UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

B3: An efficient approximation to the K-means clustering for massive data

Victor Badenas Crespo

Unsupervised and Reinforcement Learning
Master in Artificial Intelligence (FIB-MAI)

June 1st 2021

Contents

Chapter 1

Introduction

This project consists on the implementation of the paper [1], which was developed by Marco Capó and Aritz Pérez and Jose A. Lozano in the Knowledge-Based Systems journal. The paper aims to approximate the performance of KMeans, one of the most widely used clustering methods for massive datasets, while reducing the number of distance computations required to achieve a good clustering solution. The solution proposed by the authors is based on the recursive partition of the dataset following an heuristic that they call Recursive Partition based K-Means.

1.1 Definitions

In their paper, the authors start by defining mathematically a partition, subset and the cardinality of the subset and the center of mass. They define a partition P as a collection of disjoint subsets S which the union of them results in the whole dataset D. They define the weight of the subset as the cardinality of it $|S|$ and the center of mass as the mean of all the points in the subset $\bar{S} = \frac{\sum_{x \in S} x}{|S|}$.

1.2 Proposed solution

The solution proposed by the authors uses partitions as the units for fitting weighted Lloyd algorithms instead of using the data points from the dataset. The dataset is split into subsets each of them is represented by a representative sample and the cardinality of the subset.

The authors propose to divide the space where the dataset lies and partition it recursively using a quadtree, where the entire space of the subset is divided into 2^d subspaces. In a $2D$ space, the plane would be divided into quadrants, then each of the quadrants would be divided into 4 "subquadrants" and so on. Thanks to kthis heuristic, the number of elements is controlled, always satisfying $|P| < n$ or in other words, that the length of the partition ant every given moment is smaller than the number of instances in the dataset.

In 1.1 we show a $2D$ gaussian mixture with 3 gaussian and an overlap of 5%. Each partition has the subsets in different colors to be able to discriminate them easily. Each partition contains subsets thinner than the previous one, and the subsets of the partition P_{i+1} are generated by spliiting subsets from the partition P_i. The authors define that a partition P_i is thinner than P_j if every subset of P_i can be written as the union of subsets of P_j which applies in the case of the RPKM algorithm.

The authors propose an approach where for each one of the partitions mentioned above, a weighted Lloyd (WL) algorithm is trained and the centers extracted in each of the iterations is used

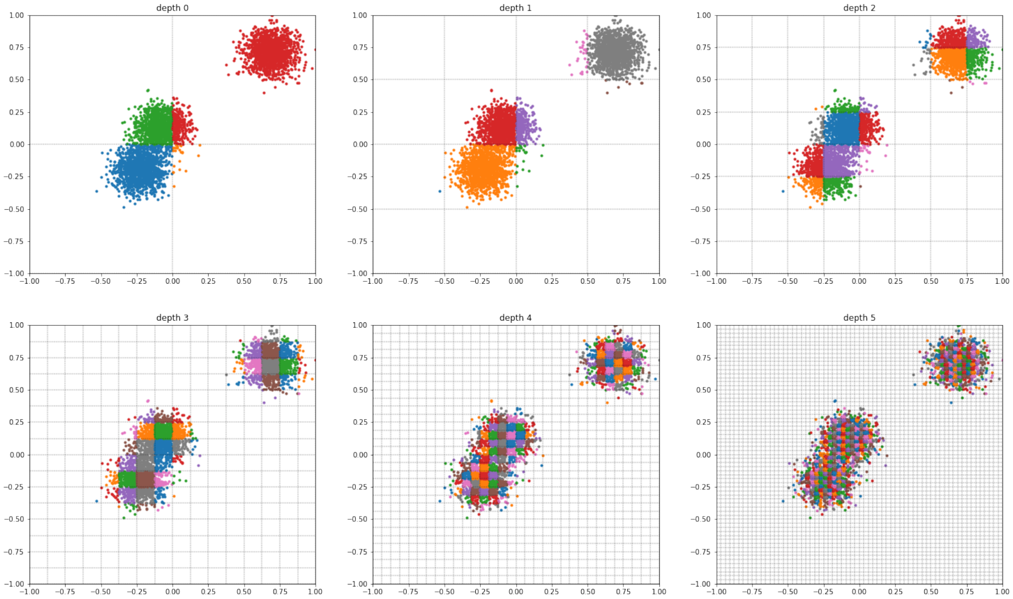


Figure 1.1: Partition example for a 2d gaussian mixture

as initialization for the next partition. This drastically reduces the number of distance computations that the algorithm realizes as the number of representative samples that will be used to fit the WL algorithm is always smaller than n.

Algorithm 1: WL algorithm

```
Input: Set of representatives $\{\bar{S}\}_{S \in P}$ and weights $\{|S|\}_{S \in P}$, for the partition $P$. Number
       of clusters K and initial set of centroids $C_0$.
Output: Set of centroids $C_r$ and the Corresponding clustering pattern $G_r$
$G_0 \leftarrow C_0; r = 0;$
while not Stopping Condition do
    $r = r + 1;$
    $C_r \leftarrow G_{r-1};$
    $G_r \leftarrow C_r;$
end
return $C_r$ and $G_r$
```

The weighted Lloyd algorithm is shown in 1 and it shows that given a set of representatives, a set of weights, a number of clusters and an initial set of centroids C_0, it iteratively improves the centers by assigining the weighted mean of all the instances in a group or cluster as their centroid and then computing the group of each instance as the center index that is closer to each instance.

Algorithm 2: RPKM algorithm

```
Input: Dataset D, number of clusters K, maximum number of iterations m
Output: Set of centroids approximation $C_i$
Compute the set of weights and representatives of the sequence of thinner partitions,
  $P_1, \ldots, P_m$, backwards. ;
i := 1;
while not Stopping Criterion do
    Update the centroid's set approximation, $C_i = \{c_j^i\}_{j=1}^K$:
      $C_i = WL(\{\bar{S}\}_{S \in P_i}, \{|S|\}_{S \in P_i}, K, C_{i-1})$;
    $i = i + 1$
end
return $C_i$
```

The RPKM algorithm is exposed in 2. In the algorithm we can appreciate that the autors compute all the weights and representatives for the given dataset D and then iteratively update the centers for each iteration by performing a WL computation on the representative samples and cardinality of the subsets. Additionally, they use the cluster centers from the previous RPKM iteration as initialization.

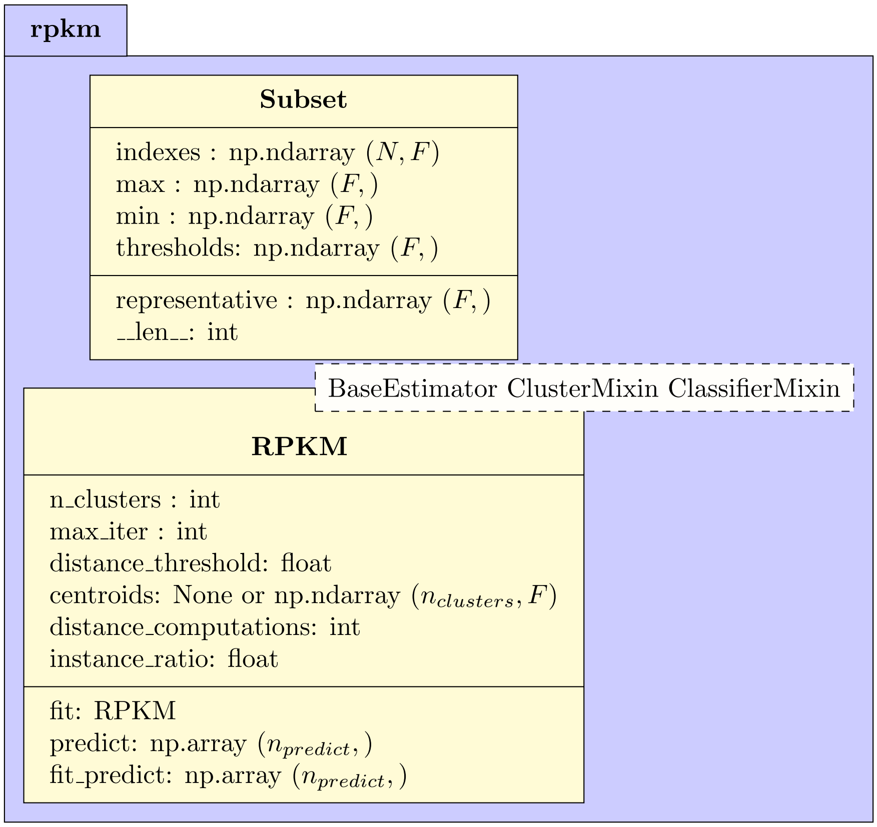
Both algorithms hav similar stop criteria, a threshold on the distance between the centers of the previous and new iteration and a maximum number of iterations. However, the authours have removed the distance limitation in their implementation in order to see the effects of the algorithm fully.

Chapter 2

Implementation

2.1 Implementation

The algorithm was created in python 3 using scikit-learn coding guidelines and API conventions. The Classes will be defined as shown in 2.1 UML diagram.


rpkm
Subset
indexes : np.ndarray (N, F)
max : np.ndarray $(F,)$
min : np.ndarray $(F,)$
thresholds: np.ndarray $(F,)$
representative : np.ndarray $(F,)$
__len__: int
BaseEstimator ClusterMixin ClassifierMixin
RPKM
n_clusters : int
max_iter : int
distance_threshold: float
centroids: None or np.ndarray $(n_{clusters}, F)$
distance_computations: int
instance_ratio: float
fit: RPKM
predict: np.array $(n_{predict},)$
fit_predict: np.array $(n_{predict},)$

Only the public methods are specified in the diagram. N in the diagram represents the number of instances in a subset. The F value indicates the number of features in the dataset. $n_{predict}$ represents the number of instances to be predicted in the inference dataset.

2.2 Subset

Subset would be a struct in other programming languages but in python it is defined as a class. The class contains:

1. **The indexes of the original data in this subset.** This is done in order to store pointers to a data structure instead of cloning the memory on the numpy arrays, as for massive datasets this can lead to memory issues.

2. **The minimum and maximum range** per each of the dimensions on the dataset and the thresholds used to build the next partition from this subset.

3. The threshold t is defined as the value that divides the range in to equal size subspaces for each dimension $\overrightarrow{t} = \frac{\overrightarrow{max}+\overrightarrow{min}}{2}$.

4. A method to compute the **cardinality**. In this case python's __len__ method is used to return the length of the index array.

5. A method to compute the **representative** for the cluster. As mentioned before, this is computed as the mean of the instances in the subset. The *np.mean* function was used to perform the operation effectively.

The Subset class has been created to follow an obejct oriented approach to the data structures.

2.3 RPKM

The RPKM algorithm was implemented in another class which inherits from *sklearn's BaseEstimator , ClassifierMixin and ClusterMixin* classes. The *BaseEstimator* is the main base class that all sklearn estimators inherit from. it provides parameter setters and getters for the class as well as a string representation of the class as well as some utility methods.

1. **set_params(**params)**: Set the parameters of this estimator.

