Fast Exact k-Means, k-Medians and Bregman Divergence Clustering in 1D

Allan Grønlund[*] Kasper Green Larsen[†] Alexander Mathiasen[‡]
Jesper Sindahl Nielsen [§] Stefan Schneider [¶] Mingzhou Song [||]

Abstract

The k-Means clustering problem on n points is NP-Hard for any dimension $d \geq 2$, however, for the 1D case there exists exact polynomial time algorithms. Previous literature reported an $O(kn^2)$ time dynamic programming algorithm that uses $O(kn)$ space. It turns out that the problem has been considered under a different name more than twenty years ago. We present all the existing work that had been overlooked and compare the various solutions theoretically. Moreover, we show how to reduce the space usage for some of them, as well as generalize them to data structures that can quickly report an optimal k-Means clustering for any k. Finally we also generalize all the algorithms to work for the absolute distance and to work for any Bregman Divergence. We complement our theoretical contributions by experiments that compare the practical performance of the various algorithms.

1 Introduction

Clustering is the problem of grouping elements into clusters such that each element is similar to the elements in the cluster assigned to it and not similar to elements in any other cluster. It is one of, if not the, primary problem in the area of machine learning known as Unsupervised Learning and no clustering problem is as famous and widely considered as the k-Means problem: Given a multiset $\mathcal{X} = \{x_1, ..., x_n\} \subset \mathbb{R}^d$ find k centroids $\mathcal{M} = \{\mu_1, ..., \mu_k\} \subset \mathbb{R}^d$ minimizing $\sum_{x \in \mathcal{X}} \min_{\mu \in \mathcal{M}} ||x - \mu||^2$. Several NP-Hardness results exist for finding the optimal k-Means clustering in general, forcing one to turn towards heuristics. k-Means is NP-hard even for $k = 2$ and general dimension [4] and it is also NP-hard for $d = 2$ and general k [17]. Even hardness of approximation results exist [16, 8]. In [8] the authors show there exists an $\varepsilon > 0$ such that it is NP-hard to approximate k-Means to within a factor $1 + \varepsilon$ of optimal, and in [16] they prove that $\varepsilon \geq 0.0013$. On the upper bound side the best known polynomial time approximation algorithm for k-Means has an approximation factor of 6.357 [3]. In practice, Lloyd's algorithm is a popular iterative local search heuristic that starts from some random or arbitrary clustering. The running time of Lloyd's algorithm is $O(tknd)$ where t is the number of rounds of the local search procedure. In theory, if Lloyd's algorithm is run to convergence to a local minimum, t could be exponential and there is no guarantee on how well the solution found approximates the optimal solution [6, 21]. Lloyd's algorithm is often combined with the effective seeding technique for selecting initial centroids due to [7] that gives an expected $O(\lg k)$ approximation ratio for the initial clustering, which can then be further improved by Lloyd's algorithm.

For the one-dimensional case, the k-Means problem is not NP-hard. In particular, there is an $O(kn^2)$ time and $O(kn)$ space dynamic programming solution for the 1D case, due to work by [23] giving focus to this simpler special

[*]Aarhus University. Email: `jallan@cs.au.dk`. Supported by MADALGO - Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

[†]Aarhus University. Email: `larsen@cs.au.dk`. Supported by MADALGO, a Villum Young Investigator Grant and an AUFF Starting Grant.

[‡]Aarhus University. Email: `alexander.mathiasen@gmail.com`. Supported by MADALGO and an AUFF Starting Grant.

[§]Aarhus University. Email: `jasn@cs.au.dk`. Supported by MADALGO.

[¶]University of California, San Diego. Email: `stschnei@cs.ucsd.edu`. Supported by NSF grant CCF-1213151 from the Division of Computing and Communication Foundations. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

[||]New Mexico State University. Email: `joemsong@cs.nmsu.edu` case of k-Means. The 1D k-Means problem is encountered surprisingly often in practice, some examples being in data analysis in social networks, bioinformatics and retail market [5, 14, 19]. Note that an optimal clustering in 1D simply covers the input points with non-overlapping intervals.

It is natural to try other reasonable distance measures for the data considered and define different clustering problems instead of using the sum of squares of the Euclidian distance that defines k-Means. For instance, one could use any L_p norm instead. The special case of $p = 1$ is known as k-Medians clustering and has also received considerable attention. The k-Medians problems is also NP-hard in dimensions 2 and up, and the best polynomial time approximation algorithm has an approximation factor of 2.633 [8].

In [9] the authors consider and define clustering with Bregman Divergences. Bregman Divergence generalizes squared Euclidian distance and thus Bregman Clusterings include the k-Means problem, as well as a wide range of other clustering problems that can be defined from Bregman Divergences like e.g. clustering with Kullback-Leibler divergence as the cost. Interestingly, the heuristic local search algorithm for Bregman Clustering [9] is basically the same approach as Lloyd's algorithm for k-Means. Clustering with Bregman Divergences is clearly NP-Hard as well since it includes k-Means clustering. We refer the reader to [9] for more about the general problem. For the 1D version of the problem, [18] generalized the algorithm from [23] to the k-Median problem and Bregman Divergences achieving the same $O(kn^2)$ time and $O(kn)$ space bounds.

1.1 Even Earlier Work

Unknown to both [23, 18], it turns out that the k-Means problem has been studied before under a different name. The paper [25] from 1980 considers a 1D data discrete quantization problem, where the problem is to quantize n (sorted) weighted input points to k representatives, and the goal is to compute the k representatives that minimizes the quantization error. The error measure for this quantization is weighted least squares, which is a simple generalization of the 1D k-Means cost. Formally, given n points $x_1, \ldots, x_n$ with weights $w_1, \ldots, w_n$, find centroids $\mathcal{M} = \{\mu_1, ..., \mu_k\} \subset \mathbb{R}$ minimizing the cost

$$\sum_{i=1}^{n} w_i \min_{\mu \in \mathcal{M}} (x_i - \mu)^2 \tag{1}$$

The algorithm given in [25] use $O(kn)$ time (and space). This was later improved to $O(k\sqrt{n \lg n})$ time and space in [1] that also introduces a simple $O(n \lg U)$ time and linear space algorithm where U is a *universe* size that depends on the input. Finally, the running time was reduced to $n2^{O(\sqrt{\lg \lg n \lg k})}$ by an algorithm of Schieber [20] in 1998 for the case $k = \Omega(\lg n)$. This algorithm uses linear space. This is, to the best of our knowledge, currently the best bound known for 1D k-Means.

1.2 Our Contribution

In this paper we present an overview of different 1D k-Means algorithms. We show how to reduce the space for the presented dynamic programming algorithm and how this leads to a simple data structure that uses $O(n)$ space and can report an optimal k-Means clustering for any k in $O(n)$ time. We generalize the fast algorithms for k-Means to all Bregman Divergences and the Absolute Distance cost function. Finally, we give a practical comparison of the algorithms and discuss the outcome. The main purpose of this paper is clarifying the current state of algorithms for 1D k-Means both in theory and practice, introducing a regularized version of 1D k-Means, and generalizing the results to more distance measures. All algorithms and proofs for k-Means generalize to handle the weighted version of the k-Means cost (Equation 1) considered in [25]. For simplicity, we present the algorithms only for the standard unweighted definition of the k-Means cost.

We start by giving the dynamic programming formulation [23, 25] that yields an $O(kn^2)$ time algorithm. We then describe how [25] improves this to an algorithm that runs in $O(n \lg n + kn)$, or $O(kn)$ time if the input is already sorted. Both algorithms compute the cost of the optimal clustering using k' clusters for all $k' \leq k$. This is relevant for instance for model selection of the right k. Second, we present the reduction from 1D k-Means to the shortest path graph problem considered in [20], that yields a 1D k-Means clustering algorithm that uses $n2^{O(\sqrt{\lg \lg n \lg k})}$ time for $k = \Omega(\lg n)$ and $O(n)$ space. In contrast to the $O(kn)$ time algorithm, this algorithm does not compute the optimal costs for using k' clusters for all $k' \leq k$. Finally, we present an algorithm using $O(n \lg U)$ time and linear space, where $\lg U$ is the number bits used to represent each input point.

