Optimal 1D k-Means with Dynamic Programming

Abstract—A dynamic programming (DP) approach provides a practical optimal algorithm for the common one-dimensional (1D) case of the k-means clustering problem. Performance of top-down and bottom-down DP implementations is characterized analytically and empirically, and contrasted with that of a naïve recursive approach.

1 Introduction

The k-means clustering problem is the partitioning of n vectors into k clusters, where each vector belongs to the cluster with the nearest mean. An optimal solution minimizes the sum of squared distances from vectors to their corresponding cluster means.

For vector spaces with more than one dimension, the k-means clustering problem is known to be NP-hard [1], and is solved non-optimally by an approximate heuristic algorithm. In the important one-dimensional (1D) case, however, a dynamic programming approach provides a computationally tractable optimal algorithm [2].

1.1 Motivation

A wide range of real-world problems involving irregular quantization or cluster analysis benefit from efficient and effective k-means algorithms, including image segmentation or color palette selection, codebook selection in data compression, and feature identification in machine learning.

1.2 Formulation

Considering only the 1D case, let the input to the algorithm be a multiset of n scalars $x_1, \ldots, x_n$, identified by indices $i \in \{1, \ldots, n\}$. Without loss of generality, the input is ordered $x_1 \leq x_2 \leq \ldots \leq x_n$.[1] A weight w_i may be assigned to each point x_i. Then the input comprises the vectors $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$ and $\mathbf{w} = \langle w_1, \ldots, w_n \rangle$, along with a scalar k indicating the number of clusters to be found.

The k-means output is thus a multiset $\mathcal{M} = \langle \mu_1, \ldots, \mu_k \rangle$ of cluster means (centroids, or centers of mass when non-uniform weights are used). Each input point is considered to belong to the cluster around the μ_i from which it is the smallest squared (Euclidean) distance. (By definition, each candidate k-mean is the weighted arithmetic mean of the subset of input points in its cluster.) Then the optimal output will minimize the weighted sum of squared distances of inputs from their cluster centroids,

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

1.3 Sample Input

The input $\mathbf{x} = \langle -22,\ -16,\ -15,\ -13,\ -9,\ -4,\ -2,\ 1,\ 3,\ 6,\ 11,\ 12,\ 15,\ 21,\ 27 \rangle$, as shown in Figure 1, is sampled from a uniform Gaussian mixture $\langle \mathcal{N}(\mu : -15, \sigma^2 : 3^2),\ \mathcal{N}(0, 4^2),\ \mathcal{N}(20, 7^2) \rangle$, with each sample rounded to the nearest integer.[2] For now, let the weights be uniform, $\mathbf{w} = \langle 1, \ldots, 1 \rangle$.

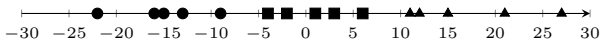

Fig. 1. Visualization of sample input from Gaussian mixture

2 Applying Dynamic Programming

Since the input $\mathbf{x}$ is weakly increasing, and the distance metric satisfies the triangle inequality, the points associated with any given cluster have contiguous indices $i, \ldots, j$. For any candidate cluster of inputs from i to j, the centroid or center of mass μ_{ij} is the weighted arithmetic mean $\frac{\sum_{\alpha=i}^{j} w_\alpha x_\alpha}{\sum_{\alpha=i}^{j} w_\alpha}$, and the cost of the cluster is $K_{ij} = \sum_{\alpha=i}^{j} w_\alpha (x_\alpha - \mu_{ij})^2$, which is $\sum_{\alpha=i}^{j} w_\alpha$ times the cluster variance. It

1. Any unordered input can be efficiently transformed into a nondecreasing one. Heapsort offers $\Theta(n \lg n)$ time; radix sort offers linear time given a fixed word size.

2. In general, sampling from a mixture model means randomly choosing a distribution from a weighted set, then sampling from the chosen distribution. For this simple example, each distribution contributes exactly 5 samples.

follows that $C(\mathcal{M})$ is simply the sum of cluster costs.

