Ordozgoiti, B.; Belanche, L. Off-the-grid: Fast and effective hyperparameter search for kernel clustering. A: European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases. "Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2020: Ghent, Belgium, September 14–18, 2020: proceedings, part II". Berlín: Springer, 2020, p. 399-415. ISBN 978-3-030-67661-2. The final authenticated version is available online at https://doi.org/10.1007/978-3-030-67661-2_24

Off-the-grid: Fast and Effective Hyperparameter Search for Kernel Clustering*

Bruno Ordozgoiti[1] ✉ and Lluís A. Belanche Muñoz[2]

[1] Aalto University, Finland <firstname>.<lastname>@aalto.fi
[2] Universitat Politècnica de Catalunya, Spain
belanche@cs.upc.edu

Abstract. Kernel functions are a powerful tool to enhance the k-means clustering algorithm via the kernel trick. It is known that the parameters of the chosen kernel function can have a dramatic impact on the result. In supervised settings, these can be tuned via cross-validation, but for clustering this is not straightforward and heuristics are usually employed. In this paper we study the impact of kernel parameters on kernel k-means. In particular, we derive a lower bound, tight up to constant factors, below which the parameter of the RBF kernel will render kernel k-means meaningless. We argue that grid search can be ineffective for hyperparameter search in this context and propose an alternative algorithm for this purpose. In addition, we offer an efficient implementation based on fast approximate exponentiation with provable quality guarantees. Our experimental results demonstrate the ability of our method to efficiently reveal a rich and useful set of hyperparameter values.

Keywords: clustering · kernels · kernel k-means · hyperparameter tuning · grid search.

1 Introduction

Clustering, the task of partitioning a given data set into groups of similar items, is one of the central topics in data analysis. Among the plethora of existing techniques for this purpose, k-means clustering, along with Lloyd's algorithm [14], is one of the most popular and well-understood methods. Despite its popularity, k-means has significant limitations, as it implicitly makes strong assumptions about the shapes of the clusters. Numerous alternative methods have been proposed to tackle challenges beyond the capabilities of k-means [8,15,16,13].

One of these involves the use of positive definite kernels [11], which enable the computation of inner products between elements of a vector space after mapping them to a different, high-dimensional space. In particular, kernels enhance the capabilities of k-means by enabling the detection of clusters of arbitrary shapes.

One drawback of kernel functions is that they usually involve hand-set parameters, which must be fine-tuned to bring forth their full potential. A common

* This work was supported by the Academy of Finland project 317085.

method to choose a value for these parameters is grid search. One considers a set of values and then evaluates the performance of the algorithm for each of them. A drawback is that one might either choose too small a set and risk missing optimal values, or an overly big one, incurring excessive —and possibly redundant— computational costs. Another way to set these values is by heuristics and rules of thumb [19,12], but these rarely apply to a wide variety of data.

Our contribution in this paper is two-fold. First, we illustrate the impact of kernel parameters in clustering by deriving a lower bound for the bandwidth parameter of RBF kernels (section 4), below which `Kernel` k`-means` will be rendered useless. We show this bound is tight. Next, we propose a method for hyperparameter search. Our method specifically searches for values that will produce different clusterings, and thus, unlike grid search, does not risk carrying out redundant computations, so no processing time is wasted. We combine methods for fast exponentiation with the properties of dyadic rationals to design an algorithm that after $\mathcal{O}\left(\log\left(\frac{|\log(b)|}{\epsilon}\right)\right)$ iterations —where b is the minimum entry in the kernel matrix— provides a $(1 \pm \epsilon)$-approximation of the next meaningful hyperparmeter value to inspect (sections 5 and 6). We validate our claims with a rich variety of experiments (section 7).

2 Related work

Kernels have been a central subfield of machine learning since their first use in conjunction with support vector machines [5]. Even though most efforts have focused on their application to supervised learning methods, they have also played a significant part in the development of clustering techniques [4,16,7]. In the seminal work by Ben-Hur et al. [4], the authors suggest to inspect the results using varying values of σ, starting from the maximizer of the pairwise squared distances $\|x-y\|^2$ over all pairs of data points. A good choice might lie within a region that yields stable clusterings. It should be noted that stability has been shown to have significant drawbacks for choosing the number of clusters [3], so it would be interesting to determine whether this applies to the kernel bandwidth as well. In the work that introduced spectral clustering [16], Ng et al. rely on a result of their own that guarantees that their algorithm will produce tight clusters if they exist in the data. They then propose to test various values of σ in search for a clustering with this property. In [2] a generalized form of the bandwidth parameter is learned based on data with known clustering. In [20] a different value of σ is computed for each point. The approach proposed by the authors relies on the distance to the k-th neighbor. In [10], the authors investigate the problem of kernel matrix diagonal dominance in clustering, which is essentially a generalization of the problem we analyze in the beginning of section 4. The heuristics they explore to alleviate the problem either require the selection of a new hyperparameter, or heavily modify the structure of the problem. The latter can even lead to the loss of positive-definiteness of the kernel matrix, which results in algorithmic oscillations and failure to converge. The mean distance to the k-th nearest neighbour is also suggested as a heuristic by Von Luxburg [19].

3 Preliminaries

We consider a finite set of data points $X \subset \mathbb{R}^d$. We define a k-partition of X as a collection of k non-empty subsets of X, $\pi_1, \ldots, \pi_k$, satisfying $\bigcup_{i=1}^k \pi_i = X$ and $\pi_i \cap \pi_j = \emptyset$ for $i, j = 1, \ldots, k$, $i \neq j$. We will refer to each π_i as a *cluster* and use $n_i = |\pi_i|$ to denote its cardinality.

The *k-means* objective is to find a k-partition of X so as to minimize

$$\sum_{i=1}^{k} \sum_{x \in \pi_i} \|x - \bar{\pi}_i\|^2, \tag{1}$$

where $\bar{\pi}_i = n_i^{-1} \sum_{x \in \pi_i} x$ is the *centroid* of cluster π_i and $\|x\|$ denotes the L_2 norm in $\mathbb{R}^d$. Optimizing this objective is known to be **NP**-hard for $k = 2$ [1]. A popular heuristic is Lloyd's algorithm [14], which repeatedly recomputes the centroid of each cluster and reassigns points to the closest centroid.