For comparison, in 1D, Lloyd's algorithm takes $O(tk \lg n)$ time if the input is sorted, where t is the number of rounds before convergence. For reasonable values of k the time is usually sublinear and extremely fast. However Lloyd's algorithm does not necessarily compute the optimal clustering, it is only a heuristic. The algorithms considered in this paper compute the *optimal* clustering quickly, so even for large n and k we do not need to settle for approximation or uncertainty about the quality of the clustering.

The $n2^{O(\sqrt{\lg \lg n \lg k})}$ and the $O(n \lg U)$ time algorithms for 1D k-Means are based on a natural regularized version of k-Means clustering where instead of specifying the number of clusters beforehand, we instead specify a per cluster cost and then minimize the cost of the clustering plus the cost of the number of clusters used. Formally, the problem is as follows: Given $\mathcal{X} = \{x_1, ..., x_n\} \subset \mathbb{R}$ and a non-negative real number $\lambda \in \mathbb{R}^+$, compute the optimal regularized clustering:

$$\underset{k, \mathcal{M}=\{\mu_1,\ldots,\mu_k\}}{\arg\min} \sum_{x \in \mathcal{X}} \min_{\mu \in \mathcal{M}} (x-\mu)^2 + \lambda k$$

Somewhat surprisingly, it takes only $O(n)$ time to find the solution to the regularized k-Means if the input is sorted.

The k-Medians problem is to compute a clustering that minimizes the sum of absolute distances to the centroid, i.e. compute $\mathcal{M} = \{\mu_1, ..., \mu_k\} \subset \mathbb{R}$ that minimizes

$$\sum_{x \in \mathcal{X}} \min_{\mu \in \mathcal{M}} |x - \mu|$$

We show that the k-Means algorithms generalize naturally to solve this problem in the same time bounds as for the k-means problem.

Let f be a differentiable real-valued strictly convex function. The Bregman Divergence D_f induced by f is

$$D_f(x, y) = f(x) - f(y) - \nabla_f(y)(x - y)$$

Notice that the Bregman Divergence induced from $f(x) = x^2$, gives squared Euclidian Distance (k-Means). Bregman divergences are not metrics since they are not symmetric in general and the triangle inequality is not necessarily satisfied. They do have other redeeming qualities, for instance Bregman Divergences are convex in the first argument, albeit not the second, see [9, 10] for a more comprenhensive treatment.

The Bregman Clustering problem as defined in [9] is to find k centroids $\mathcal{M} = \{\mu_1, ..., \mu_k\}$ that minimize

$$\sum_{x \in \mathcal{X}} \min_{\mu \in \mathcal{M}} D_f(x, \mu)$$

where D_f is a Bregman Divergence. For our case, where the inputs $x, y \in \mathbb{R}$, we assume that computing a Bregman Divergence, i.e. evaluating f and its derivative, takes constant time. We show that the k-Means algorithms naturally generalize to 1D clustering using any Bregman Divergence to define the cluster cost while still maintaing the same running time as for k-Means.

Implementation. An independent implementation of the $O(n \lg n + kn)$ time algorithm is available in the R package Ckmeans.1d.dp [22]. This is the algorithm first presented in [25]. The implementation is for k-Means clustering, and uses $O(kn)$ space.

2 Algorithms for 1D k-Means - A Review

In the following we assume sorted input $x_1 \leq ... \leq x_n \in \mathbb{R}$. Notice that there could be many ways of partitioning the input and computing centroids that achieve the same cost. This is for instance the case if the input is n identical points. The task at hand is to find any optimal solution for 1D k-Means.

Let $CC(i, j) = \sum_{\ell=i}^{j} (x_\ell - \mu_{i,j})^2$ be the cost of grouping $x_i, ..., x_j$ into one cluster with the optimal choice of centroid, $\mu_{i,j} = \frac{1}{j-i+1} \sum_{\ell=i}^{j} x_\ell$, the mean of the points.

Lemma 1. *It takes $O(n)$ time and space to construct a data structure that computes $CC(i, j)$ in constant time*

Proof. This is a standard application of prefix sums. By definition,

$$CC(i, j) = \sum_{\ell=i}^{j} (x_\ell - \mu_{i,j})^2 = \sum_{\ell=i}^{j} x_\ell^2 + \mu_{i,j}^2 - 2x_\ell \mu_{i,j} = (j - i + 1)\mu_{i,j}^2 + \mu_{i,j} \sum_{\ell=i}^{j} x_\ell + \sum_{\ell=i}^{j} x_\ell^2.$$

Using prefix sum arrays of $x_1, \ldots, x_n$ and $x_1^2, \ldots, x_n^2$ computing the centroid $\mu_{i,j}$ and the sums takes $O(1)$ time. □

2.1 The $O(kn^2)$ Dynamic Programming Algorithm

The algorithm finds the optimal clustering using i clusters for all prefixes of the points $x_1, \ldots, x_m$, for $m = 1, \ldots, n$, and $i = 1, \ldots, k$ with Dynamic Programming as follows. Let $D[i][m]$ be the cost of optimally clustering $x_1, ..., x_m$ into i clusters. If $i = 1$ the cost of optimally clustering $x_1, ..., x_m$ is the cluster cost $CC(1, m)$. That is, $D[1][m] = CC(1, m)$ for all m. By Lemma 1, this takes $O(n)$ time.

For $i > 1$

$$D[i][m] = \min_{j=1}^{m} D[i-1][j-1] + CC(j, m) \tag{2}$$

Notice that $D[i-1][j-1]$ is the cost of optimally clustering $x_1, ..., x_{j-1}$ into $i - 1$ clusters and $CC(j, m)$ is the cost of clustering $x_j, ..., x_m$ into one cluster. This makes x_j the first point in the last and rightmost cluster. Let $T[i][m]$ be the argument that minimizes (2)

$$T[i][m] := \arg\min_{j=1}^{m} D[i-1][j-1] + CC(j, m) \tag{3}$$

It is possible there exists multiple j obtaining same minimal value for (2). To make the optimal clustering unique, such ties are broken in favour of smaller j.

Notice $x_{T[i][m]}$ is the first point in the rightmost cluster of the optimal clustering. Thus, given T one can find the optimal solution by standard backtracking. One can naively compute each entry of D and T using (2) and (3). This takes $O(n)$ time for each cell, thus D and T can be computed in $O(kn^2)$ time using $O(kn)$ space. This is exactly what is described in [23]. This dynamic programming recursion is also presented in [25].

2.2 An $O(kn)$ time algorithm

The Dynamic Programming algorithm can be sped up significantly to $O(kn)$ time by reducing the time to compute each row of D and T to $O(n)$ time instead of $O(n^2)$ time. This is exactly what [25] does, in particular it is shown how to reduce the problem of computing a row of D and T to searching an implicitly defined $n \times n$ matrix of a special form, which then allows computing each row of D and T in linear time.

Define $\mathrm{C}_i[m][j]$ as the cost of the optimal clustering of $x_1, \ldots, x_m$ using i clusters, restricted to having the rightmost cluster (largest cluster center) contain the elements $x_j, \ldots, x_m$. For convenience, we define $\mathrm{C}_i[m][j]$ for $j > m$ as the cost of clustering $x_1, \ldots, x_m$ into $i - 1$ clusters, i.e. the last cluster is empty. This means that C_i satisfies:

$$\mathrm{C}_i[m][j] = D[i-1][\min\{j-1, m\}] + CC(j, m)$$

where by definition $CC(j, m) = 0$ when $j > m$ (which is consistent with the definition in Section 2). This menas that $D[i][m]$ relates to C_i as follows:

$$D[i][m] = \min_j \mathrm{C}_i[m][j]$$

where ties are broken in favor of smaller j (as defined in Section 2.1).

This means that to compute a row of D and T, we are computing $\min_j \mathrm{C}_i[m][j]$ for all $m = 1, \ldots, n$. Think of C_i as an $n \times n$ matrix with rows indexed by m and columns indexed by j. With this interpretation, computing the i'th row of D and T corresponds to computing for each row r in C_i, the column index c that corresponds to the smallest value in row r. In particular, the entries $D[i][m]$ and $T[i][m]$ correpond to the value and the index of the minimum entry in the m'th row of C_i respectively. The problem of finding the minimum value in every row of a matrix has been studied before [2]. First we need the definition of a monotone matrix.