2. **get_params()**: Get parameters for this estimator.

3. **__repr__()**: String representation of the estimator

4. **_validate_data(X)**: Validate input data and set or check the *n_features_in_* attribute, also of BaseEstimator.

The ClassifierMixin provides methods for all classifiers in the scikit-learn package. It implements the **score(X, y)** method to predict the labels for the data X and then compute the accuracy wrt y.

The ClusterMixin provides methods for all clustering methods in the scikit-learn package. It implements the **fit_predict(X, y=None)** method to fit the estimator and return the labels for the training data.

In the following subsections we will explain parts of RPKM's fit method that implements the method described by the authors to recursively partition the feature space into hypercubes.

2.3.1 fit

The fit method is the responsible of computing the partition and it's subsets for each RPKM iteration and apply a WL algorithm in every step. The algorithm in pseudocode is presented in 3.

Algorithm 3: RPKM Fit

```
Input: Dataset D, number of clusters K, maximum number of iterations m
Output: Self
reset centers;
Initialize $P_0 = \{D\}$;
i := 0;
while $i < m$ do
    create next partition (_create_partition);
    compute $R$ and cardinality of the subsets (_compute_partition_meta);
    if $|P_i| < K$ then
        there are not enough partitions to initialize clusters.;
        continue;
    else if centroids not initialized then
        choose K random representatives as cluster centers;
    else if $|P_i| >= K$ then
        there are as many subsets as samples, no point in continuing.;
        break;
    update clusters with WL (_cluster);
    i += 1;
end
return Self
```

The fit algorithm contains the main loop for the RPKM algorithm. One notable difference between the proposal of the authors and the implementation presented in this work is the change in the partition creation. The authors propose the creation of the partitions outside of the loop, creating always m partitions of the dataset. The method proposed here is to generate only the partitions that we require by integrating the creation of the partitions in the while loop. By doing this we are only computing the partitions that are required for the run (in case we reach the number of instances earlier than the iteration m).

The fit method calls internally to 3 methods: **_create_partition(X, partition)**, **_compute_partition_meta(X, partition)**, and **_cluster(R, cardinality)**. Each of the methods will be now explained.

2.3.2 _create_partition

The create partition method is the one reponsible of, given a list of sets that constitute a partition, perform a binary partition on each subset and create a list containing the new subsets. For the sake of brevity, the binary partition methodology will be explained in this section.

The binary partition method segments the data given as an input that constutes a subset S of the previous iteration's partition P_{i-1}. The subsets are created by taking the input data and a set of thresholds defined as the values that partition a dimension in two equally sized hypercubes. This operation is done for each of the dimensions in the dataset. The operation is performed by the following numpy operation:

```
comps = X > subset.thresholds[None, :]
```

This operation will generate a numpy matrix with boolean values with shape (n_{subset}, n_{dim}) where each row will contain the binary representation of a numerical index assigned to each hypercube. This matrix will be converted to an array of integer indexes containing the assignation of each of the elements to a subset. The conversion from binary to int is performed by a custom function coded with numpy. The conversion functions from int to binary and binary to int are shown:

```
def inttobin(value, n_dim):
    return ((value & (2**np.arange(n_dim)))) > 0

def bintoint(value, n_dim):
    return (value*2**np.arange(n_dim)).sum(axis=1)
```

The binary to int funcion is quite self explainatory, generate a vector of powers of 2 and multiply by the binary vector *value*, then sum for all the indexes of the second dimension to obtain the integer representation. The integer to bin funcion is a little bit more complex. First a vector of powers of two is constructed, then a logical and operation with the integer value is performed and the positions that are not > 0 are set to 0 to construct the binary representation.

After the indexes are computed, the subsets initialized using the Subset struct described above, where the min, max and threshold arrays are modified to suit the new subset's instances. The range is modified in the following way:

If the binary representation for the cluster integer value has the value 1 in a dimension position, it means that the comparison performed before was true, so all instances are in the positive half of the spatial partitions. Because of that, the minimum for that dimension is updated to be the threshold value for that dimension. On the other hand, if the value is 0 for that dimension, the max for that dimension is set to be the threshold value for that dimension. Then the threshold values are updated to be the mean between the min and the max arrays. The subsets are then returned for the create partition function to concatenate them all and build the new partition.

2.3.3 _compute_partition_meta

The compute partition meta method is responsible for the generation of the representatives for each subset in the partition as well as the cardinality values for each subset. This method initializes the R vector to be of shape $(n_{subsets}, n_{features})$ and the cardinality array to be of shape $(n_{subsets},)$. Once the arrays are initialized it procedes to populate the arrays with the result of the **representative** function of the Subset class and the length of the Subset class respectively.

2.3.4 _cluster

The cluster method is in charge of updating the centroids of the RPKM algorithm by fitting a WL algorithm with the representatives as data points and the cardinality of the subsets as the weights. In the implementation, *scikit-learn's KMeans* object was used with the *n_iter* parameter set to 1 and giving as argument the weights into the fit method. It's also important to note that KMeans supports Lloyd's algorithm as well as Elkan optimization. In order to ensure the usage of Lloyd's algorithm, the *algorithm* parameter must be set to *full*. As initialization for the KMeans algorithm, the centroids of the previous RPKM iteration are passed. After the algorithm is fitted, the cluster centers are extracted by accessing the *cluster_centers_* attribute of KMeans.

2.4 fit results

Once the implementation was done, we demonstrate the operation of the algorithm. We will use a 2 dimensional gaussian mixture with 3 gaussian distributions. The RPKM algorithm was performed with the max iterations of $m = 6$. On the first figure 2.1 we represent the creation of the partitions. where the dataset comprised in the range -1, 1 for both dimensions. In the first iteration, the subsets are the 4 quadrants of the subspace, on the next iteration, each quadrant is divided in 4, and so on. We represent each subset with different colors for better interpretability.

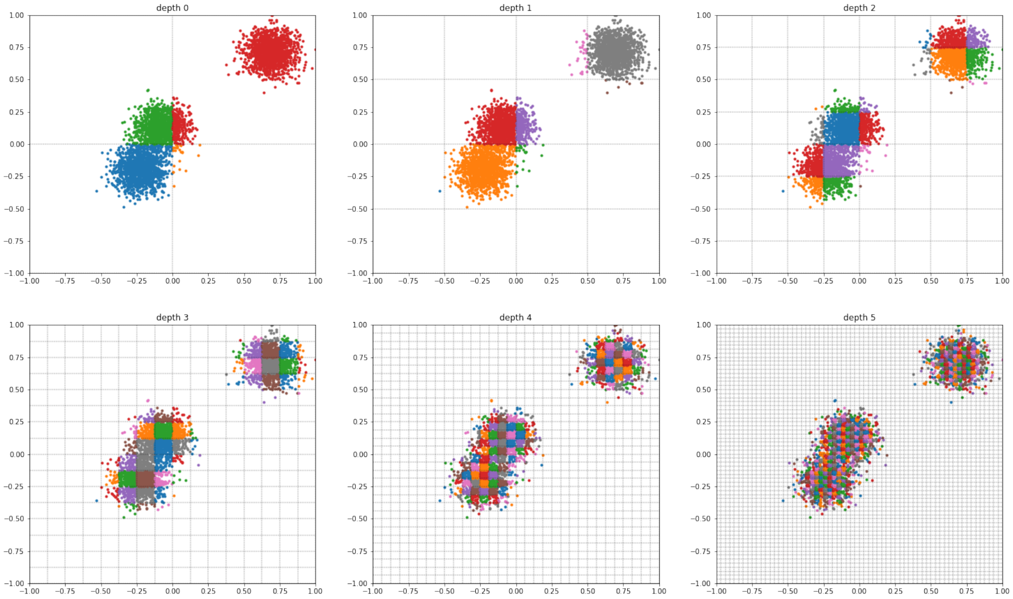


Figure 2.1: Partition example for a 2d gaussian mixture

In the next figure 2.2 we represent the data in light gray, the representative sampels for each of the subsets in black and the centers with more vivid colors.

Finally, the last figure 2.3 represent the evolution of the centers from iteration of iteration. And how the centers approximate more and more the centers of the gaussian distributions.

2.5 Artificial Dataset Generator

Additionally to the algorithm classes, a helper class able to generate datasets as specified in the paper was required to do the experiments with the artificial datasets. In the paper they specify that they create gaussian mixture datasets ensuring an overlap of less than 5%. A class was defined as the *Artificial Dataset Generator*. The class is defined in *./src/artificial_dataset_generator.py* and each time a new dataset is requested, a random set of centers in the feature space is created. Then the closest centers are detected and the overlap between gaussians is computed. Starting from an std value of 0.1, the value is modified until the maximum overlap between any two gaussians in the dataset is 0.05 and then the dataset is created with sklearn's *datasets.make_blobs*.

