Appendix Contents

A Dataset Details

As stated in Section 9.1, in our experiments, we consider 13 synthetic and real-world datasets commonly used as benchmarks for clustering tasks. The datasets we consider reflect diverse data distributions, clustering structures and application domains. Next, we describe each dataset in detail.

- The MNIST dataset [35], available via PyTorch [45] or ClustPy [37], is a collection of 28×28 (vectorized) images of handwritten grayscale digits. We draw a stratified subsample of 25000 images, ensuring that the original class proportions across the ten digit clusters are preserved. We rescale all pixel values by dividing by the maximum.
- The Double MNIST dataset is derived from MNIST by horizontally concatenating pairs of digit images. Each sample is a 28×56 (vectorized) grayscale image obtained by placing one 28×28 digit in the left position and another in the right position, and the label encodes the ordered pair of digits, yielding 100 clusters in total. For our experiments, we generate 10000 such composite images using the procedure described above, resulting in an approximately uniform distribution over all digit pairs. Unlike the MNIST dataset, the Double MNIST dataset by construction admits a natural clustering with a Khatri-Rao structure. We rescale all pixel values by dividing by the maximum.
- The HAR dataset, available via the UCI repository [29] or ClustPy [37], consists of sensor readings collected from smartphone accelerometers and gyroscopes during human activity monitoring. There are 10299 data points. Each data point is represented as a multivariate feature vector of dimension 561 derived from raw time-series measurements, and the dataset includes 6 activity clusters such as walking, standing and sitting. We standardize each feature by subtracting its mean and dividing by its standard deviation.
- The Olivetti Faces dataset, available via ClustPy [37] or scikit-learn [46] contains (vectorized) grayscale facial images of 40 individuals, with 10 images per subject captured under varying lighting conditions, facial expressions, and poses. Each image has resolution 64×64 pixels. We standardize each feature by subtracting its mean and dividing by its standard deviation.
- The CMU Faces dataset, available via the UCI repository [29] or ClustPy [37], contains 624 grayscale facial images of 20 persons varying their pose (up, straight, left and right) and expression (neutral, happy, sad, angry), and shown with and without sunglasses. The original images with a resolution of 30×32 pixels are vectorized and each feature is standardized by subtracting its mean and dividing by its standard deviation.
- The Symbols dataset, available via ClustPy[6] [37], consists of vectorized handwritten symbols. Each sample corresponds to a time series obtained from the drawing trajectory of a symbol. The symbols are drawn by 13 different individuals. The dataset contains a total of 1020 data points, each with 398 measurements. The data points are organized into 6 natural clusters. We standardize each feature by subtracting its mean and dividing by its standard deviation.
- The stickfigures dataset [19], available via ClustPy [37], consists of 900 synthetic (vectorized) silhouette images of human stick figures generated under varying poses. Each image has a resolution of 20×20 pixels. Examples of such images are given in Figure 1. Each image is provided at a fixed resolution and captures a simplified body configuration defined by joint positions and limb orientations, resulting in 9 distinct pose-based clusters. We rescale all pixel values by dividing by the maximum.
- The optdigits dataset, available via the UCI repository [29] or ClustPy [37], contains 5620 (vectorized) handwritten digit images represented as 8×8 grayscale pixel grids. We standardize each feature by subtracting its mean and dividing by its standard deviation.
- The Classification dataset, sourced by scikit-learn [46], is a synthetic dataset. While it was originally designed for benchmarking classifiers, it is also useful for evaluating clustering algorithms which simply discard class labels until the evaluation of the clustering results. Each class for classification corresponds to a cluster. Except in the experiments where we vary the number of data points, features, or clusters, the dataset consists of 5000 samples organized into 100 clusters. The data are generated with 10 informative features, with no redundant or repeated features, ensuring that all dimensions contribute meaningfully to class separability. We standardize each feature by subtracting its mean and dividing by its standard deviation.
- The Chameleon dataset, available via the clustbench Benchmark Suite [16], consists of 10000 two-dimensional point clouds exhibiting complex, nonconvex cluster shapes with varying densities [28]. The dataset corresponds to the configuration shown at the bottom of Figure 4, augmented with a substantial proportion of uniformly distributed data points that do not belong to any natural cluster.
- The Soybean Large dataset, available via the UCI repository [29] or ClustPy [37], is a dataset of categorical features. It consists of 562 plant samples belonging to one of 15 classes. For each plant, there are 35 observed categorical attributes describing the plant. After encoding categorical attributes numerically, we standardize each feature by subtracting its mean and dividing by its standard deviation.
- The Blobs dataset, sourced by scikit-learn [46], is a synthetic dataset specifically designed for controlled evaluation of clustering algorithms that consists of data points grouped around isotropic Gaussian clusters. Each cluster has standard deviation 1. The dataset consists of 5000 2-dimensional data points, arranged in 100 clusters, except in the experiments where we vary the number of data points, features and clusters. We standardize each feature by subtracting its mean and dividing by its standard deviation.
- The R15 dataset, available via the clustbench benchmark suite [16], consists of 600 two-dimensional data points forming 15 Gaussian clusters. The cluster centroids in R15 are not arranged on a regular grid but exhibit non-uniform

[6] The dataset is also available at: https://www.timeseriesclassification.com/index.php

inter-cluster distances. We standardize each feature by subtracting its mean and dividing by its standard deviation.