Kernels: Given a non-empty set $\mathcal{X}$, a symmetric function $\kappa : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ such that for all $n \in \mathbb{N}$ and every set $\{x_i\}_{i=1}^n \subset \mathcal{X}$, the matrix $K = (\kappa(x_i, x_j))_{ij}$ is positive definite, is called a (strictly) **positive definite (PD) kernel**. The matrix K is known as the *Gram* matrix or *Kernel* matrix. Since PD kernels give rise to a PD Gram matrix, they correspond to the computation of an inner product in some implicit inner-product space. The representation of an object $x \in \mathcal{X}$ in said space is often called *feature space representation*, denoted as $\phi(x)$.

A number of kernels are routinely used in practice. Probably the most popular one for the case $\mathcal{X} = \mathbb{R}^d$ is the Gaussian RBF kernel

$$\kappa(x, y) = \exp\left(\frac{-\|x - y\|^2}{\sigma}\right),$$

$\sigma > 0$, (from here on, RBF kernel). The parameter σ is commonly referred to as *bandwidth*. We will use κ_σ to denote the RBF kernel function with bandwidth parameter σ and K_σ to denote the corresponding kernel matrix.

Consider a data set X and the k-partition $\pi_1, \ldots, \pi_k$. Let m_i denote the centroid of cluster π_i in feature space, that is,

$$m_i = \frac{1}{n_i} \sum_{x \in \pi_i} \phi(x).$$

The application of kernels to the k-means objective (1) relies on the following observation: even though we cannot in general express m_i explicitly, it is possible to compute the necessary squared distances. For any $x \in X$ and $i = 1, \ldots, k$,

$$\|\phi(x) - m_i\|^2 = \kappa(x, x) - \frac{2 \sum_{y \in \pi_i} \kappa(x, y)}{n_i} + \frac{\sum_{y,z \in \pi_i} \kappa(y, z)}{n_i^2}. \tag{2}$$

The application of Lloyd's algorithm using this expression for the squared distance is known as `Kernel k-means`. See [7] for an insightful analysis. `Kernel k-means` always converges when the kernel matrix is positive semidefinite. We will refer to the k-partition at convergence as the *output* of `Kernel k-means`.

4 The use of the RBF kernel in Kernel *k*-means

RBF kernels are powerful but sensitive to the bandwidth parameter. In particular, for sufficiently small σ, a support vector machine classifier can fit any training set with no errors —or equivalently, it has infinite VC dimension [18]—, but this will generally result in poor generalization ability. In `Kernel` k-`means`, the result of an overly small bandwidth will be that the algorithm will converge in the first iteration, regardless of the current k-partition. The reason is that as σ decreases, the value of $\kappa(x, y)$ for any two distinct points $x, y \in X$ decreases as well, to the point of becoming negligible. Therefore, the only significant term in equation (2) for any x will be $\kappa(x, x)$, which means that the closest cluster to x will be the one it is currently in. A question arises naturally: how small does σ have to be for the algorithm to get stuck at the initial clustering? The following theorem provides a lower bound, which is tight up to constant factors.

Theorem 1. *Consider a data set $X \subset \mathbb{R}^d$, $|X| = n$. Let $x, y = \arg\min_{x,y \in X} \|x - y\|^2$. If $\sigma \leq (\log(3n))^{-1} \|x - y\|^2$, then* `Kernel` k-`means` *will make no cluster reassignments.*

The proof is given in the supplementary material.

A tight example. The next example shows that this result is tight up to constant factors. Consider an instance with two clusters, π_1 and π_2, containing n_1 and n_2 points respectively. For some point $y \in \pi_2$ it is $\|x - y\|_2^2 = \min_{a,b} \|a - b\|_2^2 = \epsilon$ for all $x \in \pi_1$, whereas for all $z \in \pi_2, z \neq y$, it is $\|y - z\|_2^2 = 2\epsilon$. Moreover, for all $w, z \in \pi_1$ it is $\|w - z\|_2^2 = \epsilon$ and for all $w, z \in \pi_2, w, z \neq y$, it is $\|w - z\|_2^2 = \epsilon$. Define $n = n_1 + n_2$ and consider $\sigma = \epsilon / \log(n/3)$. We know y will switch over to π_1 if $\|\phi(y) - m_1\|_2^2 < \|\phi(y) - m_2\|_2^2$, or equivalently,

$$\begin{aligned}\frac{2}{n_2} < \frac{2\sum_{x\in\pi_1}\kappa(y,x)}{n_1} - \frac{\sum_{w,z\in\pi_1}\kappa(w,z)}{n_1^2} - \frac{2\sum_{z\in\pi_2, z\neq y}\kappa(y,z)}{n_2} + \frac{\sum_{w,z\in\pi_2}\kappa(w,z)}{n_2^2} \\ = 6/n - 1/n_1 - \frac{3(n_1-1)}{nn_1} - \frac{2(n_2-1)}{n_2}\left(\frac{3}{n}\right)^2 \\ + 1/n_2 + \frac{3(n_2-1)(n_2-2)}{nn_2^2} + \frac{(n_2-1)}{n_2^2}\left(\frac{3}{n}\right)^2. \quad (3)\end{aligned}$$

The above inequality is verified when $n_1 = n_2$ and n is sufficiently large. That is, there exists a family of instances where the kernel k-means algorithm with the RBF kernel will make cluster reassignments with $\sigma = \Omega\left(\frac{\|x-y\|_2^2}{\log(n)}\right)$, where $\|x - y\|_2^2$ is minimal over all x, y in the data set.

