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

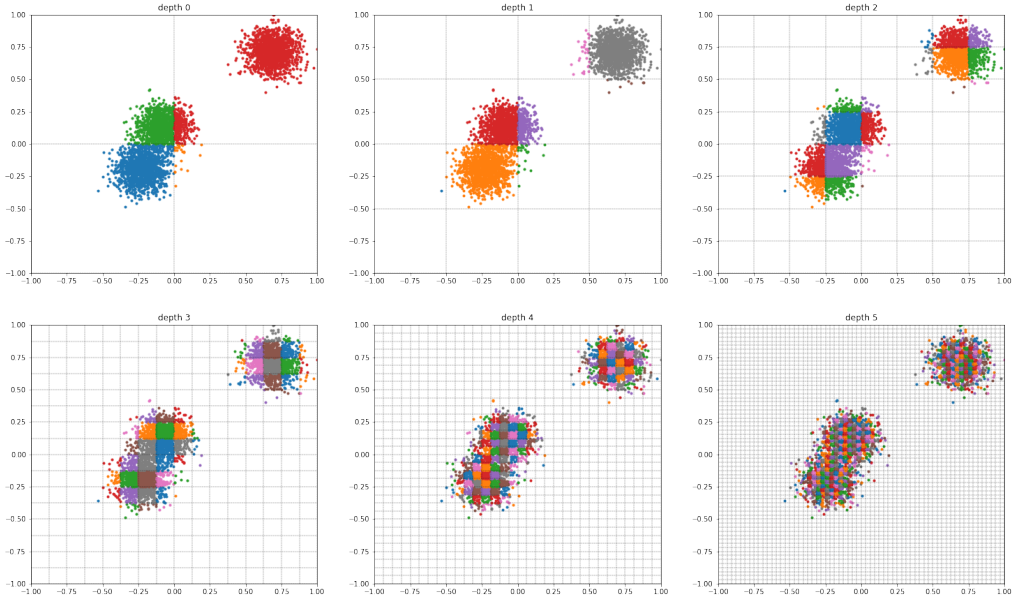


Figure 1.1: Partition example for a 2d gaussian mixture

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.

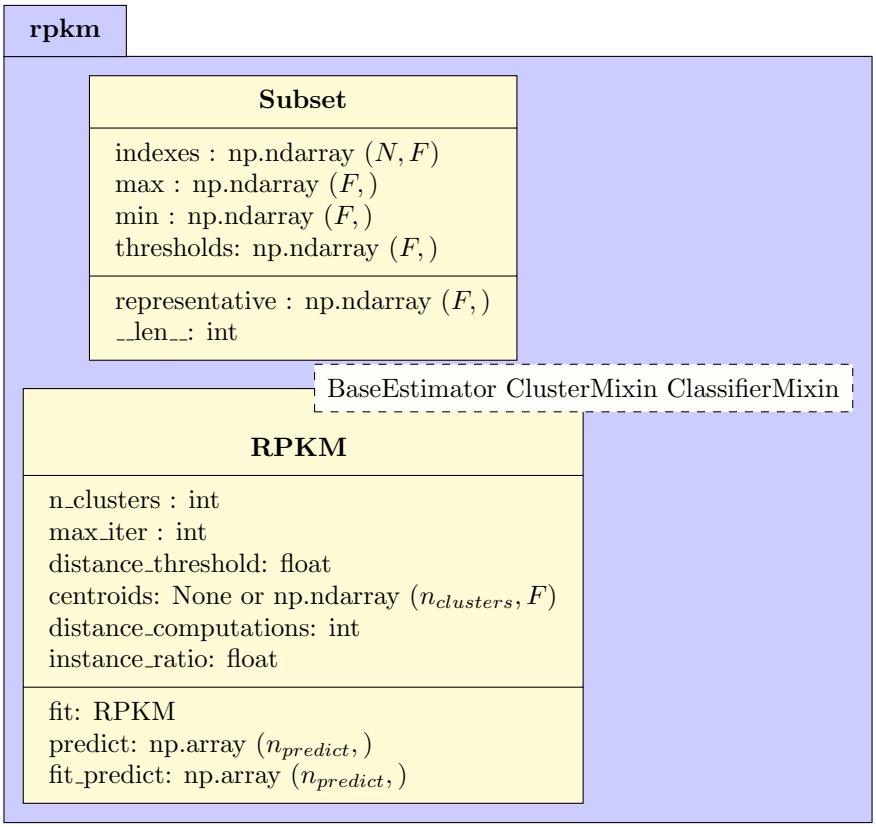
Both algorithms hav similar stop criteria, a threshold on the distance between the centers of the previous and new iteration and a maximum number of iterations. However, the authours have removed the distance limitation in their implementation in order to see the effects of the algorithm fully.

