A Distributed Framework for Trimmed Kernel k-Means Clustering

Nikolaos Tsapanos[a], Anastasios Tefas[a], Nikolaos Nikolaidis[a], Ioannis Pitas[a]

[a]*Aristotle University of Thessaloniki, BOX 54124, Thessaloniki, Greece.*

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. Kernel k-Means is a state of the art clustering algorithm. However, in contrast to clustering algorithms that can work using only a limited percentage of the data at a time, Kernel k-Means is a global clustering algorithm. It requires the computation of the kernel matrix, which takes $O(n^2d)$ time and $O(n^2)$ space in memory. As datasets grow larger, the application of Kernel k-Means becomes infeasible on a single computer, a fact that strongly suggests a distributed approach. In this paper, we present such an approach to the Kernel k-Means clustering algorithm, in order to make its application to a large number of samples feasible and, thus, achieve high performance clustering results on very big datasets. Our distributed approach follows the MapReduce programming model and consists of 3 stages, the kernel matrix computation, a novel matrix trimming method and the Kernel k-Means clustering algorithm.

1. 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 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.

An arising research trend is the so-called *Big Data* research. With the recent advances in technology, digital data is being generated, stored and broadcast at unprecedented rates. Digital cameras, including those in cell phones, are widely available and people all over the world are taking pictures or shooting video clips. The bandwidth of Internet Service Providers has also improved to the point where broadband connections are very common. The Web itself has been growing at ever increasing rates. The connection graphs of various social networks easily number nodes in the millions. As of late 2013, the Web itself has over 2 billion indexed pages. Bid data clustering is a challenging problem.

It has been observed that fuzzy c-means and other fuzzy clustering algorithms face

problems, when dealing with high dimensionality data, or large data sets [21]. Methods that involve eigenanalysis, such as the normalized graph cut approach [22] and even recent state of the art approaches [5] are also problematic. Kernel or similarity based clustering methods also have scalability issues, with respect to the required matrix calculation. Additionally, it is also possible that, even for manageable kernel or similarity matrix sizes, the data dimensionality can be in the order of several millions. The solution to this problem usually involves sub-sampling the features of each data sample, using e.g., Random Projections [23] or Conditional Random Sampling [24] approaches. In this paper, however, we will assume that the dimensionality of the data samples is small, compared to the number of data samples. This is the case, e.g., of moderately sized image clustering and image clustering after feature extraction [25].

Since the kernel matrix size is of $O(n^2)$, while time complexity is still an issue, it is the memory requirements that make it impossible to use Kernel k-Means to cluster datasets of this magnitude on average PCs, or even single high-end single machine systems. One way to work around this problem is provided by the Approximate Kernel k-Means algorithm [25]. Instead of using the full kernel matrix, only a user defined set of rows are calculated and used to measure distances and perform the clustering task.

Distributed computing can provide the means to handle problems on very large datasets that would otherwise be almost impossible to solve [26]. 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 [27]. Distributed versions of other clustering algorithms related to Kernel k-Means, like classic k-Means [28] and k-Medians [29] have already been discussed. However, to the best of our knowledge, a distributed approach to Kernel k-Means has not been proposed yet.

In this paper, we propose a distributed implementation of the Kernel k-Means clustering algorithm. We follow the MapReduce programming model [30], which is a high level framework for distributed processing on a computing cluster. The implementation uses Apache Spark [31], a cluster computing framework, which is similar to and

compatible with Hadoop [32]. 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. In order to achieve this goal, we employ a novel kernel matrix trimming algorithm, which enables us to significantly reduce the number of non-zero entries in the kernel matrix. This allows us to use memory-saving adjacency lists, instead of the full matrix, while also increasing clustering performance. The proposed distributed clustering scheme is divided into three major parts: kernel matrix computation, kernel matrix trimming algorithm and, finally, Kernel k-Means itself. The application used to perform the experimental evaluation of the proposed approach is image clustering.

The paper is organized as follows. Section 2 provides a brief introduction to the Kernel k-Means algorithm. Section 3 details the novel kernel matrix trimming algorithm. Section 4 describes the distributed computing approach to all the relevant algorithms. Section 5 presents the experiments carried out to evaluate the performance of the proposed method and study the scalability of its distributed implementation. Section 6 concludes the paper.

2. Trimmed Kernel k-Means

The Kernel k-Means algorithm [33] 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 $\Phi : \mathbb{R}^d \rightarrow \mathbb{R}^{d'}$ 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:

$$\begin{aligned} D(\mathbf{x}_i, \mathbf{m}_\delta) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_i) - 2\frac{\sum_{a_j \in C_\delta} \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)}{|C_\delta|} + \\ + \frac{\sum_{a_j \in C_\delta} \sum_{a_l \in C_\delta} \phi(\mathbf{x}_j)^T \phi(\mathbf{x}_l)}{|C_\delta|^2} = \\ = \kappa(\mathbf{x}_i, \mathbf{x}_i) - 2\frac{\sum_{a_j \in C_\delta} \kappa(\mathbf{x}_i, \mathbf{x}_j)}{|C_\delta|} + \\ + \frac{\sum_{a_j \in C_\delta} \sum_{a_l \in C_\delta} \kappa(\mathbf{x}_j, \mathbf{x}_l)}{|C_\delta|^2} = \\ = K_{ii} - 2\frac{\sum_{a_j \in C_\delta} K_{ij}}{|C_\delta|} + \frac{\sum_{a_j \in C_\delta} \sum_{a_l \in C_\delta} K_{jl}}{|C_\delta|^2} = \\ = K_{ii} - 2\frac{S_\delta^{(i)}}{n_\delta} + \frac{C_\delta}{n_\delta^2}, \end{aligned} \tag{3}$$

where $n_\delta = |C_\delta|$, $S_\delta^{(i)} = \sum_{a_j \in C_\delta} K_{ij}$, $T_\delta = \sum_{a_j \in C_\delta} \sum_{a_l \in C_\delta} K_{jl}$.

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.

In this paper, we propose a novel kernel matrix trimming algorithm that reduces the size of the clustering problem, while also improving clustering performance. We consider the kernel matrix entries to express data sample similarity. These entries have large/small values for within the same cluster/between different clusters, respectively. We aim to eliminate small K_{ij} entries, while retaining as many large K_{ij} entries as possible. In our proposed algorithm, it is possible to retain a different number of entries K_{ij} for different data samples. This is achieved by estimating the cardinality of the cluster that each sample belongs to. The cluster cardinality estimation is not performed

on a per sample basis, as all the data samples contribute, when making a decision that one or more clusters of a certain cardinality exist. Through a voting scheme, where each data sample votes for various different cardinalities, all the candidate cardinalities receive a score value and the cardinality with the highest score wins the voting round. The votes for the winning cardinality are removed and the voting process is repeated for the remaining cardinalities, until there are no more votes.

In order to provide a formalization of the reason that it is a good idea to trim small kernel matrix entries, we first attempt to answer the question "When is a data sample a_i assigned to the correct/wrong cluster?". Let us assume that there is a kernel matrix $\mathbf{K} : K_{ij} = K_{ji} \in [0, 1]$ containing the similarities, as obtained through the kernel function, between data samples from two clusters: cluster C_1 and cluster C_2. Let us also assume that $K_{ij} = 0$, if $a_i \in C_1, a_j \in C_2$ and $K_{ij} > 0$, if $a_i, a_j \in C_1$ or $a_i, a_j \in C_2$. A data sample $\mathbf{x}_i \in C_1$ is clustered into the correct cluster as long as:

$$\begin{aligned} & D(\mathbf{x}_i, \mathbf{m}_1) < D(\mathbf{x}_i, \mathbf{m}_2) \iff {}^{(3)} \\ \iff & K_{ii} - 2\frac{S_1^{(i)}}{n_1} + \frac{T_1}{n_1^2} < K_{ii} - 2\frac{S_2^{(i)}}{n_2} + \frac{T_2}{n_2^2} \iff \\ \iff & -2\frac{S_1^{(i)}}{n_1} + \frac{T_1}{n_1^2} < \frac{T_2}{n_2^2}, \end{aligned}$$

since $S_2^{(i)} = 0$. In order to exclude the effect of T_2 in our investigation, we temporarily assume that $T_2 \simeq 0$. This is possible, if every data sample of C_2 is sufficiently distant from every other sample in C_2. Thus, a_i is clustered into C_1 as long as the inequality:

$$-2\frac{S_1^{(i)}}{n_1} + \frac{C_1}{n_1^2} < 0 \tag{4}$$

holds. We shall now focus on the effect that the relationships of $\mathbf{x}_i$ with the other cluster samples have on the correct cluster assignment of $\mathbf{x}_i$. Note that $\frac{T_1}{n_1} = \frac{1}{n_1}\sum_{a_j \in C_1} S_1^{(l)}$ is the average of all $S_1^{(l)}$. Therefore:

$$\exists \beta > 0 : S_1^{(i)} = \beta\frac{T_1}{n_1} \iff T_1 = \frac{n_1 S_1^{(i)}}{\beta}.$$