Definition 1. *[2] Let A be a matrix with real entries and let* $\arg\min(i)$ *be the index of the leftmost column containing the minimum value in row i of A. A is said to be* monotone *if $a < b$ implies that* $\arg\min(a) \leq \arg\min(b)$. *$A$ is* totally monotone *if all of its submatrices are monotone.*

In [2], the authors showed:

Theorem 1. *[2] Finding* $\arg\min(i)$ *for each row i of an arbitrary $n \times m$ monotone matrix requires* $\Theta(m \lg n)$ *time, whereas if the matrix is totally monotone, the time is $O(m)$ when $m > n$ and is $O(m(1 + \lg(n/m)))$ when $m < n$.*

The fast algorithm for totally monotone matrices is known as the *SMAWK* algorithm.

Let us relate this monotonicity to the 1D k-Means clustering problem. That C_i is monotone means that if we consider the optimal clustering of the points $x_1, \ldots, x_a$ with i clusters, and we add more points $x_{a+1} \leq \cdots \leq x_b$ after x_a, then the first (smallest) point in the last (rightmost) of the i clusters can only increase (move right) in the new optimal clustering of $x_1, \ldots, x_b$. This sounds like it should be true for 1D k-Means and in fact it is. Thus, applying the algorithm for monotone matrices, one can fill a row of D and T in $O(n \lg n)$ time leading to an $O(kn \lg n)$ time algorithm for 1D k-Means, which is already a great improvement.

However, as shown in [25] the matrix C_i defined above is in fact totally monotone. This follows from the following fact about the 1D k-Means clustering cost

Theorem 2 (Monge Concave for 1D k-Means cost [25])**.** *For any $a < b$ and $u < v$, it holds that:*

$$CC(v, b) + CC(u, a) \leq CC(u, b) + CC(v, a).$$

This is the property known as the *concave monge* (concave for short) property [26, 13, 24]. We prove this for the general case of Bregman Divergences in Section 4. For completeness we include a proof that Total Monotonicity is implied by concave cost funtion below.

Lemma 2. *The matrix C_i is totally monotone if the cluster cost $CC(i, j)$ is monge concave*

Proof. As [2] remarks, a matrix A is totally monotone if all its 2×2 submatrices are monotone. To prove that C_i is totally monotone, we need to prove that for any two row indices a, b with $a < b$ and two column indices u, v with $u < v$, it holds that if $\mathrm{C}_i[a][v] < \mathrm{C}_i[a][u]$ then $\mathrm{C}_i[b][v] < \mathrm{C}_i[b][u]$.

Notice that these values correspond to the costs of clustering elements $x_1, \ldots, x_a$ and $x_1, \ldots, x_b$, starting the rightmost cluster with element x_v and x_u respectively. Since $\mathrm{C}_i[m][j] = D[i-1][\min\{j-1, m\}] + CC(j, m)$, this is the same as proving that

$$\begin{aligned} D[i-1][\min\{v-1, m\}] + CC(v, a) < D[i-1][\min\{u-1, m\}] + CC(u, a) \Rightarrow \\ D[i-1][\min\{v-1, m\}] + CC(v, b) < D[i-1][\min\{u-1, m\}] + CC(u, b) \end{aligned}$$

which is true if we can prove that $CC(v, b) - CC(v, a) \leq CC(u, b) - CC(u, a)$. Rearranging terms, what we need to prove is that for any $a < b$ and $u < v$, it holds that:

$$CC(v, b) + CC(u, a) \leq CC(u, b) + CC(v, a).$$

which is the monge concave property. □

As explained in [25] the total monotonicity property of the cost matrices and the SMAWK algorithm directly yields an $O(kn)$ time (and space) algorithm for 1D k-means.

Theorem 3. *[25] Computing an optimal k-Means clustering of a sorted input of size n for takes $O(kn)$ time.*

By construction the cost of the optimal clustering is computed for all $k' \leq k$. By storing the T table with cluster centers then for any $k' \leq k$ the cluster indices of an optimal clustering can be extracted in $O(k')$ time.

The concave propery has been used to significantly speed up algorithms, in particular (Dynamic Programming) algorithms, for several other 1D problems. For the interested reader we refer to [26, 13, 24].

2.3 A $n2^{O(\sqrt{\lg\lg n \lg k})}$ time algorithm for $k = \Omega(\lg n)$

The concave property of the k-Means cost yields and algorithm for computing the optimal k-Means clustering for a given $k = \Omega(\lg n)$ in $n2^{O(\sqrt{\lg\lg n \lg k})}$ time. The result follows almost directly from Schieber [20] who gives an algorithm with the aforementioned running time for finding the shortest path of fixed length k in a directed acyclic graph with n nodes and weights, $w(i,j)$, that satisfy the concave property. It is assumed that the weights are represented by a function (oracle) that returns the weight of a requested edge in constant time.

Theorem 4 ([20])**.** *Computing a minimum weight path of length k between any two nodes in a directed acyclic graph of size n where the weights satisfy the concave property takes $n2^{O(\sqrt{\lg\lg n \lg k})}$ time using $O(n)$ space.*

The 1D k-Means problem is reducible to this directed graph problem as follows. Sort the input in $O(n \lg n)$ time and let $x_1 \leq x_2 \leq \ldots x_n$ denote the sorted input sequence. For each input x_i associate a node v_i and add an extra node v_{n+1}. Define the weight of the edge from v_i to v_j as the cost of clustering $x_i, \ldots, x_{j-1}$ in one cluster, which is $CC(i, j-1)$. Each edge weight is computed in constant time (by Lemma 1) and the edge weights satisfy the monge concave property by construction. Finally, to compute the optimal clustering use Schieber's algorithm to compute the lowest weight path with k edges from v_1 to v_{n+1}.

Theorem 5. *Computing an optimal k-Means clustering of an input of size n for given $k = \Omega(\lg n)$ takes $n2^{O(\sqrt{\lg\lg n \lg k})}$ time using $O(n)$ space.*

It is relevant to briefly consider parts of Schieber's algorithm and how it relates to k-Means clustering, in particular a *regularized* version of the problem. Schieber's algorithm relies crucially on algorithms that given a directed acyclic graph where the weights satisfy the concave property computes a minimum weight path in $O(n)$ time [24, 15]. Note the only difference in this problem compared to above, is that the search is not restricted to paths of k edges only. As noted in [1], if the weights are integers then the algorithm solves the Monge Concave Directed Graph Problem in $n \lg U$ time where U is the largest absolute value of a weight. In the reduction from 1D k-Means the weights are not integers and we must take care.

2.4 Regularized 1D k-Means and an $O(n \lg U)$ time algorithm for k-Means

Consider a regularized version of the k-Means clustering problem where instead of providing the number of clusters k we additionally specify a penalty per cluster and ask to minimize the cost of the clustering plus the penalty λ for each cluster used. Formally, the problem is as follows: Given $\mathcal{X} = \{x_1, ..., x_n\} \subset \mathbb{R}$ and λ, compute the optimal regularized clustering:

$$\underset{k, \mathcal{M}=\{\mu_1,\ldots,\mu_k\}}{\arg\min} \sum_{x \in \mathcal{X}} \min_{\mu \in \mathcal{M}} (x-\mu)^2 + \lambda k$$

If we set $\lambda = 0$ the optimal clustering has cost zero and use a cluster for each input point. If we let λ increase towards infinity, the optimal number of clusters used in the optimal solution monotonically decreases towards one (zero clusters is not well defined). Let d_{min} be the smallest distance between input points and for simplicity assume all the input points are distinct. The optimal cost of using $n-1$ clusters is then $d_{\text{min}}^2/2$. When $\lambda > \lambda_{n-1} = d_{\text{min}}^2/2$ it is less costly to use only $n-1$ clusters compared to using n clusters since the benefit of using one more cluster is smaller than the cost of a cluster. Letting λ increase again inevitably leads to a miminum value $\lambda_{n-2} > \lambda_{n-1}$ such that for $\lambda > \lambda_{n-2}$ using only $n-2$ clusters gives a better optimal cost than using $n-1$ clusters. Following the same pattern λ_{n-2} is the difference between the optimal cost using $n-2$ clusters and $n-1$ clusters.

