Distributed, MapReduce-based Nearest Neighbor and ϵ-ball Kernel k-Means

Nikolaos Tsapanos, Anastasios Tefas, Nikolaos Nikolaidis and Ioannis Pitas
Department of Informatics
Aristotle University of Thessaloniki
Thessaloniki, Greece
{niktsap, tefas, nikolaid, pitas}@aiia.csd.auth.gr

***Abstract*—Data clustering is an unsupervised learning task that has found many applications in various scientific fields. The goal is to find subgroups of closely related data samples (clusters) in a set of unlabeled data. A classic clustering algorithm is the so-called k-Means. It is very popular, however, it is also unable to handle cases in which the clusters are not linearly separable. Kernel k-Means is a state of the art clustering algorithm, which employs the kernel trick, in order to perform clustering on a higher dimensionality space, thus overcoming the limitations of classic k-Means regarding the non linear separability of the input data. Kernel k-Means typically computes the kernel matrix, which contains the results of the kernel function for every possible sample combination. This matrix can be viewed as the weight matrix of a full graph, where the samples are the vertices and the edges are weighed according to the similarity between the samples they connect, according to the kernel function. In this context, it is possible to work on the Nearest Neighbor graph, where each sample is only connected to some of its closest samples, or only using information from samples that are sufficiently close to each other, referred to as ϵ-ball. Doing so reduces the size of the kernel matrix and can provide improved clustering results. In this paper, we present a MapReduce based distributed implementation of Nearest Neighbor and ϵ-ball Kernel k-Means.**

I. Introduction

The objective of *data clustering* is to divide a given group of unlabeled data samples in subgroups (*clusters*), so that data samples belonging to the same cluster are similar to each other and dissimilar to data samples belonging to any other clusters. Clustering has found many applications in different scientific fields. Despite the fact that there has been an extremely rich bibliography on this subject for years now [1], it is still an active research field.

One of the earliest clustering methods is the *k-Means* algorithm [2]. It is a basic textbook approach. Yet it is still popular, despite its age. It involves an iterative process, in which each data sample is assigned to the closest of the k cluster centers and then each cluster center is updated to the mean of all data samples assigned to this cluster. The initial cluster assignment can be random. The process continues, until there are no changes, or until a maximum number of iterations has been reached. The main drawback of this approach is the fact that the surfaces separating the clusters can only be hyperplanes. Thus, if the clusters are not linearly separable, the standard k-Means algorithm will not be able to give very good results.

In order to overcome this limitation, the classical algorithm has been extended into the *Kernel k-Means* [3]. The basic idea behind kernel approaches is to project the data into a higher, or even infinite dimensional space. It is possible for a linear separator in that space to have a non-linear projection back in the original space, thus solving the non-linear separability issue. The *kernel trick* [4] allows us to circumvent the actual projection to the higher dimensional space. The trick involves using a *kernel function* to implicitly calculate the dot products of vectors in the kernel space using the feature space vectors. Let $a_i, i = 1, \ldots, n$ be the data sample set to be clustered and $\mathbf{x}_i \in \mathbb{R}^d, i = 1, \ldots, n$ their d-dimensional feature vectors. If $\phi(\mathbf{x}_i)$, $\phi(\mathbf{x}_i)$ are the projections of the feature vectors $\mathbf{x}_i$ and $\mathbf{x}_j$ on the kernel space, then $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$ is a kernel function. Different kernel functions correspond to different projections. Finally, Euclidean distances in the kernel space can be measured using dot products. Kernel k-Means provides a popular starting point for many state of the art clustering schemes [5], [6], [7], [8]. A recent survey on kernel clustering methods can be found in [9], while [10] presents a comparative study which supports the superiority of kernel clustering methods, over more conventional clustering approaches.

A convenient way to have quick, repeated access to the dot products without calculating the kernel function every time, is to calculate the function once for every possible combination of two data samples. The results can be stored in a $n \times n$ matrix $\mathbf{K}$ called the *kernel matrix*, where $K_{ij} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$. This means that the i-th row of the kernel matrix contains the kernel function entry for every possible sample combination that includes $\mathbf{x}_i$. Interestingly, it has been proven that Kernel k-Means, Spectral Clustering and Normalized Graph Cuts are closely related tasks [11]. The kernel matrix can, therefore, be viewed as the weighted adjacency matrix of a full graph, whose nodes are the data samples a_i and whose edge weights are the kernel function values. Obviously, when there are n data samples, the size of the kernel matrix is $n \times n$ and, therefore, grows quadratically with respect to n.