By substituting T_1 in (4), we obtain:

$$-2\frac{S_1^{(i)}}{n_1} + \frac{n_1 S_1^{(i)}}{\beta n_1^2} < 0 \iff \frac{(1-2\beta)S_1^{(i)}}{\beta n_1} < 0 \iff$$
$$\iff 1 - 2\beta < 0 \iff \beta > \frac{1}{2}.$$

It is obvious that, if $\beta < \frac{1}{2}$, then cluster C_1 will lose sample a_i to cluster C_2, even though the effect of all entries related with C_2 on this decision is practically non-existent. On the other hand, if $\beta > \frac{1}{2}$ then

$$-2\frac{S_1^{(i)}}{n_1} + \frac{C_1}{n_1^2} < 0 \leq \frac{C_2}{n_2^2} \iff$$
$$\iff K_{ii} - 2\frac{S_1^{(i)}}{n_1} + \frac{C_1}{n_1^2} < K_{ii} \leq K_{ii} + \frac{C_2}{n_2^2} \iff$$
$$\iff D(\mathbf{x}_i, \mathbf{m}_1) < D(\mathbf{x}_i, \mathbf{m}_2).$$

This means that, as long as $\beta = \frac{n_1 S_1^{(i)}}{C_1} > \frac{1}{2}$ and $K_{jl} = 0$, if $a_j \in C_1, a_l \in C_2$, then it is impossible to assign a_i to the wrong cluster. There are two conclusions to be drawn at this point. Firstly, the compactness of the various clusters plays a significant role in the data sample assignment to clusters, as a very dense cluster is vulnerable to losing outlying samples to a neighboring sparse cluster, even if such samples have weaker ties to the sparse cluster, than to the dense cluster. Secondly, if all the between cluster entries were removed from the kernel matrix, then we would achieve perfect clustering, provided cluster compactness is not a factor.

In light of these observations, whether our approach provides a performance improvement depends on the distributions from which the data are generated and the desired ground truth. If the samples of a class are generated from multiple distributions with similar means, yet different deviations, and the ground truth groups the data from these distributions into the same cluster, then we would expect our approach to improve clustering performance, as we give more emphasis on proximity. If, on the other hand, the samples of each cluster are generated by a single distribution, but the distributions of different clusters have different deviations, then our approach is likely to incorrectly assign data samples from a sparser distribution to a cluster with a denser distribution. In such cases, approaches that provide more emphasis on local data sample density,

like the one in [18] are probably a better choice.

3. Kernel matrix trimming

In general, the proposed kernel matrix trimming algorithm attempts to determine the cardinality of the cluster that a data sample belongs to, through a voting system. Each data sample casts votes on the various candidate cluster cardinalities for itself. The votes for each cluster cardinality are summed up for every data sample. Each cardinality is then assigned a score by using a suitability function. The suitability function for cluster cardinality j essentially measures how close the number of votes for j is to the nearest integer non-zero product of j. For example, if the number of votes for cardinality 50 is 23, then cardinality 50 will not receive a very good score. If, on the other hand, the number of votes for cardinality 50 is 148, then this is a good indication that there might be 3 clusters of cardinality 50 and the suitability function score is accordingly high. The winning cardinality is the one with the highest score. Every data sample that voted for the winning cardinality value is determined to belong to a cluster of that cardinality and its votes are removed. The process is repeated on the remaining votes, with each cardinality receiving a new, updated score, until there are no votes left. We will proceed to describe this process in further detail.

This voting process is performed as follows. We begin by sorting each row $\mathbf{k}_i, i = 1, \ldots, n$ of the kernel matrix in ascending order, in a similar fashion to the Hartigan Dip Test for unimodality [19], resulting in a sorted vector $\mathbf{r}_i : r_{ij}, i = 1, \ldots, n$. We then numerically calculate the first derivative of r_{ij} as:

$$r'_{ij} = \frac{1}{3} \sum_{h=1}^{3} \frac{r_{i(j+h)} - r_{i(j-h)}}{2h}. \tag{5}$$

A high value of the first derivative implies that there is a significant data sample similarity gap between the data samples $\mathbf{x}_i, 1 \leq i < j$ and $\mathbf{x}_i, j \leq i \leq n$ and, thus, indicates a possible cluster cardinality j. In this view, $\mathbf{r}'_i$ is binarized to create a binary vote vector $\mathbf{v}_i$, $\mathbf{v}_i = [v_{i1}, v_{i2}, \ldots, v_{in}]^T$ containing the cluster cardinality votes as follows: $v_{ij} = 1$, if r'_{ij} is among the 10% of highest values of $r'_{ij}, j = 1, \ldots, n$, and $v_{ij} = 0$, otherwise. This threshold was selected because we considered it large

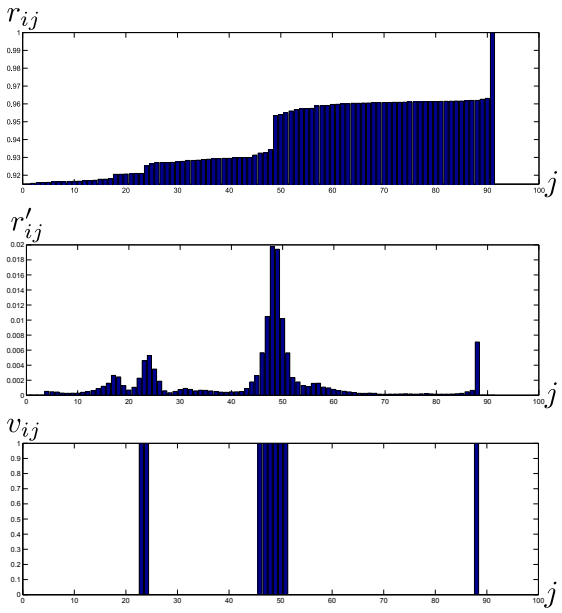


Figure 1: The suitability function vote determination process for a single data sample: a) the corresponding row values sorted in ascending order, b) the numerically calculated first derivative of the sorted sequence and c) the binary votes.

enough, so that potentially important votes will not be cut, while being small enough to avoid cluttering the votes. We present an experimental justification for this choice in Section 5. An example of such a binary vote vector is illustrated in Figure 1.

