Solving k-means on High-dimensional Big Data

Jan-Philipp W. Kappmeier[1], Daniel R. Schmidt[2] and Melanie Schmidt[2]

[1] *Technische Universität Berlin, Germany,* kappmeier@math.tu-berlin.de
[2] *Carnegie Mellon University, Pittsburgh PA,* {schmidtd,mschmid1}@andrew.cmu.edu

June 1, 2015

In recent years, there have been major efforts to develop data stream algorithms that process inputs in one pass over the data with little memory requirement. For the k-means problem, this has led to the development of several $(1 + \varepsilon)$-approximations (under the assumption that k is a constant), but also to the design of algorithms that are extremely fast in practice and compute solutions of high accuracy. However, when not only the length of the stream is high but also the dimensionality of the input points, then current methods reach their limits.

We propose two algorithms, piecy and piecy-mr that are based on the recently developed data stream algorithm BICO that can process high dimensional data in one pass and output a solution of high quality. While piecy is suited for high dimensional data with a medium number of points, piecy-mr is meant for high dimensional data that comes in a very long stream. We provide an extensive experimental study to evaluate piecy and piecy-mr that shows the strength of the new algorithms.

1 Introduction

Partitioning points into subsets (*clusters*) with similar properties is an intuitive, old and central question. *Unsupervised* clustering aims at finding structure in data without the aid of class labels or an experts opinion. It has many applications ranging from computer science applications like image segmentation or information retrieval to applications in other sciences like biology or physics where it is used on genome data and CERN experiments. For an overview on the broad subject, see for example the survey by Jain [13]. The *k-means problem* asks to cluster data such that the sum of the squared error is minimized. It has been studied since the fifties [17, 23] and optimizing it is likely 'the most commonly used partitional clustering strategy' [14]. It measures the quality of a partitioning of points from $\mathbb{R}^d$ based on the squared Euclidean distance function. Each cluster in the partitioning is represented by a center, and the objective function is the sum of the squared distances of all points to their respective center.

The popularity of the k-means problem is underlined by the fact that the most popular algorithm for it, Lloyd's algorithm, was named one of the ten most influential algorithms in the data mining community by the organizers of the IEEE International Conference on Data Mining (ICDM) in 2008, see Wu et. al. [25]. Lloyd's algorithm [18] (independently developed by Steinhaus [23]) is a local search heuristic that iterates the following two steps. First, it obtains an initial solution consisting of k centers, e. g., by drawing k centers uniformly at random from the input. Then, the following two steps are alternated: Assign every point to its closest center to obtain a partitioning into k subsets, compute the centroid of each subset and replace the center by this centroid. Both steps can only decrease the cost. Assigning points to their closest center is optimal for the given centers, and for each subset, the centroid is the optimal center. Thus, the new solution is either cheaper or of equal cost. In the latter case, the algorithm has converged[1].

The quality in terms of the sum of squared errors of the output of Lloyd's algorithm depends on the local optimum that is reached. Finding a good local optimum can be achieved by initializing the algorithm with a good initial solution. Arthur and Vassilvitskii [3] propose the *k-means++* method as an improved version of Lloyd's algorithm. It chooses the initial solution randomly, but only the first center is chosen uniformly at random. The ith center is chosen by computing all points squared distances to their closest center and then chosing each point with a probability proportional to its cost as the next center. This way, it is likely that most optimal centers have a close center in the start solution. This initialization method produces centers which are an $\mathcal{O}(\log k)$-approximation in expectation, and experiments indicate that the local optimum found from this start solution is usually of high quality.

The k-means++ method therefore provides a great tool for solving the k-means problem in practice, with an (expected) worst-case guarantee, a very good practical performance and the advantage that it is very easy to implement. The theoretically best approximation algorithms for the k-means problem provide a constant factor approximation for the general case [15, 16] and a $(1+\varepsilon)$-approximation (even in linear time) if k and ε are assumed to be constants [8].

For big data, running Lloyd's algorithm or k-means++ is less viable. Asymptotically, the running time of both algorithms is $\mathcal{O}(ndk)$ if the number of iterations is bounded to a constant. This looks convincing since a straightforward implementation of finding the closest center for a point takes $\Theta(dk)$ time, so even evaluating a solution then has running time $\Theta(ndk)$. Additionally, the input size is already $\mathcal{O}(nd)$, so the running time is linear for constant values of k. However, both algorithms need random access to the data and iterate over it several times. As soon as the data does not fit into main memory, the algorithms do thus not scale very well. For example, k-means++ needed over seven hours to compute 50 centers for a 54-dimensional data set (*Covertype*) with half a million points [1].

A natural strategy to cope with this problem is to summarize the data before running the respective algorithm. A famous example for this is BIRCH [26], a SIGMOD Test of Time Award winning algorithm that computes a summary by one pass over the input data and then clusters the points in the summary. BIRCH is very fast and thus enables the processing of large data sets. However, the quality in terms of the sum of squared errors can be low [1, 10].

A more recent development is the design of fast data stream algorithms that are based on *coresets*. A coreset S of a point set P is a weighted summary of P that maintains a strong quality guarantee: For any choice C of k centers, the k-means costs of the clustering induced by C on S are within an $(1+\varepsilon)$-factor of the k-means clustering that C induces on P. Thus, executing any k-means algorithm on the coreset gives a good approximation of what the same algorithm would have produced on P. Coreset constructions are generally designed with a focus on strong theoretic bounds, but can be made viable in practice with slight heuristic changes.

StreamKM++ is such an algorithm [1]. It computes a coreset in one pass over the data and then runs k-means++ on the coreset. The size of the coreset is polylogarithmic in the input sizes if the dimension of the data is constant. The total memory requirement is also polylogarithmic. Experiments show that the quality of the solutions is comparable to the k-means++ solutions (on the full data set) while the running time is a small fraction. For example, the above mentioned covertype is processed in ten minutes instead of seven hours, with a result of similar quality.

BICO is a recent algorithm that outperforms StreamKM++ on all data sets that are tested in [1, 10] and enables the processing of data sets with millions of points in less than an hour[2]. The above mentioned test case needs 27 seconds instead of ten minutes for StreamKM++ and seven hours for k-means++, and larger instances show even higher acceleration. BICO is also based on a coreset construction, using a slight variation of an algorithm with a strong theoretical guarantee. The quality

[1] Since there are finitely many partitionings, the algorithm eventually converges to a local optimum. It is also common to stop the algorithm after a predefined number of iterations, or when the decrease of the cost function is small.

[2] One data set is *BigCross*, containing three million points in 68 dimensions and is processed in under twenty minutes for $k \leq 250$.

of the computed solutions in experiments is as good as that of StreamKM++. The source code of BICO is written in C++ and is available online.

For data sets with up to around 100 dimensions, this is a pleasant state of affair. However, both the analysis of the running time and memory requirement of StreamKM++ and BICO assume that the dimension is a constant. At least for BICO, this is not a theoretically imposed restriction, but does indeed correspond to an unfavorable dependency on the dimension. The reason is that BICO covers the input data by spheres (in order to summarize all points in the same sphere by one point). When the number of spheres is too large, a rebuilding step reduces it by merging certain spheres. Covering a set by spheres gets increasingly difficult as the dimension gets higher, which results in several rebuilding steps of BICO, and in a higher running time.

On the theoretical level, however, there are several results saying that it is possible to compute a coreset of a point set in one pass and with low memory requirements. For example, Feldman and Langberg [8] propose a one-pass algorithm that computes a coreset with storage size of $\mathcal{O}\left(kd\log^4 n\varepsilon^{-3}\log 1/\varepsilon\right)$. It is thus theoretically possible to compute coresets which scale well with the dimension, but there is no practical algorithm yet that achieves a high quality summary and can cope with very high dimensional, large data sets.

1.1 Our Contribution

We develop two new algorithms, *piecy* and *piecy-mr* that can deal with high-dimensional big data. For that, we combine BICO with a dimensionality reduction. This reduction is done by projecting onto the best fit subspace (of a parameterized dimension) which can be computed by the *singular value decomposition* (SVD). This is theoretically supported by recent results [5, 9] that say that projecting onto the best fit subspace of dimension $\lceil k/\varepsilon\rceil$ and then solving the k-means problem gives a $(1+\varepsilon)$-approximation guarantee. We find that $3k/2$ dimensions are often sufficient to give highly accurate results. This might be due to the spectrum of the data we used.

The next challenge is to intertwine the dimensionality reduction with the coreset computation in order to do both in one pass over the data. The first algorithm, *piecy*, reads chunks (pieces) of the data and processes, reduces the dimensionality of each chunk and feeds the resulting points into BICO. The drawback of this approach is that the total dimensionality of the complete point set that is fed into BICO increases with the number of pieces. For large data sets and high input dimension, this approach will eventually run into the same trouble as BICO (but only for data sets that are larger and higher dimensional than those BICO can process). In *piecy-mr*, we resolve this potential limitation by adapting a technique called *Merge-and-Reduce* [12]. It is a method that shows that any coreset computation can be turned into a one-pass algorithm at the cost of additional polylogarithmic factors. We adapt it to take advantage of the fact that we use a coreset computation (BICO) which already *is* a one-pass algorithm.

As intermediate steps of our work, we evaluate two implementations for the singular value decomposition, an implementation in Lapack++ [24] and the implementation called redSVD [21]. We compare their speed and quality. Furthermore, we extend the algorithm BICO to process weighted inputs (which is necessary for our piecy-mr approach).

2 The algorithms

In the following, we describe the three algorithms that we tested: BICO and our two new algorithms, *piecy* and *piecy-mr*. For a point set P, we denote the centroid of P by $\mu(P) := \sum_{x\in P} x/|P|$.

2.1 BICO

BICO uses a data structure based on *clustering features*. A clustering feature of a point set S consists of the number of points $|S|$, the sum of the points $\sum_{x\in S} x$ and the sum of the squared length of the points $\sum_{x \in S} x^t x$. By the well-known formula

$$\sum_{x \in P} ||x - c||^2 = |P| \cdot ||\mu(P) - c||^2 + \sum_{x \in P} ||x - \mu(P)||^2,$$

which holds for every point set P, a clustering feature is enough to exactly compute the cost between a point set and *one* center. BICO uses spheres that cover the input data. The point set inside each sphere is represented by a clustering feature. When a point arrives, it can be added to a clustering feature in constant time. The challenge for BICO is to decide into which clustering feature a point shall be added in order to equally distribute the error and to keep the overall error small. This is achieved by managing the clustering features in a well organized tree. Finding an appropriate clustering feature to add a point dominates the insertion time of a point. It lies between $\Theta(1)$ and $\Theta(m)$ for each point, where m is the coreset size. BICO includes several heuristics to speed up the identification process of a good clustering feature such that the running time is often closer to $\Theta(1)$ per point. How well these heuristics work depends on the dimension of the input point set.

Whenever the number of spheres (and thus clustering features) exceeds m, BICO performs a rebuilding step that merges some of the spheres and their features together. For high-dimensional data sets, this may occur more often unless the spheres become large enough. More rebuilding steps imply a higher running time.

2.2 Piecy

Our aim is to compute coresets for large high-dimensional data sets by using BICO and dimensionality reduction techniques, but in *only one pass* over the data. *Piecy* pursues the idea of running only a single instantiation of BICO and subsequently feeding it with chunks of low dimensional points. Thus, piecy reads a piece of p points, reduces its intrinsic dimension and inputs the resulting points into BICO.

Choice of dimensionality reduction technique and number of dimensions. We use the projection to the best fit subspace of dimension ℓ, where ℓ is a parameter to be optimized. The best fit subspace can be computed by using the *singular value decomposition*. The theoretical background of this approach is that projecting to best fit subspaces yields a good approximation of the squared pairwise distances [5, 6]. When projecting to k dimensions, a 2-approximation is guaranteed, while projecting to $\lceil k/\varepsilon \rceil$ guarantees a $(1+\varepsilon)$-approximation. Thus, we test values between k and moderate multiples of k to get a reasonable compromise between approximation factor and running time.

Using SVD to project to the best fit subspace. When we say that we use 'the' SVD, we mean the SVD of the matrix $A \in \mathbb{R}^{n \times d}$ where the input points are stored in the columns. The SVD of A has the form $A = UDV^T$ for matrices $U \in \mathbb{R}^{n \times n}, D \in \mathbb{R}^{n \times d}, V \in \mathbb{R}^{d \times d}$, where U and V are unitary matrices and D is a diagonal matrix. The matrix V contains the right singular vectors of A. The projection of (the points stored in) A to the best fit subspace of dimension ℓ is the matrix $A_\ell = UD_\ell V^T$, where D_ℓ is obtained by replacing all but the first ℓ diagonal elements by zero. Notice that the resulting matrix still contains d-dimensional points, but their *intrinsic* dimension is reduced to ℓ. This still helps, since the ℓ-dimensional point set is easier to cover for BICO.

Computation of the SVD. Numerically stable computation of the singular value decomposition is a research field of its own. Basic methods that compute the *full* SVD, e.g. U, V and D, have a running time of $\Omega\left(nd \min\left(n, d\right)\right)$. This full SVD can be used by dropping the appropriate entries of D to obtain a matrix D_ℓ and evaluating the matrix product $UD_\ell V^t$ to obtain the projection onto the best fit subspace of dimension ℓ. However, a variety of more efficient algorithms have been developed for this specific task, which are known as algorithms for the *truncated* SVD that computes a decomposition $A_\ell = U_\ell D_\ell V_\ell^t$ directly without computing the full SVD of A. Additionally, random variations are known that reduce the running time sufficiently at the cost of a small error. Mahoney [20] gives a very nice overview on different methods to compute the singular value decomposition, then continuing with a detailed view on randomized methods and also discussing practical aspects. For this work, we use an implementation that is based on the randomized algorithm presented in [11] that multiplies A with a randomly drawn matrix to reduce the number of its columns before computing the SVD. The implementation is called *redSVD* [21]. In addition to reducing the number of columns, it also reduces the number of rows before computing the SVD. Below, we experimentally compare the performance of redSVD to the performance of the *lapack++* implementation of the full SVD computation.

Parameters. The authors of BICO propose using a coreset size of $200k$ for BICO, which we adopt. That given, there are two parameters to be chosen: The size of the pieces that are the input for one SVD, and the number of dimensions we project to. As we argued above, the latter should be at least k and not more than a reasonable multiple of k.

Memory requirement. At each point in time, we store at most one piece of the input, one SVD object and one BICO object. The memory requirement of BICO is proportional to the output size, i. e., to $200k$.

Obtaining a solution. Running *piecy* computes a summary of the input points. In order to obtain an actuall solution for the k-means problem, we run k-means++[3] on the summary.

2.3 Piecy-MR

Notice that each chunk of data that is processed by piecy adds (in the worst case) m dimensions to the intrinsic dimension of the point set that is stored by the BICO instance, as long as the maximum dimension is reached. For large data sets, this is unfavorable.

