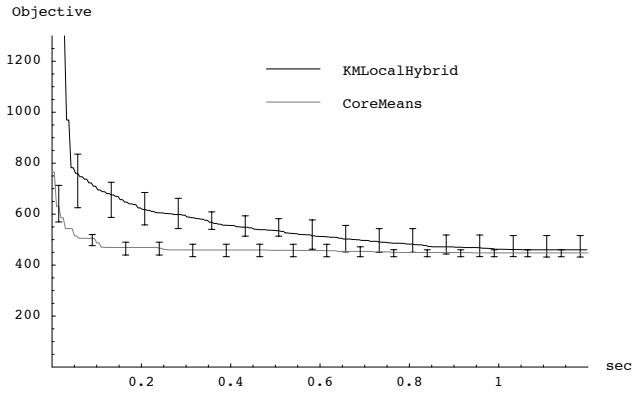

Figure 9: Data set Frymire; k = 50; excluding setup time.

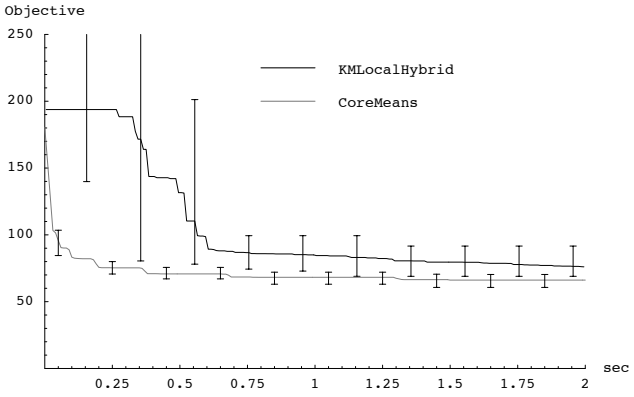

Figure 12: Data set Tower; k = 50; excluding setup time.