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.



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:

```
1 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 the *np.packbits* method.

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.

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.

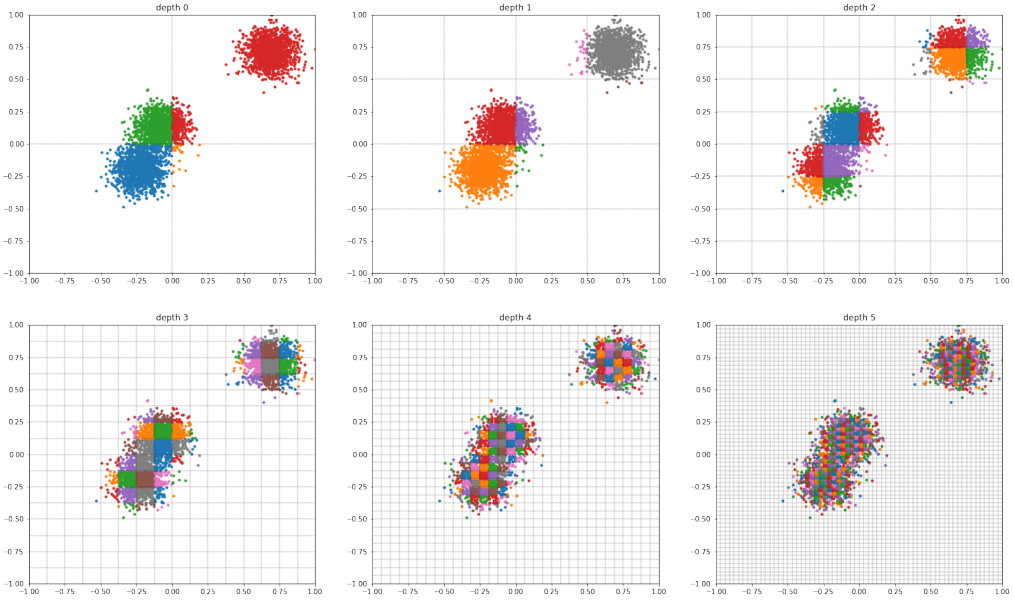


Figure 2.1: Partition example for a 2d gaussian mixture

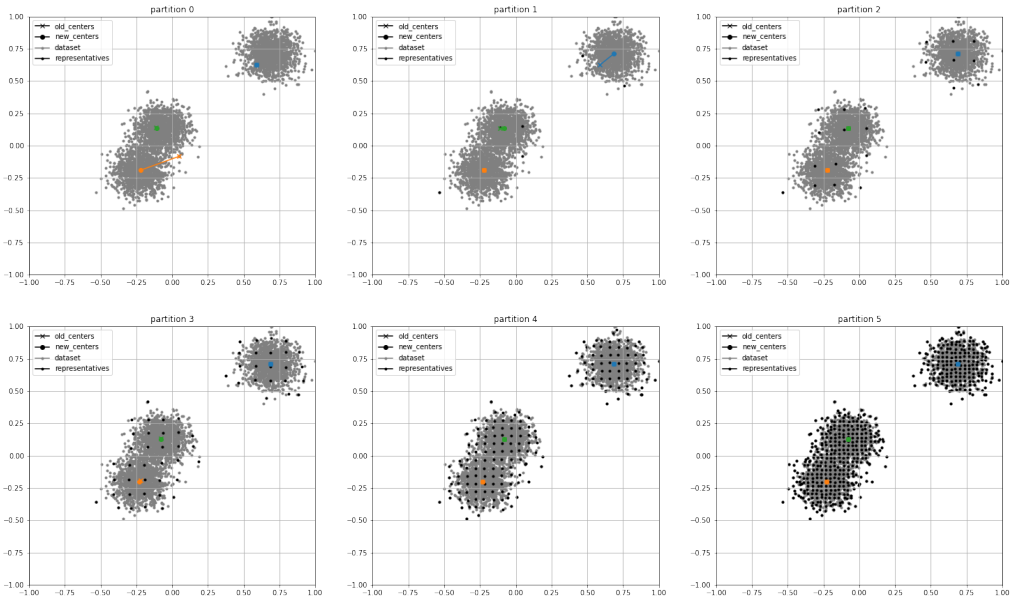


Figure 2.2: Centers and representatives in each iteration

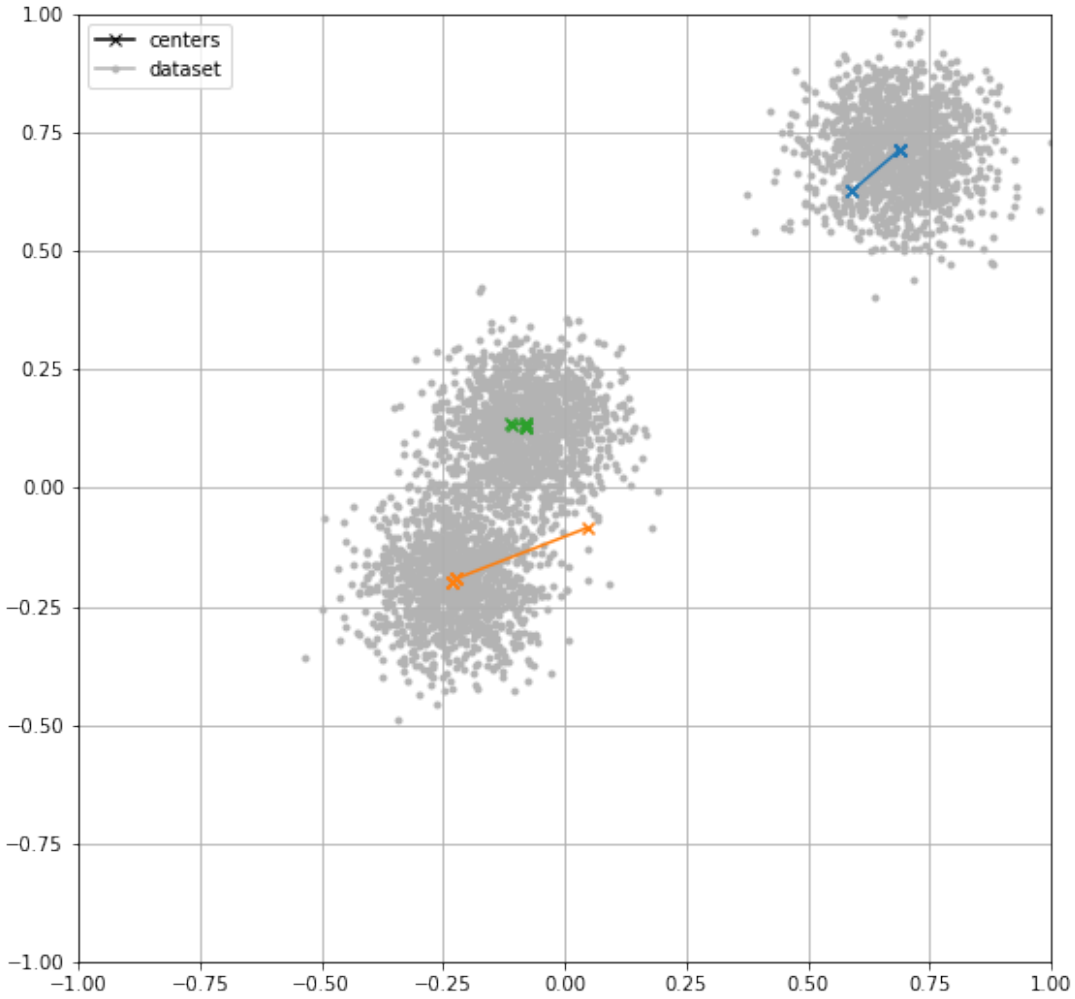


Figure 2.3: Center evolution

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*.