5 Optimizing bandwidth

As demonstrated above, the choice of bandwidth parameter is crucial when using RBF kernels for clustering. For some choices of σ, the output of `Kernel`

k-means will be unchanged from the initial k-partition. In fact, for any value of σ the algorithm will converge at some point —provided that the kernel matrix is positive semidefinite— and stop making changes. However, if the chosen value is inadequate the output might still be of poor quality, so it is often desirable to further refine σ in order to obtain a better result. We already know, by virtue of Theorem 1, a value of σ such that Kernel k-means will stop making changes. The following question arises naturally. *How big does σ have to be in order to guarantee that* Kernel k-means *will change the initial k-partition?*, and more generally, *once* Kernel k-means *has converged, how much do we have to increase σ to ensure it will make new changes?* We define this as the *critical bandwidth value.*

Definition 1. *(Critical bandwidth value) Let X a data set. Suppose* Kernel k-means *outputs a k-partition $P = (\pi_1, \ldots, \pi_k)$ of X when run using an RBF kernel with bandwidth parameter σ. We define $S \subset \mathbb{R}$ to be the set satisfying the following: if* Kernel k-means *is initialized with k-partition P and run with $K_{\sigma'}$, with $\sigma' \in S$, it will output a k-partition $P' \neq P$, that is, it will make changes. We define the critical bandwidth value with respect to (K_σ, P) to be the infimum of S, or ∞ if $S = \emptyset$.*

In other words, the critical bandwidth value reveals the "minimal" value the RBF kernel bandwidth needs to take so that Kernel k-means "snaps out" of convergence and yields a new k-partition. Any value strictly larger than the critical value will suffice. This concept is the cornerstone of our contribution.

5.1 Finding the critical value

Possibly the most straightforward method to find a value of σ —or virtually any hyperparameter— is grid search. This consists in running the clustering algorithm for a predetermined set of values of the hyperparameter and choosing the one which provides the best performance, as measured by e.g. objective function values or clustering quality indices [17]. This approach, however, has significant disadvantages. If the set of values to test is too small, one can fail to detect one that yields good performance; if it is too large, running times can be prohibitive and some computations redundant.

Here we propose an alternative approach. Roughly, we proceed as follows. First, we choose a sufficiently small value of σ —e.g. guided by Theorem 1— and run Kernel k-means. We then search for the critical bandwidth value with respect to the current kernel matrix and k-partition and rerun Kernel k-means until convergence. We can keep doing this until no further changes are observed, to finally obtain a set of possible hyperparameter choices. The question that arises now is how to find said value efficiently. Next, we illustrate the fact that this value can be located using optimization methods.

A first approach Let κ_σ denote the RBF kernel function parametrized by σ. In a Kernel k-means iteration, a point x is assigned to the cluster π_i which

maximizes the *proximity* function δ:

$$\delta_\sigma(x, m_i) = \frac{2\sum_{y\in\pi_i}\kappa_\sigma(x,y)}{n_i} - \frac{\sum_{y,z\in\pi_i}\kappa_\sigma(y,z)}{n_i^2}. \tag{4}$$

Now, observe that if we change the value of the bandwidth parameter to σ', the new value of the kernel for any pair of points x, y can be computed as follows:

$$\kappa_{\sigma'}(x,y) = \kappa_\sigma(x,y)^{\sigma/\sigma'},$$

and we can thus compute the new proximity functions $\delta_{\sigma'}(x, m_i)$ accordingly. For simplicity, we consider the case of two clusters π_1, π_2. Assume $x \in \pi_1$. x will switch over to π_2 when

$$\delta_{\sigma'}(x, m_1) < \delta_{\sigma'}(x, m_2) \Leftrightarrow \delta_{\sigma'}(x, m_1) - \delta_{\sigma'}(x, m_2) < 0.$$

That is, we can find the value of σ' that will result in a different clustering by finding a root of $\delta_{\sigma'}(x, m_1) - \delta_{\sigma'}(x, m_2)$.

A useful observation is that $\kappa_\sigma(x,y)^\sigma$ is constant with respect to σ'. Therefore, we can easily derive $\delta_{\sigma'}(x, m_1) - \delta_{\sigma'}(x, m_2)$ with respect to σ'. In particular, define $g(x, \sigma') = \delta_{\sigma'}(x, m_1) - \delta_{\sigma'}(x, m_2)$. Then

$$\begin{aligned}\frac{dg}{d\sigma'} = \frac{2\sum_{y\in\pi_2}\log\left(\kappa_\sigma(x,y)^\sigma\right)\kappa_\sigma(x,y)^{\sigma/\sigma'}}{\sigma'^2 n_2} - \frac{\sum_{y,z\in\pi_2}\log\left(\kappa_\sigma(y,z)^\sigma\right)\kappa_\sigma(y,z)^{\sigma/\sigma'}}{\sigma'^2 n_2^2} \\ - \frac{2\sum_{y\in\pi_1}\log\left(\kappa_\sigma(x,y)^\sigma\right)\kappa_\sigma(x,y)^{\sigma/\sigma'}}{\sigma'^2 n_1} + \frac{\sum_{y,z\in\pi_1}\log\left(\kappa_\sigma(y,z)^\sigma\right)\kappa_\sigma(y,z)^{\sigma/\sigma'}}{\sigma'^2 n_1^2}.\end{aligned} \tag{5}$$

This implies that we can use iterative root-finding algorithms, such as Newton's method, to efficiently find a root of the above function, that is, the minimum value of σ' that will result in a clustering change, or the critical bandwith value.

This approach, however, can be slow and numerically unstable. In the next section we propose an alternative optimization method able to efficiently locate the critical bandwidth value to arbitrary precision while overcoming these drawbacks.

6 Fast and effective hyperparameter search

The approach outlined above has several drawbacks, namely (1) using an iterative root-finding algorithm entails repeatedly recomputing the kernel matrix, either directly or by element-wise exponentiation, which can be slow in practice when dealing with large matrices and (2) the operations required for the derivative of g and the fractional computations can induce numerical instability.

Here we propose an alternative approach to sidestep these issues. The proposed method rests on the following fact: *computing products and square roots of real numbers can be much faster than computing powers with arbitrary exponents* [9]. Our method has the additional advantage of being numerically stable.

6.1 Dyadic rationals and fast approximate exponentiation

To develop an efficient method for hyperparameter search, we first propose an algorithm for fast approximate exponentiation that only uses products and square roots. This algorithm (Algorithm 1) forms the basis of our approach.

