Fast Kernel Matrix Computation for Big Data Clustering

Nikolaos Tsapanos[1], Anastasios Tefas[1], Nikolaos Nikolaidis[1], Alexandros Iosifidis[1], and Ioannis Pitas[1]

Aristotle University of Thessaloniki Thessaloniki, Greece
{niktsap, tefas, nikolaid, aiosif, pitas}@aiia.csd.auth.gr

Abstract
Kernel k-Means is a basis for many state of the art global clustering approaches. When the number of samples grows too big, however, it is extremely time-consuming to compute the entire kernel matrix and it is impossible to store it in the memory of a single computer. The algorithm of Approximate Kernel k-Means has been proposed, which works using only a small part of the kernel matrix. The computation of the kernel matrix, even a part of it, remains a significant bottleneck of the process. Some types of kernel, however, can be computed using matrix multiplication. Modern CPU architectures and computational optimization methods allow for very fast matrix multiplication, thus those types of kernel matrices can be computed much faster than others.

Keywords: Kernel Matrix, Approximate, Kernel k-Means, Big Data, Clustering

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 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 [7], it is still an active research field.

One of the earliest clustering methods is the *k-Means* algorithm [13]. 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* [15]. 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* [1] 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 [18, 19, 8, 5]. A recent survey on kernel clustering methods can be found in [6], while [9] presents a comparative study which supports the superiority of kernel clustering methods, over more conventional clustering approaches.

With the expansion of the Internet and the proliferation of social media, multimedia data are being generated at unprecedented rates. This resulted in the rise of the so-called Big Data issue. Datasets are growing larger and it is becoming increasingly harder to handle them with traditional algorithms. With respect to clustering, a way to work around this problem is provided by the Approximate Kernel k-Means algorithm [2]. Instead of using the full kernel matrix, only a smaller, user defined set of matrix rows is calculated and used to measure distances and perform the clustering task. However, even the partial kernel matrix is extremely large and time consuming to compute. In this paper, we show that, for certain types of kernel, it is possible to calculate the kernel matrix using matrix multiplication, which in turn can be carried out very fast on modern CPU architectures and using specialized software, namely BLAS implementations. This demonstrates that even basic matrix operations have applications in the analysis of modern data.

This paper is organized as follows: Sections 2 and 3 briefly describe Kernel k-Means and Approximate Kernel k-Means, respectively. Section 4 discusses the factors that constitute the extremely fast computation of matrix multiplication possible. Section 5 Describes how some kernels can be expressed in matrix multiplication form and can, therefore, be computed extremely quickly. Experiments evaluating the speed and performance that the combination of fast kernel matrix computation and Approximate Kernel k-Means yields on Big Data can be found in Section 6, while Section 7 concludes the paper.

2 Kernel k-Means

The Kernel k-Means algorithm [3] 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:

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

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.

3 Approximate Kernel k-Means

In order to derive the Approximate Kernel k-Means algorithm [2], it is useful to express the goal of Kernel k-Means in matrix form. According to [4], the goal of Kernel k-Means is equivalent to the following optimization problem:

$$\min_{\mathbf{U} \in \mathcal{P}} \max_{\{\mathbf{c}_k \in \mathcal{H}_\kappa\}_{k=1}^C} \sum_{k=1}^{C} \sum_{i=1}^{n} U_{ki} |\mathbf{c}_k(\cdot) - \kappa(\mathbf{x}_i, \cdot)|^2_{\mathcal{H}_\kappa}, \tag{4}$$

where $\mathbf{U}$ is a matrix containing the cluster assignment vectors, $\mathbf{c}_k$ are the cluster centers, $\mathcal{P}$ is the domain of the assignment matrices and $\mathcal{H}_\kappa$ is the kernel Hilbert space. This optimization can be relaxed into

$$\min_{\mathbf{U}} (\mathrm{tr}(\mathbf{K}) - \mathrm{tr}(\tilde{\mathbf{U}}\mathbf{K}\tilde{\mathbf{U}}^T)), \tag{5}$$

where $\mathbf{K}$ is the kernel matrix, $\tilde{\mathbf{U}} = [\mathrm{diag}(\sqrt{n_1} \dots \sqrt{n_C})]^{-1}\mathbf{U}$ and $\mathrm{tr}(\cdot)$ denotes the trace of the respective matrix, i.e., the sum of its diagonal elements. The cluster centers for this relaxed optimization problem are given by

$$\mathbf{c}_k(\cdot) = \sum_{i=1}^{n} \hat{U}_{ki} \kappa(\mathbf{x}_i, \cdot), \tag{6}$$

where $\hat{\mathbf{U}} = [\text{diag}(n_1 \dots n_C)]^{-1}\mathbf{U}$.