Before attacking the heart of the k-means problem, it is worthwhile to spend $\Theta(n)$ time and space computing three prefix-sum vectors:[3]

$$\mathbf{p}_0 = \langle p_{0,1} : 0, p_{1,1} : w_1, \ldots, p_{n,1} : \textstyle\sum_{\alpha=1}^{n} w_\alpha \rangle$$
$$\mathbf{p}_1 = \langle p_{0,2} : 0, p_{1,2} : w_1 x_1, \ldots, p_{n,2} : \textstyle\sum_{\alpha=1}^{n} w_\alpha x_\alpha \rangle$$
$$\mathbf{p}_2 = \langle p_{0,3} : 0, p_{1,3} : w_1 x_1^2, \ldots, p_{n,3} : \textstyle\sum_{\alpha=1}^{n} w_\alpha x_\alpha^2 \rangle$$

PREFIX-SUMS(x, w)
1 $n = x.length$ // same as $w.length$
2 let $p[0\,..\,n, 0\,..\,2]$ be a new table
3 $p[0, 0\,..\,2] = 0, 0, 0$
4 **for** $i = 1$ **to** n
5 $\quad p[i, 0] = p[i-1, 0] + w[i]$
6 $\quad p[i, 1] = p[i-1, 1] + w[i] \times x[i]$
7 $\quad p[i, 2] = p[i-1, 2] + w[i] \times \mathrm{pow}(x[i], 2)$
8 **return** p

Then

$$\begin{aligned} K_{ij} &= \textstyle\sum_{\alpha=i}^{j} w_\alpha (x_\alpha - \mu_{ij})^2 \\ &= \mu_{ij}^2 \textstyle\sum_{\alpha=i}^{j} w_\alpha - 2\mu_{ij} \sum_{\alpha=i}^{j} w_\alpha x_\alpha + \sum_{\alpha=i}^{j} w_\alpha x_\alpha^2 \\ &= \mu_{ij}^2 (p_{j,0} - p_{i-1,0}) - 2\mu_{ij} (p_{j,1} - p_{i-1,1}) \\ &\quad + (p_{j,2} - p_{i-1,2}) \end{aligned}$$
$$\mu_{ij} = \frac{\sum_{\alpha=i}^{j} w_\alpha x_\alpha}{\sum_{\alpha=i}^{j} w_\alpha} = \frac{p_{j,1} - p_{i-1,1}}{(p_{j,0} - p_{i-1,0})}$$

Thereafter we can compute any K_{ij} and μ_{ij} in $\Theta(1)$ time.

CLUSTER(p, i, j)
1 $w = p[j, 0] - p[i-1, 0]$
2 $wx = p[j, 1] - p[i-1, 1]$
3 $wxx = p[j, 2] - p[i-1, 2]$
4 $centroid = wx/w$
5 $a = \mathrm{pow}(centroid, 2) \times w$
6 $cluster\text{-}cost = a - 2 \times centroid \times wx + wxx$
7 **return** $centroid$, $cluster\text{-}cost$

2.1 Structure of an Optimal Solution

Roughly following the notation of [3] (which uses the opposite meaning of i and m and indexing convention as compared to [2]), we find the optimal clustering of $x_1, \ldots, x_n$ into k clusters by solving all smaller subproblems in both dimensions. That is, we find the optimal clustering using $i = 1, \ldots, k$ clusters for all prefixes of the original multiset of points: $x_1, \ldots, x_m$ for $m = 1, \ldots, n$.

Consider the $k+1$ by $n+1$ array D, where $D[i, m]$ is the optimal (minimum) $C(\mathcal{M}_{i,m})$ of the subproblem with i clusters and point multiset prefix of length m. Then $D[k, n]$ will be the optimal $C(\mathcal{M})$ corresponding to the overall solution.

In a given (sub)problem, consider the optimal solution for $D[i, m]$. The final cluster i contains a smallest point at index j.[4] Then consider the subproblem $D[i-1, j-1]$. Effectively, this is the subproblem with the last cluster removed. The optimal solution to this subproblem must be part of the optimal solution $D[i, m]$, because any more optimal solution could be subtituted without affecting the "removed" cluster—thereby producing a more optimal solution to $D[i, m]$. This demonstrates the *optimal substructure* required for DP and leads directly to a recursive formulation.