Additionally, there have been approaches attempting to take advantage of the local area information around each sample, in order to improve performance, make the kernel matrix sparse, or determine the number of clusters. It is possible to use only a small number of entries in each row of the kernel matrix, instead of the entire matrix [12]. This can be accomplished by either working on the *k-Nearest Neighbor graph*, where each sample is only connected to its k closest

978-1-4799-7560-0/15 $31.00 © 2015 IEEE
DOI 10.1109/SSCI.2015.81

samples [13], [14], or only using information from samples that are sufficiently close to each other [15], [16], which is referred to as *ϵ-ball*, *ϵ-neighborhood* or *r-graph* clustering. Furthermore, it is also possible to introduce additional weights to the connections between samples, based on estimating the local scaling parameter, i.e., by taking into account how densely or sparsely populated the area around each sample is [17], [18]. The *Hartigan Dip Test for unimodality* [19] is used in [20], in order to determine whether a cluster should be further divided into subclusters. The Dip Test is applied for each sample, referred to as a *viewer* in [20], on the relative distances between itself and other samples of the cluster. If there are viewers for which the unimodality test fails, then the cluster is further split.

Furthermore, in Media Production, footage is shot through multiple cameras, several of them often containing actors and multiple takes of the same shot are filmed. This generates very large amounts of images, which will have to be put together in the editing stage for the final product to be complete. Being able to retrieve relevant footage through queries would be very useful in quickly finding the proper video clips or video segments that contain the desired content. For example, it may a query may request footage where a certain actor appears, or a certain action is performed. In the case of actors, they are mostly identifiable by their face. Manual annotation of actors present in footage would be easier and faster, if several images of the same actor can be presented to the annotator. Clustering can be applied to facial images to facilitate the quicker annotation and image retrieval tasks.

Distributed computing can provide the means to handle problems on very large datasets that would otherwise be almost impossible to solve [21]. It provides virtually limitless memory and processing power. Provided that a task can be split into many independent subtasks, then it can theoretically be performed in a reasonable amount of time, regardless of the data size, given enough processing units. A distributed approach that can work with any serial clustering algorithm entails using the serial algorithm on data subsets, then merging the clusters [22]. Distributed versions of other clustering algorithms related to Kernel k-Means, like classic k-Means [23] and k-Medians [24] have already been discussed.

In this paper, we present a distributed implementation of the Nearest Neighbor and ϵ-ball variations of Kernel k-Means. The implementation uses Apache Spark [25], a cluster computing framework, which is similar to and compatible with Hadoop [26]. The computing cluster can include a wide variety of hardware from high-end, multiprocessor computers with large amounts of RAM, to average modern PCs. The focus of the proposed implementation is to avoid requiring the storage of n^2 kernel matrix entries into the distributed memory at the same time, if possible. This paper is organized as follows: Section II briefly introduces Kernel k-Means, Section III describes the distributed implementation, Section IV presents the experimental evaluation of the implementation and Section V concludes tbe paper.

II. Kernel k-Means

The Kernel k-Means algorithm [27] is an extension of the classic k-Means clustering algorithm. Taking advantage of the kernel trick, it implicitly projects the data onto a higher dimensional space and measures Euclidean distances between data samples in that space. This circumvents the limitation of linear separability imposed by k-Means. Let there be k clusters $C_\delta, \delta = 1, \ldots, k$ and data samples $a_i, i = 1, \ldots, n$. Each cluster C_δ has a center $\mathbf{m}_\delta$ in the higher dimensional space $\mathbb{R}^{d'} (d << d')$, where $\Phi : \mathbb{R}^d \to \mathbb{R}^{d'}$ is the mapping function. Assuming that there is an assignment of every data sample to a cluster, then the center of cluster C_δ is computed as follows:

$$\mathbf{m}_\delta = \frac{\sum_{a_j \in C_\delta} \phi(\mathbf{x}_j)}{|C_\delta|}, \tag{1}$$

where $|C_\delta|$ is the cardinality of cluster C_δ. The squared distance $D(\mathbf{x}_i, \mathbf{m}_\delta) = ||\phi(\mathbf{x}_i) - \mathbf{m}_\delta||^2$ between the vectors $\mathbf{x}_i$ and $\mathbf{m}_\delta$ can be written as:

$$D(\mathbf{x}_i, \mathbf{m}_\delta) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_i) - 2\phi(\mathbf{x}_i)^T \mathbf{m}_\delta + \mathbf{m}_\delta^T \mathbf{m}_\delta. \tag{2}$$

