# clustering.sc.dp: Optimal Clustering with Sequential Constraint by Using Dynamic Programming

*by Tibor Szkaliczki*

**Abstract** The general clustering algorithms do not guarantee optimality because of the hardness of the problem. Polynomial-time methods can find the clustering corresponding to the exact optimum only in special cases. For example, the dynamic programming algorithm can solve the one-dimensional clustering problem, i.e., when the items to be clustered can be characterised by only one scalar number. Optimal one-dimensional clustering is provided by package **Ckmeans.1d.dp** in R. The paper shows a possible generalisation of the method implemented in this package to multidimensional data: the dynamic programming method can be applied to find the optimum clustering of vectors when only subsequent items may form a cluster. Sequential data are common in various fields including telecommunication, bioinformatics, marketing, transportation etc. The proposed algorithm can determine the optima for a range of cluster numbers in order to support the case when the number of clusters is not known in advance.

## Introduction

Clustering plays a key role in various areas including data mining, character recognition, information retrieval, machine learning applied in diverse fields such as marketing, medicine, engineering, computer science, etc. A clustering algorithm forms groups of similar items in a data set which is a crucial step in analysing complex data. Clustering can be formulated as an optimisation problem assigning items to clusters while minimising the distances among the cluster members. The normally used clustering algorithms do usually not find the optimal solution because the clustering problem is NP-complete in the general case. This paper introduces a package implementing an optimisation method for clustering in a special case when a sequential constraint should be met, i.e., when the items to be clustered are sorted and only subsequent items may form a cluster. This constraint is common when clustering data streams, e.g., audio and video streams, trajectories, motion tracks, click-streams etc. The good news is that the exact optimum can be found in polynomial time in this case.

The algorithm recommended for clustering sequential data is based on the dynamic programing approach developed by Wang and Song (2011). They gave a polynomial-time algorithm for one-dimensional clustering, i.e., when the items can be characterised by only one scalar number. Similarly to the heuristic $k$-means algorithm, it divides data into $k$ groups and it minimises the within-cluster sum of squared distances (WCSS or *withinss* for short). The algorithm guarantees a solution minimising the optimisation goal. The source code of the algorithm is available in the R package **Ckmeans.1d.dp** (Song and Wang, 2011). The generalisation of the algorithm to the multiple dimensional space has been open so far. We extended the dynamic programming approach from one-dimensional clustering to multidimensional clustering with sequential constraint (i.e., only subsequent elements of the input may form a cluster). The method finds the exact optimum in this case as well. We implemented the algorithm in the R package **clustering.sc.dp** (Szkaliczki and Song, 2015).

Although the original algorithm has been developed to find the optimal solution with exactly $k$ clusters it can determine the optimal value for all numbers of clusters less than or equal to $k$ in a single run. For this reason, we implemented two variants of the algorithm. The first one finds the optimal solution for a specific $k$ which can be used if the number of clusters is known in advance. The second variant returns the vector containing the minimal *withinss* for all cluster numbers less than or equal to $k$. This extension of the algorithm is useful if the number of clusters is not known in advance which is a common case.

The remainder of this paper is organized as follows: In the next section, a brief overview of the related work is presented. Then the optimization problem is formally described and the developed optimization algorithm is introduced in detail. Some evaluation results are also presented and the usage of the implemented package is introduced. A brief summary concludes the paper.

## Related work

Several clustering models and a broad variety of clustering methods are available in the literature (Jain, 2010; Tan et al., 2006). Minimising *withinss* is a common optimisation goal used, e.g., in the

popular $k$-means method (Lloyd, 1982) to find a solution for a specific number of clusters and in the Ward's method (Ward, Jr., 1963) belonging to the hierarchical clustering methods. The problem is NP-complete (Aloise et al., 2009; Dasgupta and Freund, 2009; Mahajan et al., 2009). The general clustering methods cannot be directly applied to our problem because the produced solution usually violates the sequential constraint.

As mentioned, one-dimensional clustering can be solved in polynomial time (Wang and Song, 2011). The problem represents a special kind of clustering with sequential constraint because a necessary condition for the optimality in one dimension is that only subsequent items may form a cluster if the items are considered in their scalar order. The package presented in this paper generalised the one-dimensional clustering method to the multidimensional case. The dynamic programming method used for optimal clustering in one dimension is essentially the same as the one first applied by Bellman (1961) for linear curve approximation. For this reason, our package can be also considered as an implementation of the optimal dynamic programming clustering method proposed by Bellman.