2.1.1 Recursive Definition of an Optimal Solution

For each subproblem defined by i and m, we consider the cluster from index j to m; its cost K_{jm} can be looked up in constant time using the prefix-sums array. The optimal cost for a given j then includes the optimal subproblem cost for the remaining points $D[i-1, j-1]$.

Then the optimal solution for $D[i, m]$ is simply the best such solution over all possible choices of j. We can record the best choice of j for each (i, m) in an additional array T of the same dimensions as D. The recursive formulation is thus[5]

$$D[i, m] = \min_{j=1}^{m} D[i-1, j-1] + K_{jm}$$
$$T[i, m] = \arg\min_{j=1}^{m} D[i-1, j-1] + K_{jm}$$

As the same subproblems are used repeatedly, the *overlapping subproblems* requirement for applying DP is satisfied. As an initial condition for the recurrence, $D[1, m]$ is K_{1m} for all m; that is, the optimal solution for a single cluster is simply the cost of the only possible cluster. (This implies we can omit row 0 to obtain a k by $n+1$ array.)

2.2 Computing an Optimal Solution

2.2.1 Recovering the Optimal Solution

If some algorithm has filled the required entries in D and T, and the minimal total clustering cost is $D[k, m]$, then the first (lowest-valued) point in the last (rightmost) cluster has index $T[k, n]$ [3]. The

3. For uniform weights, only two are required: $\mathbf{p}_1$ is trivial.

4. To obtain a deterministic result, ties can be arbitrarily broken by choosing the lower index [2].

5. As in [2], the minimum can actually be found over $i \le j \le m$ instead of $1 \le j \le m$ as above, since an optimal solution will have at least one point in each cluster.

following backtracking algorithm prints the indices defining the optimal clusters; it could just as easily populate a data structure.

```
PRINT-SOLUTION(T, k, n)
1  while k > 0
2      i = T[k, n]
3      j = n
4      print "cluster" k "from" i "to" j
5      k = k − 1
6      n = i − 1
```

2.2.2 Naïve Recursive Algorithm

While the recursive formulation is already described in terms of arrays of optimal subproblem solutions, leading naturally to a DP approach, it is possible to implement an inefficient, entirely recursive approach. Here the array D from the recursive formulation is implicit, although the array T is still filled so the optimal solution can be recovered by backtracking.

```
RECURSIVE-K-MEANS(x, w, k)
1  n = x.length    // same as w.length
2  let p[0 . . n, 0 . . 2] = PREFIX-SUMS(x, w)
3  let T[1 . . k, 0 . . n] be a new array
4  fill T[1 . . k, 0 . . n] with 1
5  RECURSIVE-AUX(p, T, k, n)    // returns cost
6  PRINT-SOLUTION(T, k, n)

RECURSIVE-AUX(p, T, i, m)
 1  if m == 0
 2      return 0
 3  elseif i == 1
 4      mu, d = CLUSTER(p, 1, m)
 5      return d
 6  d = ∞
 7  for j = 1 to m
 8      mu, K = CLUSTER(p, j, m)
 9      s = RECURSIVE-AUX(p, T, i − 1, j − 1)
10      if s + K < d
11          d = s + K
12          T[i, m] = j
13  return d
```

In a given non-leaf invocation of RECURSIVE-AUX with parameters i and m, $\Theta(m)$ work is done in the loop body; plus, m subproblems of sizes from $i-1,1$ to $i-1,m-1$ are recursively required. The average size of the required subproblems is a constant times m, which is also $\Theta(m)$.

Hence in the first level there is $\Theta(m)$ work to do; in the second, there is $\Theta(m)$ work in each of $\Theta(m)$ subproblems, or $\Theta(m^2)$, and so on. The maximum recursion depth is i, so the work done is $\Theta(m) + \Theta(m^2) + \ldots + \Theta(m^i) = \Theta(m^i)$, and since the top-level invocation has parameters k and n, the overall algorithm has exponential running time $\Theta(n^k)$. As a result, it is completely impractical for even medium-sized problems.