Exponentiation algorithm overview. As hinted above, we wish to avoid computing element-wise powers of the kernel matrix, and instead use element-wise products and square roots. To accomplish this, suppose we want to compute the power b^p, for some arbitrary positive reals b and p. We first decompose p as $p = z + f$, where z is the integral part and f the decimal part of p. We then compute b^z and approximate b^f as $b^{f'}$ using two separate fast methods for integral and rational exponents and finally return $b^z b^{f'} \approx b^p$.

To design our algorithm, we rely on two simple results. First, we make use of the following recursive representation of a positive integer based on its binary representation, which has long been employed in the design of fast algorithms for power computation with integral exponents [9].

Lemma 1. *Consider a number $n \in \mathbb{N}$, and let $b_0 \dots b_t$, where $t = \lfloor \log_2 n \rfloor$, be its binary representation, i.e. $n = \sum_{i=0}^{t} 2^{t-i} b_i$. Then $n = n_t$, where*

$$n_i = \begin{cases} 1 & \text{if } i = 0 \\ 2n_{i-1} + b_i & \text{if } 0 < i \leq t \end{cases}$$

Lemma 1 reveals how to compute a power of the form b^i, where b is a positive real number and i is a natural number, using a small number of products. In particular, this operation is carried out in lines 5 and 6 of Algorithm 1.

The next result we rely on is a consequence of the properties of dyadic rationals. Dyadic rationals are rational numbers of the form $n/2^i$, where n is an integer and i is a natural number. It is well known that dyadic rationals are dense in $\mathbb{R}$, that is, any real number can be approximated arbitrarily well by a dyadic rational. The next result reveals how to obtain such an approximation for numbers in the interval $(0, 1)$, which will be useful in our context.

Lemma 2. *Let $a \in (0, 1)$. There exists a sequence (m_i), with $m_i \in \{-1, 1\}, i = 1, \dots$ such that $\lim_{t\to\infty} \sum_{i=1}^{t} m_i 2^{-i} = a$.*

Proof. Let $m_1 = 1$. Choose the j-th term of (m_i) (for $j > 1$) to be 1 if $\sum_{i=1}^{j-1} m_i 2^{-i} < a$, -1 if $\sum_{i=1}^{j-1} m_i 2^{-i} > a$, 0 otherwise. Clearly, $\left| a - \sum_{i=1}^{k} m_i 2^{-i} \right| \leq 2^{-k}$.

The set of dyadic rationals is clearly closed under addition, and thus the above series provides an approximation by means of a dyadic rational.

Now, suppose we want to approximately compute the power b^p, by an approximation of p to within an error of 2^{-j}. The above result implies that it suffices to compute j operations, at each step either multiplying or dividing by

Algorithm 1 Fast approximate exponentiation

Input: base b, exponent p, depth i

1: $z \leftarrow bin(\lfloor p \rfloor)[1:]$
2: $f \leftarrow p - \lfloor p \rfloor$
3: $b_1 \leftarrow b$; $b_2 \leftarrow b$
4: $j \leftarrow 1$
5: **for** d in z **do**
6: $\quad b_1 \leftarrow b_1^2 b^d$
7: $n \leftarrow 1$; $d \leftarrow 2$
8: **for** $j = 1, \ldots, i$ **do**
9: $\quad b \leftarrow \sqrt{b}$; $n \leftarrow 2n$; $d \leftarrow 2d$
10: $\quad$ **if** $n/d > f$ **then**
11: $\quad\quad n \leftarrow n - 1$; $b_2 \leftarrow b_2/b$
12: $\quad$ **if** $n/d < f$ **then**
13: $\quad\quad n \leftarrow n + 1$; $b_2 \leftarrow b_2 b$
14: $\quad$ **if** $n/d = f$ **then**
15: $\quad\quad$ ȷ $\leftarrow i + 1$ // Exact exponent matched, so exit loop
16: Output $b_1 \times b_2$

successive square roots of b. This is done in lines 8 through 15 of Algorithm 1. The following result characterizes the quality of the approximation achieved by Algorithm 1, and the required number of operations.

Theorem 2. *Algorithm 1 yields a* $(1 \pm \epsilon)-$*approximation of* b^p *after performing* $\mathcal{O}\left(\log\left(\frac{|\log(b)|}{\epsilon}\right)\right)$ *operations.*

Proof. First, note that the algorithm computes at most $2i$ multiplications in the first phase, and i square roots or multiplications in the second.

Assume $b > 1$. We treat the alternative later. By lemma 2, the output of Algorithm 1 is bounded as follows

$$\frac{b^p}{b^{1/2^i}} = b^{p-1/2^i} \leq r \leq b^{p+1/2^i} = b^p b^{1/2^i}.$$

Observe that $b^p b^{1/2^i} = b^p + b^p(b^{1/2^i} - 1)$ and set $\epsilon = b^{1/2^i} - 1$. We thus have $\frac{1}{2^i} = \frac{\log(1+\epsilon)}{\log(b)}$ and thus $i = \mathcal{O}\left(\log\left(\frac{\log(b)}{\epsilon}\right)\right)$. Similarly, we can write $b^p b^{-1/2^i} = b^p - b^p(1 - b^{-1/2^i})$, arriving at an equivalent result for the $1 - \epsilon$ bound.

The analysis for the case $b < 1$ is the same, but noting that the output is bounded as $b^{p+1/2^i} \leq r \leq b^{p-1/2^i}$. The negative sign of $\log(b)$ is cancelled out in the arithmetic. The case $b = 1$ is obviously of no interest. □

Algorithm 1 approximates a power computation by a dyadic rational approximation w/z of the exponent. Based on the principles behind Algorithm 1 we can design an efficient method to find the critical value of σ for `Kernel` k-`means`.

Finding the critical value. Our algorithm for hyperparameter search is detailed as Algorithm 2. In the pseudocode, $\circ$ and $/\circ$ denote element-wise multiplication and division, respectively, and $\sqrt{K}$ is the element-wise square root of matrix K.

In essence, our algorithm emulates Algorithm 1, using the kernel matrix K_σ as the basis of the power to compute, with some key differences. The first difference is that instead of approximating a known exponent p, we aim to approximate the *unknown* critical value of σ. Since this quantity is unknown, instead of testing whether the current approximation is larger or smaller than the target exponent, we query the `Kernel` k`-means` algorithm to determine whether the current value will result in new changes. Note that this amounts to running a single iteration of `Kernel` k`-means`. Later we show that we can further optimize these queries.