Several papers (e.g., Himberg et al. 2001, Terzi 2006, Tierney et al. 2014) are dealing with clustering with sequential constraints because processing data sequences has a broad application area. Leiva and Vidal (2013) gave a clustering algorithm called Warped $k$-means for minimising *withinss* while considering the sequential constraint. The algorithm tries to reach the optimum by moving items between subsequent clusters. It does not guarantee optimality. Their paper provides a good overview as well on the taxonomy of the problem.

## The problem specification

Clustering methods divide a dataset $X = \{\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_n\}$ of $d$-dimensional vectors into a set $\Pi = \{C_1, C_2, \ldots, C_k\}$ of disjoint clusters where $n$ and $k$ denote the number of items to be clustered and the number of clusters, respectively. Throughout the paper, vectors are distinguished from scalars by a bar over their symbol. In case of clustering with sequential constraint, the items are sorted and the clusters are formed only by subsequent items: $C_j = \left\{\overline{x}_{b_j}, \overline{x}_{b_j+1}, \ldots, \overline{x}_{b_j+n_j-1}\right\}$ where $b_j$ and $n_j$ denote the first item and the number of items in cluster $C_j$, respectively. The optimisation goal is to minimise the within-cluster sum of squared distances (*withinss*) also called sum of squared error (SSE), sum of quadratic errors (SQE) or distortion which is a common measurement of quality in clustering. It is formally defined as follows:

$$withinss = \sum_{j=1}^{k} \sum_{\overline{x}_i \in C_j} \left\| \overline{x}_i - \overline{\mu}_j \right\|^2, \tag{1}$$

where $\|\overline{x}\|$ denotes the Euclidean norm of vector $\overline{x}$ and $\overline{\mu}_j$ is the cluster mean:

$$\overline{\mu}_j = \frac{1}{n_j} \sum_{\overline{x}_i \in C_j} \overline{x}_i. \tag{2}$$

Now, we can formulate our problem as follows:

Input:

Items to be clustered: $X = \{\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_n\}$,

Number of clusters: $k$.

Output:

Optimal clustering: $\Pi = \{C_1, C_2, \ldots, C_k\}$.

Minimise

within-cluster sum of squared distances (*withinss*): Eq. (1).

Subject to

sequential constraint:

$$\left(\overline{x}_{i_1} \in C_j\right) \wedge \left(\overline{x}_{i_2} \in C_j\right) \wedge (i_1 \leq i_3 \leq i_2) \Rightarrow \left(\overline{x}_{i_3} \in C_j\right). \tag{3}$$

general clustering conditions:

each item is clustered:

$$\forall \overline{x}_i \exists C_j : \overline{x}_i \in C_j. \tag{4}$$

one cluster is assigned to each item

$$\left(\overline{x}_i \in C_{j_1}\right) \wedge \left(\overline{x}_i \in C_{j_2}\right) \Rightarrow (j_1 = j_2)\,. \tag{5}$$

## The dynamic programming algorithm

The recursive formula used in the dynamic programming formulation is based on the fact, that if clustering for the first $i$ items in $m$ clusters is optimal then after dropping the last cluster the resulting clustering is optimal with $m-1$ clusters for the remaining items. Let $D[i,m]$ denote the value of the minimal within-cluster sum of squared distances (*withinss*) of the clustering for the first $i$ items by using $m$ clusters. If $j$ denotes the first item of the $m$th cluster then the optimality of $D[i,m]$ implies the optimality of $D[j-1,m-1]$ as well. $D[n,k]$ gives the minimal *withinss* for clustering all items in $k$ clusters where $n$ denotes the total number of items.

The recursive formula applied in the dynamic programming approach is defined as follows (Wang and Song, 2011):

$$D[i,m] = \min_{m \le j \le i} \left\{ D[j-1,m-1] + d\left(\overline{x}_j, \ldots, \overline{x}_i\right) \right\}, 1 \le i \le n, 1 \le m \le k \tag{6}$$

where $d\left(\overline{x}_j, \ldots \overline{x}_i\right)$ is the sum of squared Euclidean distances from $\overline{x}_j, \ldots \overline{x}_i$ to their mean.