2.2.3 Top-Down/Memoized Algorithm

This is very similar to RECURSIVE-K-MEANS, but the D array from the recursive formulation becomes explicit, and all entries are initialized to a negative value to enable memoization. Since individual cluster costs are nonnegative, the recursion can return immediately whenever it encounters a nonnegative entry—indicating a particular subproblem that has already been solved. Instead of returning a cost, the memoized routine populates the appropriate entries in D; when the top-level instance returns, $D[k, n]$ will be the total cost of the optimal solution.

```
TOP-DOWN-K-MEANS(x, w, k)
1  n = x.length    // same as w.length
2  let p[0 . . n, 0 . . 2] = PREFIX-SUMS(x, w)
3  let D[1 . . k, 0 . . n] , T[1 . . k, 0 . . n] be arrays
4  fill D[1 . . k, 0 . . n] with −∞    // −1 suffices
5  fill T[1 . . k, 0 . . n] with 1
6  TOP-DOWN-AUX(p, D, T, k, n)
7  PRINT-SOLUTION(T, k, n)    // cost is D[k, n]

TOP-DOWN-AUX(p, D, T, i, m)
 1  if m == 0
 2      D[i, m] = 0
 3  elseif i == 1
 4      mu, D[i, m] = CLUSTER(p, 1, m)
 5  elseif D[i, m] < 0    // unsolved subproblem?
 6      D[i, m] = ∞
 7      for j = 1 to m
 8          mu, K = CLUSTER(p, j, m)
 9          TOP-DOWN-AUX(p, D, T, i − 1, j − 1)
10          if D[i − 1, j − 1] + K < D[i, m]
11              D[i, m] = D[i − 1, j − 1] + K
12              T[i, m] = j
```

The size of the table D is $\Theta(kn)$. TOP-DOWN-AUX thus encounters an unsolved subproblem $\Theta(kn)$ times. For each, it does $\Theta(n)$ work in the innermost loop. The number of recursive invocations does not matter so much because those not corresponding to unsolved subproblems will complete in $\Theta(1)$ time, and we already know how many unsolved subproblems there will be overall. Hence the running time is the product of the $\Theta(kn)$ table cells and the $\Theta(n)$ in the innermost loop, or

$\Theta\left(kn^2\right)$ overall; and this is also the running time of TOP-DOWN-K-MEANS, as other than TOP-DOWN-AUX it only does a constant number of operations each taking at most $\Theta\left(n\right)$ time.

2.2.4 Bottom-Up Algorithm

The bottom-up version transforms recursion into iteration; it uses extra initialization to handle boundary conditions; and it chooses iteration order carefully to ensure that required subproblems have already been solved. If $0\,..\,n$ is one of k rows in D, with $D[k,n]$ at the bottom right, then to fill any given entry in D, only those to the left from the preceding row need to be considered. Hence filling rows from top to bottom and filling each row from left to right is a suitable iteration order.

BOTTOM-UP-K-MEANS(x, w, k)

```
 1  n = x.length     // same as w.length
 2  let p[0..n, 0..2] = PREFIX-SUMS(x, w)
 3  let D[1..k, 0..n] , T[1..k, 0..n] be arrays
 4  fill T[1..k, 0..n] with 1
 5  for i = 1 to k
 6       D[i, 0] = 0
 7  for m = 1 to n
 8       mu, D[1, m] = CLUSTER(p, 1, m)
 9  for i = 2 to n
10       for m = 1 to n
11            D[i, m] = ∞
12            for j = 1 to m
13                 mu, K = CLUSTER(p, j, m)
14                 if D[i − 1, j − 1] + K < D[i, m]
15                      D[i, m] = D[i − 1, j − 1] + K
16                      T[i, m] = j
17  PRINT-SOLUTION(T, k, n)     // cost is D[k, n]
```

The same argument as for TOP-DOWN-K-MEANS implies a $\Theta\left(kn^2\right)$ running time for BOTTOM-UP-K-MEANS as well.