By substituting $\mathbf{m}_\delta$ from (1) into (2), we get:

$$D(\mathbf{x}_i, \mathbf{m}_\delta) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_i) - 2\frac{\sum_{a_j \in C_\delta} \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)}{|C_\delta|} + \\ + \frac{\sum_{a_j \in C_\delta} \sum_{a_l \in C_\delta} \phi(\mathbf{x}_j)^T \phi(\mathbf{x}_l)}{|C_\delta|^2} =$$

we can calculate the dot products using the kernel function:

$$D(\mathbf{x}_i, \mathbf{m}_\delta) = \kappa(\mathbf{x}_i, \mathbf{x}_i) - 2\frac{\sum_{a_j \in C_\delta} \kappa(\mathbf{x}_i, \mathbf{x}_j)}{|C_\delta|} + \\ + \frac{\sum_{a_j \in C_\delta} \sum_{a_l \in C_\delta} \kappa(\mathbf{x}_j, \mathbf{x}_l)}{|C_\delta|^2} =$$

Since the kernel function results are stored in the kernel matrix, $\kappa(\mathbf{x}_i, \mathbf{x}_j) = K_{ij}$, we finally obtain

$$D(\mathbf{x}_i, \mathbf{m}_\delta) = K_{ii} - 2\frac{\sum_{a_j \in C_\delta} K_{ij}}{|C_\delta|} + \frac{\sum_{a_j \in C_\delta} \sum_{a_l \in C_\delta} K_{jl}}{|C_\delta|^2} \tag{3}$$

After measuring the distance of data sample $\mathbf{x}_i$ to each of the k clusters centers, the data sample is reassigned to the cluster C_δ with the minimum distance $D(\mathbf{x}_i, \mathbf{m}_\delta)$. This is an iterative process, in which the distances are measured and the cluster assignments are updated, until there are no more changes in the cluster entry assignments, or a maximum number of iterations has been reached. The initial cluster entry assignments can either be manual, or completely random.

III. Distributed Kernel k-Means

In this section, we provide the algorithms that implement Nearest Neighbor and ϵ-ball Kernel k-Means in a distributed fashion, following the MapReduce programming model. We shall begin with a small introduction to the MapReduce model itself and then proceed to detail each major algorithmic step in this framework, namely the kernel matrix computation, the kernel matrix trimming algorithm and the Kernel k-Means algorithm, in separate subsections.

A. MapReduce computing framework

The *MapReduce* programming model for distributed computing was inspired by the map and reduce procedures of functional programming languages, like Lisp [28]. MapReduce implementations include Hadoop and Spark. It simplifies the coding of distributed programs that follow this model. It was specifically developed to allow easy processing of very big datasets on computing clusters consisting of many workers. A master node in the MapReduce framework automatically splits the dataset up into smaller data sample collections and distributes them to the workers, where each worker can process the assigned data collection, independently of other workers.

For example, if our goal is to compute the squared sum of a very large vector of numbers, we can map the square function on each vector entry and then reduce the results with the addition operation. The system will distribute the vector entries to every worker, then each worker will square the assigned vector entries, sum them up then return the partial sum to the master, which will add up all the partial sums it received from all the workers to compute the final result. For a more theoretical analysis of the MapReduce model, the interested reader may refer to [29].

As the name implies, there are two major components to this programming model. The *Map* command, in which every worker applies a user defined function to each data sample. Each worker can then return the results to the master node, thus computing that function output for the entire dataset. Additionally using the *Reduce* command, a worker applies a commutative and associative operation to collect the data elements, or the results of a previously mapped function, into a single result. As the operation is commutative and associative, the results for each worker are independent from other workers and they can also be combined in the same way on the master node. A variation of the Reduce command is ReduceByKey, in which, given a distributed set of (*key*, *value*) pairs and a target operation, the operation is performed on the *value* parts for each key separately. If there were k total keys, then the output would be a k (*key*, *total*) pairs, where each *total* is the result of performing the operation only on the *value* parts that are associated with the specific *key*.

For our implementation, we chose the Apache Spark [25] cluster computing framework. Its main advantage over Hadoop is its ability to cache distributed data into the worker memories, while automatically "spilling" excess data that cannot fit to the hard disk and reading them back, whenever they are needed. This reduces or, at best, eliminates the time spent reading from and writing to the disk. Our main goal, therefore, is to reduce the size of the data that must be stored in the distributed memory as much as possible, so that data spilling to the hard disk is minimized.

B. Distributed kernel matrix computation

Computing the kernel matrix under the MapReduce model is pretty straight forward. Assuming there are n data samples, each of which has d features, we read the data samples into n d-dimensional data vectors, which are distributed to the cluster worker nodes. Then we iterate through every data vector and map the kernel function of the current vector with every other vector. This provides us with a single row of the kernel matrix, which we can then write to the disk. After n iterations, the computation is complete. This step requires $O(nd)$ distributed memory and $O(n^2 d)$ operations.