The optimal solution can be determined by dynamic programming in two steps. First, the recursive formula is used to find the minimal *withinss*. Then backtracking finds the optimal clustering.

$B[i,m]$ stores the index of the first item $b_m$ of the last cluster in the partial solution belonging to $D[i,m]$ which is used for backtracking the optimal solution after determining the minimal *withinss*. We apply the dynamic programming method to solve optimal clustering for a range of cluster numbers and $k$ denotes the maximum number of clusters in our algorithm ($1 \le k \le n$). The steps of calculating $D[i,m]$ can be implemented as follows:

```
for i := 0 to n
  D[i, 0] := 0        // initialisations
for i := 1 to n
  for m := 1 to min(i, k)
    D[i, m] := MAX_DOUBLE;
    for j := i downto m        // calculating the recursive formula
      if D[i, m] > D[j - 1, m - 1] + d(xj, ..., xi)
        D[i, m] := D[j - 1, m - 1] + d(xj, ..., xi)
        B[i, m] := j
Return D[n, m] for m = 1, ..., k
```

$D[n,m], m = 1, \ldots, k$ gives the minimal distances for different number of clusters. If the number of clusters is known in advance, it is enough to return $D[n,k]$. Otherwise, $D[n,m]$ for all $m \le k$ can be returned for further processing by the user in order to select the proper number of clusters.

The algorithm finds the exact optimum in polynomial time. It runs $O\left(n^2 k\right)$ iterations in which $D[i,m]$ is checked and, if necessary, updated. Each iteration can be performed in time proportional to the dimensions of the vectors ($O(d)$) independently from the number of items and clusters if $d\left(\overline{x}_j, \ldots, \overline{x}_i\right)$ is computed progressively based on $d\left(\overline{x}_{j+1}, \ldots, \overline{x}_i\right)$ and the average of the items. This can be done similarly to the one-dimensional case in the following way. Let $\overline{\mu}_{j,i}$ denote the mean of the items with index between $j$ and $i$. If $j = i$ $d\left(\overline{x}_j, \ldots, \overline{x}_i\right) = 0, \overline{\mu}_{j,i} = \overline{x}_i$. For index $j$ from $i-1$ down to $m$, the algorithm iteratively computes

$$d\left(\overline{x}_j, \ldots, \overline{x}_i\right) = d\left(\overline{x}_{j+1}, \ldots, \overline{x}_i\right) + \frac{i-j}{i-j+1}\left(\overline{x}_j - \overline{\mu}_{j,i-1}\right)^2$$

$$\overline{\mu}_{j,i} = \frac{\overline{x}_j + (i-j)\,\overline{\mu}_{j+1,i}}{i-j+1}$$

Using the above iterative computation, the overall running time is quadratic in the number of items and linear in the number of clusters and the dimensions of the vectors: $O\left(n^2 k d\right)$.

The optimal solution with cluster number $m$ can be backtracked by the help of $B[i,m]$ (Wang and Song, 2011):

$$B[i,m] = \text{argmin}_{m \le j \le i} \left\{ D[j-1,m-1] + d\left(\overline{x}_j, \ldots, \overline{x}_i\right) \right\}, 1 \le i \le n, 1 \le m \le k \tag{7}$$

$B[n, m]$ is equal to the first item of the last cluster in clustering with $m$ clusters. The last cluster contains items $\overline{x}_{B[n,m]}, \ldots, \overline{x}_n$. The further clusters can be determined by using backtracking as follows: if $j$ is the first item of the $l$th cluster then the preceding cluster is formed by the items $\overline{x}_{B[j,l-1]}, \ldots, \overline{x}_{j-1}$. The steps of backtracking to the find optimal clustering using $m$ number of clusters are as follows:

```
the mth cluster is B[n, m], ..., n
j := B[n, m]
for l := m to 2
  the (l - 1)th cluster is B[j, l - 1], ..., j - 1
  j = B[j, l - 1]
```

Backtracking can be performed in linear time in the number of the clusters ($O(k)$).