Subsequently, we add all the voting vectors for every data sample into vector $\mathbf{v}^* = \sum_{i=1}^{n} \mathbf{v}_i$, where v_j^* is the number of votes for cluster cardinality j. We calculate the score vector $\mathbf{s}$ from $\mathbf{v}^*$ as follows:

$$s_j = (1 - \frac{1}{j}) \max(e^{-|\frac{v_j^* - \lfloor \frac{v_j^*}{j} \rfloor j}{j}|}, e^{-|\frac{v_j^* - \lceil \frac{v_j^*}{j} \rceil j}{j}|}), \tag{6}$$

where $|\frac{v_j^* - \lfloor \frac{v_j^*}{j} \rfloor j}{j}|$ is the normalized distance of v_j^* to the closest integer product of j from below and $|\frac{v_j^* - \lceil \frac{v_j^*}{j} \rceil j}{j}|$, respectively, from above. Both of these distances are passed through an exponential function and the best result is retained. Finally, the score is weighed by a factor of $1 - \frac{1}{j}$, which represents the probability that the score is not the result of random chance. It is easy to see that any non-zero score for cardinality 2, for example, has a $1/2$ chance of being a perfect score. Since there are only j

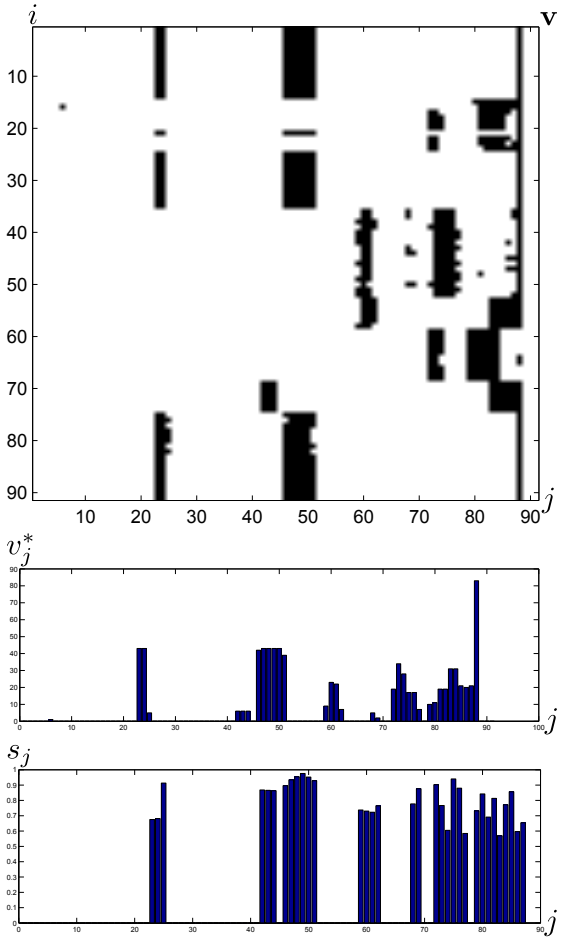


Figure 2: The voting and scoring process: a) the vote matrix, where each row represents the votes of the corresponding data sample in black, b) the column-wise sum of the voting matrix and c) the results of the scoring function, the winning cardinality is 43.

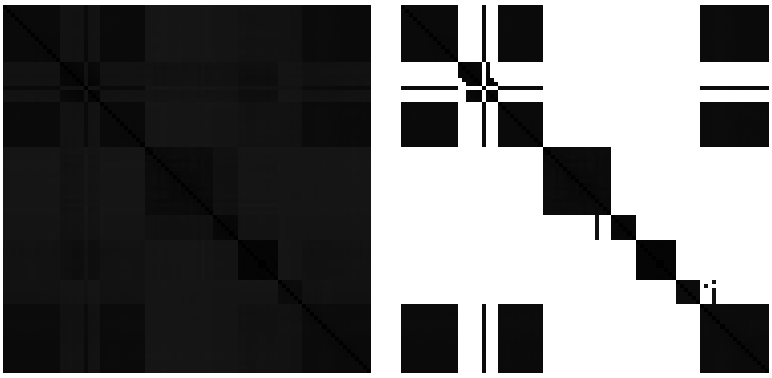
possible values for the unweighted score of cluster cardinality j, we assume that there is a $\frac{1}{j}$ probability that this happened by chance. The use of the factor $1 - \frac{1}{j}$ generally favours larger clusters and prevents the process from degenerating into finding a very big number of very small cluster cardinalities. The winning cluster cardinality $w = \arg\max_j(s_j)$ is selected. Every data sample a_i that voted for cluster cardinality w in its binary vote vector $\mathbf{v}_i$, i.e., $v_{iw} = 1$, is determined to belong to a cluster of cardinality w and its $\mathbf{v}^{(i)}$ is subtracted from $\mathbf{v}^*$ for the next iteration. The voting and

scoring process is illustrated in Figure 2.

When there are no more votes in $\mathbf{v}^*$, it means that every data sample has received an estimate of the cardinality of the cluster it belongs to. The trimming of the kernel matrix $\mathbf{K}$ entries is performed in a row-wise manner. Suppose that the estimated cluster cardinality for data sample a_i is w_i. We proceed to zero every entry K_{ij} in the i-th row of $\mathbf{K}$ whose value is less than the w_i-th largest value of the row. The pseudocode for the method described in this section can be found in Algorithm 1. Let $\hat{\mathbf{K}}$ be the resulting matrix, after every row of $\mathbf{K}$ has been trimmed. Since $\hat{\mathbf{K}}$ may no longer be symmetric, the final trimmed matrix is obtained as $\mathbf{K}^* = \max(\hat{\mathbf{K}}, \hat{\mathbf{K}}^T)$.

In order to provide an upper bound for the number of voting rounds as a function of n, we will assume that every data sample places its votes into its vote vector with a uniformly random distribution, until a percentage P of n individual cardinalities have been voted for. In our specific case, we set $P = 0.1$, or 10%. When a winning cardinality is determined, Pn of the active data samples voted for it on average. These data samples then have their votes removed, thus becoming inactive. The number of active data samples at voting round R is $(1-P)^R n$ on average, which means that this number is reduced exponentially, as the iterations proceed. The number of voting rounds required for the number of active data samples to drop to 1 is therefore $\log_{\frac{1}{P}} n$ on average. According to the logarithmic property that $\log_a x = \frac{\log_b x}{\log_b a}$, we come to the conclusion that the upper bound is $\log_{\frac{1}{P}} n = O(\log n)$ on average. In practice, votes are not random and we expect that several samples that belong to clusters of similar cardinality will become inactive, when a cardinality that lies within their voting range wins.

Note that the voting process can be implemented, using $O(n)$ memory and has a computational complexity of $O(n^2 \log n)$, since it requires the sorting of n sequences of size n. Subsequent executions of the Kernel k-Means algorithm can run in $O(n_z)$ time and memory, where n_z is the number of non-zero entries of $\mathbf{K}^*$ [34]. Also note that, as a by-product of this process, we also acquire an estimate for the total number of clusters and their cardinalities. Figure 3 shows the results of applying this process on a sample kernel matrix. The estimated cluster number is 6 clusters with cardinalities $43, 17, 10, 7, 7, 7$, while the ground truth is 6 clusters with cardinalities



(a) (b)

Figure 3: The trimming algorithm applied to a sample kernel matrix (black is 1, white is 0). a) Original kernel matrix. b) The trimmed kernel matrix.

$43, 17, 10, 9, 6, 6.$

4. The distributed Kernel k-Means framework

In this section, we provide the algorithms that implement every part of the proposed clustering scheme 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.

4.1. MapReduce computing framework

The *MapReduce* programming model for distributed computing was inspired by the map and reduce procedures of functional programming languages, like Lisp [30]. 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.

Algorithm 1 Kernel matrix trimming algorithm pseudocode.

```
let r_i be the i-th kernel matrix row
v* = 0
c = 0
for i=1 to n do
   numerically sort r_i
   calculate the derivative r'_i of the sorted r_i
   obtain votes v_i
   v* = v* + v_i
end for
while any v* ≠ 0 do
   calculate all scores according to (6)
   determine winning cardinality w
   for i=1 to n do
      if r_i voted for w then
         c_i = w
         v* = v* − v_i
      end if
   end for
end while
for i=1 to n do
   trim r_i according to c_i
end for
```

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 [35].

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 [31] 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.

4.2. 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 4. 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.

In terms of communication costs, the feature vectors will have to be distributed to the workers, Regardless of how many workers there are in the cluster, the total data to be transferred is $O(nd)$. During the computations, the master will have to send every feature vector to every worker, so that each worker can map the kernel function to it. If q is the number of workers, then the communication cost for the kernel matrix computation is $O(qnd)$. If the workers are also nodes in a distributed file system, then it is possible for the output to be written to a file in that distributed file system. If the output has to be sent back to the master, there is an additional communication cost of $O(n^2)$ involved.