The distributed operations are illustrated in Figure 1. In that particular example, worker 1 has received the d-dimensional data samples $\mathbf{x}_1$, $\mathbf{x}_2$ and $\mathbf{x}_3$, worker j has received data samples $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$ and the last worker w has the last data sample $\mathbf{x}_n$. At the i-th iteration, the kernel function $\kappa(\_, \mathbf{x}_i)$ is mapped to very data sample, where _ (underscore) is replaced with the corresponding data sample and $\mathbf{x}_i$ is the i-th data sample. In practice, in order to cut down on overhead costs and maximize CPU utilization, it is a good idea to use map to compute batches of, e.g., 100 lines of the kernel matrix at a time. It is also possible to fork a new thread to write the output, so that it will not delay the distributed computations.

C. Distributed kernel matrix trimming

After the kernel matrix has been computed and written to the disk, we read the n-dimensional kernel matrix rows and distribute them to the cluster nodes. Note that we shall never need every one of these rows in memory at the same time. Therefore, the framework can swap them to and from the disk, whenever any row is needed. This is an iterative process, in which the nodes vote for cluster cardinalities, the winning cardinality is determined, the votes of the nodes that voted for the winner are removed and the corresponding rows are trimmed.

We begin an iteration by mapping a sorting function on every matrix row. We then set the cut-off threshold to either the value of ϵ for ϵ-ball, or the value of the appropriate element of the sorted row, which corresponds to the desired number for Nearest Neighbors. This process takes $O(n^2 \log n)$ operations, due to the fact that every row of the kernel matrix is sorted. The distributed operations are illustrated in Figure 2. The workers are not shown, to avoid cluttering the figure. The _ (underscore) in functions is replaced with the corresponding vector of the previous step.

D. Distributed Kernel k-Means

In order to save memory, instead of reading the full kernel $n \times n$ matrix $\mathbf{K}$ as a set of n n-dimensional data vectors, we instead read the trimmed kernel matrix that resulted from the previous step as a set of n adjacency lists. The adjacency list for row $\mathbf{k}_i^*$ contains all the non-zero K_{ij}^* of the trimmed kernel matrix $\mathbf{K}^*$.

We initialize the data sample assignment to clusters randomly. The assignment is a n-dimensional vector $\mathbf{o}$, where the i-th entry indicates which cluster (1 to k) data sample a_i belongs to. This assignment is updated at every iteration and will eventually contain the final cluster assignment of every data sample.

We will now provide an algorithm to compute (3) in a distributed fashion. Note that the sum $\sum_{a_j \in C_\delta} \sum_{a_l \in C_\delta} K_{jl}$ remains the same for every individual cluster C_δ. Therefore, it only must be computed once for each corresponding cluster. The first step is to compute the k such sums. This can be accomplished by mapping a function that takes the cluster assignment vector $\mathbf{o}$, the node ID j and the node adjacency

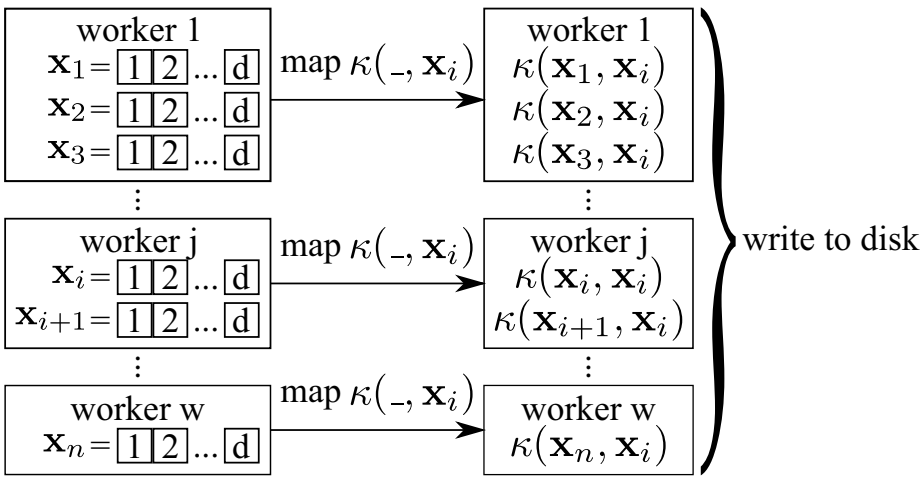

Fig. 1. Illustrated example of the distributed kernel matrix computation algorithm.