Continuing this way yields the very interesting sequence $0 < \lambda_{n-1} \leq \ldots \lambda_2 \leq \lambda_1$ that encodes the only relevant choices for the regularization parameter λ, where $\lambda_i =^{\text{def}} \text{OPT}_i - \text{OPT}_{i+1}$ and OPT_i is the cost of an optimal k-Means clustering with i clusters. Note that the $O(nk)$ algorithm actually yields $\lambda_1, \ldots, \lambda_{k-1}$ since it computes the optimal cost for all $k' \leq k$. It is an interesting open problem if one can compute this sequence in $n \lg^{O(1)} n$ time, since this encodes all the relevant information about the input instance using linear space, and from that the 1D k-Means clustering can be reported in $O(n)$ time for any k.

In the reduction to the directed graph problem, adding a cost of λ for each cluster used corresponds to adding λ to the weight of each edge. Note that the edge weights still satisfy the concave property. Thus, solving the regularized version of k-Means clustering correponds to finding the shortest path (of any length) in a directed acyclic graph where the weights satisfy the concave property. By the algorithms in [24, 15] this takes $O(n)$ time.

Theorem 6. *Computing an optimal regularized 1D k-Means clustering of a sorted input of size n takes $O(n)$ time.*

Now if we actually use λ_k as the cost per cluster in regularized 1D k-Means, or any $\lambda \in [\lambda_k, \lambda_{k+1}]$ there is an optimal regularized cost solution that uses k clusters which is also an optimal k-Means clustering.

This leads to an algorithm for 1D k-Means based on a (parametric) binary search for such a parameter λ starting with an initial interval of $[\lambda_1, \lambda_{n-1}]$ (both easily computed). However, there are a few things that need to be considered to ensure that such a binary search can return an optimal clustering. Most importantly, it may be the case that $\lambda_k = \lambda_{k+1}$ This occurs when the decrease in the clustering cost for going from k to $k+1$ clusters is the same as the decrease when going from $k-1$ to k clusters. In this case the regularized cost with $\lambda = \lambda_k = \lambda_{k+1}$ using $k+1$ and k clusters are the same. This generalizes to ranges $\lambda_{j+1} < \lambda_j = \lambda_{j-1} = \cdots = \lambda_i < \lambda_{i-1}$ for some $j > i+1$. In this case the optimal regularized cost of using any $k \in \{i+1, \ldots, j\}$ is found at λ_j and even if we can find this value we still need a way to extract an optimal regularized clustering with exactly k clusters. Fortunately, it is straight forward to compute both an optimal cost regularized clustering with the fewest clusters and an optimal regularized clustering with the most clusters and an optimal regularized clustering with any number of clusters in between these two [1, 20].

In conclusion a binary search on λ ends when the algorithm probes a value where the the number of clusters possible in the optimal regularized clustering contains k. In the worst case this interval is a point, so with $\lg U$ bits integer coordinate input points, this takes in $O(\lg U)$ steps in the worst case.

Theorem 7. *Computing an optimal k-Means clustering of an input of size n takes $O(n \lg U)$ time and $O(n)$ space.*

3 Space Reduction for Dyamic Programming and 1D k-Means Reporting Data Structure

The space consumption of the $O(kn)$ time dynamic programming algorithm is $O(kn)$ for storing the tables D and T (Equation 2, 3). This can easily be reduced to $O(n)$ by not storing T at all and only storing the last two considered rows D during the algorithm (computation of each row of D depends only on the values of the previous row). However, doing this has the downside that we can no longer report an optimal k-Means clustering, only the cost of one. This problem can be handled by a space reduction technique of Hirschberg [12] that reduce the space usage of the $O(kn)$ dynamic programming algorithm to just $O(n)$ while maintaining $O(kn)$ running time. For completeness we have given this constuction in the appendix. This is the standard way of saving space for dynamic programming algorithms and is also applied for a monge concave problem in [11]. The problem is that it does cost a constant factor in the running time.

However, for the 1D k-Means problems (and other monge concave problems), we actually do not need to bog down the dynamic programming algorithm at all. So instead simply do as follows: First run the dynamic programming algorithm for $k+1$ clusters, storing only the last two rows as well as the last column of the dynamic programming table D. Remeber that the i'th entry in the last column stores the cost of the optimal k-Means clustering with i clusters i.e. OPT_i. This takes $O(kn)$ time uses linear space and returns $\mathrm{OPT}_1, \ldots, \mathrm{OPT}_k, \mathrm{OPT}_{k+1}$. Given these optimal costs, to extract an optimal clustering compute $\lambda_k = \mathrm{OPT}_k - \mathrm{OPT}_{k+1}$ and apply the regularized k-Means clustering algorithm with λ_k. By our analysis in Section 2.4, this λ_k yields an optimal regularized clustering of size k, and the algorithm directly returns an actual optimal clustering. Of course this works for any $k' \leq k$, storing only the optimal k-Means costs $\mathrm{OPT}_1, \ldots, \mathrm{OPT}_k, \mathrm{OPT}_{k+1}$ we can report an optimal clustering for any $k' \leq k$ in $O(n)$ time, completely forgoing the need to store dynamic programming tables. We directly get

Theorem 8. *Computing an optimal k-Means clustering of a sorted input of size n takes $O(kn)$ time and $O(n)$ space.*

To compute the cost of the optimal clustering for all $k' \leq k$ we keep the last column of the cost matrix D which requires an additional $O(k) = O(n)$ space.

Theorem 9. *There is a data structure that uses $O(n)$ space that can report the optimal 1D k-Means clustering for any k in $O(n)$ time.*

Note that the best preprocessing time we know for this data structure is $O(n^2)$ by simply running the dynamic programming algorithm with $k = n$, but of course it can be constructed using a smaller k at the cost of only being able to report optimal clusterings up to size k. Furthermore, this construction does not depend on actual using the k-Means cost function, just the Monge Concave property and the directed graph shortest path problem and thus generalize to all problems considered by Schieber [20].

4 Extending to More Distance Measures

In the following we show how all the above algorithms generalize to all Bregman Divergences and the sum of absolute distances while retaining the same running time and space consumption.

4.1 Bregman Divergence Clusterings

First, let us remind ourselves what a Bregman Divergence and a Bregman Clustering is. Let f be a differentiable real-valued strictly convex function. The Bregman Divergence D_f defined by f is defined as

$$D_f(x, y) = f(x) - f(y) - \nabla_f(y)(x - y)$$

Bregman Clustering. The Bregman Clustering problem as defined in [9], is to find a set of centers, $\mathcal{M} = \{\mu_1, ..., \mu_k\}$, that minimizes

$$\sum_{x \in \mathcal{X}} \min_{\mu \in \mathcal{M}} D_f(x, \mu)$$

Notice that the cluster center is the second argument of the Bregman Divergence. This is important since Bregman Divergences are not in general symmetric.

For the purpose of 1D clustering, we mention two important properties of Bregman Divergences. For any Bregman Divergence, the unique element that minimizes the summed distance to a multiset of elements is the mean of the elements, exactly as it was for squared Euclidian distance. This is in a sense the defining property of Bregman Divergences [9]. The second property is the linear separator property, which is crucial for clustering with Bregman Divergences but also for Bregman Voronoi Diagrams [9, 10].

Linear Separators For Bregman Divergences. For all Bregman divergences, the locus of points that are equidistant to two fixed points μ_1, μ_2 in terms of a Bregman divergence is given by $\{x \in \mathcal{X} \mid D_f(x, p) = D_f(x, q)\}$. Plugging in the definition of a Bregman Divergence this is

$$\{x \in \mathcal{X} \mid D_f(x, p) = D_f(x, q)\} = \{x \in \mathcal{X} \mid x(\nabla_f(\mu_1) - \nabla_f(\mu_2)) = f(\mu_1) - \mu_1 \nabla_f(\mu_1) - f(\mu_2) + \mu_2 \nabla_f(\mu_2)\}$$

which is a hyperplane. The points μ_1, μ_2 sit on either side of the hyperplane and the Voronoi cells defined by Bregman divergences are connected.