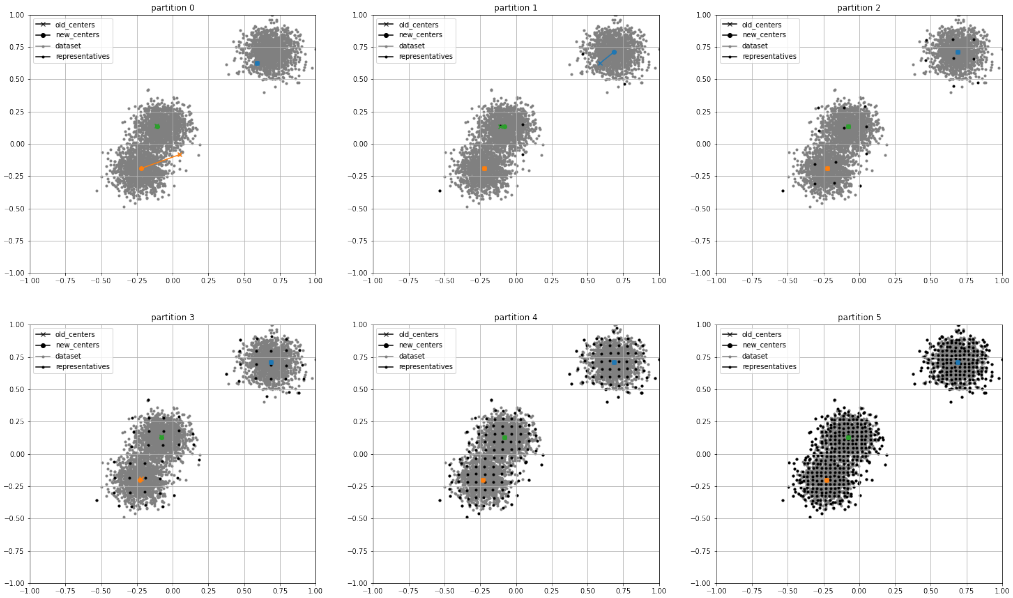


Figure 2.2: Centers and representatives in each iteration

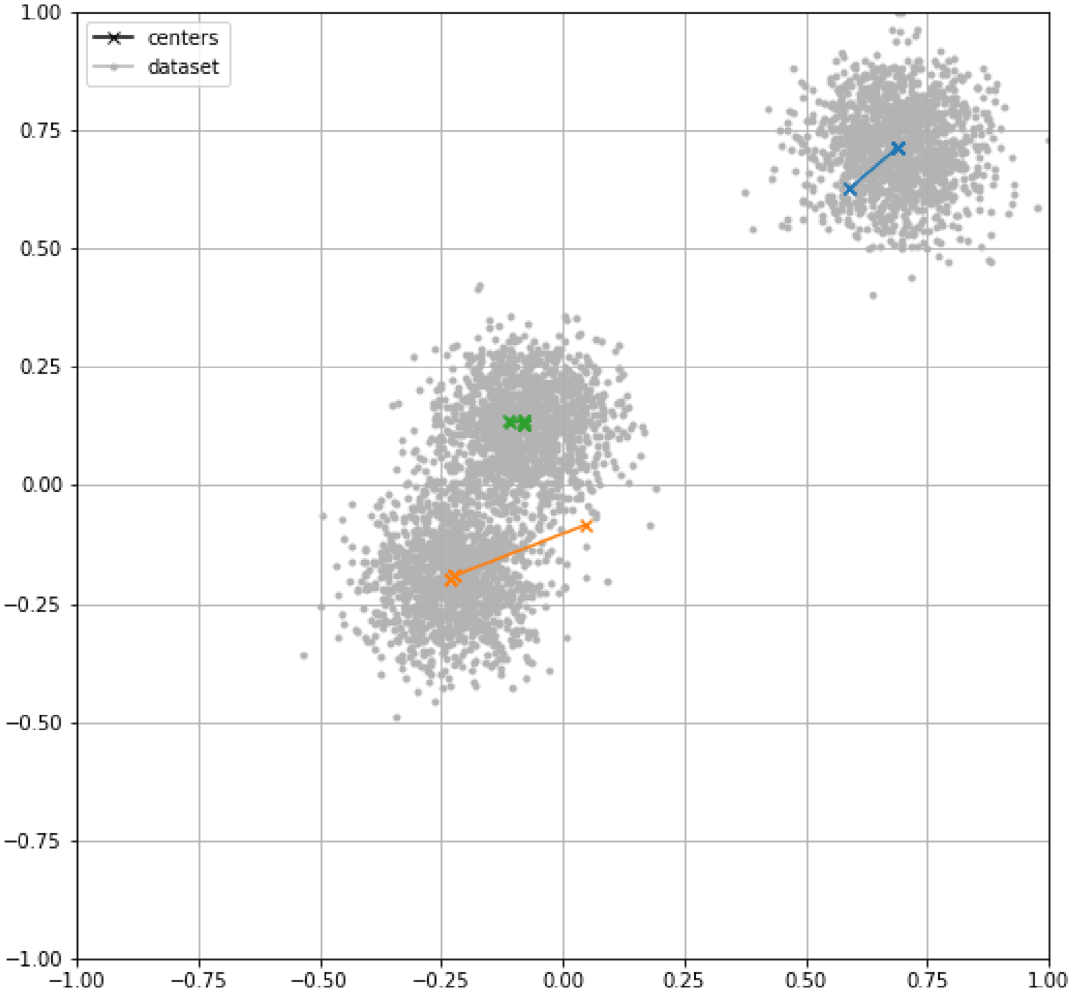


Figure 2.3: Center evolution

Chapter 3

Experiments

In this section we will reason and explain the experiments done with the algorithm to demonstrate the good performance and efficiency of the algorithm. The experiments were replicated from the paper. Six main experiments were performed, 3 with artificial datasets and 3 with real datasets.

The artificial datasets will be created using the Artificial Dataset Generator class mentioned in the previous section. The number of gausssians will be set to the same number of clusters to be detected in the clustering algorithms and the standard deviation will be set to ensure an overlap of around 5%. For the real dataset, we will use the gas-sensor dataset [2] from the UCI repository [3], particularly the ethylene_methane dataset which contains 4178504 instances and 16 features. In order to construct the dataset of the wanted size, a random number of instances will be selected and a random number of features will be selected as well acording to the parameters K and D that will be set in each of the experiments.

Three main experiments were performed with each dataset: The computation of distances comparison experiment, the standard error experiment and finally the instance ratio experiment.

3.1 Distance computation

The first experiment done was the distance comparison experiment. RPKM was evaluated against scikit-learn's KMeans initialized with the k-means++ algorithm and scikit-learn's MiniBatchKMeans with batch sizes of $b \in \{100, 500, 1000\}$. The RPKM algorithm was evaluated for $m \in [1 \dots 6]$. This experiment was replicated for the artificial and real datasets. In the case of artificial dataset, the distance computation values were averaged over 10 replicas of the dataset, meaning that 10 random datasets with the same number of clusters K, number of dimensions D and number of instances N are generated and then their results are averaged. In the case of the real dataset, 10 replicas were used as well. In this case, a replica corresponds to a random selection of D features and N instances from the dataset.

In the case of the RPKM algorithm the number of distance computations are calculated as:

$$d_{RPKM} = \sum_{i=1}^{m} niter_i * |R_i| * K$$

Or in other words, the distance computations nd at iteration i are the number of iterations for the WL algorithm at that iteration $niter_i$ times the number of representatives $|R_i|$ at that iteration times the number of clusters K. Then the distance computations for all RPKM iterations are added.

For the KMeans algorithm, the number of distances are computed by the following expression:

$$d_{kmeans} = d_{init} + d_{lloyd}$$

$$d_{init}\Big|_{init=k-means++} = N * \sum_{i=1}^{K} i = \frac{K*(K+1)}{2} * N$$

$$d_{lloyd} = niter * K * N$$

Or in other words, the distance computations for the KMeans is the sum of the number of distances of the initialization and the number of instances of lloyd's algorithm. For the k-means++ inicialization, for each center to be created the algorithm computes the distances of the distances of the cluster centers already initialized, which at each iteration increases by 1 until reacing K clusters. this can be computed as the sum of all natural number until K times the dataset size N. The number of distance computations for the lloyd algorithm is defined as the number of iterations done until convergence times a K * N distance computation matrix that is computed each iteration to assign each instance to a cluster.

Finally, the MiniBatchKMeans algorithm follows a similar computation scheme as KMeans but the d_{lloyd} is computed differently as it is performed with a batch size b.

$$d_{kmeans} = d_{init} + d_{lloyd}$$

$$d_{init}\Big|_{init=k-means++} = \frac{K*(K+1)}{2} * N$$

$$d_{lloyd} = niter * K * b$$

Where the only notable difference is that the distances computed in each iteration of the MiniBatchKMeans are not $K * N$ as in KMeans but rather $K * b$ due to the computation being done through batches.

Using this computation schemes, we perform a sweep for $K \in \{3, 9\}$, $D \in \{2, 4, 8\}$. For the artificial datasets, $N \in \{1e2, 1e3, 1e4, 1e5, 1e6\}$ while for the real dataset the values of N were chosen to be $N \in \{4000, 12000, 40000, 120000, 400000, 1200000, 4000000\}$ as in the paper. The results obtained with this experiment are shown in A.1 and A.2 containing the results for the artificial and real dataset respectively.

The figures show promising results where the number of distance computations is usually smaller than the number of computations for k-means++. In the artificial dataset's results we can see a little bit of overlap, especially in the case of D=8 and K=9, where all but the RPKM with $m \in \{1, 2, 3\}$ seem to have very similar performance. On the other hand, the real dataset's results show very promising results where none of the distance computations for any of the K, D combinations use more than the distances computed by MiniBatchKMeans and KMeans.

3.2 Instance Ratio

We define the instance ratio as the size of the partition divided by the number of instances in the dataset $|P|/n$. The goal of this experiment is to demonstrate the reduction of data points given to the WL algorithm at each iteration of the PRKM algorithm. In order to demonstrate the reduction, the artificial and real dataset were used like in the previous experiment. The number of replicas was set to 10, the number of instances N, the number of clusters K and the number of dimensions D were kept as the same as the previous experiment. In this experiment, only the RPKM algorithm was tested as it makes no sense to talk about partitions in KMeans and MiniBatchKMeans.

The instance ratio was computed for each max_iter (m) value for the RPKM and extracted at the end of the fit stage as the number of subsets in the m^{th} partition divided by the number of instances of the dataset. The results for the experiments can be seen in A.3 and A.4 containing the results for the artificial and real dataset respectively.

In the figures we can appreciate that on the first iterations of the RPKM algorithm, the number of representatives (or subsets) is very small wrt. the number of samples in the dataset as is expected. The exponential growth of the number of subsets is very apparent in the real dataset's results but in the artificial dataset's results, it is less apreciable as it reaches a ratio closer to 1 very quickly compared to the real dataset. For smaller number of samples in the dataset it's quite more apparent that the ratio grows a lot quicker than in the datasets that have a considerable ammount of instances. The results are promising as the algorithm is ment to be used with large datasets, and it seems to use a very small number of instances. We can note as well that the ratio increases very quickly wrt to the number of dimensions, as the number of hypercubes is exponential wrt the number of dimensions and the number of partitions will be bigger.

3.3 Quality of the approximation

In this section we will discuss the standard error (std.error) metric proposed in the paper and the experiment realized to quantitatively analyze the difference between the clustering centers extracted with RPKM and the clusters extracted with a k-means++ algorithm. In the paper, they define the clustering error as:

$$E(C) = \sum_{x \in D} \min_{k=1,\ldots,K} ||x - c_k||^2$$

This expression will be used to compute the clustering error of the RPKM algorithm and the KMeans algorithm using the following expression:

$$\rho = \frac{E_{km++} - E_{RPKM}}{E_{km++}}$$

This will always yield $\rho < 0$ and the more negative the value of ρ the worse is the approximation of the RPKM centers. The results for this experiment can be seen in A.5 and A.6 containing the results for the artificial and real datasets respectively.

From both figures we can extract the conclusion that the error decreases as m increases and the number of representatives with it. The more clusters the error becomes more apparent wrt the k-means++ algorithm. Also as the number of dimensions increase, as the number of partitions increases the ratio approaches 1 and then the algorithm approximates very well the k-means++ error but by means of brute force.

Chapter 4

Additional Experiments

In this chapter we will conduct additional experiments of the ones presented on the paper. The authors claimed two achievements in the paper, the reduction of distance computations for large datasets, which has been thorughly validated in the paper and in the previous secion of the report and that the RPKM algorithm was able to approximate the performance of a regular KMeans. The goal of the experiments in this chapter is to corroborate the claim of the authors that the clustering method approximates scikit-learn's kmeans.

The rest of the chapter will be divided into three sections. The first section will contain an analysis of some internal criteria between scikit-learn's KMeans and the RPKM algorithm will be compared. The second section will contain the evaluation and discussion of external criteria. The final section will contain some additional experiments on real datasets from the UCI repository.

4.1 Internal Criteria

In this section we will compute and discuss three internal criteria for the clustering results obtained with PRKM and how they compare to the same criteria obtained for scikit-learn's KMeans. The three criteria selected were silhouette, calinski_harabasz, and davies_bouldin.