Helpful coreset properties. A convenient property of coresets helps here. Assume that S_1 and S_2 are coresets for points sets P_1 and P_2, i. e., their weighted cost approximates the weighted cost of P_1 or P_2, respectively, for any possible solution, and up to an ε-fraction. Then the weighted cost of their union $S_1 \cup S_2$ approximates the cost of $P_1 \cup P_2$ for any solution up to an ε-fraction as well. Furthermore, if we use a coreset construction to reduce $S_1 \cup S_2$ to a smaller set (since $|S_1 \cup S_2|$ will be larger than the size of one coreset), then we obtain a coreset for $P_1 \cup P_2$. The error gets larger but is bounded by a (3ε)-fraction of the cost of $P_1 \cup P_2$ (which can be compensated by choosing a smaller ε to begin with).

The Merge-and-Reduce technique. Assume for a moment that our aim is solely to compute a coreset with no thoughts about the intrinsic dimension of the points, but given a coreset computation that needs random access to the data. Then an intuitive approach is to read chunks of the data, computing a coreset for each chunk and joining it with previous corsets, until the union becomes too large. Then we could reduce the union by another coreset construction. The problem with this approach is that the first chunk of the data will participate in all following reduce steps, making the error unnecessary high. The Merge-and-Reduce technique [4] (for clustering for example used in [2, 12]) organizes the merge and reduce steps in a binary tree such that each point takes part in at most $\mathcal{O}(\log n)$ reduce steps for a stream of n points.

Our computation tree. We have a different problem since the coreset construction that we use, BICO, does not require random access to the data. Instead, we wish to keep the dimension of the input data small. Assume we would consider this problem independently from the coreset computation, by just computing the SVD of chunks of the data and keeping the reduced points in memory (maybe performing a second pass over the data to compute the coreset). This is infeasible since the number of points is not reduced and hence we would store the complete data set (with a lower intrinsic dimension). Imagine even that at each point in time, an oracle could provide us with the best fit subspace of dimension ℓ of all points seen so far. We could still not easily use this information since the best fit subspace would change over time. So if we use one instance of BICO, and input each

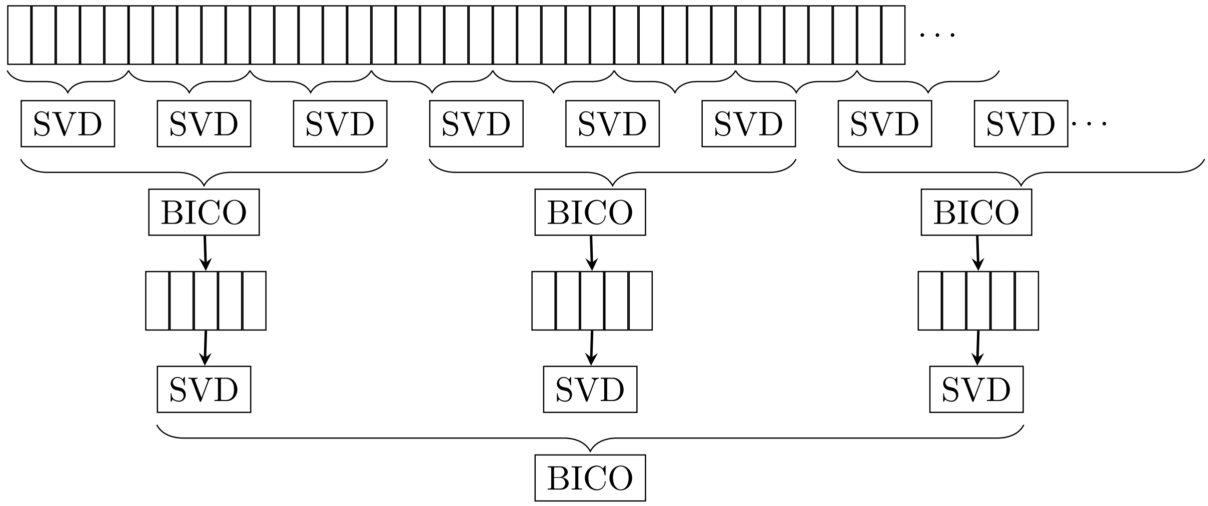

Figure 1: The Merge-and-Reduce style tree built by piecy-mr with an exemplary *piece size* of 5 and a *number of pieces* of 3. Every chunk with *piece size* many points is first fed into a singular value decomposition. The result of the SVD contains the same number of points but has a smaller intrinsic dimension. It is then fed into an instantiation of the BICO algorithm. After *number of pieces* many chunks, the BICO algorithm computes a coreset of size *piece size*. Thus, we can continue on the next layer. On each layer, the number of points is reduced by a factor of *number of pieces*. We continue to call the SVD on each layer to keep the intrinsic dimension of the point set small.

point into it, projected to the best fit subspace of all points seen so far, then we would still get a high intrinsic dimension for the points stored in BICO.

By also embedding BICO into the Merge-and-Reduce tree, we solve these problems. The first way of doing this would be to view the two steps of reducing the dimension and entering the points into BICO as one coreset computation, and just embed this into the Merge-and-Reduce technique. However, this has the drawback that we perform the same number of dimensionality reductions as we use BICO for reducing sets to smaller sets. We do, however, expect that the union of multiple dimensionality reduced sets will not immediately have a high intrinsic dimension. In particular if the data evolves over time, then multiple consecutive pieces of the input data will have approximately the same best fit subspace (but over time, the subspace will change). We add more flexibility to the algorithm by running more than one copy of BICO, while allowing that more than one SVD output is processed by the same BICO instance. The actual computation tree is visualized in Figure 1.

Parameters. The algorithm has three parameters, the dimension that the SVD reduces to, the *piece size* which is the number of points that are read as input for one SVD computation, and the *number of pieces*, which is the number of SVD outputs that are processed by one instance of BICO. When BICO reaches the limit, the computed coreset is given to a SVD instance and then entered into a BICO on a higher level. It is convenient to set the piece size to $200k$, which also means that BICO computes a summary of size $200k$, the summary size suggested in the original BICO publication.

Memory requirement. We store one BICO element for each level of the computation tree. The degree of the tree is equal to the number of pieces b, so we have $\log_b n$ levels. At each point in time, there is at most one SVD object in the memory since there is always at most one SVD computation at the same time. If the piece size is equal to $200k$, then the memory requirement of each BICO element is proportional.

2.3.1 Weighted BICO

In the original implementation, BICO processes unweighted input points. In the *piecy-mr* computation tree, the instances of BICO on higher levels of the computation tree have to process weighted inputs (since the coreset points are weighted). Thus, we extended the source code of BICO to work for weighted inputs. For an input point x with weight w, we have to simulate what BICO would do for w copies of x. The main observation is that in most routines of BICO, multiple copies of the same point can be treated as one. For example, finding the closest reference point that is currently in the data structure can be done once and the result is then valid for all copies of x. Additionally, if we decide to open a new clustering feature with x as the reference point, we can insert all (not yet inserted) copies into this clustering feature at no cost.

What we have to adjust is the insertion process into already existing clustering features, and the initial values for new clustering features. Setting the correct values for a new clustering feature is straightforward: The new clustering feature has reference point x, its sum of points is $w \cdot x$, the sum of squares is $w \cdot x^2$ and the number of points stored in the feature is w. When we add w copies of a point x to an existing clustering feature with centroid μ and s points in it, then the actual increase of the error due to this is

$$\begin{aligned} s \cdot \|\mu - \mu_n\|^2 + w\|x - \mu_n\|^2 &= s \cdot \left\|\mu - \frac{s\mu + wx}{s+w}\right\|^2 + w\left\|x - \frac{s\mu + wx}{s+w}\right\|^2 \\ &= \frac{sw^2}{(s+w)^2}\|x-\mu\|^2 + \frac{ws^2}{(s+w)^2}\|x-\mu\|^2 \\ &= \frac{sw}{s+w}\|x-\mu\|^2 \end{aligned}$$

where we denote the new centroid after adding w copies of x by μ_n. We conclude that the total error made in the feature after inserting w points is $c + \frac{sw}{s+w}\|x-\mu\|^2$, where c denotes the original error made in the feature.

The original BICO implementation would have inserted the w copies sequentially into the clustering feature until the features threshold error of T would have been surpassed. It actually uses $\|x-\mu\|^2$ to measure the additional error and thus overestimates it. When adding single points, the effect of this overestimation decreases with each added point such that this works well for BICO. In the weighted version, however, using $w \cdot \|x-\mu\|^2$ is can be off by a large margin.

Instead, we compute how many copies w' of x can be inserted into the feature without surpassing the threshold:

$$c + \frac{sw'}{s+w'}\|x-\mu\|^2 \le T \iff w'\big(s\|x-\mu\|^2 - T + c\big) \le sT - sc$$

If $s \cdot \|x-\mu\|^2 - T + c \le 0$, the threshold will not be reached for any $w' \ge 0$. We can thus insert all w copies. Otherwise, we insert

$$w' = \min\left(w, \frac{sT - sc}{s\|x-\mu\|^2 - T + c}\right)$$

many copies of x. If the threshold is reached before all w copies of x are inserted, i.e., if $w' < w$, we continue recursively as in the original BICO implementation.

2.3.2 Best fit subspace for weighted points

The singular value decomposition of a matrix is defined in an unweighted fashion, yet we want to use it for reducing the dimensionality of the weighted coreset points that result from BICO runs. What we want to do is project the points to the best fit subspace of the point set where each point is replaced by several copies of itself according to its weight. Translated into the matrix notation, this means that we want to compute the projection of A to the best fit subspace of dimension ℓ of a matrix F which contains multiple copies of the points from A according to their (integral) weight[3].

[3]The weights that are computed by BICO are always integral. In fact, they sum up to the number of points BICO has processed.

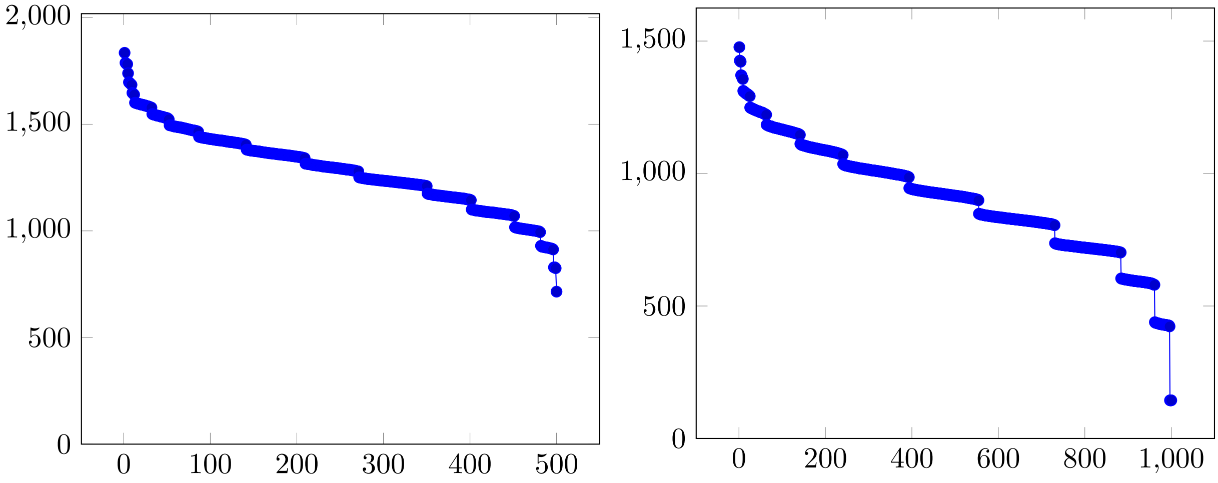

Figure 2: Spectrum of two *StructuredWithNoise* data sets with $d = 500$ and $d = 1000$, containing 50 clusters of 5000 points each.

Certainly, we do not want to actually create F. Instead, we construct a matrix A' where each row A_{i*} is replaced by $\sqrt{w_i}A_{i*}$ where w_i is the weight of the ith point. By linear algebra, we can verify that for each pair of left and right singular vectors u and v of F with singular value σ, there exists a vector u' such that u' and v are a pair of left and right singular vectors of A' for the same singular value. The reverse direction also holds. Thus, A' and F have the same best fit subspace and we can compute the SVD of A' in order to obtain it. After obtaining A'_ℓ, we divide each row i by $\sqrt{w_i}$ to get the projection of the points in A. Their weight does not change.

Notice that we cannot replace weighted points by some multiplied version when we input the points into BICO since the clustering behaviour of a weighted point differs from the clustering behaviour of any multiple (imagine a center that lies at the weighted point, so that it has no cost – but any multiplied point would have).

3 Experiments

The experiments were performed in three settings. For class I, all source codes were compiled using gcc 4.9.1, and experiments were performed on 20 identical machines with a 3.2 GHz AMD Phenom II™ X6 1090T processor and 8 GiB RAM. For class II, all source codes were compiled with gcc 4.8.2 and all experiments were performed on 7 identical machines with a 2.8 GHz Intel® E7400 processor and 8 GiB RAM. In class III, all source codes were compiled with gcc 4.9.1 and all experiments were performed on one machine with a 2.6 GHz Intel® Core™ i5-4210M CPU processor and 16 GiB RAM.

Our testbed consists of the following instances. Notice that we computed the spectrum for examples of the data set families. This gives an additional insight on the structure of the data sets.

Caltech128 The `Caltech128` instance was created from the Caltech101 image database [7] and consists of 128 SIFT descriptors [19], resulting in 128 dimensions and about 3.1 million points. The instance was used in [10] for BICO benchmarks and was provided to the authors by Grzeszick in a private communication.