This means, in particular, that between any two points in 1D, $\mu_1 < \mu_2$, there is a hyperplane (point) h with $\mu_1 < h < \mu_2$ and all points smaller than h are closer to μ_1 and all points larger than h are closer to μ_2. For 1D Bregman Divergence Clustering it means the optimal clusters correponds to intervals (as was the case in 1D k-Means). We capture what we need from this observation in a simple "distance" lemma:

Lemma 3. *Given two fixed real numbers $\mu_1 < \mu_2$, then for any point $x_r \geq \mu_2$, we have $D_f(x_r, \mu_1) > D_f(x_r, \mu_2)$, and for any point $x_l \leq \mu_1$ we have $D_f(x_l, \mu_1) < D_f(x_l, \mu_2)$*

Computing Optimal Cluster Costs for Bregman Divergences. Since the mean minizes Bregman Divergences, the centroids used in optimal clusterings are unchanged compared to the k-Means case. The prefix sums idea used to implement the data structure used for Lemma 1 generalizes to Bregman Divergences as observed in [18] (under the name Summed Area Tables). The formula for computing the cost of grouping the points $x_i, \ldots, x_j$ in one cluster is as follows. Let $\mu_{i,j} = \frac{1}{j-i+1}\sum_{\ell=i}^{j} x_\ell$ be the arithmetic mean of the points $x_i, \ldots, x_j$, then

$$\begin{aligned} CC(i,j) &= \sum_{\ell=i}^{j} D_f(x_\ell, \mu_{i,j}) \\ &= \sum_{\ell=i}^{j} f(x_\ell) - f(\mu_{i,j}) - \nabla_f(\mu_{i,j})(x_\ell - \mu_{i,j}) \\ &= \left(\sum_{\ell=i}^{j} f(x_\ell)\right) - (j-i+1)f(\mu_{i,j}) - \nabla_f(\mu_{i,j}) \left(\left(\sum_{\ell=i}^{j} x_\ell\right) - (j-i+1)\mu_{i,j} \right) \end{aligned}$$

Rearranging terms this is

$$\left(\sum_{\ell=i}^{j} f(x_\ell)\right) - (j-i+1)f(\mu_{i,j}) - \nabla_f(\mu_{i,j}) \left(\left(\sum_{\ell=i}^{j} x_\ell\right) - (j-i+1)\mu_{i,j} \right).$$

Thus the Bregman Divergence cost of a consecutive subset of input points can be computed in in constant time with stored prefix sums for $x_1, \ldots, x_n$ and $f(x_1), \ldots, f(x_n)$.

4.1.1 Totally Monotone Matrix and Monge Concave property for Bregman Divergences

Lemma 4. *For any Bregman Divergence D_f the induced cluster cost $CC(i,j)$ function is concave. Formally, for any $a < b$ and $u < v$, it holds that:*

$$CC(v,b) + CC(u,a) \leq CC(u,b) + CC(v,a).$$

Proof. We start by handling the special case where $v > a$. In this case, we have by definition that $CC(v,a) = 0$, thus we need to show that $CC(v,b) + CC(u,a) \leq CC(u,b)$. This is the case since any point amongst $x_u, \ldots, x_b$ is included in at most one of $x_v, \ldots, x_b$ and $x_u, \ldots, x_a$ (since $a < v$). Thus $CC(v,b) + CC(u,a)$ is the cost of taking two disjoint and consecutive subsets of the points $x_u, \ldots, x_b$ and clustering the two sets using the optimal choice of centroid in each. Clearly this cost is at most the cost of clustering all the points using one centroid (both groups could use the same cluster center as the cluster center for all the points).

We now turn to the general case where $u < v \leq a < b$. Let $\mu_{v,a}$ be the mean of $x_v, \ldots, x_a$ and $\mu_{u,b}$ be the mean of $x_u, \ldots, x_b$ and assume that $\mu_{v,a} \leq \mu_{u,b}$ (the other case is symmetric). Finally, let $CC(w,z)_\mu = \sum_{\ell=w}^{z} D_f(x_\ell, \mu)$ denote the cost of grouping the elements $x_w, \ldots, x_z$ into a cluster with centroid μ.

Split the cost $CC(u,b)$ into the cost of the elements $x_u, \ldots, x_{v-1}$ and the cost of the elements $x_v, \ldots, x_b$ i.e.

$$CC(u,b) = \sum_{\ell=u}^{v-1} D_f(x_\ell, \mu_{u,b}) + \sum_{\ell=v}^{b} D_f(x_\ell, \mu_{u,b}) = CC(u, v-1)_{\mu_{u,b}} + CC(v,b)_{\mu_{u,b}}.$$

We trivially get $CC(v,b)_{\mu_{u,b}} \geq CC(v,b)$ since $CC(v,b)$ is the cost using the optimal centroid.

Since $\mu_{v,a} \leq \mu_{u,b}$ and all elements $x_u, \ldots, x_{v-1}$ are less than or equal to $\mu_{v,a}$ (since $\mu_{v,a}$ is the mean of points $x_v, \ldots, x_a$ that all are greater than $x_u, \ldots, x_{v-1}$), then by Lemma 3,

$$CC(u, v-1)_{\mu_{u,b}} \geq CC(u, v-1)_{\mu_{v,a}}$$

Adding the cluster cost $CC(v,a)$ to both sides of this inequality we get that

$$CC(u, v-1)_{\mu_{u,b}} + CC(v,a) \geq CC(u, v-1)_{\mu_{v,a}} + CC(v,a) = CC(u,a)_{\mu_{v,a}} \geq CC(u,a)$$

Combining the results,

$$CC(v,b) + CC(u,a) \leq CC(v,b)_{\mu_{u,b}} + CC(u, v-1)_{\mu_{u,b}} + CC(v,a) = CC(u,b) + CC(v,a).$$

□

It follows that all the results achieved for 1D k-Means presented earlier generalize to any Bregman Divergence.

4.2 k-Median Clustering

For the k-Medians problem we replace the the sum of squared Euclidian distances with the sum of absolute distances. Formally, the k-Medians problem is to compute a clustering, $\mathcal{M} = \{\mu_1, ..., \mu_k\}$, minimizing

$$\sum_{x\in\mathcal{X}} \min_{\mu\in\mathcal{M}} |x-\mu|$$

Note that in 1D, all L_p norms are the same and reduce to this case. Also note that the minimizing centroid for a cluster is no longer the mean of the points in that cluster, but the median. To solve this problem, we change the centroid to be the median, and if the number of points is even, we fix the median to be the exact middle point between the two middle elements, making the choice of centroid unique.

As for Bregman Divergences, we need to show that we can compute the cluster cost $CC(i,j)$ with any Bregman Divergence in constant time. Also, we need to compute the centroid in constant time and argue that the cost is monge moncave which implies the implicit matrix C_i is totally monotone. The arguments are essentially the same, but for completeness we briefly cover them below.

Computing Cluster Costs for Absolute Distances. Not surprisingly, prefix sums still allow constant time computation of $CC(i,j)$. Let $m_{i,j} = \frac{j+i}{2}$, and compute the centroid as $\mu_{i,j} = \frac{x_{\lfloor m_{i,j}\rfloor} + x_{\lceil m_{i,j}\rceil}}{2}$ then

$$CC(i,j) = \sum_{\ell=i}^{j} |x_\ell - \mu_{i,j}| = \sum_{\ell=i}^{\lfloor m_{i,j}\rfloor} \mu_{i,j} - x_\ell + \sum_{\ell=1+\lfloor m_{i,j}\rfloor}^{j} x_\ell - \mu_{i,j}$$

which can be computed in constant time with access to a prefix sum table of $x_1, \dots, x_n$. This was also observed in [18].

Monge Concave - Totally Monotone Matrix. The monge concave and totally monotone matrix argument above for Bregman Divergences (and for squared Euclidian distance) remain valid since first of all, we still have $x_u, \dots, x_{v-1} \le \mu_{v,a}$ as $\mu_{v,a}$ is the median of points all greater than $x_u, \dots, x_{v-1}$. Furthermore, it still holds that when $\mu_{v,a} \le \mu_{u,b}$ and all elements $x_u, \dots, x_{v-1}$ are less than or equal to $\mu_{v,a}$, then $CC(u, v-1)_{\mu_{u,b}} + CC(v,a) \ge CC(u, v-1)_{\mu_{v,a}} + CC(v,a) = CC(u,a)_{\mu_{v,a}}$. It follows that the algorithms we specified for 1D k-Means generalize to the 1D k-Median problem.