The silhouette metric will provide a measure of quality of the separation between the clusters created by both algorithms. The silhouette metric ranges from -1 to 1, where a high value will indicate that the each object is well matched to its own cluster and poorly matched to neighboring clusters.

The calinski metric will provide a measure of the interclass to intraclass distance ratio, determining if the clusters are compact in the center of the clusters and disperse as the instance are further away of the centers. The calinski metric does not have a suitable range to be compared, but the higher the value the better. This particular metric is very suitable for spherical clusters with very compact centers such as normal point distributions. Because of that it was chosen to use it, as the artificial dataset that we will be using is formed by a gaussian mixture of data points.

The davies_bouldin metric will provide a measure of the average similarity measure of each cluster with its most similar cluster. The lower the score, the more separated the clusters are. The range of the metric's values is from 0 to infinite, where the lower the value, the better is the approximation.

The metrics were extracted with the artificial dataset generator with the number of instances $N \in \{100, 1000, 10000\}$, the number of elements in the high end of the spectrum was ommited due to computation time. The algorithms were also evaluated for the number of clusters $K \in \{3, 9\}$, for the number of dimensions $D \in \{2, 4, 8\}$. $N_{replicas} = 10$ were used to average the results and obtain a more realistic measure of the metrics.

The raw results for all the algorithms and the metrics obtained will be shown in the appendixes' figures B.1, B.2 and B.3 containing the raw results for each rpkm configuration and every (K, D) configuration of the silhouette, calinski and davies metrics respectively.

The results from the data shown in the previous figures was then aggregated to perform a simplified analysis of the metrics. We compared the metrics of each of the m configurations o the RPKM algorithm to the metrics obtained with the clusters obtained from *scikit-learn's KMeans++*. As the range of the values was very disperse, a simple similarity measure is performed similarly to the std.error presented in section 3.3 but as the error is not negative by definition in this case, the absolute value of the difference was taken in order to evaluate the absolute value of the error between the metrics. We define the value as the absolute standard error.

$$\rho' = |\rho| = \frac{|m_{km++} - m_{RPKM}|}{m_{km++} + m_{RPKM}}$$

The ρ' function is scale invariant and its values are [0, 1] for all the positive values of the metrics. The only metric that can be negative is the silhouette score. In order to avoid any issue, the silouette score will be scaled from the [-1, 1] range to the [0, 1] range linearly for the sake of this operation. Additionally, if both metrics are 0, ρ' is undefined, but the limit tends to 1, so this case will be controlled. Once they are solved, the ρ' value will give a similarity value where 0 means that the metrics were identical and 1 means that the metrics were completely different.

The metrics obtained were averaged through the different (K, D) combinations shown in the previous tables. So each value of the table comes from the comparison of every (K, D) combination compared to the KMeans++ algorithm and then averaged.

$$\bar{\rho}\,'(m) = \frac{1}{|K| \cdot |D|} \sum_{k \in K} \sum_{d \in D} \rho'(k, d, m)$$

	max iterations					
metric	1.0	2.0	3.0	4.0	5.0	6.0
silhouette	0.025	0.017	0.010	0.009	0.008	0.008
calinski_harabasz	0.196	0.162	0.140	0.132	0.130	0.126
davies_bouldin	0.113	0.094	0.073	0.070	0.070	0.069

Table 4.1: absolute standard error of internal criteria metrics

The results obtained with this metric are shown in 4.1. The closer the ρ' value is to 0, the better the approximation is for that algorithm. In the table we show the results for the different configurations of m as columns and the three metrics as rows. From the table we can see that the error values for the algorithm decreases as the number of RPKM iterations m increases. This represents a higher cluster similarity as the number of representatives increases, as the authors also stated. Additionally we can see that the similarity values for the silhouette score are better than the other two metrics, that could mean that the clusters are well defined but there is not much separation between them, as it's expected from a dataset with clusters with some overlap between clusters.

4.2 External Criteria

In this section we will perform some comparisons using the following external criteria: normalized mutual info, adjusted mutual info and adjusted random score. These metrics compare two sets of labels of the data to find relations between them. Usually, the labels of the clustering algorithm would be compared to a set of ground truth labels to compute the quality of the clustering algorithm, however, this is not the goal of the experiments. The goal of this section is to compare the KMeans++ algorithm to the RPKM and evaluate the approximation of the later one of the KMeans++ algorithm. In order to perform this comparison, we will use the clustering labels extracted from a KMeans++ run as the ground truth and we will measure the extracted RPKM labels to them.

The Normalized Mutual Information (NMI) is a normalization of the Mutual Information (MI) score to scale the results between 0 and 1. A value of 0 corresponds to no correlation between the two sets of labels while a value of 1 corresponds to a perfect correlation between the two sets of labels. This metric does not take the probability into account. The NMI metric is independent of the values of the labels, a permutation of the class labels will not alter the result of the metric in any way.

The Adjusted Mutual Information (AMI) is another adjustment of the MI metric to account for chance. It accounts for the fact that when the number of clusters is large for two clustering schemes, the MI is higher even though there might not be more information shared. As the NMI metric, AMI is independent on the values of the labels. The metric ranges from 0 for random asignations to 1 for perfectly matched partitions.

The Adjusted Rand Score (ARS) computes a similarity measure between two clustering s by considering all pairs of samples and couynting pairs that are assigned in the same or different clusters in the predicted and true label sets. The value of the metric ranges from 0 for completely random label sets to 1 for identical label sets up to a permutation.

The metrics were extracted with the artificial dataset generator with a number of instances $N \in \{100, 1000, 10000, 100000, 1000000\}$, number of clusters $K \in \{3, 9\}$, for the number of dimensions $D \in \{2, 4, 6, 8\}$. $N_{replicas} = 10$ were used to average the results and obtain a more realistic measure of the metrics. The results of the experiments can be found in B.4, B.5, and B.6 showing the NMI, AMI, and ARS metrics respectively.

The first appreciation in the plots is that the NMI and AMI scores are almost identical due to the fact that the clustering is not random but it follows a heuristic. From now on the appreciations done for AMI and NMI are equivalent. The NMI metric demonstates the same conclusion that has been recurrent in the report which is that the results support the claims by the authors that the algorithm is a good aproximation of the KMeans++ algorithm and the bigger the number of iterations m the better the approximation. Also, the AMI and NMI metrics are very similar due that the number of clusters used in the experiments are quite modest. Finally the ARS is quite similar as well as all of them are quite similar in concept, but from here we can see that the clustering is very similar to the reference and it's a good approximation of it.

4.3 Real World Datasets

In this section we will examine the performance of the algorithm on datasets from the UCI repoository. The following datasets were used in the experiment: arrhythmia, balance, cpu, dermatology, ecoli, german, glass, haberman, heart, iono, iris, letter, segment, sonar, tae, thy, vehicle, vowel, wisc, and zoo. All of them were used to fit a RPKM with $m = 6$. The results can be seen in 4.2.

Also, as we are working with datasets of different characteristics, the stopping condition will be used to limit the number of iterations of the RPKM algorithm.

dataset	**n_samples**	**n_dim**	**n_iter**	**RPKM_dist**	**km++_dist**	**instance_ratio**	**stderror**	**ami**	**sil_error**	**chs_error**	**dbs_error**
arrhythmia	452	262	2	83453.5	52884	1.000	-0.006	0.627	0.005	0.033	0.012
balance-scale	625	4	2	532.5	7500	0.565	-0.038	0.177	0.001	0.006	0.003
cpu	209	6	3	109910	1491006	0.885	-2.184	0.650	0.039	0.485	0.192
dermatology	366	34	2	16434	12078	1.000	-0.065	0.853	0.016	0.055	0.089
ecoli	336	7	1	64	14784	0.024	-1.901	0.340	0.101	0.663	0.026
german	1000	20	2.5	14423	8000	0.989	-0.044	0.500	0.005	0.202	0.043
glass	214	9	5	8163	10914	0.864	-0.213	0.582	0.034	0.153	0.021
haberman	306	3	2.5	153	2448	0.123	-0.330	0.340	0.046	0.364	0.138
heart-statlog	270	13	2	1892	1890	0.815	-0.065	0.544	0.019	0.178	0.063
iono	351	34	3	4504	3159	0.957	0.000	1.000	0.000	0.000	0.000
iris	150	4	5.5	2776.5	3375	0.937	-0.264	0.791	0.003	0.132	0.023
letter	20000	16	6	35591621	10140000	0.933	-0.027	0.763	0.000	0.021	0.014
segment	2310	19	6	124670	161700	0.890	0.068	0.801	0.023	0.046	0.092
sonar	208	60	2	2224	1456	1.000	0.000	0.173	0.011	0.001	0.019
tae	151	5	2	154.5	1812	0.192	-0.328	0.368	0.026	0.318	0.054
thy	215	5	5	1779	4515	0.772	0.000	1.000	0.000	0.000	0.000
vehicle	846	18	4	43242	21996	1.000	-0.011	0.767	0.004	0.009	0.017
vowel	990	13	5	125818	119790	0.989	-0.120	0.813	0.036	0.075	0.056
wisc	699	9	3	3912	6291	0.561	0.000	1.000	0.000	0.000	0.000
zoo	101	16	2	2124.5	4242	0.554	-0.039	0.775	0.017	0.059	0.049

Table 4.2: Results on real world datasets from UCI

In the table, the RPKM fitted with each dataset was compared to sklearn's kmeans++. First we mention the number of instances and number of dimensions as they are relevant to the results obtained from the experiment. The number of distance computations in kmeans and RPKM, the instance ration at the n_iter iteration is also shown for comparison. Finally we expose the metrics of AMI and the three internal criteria metrics used in 4.1.

Even though the number of distances was reduced for the balance, cpu, glass, haberman, iris, segment, tar, thy and zoo datasets, the number of computations for the other tested datasets was not reduced significantly, probably due to the elevated number of features in most of the datasets and the reduced number of instances. However, the metrics of the real world dataset verify the findings that have been done in the previous two sections, the clusters are a good aproximation.

Chapter 5

Conclusions

In this work we presented an implementation of the PRKM algorithm proposed by Marco Capó and Aritz Pérez and Jose A. Lozano in their paper [1] An efficient approximation to the K-means clustering for massive data. We have demonstrated though the experiments that the paper proposed that the solution implemented in this work is comparable to the implementation of the authors. We also reached results comparable to the ones that they achieved in their paper and corroborated the results and conclusions of their paper.

The RPKM algorithm is a good alternative to scikit-learn's KMeans and MiniBatchKMeans when dealing with large ammounts of data and with large ammounts of features as they reduce drastically the number of distance computations while obtaining a good aproximation of the cluster error. We believe that this algorithm has potential to be used efficiently when clustering large ammounts of data. This afirmation is backed by the results obtained in the experiments.