StructuredWithNoise The idea of the `StructuredWithNoise` instances is to hide $\ell \in \mathbb{N}$ random point sets of $y \in \mathbb{N}$ points in $\mathbb{R}^d$. To build cluster $i \in \{1, \ldots, \ell\}$, select x dimensions $D_i = \{d_1, \ldots, d_x\} \subseteq \{1, \ldots, d\}$ uniformly at random. Then we build the y points for cluster i: For point j, choose the coordinates corresponding to D_i uniformly at random from $[-\Delta, \Delta]$. Select the remaining coordinates, i.e.,, the noise, uniformly at random from $[-\delta, \delta]$. This yields an instance with $\ell \cdot y$ points of dimension d. We fix $\Delta = 10$, $\delta = 1/2$. Figure 2 shows the spectrum

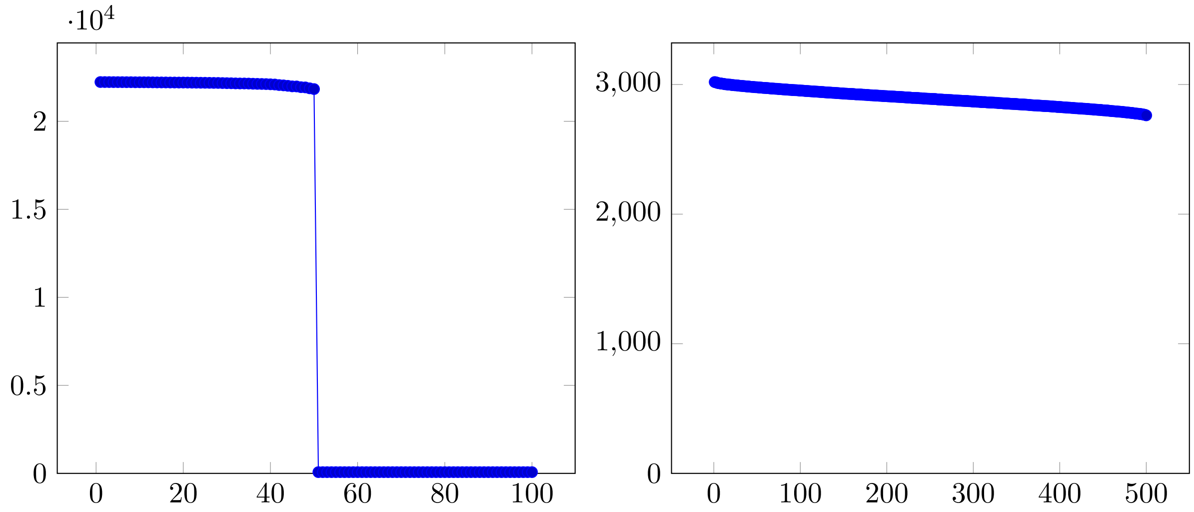

Figure 3: Spectrum of a LowerBound data set with $d = 10000$ and a random data set with $d = 10000$.

of two `StructuredWithNoise` data sets. We see that the first singular values are large, followed by slowly decreasing values until the descent steepens again.

LowerBound Arthur and Vassilvitskii [3] propose the following class of worst-case instances for the `kmeans++` algorithm. Define the (affine) (k, Δ)-simplex as the convex combination of the k unit vectors $e_1, \ldots, e_k$ in $\mathbb{R}^k$, scaled by $\Delta > 0$. Now, embed such a (k, Δ)-simplex $\mathfrak{S}$ in the first k dimensions of $\mathbb{R}^{k+n}$. Then use the remaining n dimensions of $\mathbb{R}^{k+n}$ to place a $(n/k, \delta)$-simplex S_i in each vertex i of $\mathfrak{S}$ such that all S_i use disjoint dimensions. Arthur and Vassilvitskii [3] prove that the `kmeans++` algorithm can achieve no better approximation ratio then $\Omega(\log N)$ on this class of instances, where N is the number of input points. We use a generator by Stallmann [22] to generate instance of this type. We fix $\delta = 100$ and $\Delta = 1000$. The `LowerBound` data sets have a nice structure for our experiments since the only the first singular values are significant as can be seen in the left diagram in Figure 3. Notice that we computed the first 100 singular values for a 10000-dimensional data set. The remaining values can only be smaller.

Random A `Random` data set is created by computing n^2 random numbers from $[-\Delta, \Delta]$ to form an n-dimensional data set with n points. We used $\Delta = 10$. Notice that the expected directional width is not equal for all directions (the points are drawn uniformly from a cube, not from a sphere). The resulting spectrum is slightly decreasing (see Figure 3, right diagram).

Since the algorithms are randomized, we repeated all experiments five times with the exception of the the test cases for the three largest `StructuredWithNoise` data sets because of computation times.

3.1 redSVD as a replacement for the lapack++ SVD

Replacing the exact SVD computation in our algorithm by an approximative one as outlined in Section 2 can only work if the approximation is fast and provides reliable results.

Additionally, we are interested in the factor of speed that can be gained by switching to redSVD from a full SVD computation.

To evaluate the redSVD performance, we use a test bed of `StructuredWithNoise` instances with varying values for y and d thus yielding instances from small to huge size. The results are depicted in Table 1.

We use redSVD to replace the input A by a matrix A'_ℓ. To measure the error of redSVD, we compare $||A - A'_\ell||_F^2$ to $||A - A_\ell||_F^2$, where A_ℓ is the matrix computed by the full SVD implementation in lapack++. The matrix obtained by projecting A to its best fit subspace of dimension ℓ minimizes the Frobenius norm of the difference to A, so this is a suitable measure to evaluate the redSVD result. The table shows the deviation of redSVD compared to the Frobenius distance of the matrix computed by the full SVD.

We performed the SVD comparison reducing the dimension to values in $\{100, 125, 150\}$.

We found that the error made by redSVD is indeed very small (less than 7% in all cases) while computation times become significantly faster: instances with 30,000 rows in 1000 columns can still be solved by redsvd in about 3s while `lapack++`'s takes 3000s on the same instace. RedSVD was able to compute approximate SVDs of matrices with 500,000 rows and 500 columns in 40s.

The limiting factor to solve larger instances is in both cases the memory limitation. The largest instance that we could compute full SVD on was with contains $n = 30000$ points in $d = 1000$ dimensions (constructed with $y = 300$ and $k = 100$). The redSVD approach uses much smaller matrices and thus it is possible to solve `StructureWithNoise` instances up to $n = 500000$ and $d = 500$. Then, however, it also stops working. Observe that computing the redSVD on this 500.000×500 matrix is still faster than one computation of a full SVD for instances with size $n = 10000$ and $d = 500$.

Group	Percent error				Full SVD CPU			redSVD CPU		
SWN, $k = 100$	min	max	avg	med	min	max	avg	min	max	avg
y-100-d-500-svd-100	0.52	0.79	0.66	0.68	167	169	168	0	0	0
y-100-d-500-svd-125	−2.91	−2.68	−2.76	−2.75	167	169	168	0	0	0
y-100-d-500-svd-150	−6.41	−6.19	−6.27	−6.25	167	169	168	0	0	0
y-100-d-1000-svd-100	1.38	1.51	1.44	1.45	350	353	351	0	0	0
y-100-d-1000-svd-125	−0.29	−0.13	−0.21	−0.21	350	353	351	0	0	0
y-100-d-1000-svd-150	−1.96	−1.82	−1.89	−1.88	350	353	351	0	1	1
y-200-d-500-svd-100	0.01	0.16	0.10	0.11	657	663	659	0	0	0
y-200-d-500-svd-125	−3.35	−3.14	−3.24	−3.25	657	663	659	1	1	1
y-200-d-500-svd-150	−6.77	−6.57	−6.66	−6.67	657	663	659	1	1	1
y-200-d-1000-svd-100	1.00	1.16	1.06	1.04	1347	1356	1351	1	1	1
y-200-d-1000-svd-125	−0.58	−0.43	−0.51	−0.51	1347	1356	1351	1	1	1
y-200-d-1000-svd-150	−2.17	−2.06	−2.12	−2.11	1347	1356	1351	1	2	2
y-300-d-500-svd-100	−0.17	−0.05	−0.13	−0.08	1480	1485	1483	1	1	1
y-300-d-500-svd-125	−3.51	−3.41	−3.46	−3.44	1480	1485	1483	1	1	1
y-300-d-500-svd-150	−6.88	−6.78	−6.84	−6.81	1480	1485	1483	1	2	2
y-300-d-1000-svd-100	0.91	0.93	0.92	0.93	3028	3039	3034	2	2	2
y-300-d-1000-svd-125	−0.66	−0.63	−0.65	−0.63	3028	3039	3034	2	2	2
y-300-d-1000-svd-150	−2.25	−2.21	−2.22	−2.21	3028	3039	3034	3	3	3
y-500-d-500-svd-100	—	—	—	—	—	—	—	2	2	2
y-500-d-500-svd-125	—	—	—	—	—	—	—	2	2	2
y-500-d-500-svd-150	—	—	—	—	—	—	—	3	3	3
y-500-d-1000-svd-100	—	—	—	—	—	—	—	3	3	3
y-500-d-1000-svd-125	—	—	—	—	—	—	—	4	4	4
y-500-d-1000-svd-150	—	—	—	—	—	—	—	4	4	4
y-1000-d-500-svd-100	—	—	—	—	—	—	—	4	4	4
y-1000-d-500-svd-125	—	—	—	—	—	—	—	5	5	5
y-1000-d-500-svd-150	—	—	—	—	—	—	—	6	6	6
y-1000-d-1000-svd-100	—	—	—	—	—	—	—	7	7	7
y-1000-d-1000-svd-125	—	—	—	—	—	—	—	8	8	8
y-1000-d-1000-svd-150	—	—	—	—	—	—	—	10	10	10
y-2000-d-500-svd-100	—	—	—	—	—	—	—	9	9	9
y-2000-d-500-svd-125	—	—	—	—	—	—	—	11	11	11
y-2000-d-500-svd-150	—	—	—	—	—	—	—	13	13	13
y-2000-d-1000-svd-100	—	—	—	—	—	—	—	16	16	16
y-2000-d-1000-svd-125	—	—	—	—	—	—	—	20	20	20
y-2000-d-1000-svd-150	—	—	—	—	—	—	—	23	23	23
y-5000-d-500-svd-100	—	—	—	—	—	—	—	24	24	24
y-5000-d-500-svd-125	—	—	—	—	—	—	—	29	29	29
y-5000-d-500-svd-150	—	—	—	—	—	—	—	36	36	36
y-5000-d-1000-svd-100	—	—	—	—	—	—	—	—	—	—
y-5000-d-1000-svd-125	—	—	—	—	—	—	—	—	—	—
y-5000-d-1000-svd-150	—	—	—	—	—	—	—	—	—	—
y-10000-d-500-svd-100	—	—	—	—	—	—	—	—	—	—
y-10000-d-500-svd-125	—	—	—	—	—	—	—	—	—	—
y-10000-d-500-svd-150	—	—	—	—	—	—	—	—	—	—
y-10000-d-1000-svd-100	—	—	—	—	—	—	—	—	—	—

Table 1: Comparison of the full SVD by lapack++ with redSVD on various randomized instances with $k = 100$ and varying parameters. The table shows error percentage of the approximate solution and the running times in seconds. Notice that the number of points in the instances is $k \cdot y$. Experiment belongs to class I. Given a matrix A, its full SVD A_ℓ and its approximate SVD A'_ℓ, we verify the accuracy of the redSVD approximation by comparing $||A - A'_\ell||_F^2$ to $||A - A_\ell||_F^2$ on instances of the `StructureWithNoise` class. The matrix obtained by projecting A to its best fit subspace of dimension ℓ minimizes the Frobenius norm of the difference to A, so this is a suitable measure to evaluate the redSVD result.

3.2 Performance of BICO, Piecy and Piecy-MR

BICO.

Table 2 contains the basic test cases and reports the results that BICO achieved when run on the test case directly. Notice that we use the current version of the source code from the BICO website. In contrast to the version used in [9], this version has varying running times. This shows both in the BICO experiments itself as in the experiments for *piecy* and *piecy-mr* since they both use BICO. For example, consider the varying running time of BICO on the `enron` data set. Obviously, *piecy* and *piecy-mr* will improve when the source code of BICO is updated. For this reason, we will pay most attention to the median of the running times and not the average running time.

In all tables, the parameters are listed in the caption if they are equal for all test cases in the table, or at the start of each line if they vary. We denote the number of points by n, the dimension by d and the number of centers by k.

Group	Cost				Running time			
	min	max	average	median	min	max	avg	med
`LowerBound`, experiments belong to class II								
k-10-n-10^4-d-10010	5.00×10^7	5.00×10^7	5.00×10^7	5.00×10^7	74	77	75.6	76
k-50-n-10^4-d-10050	4.98×10^7	14.88×10^7	8.94×10^7	4.98×10^7	78	79	78.7	79
`BagOfWords`, experiments belong to class II								
enron-k-10	1.63×10^7	1.69×10^7	1.65×10^7	1.66×10^7	480	1679	611.9	491
kos-k-2	3.90×10^5	3.95×10^5	3.92×10^5	3.91×10^5	10	11	10.9	11
`Caltech128`, experiments belong to class I								
k-5	4.23×10^{11}	4.23×10^{11}	4.23×10^{11}	4.23×10^{11}	319	319	319.1	319
k-10	4.13×10^{11}	4.13×10^{11}	4.13×10^{11}	4.13×10^{11}	366	366	366.0	366
k-50	3.43×10^{11}	3.43×10^{11}	3.43×10^{11}	3.43×10^{11}	428	428	427.6	428
k-100	3.04×10^{11}	3.04×10^{11}	3.04×10^{11}	3.04×10^{11}	503	503	502.9	503
k-250	2.74×10^{11}	2.74×10^{11}	2.74×10^{11}	2.74×10^{11}	571	571	571.1	571
k-1000	2.34×10^{11}	2.34×10^{11}	2.34×10^{11}	2.34×10^{11}	560	560	559.7	560
`Random`, experiments belong to class II								
n-10^6-d-1000-k-10	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	1058	2126	1718.6	1816
n-10^6-d-1000-k-20	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	2578	4792	3522.8	2952
n-10^6-d-1000-k-50	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	1004	4466	2326.8	1819
`StructuredWithNoise`, experiments belong to class I								
y-5000-d-1000-k-10	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	368	1227	610.1	592
y-5000-d-1000-k-20	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	591	2204	1217.8	1085
y-5000-d-1000-k-50	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	547	2865	1133.8	872
y-5000-d-1000-k-100	1.69×10^9	1.70×10^9	1.70×10^9	1.70×10^9	714	7679	2275.1	1359
y-10000-d-500-k-10	3.36×10^9	3.37×10^9	3.36×10^9	3.36×10^9	411	1284	740.9	691
y-10000-d-500-k-20	3.35×10^9	3.37×10^9	3.36×10^9	3.36×10^9	435	2392	1030.0	805
y-10000-d-500-k-50	3.34×10^9	3.37×10^9	3.36×10^9	3.36×10^9	576	4772	2295.2	2084
y-10000-d-500-k-100	3.32×10^9	3.37×10^9	3.34×10^9	3.34×10^9	846	6233	2168.6	1434
y-10000-d-1000-k-10	3.41×10^9	3.41×10^9	3.41×10^9	3.41×10^9	722	2669	1454.9	1244
y-10000-d-1000-k-20	3.40×10^9	3.41×10^9	3.41×10^9	3.41×10^9	770	4521	2350.0	2230
y-10000-d-1000-k-50	3.40×10^9	3.41×10^9	3.40×10^9	3.40×10^9	1299	8648	4547.8	4897
y-10000-d-1000-k-100	3.39×10^9	3.41×10^9	3.40×10^9	3.40×10^9	1477	8626	3602.6	2605
`StructuredWithNoise`, experiments belong to class III								
y-1000000-d-500-k-50	3.34×10^9	3.35×10^9	3.34×10^9	3.34×10^9	335	4192	1280.2	938

Table 2: BICO results.