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 MiniBatchK-Means 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 3.1 and 3.2 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

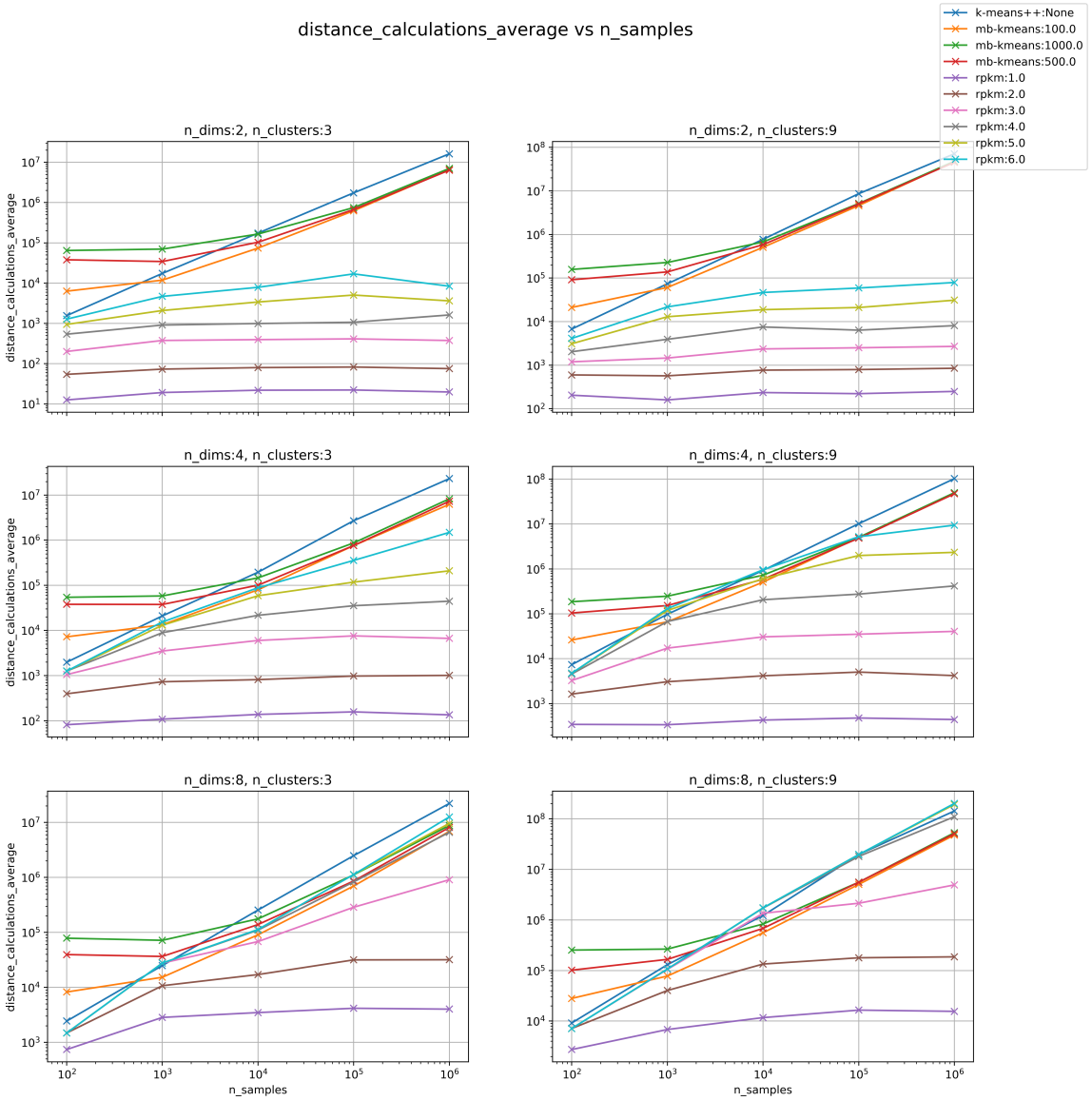


Figure 3.1: Distance computation results in artificial datasets