4.3. 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

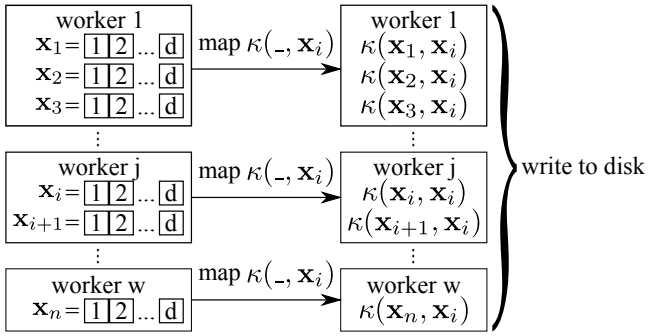


Figure 4: Illustrated example of the distributed kernel matrix computation algorithm.

and the corresponding rows are trimmed.

We begin an iteration by mapping a sorting function on every matrix row. We then map the numerical derivative computation function, as detailed in 5. Finally, we map a function that returns the vote vectors $\mathbf{v}_i$ of each node, as described in Section 3. We use the Reduce operation to add up all the voting vectors into the vector $\mathbf{v}^*$ containing the total votes for every cluster cardinality. The scoring function 6 is applied to this vector per data sample and the winning cluster cardinality w is determined.

In order to remove the winning votes, we map a function that takes the winning cluster cardinality w and outputs the vote vector $\mathbf{v}_i$ of a node, if that node voted for cardinality w, or an all-zero vector $\mathbf{0}$ otherwise. Again, we use the Reduce operation to obtain the vector summing the winning votes, which is then subtracted from the total votes vector $\mathbf{v}^*$, to obtain the remaining votes. As a final step of the iteration, we map the trimming function, which sets the row entry of every row that voted for w, which is not in the top w highest weights, to zero.

This completes an iteration step. The process is repeated, until the cluster cardinality of every node is determined and all the rows are trimmed accordingly. 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 5. The workers are not shown, to avoid cluttering the figure. The _ (underscore) in functions is replaced with the corresponding vector of the previous step. In that particular example, data sample a_i voted for the winning cardinality w, while data samples a_1 and a_n did not. As such,

only the kernel matrix row $\mathbf{k}_i$ of data sample a_i is trimmed at this iteration.

In practice, it is reasonable to expect that the voting vectors will contain long stretches of 0 entries, while the votes occur in intervals. This can be observed in Figure 1, where the derivative has 3 spikes that exceed the voting threshold and the resulting vote vector can be seen containing votes in 3 intervals. This makes the voting vector highly compressible, as one can simply use a triplet of the form $(start, finish, value)$ to represent an interval and, thus, use a list of such triplets to avoid storing the entire vector. This compression scheme is similar to *Run-Length Encoding* (*RLE*) [36]. Instead of adding the vectors to obtain the total votes, it is possible to merge any two lists of triplets to obtain the compressed form of the sum of every vote vector thus far. We used a scan line approach, in which we started from the beginning of both lists and processed the $start$ and $finish$ components of every triplet of each list in ascending order, carefully updating the current total $value$, and building the output list or triplets.

As an example, suppose that we have to merge the following triplets into one list: $(11, 20, 1)$, $(15, 25, 1)$ and $(17, 30, 1)$. We begin by merging $(11, 20, 1)$ and $(15, 25, 1)$ into $[(11, 14, 1), (15, 20, 2), (21, 25, 1)]$. We then merge $(22, 30, 1)$ into that result to obtain the final list $[(11, 14, 1), (15, 16, 2), (17, 20, 3), (20, 25, 2), (26, 30, 1)]$. Note that merging lists cannot result in a list that requires more than $3n$ variables to store, with the worst case being n triplets in which $start$ and $finish$ coincide and no pair of neighboring $value$s matches.

Using the vote compression scheme described above, each worker can load one row of the kernel matrix, sort it and then store the compressed vote vector list in memory, discarding the row. This means that, during the iterative voting process, the spilling to the disk can be minimized, or even eliminated. In a similar fashion, a trimmed row of the kernel matrix can be expressed as an adjacency list of neighbors, along with the associated kernel matrix entry for each neighbor. Let $r_1 <= 1$ denote the average compression ratio of the vote vectors, and $r_2 <= 1$ denote the average compression ratio of the trimmed rows. The distributed memory required for this step is $O(r_1 n^2 + r_2 n^2)$.

Regarding the communication costs, if the workers are also nodes in a distributed file system, where the full kernel matrix was written, then each worker can read the

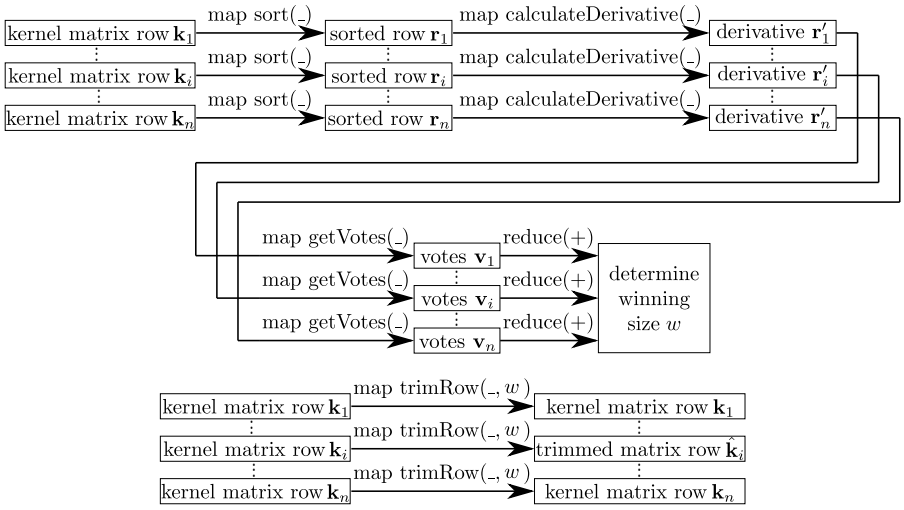


Figure 5: Illustrated example of the distributed kernel matrix trimming algorithm.

rows that have been assigned to it from that file system. Otherwise, the kernel matrix will have to be distributed to the workers for a total cost of $O(n^2)$. As already mentioned, the merged lists of the compressed vector votes are $O(n)$ in size. Let q denote the number of workers. For every Reduce operation, $q/2$ workers will have to send their merged lists to the other $q/2$ workers, in order for the lists to be further merged. After this, $q/4$ of the previous $q/2$ workers will have to send their result and so on. The total cost of Reduce operation is $O(n \sum_{d=1}^{\log q} \frac{q}{2^d}) = O(qn)$, since $\sum_{d=1}^{\infty} \frac{1}{2^d}$ converges to 1. If the voting process takes i_v iterations, then the total cost is $O(i_v qn)$. If the workers also need to send the trimmed rows to the master, because they are not part of a distributed file system, then there is an additional cost of $O(r_2 n^2)$.

4.4. 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

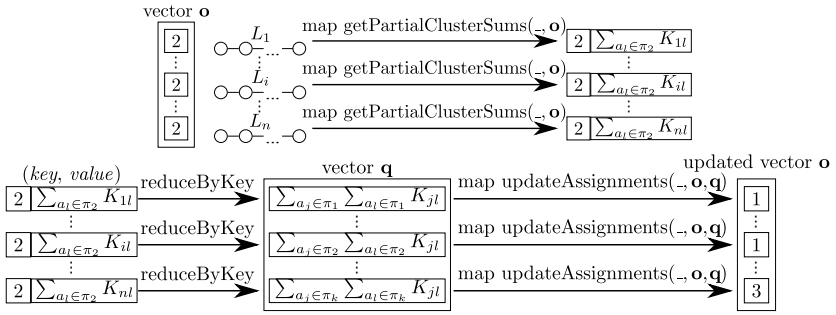


Figure 6: Illustrated example of the distributed Kernel k-Means algorithm.

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 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 6. 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.