2.3 Sample Output

Figure 2 demonstrates optimal k-means clusters for the sample input of section 1.3 for several values of k. In this case, the case $k = 3$ correctly separates points from the three distributions in the Gaussian mixture.

Figure 3 repeats Figure 2, but with weights equal to the squares of the point values; points farther away from the center have much more influence. As k increases with this weighting, the heavily-weighted edges of the range quickly accumulate single-element clusters.

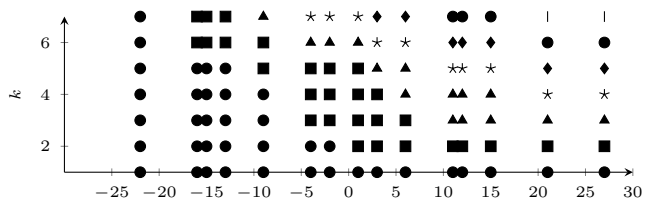

Fig. 2. Clusters for original sample input, $k \in \{1, \ldots, 7\}$

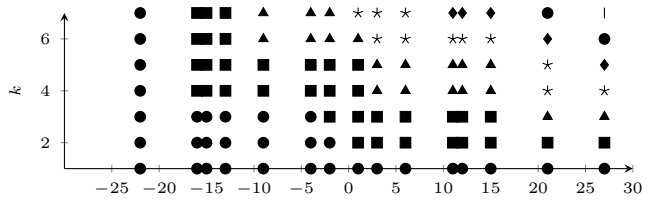

Fig. 3. Clusters for original $\mathbf{x}$ with $w_i = x_i^2$, $k \in \{1, \ldots, 7\}$

3 BENCHMARKS

In all cases, algorithms are benchmarked using a high-resolution monotonic timer. Benchmark code is integrated in the implementations and does not include reading the input and weights or enforcing the initial sorted input invariant. It does include allocating and initializing working storage, the core algorithm, and solution recovery by backtracking.

Each combination of n and k is executed at least 10 times to reduce measurement variance, each time on a freshly generated input vector with values drawn from a uniform distribution.

3.0.1 Naïve Recursive Algorithm

Figure 4 shows the running time of the naïve recursive algorithm for n from 10 to 100 and for k from 2 to 9; each higher curve represnets a successively higher value of k. This is obviously an impractical algorithm for other than extremely low k; the case with $(n, k) = (80, 9)$ already takes over 15 minutes to solve a single problem!

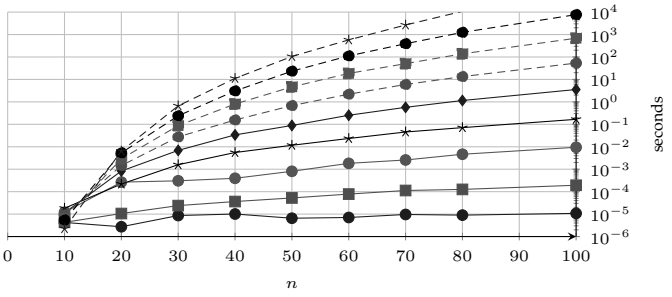

Fig. 4. Recursive running time $k \in \{2, \ldots, 10\}$, $n = 10 \rightarrow 100$

Looking at the running time for $n = 100$ in Figure 4, note that each time k increases by one, the running time moves up the axis by a roughly constant distance. Since the time axis is logarithmic, this suggests a running time that is exponential with respect to k, and is consistent with our previous claim of $\Theta\left(n^k\right)$.

3.0.2 Top-Down/Memoized Algorithm

Figure 5 shows the running time of the top-down DP solution for the same problem sizes the recursive algorithm struggled with in Figure 4, plus a few higher values of k.