Piecy.

For piecy, we test the influence of two parameters, the piece size, abbreviation *ps*, and the number of dimensions to which we project the points, abbreviation *svd*. We computed an extensive number of test cases for the data set `CalTech128` to study the influence of the parameters. Table 3 summarizes the results for piecy. For $k = 5, 10, 50$, piecy is *always* faster than BICO. The table shows that larger values of *svd* increase the running time, which is expected, but stays below the running time of BICO for these test cases. The accuracy of piecy is high, in particular for larger svd values. At $k = 100$, the situation starts to change as there are three test cases where piecy is slower than BICO. For $k = 250, 1000$ the results by piecy become somewhat unpredictable. Notice that the number of centers is here higher than the input dimension of the points (which is 128). Thus, piecy cannot gain anything from projecting to a number of dimensions $\geq k$, and the SVD processing becomes overhead. It is thus clear that piecy does not perform as well on these test cases.

Group	Cost				Running time			
	min	max	average	median	min	max	avg	med
$k = 5$								
ps-1000-s-10	4.37×10^{11}	4.65×10^{11}	4.52×10^{11}	4.54×10^{11}	202	244	217.7	209
ps-1000-s-20	4.35×10^{11}	4.46×10^{11}	4.38×10^{11}	4.36×10^{11}	201	259	228.8	235
ps-1000-s-50	4.25×10^{11}	4.51×10^{11}	4.36×10^{11}	4.36×10^{11}	224	256	236.5	234
ps-1000-s-75	4.22×10^{11}	4.36×10^{11}	4.28×10^{11}	4.27×10^{11}	255	285	274.4	279
ps-1000-s-100	4.20×10^{11}	4.66×10^{11}	4.38×10^{11}	4.40×10^{11}	294	333	313.8	317
ps-2000-s-10	4.39×10^{11}	4.65×10^{11}	4.51×10^{11}	4.51×10^{11}	203	246	220.4	222
ps-2000-s-20	4.39×10^{11}	4.64×10^{11}	4.50×10^{11}	4.45×10^{11}	211	297	248.8	241
ps-2000-s-50	4.22×10^{11}	4.44×10^{11}	4.32×10^{11}	4.31×10^{11}	234	289	253.7	249
ps-2000-s-75	4.20×10^{11}	4.44×10^{11}	4.35×10^{11}	4.36×10^{11}	262	294	282.0	287
ps-2000-s-100	4.20×10^{11}	4.44×10^{11}	4.32×10^{11}	4.32×10^{11}	280	310	297.4	296
ps-5000-s-10	4.47×10^{11}	4.71×10^{11}	4.57×10^{11}	4.59×10^{11}	213	257	237.6	239
ps-5000-s-20	4.30×10^{11}	4.44×10^{11}	4.36×10^{11}	4.33×10^{11}	240	290	256.5	245
ps-5000-s-50	4.21×10^{11}	4.33×10^{11}	4.28×10^{11}	4.28×10^{11}	245	296	264.7	260
ps-5000-s-75	4.28×10^{11}	4.33×10^{11}	4.31×10^{11}	4.32×10^{11}	263	319	291.8	290
ps-5000-s-100	4.25×10^{11}	4.41×10^{11}	4.32×10^{11}	4.30×10^{11}	277	310	292.8	291
ps-10000-s-10	4.46×10^{11}	4.60×10^{11}	4.54×10^{11}	4.53×10^{11}	215	291	241.4	242
ps-10000-s-20	4.36×10^{11}	4.44×10^{11}	4.39×10^{11}	4.37×10^{11}	214	271	243.6	245
ps-10000-s-50	4.27×10^{11}	4.30×10^{11}	4.28×10^{11}	4.28×10^{11}	238	251	244.4	244
ps-10000-s-75	4.27×10^{11}	4.60×10^{11}	4.42×10^{11}	4.38×10^{11}	262	283	272.4	267
ps-10000-s-100	4.28×10^{11}	4.49×10^{11}	4.40×10^{11}	4.41×10^{11}	268	291	280.0	281
$k = 10$								
ps-1000-s-10	4.13×10^{11}	4.22×10^{11}	4.18×10^{11}	4.20×10^{11}	225	280	248.5	252
ps-1000-s-20	3.99×10^{11}	4.13×10^{11}	4.06×10^{11}	4.07×10^{11}	266	390	312.6	303
ps-1000-s-50	3.94×10^{11}	4.10×10^{11}	4.00×10^{11}	3.99×10^{11}	242	319	271.0	263
ps-1000-s-75	3.89×10^{11}	4.07×10^{11}	3.97×10^{11}	3.97×10^{11}	280	367	321.3	336
ps-1000-s-100	3.95×10^{11}	4.06×10^{11}	3.98×10^{11}	3.95×10^{11}	309	341	327.2	331
ps-2000-s-10	4.21×10^{11}	4.62×10^{11}	4.33×10^{11}	4.29×10^{11}	227	339	269.6	264
ps-2000-s-20	4.01×10^{11}	4.18×10^{11}	4.07×10^{11}	4.06×10^{11}	238	268	253.1	250
ps-2000-s-50	3.93×10^{11}	3.99×10^{11}	3.97×10^{11}	3.97×10^{11}	252	282	268.1	268
ps-2000-s-75	3.89×10^{11}	4.07×10^{11}	3.94×10^{11}	3.90×10^{11}	267	363	312.6	305
ps-2000-s-100	3.94×10^{11}	4.05×10^{11}	3.99×10^{11}	4.00×10^{11}	318	384	350.5	359
ps-5000-s-10	4.23×10^{11}	4.30×10^{11}	4.28×10^{11}	4.29×10^{11}	241	364	303.4	285
ps-5000-s-20	4.03×10^{11}	4.23×10^{11}	4.12×10^{11}	4.14×10^{11}	233	272	256.3	264
ps-5000-s-50	3.94×10^{11}	4.12×10^{11}	4.03×10^{11}	4.04×10^{11}	245	347	283.6	287
ps-5000-s-75	3.89×10^{11}	4.01×10^{11}	3.97×10^{11}	3.97×10^{11}	266	318	282.2	271
ps-5000-s-100	3.87×10^{11}	4.14×10^{11}	3.99×10^{11}	3.98×10^{11}	298	324	308.9	308
ps-10000-s-10	4.24×10^{11}	4.29×10^{11}	4.27×10^{11}	4.27×10^{11}	260	295	277.2	281
ps-10000-s-20	4.05×10^{11}	4.27×10^{11}	4.15×10^{11}	4.12×10^{11}	239	263	253.6	254
ps-10000-s-50	3.88×10^{11}	4.07×10^{11}	3.99×10^{11}	4.00×10^{11}	253	311	282.1	277
ps-10000-s-75	4.01×10^{11}	4.07×10^{11}	4.04×10^{11}	4.04×10^{11}	277	310	293.8	289
ps-10000-s-100	3.92×10^{11}	4.03×10^{11}	3.97×10^{11}	3.96×10^{11}	308	367	326.9	313
$k = 50$								
ps-1000-s-10	3.66×10^{11}	3.74×10^{11}	3.69×10^{11}	3.70×10^{11}	333	442	381.5	367
ps-1000-s-20	3.43×10^{11}	3.56×10^{11}	3.51×10^{11}	3.51×10^{11}	269	376	334.7	328
ps-1000-s-50	3.30×10^{11}	3.34×10^{11}	3.32×10^{11}	3.33×10^{11}	306	415	355.6	341
ps-1000-s-75	3.25×10^{11}	3.37×10^{11}	3.30×10^{11}	3.31×10^{11}	351	470	395.3	385
ps-1000-s-100	3.24×10^{11}	3.33×10^{11}	3.29×10^{11}	3.30×10^{11}	377	412	397.8	410
ps-2000-s-10	3.73×10^{11}	3.80×10^{11}	3.76×10^{11}	3.77×10^{11}	312	480	364.6	350
ps-2000-s-20	3.46×10^{11}	3.55×10^{11}	3.52×10^{11}	3.53×10^{11}	329	485	419.4	415
ps-2000-s-50	3.30×10^{11}	3.44×10^{11}	3.37×10^{11}	3.36×10^{11}	343	477	399.2	372

Table 3: Piecy on `Caltech128`, experiments belong to class I.

	min	max	average	median	min	max	avg	med
ps-2000-s-75	3.27×10^{11}	3.31×10^{11}	3.28×10^{11}	3.28×10^{11}	310	505	387.2	343
ps-2000-s-100	3.28×10^{11}	3.34×10^{11}	3.31×10^{11}	3.31×10^{11}	344	423	370.9	357
ps-5000-s-10	3.78×10^{11}	3.83×10^{11}	3.80×10^{11}	3.80×10^{11}	312	414	361.0	361
ps-5000-s-20	3.53×10^{11}	3.59×10^{11}	3.56×10^{11}	3.56×10^{11}	286	444	344.2	308
ps-5000-s-50	3.29×10^{11}	3.39×10^{11}	3.34×10^{11}	3.36×10^{11}	310	427	357.0	341
ps-5000-s-75	3.25×10^{11}	3.41×10^{11}	3.34×10^{11}	3.35×10^{11}	326	401	355.2	348
ps-5000-s-100	3.27×10^{11}	3.35×10^{11}	3.30×10^{11}	3.28×10^{11}	324	505	411.8	391
ps-10000-s-10	3.81×10^{11}	3.90×10^{11}	3.85×10^{11}	3.85×10^{11}	283	471	337.1	313
ps-10000-s-20	3.56×10^{11}	3.65×10^{11}	3.60×10^{11}	3.60×10^{11}	321	396	349.8	348
ps-10000-s-50	3.30×10^{11}	3.42×10^{11}	3.35×10^{11}	3.36×10^{11}	313	384	356.6	362
ps-10000-s-75	3.28×10^{11}	3.39×10^{11}	3.32×10^{11}	3.29×10^{11}	327	452	377.7	363
ps-10000-s-100	3.25×10^{11}	3.38×10^{11}	3.33×10^{11}	3.35×10^{11}	348	479	399.5	381
$k = 100$								
ps-1000-s-10	3.52×10^{11}	3.57×10^{11}	3.54×10^{11}	3.53×10^{11}	399	629	552.8	572
ps-1000-s-20	3.24×10^{11}	3.30×10^{11}	3.27×10^{11}	3.28×10^{11}	373	625	454.3	398
ps-1000-s-50	3.02×10^{11}	3.13×10^{11}	3.07×10^{11}	3.06×10^{11}	420	608	503.1	471
ps-1000-s-75	3.04×10^{11}	3.08×10^{11}	3.05×10^{11}	3.05×10^{11}	366	481	429.8	424
ps-1000-s-100	3.00×10^{11}	3.09×10^{11}	3.04×10^{11}	3.06×10^{11}	464	795	557.4	484
ps-2000-s-10	3.54×10^{11}	3.59×10^{11}	3.57×10^{11}	3.57×10^{11}	381	746	497.3	444
ps-2000-s-20	3.28×10^{11}	3.35×10^{11}	3.32×10^{11}	3.32×10^{11}	438	527	487.9	490
ps-2000-s-50	3.06×10^{11}	3.16×10^{11}	3.10×10^{11}	3.09×10^{11}	341	475	412.2	399
ps-2000-s-75	3.05×10^{11}	3.11×10^{11}	3.08×10^{11}	3.08×10^{11}	382	515	422.2	393
ps-2000-s-100	2.98×10^{11}	3.08×10^{11}	3.04×10^{11}	3.04×10^{11}	492	657	587.2	595
ps-5000-s-10	3.63×10^{11}	3.68×10^{11}	3.66×10^{11}	3.67×10^{11}	368	494	416.7	404
ps-5000-s-20	3.31×10^{11}	3.39×10^{11}	3.34×10^{11}	3.33×10^{11}	448	723	550.0	528
ps-5000-s-50	3.03×10^{11}	3.16×10^{11}	3.08×10^{11}	3.09×10^{11}	351	602	471.5	493
ps-5000-s-75	3.05×10^{11}	3.14×10^{11}	3.08×10^{11}	3.07×10^{11}	357	812	487.9	421
ps-5000-s-100	3.00×10^{11}	3.10×10^{11}	3.05×10^{11}	3.02×10^{11}	458	610	516.2	511
ps-10000-s-10	3.65×10^{11}	3.71×10^{11}	3.67×10^{11}	3.66×10^{11}	306	441	382.7	378
ps-10000-s-20	3.35×10^{11}	3.40×10^{11}	3.37×10^{11}	3.37×10^{11}	341	534	449.3	468
ps-10000-s-50	3.08×10^{11}	3.17×10^{11}	3.11×10^{11}	3.09×10^{11}	426	652	508.4	467
ps-10000-s-75	3.03×10^{11}	3.16×10^{11}	3.08×10^{11}	3.07×10^{11}	358	610	474.3	441
ps-10000-s-100	3.00×10^{11}	3.12×10^{11}	3.05×10^{11}	3.04×10^{11}	390	492	465.2	482
$k = 250$								
ps-1000-s-10	3.27×10^{11}	3.29×10^{11}	3.28×10^{11}	3.29×10^{11}	466	976	817.2	939
ps-1000-s-20	2.97×10^{11}	3.04×10^{11}	2.99×10^{11}	2.98×10^{11}	441	883	681.4	667
ps-1000-s-50	2.77×10^{11}	2.82×10^{11}	2.79×10^{11}	2.79×10^{11}	437	736	599.8	650
ps-1000-s-75	2.73×10^{11}	2.79×10^{11}	2.75×10^{11}	2.75×10^{11}	440	767	633.9	608
ps-1000-s-100	2.68×10^{11}	2.80×10^{11}	2.74×10^{11}	2.73×10^{11}	468	816	686.6	692
ps-2000-s-10	3.33×10^{11}	3.38×10^{11}	3.36×10^{11}	3.36×10^{11}	414	1226	759.5	843
ps-2000-s-20	3.03×10^{11}	3.09×10^{11}	3.07×10^{11}	3.09×10^{11}	435	815	601.0	581
ps-2000-s-50	2.78×10^{11}	2.87×10^{11}	2.82×10^{11}	2.82×10^{11}	472	672	539.1	531
ps-2000-s-75	2.72×10^{11}	2.83×10^{11}	2.78×10^{11}	2.79×10^{11}	386	729	540.2	447
ps-2000-s-100	2.72×10^{11}	2.82×10^{11}	2.78×10^{11}	2.78×10^{11}	489	642	558.4	513
ps-5000-s-10	3.41×10^{11}	3.47×10^{11}	3.45×10^{11}	3.45×10^{11}	330	713	509.0	471
ps-5000-s-20	3.08×10^{11}	3.17×10^{11}	3.12×10^{11}	3.12×10^{11}	461	868	686.7	791
ps-5000-s-50	2.83×10^{11}	2.86×10^{11}	2.84×10^{11}	2.84×10^{11}	435	726	594.9	663
ps-5000-s-75	2.72×10^{11}	2.81×10^{11}	2.75×10^{11}	2.74×10^{11}	437	762	658.0	732
ps-5000-s-100	2.73×10^{11}	2.77×10^{11}	2.75×10^{11}	2.76×10^{11}	445	775	646.8	673
ps-10000-s-10	3.45×10^{11}	3.50×10^{11}	3.48×10^{11}	3.47×10^{11}	314	619	485.4	469
ps-10000-s-20	3.12×10^{11}	3.19×10^{11}	3.14×10^{11}	3.13×10^{11}	462	829	690.7	703
ps-10000-s-50	2.81×10^{11}	2.83×10^{11}	2.82×10^{11}	2.83×10^{11}	450	823	589.3	589
ps-10000-s-75	2.74×10^{11}	2.86×10^{11}	2.79×10^{11}	2.81×10^{11}	417	742	556.9	485
ps-10000-s-100	2.71×10^{11}	2.82×10^{11}	2.76×10^{11}	2.77×10^{11}	476	823	671.3	727
$k = 1000$								
ps-1000-s-10	2.94×10^{11}	2.97×10^{11}	2.96×10^{11}	2.96×10^{11}	379	9746	3319.7	478
ps-1000-s-20	2.62×10^{11}	2.68×10^{11}	2.65×10^{11}	2.66×10^{11}	428	6384	3026.1	3668
ps-1000-s-50	2.35×10^{11}	2.43×10^{11}	2.39×10^{11}	2.40×10^{11}	883	4566	2324.2	1801
ps-1000-s-75	2.32×10^{11}	2.38×10^{11}	2.35×10^{11}	2.35×10^{11}	586	4788	1601.1	859
ps-1000-s-100	2.32×10^{11}	2.33×10^{11}	2.33×10^{11}	2.33×10^{11}	1076	3793	2248.7	3417
ps-2000-s-10	3.03×10^{11}	3.09×10^{11}	3.06×10^{11}	3.05×10^{11}	317	6692	3446.5	3786
ps-2000-s-20	2.71×10^{11}	2.77×10^{11}	2.74×10^{11}	2.73×10^{11}	459	5175	2885.6	3583
ps-2000-s-50	2.39×10^{11}	2.45×10^{11}	2.42×10^{11}	2.42×10^{11}	573	5388	2791.4	2773
ps-2000-s-75	2.32×10^{11}	2.37×10^{11}	2.35×10^{11}	2.36×10^{11}	653	2866	1701.0	1947
ps-2000-s-100	2.29×10^{11}	2.38×10^{11}	2.33×10^{11}	2.33×10^{11}	585	5912	1748.2	706
ps-5000-s-10	3.17×10^{11}	3.18×10^{11}	3.17×10^{11}	3.18×10^{11}	4664	7886	6440.7	6910