In order to subsample the kernel matrix, Approximate Kernel k-Means replaces the full kernel matrix $\mathbf{K}$ in (5) with $\mathbf{K}_B\hat{\mathbf{K}}^{-1}\mathbf{K}_B^T$, where $\mathbf{K}_B$ is the $n \times m$ partial kernel matrix between the m sampled data points and the n total data points and $\hat{\mathbf{K}}$ is the $m \times m$ kernel matrix between the m sampled data points themselves. This is essentially the Nystrom low rank kernel matrix approximation. Some theoretical guarantees regarding the bound of the performance loss that results from the use of this approximation, instead of the full kernel matrix can be found in [2]. When $m \ll n$, this approximation makes clustering huge datasets viable.

4 Fast Matrix Multiplication

Basic Linear Algebra Subprograms (*BLAS*) [11] is a collection of high performance functions that provide basic tools for Linear Algebra related tasks, notably used in the *Linear Algebra Package* (*LAPACK*) library and MATLAB. There are several different implementations of BLAS, which usually specialize in different processors, CPU architectures, or operating systems. The function relevant to this paper is *general matrix multiplication* (*GEMM*).

GEMM implements the operation $\mathbf{C} \leftarrow \alpha\mathbf{AB} + \beta\mathbf{C}$. In order to obtain the product $\mathbf{AB}$, one can simply set $\alpha = 1$ and $\beta = 0$, but this general form of the operation can also prove useful.

The speed of BLAS is a result of a combination of factors. It recursively splits the multiplication into smaller subtasks. It also takes advantage of the memory locality of data in consecutive memory positions, in order to better utilize the CPU's much faster cache memory. It employs loop unrolling, to reduce the computational time taken up by the program's control flow. The *Single Input Multiple Data* (*SIMD*) multiprogramming capabilities of the CPU and, potentially, its *Advanced Vector Extensions* (*AVX*) are exploited, in order to perform the same computational operation on multiple matrix elements simultaneously. Finally, multiple threads are used, so that every core of the CPU can contribute to the calculations.

There already exist matrix multiplication algorithms that improve the asymptotic computational complexity $O(n^3)$ of naive matrix multiplication, such as the Strassen algorithm [16] with $O(n^{log_2 7}) \approx O(n^{2.8})$, or other recent advancements [12] with about $O(n^{2.373})$. In practice, however, they are not used in BLAS implementations, as they face numerical issues, take up a lot more memory, or require unrealistically large matrices, in order to provide an improvement.

5 Appropriate Kernels

In this section, we will see how it is possible to use the fast matrix multiplication tools provided by BLAS to quickly compute the subsampled kernel matrix $\mathbf{K}_B$ of very large datasets required by Approximate Kernel k-Means. The elements of $\hat{\mathbf{K}}^{-1}$ are already contained in $\mathbf{K}_B$ and need no additional calculations.

Let us consider the *Radial Basis Function* (*RBF*) kernel. This kernel is defined as

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma||\mathbf{x}_i - \mathbf{x}_j||^2}, \tag{7}$$

where $||\cdot||$ denotes the Euclidean distance. Suppose that n is the total number of data points, m is the number of subsampled data points for Approximate Kernel k-Means and d is the dimensionality of the data. All the data points are stored in $n \times d$ matrix $\mathbf{X}$, while the subsampled data points are stored in $m \times d$ matrix $\hat{\mathbf{X}}$. Normally, one would subtract vector $\mathbf{x}_j$

$$
\begin{bmatrix}
\ddots & & \cdots & & \iddots \\
 & \sum_{l=1}^{d} x_{(i-1)l}^2 & \sum_{l=1}^{d} x_{(i-1)l}^2 & \sum_{l=1}^{d} x_{(i-1)l}^2 & \\
\vdots & \sum_{l=1}^{d} x_{il}^2 & \sum_{l=1}^{d} x_{il}^2 & \sum_{l=1}^{d} x_{il}^2 & \vdots \\
 & \sum_{l=1}^{d} x_{(i+1)l}^2 & \sum_{l=1}^{d} x_{(i+1)l}^2 & \sum_{l=1}^{d} x_{(i+1)l}^2 & \\
\iddots & & \cdots & & \ddots
\end{bmatrix}
$$