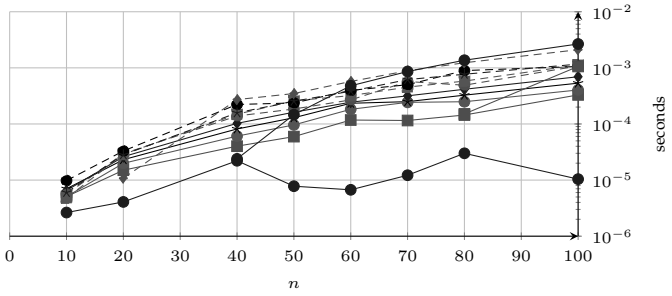

Fig. 5. Top-down running time $2 \leq k \leq 80$, $n = 10 \rightarrow 100$

This plot is too crowded to easily pick out individual values of k; what is important is that the running time is relatively insensitive to k, even as it approaches n. Meanwhile, the running time with respect to n is sublinear on a semilog plot, indicating that it is less than exponential. Both are consistent with the claimed $\Theta\left(kn^2\right)$ running time.

In any case, the usefulness of DP is evident; the same $(n, k) = (80, 9)$ problem that took the recursive approach over 15 minutes is solved in under a millisecond.

3.0.3 Bottom-Up Algorithm

Figure 6 shows the running time of the bottom-up DP solution. Differences from Figure 5 are minimal, except that the linear memory access patterns and iterative implementation lead to more consistent running times, and modestly shorter ones for small-ish k. The bottom-down implementation seems more sensitive to large values of k, suggesting that the top-down approach might be able to skip some unnecessariy subproblems in these cases.

4 Parameters and Extensions

As shown in Figure 3, the choice of input weights can substantially affect the optimal solution, although it should not generally affect the running

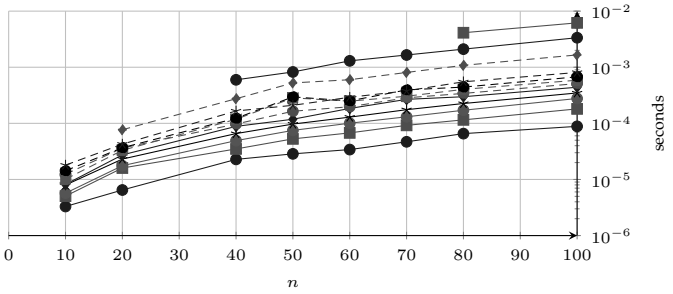

Fig. 6. Bottom-up running time $2 \leq k \leq 80$, $n = 10 \rightarrow 100$

time. In practice, these weights could reflect varying levels of certainty in distinct input measurements. The effect of negative weight values has not been considered; it is not immediately clear whether the algorithm remains correct or meaningful in their presence. Note that the weights not only affect the assignment of input values to clusters, but also the position of the center for a given cluster.

An extended version of the algorithm could implement an alternative cost function; instead of the squared Euclidean distance, it could use absolute distances (k-medians) or some other metric. While the DP algorithm would not remain correct under all possible cost functions, it turns out that 1D clustering with k-medians or any of the broader class of metrics known as Bregman divergences can be done with the same efficiency as k-means [3].

5 Conclusion

An algorithm employing dynamic programming provides an optimal alternative to the standard heuristic algorithm for the k-means problem in the one-dimensional case. It offers polynomial-time performance suitable for medium-sized problems, and points the way to more sophisticated optimal algorithms with even lower asymptotic complexity.

References

[1] M. Mahajan, P. Nimbhorkar, and K. Varadarajan, "The planar k-means problem is np-hard," in *WALCOM: Algorithms and Computation*, S. Das and R. Uehara, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 274–285.

[2] H. Wang and M. Song, "Ckmeans.1d.dp: Optimal k-means clustering in one dimension by dynamic programming," *The R Journal*, vol. 3, no. 2, pp. 29–33, 2011. [Online]. Available: https://journal.r-project.org/archive/2011-2/RJournal_2011-2_Wang+Song.pdf

[3] A. Grønlund, K. G. Larsen, A. Mathiasen, and J. S. Nielsen, "Fast exact k-means, k-medians and bregman divergence clustering in 1d," *CoRR*, vol. abs/1701.07204, 2017. [Online]. Available: http://arxiv.org/abs/1701.07204