We would like to mention how to combine the dynamic programming method with methods finding the proper number of clusters. The cluster numbers are typically determined by using a measure of validity (e.g., *withinss*) indicating the goodness of clustering. The "best" $k$ is chosen based on the analysis of the values of the measure for each $k$ within the range of the possible number of clusters. A huge variety of methods are available in the literature to determine the cluster numbers (Milligan and Cooper, 1985; Dimitriadou et al., 2002). Typically, they are applied on the output generated by hierarchical clustering methods. Although our dynamic programming approach does not belong to the hierarchical clustering similar methods can be used on the result of our algorithm for finding the proper number of clusters.

Backtracking can be performed only once if the number of clusters is known in advance or it can be selected by analysing *withinss* contained in the last column of matrix $D$. Otherwise, backtracking should be executed for each possible cluster numbers for further analysis. The method can efficiently determine the optimal clustering for all numbers of clusters less than or equal to $k$ essentially because the most time-consuming part is determining $D$ and $B$ which should be executed only once.

### Implementation

We implemented this dynamic programming algorithm in C++. The implementation was built on source code from the R package **Ckmeans.1d.dp** and we created a new R package **clustering.sc.dp**. In the name of the package, *sc* and *dp* refer to sequential constraint and dynamic programming, respectively. The open-source approach made it possible to reuse the code from package **Ckmeans.1d.dp** but it was beneficial for the original package as well: we suggested a minor change in the code to speed up the code which was incorporated into **Ckmeans.1d.dp** ($\geq$ version 3.3.0).

## Evaluation

### Optimality

If the sequential constraint is considered in clustering than the optimal *withinss* is usually larger than in the general case without the constraint since the constraint excludes many possible solutions. In order to compare our algorithm with general clustering methods such as the $k$-means method, we generated a dataset where the optima without and with sequential constraint are equal. For this purpose, we created a totally ordered vector set where one vector is simultaneously larger or smaller in each coordinate than another vector ($\forall i, j \in \{1, 2, \ldots, n\} \forall k, l \in \{1, 2, \ldots, d\} x_{ik} < x_{jk} \implies x_{il} < x_{jl}$ where $x_{yz}$ denote the $z$th coordinate of item $x_y$). We used a random walk as a totally ordered vector set where the steps between subsequent items were generated by using the exponential distribution. The generated dataset consisted of 10,000 two-dimensional vectors.

We compared the dynamic programming algorithm with the `kmeans()` function in R which provides the Hatigan and Wong (1979) implementation of $k$-means. We ran the algorithms with different cluster numbers from 2 to 50. The minimal *withinss* found by $k$-means was always greater or equal to the optimum value found by `clustering.sc.dp()`.

We used the relative difference in *withinss* from the `kmeans()` result to the optimal value produced by `clustering.sc.dp()` for measuring deviation of the $k$-means result from the optimum. Figure 1 shows the relative difference as a function of the cluster numbers. It can be seen that $k$-means is able to find the optimum if the cluster number is at most 10. For larger cluster numbers, its error starts increasing and the relative difference is more than 20% if the cluster number is 50.

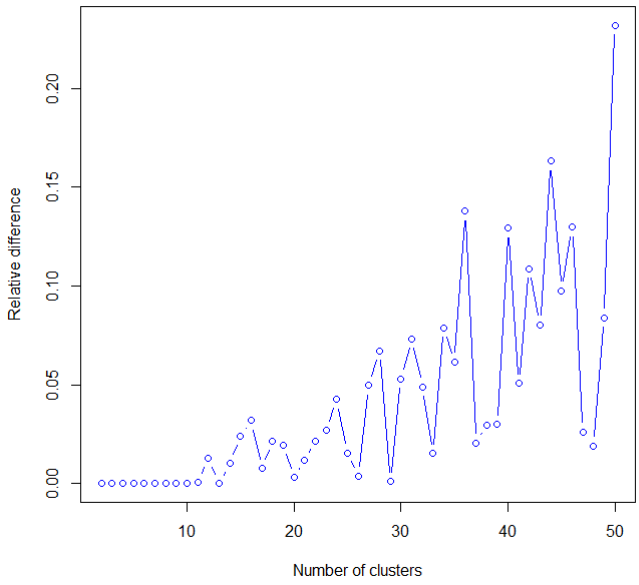


**Figure 1:** The relative difference in *withinss* from `kmeans()` to the optimal value returned by `clustering.sc.dp()`. The input data set of size 10 000 were generated as a random walk having a step size that varies according to an exponential distribution with rate 1.0 in each coordinate.