Appendices

Appendix A

Experiments Results

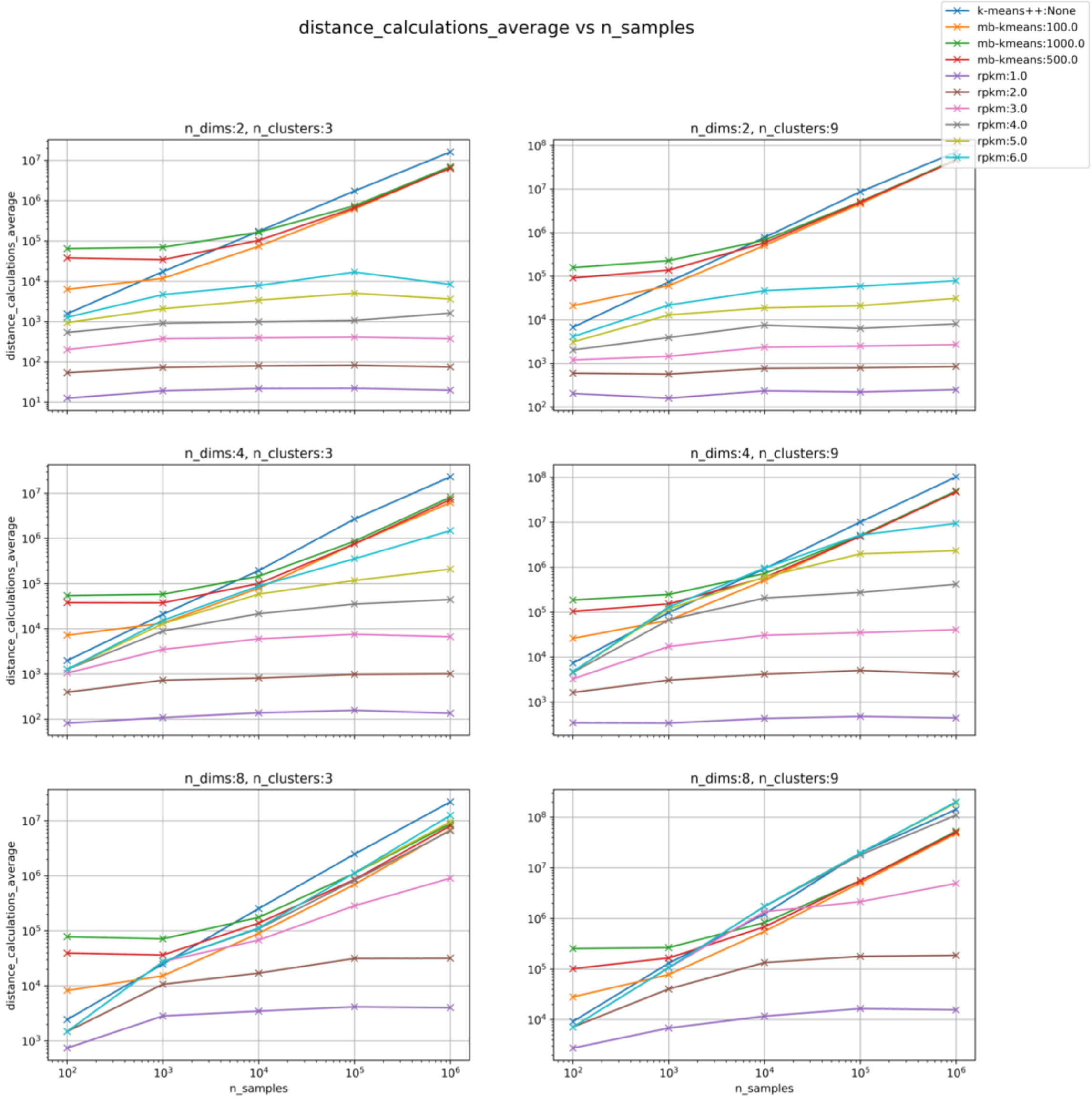


Figure A.1: Distance computation results in artificial datasets

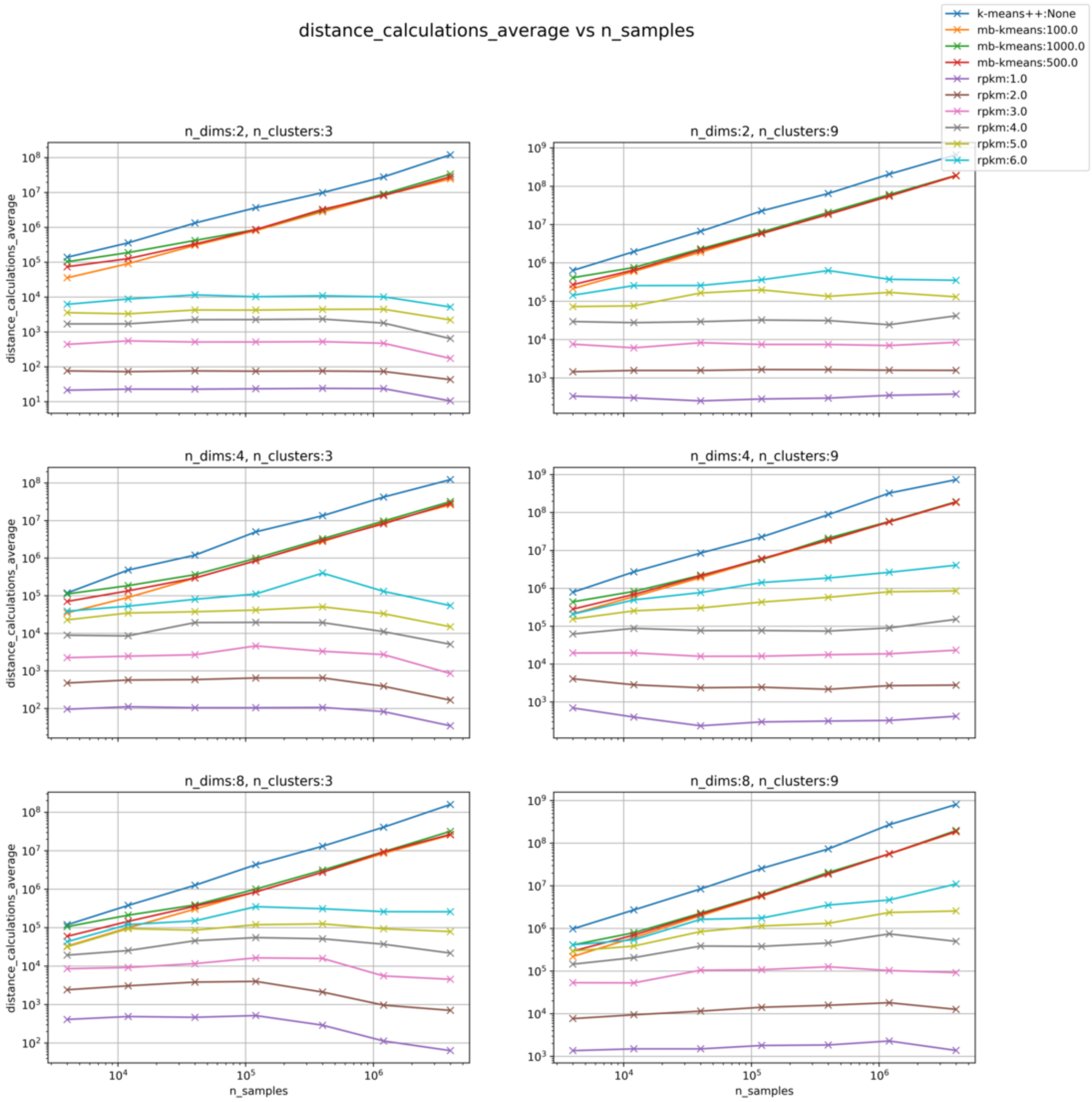


Figure A.2: Distance computation results in real datasets

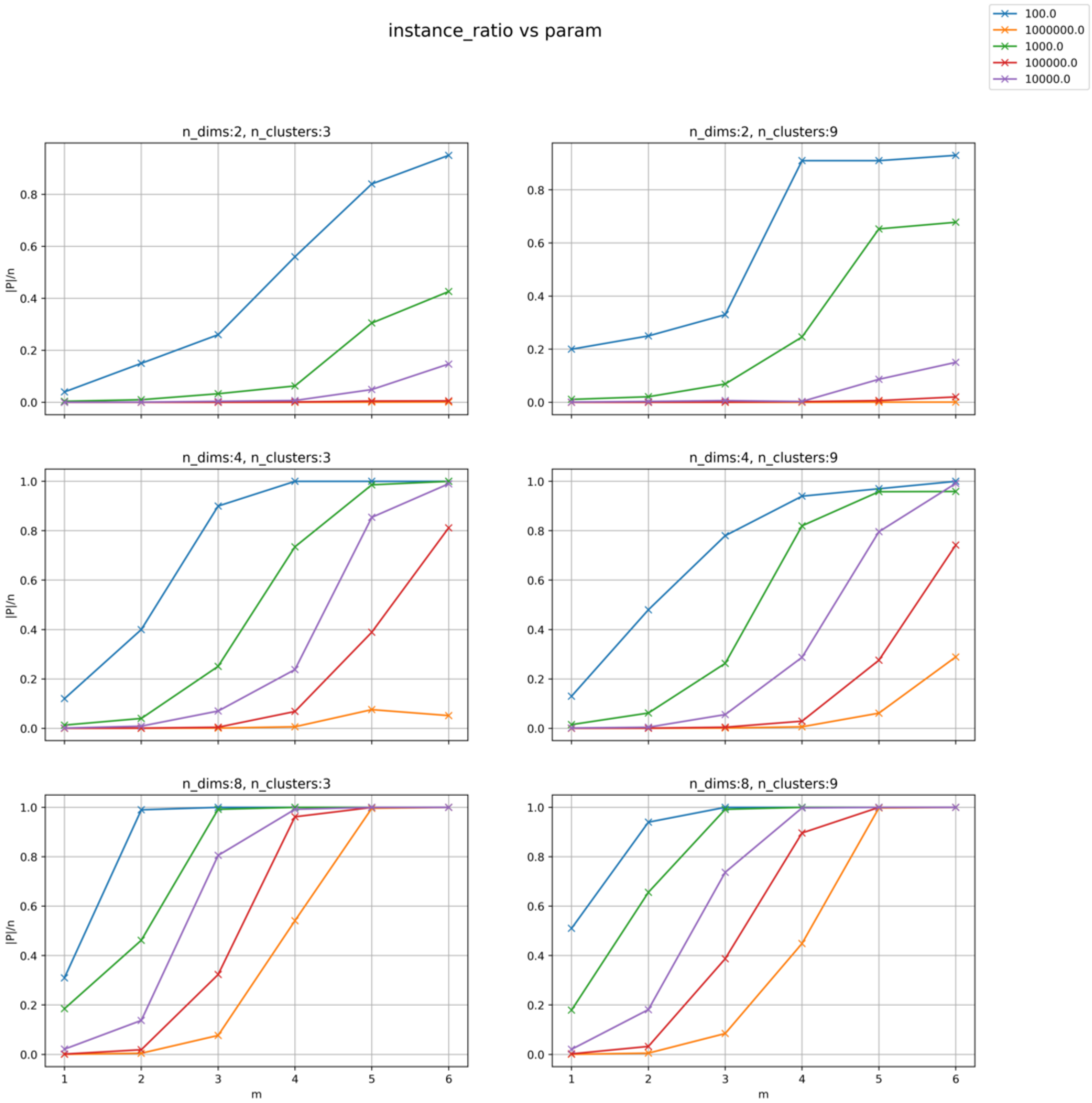


Figure A.3: Instance ratio computation results in artificial datasets

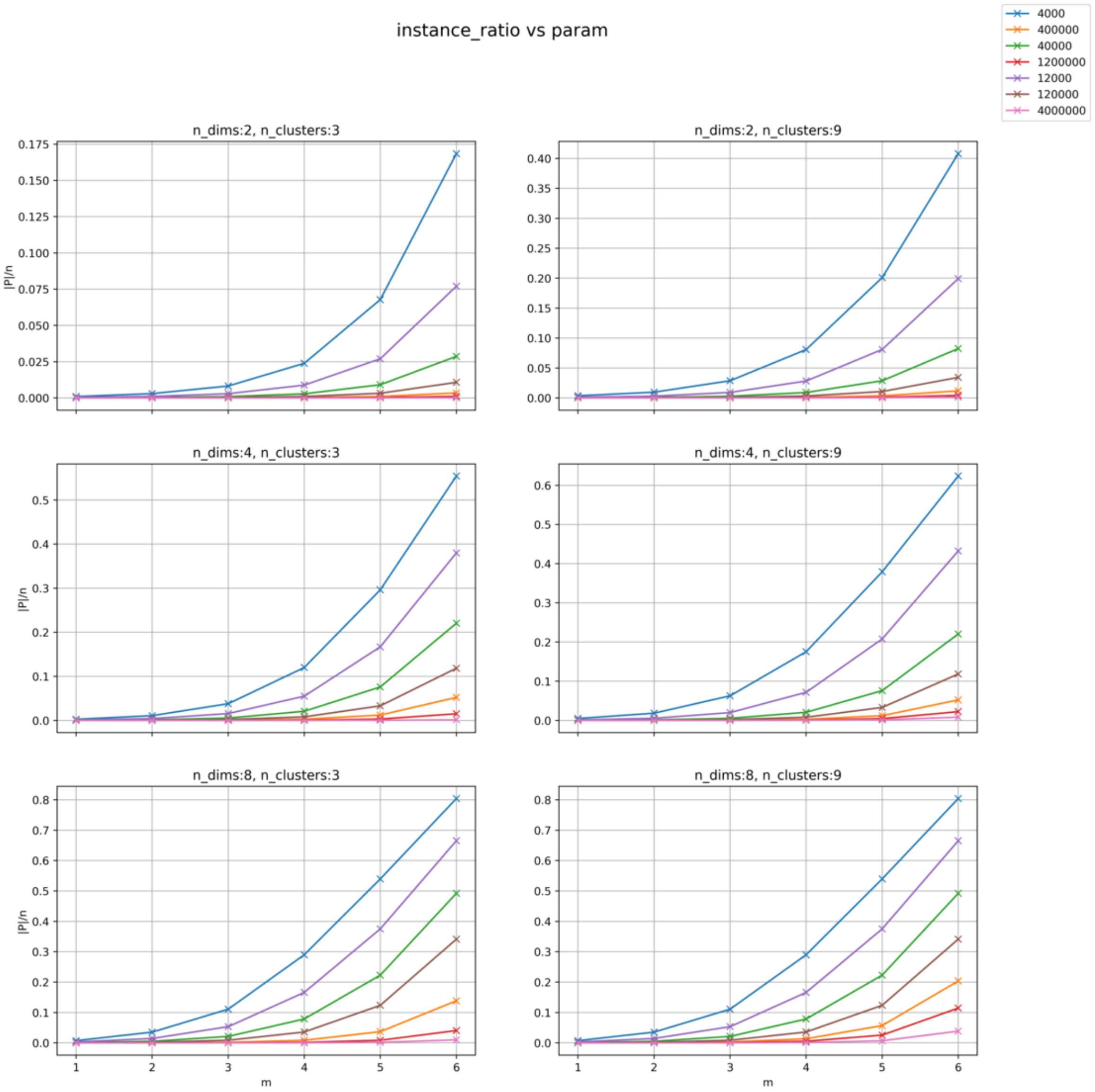


Figure A.4: Instance ratio computation results in real datasets

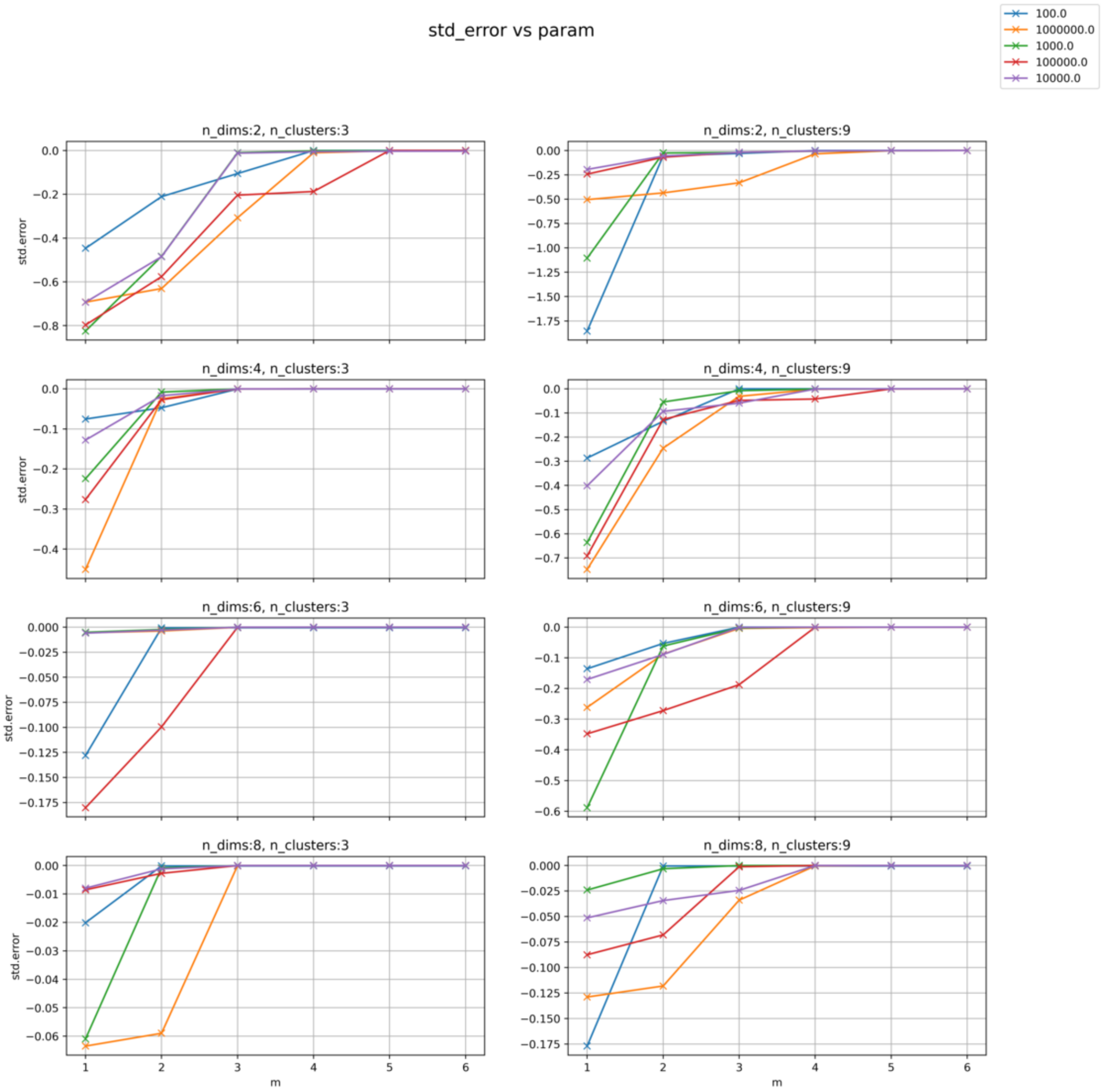