instances of the dataset. The results for the experiments can be seen in 3.3 and 3.4 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

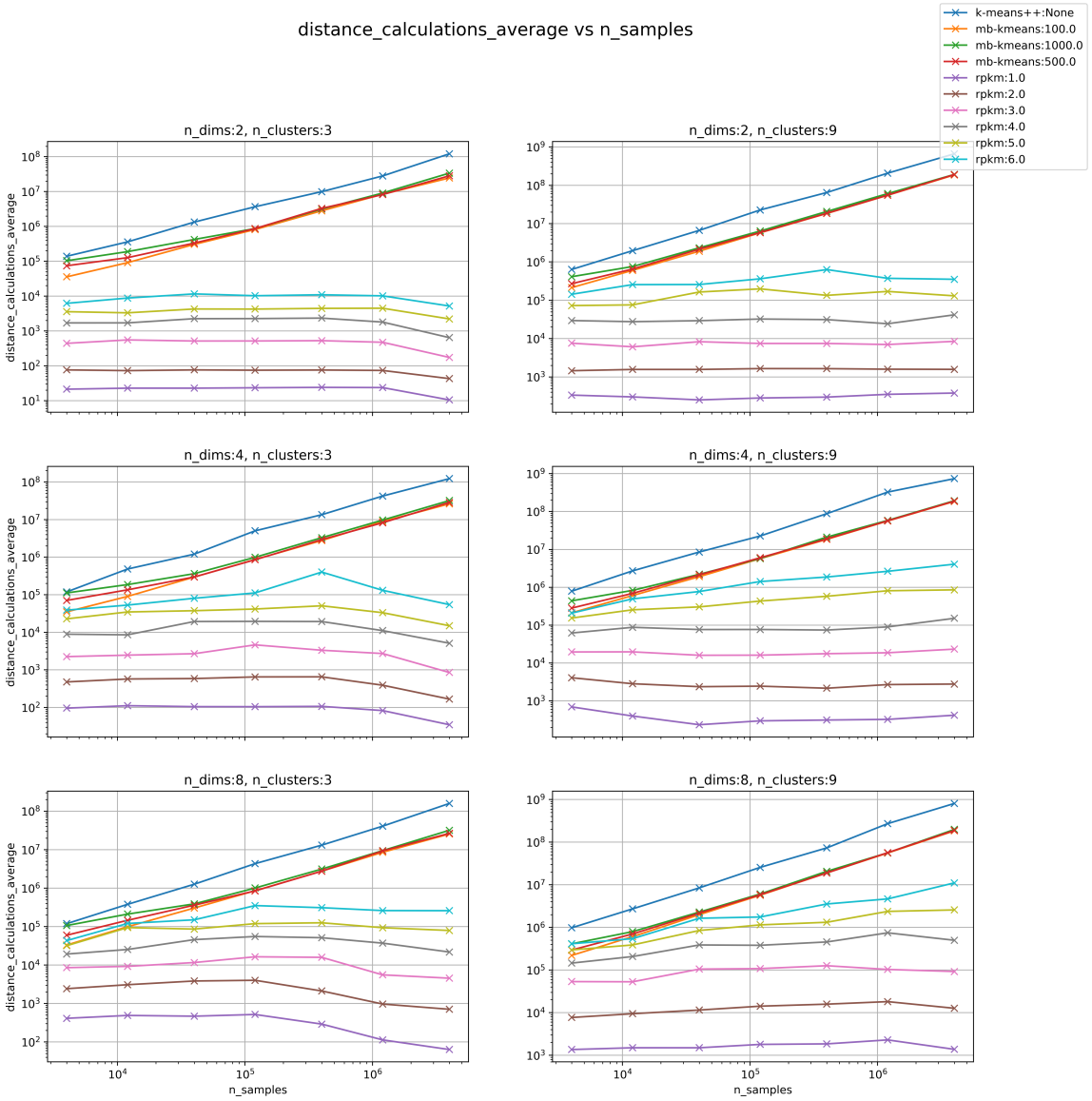


Figure 3.2: Distance computation results in real datasets

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:

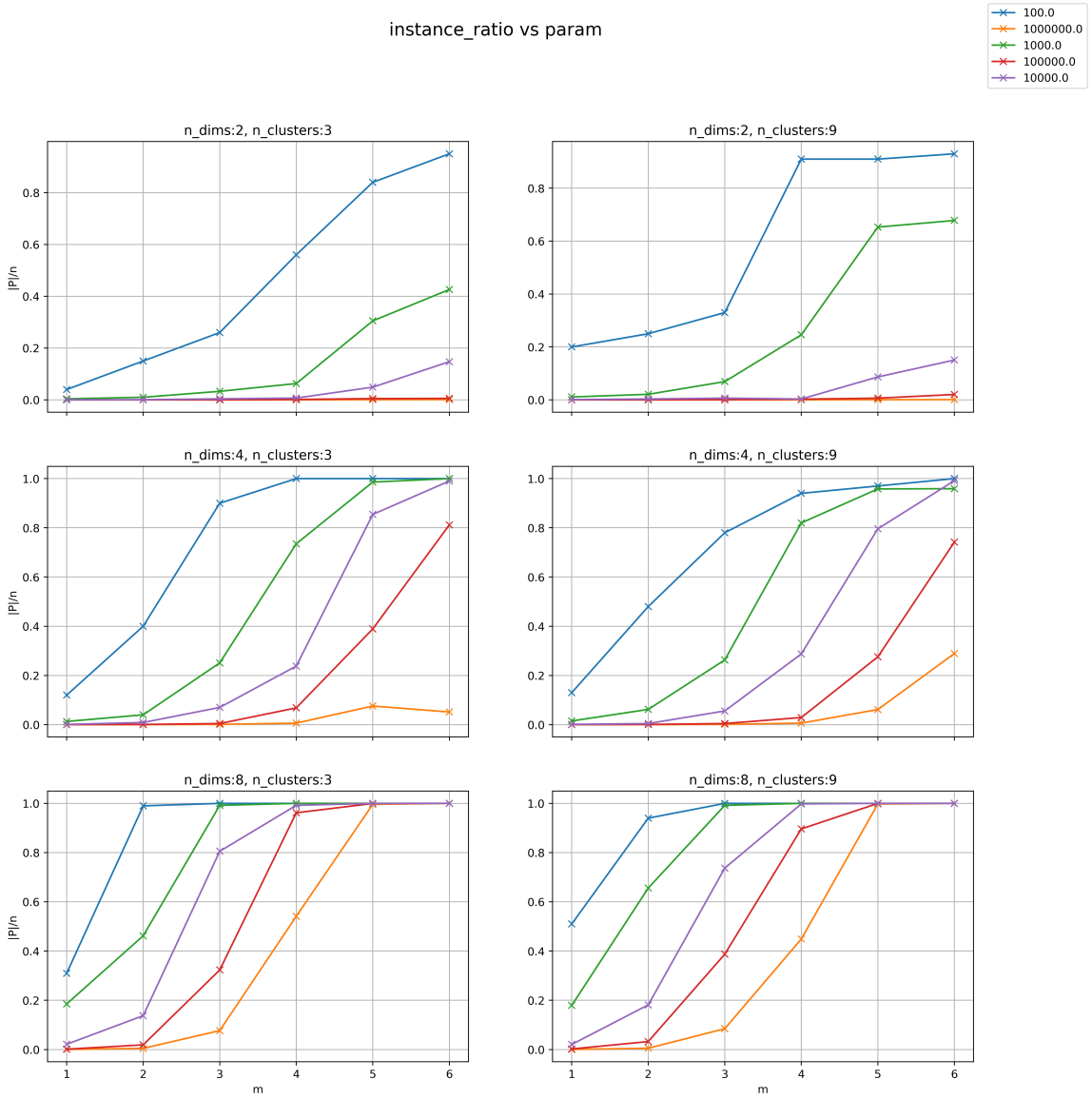