$$
-2
\begin{bmatrix}
\ddots & & \cdots & & \iddots \\
 & \sum_{l=1}^{d} x_{(i-1)l}\hat{x}_{(j-1)l} & \sum_{l=1}^{d} x_{(i-1)l}\hat{x}_{jl} & \sum_{l=1}^{d} x_{(i-1)l}\hat{x}_{(j+1)l} & \\
\vdots & \sum_{l=1}^{d} x_{il}\hat{x}_{(j-1)l} & \sum_{l=1}^{d} x_{il}\hat{x}_{jl} & \sum_{l=1}^{d} x_{il}\hat{x}_{(j+1)l} & \vdots \\
 & \sum_{l=1}^{d} x_{(i+1)l}\hat{x}_{(j-1)l} & \sum_{l=1}^{d} x_{(i+1)l}\hat{x}_{jl} & \sum_{l=1}^{d} x_{(i+1)l}\hat{x}_{(j+1)l} & \\
\iddots & & \cdots & & \ddots
\end{bmatrix}
$$

$$
+
\begin{bmatrix}
\ddots & & \cdots & & \iddots \\
 & \sum_{l=1}^{d} \hat{x}_{(j-1)l}^2 & \sum_{l=1}^{d} \hat{x}_{jl}^2 & \sum_{l=1}^{d} \hat{x}_{(j+1)l}^2 & \\
\vdots & \sum_{l=1}^{d} \hat{x}_{(j-1)l}^2 & \sum_{l=1}^{d} \hat{x}_{jl}^2 & \sum_{l=1}^{d} \hat{x}_{(j+1)l}^2 & \vdots \\
 & \sum_{l=1}^{d} \hat{x}_{(j-1)l}^2 & \sum_{l=1}^{d} \hat{x}_{jl}^2 & \sum_{l=1}^{d} \hat{x}_{(j+1)l}^2 & \\
\iddots & & \cdots & & \ddots
\end{bmatrix}
$$

$$
=
\begin{bmatrix}
\ddots & & \cdots & & \iddots \\
 & ||\mathbf{x}_{(i-1)} - \hat{\mathbf{x}}_{(j-1)}||^2 & ||\mathbf{x}_{(i-1)} - \hat{\mathbf{x}}_{j}||^2 & ||\mathbf{x}_{(i-1)} - \hat{\mathbf{x}}_{(j-1)}||^2 & \\
\vdots & ||\mathbf{x}_{i} - \hat{\mathbf{x}}_{(j-1)}||^2 & ||\mathbf{x}_{i} - \hat{\mathbf{x}}_{j}||^2 & ||\mathbf{x}_{i} - \hat{\mathbf{x}}_{(j+1)}||^2 & \vdots \\
 & ||\mathbf{x}_{(i+1)} - \hat{\mathbf{x}}_{(j-1)}||^2 & ||\mathbf{x}_{(i+1)} - \hat{\mathbf{x}}_{j}||^2 & ||\mathbf{x}_{(i+1)} - \hat{\mathbf{x}}_{(j+1)}||^2 & \\
\iddots & & \cdots & & \ddots
\end{bmatrix}
$$

Figure 1: The computation of the Euclidean distance matrix.

from $\mathbf{x}_i$, then square the results and, finally sum. However, taking advantage of the identity $(\alpha - \beta)^2 = \alpha^2 - 2\alpha\beta + \beta^2$, we can reformulate the Euclidean distance using matrices as:

$$\mathbf{Y} = ((\mathbf{X} \circ \mathbf{X})\mathbf{1}_d) \otimes \mathbf{1}_m^T - 2\mathbf{X}\hat{\mathbf{X}}^T + ((\hat{\mathbf{X}}^T \circ \hat{\mathbf{X}}^T\mathbf{1}_d^T) \otimes \mathbf{1}_n), \tag{8}$$

where each element y_{ij} of matrix $\mathbf{Y}$ will store the Euclidean distance between data points $\mathbf{x}_i$ and $\mathbf{x}_j$, $\circ$ denotes the Hadamard matrix product, $\otimes$ denotes the Kronecker product and $\mathbf{1}$ are column vectors with every element equal to 1 and their size is determined by their index. This computation is illustrated in Figure 1.