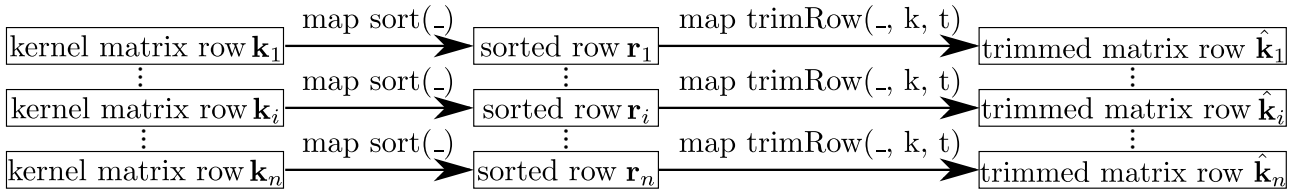

Fig. 2. Illustrated example of the distributed kernel matrix trimming algorithm.

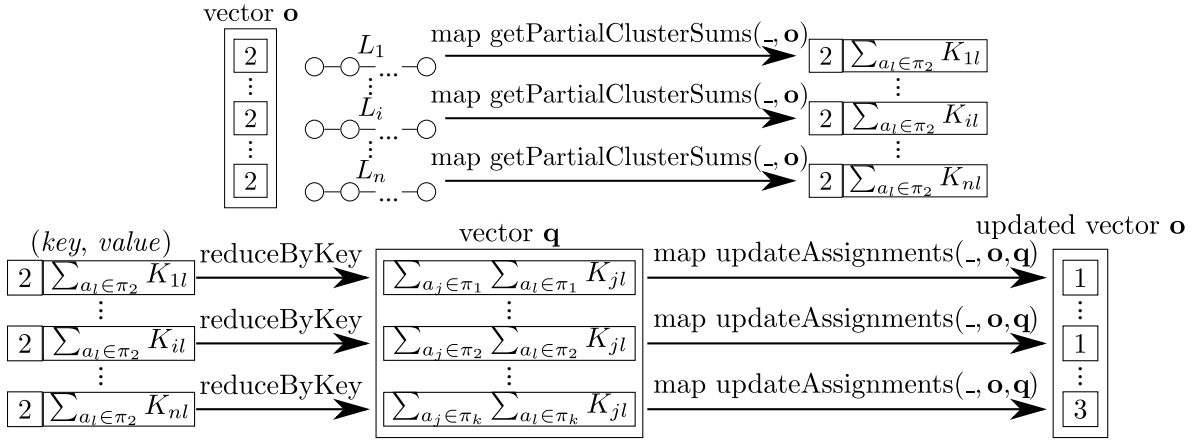

Fig. 3. Illustrated example of the distributed Kernel k-Means algorithm.

TABLE I. THE RESULTS OF THE EXPERIMENTS. THE TYPE OF KERNEL, THE NAME OF THE APPROACH, THE TIME REQUIRED FOR THE DISTRIBUTED COMPUTATIONS AND THE NMI PERFORMANCE ARE PROVIDED FOR EACH ROW. OUR APPROACH IS LABELLED AS *DKKM*, WHILE APPROXIMATE KERNEL k-MEANS IS LABELLED AS *AKKM*. NMI VALUES ARE PRESENTED AS A *mean* (*standard deviation*) PAIR. THE ROWS ARE SORTED IN ASCENDING NMI ORDER.

Kernel and approach	Kernel matrix calculation time	Kernel matrix trimming time	Kernel k-Means execution time	NMI
RBF kernel, 0.997-ball DKMM	40 min	59 min	5.3 min	0.2546(0.0121)
RBF kernel, 0.996-ball DKKM	40 min	66 min	4.8 min	0.3079(0.0223)
RBF kernel, AKKM	N/A	N/A	24 min ([30])	0.4936(0.0136)
Polynomial kernel, AKKM	N/A	N/A	24 min ([30])	0.4945(0.0136)
Neural kernel, AKKM	N/A	N/A	25 min ([30])	0.4982(0.0226)
RBF kernel, 100-NN DKKM	40 min	45 min	3.3 min	0.5476(0.0468)
RBF kernel, 1000-NN DKKM	40 min	53 min	7.2 min	0.5835(0.0540)

TABLE II. CLUSTERING ACCURACIES OF THE METHODS IN [34] AND OUR APPROACH.