Figure 3.3: Instance ratio computation results in artificial datasets

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

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 3.5 and 3.6 containing the

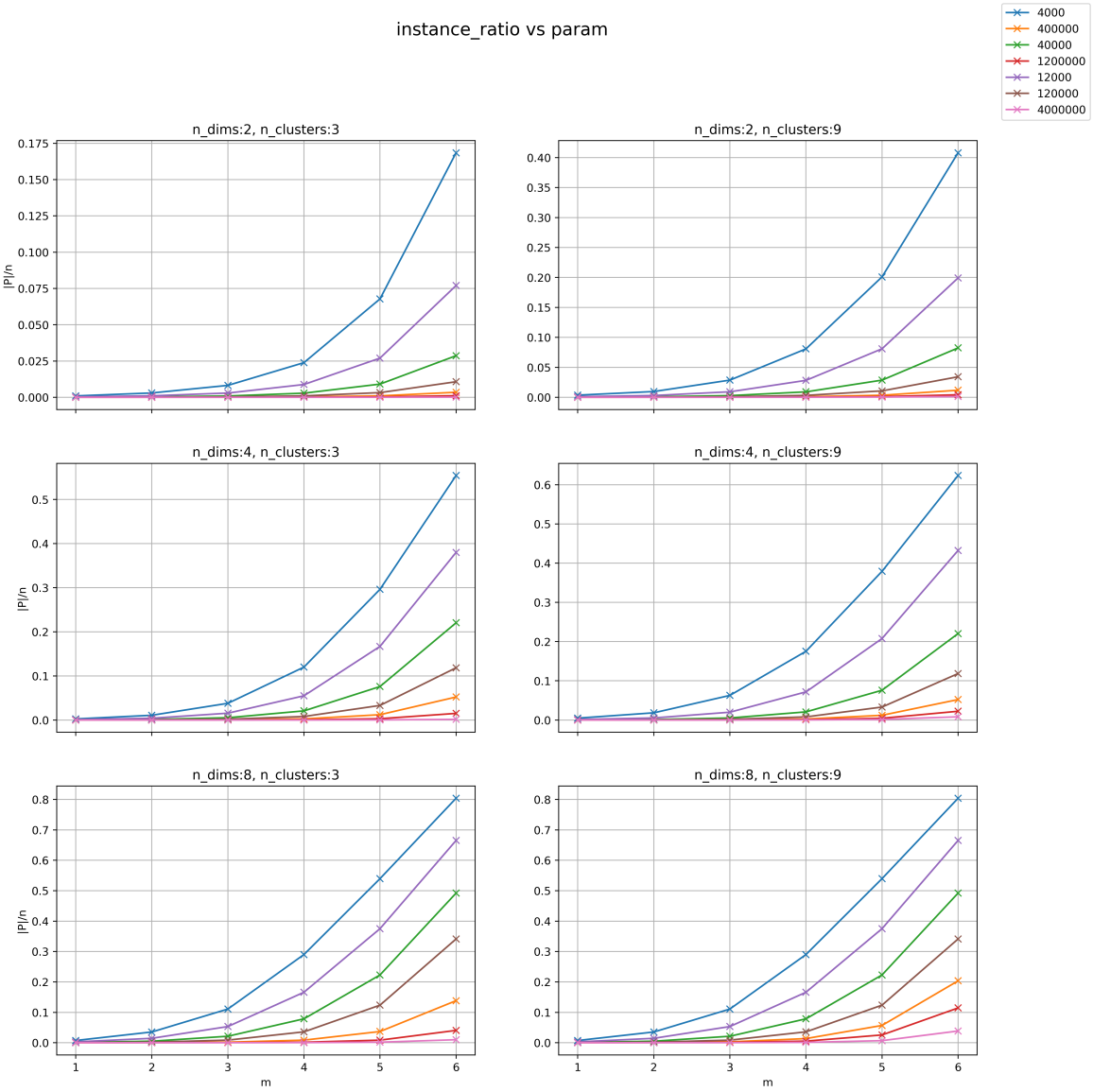


Figure 3.4: Instance ratio computation results in real datasets

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.