Matrix $((\mathbf{X}\circ\mathbf{X})\mathbf{1}_d)\otimes\mathbf{1}_m^T$ is essentially a column vector containing the sum of squares of every data point $\mathbf{x}$ repeated m times to form a $n \times m$ matrix. Respectively, $((\hat{\mathbf{X}}^T \circ \hat{\mathbf{X}}^T\mathbf{1}_d^T) \otimes \mathbf{1}_n)$ is essentially a row vector containing the sum of squares of every subsampled data point $\hat{\mathbf{x}}$ repeated n times to form another $n \times m$ matrix. Both of these matrices are more trivial to compute than their matrix forms might indicate. The bottleneck of this computation is $-2\mathbf{X}\hat{\mathbf{X}}^T$, which is a traditional matrix multiplication that can be computed very fast, as described in Section 4. Finally, in order to obtain the RBF kernel matrix, every element of $\mathbf{Y}$ is multiplied by $-\gamma$ and used as an exponent for e to obtain the final kernel matrix.

Every kernel that involves the Euclidean distance can therefore be computed in the above manner. Kernels that involve the dot product of two data points are more straight-

forward to compute, as matrix $\mathbf{X}\hat{\mathbf{X}}^T$ provides the dot product of every data point combination directly. However, kernels that do not include either still have to be computed in the traditional way. Kernels that involve the Euclidean distance besides the RBF kernel include the Rational Quadratic Kernel $\left(\kappa(\mathbf{x}_i, \mathbf{x}_j) = 1 - \frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{||\mathbf{x}_i - \mathbf{x}_j||^2 + c}\right)$, the Multiquadratic Kernel $\left(\kappa(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{||\mathbf{x}_i - \mathbf{x}_j||^2 + c^2}\right)$, its Inverse $\left(\kappa(\mathbf{x}_i, \mathbf{x}_j) = \frac{1}{\sqrt{||\mathbf{x}_i - \mathbf{x}_j||^2 + c^2}}\right)$ and the Cauchy kernel $\left(\kappa(\mathbf{x}_i, \mathbf{x}_j) = \frac{1}{1 + \frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{\sigma^2}}\right)$. Kernels that involve the dot product include the Linear kernel $\left(\kappa(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \mathbf{x}_j + c\right)$, the Polynomial kernel $\left(\kappa(\mathbf{x}_i, \mathbf{x}_j) = (a\mathbf{x}_i^T \mathbf{x}_j + c)^b\right)$ and the Hyperbolic Tangent kernel $\left(\kappa(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\mathbf{x}_i^T \mathbf{x}_j + c)\right)$. Kernels that involve any combination of data points except the Euclidean distance and the dot product include the χ^2 kernel $\left(\kappa(\mathbf{x}_i, \mathbf{x}_j) = 1 - \sum_{l=1}^{d} \frac{(x_{il} - x_{jl})^2}{\frac{1}{2}(x_{il} - x_{jl})}\right)$ and the Histogram Intersection kernel $\left(\kappa(\mathbf{x}_i, \mathbf{x}_j) = \sum_{l=1}^{d} \min(x_{il}, x_{jl})\right)$.

6 Experiments

In order to evaluate the speed with which datasets of Big Data magnitude can be clustered and the performance of that clustering, we used the Youtube Faces dataset [17]. It contains feature vectors for 621126 faces of politicians, actors, athletes and other celebrities, extracted from videos on Youtube. The feature vectors are *Local Binary Patterns* (*LBP*s) [14]. Since the features are essentially histograms, a kernel function more suited to such features would be preferable, like Histogram Intersection of χ^2, however, we are limited to kernels we can compute using matrix multiplication. We used 3 of the most commonly used kernels, RBF, linear and polynomial. We calculated a partial 1000×621126 kernel matrix. For the RBF kernel parameter γ, it was experimentally determined that a value of 0.05 provided the best results. For the linear kernel, c was set to 1. The polynomial kernel was of the 2nd degree. We run the Approximate Kernel k-Means (AKKM) algorithm for each kernel 10 times, measuring the time for the kernel calculation, the time for AKKM and the *Normalized Mutual Information* (NMI) [10] performance of the resulting clustering. An Intel Xeon E5-2680 at 2.8GHz CPU was used for all the experiments. The results of these experiments are presented in Table 1, in the format of *mean* (*standard deviation*).