The second observation is that we only ever need to compute exponents in the interval $(0, 1)$. This is because if we assume `Kernel` k`-means` to have converged for the matrix K_σ, we know that the next value of σ we seek is larger than the current one. Note that we can use our result from Theorem 1 for a starting value of σ without running an initial execution of `Kernel` k`-means`.

By virtue of Theorem 2, Algorithm 2 thus finds an arbitrarily good approximation of the critical bandwidth value, in the following sense:

Corollary 1. *Suppose* `Kernel` k`-means` *has converged for K_σ, producing a k-partition P, and let σ' be the critical bandwidth value with respect to (K_σ, P). If we run Algorithm 2 with a depth value of $i = \mathcal{O}\left(\log\left(\frac{|\log(b)|}{\epsilon}\right)\right)$ —where b is the minimum entry in the kernel matrix—, it will output a matrix K_ρ satisfying*

$$(1-\epsilon)K_{\sigma'} \leq_\circ K_\rho \leq_\circ (1+\epsilon)K_{\sigma'},$$

where $\leq_\circ$ denotes element-wise inequality.

That is, it will output a good approximation of the "next" kernel matrix for which `Kernel` k`-means` will make changes. Note that this result also characterizes the computational complexity of our approach, as element-wise operations take $\mathcal{O}(n^2)$ computations. In addition, element-wise operations are trivially parallelizable, so our method can scale to large kernel matrices. Finally, note that even though $\log(b)$ is unbounded, after a few iterations only very small entries, close to zero, would suffer considerable relative error.

An advantage of the algorithm is that we can choose the maximum value of the denominator in the rational approximation of the exponent (maximum depth d). This provides a nice trade-off between speed and accuracy.

6.2 Further optimizations

Our approach lends itself naturally to various optimizations. We discuss them briefly here.

Algorithm 2 Hyperparameter search

Input: kernel matrix K, depth i, k-partition P of X.

1: $K' \leftarrow K$
2: $P' \leftarrow P$
3: $j \leftarrow 1$
4: **for** $i = 1, \ldots, i$ **do**
5: $K' \leftarrow \sqrt{K'}$ //Element-wise square root
6: **if** $P' \neq P$ **then**
7: $K \leftarrow K / \circ K'$
8: **else**
9: $K \leftarrow K \circ K'$
10: $P' \leftarrow kkm(K)$ //Run `Kernel k-means`
11: Output K

Hierarchical search. Our algorithm enables a trade-off between running time and accuracy by means of the depth parameter. The larger it is, the more precise the critical values of σ found. We argue that this parameter can be employed to improve speed without significantly sacrificing accuracy. In particular, the algorithm can be run with increasing depth values, constraining the search to promising regions. For instance, we first set depth to 1, run the algorithm and pick the two values of σ that yield the best performance. We then increase the depth value by 1 and run the algorithm again, setting the lower and upper limits of our search to the two previously picked values of σ. This way we first perform a coarse-grained search to identify a potentially good interval for σ, and then increasingly refine the search.

Limiting checks. As described above, the way our algorithm approximates the critical value of σ is by testing whether or not `Kernel k-means` will switch at least one point from one cluster to another. Often, most points will not switch clusters at the critical value. Thus, it is not necessary to compute the proximity function (Equation (4)) for all point-cluster pairs, and we can limit checks to those points most likely to change. To do this, we can employ different heuristics. For instance, we can limit checks to points such that the proximity function is close for different clusters. We can also limit checks to those points that switch clusters the first time we observe a change (line 6 of Algorithm 2).

6.3 Use with other kernels

Our approach is not limited to the RBF kernel. Obviously, any kernel that is exponential in the parameters can be directly used with our method. This includes the popular polynomial kernel, defined as $\kappa(x, y) = (x^T y + c)^d$, for the optimization of the parameter d. We can also benefit from the fact that any linear combination of kernels is also a kernel, to accommodate a wider variety of kernel functions. To use our algorithm with a linear combination of differently-parametrized kernels, it suffices to store the kernel matrix separately for each

term of the sum. As currently described, our method only allows the optimization of one parameter at a time, but it can be employed as a building block for more sophisticated multiparameter optimization approaches.

7 Experiments

We conduct a series of numerical experiments to evaluate the performance of the proposed algorithm. We mainly want to determine whether our method (1) can reveal good value of σ and (2) can do it efficiently. We compare it to other approaches for hyperparameter search, which we now describe.

Baselines We consider the following methods to choose the hyperparameter of the RBF kernel[1].

MKNN: We set σ to be the mean distance to the k-th nearest neighbour as suggested by Von Luxburg [19] (the median yields similar results). We try different values of k, namely $k = 1, \dots 2(\log n + 1)$.

GridSearch: We run the `Kernel k-means` algorithm with σ taking values in $\{10^i : i = -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6\}$

We refer to Algorithm 2 as OURS.

All methods, as well as `Kernel k-means`, were implemented using Python 3, using matrix and vector operations whenever possible for efficiency[2].

Quality measures: We consider the following functions to evaluate the quality of the clustering results.

NMI (Normalized Mutual Information): We use a well-known clustering performance index[3], which we now define. Given two indicator vectors y and z, we define

$$\mathrm{NMI}(y, z) = \frac{2I(y, z)}{H(y) + H(z)} \tag{6}$$

where $I(y, z) = \sum_i \sum_j p(y = i, z = j) \log\left(\frac{p(y=i,z=i)}{p(y=i)p(z=i)}\right)$ denotes the mutual information of y and z, and $H(y) = -\sum_i p(y = i) \log p(y = i)$ denotes the entropy of y [6] (we abuse notation and overload y for the vector and its entries). We use this index by taking y to be the indicator vector of ground-truth labels and z to be the indicator vector of the k-partition output by `Kernel k-means`.

c-NNC: In addition, we propose our own clustering cost function. Our goal is to measure the quality of the resulting k-partition in a way that (1) arbitrarily shaped clusters are considered and (2) is independent of the value of σ. Note that some well-known clustering quality indices and cost functions, such as silhouette [17] and normalized cuts [7], do not qualify.