Method	Accuracy (%)
3-NN Kernel k-Means	36.2 ± 2.95
5-NN Kernel k-Means	36.2 ± 3.373
9-NN Kernel k-Means	36.72 ± 0.71
25-NN Kernel k-Means	39.56 ± 1.86
Unsupervised Logistic Discriminative Metric Learning-kmeans [34]	44.08 ± 2.8
0.98-ball Kernel k-Means	44.51 ± 2.35
0.985-ball Kernel k-Means	44.96 ± 1.08
Penalized Probabilistic Clustering [34]	46.07 ± 5.52
100-NN Kernel k-Means	46.76 ± 3.62
Unsupervised Logistic Discriminative Metric Learning-clustering [34]	49.29 ± 0
Hidden Markov Random Fields-com [34]	50.30 ± 2.73
1000-NN Kernel k-Means	50.53 ± 4.21

list L as arguments and returns a (*key*, *value*) pair. In such a pair, *key* is the cluster that data sample a_j is assigned to (o_j) and *value* is the partial sum $\sum_{a_l \in C_\delta} K_{jl}$, where C_δ is the cluster identified by *key* and the entries K_{jl} are retrieved from the adjacency list L. The function goes through the node adjacency list and sums every entry that belongs to the same cluster as node j. The total sums for every cluster are obtained from these (*key*, *value*) pairs by applying the ReduceByKey operation to add the appropriate partial sums for each cluster and store them in vector $\mathbf{q}$.

In the next distributed processing step, the distance computations are completed and the new node assignments are determined in the same function. This is accomplished by mapping a function that takes the cluster assignment vector $\mathbf{o}$, the node ID i and the cluster sums vector $\mathbf{q}$ as arguments and returns the new cluster assignment for node i. The function initializes a vector $\mathbf{d}_i$, which is meant to store the distance of data sample a_i to every cluster, so each entry is initialized to $\mathbf{d}_{i\delta} = K_{ii} + \frac{1}{|C_\delta|^2}\mathbf{q}_\kappa$. It then goes through i node adjacency list L_i and subtracts the corresponding values $2\frac{K_{ij}}{|C_\delta|}$ from the appropriate entry in vector $\mathbf{d}$. When it goes through the entire list, then each entry of vector $\mathbf{d}$ will contain the value of $-2\frac{\sum_{a_j \in C_\delta} K_{ij}}{|C_\delta|} + \frac{\sum_{a_j \in C_\delta} \sum_{a_l \in C_\delta} K_{jl}}{|C_\delta|^2}$ for every cluster. The new cluster assignment of node i is determined by the minimum entry in vector $\mathbf{d}$. This final step requires $O(n_z)$ operations and memory space, where n_z is the number of non-zero entries of the trimmed kernel matrix. Note that this distributed algorithm can also work on the full kernel matrix, by using $O(n^2)$ operations and memory space. The distributed operations are illustrated in Figure 3. The workers are not shown, to avoid cluttering the figure. The _ (underscore) in functions is replaced with the corresponding list or vector of the previous step. In that particular example, data samples a_1, a_i and a_n, with their corresponding adjacency lists L_1, L_i and L_n, are initially assigned to cluster 2, as shown in assignment vector $\mathbf{o}$. Mapping getPartialClusterSums(_, $\mathbf{o}$) provides the (*key*, *value*) pairs (2, $\sum_{a_l \in C_2} K_{1l}$), (2, $\sum_{a_l \in C_2} K_{il}$) and (2, $\sum_{a_l \in C_2} K_{nl}$) for data samples 1, i and n, respectively. After the reduceByKey operation, the values are added, along with the results of all other data sample assigned to cluster 2, and are stored in the second entry of vector $\mathbf{q}$. Note that $\mathbf{q}$ has k entries, as there are k clusters. Vector $\mathbf{q}$ is passed as an argument, when mapping updateAssignments(_,$\mathbf{o}$,$\mathbf{q}$) to obtain the new assignments. In this case, data samples a_1 and a_i were reassigned to cluster 1, while data sample a_n was reassigned to cluster 3.

IV. EXPERIMENTS

In order to evaluate the performance of our distributed Nearest Neighbor and ϵ-ball Kernel k-Means in a large database, we used the MNIST handwritten digits database. This is a dataset of grayscale small images, each depicting a handwritten digit, 0-9. It contains 70000 total images, almost equally, but not exactly, clustered to the respective 0-9 digits. We will compare our approaches with Approximate Kernel k-Means [30], whose performance is almost identical to baseline Kernel k-Means.