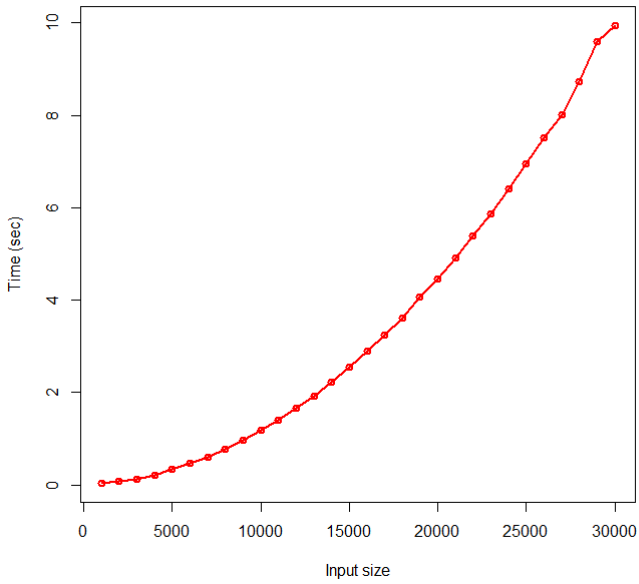


**Figure 2:** Runtime as a function of the number of the items to be clustered.

### Runtime

We tested the dynamic programming algorithm on inputs with different sizes, dimensions and cluster numbers in order to find its performance bounds and examine experimentally how the running time depends on the input sizes. The simulations were run on a desktop computer with a Pentium Dual-Core 2.93 GHz processor and 4 GB memory, running Windows 10 operation system. We generated multidimensional Gaussian random walks as data sets for performance tests. The steps between subsequent items were generated independently for each coordinate by using the Gaussian random distribution with zero mean and standard deviation of 0.1.

In the first setting (Figure 2), runtime is obtained as a function of input data size for running `clustering.sc.dp()`. The size of the input varies from 1,000 to 30,000 with a step size of 1,000. The input data consists of two-dimensional vectors. The number of clusters is set to 2. The runtime increases quadratically in the number of items to be clustered.

Table 1 presents some runtime data for a different magnitude of the number of items in order to

| Size | 1000 | 10,000 | 100,000 | 1,000,000 |
|---|---|---|---|---|
| Runtime (sec) | 0.03 | 1.19 | 144.00 | 14,479.88 |

**Table 1:** Runtime for different numbers of the items to be clustered.

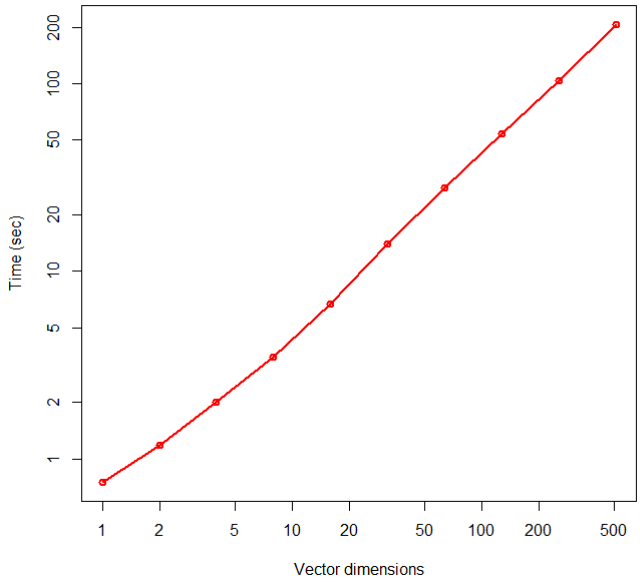


**Figure 3:** Runtime as a function of the dimension of the items to be clustered.

find performance bounds of the method. One can see that the optimal clustering was found very quickly if the number of items is 1,000, it took more than a second, almost two and a half minutes, about four hours for 10,000, 100,000 and one million items, respectively.

In the second performance test we examine the dependency of the runtime on the dimensions. All input data sets are of the same size 10,000 and the dimensions of the vectors varies from 1 to 512. The dimensions are doubled in each run. Figure 3 shows the results. The algorithm runs less than a second if the input contains one-dimensional vectors (i.e., scalars). The runtime increases linearly with the dimension and it can be solved within two and a half minutes if the dimension of the processed vectors is 512.