[1] Some of these methods, as originally described, define the kernel as $\kappa(x, y) = \exp(-\|x-y\|/(2\sigma^2))$. We take this difference into account in our experimental setup.

[2] Source code: `https://github.com/justbruno/off-the-grid/`

[3] Results for Adjusted Rand-Index were similar and are thus omitted.

We first introduce some notation. Given a data set X and a point $x_i \in X$, let $\nu_j(x_i)$ be the j-th nearest neighbour of x_i in X. Given a k-partition of the data set X into k clusters, $c(x_i)$ denotes the cluster x_i is assigned to, i.e. $x_i \in c(x_i)$.

We first define $\mathrm{NNC}(i, c)$ to be the fraction of points among the c nearest neighbours of x_i which are not in the same cluster as x_i.

$$\mathrm{NNC}(i, c) = \frac{1}{c} \sum_{j=1}^{c} \mathbb{I}\{c(x_i) \neq c(\nu_j(x_i))\}.$$

To measure the quality of a single cluster π, we take a weighted sum of the above index for all c. We scale the value of $\mathrm{NNC}(i, c)$ by $\frac{1}{c}$ to reduce the penalty incurred by disagreements with further neighbours.

$$\mathrm{NNC}_{cluster}(\pi) = \frac{1}{C \max\{1, |\pi|)\}} \sum_{i \in \pi} \sum_{c=1}^{n} \frac{1}{c} \mathrm{NNC}(i, c).$$

Here, $C = \log(n-1) + \gamma + \frac{1}{2n-2}$, where γ is the Euler-Mascheroni constant, ensures that the quantity is upper-bounded by 1 (note that without this scaling factor, the sum for each point is tightly upper bounded by a harmonic series).

We now define the cost function as

$$\text{c-NNC}(P) = \frac{D + \sum_{\pi \in P} \mathrm{NNC}_{cluster}(\pi)}{k}.$$

Here, P is the k-partition output by `Kernel k-means`, k is the number of clusters given to `Kernel k-means` and D is the number of empty clusters. We count empty clusters to penalize trivial solutions (e.g. a single cluster).

Datasets : We employ a variety of publicly available synthetic[4] and real[5] data sets. Since we use vanilla `Kernel k-means`, which requires handling the complete kernel matrix, we employ data sets of limited size (up to 8 000 instances). However, our method can in principle be employed with techniques for scalable kernel-based algorithms. A summary of the data sets is given in Table 1. In the case of real data sets, we scale the variables to unit-variance, as this enables a much better performance of `Kernel k-means` in most cases.

7.1 Performance

In this section we report the performance of our method, as evaluated by our quality measures, in comparison to the selected baselines. We proceed as follows: we first choose a random initial k-partition, which we set as starting point for all methods. To evaluate our method, we set the initial value of σ to be the 1st percentile of pairwise distances in the data set. Note this is similar to our lower bound given in section 4, but a little less stringent. We run Algorithm 2

[4] `http://cs.joensuu.fi/sipu/datasets`
[5] `https://archive.ics.uci.edu/ml/index.php`

Table 1. Summary of data set characteristics

Dataset	Rows	Columns	Classes	Dataset	Rows	Columns	Classes
Aggr.	788	2	7	Spiral	312	2	3
Compound	399	2	6	Audit	775	23	2
D31	3100	2	31	Derma.	358	34	6
Flame	240	2	2	WDBC	569	30	2
Jain	373	2	2	WiFi	2000	7	4
Pathbased	300	2	3	Wine	178	13	3
R15	600	2	15	MNIST (sampled)	1k,2k,4k,8k	784	10

with depth=1 and pick the value of σ that corresponds to the best observed k-partition (as measured by c-NNC), run `Kernel k-means` and rerun our method starting from the resulting k-partition with depth= 2. Note that this resembles the hierarchical search described in section 6. For each method, we collect the best value of NMI and c-NNC among the produced clusterings. We report the average over 50 runs, each with a different initial k-partition. Results are shown in Table 2. Our method achieves better values of both measures in most cases.

Table 2. Comparison of the different methods in terms of quality measures

	NMI			c-NNC		
Dataset	MKNN	GridSearch	OURS	MKNN	GridSearch	OURS
Aggr.	0.690	0.864	**0.872**	0.255	0.210	**0.203**
Compound	0.689	**0.778**	0.730	0.239	0.230	**0.215**
D31	0.810	0.931	**0.951**	0.356	0.332	**0.316**
Flame	0.489	0.521	**0.615**	0.106	0.096	**0.093**
Jain	0.229	**0.361**	0.353	0.116	0.062	**0.062**
Pathbased	0.820	0.662	**0.902**	0.169	**0.134**	0.137
R15	0.922	0.954	**0.979**	0.302	0.300	**0.274**
Spiral	0.187	0.145	**0.239**	0.175	0.155	**0.151**
Audit	**0.717**	0.685	0.703	0.097	**0.082**	**0.082**
Derma.	0.889	0.877	**0.913**	0.249	0.256	**0.238**
WDBC	0.531	0.547	**0.550**	0.123	0.108	**0.107**
WiFi	0.781	0.835	**0.856**	0.157	0.140	**0.137**
Wine	**0.923**	0.913	**0.923**	0.143	**0.142**	0.143

7.2 Running times and Scalability

In this section we evaluate the efficiency of our method. We report the average total running times in the previously described experiment for all algorithms in Table 3. Our method generally sits between GridSearch and MKNN. It performs significantly more iterations than the baselines, and thus better running times could be obtained by limiting the number of inspected values if necessary.