B Implementation Details

In this section, we discuss implementation details for all the algorithms considered in the experiments presented in Section 9.

Implementation details of the naïve approach to Khatri-Rao clustering. For the naïve Khatri-Rao clustering baseline used in our experiments, we first run the scikit-learn [46] implementation of standard k-Means to extract the desired number of cluster centroids. For instance, if the goal is to extract two sets of h_1 and h_2 protocentroids, k-Means retrieves h_1h_2 clusters. Then, using a coordinate-descent procedure implemented in Python with closed-form updates, we decompose a set of centroids into two smaller sets of protocentroids whose Khatri–Rao product approximates the original centroids. The coordinate-descent procedure alternates between updating the protocentroids of the first and second set, holding the other set fixed. At each step, we use the closed-form updates illustrated in Equation (8) in Section 5. The procedure stops either when a maximum number of iterations is reached (5000 by default) or when the total sum of squared differences between the initial centroids and the Khatri-Rao aggregation of the protocentroids becomes smaller than a user-specified threshold (10^{-4} by default). Upon termination of the described coordinate-descent procedure, we have the output sets of protocentroids. The corresponding centroids are readily obtained by aggregating protocentroids. To conclude, we assign each data point to the closest centroid. The implementation is available online in our code repository [7].

Implementation details of standard k-Means. We rely on the well-established scikit-learn implementation of standard k-Means. The two k-Means baselines (namely k-Means with $h_1 + h_2$ centroids and h_1h_2 centroids) are obtained by simply specifying the number of centroids $h_1 + h_2$ and h_1h_2 as input. In the scalability experiments, instead, to ensure a fair comparison, we use an implementation of k-Means which mirrors the implementation of Khatri-Rao-k-Means described next.

Implementation details of Khatri-Rao-k-Means. Our experiments rely on a simple Python implementation of Khatri-Rao-k-Means that is purely built on NumPy [8], taking advantage of vectorized operations for efficiency. Such implementation is available online [9].

For initialization, by default we sample random data points as the initial protocentroids (as in Algorithm 1). Alternatively, we can adopt the strategy inspired by k-Means++ described in Section 6, which selects representative data points based on distance criteria. Our implementation of this strategy either deterministically chooses the data point farthest from the previously selected centroids, or samples data points with probability proportional to their distance from those centroids. After initialization, we iteratively compute assignments, protocentroids and corresponding centroids. Thanks to the closed-form updates introduced in Section 6, the updates of protocentroids (and centroids) are implemented in a fully-vectorized manner. At each iteration we monitor convergence by tracking the movement of all reconstructed centroids; the algorithm terminates once this movement falls below a user-specified threshold or a maximum number of iterations is reached.

During the execution of the algorithm, in case empty clusters arise, they are handled by reinitializing the corresponding protocentroid to a random data point.

Our implementation is deliberately simple and easy to follow. In the future, more optimized or parallelized implementations could be developed for larger-scale settings.

Khatri-Rao-k-Means admits a time-efficient and a memory-efficient implementation. Algorithm 1 presents the memory-efficient implementation that avoids storing the full set of centroids by computing them on the fly from the stored set of protocentroids. This approach can reduce memory requirements, particularly when the number of clusters is large since memory requirements only grow additively with the total number of protocentroids instead of multiplicatively. For example, h_1h_2 clusters demand storing only $h_1 + h_2$ protocentroids instead of h_1h_2 centroids. However, computing centroids on the fly incurs a runtime overhead. A time-efficient implementation is obtained by computing centroids once and storing them.

Implementation details of standard deep clustering algorithms. For DKM and IDEC, we rely on the off-the-shelf PyTorch-based implementations provided by the ClustPy library [10]. To find initial centroids, DKM and IDEC use the scikit-learn implementation of standard k-Means.

Implementation details of Khatri-Rao deep clustering algorithms. For the implementation of Khatri-Rao deep clustering algorithms, we build the implementation of Khatri-Rao DKM and Khatri-Rao IDEC on top of the ClustPy implementations of the corresponding standard deep clustering algorithms.