In the third performance test, the runtime is examined as a function of the number of clusters. The number of clusters is between 1 and 25. The input data set consist of 10,000 two-dimensional vectors. The black line with circles in Figure 4 shows the result. The runtime increases linearly with the number of clusters.

Finally, we compared the runtime of different functions within the package. The input data were the same as in the previous setting. One can see in Figure 4 that the runtime of `findwithinss.sc.dp()` is slightly larger than the one of `clustering.sc.dp()`. The runtime of `backtracking.sc.dp()` remains small even for large cluster numbers. The package has three typical ways of usage:

- `clustering.sc.dp()` can be called when the number of clusters is known in advance.
- `findwithinss.sc.dp()` can be called first and then `backtracking.sc.dp()` for a selected number of clusters.
- `findwithinss.sc.dp()` can be called first and then `backtracking.sc.dp()` for all possible number of clusters.

The subsequent call of `findwithinss.sc.dp()` and `backtracking.sc.dp()` is only slightly slower than calling `clustering.sc.dp()`. Furthermore, the total runtime of calling `findwithinss.sc.dp()` for cluster number $k$ and `backtracking.sc.dp()` for all cluster numbers less than or equal to $k$ is less than the double of the runtime of `clustering.sc.dp()` called for the same cluster number. This is much faster than calling the clustering function $k$ times which would be required without splitting `clustering.sc.dp()` into phases of finding the optimal *withinss* (`findwithinss.sc.dp()`) and backtracking (`backtracking.sc.dp()`).

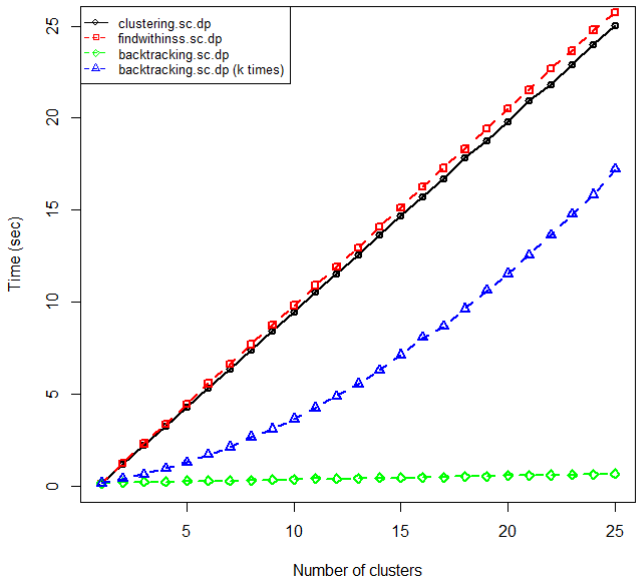


**Figure 4:** Comparison of runtime as a function of number of clusters between different functions provided by package **clustering.sc.dp**: `clustering.sc.dp()`, `findwithinss.sc.dp()`, `backtracking.sc.dp()`. It also contains the total running time of `backtracking.sc.dp()` when it is called for all cluster numbers less than or equal to the specific cluster number.

## Introduction to the R package clustering.sc.dp

R package **clustering.sc.dp** offers functions to perform optimal clustering on multidimensional data with sequential constraint. Method `clustering.sc.dp()` can find the optimal clustering if the number of clusters is known. Otherwise, methods `findwithinss.sc.dp()` and `backtracking.sc.dp()` can be used.

The following examples illustrate how to use the package. Function `clustering.sc.dp()` outputs the same fields describing clustering as `Ckmeans.1d.dp()` does in the original package:

- *cluster*: a vector of cluster indices assigned to each element in $x$. Each cluster is indexed by an integer from 1 to $k$.
- *centers*: a matrix whose rows represent the vectors of cluster centres (the average of the points within the cluster).
- *withinss*: the within-cluster sum of squared distances for each cluster.
- *size*: a vector containing the number of points in each cluster.