Concerning the communication cost analysis for the distributed Kernel k-Means, we will begin by breaking down the cost of each relevant substep for one iteration. First, the master has to send the current labels to all workers. Assuming q workers, this has a communication cost of $O(qn)$. After this, the workers must perform the ReduceByKey operation to calculate the cluster sums, which, in a similar manner as the Reduce operation in the Kernel matrix trimming step, can be considered to have a cost of $O(qk)$. The master, then, has to send the current labels and the cluster sums to each work, which implies $O(qn + qk))$ communication cost. Finally, the workers must send the new labels to the master for an additional cost of $O(n)$. Let i_k denote the iterations of Kernel k-Means. The final communication cost for this step is $O(i_k qn + i_k qk + i_k qn + i_k n) = O(i_k qn + i_k qk)$

4.5. Relation to the MapReduce class of algorithms

A theoretical model for the efficiency of MapReduce computations is presented in [35]. It introduces the *MapReduce Class* ($\mathcal{MRC}$) of algorithms, which enforces limitations on the memory, number of processors and execution time of an algorithm that belongs to it. We summarize the definition of $\mathcal{MRC}^i$ from [35] here, for ease of reference:

Definition 1. *Fix an $\epsilon > 0$. An alforithm belongs to $\mathcal{MRC}^i$ if:*

- *The Map and Reduce operations are implemented by a RAM with $O(\log n)$ words, $O(n^{1-\epsilon})$ available space and execute in time polynomial to n.*
- *The total memory space required is $O(n^{2-2\epsilon})$.*
- *The number of MapReduce rounds R is $O(\log^i n)$.*

Note that the above restrictions imply the existence of $\Theta(n^{1-\epsilon})$ available machines, something that is also clearly stated in [35].

It is easy to see that, as is, our clustering scheme does not belong to any $\mathcal{MRC}^i$, as it has to load a matrix row of $O(n)$ elements, even momentarily before trimming and compressing it, in $O(n^{1-\epsilon})$ available space. However, if the input is subsampled to a size of $O(n^{1-\epsilon})$, then we can prove that all algorithms presented in this paper belong to $\mathcal{MRC}^1$. Our algorithms are designed to work on the standard 32/64-bit word architectures, so we will not further consider the $O(\log n)$ word size requirement. We will proceed to prove that each part of our clustering scheme belongs to $\mathcal{MRC}^1$. We fix $\epsilon = 0.5$.

4.5.1. Kernel matrix computation

Under our assumption in the introduction that the dimensionality of the data samples is small, compared to the number of data samples, we can reasonably assume that $d = O(n^{1-\epsilon})$. The total memory required to store the data is $nd = O(n^{2-2\epsilon})$. There are $\Theta(n^{1-\epsilon})$ machines to which data of size $O(n^{1-\epsilon})$ must be distributed. The computation time required for each MapReduce round is $O(n^{2-2\epsilon})$. In order for the computation to run in $O(\log n)$ rounds, we compute the kernel matrix in batches of

$n^{1-\epsilon}/\log n$ rows. The size of the rows is $O(n^{1-\epsilon}/\log n) = O(n^{1-\epsilon})$, which does not exceed the available memory. Therefore, our kernel matrix computation belongs to $\mathcal{MRC}^1$.

4.5.2. Kernel matrix trimming

The kernel matrix computation algorithm outputs a $n^{1-\epsilon} \times n^{1-\epsilon}$ matrix, which fits in $O(n^{2-2\epsilon})$ memory, with each of the $\Theta(n^{1-\epsilon})$ machines receiving $O(n^{1-\epsilon})$ data. The votes are determined in a single MapReduce round in $O(n^{1-\epsilon} \log n^{1-\epsilon})$ time. Each voting round executes in $O(n^{1-\epsilon})$ time. Recall from Section 3 that the number of voting rounds is logarithmically upper bound in average, therefore the number of MapReduce rounds is $O(\log n^{1-\epsilon}) = O(\log n)$. Our kernel matrix trimming algorithm belongs to $\mathcal{MRC}^1$.

4.5.3. Kernel k-Means

The trimmed kernel matrix does not exceed $O(n^{2-2\epsilon})$ in size. Again, each of the $\Theta(n^{1-\epsilon})$ machines receives $O(n^{1-\epsilon})$ data. The Map and Reduce operations execute in $O(n^{1-\epsilon})$ time. The maximum number of MapReduce rounds (Kernel k-Means iterations) is user defined could be argued to be a constant, thus placing the algorithm in $\mathcal{MRC}^0$. Since the rest of the algorithms are already in $\mathcal{MRC}^1$, we can also assume that the maximum iterations can be set to be $O(\log n)$, which places the algorithm and the entire clustering scheme in $\mathcal{MRC}^1$.

4.5.4. Subsampling

Regarding the subsampling required to reduce the input size to $O(n^{1-\epsilon})$, a simple approach is to select $n^{1-\epsilon}$ samples uniformly at random, then use our algorithms to cluster them. The remaining $n - n^{1-\epsilon}$ samples can each be assigned to the cluster of its nearest neighbor in a single MapReduce round. Alternatively, we can use the subsampling process described in [29], in which points are selected to represent other points close to them, until the size of the input is appropriately reduced.

5. Experiments

In this section, we will present the results of the conducted experiments, in order to evaluate the clustering performance and speed of the proposed distributed clustering

framework. Initially, we used the MATLAB implementation of the methods involved. However, for the really big datasets, we used the distributed implementation described above. The first set of experiments concerns the direct comparative evaluation of our kernel trimming algorithm against baseline and Approximate Kernel k-Means [25], in terms of both clustering performance and kernel matrix size reduction. The second set of experiments concerns a comparison with a state of the art facial image clustering approach. The third set of experiments was conducted, in order to study the run time speed up of our method, vs the number of available computing cores.

5.1. Computing cluster structure

We used VirtualBox to create a Virtual Machine (VM) on the computers of our lab. We installed Ubuntu and Spark on every VM. The VMs are connected over a Local Area Network (LAN). The computers that form the cluster include a high-end workstation, with 2 XEON processors with 10 cores each and 240GB of RAM, a few high-end PCs, with Core i7 processors and 16 GB of RAM or similar, and some average PCs with Core i5 and 8GB of RAM or less. An illustration of the cluster structure can be seen in Figure 7.

5.2. Datasets

We will now briefly describe the datasets used in our experiments. All of them consist of either image descriptors or the images themselves. Most of them were captured from video clips. The smallest dataset includes over 17000 data samples, while the largest contains 621126 data samples.

MNIST handwritten digits: 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. This is used to evaluate the performance improvement over the baseline Kernel k-Means and Approximate Kernel k-Means [25] and determine the trade-off between clustering performance and kernel matrix size reduction. It was selected, because it was also used in [25], where Approximate Kernel k-Means was proposed. The dataset is also used to study the behavior of the distributed implementation on clusters of various core numbers.

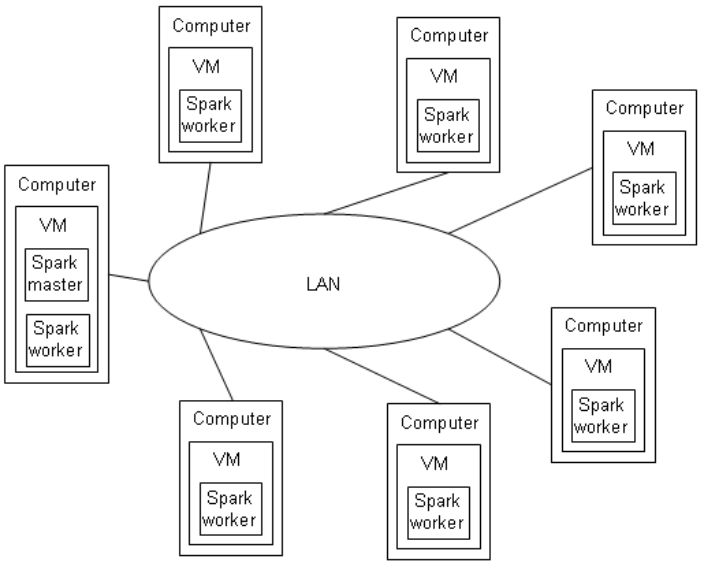


Figure 7: Computing cluster structure.

BF0502: 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" [37]. The 17000 images are the result of facial image tracking. This dataset is used to compare our approach with a recent, state of the art approach [38], that utilizes constraints derived from the facial image tracking trajectories to subsample and improve results.