In accordance with [31] and [30] for the MNIST handwritten digit dataset, each sample image was concatenated into a vector, then each feature of the vector was divided by 255, thus normalizing every image feature in $[0, 1]$. The following kernel functions were used: the *Neural* kernel $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\alpha \mathbf{x}_i^T \mathbf{x}_j + \beta)$, the *Polynomial* kernel $\kappa(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^T \mathbf{x}_j + 1)^d$ and the *Radial Basis Function* (RBF) kernel $\kappa(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma ||\mathbf{x}_i - \mathbf{x}_j||^2}$. Again, in accordance with [31] and [30], we set $\alpha = 0.0045$, $\beta = 0.11$ and $d = 5$. For the RBF kernel, we chose $\gamma = 1$. We used the *Normalized Mutual Information* (NMI) metric [32] to measure the similarity between the clustering results and the ground truth. Each approach was executed 10 times. The computing cluster consisted of 8 VirtualBox Virtual Machines, each of which had 2 cores and 4GB of RAM available. The results of the experiments are presented in Table I.

Overviewing the results, we can see that the running times for the various distributed steps is similar among all relevant entries. In this dataset, the ϵ-ball approach seems to fare rather poorly, providing the worst result. Approximate Kernel k-Means provide the middle ground of NMI performance. Our approach with 100 and 1000 Nearest Neighbor graphs provides the best performance, with the 1000 Nearest Neighbor one giving the absolute best result, 0.5835.

In order to evaluate the performance of our distributed Nearest Neighbor and ϵ-ball Kernel k-Means with a state of the art facial image clustering approach, we used the BF0502 dataset. This dataset contains descriptors of the faces of the protagonists of the 2-nd episode of the 5-th season of the TV series "Buffy, the Vampire Slayer" [33]. The 17000 images are the result of facial image tracking. This dataset is used to compare our approach with the one presented in [34],

that utilizes constraints derived from the facial image tracking trajectories to subsample and improve results.

The RBF kernel was used in this instance. We calculated the clustering accuracy of our algorithm in the same fashion as in [34], by constructing the confusion matrix and measuring the trace of that matrix, divided by the total images. We used 3-, 5-, 9-, 25-, 100- and 1000-Nearest Neighbor Kernel k-Means, as well as ϵ-ball Kernel k-Means with values 0.98 and 0.985 on this dataset. Again, in accordance with [34], we run our methods 30 times and measured the performance percentages as *mean*$\pm$*standard deviation*. The kernel computation took about 4.7 minutes, the trimming took at most 3.4 minutes and the Kernel k-Means algorithm took about 4.2 minutes. Table II presents the results of our approaches and the best results several methods reported in [34] in increasing clustering accuracy order. The accuracy of our approach for 1000-Nearest Neighbor (50.53%) surpasses the performance of the state of the art approach (50.30%).

V. Conclusion

In this paper, we have presented how the various steps of Nearest Neighbor and ϵ-ball Kernel k-Means, namely the kernel matrix computation, the kernel matrix trimming and the Kernel k-Means algorithm, can be performed in a distributed fashion. The algorithms were implemented using the Spark cluster computing MapReduce framework. Our Nearest Neighbor approach can provide improved results over baseline Kernel k-Means and Approximate Kernel k-Means in under 2 hours.

Acknowledgment

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 316564 (IMPART). This publication reflects only the authors' views. The European Union is not liable for any use that may be made of the information contained therein.

References

[1] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *ACM Comput. Surv.*, vol. 31, pp. 264–323, Sept. 1999.

[2] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, pp. 281–297, 1967.

[3] B. Schölkopf, A. Smola, and K.-R. Müller, "Nonlinear component analysis as a kernel eigenvalue problem," *Neural Comput.*, vol. 10, pp. 1299–1319, July 1998.

[4] A. Aizerman, E. M. Braverman, and L. I. Rozoner, "Theoretical foundations of the potential function method in pattern recognition learning," *Automation and Remote Control*, vol. 25, pp. 821–837, 1964.

[5] S. Yu, L.-C. Tranchevent, X. Liu, W. Glanzel, J. A. Suykens, B. D. Moor, and Y. Moreau, "Optimized data fusion for kernel k-means clustering," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 5, pp. 1031–1039, 2012.

[6] F. Zhou, F. De la Torre Frade, and J. K. Hodgins, "Hierarchical aligned cluster analysis for temporal clustering of human motion," *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, vol. 35, pp. 582–596, March 2013.

[7] H. Jia, Y. ming Cheung, and J. Liu, "Cooperative and penalized competitive learning with application to kernel-based clustering," *Pattern Recognition*, no. 0, pp. –, 2014.

[8] M. R. Ferreira and F. de A.T. de Carvalho, "Kernel-based hard clustering methods in the feature space with automatic variable weighting," *Pattern Recognition*, no. 0, pp. –, 2014.

[9] M. Filippone, F. Camastra, F. Masulli, and S. Rovetta, "A survey of kernel and spectral methods for clustering," *Pattern Recognition*, vol. 41, no. 1, pp. 176 – 190, 2008.