5 Experiments

To asses the practicality of 1D k-Means algorithms we have implemented and compared different versions of the $O(kn)$ time algorithm and the $O(n \lg U)$ time binary search algorithm. We believe Schiebers $n2^{O(\sqrt{\lg\lg n \lg k})}$ time algorithm [20] is mainly of theoretical interest, at least the constants involved in the algorithm seem rather large. Similarly for the $O(k\sqrt{n \lg n})$ time algorithm of [1]. Both are slightly similar to the $O(n \lg U)$ time algorithm in the sense they work by trying to find the right λ value for the regularized k-Means problem so that it uses k clusters. However, to get data independent bounds (independent of U) the algorithms become much more complicated. For reasonable values of k we expect the simple $O(kn)$ time algorithm to be competitive with these more theoretical efficient algorithms. Our main interest is seeing how the binary search algorithm compares.

5.1 Data Sets

For the experiments we consider two data sets.

Uniform: The `Uniform` data set is, as the name suggest, created by sampling the required number of points uniformly at random between zero and one.

Gaussian Mixture: The `Gaussian` Mixture data set is created by defining 16 Gaussian Distributions, each with variance 100, and means placed one million apart. Each data point required is created by uniformly at random sampling one of the 16 gaussians and then sample a point from that.

5.2 Algorithm Setup

Dynamic Programming. For the dynamic programming algorithm we have implemented the $O(kn \lg n)$ and the $O(kn)$ time algorithm. The difference lies only in how the monotone matrix search is performed (Section 2.2). The latter using the SMAWK algorithm and the former a simple divide and conquer approach. This is done because the SMAWK algorithm may in fact be slower in practice. Both versions have been implemented using only $O(n)$ space in two different ways, the first version only storing the last row of the dynamic programming matrix and the second version reducing space using the Hirschberg space saving technique (Section 3).

Binary Search. For the binary search algorithm we have considered two algorithms for regularized k-Means, namely the algorithms in [24] and [15]. The invariants and pseudo-code in [15] are not entirely correct and does not directly turn into working code. We were able to fix these issues and made the algorithm run, however, our implementation of [15] is clearly slower than our implementation of [24]. Therefore, for the experiments we only consider the linear time algorithm from [24].

Remember that λ_i is the minimum value for $\lambda \geq 0$ where the optimal regularized cost of using i clusters is less than the optimal regularized cost of using $i + 1$ clusters. The standard implementation is a binary search on the regularization coefficient (λ), that ends when a λ that gives an optimal k-Means clustering is found. The starting range for λ is $[0, M]$ where M is the cost of clustering all points in one cluster (OPT_1). For $\lambda = 0$, the optimal clustering uses n clusters and has a cost of zero. For $\lambda = M$, the optimal clustering uses one cluster, as its regularized cost is $2M$, and using at least 2 clusters costs strictly more than $2M$ (assuming any cluster costs more than 0). The algorithm simply maintains the current range for λ, λ_{low} and λ_{high}. We denote by k_{low}, and k_{high} the number of clusters used in the optimal regularized cost solution at λ_{low} and λ_{high} respectively. Note that $k_{\text{high}} < k_{\text{low}}$, and let $I_k = \{k_{\text{high}}, \ldots, k_{\text{low}}\}$ be the current interval for the number clusters that includes k. Finally, let c_{low} and c_{high} denote the unregularized cost of using k_{low} and k_{high} clusters respectively.

Many different values for λ give the same regularized clustering (same number of clusters), which means if we attempt two or more such values for λ, we make almost no progress. Also, the middle of the current range for λ does not in any way translate to the middle of the range $k_{\text{high}}, k_{\text{low}}$. Early experiments showed that the standard binary search actually had this issue.

To overcome this issue, we modified the search to work in a more data dependent way to guarantee progress by picking the next λ as the value where the regularized costs of using k_{high} clusters and k_{low} clusters are the same. At this position we are guaranteed that there is an optimal cost regularized clustering using k' clusters where k' is strictly contained in k_{high} and k_{low} which we prove below.

To be able to do this we update the algorithm to also maintain $k_{\text{high}}, k_{\text{low}}$, and $c_{\text{high}}, c_{\text{low}}$. Given these computing λ' is straight forward, namely.

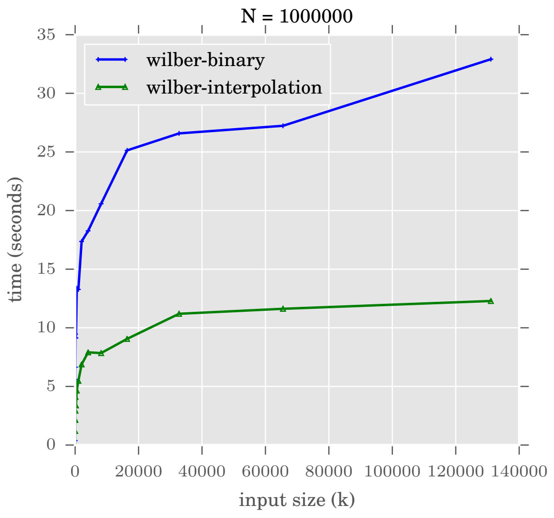
$$\lambda' = (c_{\text{high}} - c_{\text{low}})/(k_{\text{low}} - k_{\text{high}})$$

which is the average increase in the optimal cost per cluster added.

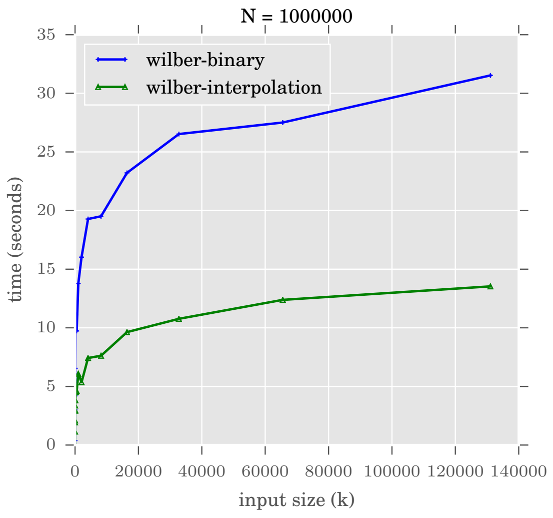
Lemma 5. *Let $\lambda' = (c_{high} - c_{low})/(k_{low} - k_{high})$, at λ' there is an optimal cost regularized clustering with k' clusters where k' is contained in $\{k_{high} + 1, \ldots, k_{low} - 1\}$*

Proof. Consider the regularized k-Means cost as a function of λ. This is a piecewise linear function since it corresponds to the the minimum of n lines, the k'th line representing the regularized cost of an optimal clustering using k clusters, having slope k and intercept (value at $\lambda = 0$) the optimal (non-regularized) cost of using k clusters. For each line the subset of the real line where this line is the minimun of the n lines is a line segment, and the line segment's end points correspond to when the number of clusters in the optimal cost regularized clustering change.

This means that the value λ' we try in the binary search is the intersection of the two line segments corresponding to λ_{low} and λ_{high}. At this λ' the regularized cost of using k_{low} clusters and k_{high} clusters is the same. For all k' betwen



(a) Running times on `Gaussian` Data Set



(b) Running times on `Uniform` Data Set

Figure 1: Running time comparison binary search algorithms as a function of k

k_{high} and k_{low} the regularized cost of the optimal regularized clustering at λ' is at most that of using k_{low} or k_{high} clusters (that are the same).

Consider a number of clusters k' strictly between k_{low} and k_{high} and the associated line for using k' clusters. This line must intersect the line for k_{low} at some value $x \leq \lambda'$, otherwise the line can never be the minimum line since for $x < \lambda'$ it would be above the line for k_{low} because of its lower slope, and for $x > \lambda'$ it would be above the line for k_{high} due to higher slope. Finally, since the line for k' clusters has lower slope than the line for k_{low} and intersects it at $x \leq \lambda'$ the regularized cost at λ' of using k' clusters is at most the cost of using k_{low}. $\square$

Notice, that if k_{high} clusters achieve the smallest regularized cost at λ', then the regularized cost of all $k' \in I_k$ at λ' are the same and the optimal k-Means clustering is found.

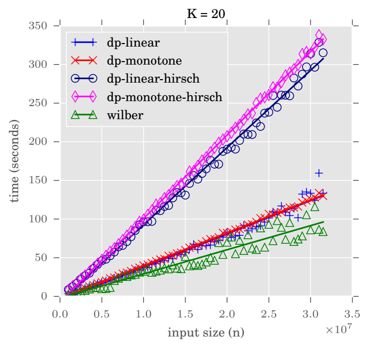
The unfortunate side effect of using this strategy is the running time of $O(n \lg U)$ may no longer hold. That is easily remedied by for instance every $t = O(1)$ steps query the midpoint of the remaining range, however no such trick is used in our implementation.