Table 1: The experimental results for all the types of kernel used.

Kernel	RBF	Linear	Polynomial
Matrix calculation time (seconds)	61.5336 (14.1589)	8.5485 (1.0354)	8.5827 (1.0095)
AKKM time (seconds)	2887 (498.8669)	2775 (452.6060)	2.664 (49.1836)
NMI performance	0.8232 (0.0022)	0.8363 (0.0014)	0.8311 (0.0019)

Overviewing the results, we notice that the kernel matrix computation is extremely fast, mere seconds, in fact. In fact, it is faster than performing other operations on a per element basis and additions, as evidenced by the fact that the best time was achieved by the linear kernel, with a significant advantage over the RBF kernel, which also needs to add the squared values of the data vector elements. We notice that the time to perform AKKM was order of magnitude longer than the kernel matrix computation, though this can be attributed to the fact that the AKKM implementation was not as exhaustively optimized as the BLAS matrix

multiplication. In terms of performance, all kernels performed similarly, with the linear kernel being the best, slightly above the polynomial kernel.

7 Conclusions

In this paper, we demonstrated that matrix multiplication can significantly improve the computation speed of the kernel matrix for appropriate kernels. When using a kernel function, in which the correlation of the data is done through the Euclidean distance or the dot product, it is possible to express the computation in terms of matrix multiplication. This matrix operation can be performed extremely fast with modern hardware and the BLAS library, a highly specialized and optimized piece of software, thus allowing for the extremely fast computation of a kernel matrix. We used this approach in conjunction with Approximate Kernel k-Means, in order to perform clustering on the Youtube Faces dataset that, which includes 621126 data samples.

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

[2] Radha Chitta, Rong Jin, Timothy C. Havens, and Anil K. Jain. Approximate kernel k-means: solution to large scale kernel clustering. In Chid Apte', Joydeep Ghosh, and Padhraic Smyth, editors, *KDD*, pages 895–903. ACM, 2011.

[3] Inderjit S. Dhillon, Yuqiang Guan, and Brian Kulis. Weighted graph cuts without eigenvectors a multilevel approach. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(11):1944–1957, November 2007.

[4] Chris HQ Ding, Xiaofeng He, and Horst D Simon. On the equivalence of nonnegative matrix factorization and spectral clustering. In *SDM*, volume 5, pages 606–610. SIAM, 2005.

[5] Marcelo R.P. Ferreira and Francisco de A.T. de Carvalho. Kernel-based hard clustering methods in the feature space with automatic variable weighting. *Pattern Recognition*, (0):–, 2014.

[6] Maurizio Filippone, Francesco Camastra, Francesco Masulli, and Stefano Rovetta. A survey of kernel and spectral methods for clustering. *Pattern Recognition*, 41(1):176 – 190, 2008.

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

[8] Hong Jia, Yiu ming Cheung, and Jiming Liu. Cooperative and penalized competitive learning with application to kernel-based clustering. *Pattern Recognition*, (0):–, 2014.

[9] Dae-Won Kim, Ki Young Lee, Doheon Lee, and Kwang H. Lee. Evaluation of the performance of clustering algorithms in kernel-induced feature space. *Pattern Recognition*, 38(4):607 – 611, 2005.

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

[11] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[12] François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC '14, pages 296–303, New York, NY, USA, 2014. ACM.

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

[14] Timo Ojala, Matti Pietikäinen, and David Harwood. A comparative study of texture measures with classification based on featured distributions. *Pattern Recognition*, 29(1):51–59, January 1996.

[15] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Comput.*, 10(5):1299–1319, July 1998.

[16] S.S. Skiena. *The Algorithm Design Manual*. Springer, 2009.

[17] Lior Wolf, Tal Hassner, and Itay Maoz. Face recognition in unconstrained videos with matched background similarity. In *in Proc. IEEE Conf. Comput. Vision Pattern Recognition*, 2011.

[18] Shi Yu, Leon-Charles Tranchevent, Xinhai Liu, Wolfgang Glanzel, Johan A.K. Suykens, Bart De Moor, and Yves Moreau. Optimized data fusion for kernel k-means clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(5):1031–1039, 2012.

[19] Feng Zhou, Fernando De la Torre Frade, and Jessica K Hodgins. Hierarchical aligned cluster analysis for temporal clustering of human motion. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 35(3):582–596, March 2013.