Figure 5 visualizes the input data and the clusters created by `clustering.sc.dp()`. See below for the R source code of the example.

```
# Example1: clustering data generated from a random walk
x <- rbind(0, matrix(rnorm(99 * 2, 0, 0.1), nrow = 99, ncol = 2))
x <- apply(x, 2, cumsum)

k <- 2
result <- clustering.sc.dp(x, k)
plot(x, type = "b", col = result$cluster)
points(result$centers, pch = 24, bg = 1:k)
```

The next example demonstrates the usage of functions `findwithinss.sc.dp()` and `backtracking.sc.dp()`. Similarly to the previous example, it also processes data of a random walk. Function `findwithinss.sc.dp()` finds optimal *withinss* for a range of cluster numbers. It returns a list with two components:

- *withinss*: a vector of total within-cluster sum of squared distances of the optimal clusterings for each number of clusters less than or equal to `k`.
- *backtrack*: backtrack data used by `backtracking.sc.dp()`.

In our example, the first cluster number where *withinss* drops below a threshold is selected as the number of clusters. Function `backtracking.sc.dp()` outputs the optimal clustering for the selected

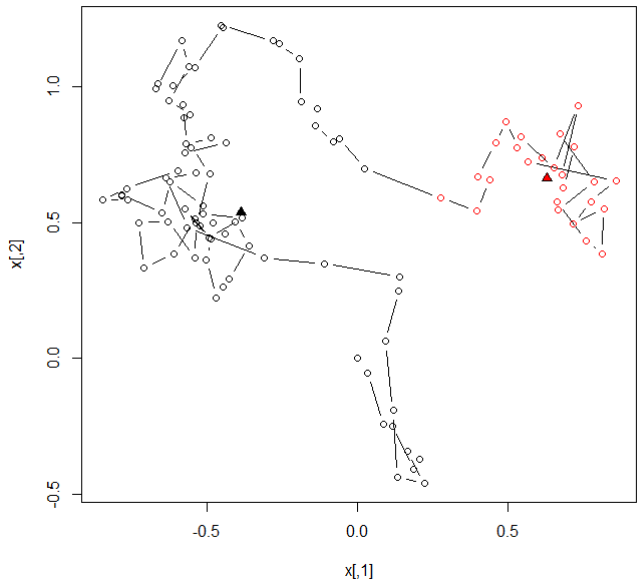


**Figure 5:** Clustering data representing a two-dimensional random walk into two clusters. The points to be clustered are represented by circles. The clusters are indicated using different colours. The cluster centres are denoted by triangles.

cluster number in the same format as `clustering.sc.dp()` does without running the whole clustering process again (Figure 6).

```
# Example2: clustering data generated from a random walk with small withinss
x <- rbind(0, matrix(rnorm(99 * 2, 0, 0.1), nrow = 99, ncol = 2))
x <- apply(x, 2, cumsum)

k <- 10
r <- findwithinss.sc.dp(x, k)

# select the first cluster number where withinss drops below a threshold
k_th <- which(r$twithinss <= 5.0)[1]

# backtrack
result <- backtracking.sc.dp(x, k_th, r$backtrack)
plot(x, type = "b", col = result$cluster)
points(result$centers, pch = 24, bg = 1:k_th)
```

## Summary

Clustering data with sequential constraint is a polynomial time solvable variant of the clustering problem. The paper introduced a package implementing a dynamic programming approach that finds the exact optimum of the problem. The algorithm represents an extension of the one-dimensional dynamic programming strategy of **Ckmeans.1d.dp** to multiple dimensional spaces which has been an open problem in the paper of Wang and Song (2011). The package supports both cases when the exact number of clusters is given and when the number of clusters is not known in advance. It can also be used to evaluate approximation algorithms for clustering with sequential constraint due to its optimality. The runtime evaluations indicate how fast the algorithm can solve problems with different sizes and parameters. Our future plan is to use the dynamic programming method in video summarisation.

## Acknowledgements

Research is supported by the Hungarian National Development Agency under grant HUMAN_MB08-1-2011-0010.