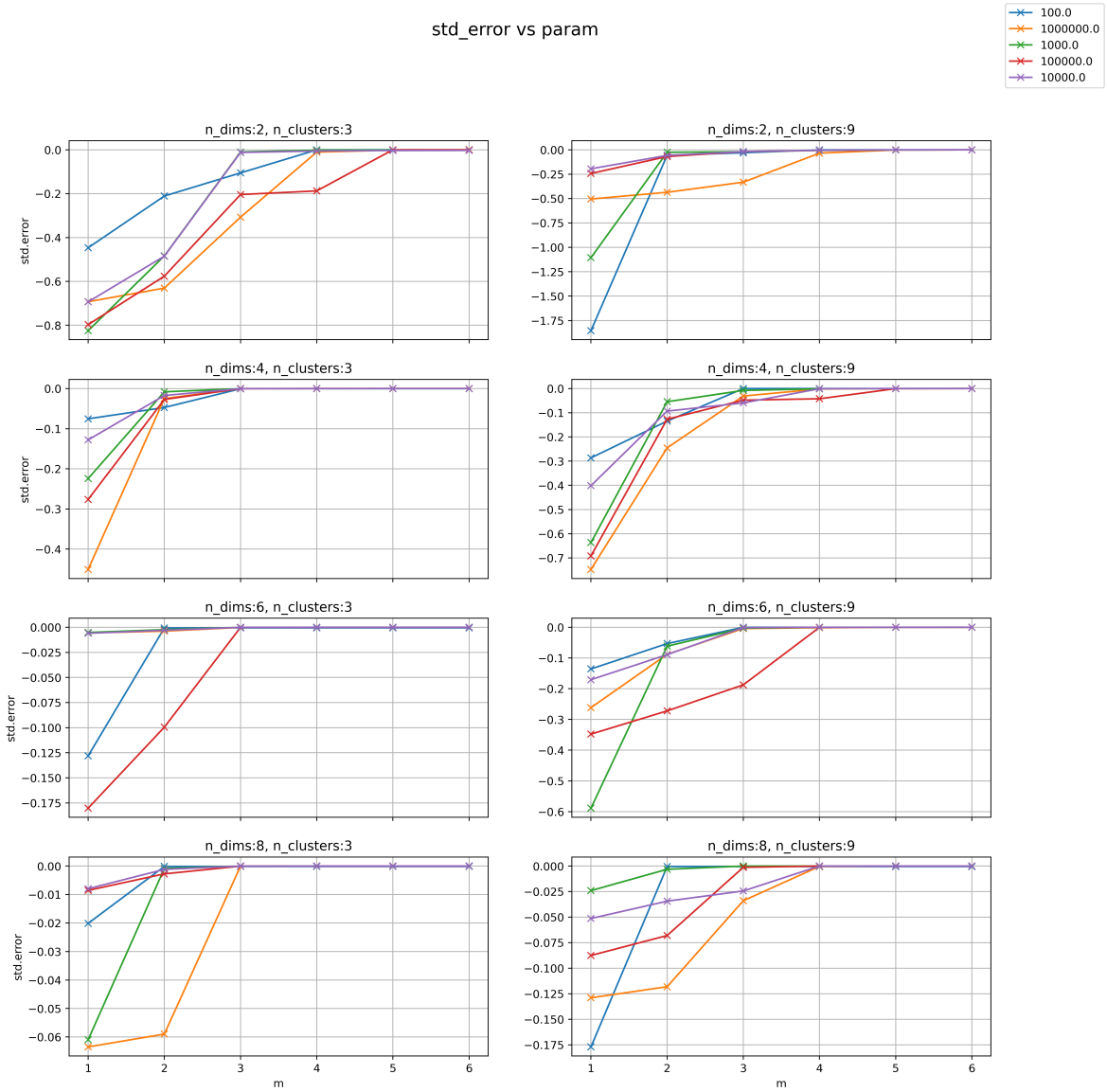


Figure 3.5: Standard error computation results in artificial datasets

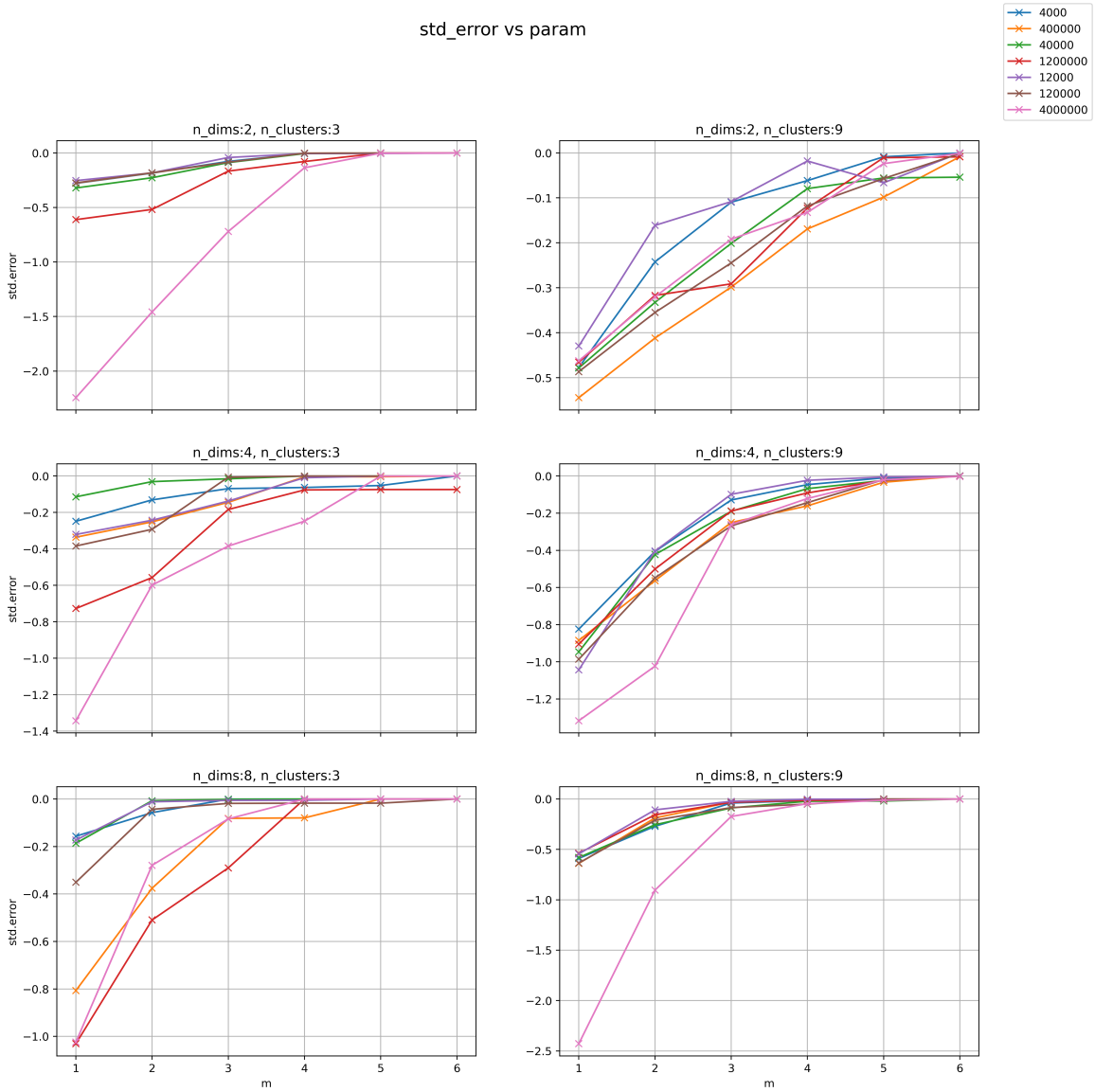


Figure 3.6: Standard error computation results in real datasets

Chapter 4

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.

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.