Table 3: Piecy on `Caltech128`, experiments belong to class I.

	min	max	average	median	min	max	avg	med
ps-5000-s-20	2.78×10^{11}	2.85×10^{11}	2.81×10^{11}	2.81×10^{11}	536	5521	3058.6	3454
ps-5000-s-50	2.43×10^{11}	2.46×10^{11}	2.44×10^{11}	2.44×10^{11}	526	2366	954.0	590
ps-5000-s-75	2.35×10^{11}	2.38×10^{11}	2.37×10^{11}	2.37×10^{11}	809	3889	1839.0	961
ps-5000-s-100	2.32×10^{11}	2.36×10^{11}	2.33×10^{11}	2.33×10^{11}	626	6184	3354.7	4155
ps-10000-s-10	3.28×10^{11}	3.30×10^{11}	3.29×10^{11}	3.29×10^{11}	262	4210	2197.8	1526
ps-10000-s-20	2.82×10^{11}	2.87×10^{11}	2.84×10^{11}	2.84×10^{11}	960	4509	2909.1	3196
ps-10000-s-50	2.44×10^{11}	2.50×10^{11}	2.47×10^{11}	2.48×10^{11}	496	4026	1776.6	791
ps-10000-s-75	2.35×10^{11}	2.38×10^{11}	2.37×10^{11}	2.37×10^{11}	598	5633	2186.3	1202
ps-10000-s-100	2.31×10^{11}	2.34×10^{11}	2.33×10^{11}	2.33×10^{11}	592	4653	1874.3	864

Table 3: Piecy on `Caltech128` (continued), experiments belong to class I.

On the `Random` instance, piecy performs rather badly. The instance is large (one million points with 1000 dimensions, i.e., a total of 10^9 input numbers). In this case, most of the advantage due to the dimensionality reduction is lost because too many pieces are processed and contribute to the intrinsic dimension of the point set that is given to BICO. A similar behavior can be observed for the three largest `StructuredWithNoise` data sets. In particular when n reaches a million points, piecys running time goes up.

On the smaller `LowerBound` test cases though, piecy again outperforms BICO's running time. The `LowerBound` instances have a huge dimension of 10^5 but the number of points is also bounded by 10^5. Thus, there is less time for piecy to accumulate to many intrinsic dimensions.

Group	Cost				Running time			
	min	max	average	median	min	max	avg	med
$k = 10$								
ps-2000-svd-10	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	1013	1464	1283.2	1289
ps-2000-svd-20	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	997	1841	1536.3	1666
ps-2000-svd-50	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	945	2235	1452.7	1347
ps-2000-svd-75	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	755	2399	1771.7	1981
ps-4000-svd-10	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	1178	2048	1491.4	1340
ps-4000-svd-20	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	1083	2156	1536.4	1408
ps-4000-svd-50	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	759	1890	1423.3	1561
ps-4000-svd-75	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	953	2148	1628.7	1495
ps-10000-svd-10	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	1447	2065	1650.2	1511
ps-10000-svd-20	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	1091	1887	1500.9	1537
ps-10000-svd-50	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	1058	2752	1977.8	2115
ps-10000-svd-75	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	1071	2836	1711.0	1523
$k = 20$								
ps-2000-svd-20	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	2484	4608	3539.3	3853
ps-2000-svd-50	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	950	4724	3133.2	3233
ps-2000-svd-75	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	1359	2931	2303.3	2545
ps-4000-svd-20	3.32×10^{10}	3.33×10^{10}	3.32×10^{10}	3.32×10^{10}	2160	4826	3257.8	2846
ps-4000-svd-50	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	2469	5351	3685.4	3384
ps-4000-svd-75	3.32×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	1119	2563	1896.9	1930
ps-10000-svd-20	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	1623	4650	2893.0	2634
ps-10000-svd-50	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	1493	4866	3010.0	2701
ps-10000-svd-75	3.32×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	2523	3985	3268.6	3454
$k = 50$								
ps-2000-svd-50	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	1798	4881	3537.6	3742
ps-2000-svd-75	3.32×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	1047	9855	4069.4	3864
ps-4000-svd-50	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	1350	7193	4015.3	4781
ps-4000-svd-75	3.32×10^{10}	3.33×10^{10}	3.32×10^{10}	3.32×10^{10}	1382	6330	4421.8	4843
ps-10000-svd-50	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	1138	7283	3716.0	3536
ps-10000-svd-75	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	1585	7521	4017.2	2233

Table 4: Piecy on `Random` instances with $n = 10^6$ and $d = 1000$ and varying parameters, experiments belong to class II.

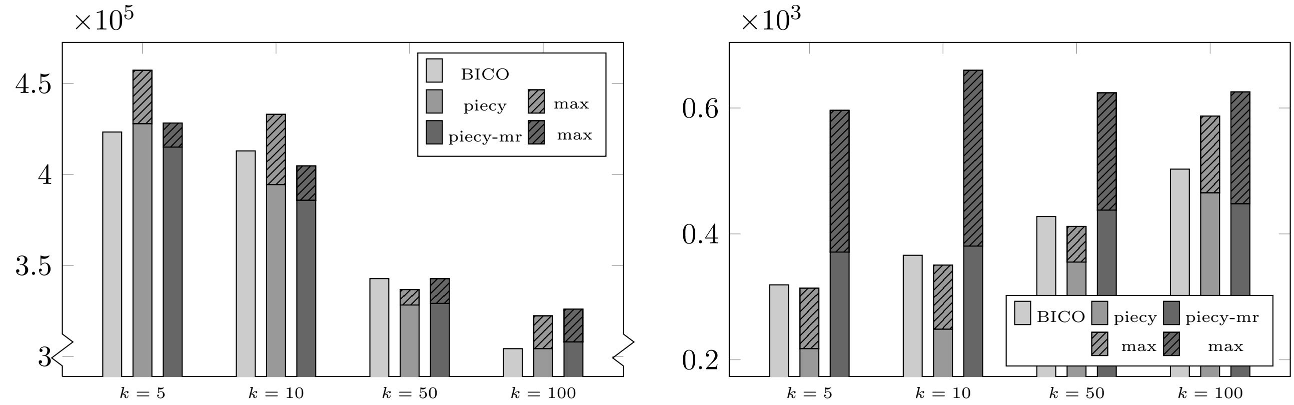

Figure 4: Results for the `Caltech128` data set. Left side reports quality, right side run times. Variances stem from different parameters.

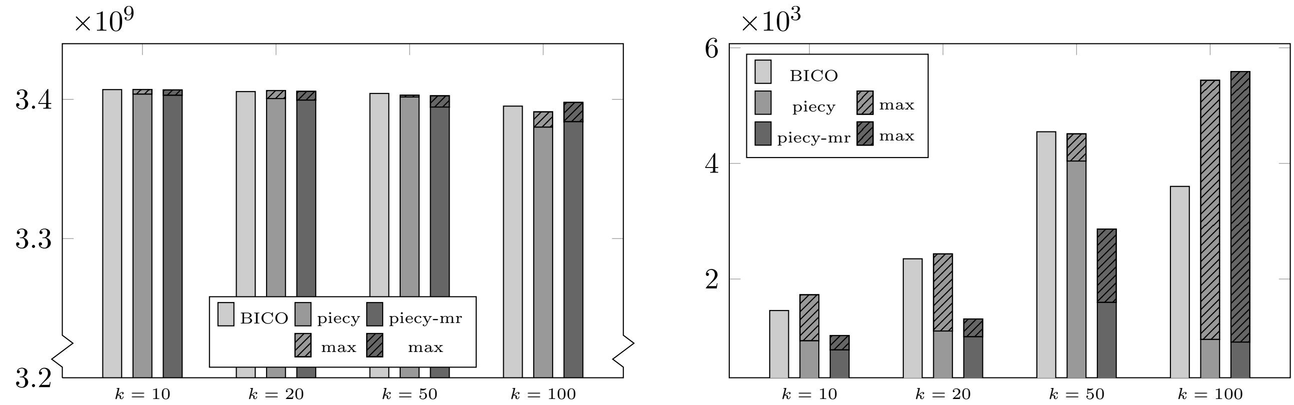

Figure 5: Results for a `StructuredWithNoise` data set with 10^6 points in 10^3 dimensions. Left side reports quality, right side run times. Variances stem from different parameters.

Group	Cost				Running time			
	min	max	average	median	min	max	avg	med
k-10-d-10010-k-10-svd-15	5.72×10^7	6.11×10^7	5.86×10^7	5.84×10^7	53	76	62.9	58
k-10-d-10010-k-10-svd-20	5.49×10^7	5.68×10^7	5.58×10^7	5.58×10^7	56	94	69.1	61
k-50-d-10050-k-50-svd-75	5.69×10^7	5.77×10^7	5.73×10^7	5.73×10^7	78	78	78.2	78
k-50-d-10050-k-50-svd-100	5.49×10^7	5.52×10^7	5.50×10^7	5.50×10^7	79	80	78.9	79

Table 5: Piecy on `LowerBound` instances with $n = 10000$ and a piece size of 2000, experiments belong to class II.

Piecy-mr.

Piecy-mr also uses *ps*, the piece size, as a parameter, as well as *svd*, the number of dimensions to project to. The additional parameter *np* is the number of pieces that are processed into the same BICO instance.

For `CalTech128`, the overhead of piecy-mr does not pay off and it performs worse than piecy. Results for this data set ar shown in Figure 4 On the `LowerBound` test cases, piecy-mr is always slightly faster than BICO and comparable to piecy. On the `Random` instances, piecy-mr is much faster than BICO, close to a factor of 2 on most test cases. This is in particular a much better running time than for piecy. The fact that `Random` has both a huge number of points and a high dimension means that the strength of piecy-mr shows and is not dominated by the overhead of the computation tree. The study of the three `StructuredWithNoise` data sets confirms this behaviour. In all three cases, the running time of piecy-mr is much faster or at least comparable to BICO with very few exceptions. This effect is particularly clear for the largest data set with one million points and a dimension of 1000, showing the speed of piecy-mr for large high-dimensional data sets. Figure 5 shows results for this data set. Notice that the large variance for piecy and piecy-mr is due to very different parameter choices. The best parameter choices yield a significant speed-up, particularly for large values of k.