Youtube Faces: Finally, in order to provide performance results on really Big Data and to evaluate the scaling of the proposed distributed approach runtime in relation to available computing cores, we used the Youtube Faces dataset. It consists of LBP descriptors for 621126 faces of various celebrities, e.g., actors, athletes and politicians, extracted from Youtube videos [39]. There are 3 different, yet closely related types of descriptors provided by the dataset: Local Binary Patterns (LBP) [40], Center-Symmetric LBP (CSLBP) [41] and Four-Patch LBP (FPLBP) [42]. For our experiments, we selected the original LBP features, which yield the best performance in [39]. The dimensionality of the feature vectors is 1770.

5.3. Clustering performance

In accordance with [43] and [25], in the case of 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 [43] and [25], we set $\alpha = 0.0045$, $\beta = 0.11$ and $d = 5$. For the RBF kernel, we chose $\gamma = 1$. For each function, the full kernel matrix $\mathbf{K}$ was calculated. We then used our algorithm, as described in Section 3, to obtain the trimmed kernel matrix $\mathbf{K}^*$ for every function. We run the Kernel k-Means algorithm 10 times each for all 6 possible approaches (baseline/proposed, Neural/Polynomial/RBF). We then used the *Normalized Mutual Information* (NMI) metric [44] to measure the similarity between the clustering results and the ground truth. We also measure the reduction in the size of the kernel matrix as $\frac{nz}{n^2}$. The results of this experiment are presented in Table 1, in which NMI values are presented as a *mean* (*standard deviation*) pair. We note that the clustering performance of the baseline RBF approach (0.4936) is slightly worse than those of both the baseline Neural (0.4982) and baseline Polynomial (0.4945) approaches. Looking at these results, one might think that the RBF kernel function is not the best choice for this problem. Furthermore, it appears that the proposed Trimmed Kernel k-Means algorithm hinders the Neural approach (0.4959), but provides enough improvement on the Polynomial approach (0.5108). However, the proposed RBF approach provides the absolute best performance (0.5687), with a good $0,0705$ lead over the second best approach. Looking at the kernel matrix size reduction, the proposed RBF approach retained only about 4% of the full kernel matrix, in order to achieve the best performance, while the other two proposed approaches used almost double that (8%). It appears that the RBF kernel suffers most from the presence of between cluster similarity entries in the kernel matrix, but has better properties regarding cluster compactness than the Neural and Polynomial kernels. Thus, it is able to outperform both, when most of the between cluster kernel matrix entries are removed.

In order to study the performance/kernel matrix size reduction trade-off, we used

Table 1: Experimental results on the MNIST dataset. The baseline column refers to using Kernel k-Means on the full kernel matrix, while the proposed column refers to using Kernel k-Means on the trimmed kernel matrix. The reduction column lists the ratio of retained over initial kernel matrix entries $\frac{n_z}{n^2}$ (percentage.

Kernel	NMI		Memory reduction ratio
	Baseline	Trimmed Kernel k-Means	
Neural	$0.4982(0.0226)$	$0.4959(0.0066)$	7.43%
Polynomial	$0.4945(0.0136)$	$0.5108(0.0095)$	8.66%
RBF	$0.4936(0.0136)$	$\mathbf{0.5687(0.0312)}$	$\mathbf{4.39\%}$

the Approximate Kernel k-Means algorithm on the same MNIST dataset. We randomly sampled 2000, 4000 and 5000 from the Neural matrix rows and run the experiments 10 times. The NMI performance and corresponding matrix size reduction (in percentages) achieved by Approximate Kernel k-Means can be seen in Table 2. That table also includes the best performance/reduction of our approach for quick reference. As can be seen, Approximate Kernel k-Means needs about 7% of the full kernel matrix, in order to match the full kernel matrix performance (0.4941), while our approach achieves better performance (0.5687) with about 4% of the kernel matrix size. However, since our approach requires adjacency lists, in practice it will require double that amount of memory (8%). Concluding this comparison, our approach will require about the same memory to run and yet provides a significant performance improvement over Approximate Kernel k-Means.

Table 2: Performance/reduction trade-off for the Approximate Kernel k-Means [25] and the Trimmed Kernel k-Means approach.

Method	NMI	Memory reduction ratio
Approximate	$0.4898(0.0067)$	2.85%
Kernel	$0.4917(0.0079)$	5.71%
k-Means	$0.4941(0.0124)$	7.14%
Trimmed Kernel k-Means	$0.5687(0.0162)$	4.39%

Additionally, in order to ensure that the improvement provided by the proposed kernel matrix trimming is better than kernel matrix trimming with a static cluster cardinality, we run the following experiments. Instead of dynamically determining the cluster cardinality for each data sample using our algorithm, we assumed that every data sample belongs to a cluster of the same cardinality and trimmed the RBF kernel matrix

accordingly. The cardinalities we used are the maximum cluster cardinality (7877), the average cluster cardinality (7000) and the minimum cluster cardinality (6313), according to the ground truth. The trimmed kernel matrix $\mathbf{K}^*$ was again made symmetric after the trimming process, in both trimming approaches as described in Section 3. The results are presented in Table 3, where we can see that our approach is indeed better than static cardinality determination. It must also be noted, that knowledge of the ground truth is needed to determine the cluster cardinalities in a static manner, while our proposed adaptive approach is completely oblivious of ground truth. Yet, he proposed method still provides better performance.

Table 3: Comparison between static cluster cardinality cuts and the proposed adaptively chosen ones.

Method	NMI	Memory reduction ratio
max cardinality	$0.4933(0.0123)$	16.15%
average cardinality	$0.5091(0.0063)$	14.35%
min cardinality	$0.5102(0.0160)$	12.94%
proposed	$0.5687(0.0162)$	4.39%

Furthermore, we compare the performance of our approach with a state of the art face image clustering scheme [38]. The dataset used for this comparison is BF0502, which includes 17000 facial images of the 6 main cast of the TV series "Buffy, the Vampire Slayer". In [38], each trajectory is represented by 3 randomly selected frames. Thus, the facial image dataset is subsampled. Additionally, [38] uses constraints derived from the tracking trajectories. These constraints fall into 2 categories: images appearing in the same trajectory must be included in the same cluster and images appearing in different, overlapping trajectories must not be included in the same cluster. Using our approach, we simply applied our algorithm on all the 17000 images, setting $k = 6$. The RBF kernel was used in this instance. We calculated the clustering accuracy of our algorithm in the same fashion as in [38], by constructing the confusion matrix and measuring the trace of that matrix, divided by the total images. Additionally, we also used 3-, 5-, 9-, 25- and 100-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 [38], we run our methods 30 times and measured the performance percentages as

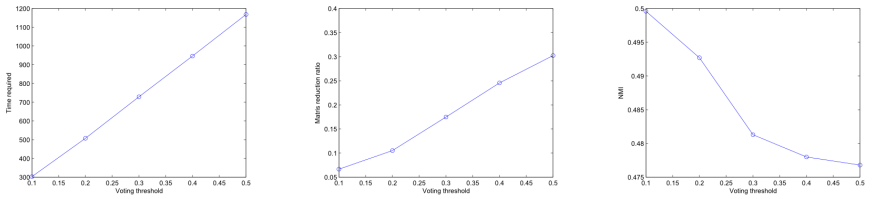
mean$\pm$*standard deviation*. Table 4 presents the results of our approaches and the best results several methods reported in [38] in increasing clustering accuracy order. The accuracy of our approach (49.96%) again improves upon the performance of baseline Kernel k-Means (47.93%) and closely rivals the performance of the state of the art approach (50.30%). Note that we did not take advantage of any constraints and viewed this problem as a general purpose clustering task.

We also use this dataset, in order to study the effect that the voting threshold has on all aspects of our trimming algorithm. Starting from the previously mentioned threshold of 10%, we proceeded to increase it by a 10% step up to and including 50%. We repeated the above experiment using the respective trimming to the kernel matrix. The results can be seen in Figure 8, which illustrates how the runtime, resulting kernel matrix size and classification performance is affected by the voting threshold value. Increasing the voting threshold slows down the trimming process. It also clutters the votes, resulting in larger estimated cluster cardinalities and, thus, larger trimmed kernel matrices. Finally, increasing the threshold also negativele affects the classification performance. We therefore would not recommend a threshold of over 10%.