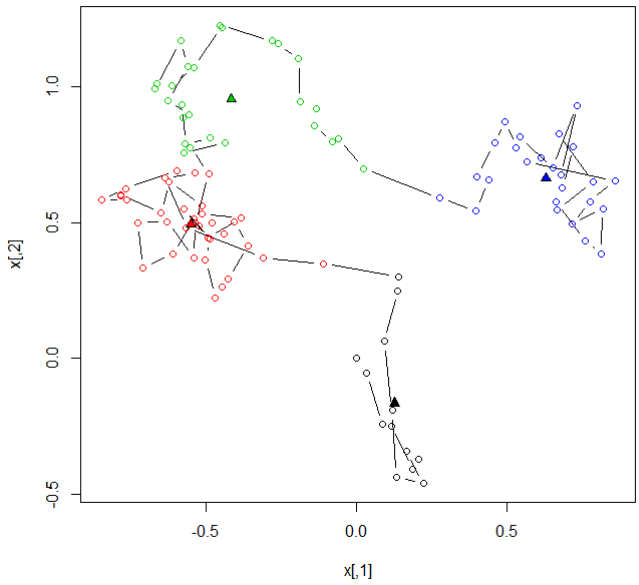


**Figure 6:** Clustering data representing a two-dimensional random walk. The number of clusters was determined by the analysis of optimal *withinss* for a range of cluster numbers. It uses the same notation as Figure 5.

## Bibliography

D. Aloise, A. Deshpande, P. Hansen, and P. Popat. NP-hardness of Euclidean sum-of-squares clustering. *Machine Learning*, 75(2):245–248, Jan. 2009. [p319]

R. Bellman. On the approximation of curves by line segments using dynamic programming. *Communication of the ACM*, 6(6):284, 1961. [p319]

S. Dasgupta and Y. Freund. Random projection trees for vector quantization. *IEEE Transactions on Information Theory*, 55(7):3229–3242, July 2009. [p319]

E. Dimitriadou, S. Dolnicar, and A. Weingessel. An examination of indexes for determining the number of clusters in binary data sets. *Psychometrika*, 67(1):137–160, 2002. [p321]

J. A. Hatigan and M. A. Wong. Algorithm AS 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society C*, 28(1):100–108, 1979. [p321]

J. Himberg, K. Korpiaho, H. Mannila, J. Tikanmaki, and H. Toivonen. Time series segmentation for context recognition in mobile devices. In *Proceedings of the IEEE International Conference on Data Mining, 2001 (ICDM 2001)*, pages 203–210, 2001. [p319]

A. K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 31(8):651–666, 2010. [p318]

L. A. Leiva and E. Vidal. Warped k-means: An algorithm to cluster sequentially-distributed data. *Information Sciences*, 237:196–210, July 2013. [p319]

S. Lloyd. Least squares quantization in PCM. *IEEE Transations on Information Theory*, 28(2):129–137, 1982. [p319]

M. Mahajan, P. Nimbhorkar, and K. Varadarajan. The planar k-means problem is NP-hard. In *Proceedings of the 3rd International Workshop on Algorithms and Computation (WALCOM '09)*, pages 274–285, Berlin, Heidelberg, 2009. Springer-Verlag. [p319]

G. W. Milligan and M. C. Cooper. An examination of procedures for determining the number of clusters in a data set. *Psychometrika*, 50(2):159–179, 1985. [p321]

M. Song and H. Wang. Ckmeans.1d.dp: Optimal k-means clustering for one-dimensional data. *R package version 3.02*, 2011. URL http://CRAN.R-project.org/package=Ckmeans.1d.dp. [p318]

T. Szkaliczki and J. Song. clustering.sc.dp: Optimal distance-based clustering for multidimensional data with sequential constraint. *R package version 1.0*, 2015. URL http://CRAN.R-project.org/package=clustering.sc.dp. [p318]

P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison Wesley, 2006. [p318]

E. Terzi. Efficient algorithms for sequence segmentation. In *Proceedings of the Sixth SIAM International Conference on Data Mining*, pages 314–325, 2006. [p319]

S. Tierney, J. Gao, and Y. Guo. Subspace clustering for sequential data. In *Proceedings of the IEEE Computer Conference on Computer Vision and Pattern Recognition*, pages 1019–1026, 2014. [p319]

H. Wang and M. Song. Ckmeans.1d.dp: Optimal k-means clustering in one dimension by dynamic programming. *The R Journal*, 3(2):29–33, 2011. [p318, 319, 320, 325]

J. H. Ward, Jr. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244, 1963. [p319]

*Tibor Szkaliczki*
*eLearning Department*
*Institute for Computer Science and Control, Hungarian Academy of Sciences*
*Hungary*
szkaliczki.tibor@sztaki.mta.hu