Figure A.5: Standard error computation results in artificial datasets

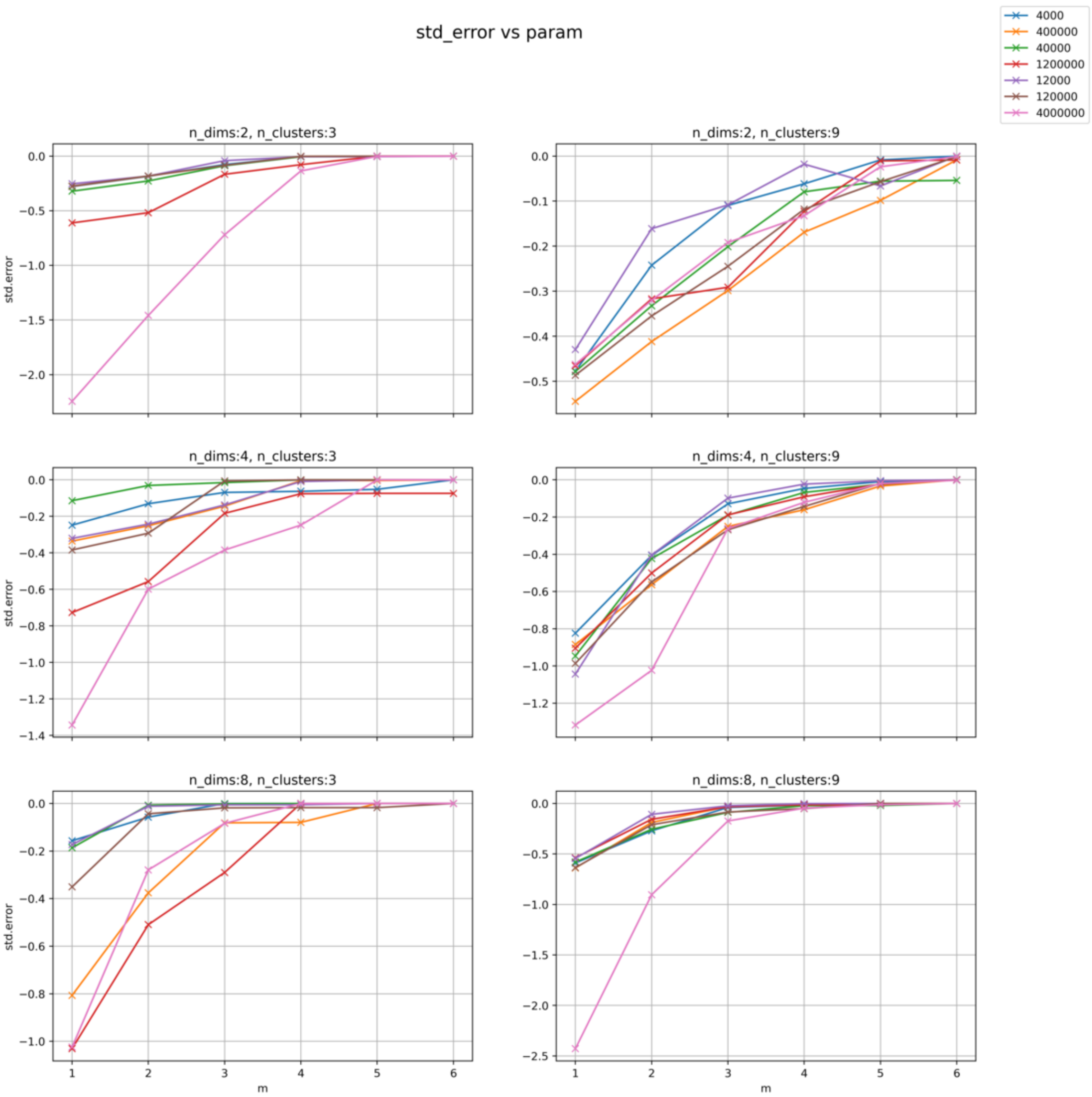


Figure A.6: Standard error computation results in real datasets

Appendix B

Additional Experiments Results

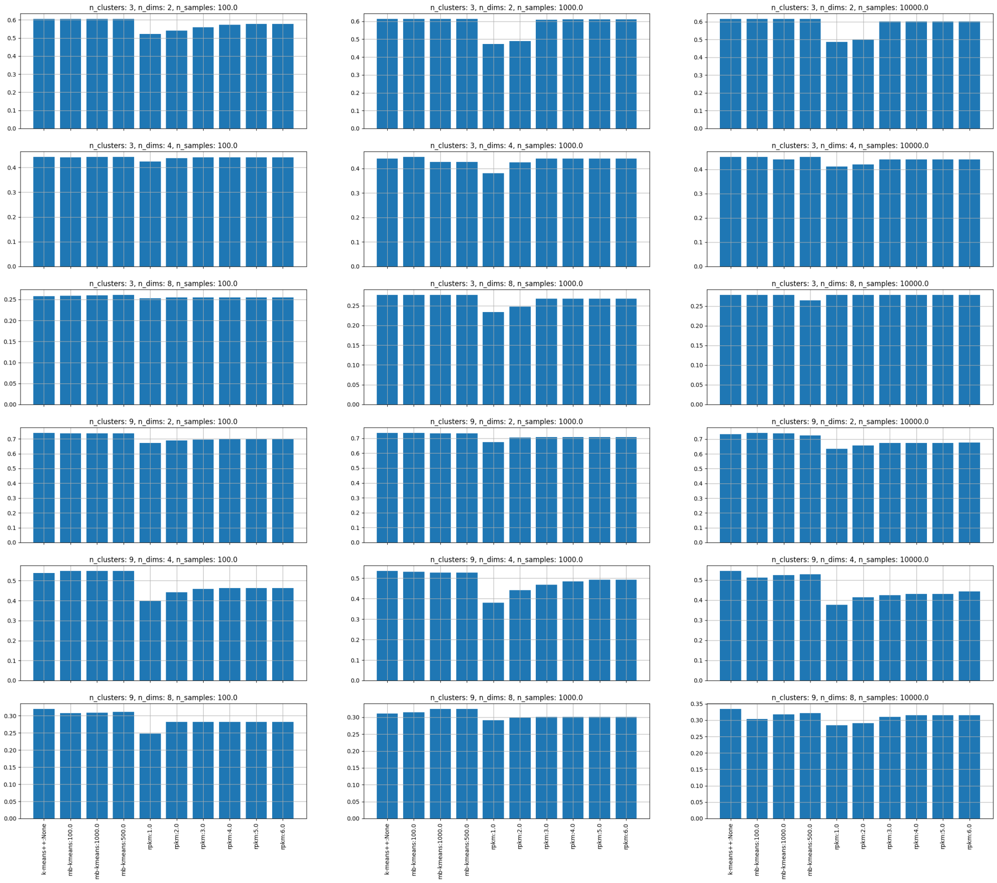


Figure B.1: Silhouette score results in artificial datasets

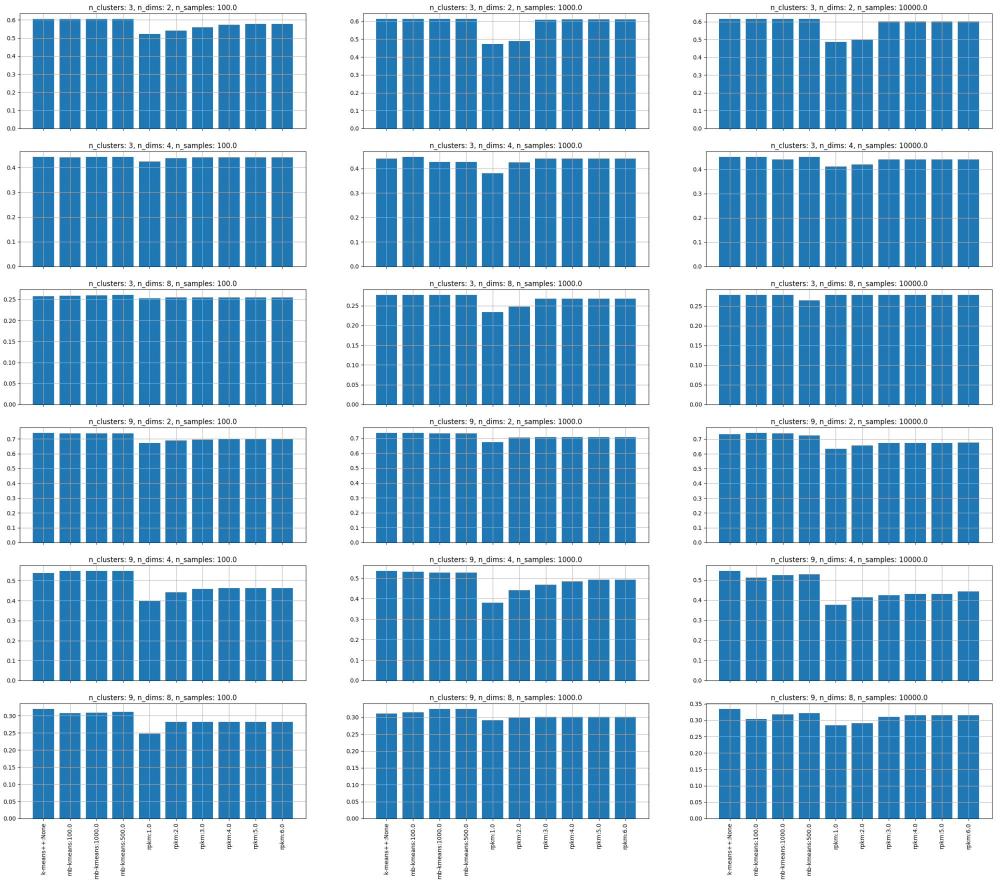


Figure B.2: Calinski Harabasz score results in artificial datasets

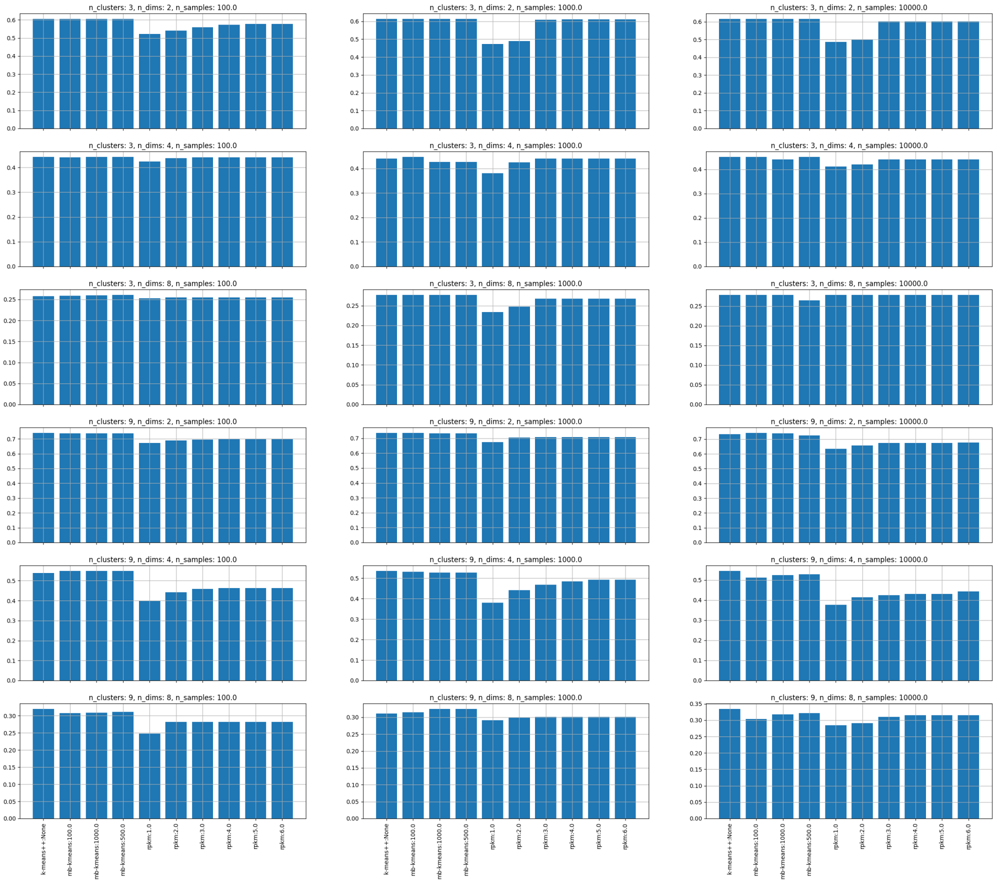