Table 4: Clustering accuracies of the methods in [38], baseline Kernel k-Means (in bold) and our method (in bold).

Method	Accuracy
3-NN Kernel k-Means	36.2 ± 0.000295
5-NN Kernel k-Means	36.2 ± 0.0003373
9-NN Kernel k-Means	36.72 ± 0.0071
25-NN Kernel k-Means	39.56 ± 0.0186
Unsupervised Logistic Discriminative Metric Learning-kmeans [38]	44.08 ± 2.8
0.98-ball Kernel k-Means	44.51 ± 0.0235
0.985-ball Kernel k-Means	44.96 ± 0.0108
Penalized Probabilistic Clustering [38]	46.07 ± 5.52
100-NN Kernel k-Means	46.76 ± 0.0362
Baseline Kernel k-Means	$\mathbf{47.93 \pm 2.78}$
Unsupervised Logistic Discriminative Metric Learning-clustering [38]	49.29 ± 0
Trimmed Kernel k-Means	$\mathbf{49.96 \pm 2.85}$
Hidden Markov Random Fields-com [38]	50.30 ± 2.73

Finally, in order to evaluate the performance improvement and acceleration of our



(a) (b) (c)

Figure 8: The effect of the voting threshold on a) the time required by the trimming process, b) the kernel matrix size reduction and c) the classification performance on the BF0502 dataset.

distributed clustering scheme, we used the Youtube Faces dataset. This dataset contains $n = 621126$ samples, which is about 10 times larger than the MNIST dataset and requires the computation of a kernel matrix that is almost 100 times larger. The number of ground truth clusters is 1595. We restricted the voting process, so that no cluster cardinality over $0.01n$ can be selected, in order to ensure that the trimmed kernel matrix is significantly reduced in size. Since the feature vectors (LBP) provided by the database are histograms, we used the *Histogram Intersection* kernel, $\kappa(\mathbf{x}, \mathbf{y}) = \sum \min(x_i, y_i)$, for this experiment. We used our distributed kernel matrix computation algorithm to calculate 10000 rows of the kernel matrix at a time, in batches of 50 rows per Map. The resulting files were then concatenated into a single 1.9 Terabyte file. We then used our distributed kernel matrix trimming algorithm, in order to obtain the trimmed matrix, which retained 0.003 (0.3%) of the original kernel matrix entries and was 15 Gigabytes in size. Finally, we used our distributed kernel k-means algorithm on the trimmed kernel matrix, to obtain a NMI performance of $\mathbf{0.857}$.

We also used the Approximate Kernel k-Means algorithm on 2000 randomly selected rows of the kernel matrix, which results in about the same kernel matrix reduction of 0.003. Unfortunately, we were unable to measure the exact NMI performance of Approximate Kernel k-Means, as it missed about 100 clusters. To circumvent this, for every missing label, we randomly changed an existing label to that missing one. The resulting NMI was 0.8402. To ensure that this is not unreasonably unfair to Approximate Kernel k-Means, we tried the same random label change to the output of our approach, and the resulting NMI dropped from 0.857 to 0.8567.

5.4. Evaluation of computational speedup

For the purposes of testing the speedup of our approach in an actual computing cluster, we used the MNIST dataset, as it is the bigger of the two datasets. We formed a computing cluster having 200GB of distributed memory. The number of cores varied from 1 to 20. We did not include the time in which the intermediate output was written, as it is bottlenecked by disk speed, but we did include the time in which input was read, as it can affect the overhead of distributing the data to the worker nodes. We also fixed the number of iterations for Kernel k-Means to 10, so that possible early stops would not contaminate the speedup results.

Ideally, if the total computational time required by a single processing core is C_t and there are p equal processing cores available, then the best running time we can possibly achieve is $\frac{C_t}{p}$. Thus, the expected curve of the plot of time with respect to number of nodes is expected to have the form of the rectangular hyperbola $f(x) = \frac{1}{x}$. The relevant plots can be seen in Figure 9. As can be observed from the matrix computation curve in Figure 9a, we noticed some saturation issues arising, when using 20 cores, so the rest of the steps (kernel matrix trimming and Kernel k-Means) were performed using only up to 8 cores. All the curves reasonably follow the predicted rectangular hyperbola. In total, performing the clustering on a single core would take about 348 minutes, while only taking about 39 minutes using a maximum of 20 cores.

Finally, we present the computational time related results of our distributed clustering scheme on the Youtube Faces dataset. It was not practical to run the kernel matrix computation in its entirety for various numbers of cores, as it would take about 150 days for a single core to finish the task, according to our estimates. In order to study the acceleration scaling with respect to the number of cores, we measured the time required by the computing cluster to calculate 50 rows of the kernel matrix. We setup the computing cluster several times with a different number of VMs as workers. Each VM had 2 available cores and 4 Gigabytes of memory. For each computing cluster configuration, we allowed to it to calculate several batches of 50 kernel matrix rows. The figures were then collected and averaged. The resulting acceleration curve can be seen in Figure 10. Again, the curve reasonably follows the predicted rectangular hyperbola. In total, the kernel matrix computation required about 300 hours. The kernel

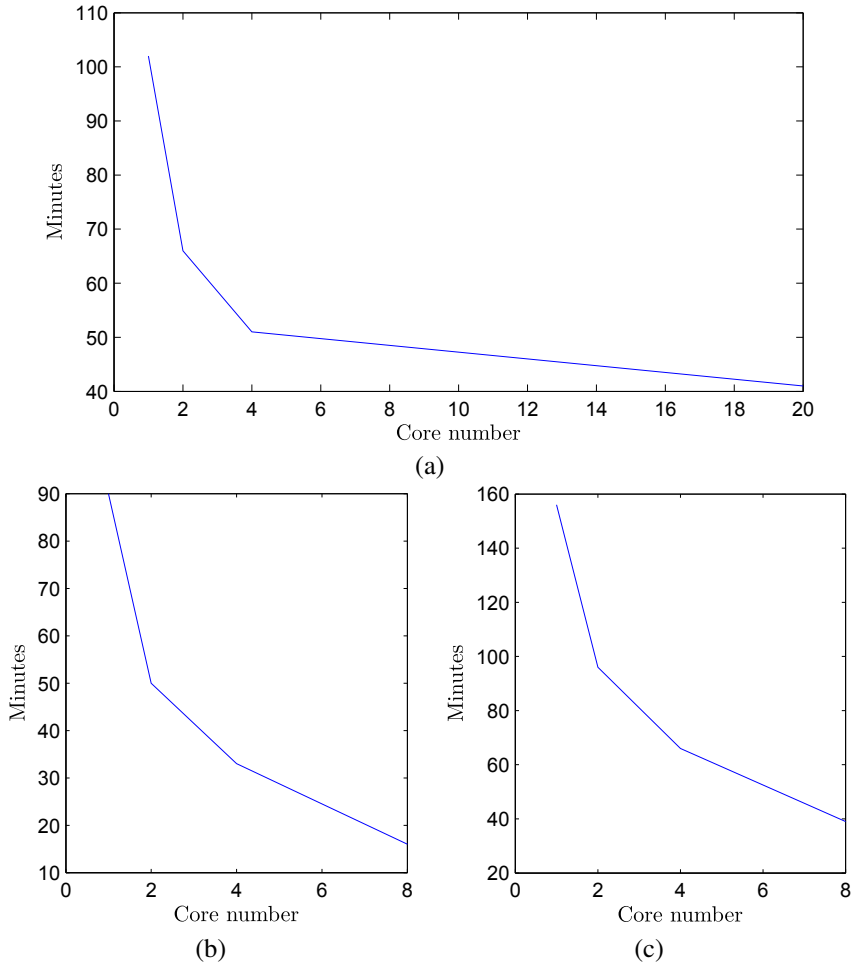


Figure 9: Minutes to finish task with respect to number of cores for a) kernel matrix computation, b) kernel matrix trimming and c) Kernel k-Means on the trimmed kernel matrix.