[10] D.-W. Kim, K. Y. Lee, D. Lee, and K. H. Lee, "Evaluation of the performance of clustering algorithms in kernel-induced feature space," *Pattern Recognition*, vol. 38, no. 4, pp. 607 – 611, 2005.

[11] I. S. Dhillon, Y. Guan, and B. Kulis, "Kernel k-means: spectral clustering and normalized cuts," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '04, (New York, NY, USA), pp. 551–556, ACM, 2004.

[12] F. R. B. M. I. Jordan and F. Bach, "Learning spectral clustering," *Advances in Neural Information Processing Systems*, vol. 16, pp. 305–312, 2004.

[13] U. von Luxburg, "A tutorial on spectral clustering," *Statistics and Computing*, vol. 17, no. 4, pp. 395–416, 2007.

[14] M. Lucińska and S. T. Wierzchoń, "Spectral clustering based on k-nearest neighbor graph," in *Computer Information Systems and Industrial Management*, pp. 254–265, Springer, 2012.

[15] M. Maier, U. von Luxburg, and M. Hein, "Influence of graph construction on graph-based clustering measures," in *Advances in neural information processing systems 21*, pp. 1025–1032, 6 2009.

[16] E. Arias-Castro, G. Chen, G. Lerman, *et al.*, "Spectral clustering based on local linear approximations," *Electronic Journal of Statistics*, vol. 5, pp. 1537–1587, 2011.

[17] A. Y. Ng, M. I. Jordan, Y. Weiss, *et al.*, "On spectral clustering: Analysis and an algorithm," *Advances in neural information processing systems*, vol. 2, pp. 849–856, 2002.

[18] L. Zelnik-Manor and P. Perona, "Self-tuning spectral clustering," in *Advances in neural information processing systems*, pp. 1601–1608, 2004.

[19] J. A. Hartigan and P. M. Hartigan, "The dip test of unimodality," *The Annals of Statistics*, p. 7084, 1985.

[20] A. Kalogeratos and A. Likas, "Dip-means: an incremental clustering method for estimating the number of clusters," in *Advances in Neural Information Processing Systems*, pp. 2393–2401, 2012.

[21] D. Agrawal, S. Das, and A. El Abbadi, "Big data and cloud computing: Current state and future opportunities," in *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, (New York, NY, USA), pp. 530–533, ACM, 2011.

[22] R. L. Ferreira Cordeiro, C. Traina, Junior, A. J. Machado Traina, J. López, U. Kang, and C. Faloutsos, "Clustering very large multi-dimensional datasets with mapreduce," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, (New York, NY, USA), pp. 690–698, ACM, 2011.

[23] L. M. Rodrigues, L. E. Zárate, C. N. Nobre, and H. C. Freitas, "Parallel and distributed kmeans to identify the translation initiation site of proteins," in *SMC*, pp. 1639–1645, IEEE, 2012.

[24] A. Ene, S. Im, and B. Moseley, "Fast clustering using mapreduce," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, (New York, NY, USA), pp. 681–689, ACM, 2011.

[25] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.

[26] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st ed., 2009.

[27] I. S. Dhillon, Y. Guan, and B. Kulis, "Weighted graph cuts without eigenvectors a multilevel approach," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, pp. 1944–1957, Nov. 2007.

[28] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[29] H. Karloff, S. Suri, and S. Vassilvitskii, "A model of computation for mapreduce," in *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, (Philadelphia, PA,

USA), pp. 938–948, Society for Industrial and Applied Mathematics, 2010.

[30] R. Chitta, R. Jin, T. C. Havens, and A. K. Jain, "Approximate kernel k-means: solution to large scale kernel clustering.," in *KDD* (C. Apte', J. Ghosh, and P. Smyth, eds.), pp. 895–903, ACM, 2011.

[31] R. Zhang and A. I. Rudnicky, "A large scale clustering scheme for kernel k-means.," in *ICPR (4)*, pp. 289–292, 2002.

[32] T. O. Kvålseth, "Entropy and correlation: Some comments," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 17, no. 3, pp. 517–519, 1987.

[33] M. Everingham, J. Sivic, and A. Zisserman, ""Hello! My name is... Buffy" – automatic naming of characters in TV video," in *Proceedings of the British Machine Vision Conference*, 2006.

[34] B. Wu, Y. Zhang, B.-G. Hu, and Q. Ji, "Constrained clustering and its application to face clustering in videos," *2013 IEEE Conference on Computer Vision and Pattern Recognition*, vol. 0, pp. 3507–3514, 2013.