Extending the ClustPy implementation of a standard deep clustering algorithm to the Khatri-Rao clustering paradigm is straightforward. During initialization, we rely on our implementation of Khatri-Rao-k-Means to obtain initial protocentroids. We then reparameterize the centroids to satisfy the Khatri-Rao structure and adjust the autoencoder parameters to conform to the Hadamard-decomposition reparameterization. Our implementation of Khatri-Rao deep clustering algorithms is available online [11].

C Proofs

In this section, we collect the proofs of the propositions stated in the paper.

Proof of Proposition 6.1.

Proof. We illustrate the proof for the product aggregator. Consider the j-th protocentroid in the first set of protocentroids. The optimal update for this protocentroid satisfies

$$\boldsymbol{\theta}_1^j = \arg\min_{\boldsymbol{\theta}} \sum_{l=1}^{h_2} \sum_{\mathbf{x}\in C_{j,l}} (\mathbf{x} - \boldsymbol{\theta} \odot \boldsymbol{\theta}_2^l)^2.$$

Therefore $\boldsymbol{\theta}_1^j$ can be found by computing the gradient of this sum of squared differences with respect to $\boldsymbol{\theta}$ and equating it to the zero vector $\mathbf{0}$. Doing so, one obtains

$$-2 \sum_{l=1}^{h_2} \sum_{\mathbf{x}\in C_{j,l}} (\mathbf{x} - \boldsymbol{\theta} \odot \boldsymbol{\theta}_2^l)\boldsymbol{\theta}_2^l = \mathbf{0},$$

[7] https://github.com/maciap/KhatriRaoClustering/blob/main/scripts/KRkmeansExperimentsLib.py
[8] https://numpy.org
[9] https://github.com/maciap/KhatriRaoClustering/tree/main/KathriRaokMeans
[10] https://github.com/collinleiber/ClustPy
[11] https://github.com/maciap/KhatriRaoClustering/tree/main/KhatriRaoDeepClustering

which holds if and only if:

$$\boldsymbol{\theta}_1^j = \frac{\sum_{l=1}^{h_2} \sum_{\mathbf{x} \in C_{j,l}} \mathbf{x} \odot \boldsymbol{\theta}_2^l}{\sum_{l=1}^{h_2} |C_{j,l}| \boldsymbol{\theta}_2^l \odot \boldsymbol{\theta}_2^l,}.$$

The derivations for the second set of protocentroids are symmetric, and the proof for the sum aggregator is also similar. □

Proof of Proposition 8.1.

Proof. Assuming p protocentroid sets of equal size and exact budget usage, each set contains $h = \frac{b}{p}$ protocentroids, so the total number of centroids that can be represented is:

$$\left(\frac{b}{p}\right)^p.$$

Maximizing this function over $p > 0$ is equivalent to maximizing its natural logarithm

$$\log\left(\frac{b}{p}\right)^p = p\log\left(\frac{b}{p}\right) = p\log b - p\log p.$$

Differentiating with respect to p yields

$$\log\left(\frac{b}{p}\right) - 1,$$

which is positive for $p < \frac{b}{e}$ and negative for $p > \frac{b}{e}$. Moreover, the second derivative

$$-\frac{1}{p} < 0 \qquad (p > 0)$$

shows that the log-objective is strictly concave, and therefore attains a unique maximum at $p = \frac{b}{e}$ in the continuous domain.

When restricting to the divisor-only setting, the optimal value of p, p^{max}, is required to be an integer that exactly divides b, so that $h^{max} = \frac{b}{p^{max}}$ is an integer. Since the objective is strictly increasing for $p < \frac{b}{e}$ and strictly decreasing for $p > \frac{b}{e}$, among all admissible values the maximum is attained at one of the two values of p that are immediately below or above $\frac{b}{e}$ (when each exists). All other admissible divisors are necessarily further away from $\frac{b}{e}$, and therefore yield a strictly smaller objective value. □

Proof of Proposition 8.2.

Proof. We present the proof for the product aggregator. However, extension to different aggregator functions is straightforward.

First, to see why it must be $p^* \geq \log_{h_{min}} k$, notice that, if $p^* < \log_{h_{min}} k$ then $h_{min}^{p^*} < k$, and $h_{min}^{p^*}$ is the maximum number of centroids that can be represented using p^* sets of at least h_{min} protocentroids. As regards the other inequality, to prove that $p^* \leq \lceil \frac{k}{h_{min}-1} \rceil$, it is sufficient to construct an example where all sets have a protocentroid with all entries equal to 1 and the remaining at least $h_{min} - 1$ protocentroids that are equal to centroids. Different sets contain different centroids. In the illustrated scenario, each set of protocentroids represents exactly at least $h_{min} - 1$ centroids. Thus, $\lceil \frac{k}{h_{min}-1} \rceil$ sets of protocentroids can always represent k centroids. □