Group	Cost				Running time			
	min	max	average	median	min	max	avg	med
$k = 5$								
ps-5000-np-5-s-50	4.28×10^{11}	4.29×10^{11}	4.28×10^{11}	4.28×10^{11}	469	504	483.6	479
ps-5000-np-5-s-75	4.25×10^{11}	4.29×10^{11}	4.27×10^{11}	4.27×10^{11}	491	514	501.5	500
ps-5000-np-5-s-100	4.23×10^{11}	4.29×10^{11}	4.27×10^{11}	4.27×10^{11}	502	548	525.5	525
ps-5000-np-10-s-50	4.19×10^{11}	4.19×10^{11}	4.19×10^{11}	4.19×10^{11}	365	420	394.6	395
ps-5000-np-10-s-75	4.17×10^{11}	4.19×10^{11}	4.18×10^{11}	4.18×10^{11}	418	480	439.0	430
ps-5000-np-10-s-100	4.17×10^{11}	4.19×10^{11}	4.19×10^{11}	4.19×10^{11}	449	490	468.0	468
ps-5000-np-15-s-50	4.17×10^{11}	4.19×10^{11}	4.18×10^{11}	4.19×10^{11}	355	394	371.1	367
ps-5000-np-15-s-75	4.16×10^{11}	4.18×10^{11}	4.17×10^{11}	4.17×10^{11}	397	461	428.3	437
ps-5000-np-15-s-100	4.16×10^{11}	4.19×10^{11}	4.17×10^{11}	4.17×10^{11}	444	478	468.4	474
ps-10000-np-5-s-50	4.16×10^{11}	4.18×10^{11}	4.17×10^{11}	4.16×10^{11}	528	591	566.8	581
ps-10000-np-5-s-75	4.14×10^{11}	4.17×10^{11}	4.15×10^{11}	4.15×10^{11}	543	621	590.0	602
ps-10000-np-5-s-100	4.14×10^{11}	4.17×10^{11}	4.16×10^{11}	4.16×10^{11}	535	647	596.3	602
ps-10000-np-10-s-50	4.18×10^{11}	4.24×10^{11}	4.21×10^{11}	4.19×10^{11}	420	551	473.4	475
ps-10000-np-10-s-75	4.18×10^{11}	4.19×10^{11}	4.18×10^{11}	4.18×10^{11}	487	556	517.1	504
ps-10000-np-10-s-100	4.17×10^{11}	4.24×10^{11}	4.19×10^{11}	4.18×10^{11}	495	579	541.3	555
ps-10000-np-15-s-50	4.15×10^{11}	4.20×10^{11}	4.17×10^{11}	4.17×10^{11}	413	451	437.4	440
ps-10000-np-15-s-75	4.15×10^{11}	4.18×10^{11}	4.17×10^{11}	4.17×10^{11}	443	485	467.9	471
ps-10000-np-15-s-100	4.16×10^{11}	4.18×10^{11}	4.17×10^{11}	4.16×10^{11}	470	630	524.3	508
$k = 10$								
ps-5000-np-5-s-50	4.01×10^{11}	4.07×10^{11}	4.03×10^{11}	4.02×10^{11}	454	531	487.4	480
ps-5000-np-5-s-75	4.01×10^{11}	4.08×10^{11}	4.05×10^{11}	4.04×10^{11}	480	513	495.4	496
ps-5000-np-5-s-100	4.00×10^{11}	4.08×10^{11}	4.03×10^{11}	4.03×10^{11}	503	557	520.8	516
ps-5000-np-10-s-50	3.94×10^{11}	3.99×10^{11}	3.97×10^{11}	3.96×10^{11}	371	428	392.2	393
ps-5000-np-10-s-75	3.92×10^{11}	3.98×10^{11}	3.94×10^{11}	3.94×10^{11}	405	446	430.2	434
ps-5000-np-10-s-100	3.92×10^{11}	3.98×10^{11}	3.95×10^{11}	3.95×10^{11}	448	489	465.1	470
ps-5000-np-15-s-50	3.90×10^{11}	3.92×10^{11}	3.91×10^{11}	3.91×10^{11}	360	404	380.6	376
ps-5000-np-15-s-75	3.88×10^{11}	3.91×10^{11}	3.89×10^{11}	3.88×10^{11}	382	419	407.2	415
ps-5000-np-15-s-100	3.87×10^{11}	3.94×10^{11}	3.90×10^{11}	3.90×10^{11}	407	481	444.5	447
ps-10000-np-5-s-50	3.87×10^{11}	3.90×10^{11}	3.88×10^{11}	3.89×10^{11}	569	725	617.7	588
ps-10000-np-5-s-75	3.84×10^{11}	3.89×10^{11}	3.86×10^{11}	3.86×10^{11}	557	645	603.2	614
ps-10000-np-5-s-100	3.84×10^{11}	3.89×10^{11}	3.86×10^{11}	3.85×10^{11}	600	701	659.9	674
ps-10000-np-10-s-50	3.89×10^{11}	3.92×10^{11}	3.91×10^{11}	3.90×10^{11}	474	512	492.2	494
ps-10000-np-10-s-75	3.87×10^{11}	3.94×10^{11}	3.90×10^{11}	3.90×10^{11}	475	541	494.4	487
ps-10000-np-10-s-100	3.87×10^{11}	3.90×10^{11}	3.88×10^{11}	3.88×10^{11}	505	552	530.1	536
ps-10000-np-15-s-50	3.89×10^{11}	3.91×10^{11}	3.90×10^{11}	3.89×10^{11}	412	484	455.9	465
ps-10000-np-15-s-75	3.87×10^{11}	3.93×10^{11}	3.90×10^{11}	3.90×10^{11}	429	494	449.1	432
ps-10000-np-15-s-100	3.85×10^{11}	3.89×10^{11}	3.87×10^{11}	3.86×10^{11}	489	609	540.7	536
$k = 50$								
ps-5000-np-5-s-50	3.46×10^{11}	3.53×10^{11}	3.50×10^{11}	3.50×10^{11}	474	494	482.2	479
ps-5000-np-5-s-75	3.41×10^{11}	3.50×10^{11}	3.47×10^{11}	3.47×10^{11}	473	569	513.0	505
ps-5000-np-5-s-100	3.44×10^{11}	3.54×10^{11}	3.48×10^{11}	3.47×10^{11}	489	576	519.8	518
ps-5000-np-10-s-50	3.44×10^{11}	3.52×10^{11}	3.47×10^{11}	3.47×10^{11}	360	429	393.5	383
ps-5000-np-10-s-75	3.43×10^{11}	3.49×10^{11}	3.46×10^{11}	3.48×10^{11}	400	442	417.2	428
ps-5000-np-10-s-100	3.39×10^{11}	3.43×10^{11}	3.41×10^{11}	3.42×10^{11}	444	499	471.0	494
ps-5000-np-15-s-50	3.41×10^{11}	3.46×10^{11}	3.43×10^{11}	3.44×10^{11}	356	431	392.0	419
ps-5000-np-15-s-75	3.34×10^{11}	3.41×10^{11}	3.38×10^{11}	3.40×10^{11}	397	435	417.3	432
ps-5000-np-15-s-100	3.32×10^{11}	3.40×10^{11}	3.36×10^{11}	3.39×10^{11}	423	502	460.6	484
ps-10000-np-5-s-50	3.35×10^{11}	3.42×10^{11}	3.39×10^{11}	3.41×10^{11}	562	652	597.8	627
ps-10000-np-5-s-75	3.33×10^{11}	3.39×10^{11}	3.36×10^{11}	3.38×10^{11}	609	642	624.2	639
ps-10000-np-5-s-100	3.28×10^{11}	3.32×10^{11}	3.30×10^{11}	3.31×10^{11}	579	661	615.3	645
ps-10000-np-10-s-50	3.32×10^{11}	3.38×10^{11}	3.35×10^{11}	3.37×10^{11}	447	574	509.1	560
ps-10000-np-10-s-75	3.30×10^{11}	3.32×10^{11}	3.31×10^{11}	3.32×10^{11}	435	488	459.2	474
ps-10000-np-10-s-100	3.28×10^{11}	3.30×10^{11}	3.29×10^{11}	3.30×10^{11}	516	607	546.1	576
ps-10000-np-15-s-50	3.36×10^{11}	3.48×10^{11}	3.43×10^{11}	3.47×10^{11}	431	452	437.7	444
ps-10000-np-15-s-75	3.31×10^{11}	3.42×10^{11}	3.36×10^{11}	3.40×10^{11}	431	508	479.7	506

Table 6: Piecy-mr on `Caltech128`, experiments belong to class I.

	min	max	average	median	min	max	avg	med
ps-10000-np-15-s-100	3.29×10^{11}	3.40×10^{11}	3.34×10^{11}	3.39×10^{11}	489	553	520.9	546
$k = 100$								
ps-5000-np-5-s-50	3.28×10^{11}	3.37×10^{11}	3.32×10^{11}	3.35×10^{11}	435	531	487.3	513
ps-5000-np-5-s-75	3.20×10^{11}	3.30×10^{11}	3.26×10^{11}	3.29×10^{11}	470	545	505.6	537
ps-5000-np-5-s-100	3.22×10^{11}	3.31×10^{11}	3.26×10^{11}	3.29×10^{11}	524	586	553.9	575
ps-5000-np-10-s-50	3.27×10^{11}	3.32×10^{11}	3.30×10^{11}	3.32×10^{11}	372	440	397.4	421
ps-5000-np-10-s-75	3.23×10^{11}	3.26×10^{11}	3.24×10^{11}	3.25×10^{11}	414	451	429.6	442
ps-5000-np-10-s-100	3.17×10^{11}	3.22×10^{11}	3.21×10^{11}	3.22×10^{11}	427	507	471.3	498
ps-5000-np-15-s-50	3.21×10^{11}	3.29×10^{11}	3.24×10^{11}	3.27×10^{11}	359	423	395.0	415
ps-5000-np-15-s-75	3.20×10^{11}	3.26×10^{11}	3.22×10^{11}	3.24×10^{11}	409	454	424.7	439
ps-5000-np-15-s-100	3.16×10^{11}	3.22×10^{11}	3.19×10^{11}	3.21×10^{11}	420	471	447.8	469
ps-10000-np-5-s-50	3.18×10^{11}	3.25×10^{11}	3.21×10^{11}	3.24×10^{11}	551	730	619.4	681
ps-10000-np-5-s-75	3.15×10^{11}	3.19×10^{11}	3.17×10^{11}	3.18×10^{11}	594	705	646.7	686
ps-10000-np-5-s-100	3.08×10^{11}	3.14×10^{11}	3.11×10^{11}	3.13×10^{11}	552	679	625.6	663
ps-10000-np-10-s-50	3.15×10^{11}	3.18×10^{11}	3.16×10^{11}	3.17×10^{11}	427	482	457.6	475
ps-10000-np-10-s-75	3.12×10^{11}	3.14×10^{11}	3.13×10^{11}	3.14×10^{11}	458	538	490.4	515
ps-10000-np-10-s-100	3.07×10^{11}	3.09×10^{11}	3.08×10^{11}	3.09×10^{11}	537	636	582.4	608
ps-10000-np-15-s-50	3.20×10^{11}	3.23×10^{11}	3.21×10^{11}	3.22×10^{11}	387	492	433.0	468
ps-10000-np-15-s-75	3.15×10^{11}	3.21×10^{11}	3.17×10^{11}	3.19×10^{11}	455	503	475.5	491
ps-10000-np-15-s-100	3.09×10^{11}	3.23×10^{11}	3.17×10^{11}	3.22×10^{11}	488	594	533.4	572

Table 6: Piecy-mr on `Caltech128` (continued), experiments belong to class I.

Group	Cost				Running time			
	min	max	average	median	min	max	avg	med
$k = 5$								
n-5000-d-1000								
k-10-ps-2000-svd-10	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	391	1018	703.4	644
k-10-ps-2000-svd-20	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	442	1155	767.6	736
k-10-ps-2000-svd-50	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	348	1018	554.4	489
k-10-ps-2000-svd-70	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	334	823	534.6	508
k-20-ps-4000-svd-10	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	439	948	698.2	676
k-20-ps-4000-svd-20	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	574	1670	937.9	834
k-20-ps-4000-svd-50	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	417	1049	720.7	675
k-20-ps-4000-svd-70	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	443	934	606.0	553
k-50-ps-10000-svd-10	1.69×10^9	1.69×10^9	1.69×10^9	1.69×10^9	402	647	478.2	461
k-50-ps-10000-svd-20	1.69×10^9	1.70×10^9	1.70×10^9	1.70×10^9	646	2021	996.5	978
k-50-ps-10000-svd-50	1.69×10^9	1.70×10^9	1.70×10^9	1.70×10^9	612	4183	1823.8	1741
k-50-ps-10000-svd-70	1.69×10^9	1.70×10^9	1.70×10^9	1.70×10^9	510	1968	1246.0	1206
k-100-ps-20000-svd-10	1.69×10^9	1.69×10^9	1.69×10^9	1.69×10^9	341	662	459.6	437
k-100-ps-20000-svd-20	1.69×10^9	1.69×10^9	1.69×10^9	1.69×10^9	473	765	593.8	595
k-100-ps-20000-svd-50	1.69×10^9	1.69×10^9	1.69×10^9	1.69×10^9	719	3507	1597.3	1528
k-100-ps-20000-svd-70	1.69×10^9	1.69×10^9	1.69×10^9	1.69×10^9	704	5198	1654.6	1340
n-10000-d-500								
k-10-ps-2000-svd-10	3.35×10^9	3.36×10^9	3.36×10^9	3.36×10^9	453	1130	699.0	565
k-10-ps-2000-svd-20	3.35×10^9	3.37×10^9	3.36×10^9	3.36×10^9	475	1237	734.9	694
k-10-ps-2000-svd-50	3.36×10^9	3.37×10^9	3.36×10^9	3.36×10^9	342	818	528.3	554
k-10-ps-2000-svd-70	3.36×10^9	3.37×10^9	3.36×10^9	3.36×10^9	347	1069	526.9	480
k-20-ps-4000-svd-10	3.34×10^9	3.35×10^9	3.35×10^9	3.35×10^9	615	1298	993.0	1023
k-20-ps-4000-svd-20	3.34×10^9	3.36×10^9	3.35×10^9	3.35×10^9	549	2391	1425.2	1315
k-20-ps-4000-svd-50	3.36×10^9	3.37×10^9	3.36×10^9	3.36×10^9	424	2032	1058.1	968
k-20-ps-4000-svd-70	3.35×10^9	3.37×10^9	3.36×10^9	3.36×10^9	400	1972	1027.7	1040
k-50-ps-10000-svd-10	3.33×10^9	3.33×10^9	3.33×10^9	3.33×10^9	484	1496	785.7	724
k-50-ps-10000-svd-20	3.33×10^9	3.35×10^9	3.34×10^9	3.34×10^9	747	2996	1567.3	1469
k-50-ps-10000-svd-50	3.34×10^9	3.37×10^9	3.35×10^9	3.35×10^9	895	4015	2259.5	2359
k-50-ps-10000-svd-70	3.34×10^9	3.37×10^9	3.35×10^9	3.35×10^9	560	4079	2100.6	2244
k-100-ps-20000-svd-10	3.32×10^9	3.32×10^9	3.32×10^9	3.32×10^9	390	579	483.4	494
k-100-ps-20000-svd-20	3.32×10^9	3.32×10^9	3.32×10^9	3.32×10^9	515	1876	1035.3	971
k-100-ps-20000-svd-50	3.32×10^9	3.33×10^9	3.33×10^9	3.33×10^9	803	8613	2701.5	2205
k-100-ps-20000-svd-70	3.32×10^9	3.37×10^9	3.33×10^9	3.33×10^9	1161	6735	3093.9	2126
n-10000-d-1000								

Table 7: Piecy on `StructuredWithNoise`, experiments belong to class I.

	min	max	average	median	min	max	avg	med
k-10-ps-2000-svd-10	3.40×10^9	3.41×10^9	3.40×10^9	3.40×10^9	912	2544	1728.3	1777
k-10-ps-2000-svd-20	3.40×10^9	3.41×10^9	3.41×10^9	3.41×10^9	862	2752	1703.0	1575
k-10-ps-2000-svd-50	3.40×10^9	3.41×10^9	3.41×10^9	3.41×10^9	661	1429	931.0	847
k-10-ps-2000-svd-70	3.41×10^9	3.41×10^9	3.41×10^9	3.41×10^9	686	1852	1062.7	1035
k-20-ps-4000-svd-10	3.40×10^9	3.40×10^9	3.40×10^9	3.40×10^9	1146	3167	2145.3	2193
k-20-ps-4000-svd-20	3.40×10^9	3.40×10^9	3.40×10^9	3.40×10^9	1164	4652	2434.6	2194
k-20-ps-4000-svd-50	3.40×10^9	3.41×10^9	3.41×10^9	3.41×10^9	770	3011	1744.1	1965
k-20-ps-4000-svd-70	3.40×10^9	3.41×10^9	3.41×10^9	3.41×10^9	741	1648	1100.2	998
k-50-ps-10000-svd-10	3.39×10^9	3.39×10^9	3.39×10^9	3.39×10^9	964	2450	1521.5	1503
k-50-ps-10000-svd-20	3.39×10^9	3.40×10^9	3.40×10^9	3.40×10^9	1967	5791	2997.4	2754
k-50-ps-10000-svd-50	3.39×10^9	3.41×10^9	3.40×10^9	3.40×10^9	1618	9826	4041.0	3011
k-50-ps-10000-svd-70	3.40×10^9	3.41×10^9	3.40×10^9	3.40×10^9	1302	8497	4512.1	4474
k-100-ps-20000-svd-10	3.38×10^9	3.38×10^9	3.38×10^9	3.38×10^9	790	1350	954.3	913
k-100-ps-20000-svd-20	3.38×10^9	3.38×10^9	3.38×10^9	3.38×10^9	1056	5533	2407.1	2209
k-100-ps-20000-svd-50	3.38×10^9	3.39×10^9	3.39×10^9	3.38×10^9	1692	12 388	5438.9	4327
k-100-ps-20000-svd-70	3.38×10^9	3.39×10^9	3.39×10^9	3.39×10^9	1311	13 853	3896.1	2904

Table 7: Piecy on `StructuredWithNoise` (continued), experiments belong to class I.