Table 3. Total running times in seconds

	Time in seconds				Time in seconds		
Dataset	MKNN	GridSearch	OURS	Dataset	MKNN	GridSearch	OURS
Aggr.	0.824	**0.486**	0.617	Spiral	0.157	**0.108**	0.133
Compound	0.200	**0.146**	0.172	Audit	0.578	**0.467**	1.111
D31	22.029	11.735	**10.757**	Derma.	0.152	0.106	**0.100**
Flame	0.064	**0.046**	0.061	WDBC	0.325	**0.204**	0.213
Jain	0.140	**0.097**	0.143	WiFi	7.095	**3.520**	4.402
Pathbased	0.110	**0.077**	0.094	Wine	0.044	**0.034**	0.036
R15	0.475	**0.329**	0.406				

To offer a finer running time comparison, as well as to evaluate scalability, we run the algorithms on samples of MNIST[6] and set the number of iterations to be the same for all methods. In particular, we set it to 13, which is the number of values tested by GridSearch. Figure 1 shows time taken per iteration, averaged over 50 runs. By iteration we refer to the set of computations required to produce and test a new value of the bandwidth parameter. The reason the running time of GridSearch increases significantly at some point is that the first values of σ are too small and `Kernel` k`-means` converges after one iteration, highlighting the wasteful nature of GridSearch. Our method benefits mostly from being able to run a small number of iterations of `Kernel` k`-means` to converge.

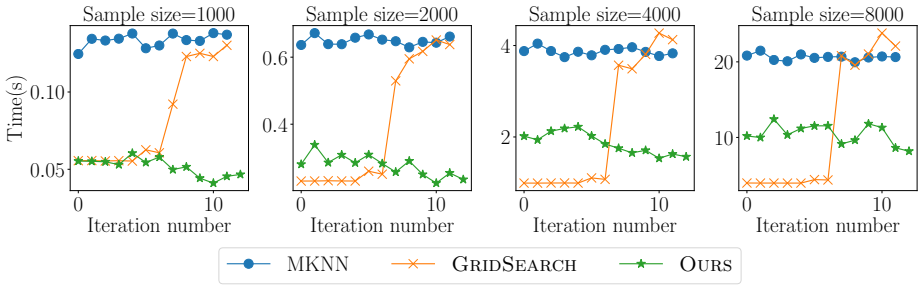

Fig. 1. Running time per iteration for different samples of the MNIST data set

7.3 Comparison with binary search

The reader might observe that our method resembles a form of binary search. Thus, one might suspect that similar results could be obtained using a conventional binary search algorithm, without going to the trouble of implementing Algorithm 2. Here we illustrate why our algorithm is a vastly superior alternative.

[6] `http://yann.lecun.com/exdb/mnist/`

The setup is as follows: we initialize σ to be the 1st percentile of the squared pairwise distances and then run iterations of binary search with a precision of 10^{-3} and Algorithm 2 with depth equal to 10. We repeat the experiment 10 times and report average iteration time and absolute error of the estimate of the critical value of σ. The results are shown in Table 4. Binary search was implemented efficiently, updating the kernel matrix with fast matrix-vector operations.

Our method achieves a speedup of about 10x in all cases, and the error is often smaller. Of course, the error can be controlled in both algorithms at the expense of running time. A noteworthy difference between both methods (not in favor of any of the two) is that binary search is designed to control absolute error, while Algorithm 2 controls the relative error of the power computation.

Table 4. Running times of our method and binary search. We report average iteration running times, speedup and mean relative error of the σ estimate over 100 iterations

	Iteration time in seconds		Speedup	Relative error: $\frac{\sigma_{\text{true}}-\sigma_{\text{estimated}}}{\sigma_{\text{true}}}$	
Dataset	BinarySearch	OURS	-	BinarySearch	OURS
Aggr.	0.941	0.080	11.7x	1.55×10^{-3}	5.3×10^{-4}
Audit	0.793	0.069	11.5x	8.341×10^{-2}	5.8×10^{-4}
Compound	0.192	0.019	9.9x	2.20×10^{-3}	5.8×10^{-4}
D31	15.740	1.148	13.7x	3.95×10^{-3}	$4.8 \times 10^{-}$
Derma.	0.139	0.014	9.7x	1.3×10^{-4}	8.368×10^{-2}
Flame	0.063	0.007	9.1x	5.1×10^{-3}	5.6×10^{-4}
Jain	0.144	0.014	10.5x	2.6×10^{-3}	1.17×10^{-2}
Pathbased	0.096	0.010	9.7x	1.98×10^{-3}	5.9×10^{-4}
R15	0.430	0.039	11.0x	4.912×10^{-2}	5.5×10^{-4}
Spiral	0.102	0.011	9.6x	1.13×10^{-3}	6.2×10^{-4}
WDBC	0.398	0.036	11.2x	10^{-6}	6.2×10^{-4}
WiFi	5.284	0.442	11.9x	6×10^{-5}	4.9×10^{-4}
Wine	0.042	0.005	7.7x	2×10^{-5}	5.7×10^{-4}

8 Conclusion

In this paper we have addressed the problem of hyperparameter search in the `Kernel` k`-means` context. Our contribution is two-fold. First, we have derived a tight lower bound for the bandwidth parameter of RBF kernels, below which `Kernel` k`-means` will be rendered useless. Second, we have proposed a method to optimize kernel hyperparameters for `Kernel` k`-means`. We have proved that our method approximates critical values of the hyperparameter to arbitary precision in a small number of iterations. Unlike grid search or other heuristics, our method does not test redundant hyperparameter values, that is, values that result in the same clustering output, and thus no computation is wasted.

Our experiments demonstrate how our approach enables the efficient evaluation of a fine variety of hyperparameter values, revealing high-quality clustering

results at a moderate computational cost. In the future it would be interesting to extend our method to other kernel-based clustering and classification algorithms.