matrix trimming required 35 hours and Kernel k-means itself run in 6 hours. We would like to note that computing the 2000×621126 kernel matrix for Approximate Kernel k-Means in MATLAB required almost a day of computations, while our distributed approach can compute 2000 rows in about 35 minutes. With a simple modification, it is possible to use our distributed approach to compute a random collection of kernel matrix rows, which can then be used by Approximate Kernel k-Means. This combination provides a very powerful tool which can provide fast clustering results with little

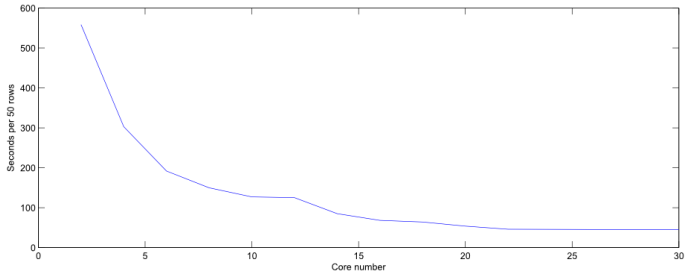


Figure 10: Seconds to compute 50 rows of the kernel matrix with respect to number of cores.

compromise in clustering performance.

6. Conclusions

In this paper, we have proposed a novel kernel matrix trimming algorithm that both improves the performance of Kernel k-Means, while significantly reducing the kernel matrix size. Through a voting scheme, we are able to estimate the cardinality of the cluster that each individual data sample belongs to. This provides a threshold, using which we can separate within cluster kernel matrix entries, i.e., entries connecting the data sample to other samples of its cluster, from between cluster entries, i.e., entries connecting the data sample to samples of other clusters. During the justification of our motivation, we also provided some insight into how cluster density in the kernel space can affect the assignment of data samples to particular cluster types, depending on their density.

Experimental results strongly indicate that our approach consistently provides an improvement over baseline Kernel k-Means, which is not possible by the static kernel matrix trimming, despite the fact that is does not use ground truth knowledge. Our approach is even almost equivalent in clustering accuracy to a state of the art face clustering approach, which though takes advantage of video tracking trajectory-derived constraints.

Additionally, we have provided a distributed implementation of all three steps of the proposed Trimmed Kernel k-Means framework. The distributed implementation is designed to minimize memory usage, thus ensuring that either the entire problem

can fit inside the distributed memory, or that the spilling of data to the disk is also minimized. The running times that were recoded while deploying our implementation to clusters of different numbers of cores reasonably follow the rectangular hyperbola curve with respect to the number of cores.

Acknowledgement

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).

References

[1] A. K. Jain, M. N. Murty, P. J. Flynn, Data clustering: a review, ACM Comput. Surv. 31 (3) (1999) 264–323.

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

[3] B. Schölkopf, A. Smola, K.-R. Müller, Nonlinear component analysis as a kernel eigenvalue problem, Neural Comput. 10 (5) (1998) 1299–1319.

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

[5] S. Yu, L.-C. Tranchevent, X. Liu, W. Glanzel, J. A. Suykens, B. D. Moor, Y. Moreau, Optimized data fusion for kernel k-means clustering, IEEE Transactions on Pattern Analysis and Machine Intelligence 34 (5) (2012) 1031–1039.

[6] F. Zhou, F. De la Torre Frade, J. K. Hodgins, Hierarchical aligned cluster analysis for temporal clustering of human motion, IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI) 35 (3) (2013) 582–596.

[7] H. Jia, Y. ming Cheung, J. Liu, Cooperative and penalized competitive learning with application to kernel-based clustering, Pattern Recognition (0) (2014) –.

[8] M. R. Ferreira, F. de A.T. de Carvalho, Kernel-based hard clustering methods in the feature space with automatic variable weighting, Pattern Recognition (0) (2014) –.

[9] M. Filippone, F. Camastra, F. Masulli, S. Rovetta, A survey of kernel and spectral methods for clustering, Pattern Recognition 41 (1) (2008) 176 – 190.

[10] D.-W. Kim, K. Y. Lee, D. Lee, K. H. Lee, Evaluation of the performance of clustering algorithms in kernel-induced feature space, Pattern Recognition 38 (4) (2005) 607 – 611.

[11] I. S. Dhillon, Y. Guan, 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, ACM, New York, NY, USA, 2004, pp. 551–556.

[12] F. R. B. M. I. Jordan, F. Bach, Learning spectral clustering, Advances in Neural Information Processing Systems 16 (2004) 305–312.

[13] U. von Luxburg, A tutorial on spectral clustering, Statistics and Computing 17 (4) (2007) 395–416.

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

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

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

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

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

[19] J. A. Hartigan, P. M. Hartigan, The dip test of unimodality, The Annals of Statistics (1985) 7084.

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

[21] R. Winkler, F. Klawonn, R. Kruse, Problems of fuzzy c-means clustering and similar algorithms with high dimensional data sets., in: W. Gaul, A. Geyer-Schulz, L. Schmidt-Thieme, J. Kunze (Eds.), GfKl, Studies in Classification, Data Analysis, and Knowledge Organization, Springer, 2010, pp. 79–87.

[22] J. Shi, J. Malik, Normalized cuts and image segmentation, IEEE Transactions on Pattern Analysis and Machine Intelligence 22 (1997) 888–905.

[23] E. Bingham, H. Mannila, Random projection in dimensionality reduction: Applications to image and text data, in: Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '01, ACM, New York, NY, USA, 2001, pp. 245–250.

[24] K. W. Church, P. Li, T. J. Hastie, Conditional random sampling: A sketch-based sampling technique for sparse data, in: In NIPS, 2006, pp. 873–880.

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

[26] D. Agrawal, S. Das, 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, ACM, New York, NY, USA, 2011, pp. 530–533.

[27] R. L. Ferreira Cordeiro, C. Traina, Junior, A. J. Machado Traina, J. López, U. Kang, 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, ACM, New York, NY, USA, 2011, pp. 690–698.

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

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

[30] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.

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

[32] T. White, Hadoop: The Definitive Guide, 1st Edition, O'Reilly Media, Inc., 2009.

[33] I. S. Dhillon, Y. Guan, B. Kulis, Weighted graph cuts without eigenvectors a multilevel approach, IEEE Trans. Pattern Anal. Mach. Intell. 29 (11) (2007) 1944–1957.

[34] I. Dhillon, Y. Guan, B. Kulis, A Unified View of Kernel k-means, Spectral Clustering and Graph Cuts, Tech. Rep. TR-04-25, UTCS (Jul. 2004).

[35] H. Karloff, S. Suri, S. Vassilvitskii, A model of computation for mapreduce, in: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Al-

gorithms, SODA '10, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2010, pp. 938–948.

[36] S. C. Hinds, J. L. Fisher, D. P. D'Amato, A document skew detection method using run-length encoding and the hough transform, in: Pattern Recognition, 1990. Proceedings., 10th International Conference on, Vol. 1, IEEE, 1990, pp. 464–468.

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

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

[39] L. Wolf, T. Hassner, I. Maoz, Face recognition in unconstrained videos with matched background similarity, in: in Proc. IEEE Conf. Comput. Vision Pattern Recognition, 2011.

[40] T. Ojala, M. Pietikäinen, D. Harwood, A comparative study of texture measures with classification based on featured distributions, Pattern Recognition 29 (1) (1996) 51–59.

[41] M. Heikkila, M. Pietikainen, C. Schmid, Description of interest regions with center-symmetric local binary patterns, in: P. Kalra, S. Peleg (Eds.), 5th Indian Conference on Computer Vision, Graphics and Image Processing (ICVGIP '06), Vol. 4338 of Lecture Notes in Computer Science (LNCS), Springer-Verlag, Madurai, India, 2006, pp. 58–69.

[42] L. Wolf, T. Hassner, Y. Taigman, Descriptor based methods in the wild, in: Real-Life Images workshop at the European Conference on Computer Vision (ECCV), 2008.

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

[44] T. O. Kvålseth, Entropy and correlation: Some comments, IEEE Transactions on Systems, Man, and Cybernetics 17 (3) (1987) 517–519.