Group	Cost				Running time			
	min	max	average	median	min	max	avg	med
$k = 10$, piece size 2000								
np-10-svd-10	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	768	863	817.4	820
np-10-svd-20	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	902	953	917.4	906
np-10-svd-50	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	873	925	902.1	906
np-10-svd-75	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	907	959	933.4	942
np-15-svd-10	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	819	902	879.8	894
np-15-svd-20	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	880	949	905.7	895
np-15-svd-50	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	870	912	887.1	882
np-15-svd-75	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	867	953	897.7	888
np-50-svd-10	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	929	1064	997.7	997
np-50-svd-20	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	989	1094	1023.9	999
np-50-svd-50	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	954	1094	1004.0	980
np-50-svd-75	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	1032	1188	1107.9	1120
$k = 20$, piece size 4000								
np-10-svd-50	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	974	1222	1136.4	1186
np-10-svd-75	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	1074	1277	1181.9	1160
np-15-svd-10	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	862	999	919.0	887
np-15-svd-20	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	1106	1202	1158.3	1170
np-15-svd-50	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	1014	1249	1121.1	1105
np-15-svd-75	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	1118	1262	1177.1	1164
np-50-svd-10	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	1238	1666	1408.0	1383
np-50-svd-20	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	3.32×10^{10}	1033	1761	1338.5	1225
np-50-svd-50	3.32×10^{10}	3.33×10^{10}	3.32×10^{10}	3.32×10^{10}	1500	1805	1638.1	1650
np-50-svd-75	3.32×10^{10}	3.33×10^{10}	3.33×10^{10}	3.33×10^{10}	1256	1843	1450.8	1358

Table 8: Piecy-mr on `random` instances with $d = 1000$ and $n = 10^6$ (continued), experiments belong to class II.

Group	Cost				Running time			
	min	max	average	median	min	max	avg	med
Group	Min cost	Max cost	Avg cost	Median cost	Min time	Max time	Avg time	Median time
svd-15	5.91×10^7	5.91×10^7	5.91×10^7	5.91×10^7	58	68	62.9	63
svd-20	5.62×10^7	5.62×10^7	5.62×10^7	5.62×10^7	62	89	70.0	66

Table 9: Piecy-mr on `LowerBound` instances with $d = 10010$, $n = 10000$ and $m, k = 10$. The piece size is fixed to 2000, the number of pieces is fixed to 10. Experiments belong to class II.

Group	Cost				Running time			
	min	max	average	median	min	max	avg	med
ps-10000-np-5-svd-75	3.34×10^9	3.35×10^9	3.34×10^9	3.34×10^9	663	1086	843.8	829
ps-10000-np-10-svd-75	3.34×10^9	3.35×10^9	3.34×10^9	3.34×10^9	453	769	632.2	629
ps-10000-np-10-svd-50	3.33×10^9	3.34×10^9	3.34×10^9	3.34×10^9	426	873	627.0	609

Table 10: Piecy-MR on `StructuredWithNoise` with $n = 1000000$, $d = 500$ and $k = 50$. Experiments belong to class III.

Group	Cost				Running time			
	min	max	average	median	min	max	avg	med
n-5000-d-1000								
k-10-ps-2000-np-10-svd-10	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	346	397	375.9	382
k-10-ps-2000-np-10-svd-20	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	401	456	422.4	422
k-10-ps-2000-np-10-svd-50	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	410	482	434.3	430
k-10-ps-2000-np-10-svd-70	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	431	495	458.3	463
k-10-ps-2000-np-15-svd-10	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	365	450	397.0	396
k-10-ps-2000-np-15-svd-20	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	399	443	423.3	431
k-10-ps-2000-np-15-svd-50	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	381	452	408.6	407
k-10-ps-2000-np-15-svd-70	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	399	454	422.0	427
k-10-ps-2000-np-50-svd-10	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	369	540	441.2	437
k-10-ps-2000-np-50-svd-20	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	394	517	458.2	487
k-10-ps-2000-np-50-svd-50	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	386	519	449.7	458
k-10-ps-2000-np-50-svd-70	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	379	529	446.2	460
k-20-ps-4000-np-10-svd-10	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	344	410	365.0	365
k-20-ps-4000-np-10-svd-20	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	365	540	449.2	464
k-20-ps-4000-np-10-svd-50	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	438	559	506.3	519
k-20-ps-4000-np-10-svd-70	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	494	585	536.1	545
k-20-ps-4000-np-15-svd-10	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	343	430	382.4	387
k-20-ps-4000-np-15-svd-20	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	420	513	473.2	482
k-20-ps-4000-np-15-svd-50	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	479	575	512.5	512
k-20-ps-4000-np-15-svd-70	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	475	532	502.0	508
k-20-ps-4000-np-50-svd-10	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	457	629	518.4	529
k-20-ps-4000-np-50-svd-20	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	416	719	518.2	518
k-20-ps-4000-np-50-svd-50	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	387	648	462.4	459
k-20-ps-4000-np-50-svd-70	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	386	626	499.3	531
k-50-ps-10000-np-10-svd-10	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	369	546	425.5	420
k-50-ps-10000-np-10-svd-20	1.69×10^9	1.70×10^9	1.70×10^9	1.70×10^9	502	934	643.9	642
k-50-ps-10000-np-10-svd-50	1.69×10^9	1.70×10^9	1.69×10^9	1.69×10^9	558	1112	772.5	769
k-50-ps-10000-np-10-svd-70	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	571	883	706.7	713
k-50-ps-10000-np-15-svd-10	1.69×10^9	1.70×10^9	1.69×10^9	1.69×10^9	371	508	432.8	449
k-50-ps-10000-np-15-svd-20	1.69×10^9	1.70×10^9	1.69×10^9	1.69×10^9	595	1108	779.6	811
k-50-ps-10000-np-15-svd-50	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	568	1195	758.1	750
k-50-ps-10000-np-15-svd-70	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	647	1345	998.4	1074
k-50-ps-10000-np-50-svd-10	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	417	1671	1035.1	1077
k-50-ps-10000-np-50-svd-20	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	510	1723	1235.6	1412
k-50-ps-10000-np-50-svd-50	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	556	2039	1172.7	1322
k-50-ps-10000-np-50-svd-70	1.70×10^9	1.70×10^9	1.70×10^9	1.70×10^9	540	3294	1575.1	1484
k-100-ps-20000-np-10-svd-10	1.69×10^9	1.69×10^9	1.69×10^9	1.69×10^9	339	480	416.3	425
k-100-ps-20000-np-10-svd-20	1.69×10^9	1.69×10^9	1.69×10^9	1.69×10^9	444	947	703.7	736
k-100-ps-20000-np-10-svd-50	1.69×10^9	1.69×10^9	1.69×10^9	1.69×10^9	690	1933	1217.5	1234
k-100-ps-20000-np-10-svd-70	1.69×10^9	1.70×10^9	1.69×10^9	1.69×10^9	1084	1751	1433.1	1539
k-100-ps-20000-np-15-svd-10	1.69×10^9	1.69×10^9	1.69×10^9	1.69×10^9	356	550	461.5	476
k-100-ps-20000-np-15-svd-20	1.69×10^9	1.69×10^9	1.69×10^9	1.69×10^9	728	1392	1064.5	1108
k-100-ps-20000-np-15-svd-50	1.69×10^9	1.69×10^9	1.69×10^9	1.69×10^9	812	2401	1488.3	1571
k-100-ps-20000-np-15-svd-70	1.69×10^9	1.70×10^9	1.69×10^9	1.69×10^9	954	2004	1719.4	1907
k-100-ps-20000-np-50-svd-10	1.69×10^9	1.69×10^9	1.69×10^9	1.69×10^9	400	759	529.2	475
k-100-ps-20000-np-50-svd-20	1.69×10^9	1.69×10^9	1.69×10^9	1.69×10^9	750	1805	1262.4	1346
k-100-ps-20000-np-50-svd-50	1.69×10^9	1.69×10^9	1.69×10^9	1.69×10^9	525	2593	1482.6	1874
k-100-ps-20000-np-50-svd-70	1.69×10^9	1.70×10^9	1.69×10^9	1.69×10^9	393	4402	2053.8	2213
n-10000-d-500								
k-10-ps-2000-np-10-svd-10	3.36×10^9	3.36×10^9	3.36×10^9	3.36×10^9	357	393	374.5	375
k-10-ps-2000-np-10-svd-20	3.35×10^9	3.36×10^9	3.36×10^9	3.36×10^9	407	429	415.1	418
k-10-ps-2000-np-10-svd-50	3.35×10^9	3.36×10^9	3.36×10^9	3.36×10^9	410	468	436.9	441

Table 11: Piecy-MR on `StructuredWithNoise`. Experiments belong to class II.