References

1. Aloise, D., Deshpande, A., Hansen, P., Popat, P.: Np-hardness of euclidean sum-of-squares clustering. Machine learning **75**(2), 245–248 (2009)
2. Bach, F.R., Jordan, M.I.: Learning spectral clustering. In: Advances in neural information processing systems. pp. 305–312 (2004)
3. Ben-David, S., Von Luxburg, U., Pál, D.: A sober look at clustering stability. In: International Conference on Computational Learning Theory. pp. 5–19. Springer (2006)
4. Ben-Hur, A., Horn, D., Siegelmann, H.T., Vapnik, V.: Support vector clustering. Journal of machine learning research **2**(Dec), 125–137 (2001)
5. Cortes, C., Vapnik, V.: Support-vector networks. Machine learning **20**(3), 273–297 (1995)
6. Cover, T.M., Thomas, J.A.: Elements of information theory. John Wiley & Sons (2012)
7. Dhillon, I.S., Guan, Y., Kulis, B.: Kernel k-means: spectral clustering and normalized cuts. In: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 551–556. ACM (2004)
8. Ester, M., Kriegel, H.P., Sander, J., Xu, X., et al.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: Kdd. vol. 96, pp. 226–231 (1996)
9. Gordon, D.M., et al.: A survey of fast exponentiation methods. J. Algorithms **27**(1), 129–146 (1998)
10. Greene, D., Cunningham, P.: Practical solutions to the problem of diagonal dominance in kernel document clustering. In: Proceedings of the 23rd international conference on Machine learning. pp. 377–384. ACM (2006)
11. Hofmann, T., Schölkopf, B., Smola, A.J.: Kernel methods in machine learning. The annals of statistics pp. 1171–1220 (2008)
12. Jaakkola, T.S., Diekhans, M., Haussler, D.: Using the fisher kernel method to detect remote protein homologies. In: ISMB. vol. 99, pp. 149–158 (1999)
13. Jain, A.K.: Data clustering: 50 years beyond k-means. Pattern recognition letters **31**(8), 651–666 (2010)
14. Lloyd, S.: Least squares quantization in pcm. IEEE transactions on information theory **28**(2), 129–137 (1982)
15. Moon, T.K.: The expectation-maximization algorithm. IEEE Signal processing magazine **13**(6), 47–60 (1996)
16. Ng, A.Y., Jordan, M.I., Weiss, Y.: On spectral clustering: Analysis and an algorithm. In: Advances in neural information processing systems. pp. 849–856 (2002)
17. Rousseeuw, P.J.: Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. Journal of computational and applied mathematics **20**, 53–65 (1987)
18. Vapnik, V.: Estimation of dependences based on empirical data. Springer Science & Business Media (2006)
19. Von Luxburg, U.: A tutorial on spectral clustering. Statistics and computing **17**(4), 395–416 (2007)
20. Zelnik-Manor, L., Perona, P.: Self-tuning spectral clustering. In: Advances in neural information processing systems. pp. 1601–1608 (2005)

Appendix

Proof of Theorem 1. Consider the squared distance as written in Eq. (2). For the choice of cluster, we can drop the constant term $\kappa(x,x)$ and compute

$$\arg\min_j \; -\frac{2\sum_{y\in\pi_j}\kappa(x,y)}{n_j}+\frac{\sum_{y,z\in\pi_j}\kappa(y,z)}{n_j^2}. \tag{7}$$

If $x\in\pi_j$ we can write

$$-\frac{2\sum_{y\in\pi_j}\kappa(x,y)}{n_j}+\frac{\sum_{y,z\in\pi_j}\kappa(y,z)}{n_j^2} \tag{8}$$

$$=-\frac{2\kappa(x,x)}{n_j}-\frac{2\sum_{y\in\pi_j,y\neq x}\kappa(x,y)}{n_j}+\frac{\sum_{y,z\in\pi_j}\kappa(y,z)}{n_j^2}. \tag{9}$$

If $x\in\pi_j$ and $\|\phi(x)-m_j\|_2^2\leq\|\phi(x)-m_i\|_2^2$ for all $i\neq j$, then a will remain in the same cluster. Considering (7) and (8), we can write this condition as

$$\begin{aligned}\frac{2\kappa(x,x)}{n_j}\geq&\frac{2\sum_{y\in\pi_i}\kappa(x,y)}{n_i}-\frac{\sum_{y,z\in\pi_i}\kappa(y,z)}{n_i^2}\\&-\frac{2\sum_{y\in\pi_j,y\neq x}\kappa(x,y)}{n_j}+\frac{\sum_{y,z\in\pi_j}\kappa(y,z)}{n_j^2}.\end{aligned}$$

Since $\kappa(x,y)\geq 0$ for any pair of points x,y, we can drop the negative terms on the right-hand side to obtain the following, more restrictive, condition:

$$\begin{aligned}\frac{2\kappa(x,x)}{n_j}&\geq\frac{2\sum_{y\in\pi_i}\kappa(x,y)}{n_i}+\frac{\sum_{y,z\in\pi_j,y\neq z}\kappa(y,z)+n_j}{n_j^2}\\&=\frac{2n_j^2\sum_{y\in\pi_i}\kappa(x,y)+n_i\left(\sum_{y,z\in\pi_j,y\neq z}\kappa(y,z)+n_j\right)}{n_in_j^2}.\end{aligned} \tag{10}$$

Here we have used $\sum_{x\in\pi_j}\kappa(x,x)=n_j$. If we define $\omega=\max_{x\neq y}\kappa(x,y)$, then the two following inequalities hold:

$$n_i\omega\geq\sum_{y\in\pi_i}\kappa(x,y),\qquad\qquad n_j^2\omega\geq\sum_{y,z\in\pi_j,y\neq z}\kappa(y,z)$$

We can thus consider the following, more restrictive, condition (recall that $\kappa(x,x)=1$):

$$\frac{2\kappa(x,x)}{n_j}\geq\frac{2n_j^2n_i\omega+n_i\left(n_j^2\omega+n_j\right)}{n_in_j^2}=3\omega+\frac{1}{n_j}\Leftrightarrow\frac{1}{n_j}\geq 3\omega. \tag{11}$$

Trivially, $\frac{1}{n}\geq 3\omega\Rightarrow\frac{1}{n_j}\geq 3\omega$. Now, after minor computational efforts, it is

$$\frac{1}{n}\geq 3\omega\Leftrightarrow-\log n\geq\log 3-\frac{\|x-y\|_2^2}{\sigma}\Leftrightarrow\frac{\|x-y\|_2^2}{\log 3n}\geq\sigma, \tag{12}$$

where $x, y = \arg\min_{x.y \in X} \|x - y\|_2^2$. Therefore, we have (12) $\Rightarrow$ (11) $\Rightarrow$ (10) $\Rightarrow$ $\|\phi(x) - m_j\|_2^2 \leq \|\phi(x) - m_i\|_2^2$ for all $i \neq j$. Since x is an arbitrary element of X, (12) is a sufficient condition for kernel k-means to make no changes. □