Figure B.3: Davies Bouldin score results in artificial datasets

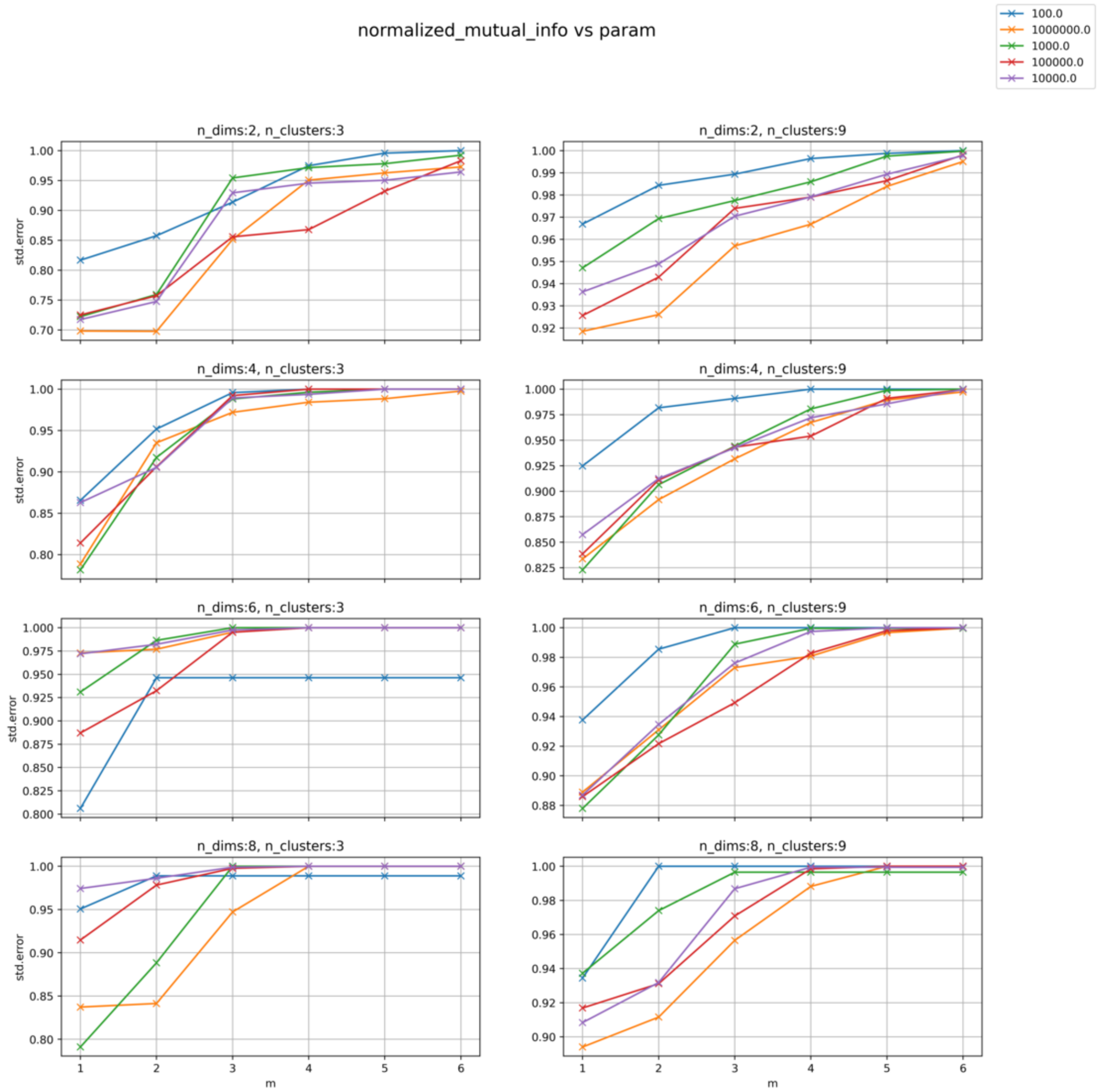


Figure B.4: Normalized Mutual Info score results in artificial datasets

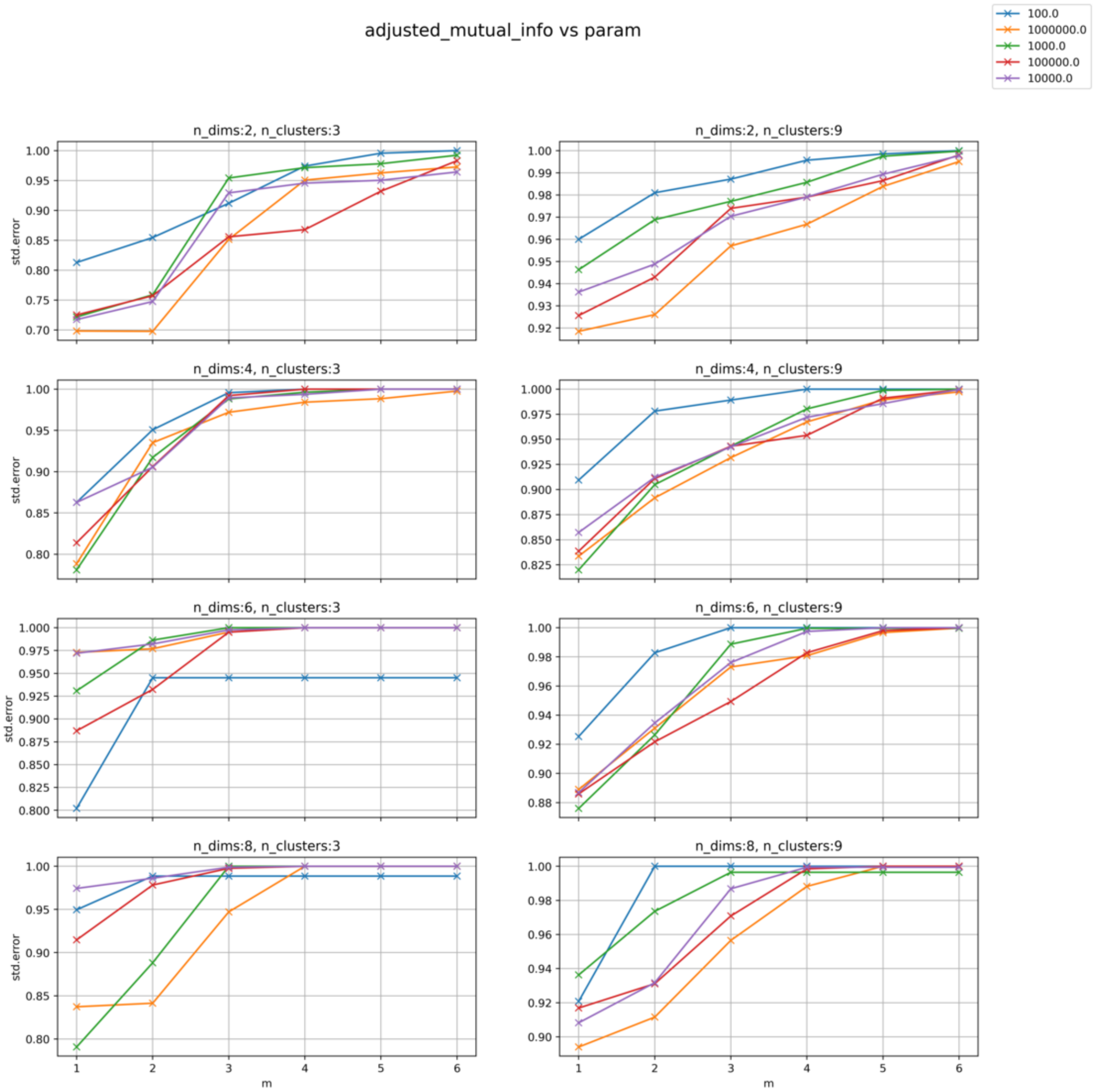


Figure B.5: Adjusted Mutual Info score computation results in artificial datasets

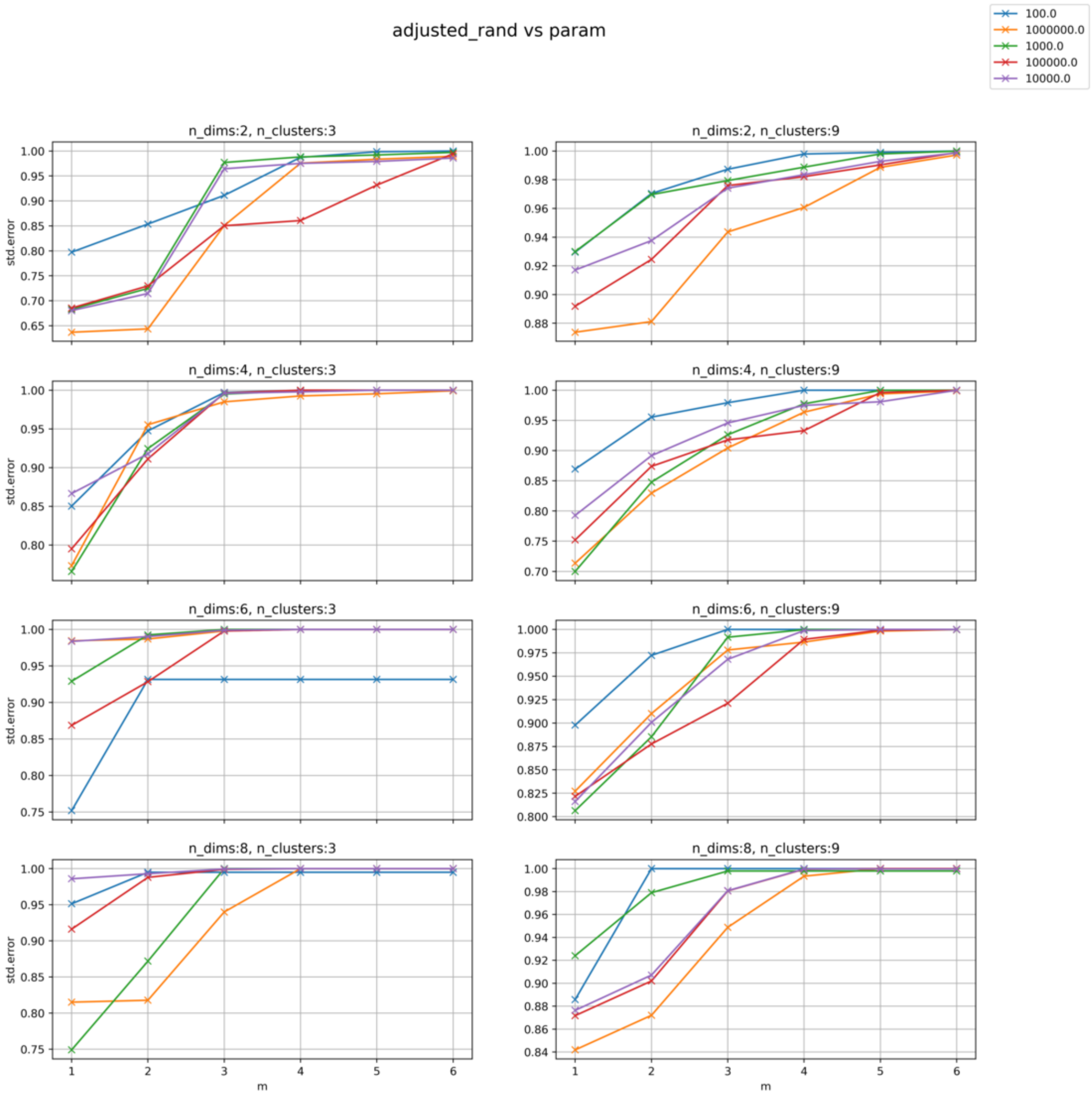


Figure B.6: Adjusted Random score results in artificial datasets

Bibliography

[1] Marco Capó, Aritz Pérez, and Jose A. Lozano. "An efficient approximation to the K-means clustering for massive data". In: *Knowledge-Based Systems* 117 (2017). Volume, Variety and Velocity in Data Science, pp. 56–69. ISSN: 0950-7051. DOI: https://doi.org/10.1016/j.knosys.2016.06.031. URL: https://www.sciencedirect.com/science/article/pii/S0950705116302027.

[2] *Gas sensor array under dynamic gas mixtures Data Set*. URL: https://archive.ics.uci.edu/ml/datasets/gas+sensor+array+under+dynamic+gas+mixtures.

[3] *UCI Machine Learning Repository*. URL: https://archive.ics.uci.edu/ml/index.php.