	min	max	average	median	min	max	avg	med
k-10-ps-2000-np-10-svd-70	3.36×10^9	3.36×10^9	3.36×10^9	3.36×10^9	419	477	449.1	448
k-10-ps-2000-np-15-svd-10	3.35×10^9	3.36×10^9	3.35×10^9	3.35×10^9	366	421	398.4	407
k-10-ps-2000-np-15-svd-20	3.35×10^9	3.36×10^9	3.36×10^9	3.36×10^9	377	436	401.0	403
k-10-ps-2000-np-15-svd-50	3.36×10^9	3.36×10^9	3.36×10^9	3.36×10^9	404	449	426.8	433
k-10-ps-2000-np-15-svd-70	3.36×10^9	3.36×10^9	3.36×10^9	3.36×10^9	427	469	446.7	450
k-10-ps-2000-np-50-svd-10	3.36×10^9	3.36×10^9	3.36×10^9	3.36×10^9	392	493	437.8	444
k-10-ps-2000-np-50-svd-20	3.35×10^9	3.36×10^9	3.36×10^9	3.36×10^9	388	501	440.8	457
k-10-ps-2000-np-50-svd-50	3.36×10^9	3.36×10^9	3.36×10^9	3.36×10^9	419	547	475.1	478
k-10-ps-2000-np-50-svd-70	3.36×10^9	3.37×10^9	3.36×10^9	3.36×10^9	427	534	490.7	512
k-20-ps-4000-np-10-svd-10	3.35×10^9	3.35×10^9	3.35×10^9	3.35×10^9	351	397	367.1	370
k-20-ps-4000-np-10-svd-20	3.34×10^9	3.35×10^9	3.35×10^9	3.35×10^9	442	538	480.6	490
k-20-ps-4000-np-10-svd-50	3.35×10^9	3.36×10^9	3.35×10^9	3.36×10^9	479	572	528.8	543
k-20-ps-4000-np-10-svd-70	3.35×10^9	3.36×10^9	3.36×10^9	3.36×10^9	540	608	564.4	568
k-20-ps-4000-np-15-svd-10	3.34×10^9	3.35×10^9	3.34×10^9	3.34×10^9	378	464	423.7	431
k-20-ps-4000-np-15-svd-20	3.35×10^9	3.35×10^9	3.35×10^9	3.35×10^9	473	535	504.0	508
k-20-ps-4000-np-15-svd-50	3.35×10^9	3.36×10^9	3.36×10^9	3.36×10^9	428	541	489.8	499
k-20-ps-4000-np-15-svd-70	3.36×10^9	3.36×10^9	3.36×10^9	3.36×10^9	462	595	539.8	573
k-20-ps-4000-np-50-svd-10	3.35×10^9	3.35×10^9	3.35×10^9	3.35×10^9	463	690	561.3	561
k-20-ps-4000-np-50-svd-20	3.35×10^9	3.35×10^9	3.35×10^9	3.35×10^9	395	698	566.8	620
k-20-ps-4000-np-50-svd-50	3.35×10^9	3.36×10^9	3.35×10^9	3.35×10^9	520	959	703.2	725
k-20-ps-4000-np-50-svd-70	3.35×10^9	3.36×10^9	3.36×10^9	3.36×10^9	567	960	698.1	686
k-50-ps-10000-np-10-svd-10	3.33×10^9	3.35×10^9	3.34×10^9	3.35×10^9	353	502	407.5	398
k-50-ps-10000-np-10-svd-20	3.33×10^9	3.35×10^9	3.34×10^9	3.34×10^9	528	922	663.0	644
k-50-ps-10000-np-10-svd-50	3.33×10^9	3.34×10^9	3.34×10^9	3.34×10^9	625	1158	815.1	788
k-50-ps-10000-np-10-svd-70	3.34×10^9	3.35×10^9	3.34×10^9	3.34×10^9	641	1116	905.3	948
k-50-ps-10000-np-15-svd-10	3.33×10^9	3.33×10^9	3.33×10^9	3.33×10^9	389	503	434.6	430
k-50-ps-10000-np-15-svd-20	3.33×10^9	3.34×10^9	3.33×10^9	3.33×10^9	544	956	692.6	726
k-50-ps-10000-np-15-svd-50	3.34×10^9	3.35×10^9	3.34×10^9	3.35×10^9	747	1510	1075.8	1099
k-50-ps-10000-np-15-svd-70	3.34×10^9	3.36×10^9	3.35×10^9	3.35×10^9	854	1487	1029.0	985
k-50-ps-10000-np-50-svd-10	3.34×10^9	3.34×10^9	3.34×10^9	3.34×10^9	583	899	706.5	737
k-50-ps-10000-np-50-svd-20	3.33×10^9	3.34×10^9	3.34×10^9	3.34×10^9	685	1797	1223.7	1346
k-50-ps-10000-np-50-svd-50	3.34×10^9	3.35×10^9	3.34×10^9	3.34×10^9	694	3635	1562.7	1628
k-50-ps-10000-np-50-svd-70	3.34×10^9	3.35×10^9	3.35×10^9	3.35×10^9	677	2375	1499.1	1566
k-100-ps-20000-np-10-svd-10	3.32×10^9	3.33×10^9	3.33×10^9	3.33×10^9	374	617	445.7	440
k-100-ps-20000-np-10-svd-20	3.33×10^9	3.33×10^9	3.33×10^9	3.33×10^9	520	860	682.1	726
k-100-ps-20000-np-10-svd-50	3.32×10^9	3.32×10^9	3.32×10^9	3.32×10^9	706	1835	1175.4	1358
k-100-ps-20000-np-10-svd-70	3.32×10^9	3.33×10^9	3.33×10^9	3.33×10^9	740	2379	1199.0	1185
k-100-ps-20000-np-15-svd-10	3.32×10^9	3.32×10^9	3.32×10^9	3.32×10^9	393	596	483.4	501
k-100-ps-20000-np-15-svd-20	3.32×10^9	3.32×10^9	3.32×10^9	3.32×10^9	454	1138	815.9	863
k-100-ps-20000-np-15-svd-50	3.32×10^9	3.34×10^9	3.33×10^9	3.33×10^9	706	2377	1229.5	1093
k-100-ps-20000-np-15-svd-70	3.33×10^9	3.34×10^9	3.33×10^9	3.33×10^9	1066	2000	1490.9	1559
k-100-ps-20000-np-50-svd-10	3.34×10^9	3.34×10^9	3.34×10^9	3.34×10^9	431	1816	938.0	1007
k-100-ps-20000-np-50-svd-20	3.33×10^9	3.33×10^9	3.33×10^9	3.33×10^9	746	3256	1520.7	1363
k-100-ps-20000-np-50-svd-50	3.33×10^9	3.33×10^9	3.33×10^9	3.33×10^9	485	4337	1847.7	1414
k-100-ps-20000-np-50-svd-70	3.33×10^9	3.34×10^9	3.33×10^9	3.34×10^9	779	3507	1895.5	1815
n-10000-d-1000								
k-10-ps-2000-np-10-svd-10	3.40×10^9	3.41×10^9	3.40×10^9	3.41×10^9	733	816	773.4	784
k-10-ps-2000-np-10-svd-20	3.40×10^9	3.41×10^9	3.40×10^9	3.40×10^9	805	902	846.6	847
k-10-ps-2000-np-10-svd-50	3.40×10^9	3.41×10^9	3.41×10^9	3.41×10^9	831	934	879.2	885
k-10-ps-2000-np-10-svd-70	3.40×10^9	3.41×10^9	3.41×10^9	3.41×10^9	840	950	883.7	884
k-10-ps-2000-np-15-svd-10	3.40×10^9	3.41×10^9	3.40×10^9	3.40×10^9	740	901	811.2	823
k-10-ps-2000-np-15-svd-20	3.40×10^9	3.40×10^9	3.40×10^9	3.40×10^9	800	879	845.1	861
k-10-ps-2000-np-15-svd-50	3.41×10^9	3.41×10^9	3.41×10^9	3.41×10^9	803	921	853.3	858
k-10-ps-2000-np-15-svd-70	3.41×10^9	3.41×10^9	3.41×10^9	3.41×10^9	855	941	894.7	903
k-10-ps-2000-np-50-svd-10	3.40×10^9	3.41×10^9	3.40×10^9	3.40×10^9	895	1082	952.4	948
k-10-ps-2000-np-50-svd-20	3.40×10^9	3.41×10^9	3.40×10^9	3.40×10^9	895	1101	963.4	948
k-10-ps-2000-np-50-svd-50	3.40×10^9	3.41×10^9	3.41×10^9	3.41×10^9	967	1095	1022.6	1028
k-10-ps-2000-np-50-svd-70	3.40×10^9	3.41×10^9	3.41×10^9	3.41×10^9	894	1163	1019.4	1028
k-20-ps-4000-np-10-svd-10	3.40×10^9	3.40×10^9	3.40×10^9	3.40×10^9	719	817	753.9	755
k-20-ps-4000-np-10-svd-20	3.40×10^9	3.40×10^9	3.40×10^9	3.40×10^9	940	1048	1000.9	1026
k-20-ps-4000-np-10-svd-50	3.40×10^9	3.41×10^9	3.40×10^9	3.40×10^9	968	1150	1059.6	1073
k-20-ps-4000-np-10-svd-70	3.40×10^9	3.41×10^9	3.41×10^9	3.41×10^9	1052	1140	1099.6	1116
k-20-ps-4000-np-15-svd-10	3.40×10^9	3.40×10^9	3.40×10^9	3.40×10^9	742	914	808.4	820
k-20-ps-4000-np-15-svd-20	3.40×10^9	3.40×10^9	3.40×10^9	3.40×10^9	915	1180	1006.0	984
k-20-ps-4000-np-15-svd-50	3.40×10^9	3.41×10^9	3.40×10^9	3.40×10^9	983	1118	1028.5	1012
k-20-ps-4000-np-15-svd-70	3.40×10^9	3.41×10^9	3.41×10^9	3.41×10^9	1005	1125	1055.5	1059
k-20-ps-4000-np-50-svd-10	3.40×10^9	3.40×10^9	3.40×10^9	3.40×10^9	1000	1280	1093.6	1070
k-20-ps-4000-np-50-svd-20	3.40×10^9	3.40×10^9	3.40×10^9	3.40×10^9	1079	1702	1253.9	1203

Table 11: Piecy-MR on `StructuredWithNoise`. Experiments belong to class II.

	min	max	average	median	min	max	avg	med
k-20-ps-4000-np-50-svd-50	3.40×10^9	3.40×10^9	3.40×10^9	3.40×10^9	899	1692	1308.5	1334
k-20-ps-4000-np-50-svd-70	3.40×10^9	3.41×10^9	3.40×10^9	3.40×10^9	971	1429	1177.9	1181
k-50-ps-10000-np-10-svd-10	3.39×10^9	3.40×10^9	3.39×10^9	3.39×10^9	732	907	834.7	844
k-50-ps-10000-np-10-svd-20	3.39×10^9	3.40×10^9	3.40×10^9	3.40×10^9	1053	1862	1403.7	1415
k-50-ps-10000-np-10-svd-50	3.39×10^9	3.40×10^9	3.39×10^9	3.39×10^9	1213	1936	1594.9	1586
k-50-ps-10000-np-10-svd-70	3.40×10^9	3.40×10^9	3.40×10^9	3.40×10^9	1323	2073	1600.7	1578
k-50-ps-10000-np-15-svd-10	3.39×10^9	3.39×10^9	3.39×10^9	3.39×10^9	791	1140	907.5	851
k-50-ps-10000-np-15-svd-20	3.39×10^9	3.39×10^9	3.39×10^9	3.39×10^9	1240	1735	1487.8	1448
k-50-ps-10000-np-15-svd-50	3.40×10^9	3.40×10^9	3.40×10^9	3.40×10^9	1328	2359	1830.6	1794
k-50-ps-10000-np-15-svd-70	3.40×10^9	3.40×10^9	3.40×10^9	3.40×10^9	1299	2310	1778.2	1786
k-50-ps-10000-np-50-svd-10	3.39×10^9	3.40×10^9	3.39×10^9	3.39×10^9	1128	3150	1898.7	1840
k-50-ps-10000-np-50-svd-20	3.39×10^9	3.39×10^9	3.39×10^9	3.39×10^9	1578	3391	2328.9	1988
k-50-ps-10000-np-50-svd-50	3.40×10^9	3.40×10^9	3.40×10^9	3.40×10^9	1105	3595	2788.2	3003
k-50-ps-10000-np-50-svd-70	3.40×10^9	3.40×10^9	3.40×10^9	3.40×10^9	1405	4388	2864.5	3102
k-100-ps-20000-np-10-svd-10	3.39×10^9	3.39×10^9	3.39×10^9	3.39×10^9	787	1526	1006.0	848
k-100-ps-20000-np-10-svd-20	3.38×10^9	3.39×10^9	3.39×10^9	3.39×10^9	1137	1886	1352.1	1303
k-100-ps-20000-np-10-svd-50	3.38×10^9	3.39×10^9	3.38×10^9	3.38×10^9	1368	3142	2028.2	1850
k-100-ps-20000-np-10-svd-70	3.38×10^9	3.39×10^9	3.39×10^9	3.39×10^9	1510	4033	2360.9	2084
k-100-ps-20000-np-15-svd-10	3.38×10^9	3.39×10^9	3.38×10^9	3.38×10^9	741	1462	908.1	826
k-100-ps-20000-np-15-svd-20	3.38×10^9	3.38×10^9	3.38×10^9	3.38×10^9	1119	1999	1566.5	1591
k-100-ps-20000-np-15-svd-50	3.38×10^9	3.39×10^9	3.39×10^9	3.39×10^9	1349	4218	2251.3	2050
k-100-ps-20000-np-15-svd-70	3.39×10^9	3.40×10^9	3.39×10^9	3.39×10^9	1631	3786	2544.3	2344
k-100-ps-20000-np-50-svd-10	3.39×10^9	3.40×10^9	3.39×10^9	3.39×10^9	1030	2222	1583.6	1697
k-100-ps-20000-np-50-svd-20	3.39×10^9	3.39×10^9	3.39×10^9	3.39×10^9	1301	6148	3452.1	3635
k-100-ps-20000-np-50-svd-50	3.39×10^9	3.40×10^9	3.40×10^9	3.39×10^9	2311	10 506	5589.8	4366
k-100-ps-20000-np-50-svd-70	3.39×10^9	3.40×10^9	3.40×10^9	3.40×10^9	912	6943	2344.5	1307

Table 11: Piecy-MR on `StructuredWithNoise`. Experiments belong to class II.

Conclusion.

The experiments show the potential speed-up by using piecy and piecy-mr. When choosing the algorithm, one should take the dimensions of the input matrix into account. For large dimension but a moderate number of points, piecy is ideal since it reduces the dimension effectively with little overhead. For data sets where the dimension is high and the number of points is also high, the additional overhead of piecy-mr pays off.

Acknowledgements.

We thank Cameron Musco and Chris Schwiegelshohn for insightful discussions on the topic of this paper, Hendrik Fichtenberger and Lukas Pradel for sharing some pieces of source code and Jan Stallmann and René Grzeszick for providing the `LowerBound` and `CalTech` data sets.

References

[1] Marcel R. Ackermann, Marcus Märtens, Christoph Raupach, Kamil Swierkot, Christiane Lammersen, and Christian Sohler. Streamkm++: A clustering algorithm for data streams. *ACM Journal of Experimental Algorithmics*, 17:article 2.4, 1–30, 2012.

[2] Pankaj K. Agarwal, Sariel Har-Peled, and Kasturi R. Varadarajan. Approximating extent measures of points. *Journal of the ACM*, 51(4):606 – 635, 2004.

[3] David Arthur and Sergei Vassilvitskii. k-means++: the advantages of careful seeding. In *Proc. of the 18th SODA*, pages 1027 – 1035, 2007.

[4] Jon L. Bentley and James B. Saxe. Decomposable searching problems i: Static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301 – 358, 1980.

[5] Michael B. Cohen, Sam Elder, Cameron Musco, Christopher Musco, and Madalina Persu. Dimensionality reduction for k-means clustering and low rank approximation. In *Proc. of the 47th STOC*, 2015. To appear.

[6] Petros Drineas, Alan M. Frieze, Ravi Kannan, Santosh Vempala, and V. Vinay. Clustering large graphs via the singular value decomposition. *Machine Learning*, 56:9–33, 2004.

[7] Li Fei-Fei, Rob Fergus, and Pietro Perona. Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. In *CVPR 2004, Workshop on Generative-Model Based Vision*. IEEE, 2004.

[8] Dan Feldman and Michael Langberg. A unified framework for approximating and clustering data. In *Proc. of the 43th STOC*, pages 569 – 578, 2011.

[9] Dan Feldman, Melanie Schmidt, and Christian Sohler. Turning Big Data into Tiny Data: Constant-size Coresets for k-means, PCA and Projective Clustering. In *Proc. of the 24th SODA*, pages 1434 – 1453, 2013.

[10] Hendrik Fichtenberger, Marc Gillé, Melanie Schmidt, Chris Schwiegelshohn, and Christian Sohler. BICO: BIRCH Meets Coresets for k-Means Clustering . In *Proc. of the 21st ESA*, pages 481–492, 2013.

[11] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review (SIREV)*, 53(2):217 – 288, 2011.

[12] Sariel Har-Peled and Soham Mazumdar. On coresets for k-means and k-median clustering. In *Proc. of the 36th STOC*, pages 291 – 300, 2004.

[13] Anil K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 31(8):651 – 666, 2010.

[14] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[15] Kamal Jain and Vijay V. Vazirani. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and lagrangian relaxation. *Journal of the ACM*, 48(2):274 – 296, 2001.

[16] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. A local search approximation algorithm for k-means clustering. *Computational Geometry*, 28(2-3):89 – 112, June 2004.

[17] Stuart P. Lloyd. Least squares quantization in PCM. *Bell Laboratories Technical Memorandum*, 1957. later published as [18].

[18] Stuart P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129 – 137, 1982.

[19] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91 – 110, 2004.

[20] Michael W. Mahoney. Randomized algorithms for matrices and data. *Foundations and Trends in Machine Learning*, 3(2):123–224, 2011.

[21] Daisuke Okanohara. C++ project: redsvd – RandomizED Singular Value Decomposition. `https://code.google.com/p/redsvd/`, 2011. accessed: 2nd of February, 2015.

[22] Jan Stallmann. Benchmarkinstanzen für das k-means Problem. Bachelorarbeit, TU Dortmund University, 2014. In german.

[23] Hugo Steinhaus. Sur la division des corps matériels en parties. *Bulletin de l'Académie Polonaise des Sciences*, IV(12):801 – 804, 1956.

[24] Lapack++ Development Team. C++ library: Lapack++ v2.5.4. `http://sourceforge.net/projects/lapackpp/`, 2010. accessed: 8th of February, 2015.

[25] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus F. M. Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1 – 37, 2008.

[26] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: A New Data Clustering Algorithm and Its Applications . *Data Mining and Knowledge Discovery*, 1(2):141 – 182, 1997.