We refer to the standard binary search algorithm as *wilber-binary* and the intersection binary search as *wilber-interpolation*. For a comparison of these two algorithms, see Figure 1. In these plots the search based on the intersection/interpolation is clearly superior on both data sets and and we consider only this version of the binary search in the following runtime comparison with the dynamic programming algorithms.

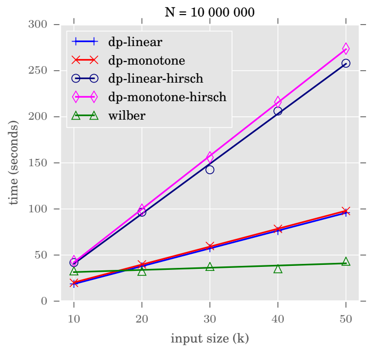
5.3 Algorithm Comparison

We refer to the $O(kn \lg n)$ algorithm with space saving mentioned in Section 2.2 as `DP-monotone` and the variant using the Hirschberg technique as `DP-monotone-hirsch`. Similarly we call the $O(kn)$ algorithm (also with space saving) `DP-linear`, and `DP-linear-hirsch` when using the Hirschberg technique. Finally we denote the $O(n \lg U)$ algorithm `Wilber` since the regularized k-Means is an implementaion of Wilber's algorithm [24]. It should be noted that the space saving technique really is necessary as n and k grow, since otherwise the space grows with the product, which is quite undesirable.

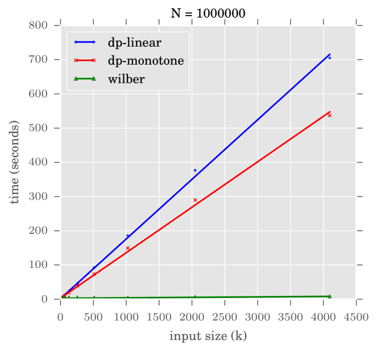
The experiments do **not** include the time for actually reporting a clustering which gives the `DP-linear` an advantage over the other algorithm since it would require an invocation of the `Wilber` algorithm to report a clustering,



(a) Running times as a function of n on the `Uniform` data set



(b) Running times as a function of k on the `Uniform` data set



(c) Running times as a function of k on the `Gaussian` data set

Figure 2: Running time comparison of different 1D k-Means algorithms. dp-* are different versions of dynamic programming and wilber is the binary search based on interpolation

while `DP-linear-hirsch` and `Wilber` extracts an optimal clustering to report during the algorithm and would have no extra cost.

Figures 2a, 2b, and 2c show the running time as a function of n or k on the `Uniform` data set. The performance of the dynamic programming algorithms are as expected. These algorithm always fill out a table of size kn and are thus never better than the worst case running time. The plots also reveal that for the values of n tested, the $O(kn \lg n)$ is in fact superior to the $O(kn)$ time algorithm albeit the difference is not large. As the plots show, when k grows, the `Wilber` algorithm is much faster than the other algorithms (even when $k = 20$ for the smallest n we tried). This is both true for `Uniform` and the `Gaussian` data set. It is also worth noting that even for moderate values of n and k the space quickly goes in the order of gigabytes for the dynamic programming solutions, if we maintain the entire table.

Notice that the dynamic programming algorithm can report the clustering cost for all $k' \leq k$ using an additive $O(k)$ space by always keeping the final column of the dynamic programming table. On the other hand `Wilber` cannot report the costs of all clusterings, but it can report *some* of them, as it searches for a cluster cost λ that yields a clustering with k clusters. For each λ that is attempted, the cost of an optimal clustering using a different k may be reported. For practitioners that want to see the plot of the cost of the best clustering as a function of either k or λ, the `Wilber` algorithm might still be sufficient, as it does provide points on that curve and one can even make an interactive plot: if a desired point on the curve is missing just compute it and add more points to the plot. For a new λ this takes linear time and for a new k we need to binary search the interval between the currently stored nearest neighbours.

The simple conclusion is that for these kinds of data, the binary search algorithm is superior even for moderate n and k, and for large n, k it is the only choice. If one prefers the guarantee of the dynamic programming algorithm, implementing `Wilber` allows for saving a non-trivial constant factor in the running time for linear space algorithms and allows reporting an optimal clustering for any $k' \leq k$ in linear time.

Final Remarks

We have given an overview of 1D k-Means algorithms, generalized them to new measures, shown the practical performance of several algorithm variants including a simple way of boosting the binary search algorithm. We have defined the obvious regularized version of 1D k-Means which is important not only for fast algorithms based on binary search but also linear space solutions for reporting of actual optimal clusterings based on the dynamic programming algorithms. We see a few important problems left open

- Is there an $n \lg^{O(1)} n$ time algorithm for 1D k-Means or maybe even an $n \lg^{O(1)} k$ time algorithm (if the input is sorted)?

- Is there an $n \lg^{O(1)} n$ or even $n^{2-O(1)}$ time algorithm for computing the optimal k-Means costs for all $k = 1, \ldots, n$ yielding the sequence $\lambda_1, \ldots, \lambda_{n-1}$ that encodes all relevant information for the given 1D k-Means instance?

- What is the running time of the search algorithm using the tweak we employed? An easy bound is $O(n^2)$ but we were not able to get such lousy running time in practice. In fact it seemed to be a really good heuristic for picking the next query point in the binary search.

- The dynamic programming algorithm with a running time of $O(kn)$ can rather easily be parallelized to run in $O((kn \lg n)/p)$ for p processors, by parallelizing the monotone matrix search algorithm (not SMAWK but the simple $O(n \lg n)$ divide and conquer algorithm). For the binary search algorithm, it is possible to try and improve the $\lg U$ to $\lg_p U$ for p processors, but it would be much better if one could parallelize the linear time 1D regularized k-Means algorithm (or a near linear time version of it).

We wish to thank Pawel Gawrychowski for pointing out important earlier work on concave property.

References

[1] A. Aggarwal, B. Schieber, and T. Tokuyama. Finding a minimum-weight-link path in graphs with the concave monge property and applications. *Discrete & Computational Geometry*, 12(3):263–280, 1994.

[2] Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter Shor, and Robert Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1):195–208, 1987.

[3] Sara Ahmadian, Ashkan Norouzi-Fard, Ola Svensson, and Justin Ward. Better guarantees for k-means and euclidean k-median by primal-dual algorithms. *CoRR*, abs/1612.07925, 2016.

[4] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. Np-hardness of euclidean sum-of-squares clustering. *Machine Learning*, 75(2):245–248, 2009.

[5] Valerio Arnaboldi, Marco Conti, Andrea Passarella, and Fabio Pezzoni. Analysis of ego network structure in online social networks. In *Privacy, security, risk and trust (PASSAT), 2012 international conference on and 2012 international confernece on social computing (SocialCom)*, pages 31–40. IEEE, 2012.

[6] David Arthur and Sergei Vassilvitskii. How slow is the k-means method? In *Proceedings of the Twenty-second Annual Symposium on Computational Geometry*, SCG '06, pages 144–153. ACM, 2006.

[7] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[8] Pranjal Awasthi, Moses Charikar, Ravishankar Krishnaswamy, and Ali Kemal Sinop. The hardness of approximation of euclidean k-means. In *31st International Symposium on Computational Geometry, SoCG 2015, June 22-25, 2015, Eindhoven, The Netherlands*, pages 754–767, 2015.

[9] Arindam Banerjee, Srujana Merugu, Inderjit S. Dhillon, and Joydeep Ghosh. Clustering with bregman divergences. *J. Mach. Learn. Res.*, 6:1705–1749, December 2005.

[10] Jean-Daniel Boissonnat, Frank Nielsen, and Richard Nock. Bregman voronoi diagrams. *Discrete & Computational Geometry*, 44(2):281–307, 2010.

[11] Mordecai J. Golin and Yan Zhang. A dynamic programming approach to length-limited huffman coding: space reduction with the monge property. *IEEE Trans. Information Theory*, 56(8):3918–3929, 2010.

[12] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, June 1975.

[13] D. S. Hirschberg and L. L. Larmore. The least weight subsequence problem. *SIAM Journal on Computing*, 16(4):628–638, 1987.

[14] Olga Jeske, Mareike Jogler, Jörn Petersen, Johannes Sikorski, and Christian Jogler. From genome mining to phenotypic microarrays: Planctomycetes as source for novel bioactive molecules. *Antonie Van Leeuwenhoek*, 104(4):551–567, 2013.

[15] Maria M. Klawe. A simple linear time algorithm for concave one-dimensional dynamic programming. Technical report, Vancouver, BC, Canada, Canada, 1989.

[16] Euiwoong Lee, Melanie Schmidt, and John Wright. Improved and simplified inapproximability for k-means. *Information Processing Letters*, 120:40–43, 2017.

[17] Meena Mahajan, Prajakta Nimbhorkar, and Kasturi Varadarajan. *The Planar k-Means Problem is NP-Hard*, pages 274–285. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[18] Frank Nielsen and Richard Nock. Optimal interval clustering: Application to bregman clustering and statistical mixture learning. *IEEE Signal Process. Lett.*, 21:1289–1292, 2014.

[19] Diego Pennacchioli, Michele Coscia, Salvatore Rinzivillo, Fosca Giannotti, and Dino Pedreschi. The retail market as a complex system. *EPJ Data Science*, 3(1):1, 2014.

[20] Baruch Schieber. Computing a minimum weight-link path in graphs with the concave monge property. *Journal of Algorithms*, 29(2):204 – 222, 1998.

[21] Andrea Vattani. k-means requires exponentially many iterations even in the plane. *Discrete & Computational Geometry*, 45(4):596–616, 2011.

[22] Haizhou Wang and Joe Song. Ckmeans.1d.dp: Optimal and fast univariate clustering; R package version 4.0.0., 2017.

[23] Haizhou Wang and Mingzhou Song. Ckmeans. 1d. dp: optimal k-means clustering in one dimension by dynamic programming. *The R Journal*, 3(2):29–33, 2011.

[24] Robert Wilber. The concave least-weight subsequence problem revisited. *Journal of Algorithms*, 9(3):418 – 425, 1988.

[25] Xiaolin Wu. Optimal quantization by matrix searching. *J. Algorithms*, 12(4):663–673, December 1991.

[26] F. Frances Yao. Efficient dynamic programming using quadrangle inequalities. In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, STOC '80, pages 429–435. ACM, 1980.

A Reducing space usage of $O(kn)$ time dynamic programming algorithm using Hirschberg

Remeber that each row of T and D (Equation 2, 3) only refers to the previous row. Thus one can clearly "forget"row $i-1$ when we are done computing row i In the following, we present an algorithm that avoids the table T entirely.

The key observation is the following: Assume $k > 1$ and that for every prefix $x_1, \dots, x_m$, we have computed the optimal cost of clustering $x_1, \dots, x_m$ into $\lfloor k/2 \rfloor$ clusters. Note that this is the set of values stored in the $\lfloor k/2 \rfloor$'th row of D. Assume furthermore that we have computed the optimal cost of clustering every suffix $x_m, \dots, x_n$ into

$k - \lfloor k/2 \rfloor$ clusters. Let us denote these costs by $\tilde{D}[k - \lfloor k/2 \rfloor][m]$ for $m = 1, \ldots, n$. Then clearly the optimal cost of clustering $x_1, \ldots, x_n$ into k clusters is given by:

$$D[k][n] = \min_{j=1}^{n} D[\lfloor k/2 \rfloor][j] + \tilde{D}[k - \lfloor k/2 \rfloor][j+1]. \tag{4}$$

The main idea is to first compute row $\lfloor k/2 \rfloor$ of D and row $k - \lfloor k/2 \rfloor$ of $\tilde{D}$ using linear space. From these two, we can compute the argument j minimizing (4). We can then split the reporting of the optimal clustering into two recursive calls, one reporting the optimal clustering of points $x_1, \ldots, x_j$ into $\lfloor k/2 \rfloor$ clusters, and one call reporting the optimal clustering of $x_{j+1}, \ldots, x_n$ into $k - \lfloor k/2 \rfloor$ clusters. When the recursion bottoms out with $k = 1$, we can clearly report the optimal clustering using linear space and time as this is just the full set of points.

From Section 2.2 we already know how to compute row $\lfloor k/2 \rfloor$ of D using linear space: Simply call SMAWK to compute row i of D for $i = 1, \ldots, \lfloor k/2 \rfloor$, where we throw away row $i-1$ of D (and don't even store T) when we are done computing row i. Now observe that table $\tilde{D}$ can be computed by taking the points $x_1, \ldots, x_n$ and reversing their order by negating the values. This way we obtain a new ordered sequence of points $\tilde{X} = \tilde{x}_1 \leq \tilde{x}_2 \leq \cdots \leq \tilde{x}_n$ where $\tilde{x}_i = -x_{n-i+1}$. Running SMAWK repeatedly for $i = 1, \ldots, k - \lfloor k/2 \rfloor$ on the point set $\tilde{X}$ produces a table $\hat{D}$ such that $\hat{D}[i][m]$ is the optimal cost of clustering $\tilde{x}_1, \ldots, \tilde{x}_m = -x_n, \ldots, -x_{n-m+1}$ into i clusters. Since this cost is the same as clustering $x_{n-m+1}, \ldots, x_n$ into i clusters, we get that the $(k - \lfloor k/2 \rfloor)$'th row of $\hat{D}$ is identical to the i'th row of $\tilde{D}$ if we reverse the order of the entries.

To summarize the space saving algorithm for reporting the optimal clustering, does as follows: Let L be an initially empty output list of clusters. If $k = 1$, append to L a cluster containing all points. Otherwise ($k > 1$), use SMAWK on $x_1, \ldots, x_n$ and $-x_n, \ldots, -x_1$ to compute row $\lfloor k/2 \rfloor$ of D and row $k - \lfloor k/2 \rfloor$ of $\tilde{D}$ using linear space (by evicting row $i-1$ from memory when we have finished computing row i) and $O(kn)$ time. Compute the argument j minimizing (4) in $O(n)$ time. Evict row $\lfloor k/2 \rfloor$ of D and row $k - \lfloor k/2 \rfloor$ of $\tilde{D}$ from memory. Recursively report the optimal clustering of points $x_1, \ldots, x_j$ into $\lfloor k/2 \rfloor$ clusters (which appends the output to L). When this terminates, recursively report the optimal clustering of points $x_{j+1}, \ldots, x_n$ into $k - \lfloor k/2 \rfloor$ clusters. When the algorithm terminates, L contains the optimal clustering of $x_1, \ldots, x_n$ into k clusters.

At any given time, the algorithm uses $O(n)$ space: We evict all memory used to compute the value j minimizing (4) before recursing. Furthermore, we complete the first recursive call (and evict all memory used) before starting the second recursive call. Finally, for the recursion, we do not need to copy $x_1, \ldots, x_j$. It suffices to remember that we are only working on the subset of inputs $x_1, \ldots, x_j$.

Let $F(n, k)$ denote the time used by the above algorithm to compute an optimal clustering of n sorted points into k clusters. Then there is a constant $C > 0$ such that $F(n, k)$ satisfies the recurrence: $F(n, 1) \leq Cn$, and for $k > 1$:

$$F(n, k) \leq \max_{j=1}^{n} F(j, \lfloor k/2 \rfloor) + F(n-j, k - \lfloor k/2 \rfloor) + Cnk.$$

We claim that $F(n, k)$ satisfies $F(n, k) \leq 3Ckn$. We prove the claim by induction in k. The base case $k = 1$ follows trivially by inspection of the formula for $F(n, 1)$. For the inductive step $k > 1$, we use the induction hypothesis to conclude that $F(n, k)$ is bounded by

$$\begin{aligned} & \max_{j=1}^{n} 3Cj\lfloor k/2 \rfloor + 3C(n-j)(k - \lfloor k/2 \rfloor) + Cnk \\ & \leq \max_{j=1}^{n} 3Cj\lceil k/2 \rceil + 3C(n-j)\lceil k/2 \rceil + Cnk \\ & = 3Cn\lceil k/2 \rceil + Ckn. \end{aligned}$$

For $k > 1$, we have that $\lceil k/2 \rceil \leq (2/3)k$, therefore:

$$F(n, k) \leq 3Cn(2/3)k + Ckn = 3Ckn.$$