Bachelorarbeit

Kernel k-means clustering framework in Python

Verfasser

Paul Philipp Maria Theis

angestrebter akademischer Grad

Bachelor of Science (BSc)

Wien, 2023

Studienkennzahl lt. Studienblatt:	A 033 521
Fachrichtung:	Informatik - Informatik Allgemein
Betreuer:	Dipl.-Ing. Marian Lux

Abstract

K-means is, despite its restriction on linearly seperable data, one of the most popular clustering algorithms. Kernel k-means alleviates this restriction by utilizing the kernel-trick to make originally non-seperable data seperable. Despite this utility, there is no robust implementation of kernel k-means in Python. This thesis provides both an in-depth review of the algorithm and a robust, fast and well-tested kernel k-means implementation.

Contents

1 Motivation

According to [8], clustering is considered the most important task in unsupervised learning, and the k-means algorithm[1] is one of the most popular existing clustering algorithms. As such, there are lots of implementations in a lot of different programming languages available. Despite its popularity, the algorithm has some problematic shortcomings, the biggest being that it is only able to to cluster linearly seperable data. Also, since the k-means problem is NP-hard (even for two dimensions) [20], one has to rely on heuristics such as the one provided by [18] which don't necessarily find a global optimum. They therefore heavily rely on the initial seeding of the clusters, in respect to both iterations necessary till convergence and quality of the result.

Inderjit et al [7] provide (among other things) an algorithm utilizing the kernel-trick to enable the k-means clustering of non-linearly seperable data. With kmeans++ [2], there also exists a method to consistently achieve good initial seedings

This thesis aims to change that by providing a robust, well-tested and fast implementation of the kernel k-means algorithm from the ground up, along with utilities like kmeans++.

A problem that appears ubiquitously when clustering is that the number of clusters is usually not known before. To alleviate this, some quality metrics have also been implemented to provide an indicator on how well the algorithm has performed.

2 Related Work

Despite a lot of research and the popularity of the original k-means, the author has only found two implementations of kernel k-means in Python. One was created as a github gist and did not evolve above this stage [4], and one is included in tslearn [27]. Both build heavily upon scikit-learn and SciPy and both break when encountering empty clusters which, especially when not using a good heuristic to initialize clusters or simply through bad luck, can happen frequently. Also, both make frequent use of NumPy's boolean indexing, which leads to performance issues when working with a huge dataset as it creates a copy of the underlying data.

Other than NumPy, this implementation uses only Cython to avoid said expensive indexing and is otherwise independent from other libraries. Also, the framework is robust in regards to events like empty clusters.

[1] [18] appears to be the defacto standard for a heuristic that solves the k-means problem, often "lloyd's algorithm" and k-means clustering are used synonymously [7][2]. In this thesis this practice is continued, the heuristic from [18] is meant when referring to "the k-means algorithm".

3 Preliminaries

3.1 The k-means problem

Using the classifications of clustering methods provided in [10], k-means falls into the category of partitioning clustering methods; meaning algorithms that attempt to split n given datapoints (which usually can be interpreted as elements of $\mathbb{R}^d$) into k clusters so that some given objective function is optimized.

For k-means, the objective function is to minimize the sum of squared distances (also called variance or inertia [24]) from each point to the center of the cluster it is assigned to. In other words: for a dataset $\mathcal{X}$, k-means tries to find a set of clusters $\mathcal{C} = \{C_1, \ldots, C_k\}$ so that the error function $\sum_{i=1}^{k} \sum_{x \in C_i} ||x - \mu_i||_2^2$ is minimized. μ_i is the average of all points $\frac{1}{|C_i|} \cdot \sum_{x \in C_i} x$ in cluster C_i.

As stated before, finding the global optimum for this objective is NP-hard, therefore heuristics have to be used. There are many alternatives to Lloyd's algorithm, for example there is also a version from Macqueen [19] and one from Hartigan [12], the latter having been investigated as an attractive alternative to Lloyd's [26]. Since both Macqueen's and Hartigan's would be heavily penalized when utilizing kernels this thesis will focus on Lloyd's version.

3.2 Lloyd's algorithm

Algorithm 1 Lloyd's k-means

Input: Data $\mathcal{X}$, number of clusters k
Output: Clustering $\mathcal{C}$

1: Randomly choose cluster centers $\{\mu_1, \ldots, \mu_k\}$ from $\mathcal{X}$
2: **while** `not Converged` **do** // criteria vary
3: **for all** $x \in \mathcal{X}$ **do**
4: $\boldsymbol{i}^*(x) \leftarrow \min_i ||x - \mu_i||_2^2$
5: **for** i `in` $\{1, \ldots, k\}$ **do**
6: $C_i \leftarrow \{x \mid \boldsymbol{i}^*(x) = i\}$
7: $\mu_i \leftarrow \frac{1}{|C_i|} \cdot \sum_{x \in c_i} x$
8: **return** $\mathcal{C} = \{C_i \mid i \in \{1, \ldots, k\}\}$

More informal, the algorithm first calculates the distances from each point to each cluster center, then assigns each point to the closest center, thereby forming new clusters which in turn have new centers. This repeats until convergence is reached, usually by having reached clusters that don't change over the iterations, a maximal number of iterations reached or the cluster-centers only moving below a given threshold.

The algorithmic complexity of this is $\mathcal{O}(nkt)$[2], n being the amount of datapoints $|\mathcal{X}|$, k the number of clusters and t the amount of iterations needed to

[2]This assumes k being bigger than the dimensionality d of the feature space, as there are nt distance calculations aswell (each one being in $\mathcal{O}(d)$).

converge. Utilizing the triangle-inequality (and saving the μ for one iteration) there has been achieved significant speedups for this algorithm [9] by avoiding unnecessary distance calculations.

3.3 MacQueen, Hartigan-Wong

3.3.1 MacQueen

The difference between MacQueen's and Lloyd's is that MacQueens algorithm computes the center of a cluster anew each time a point is assigned to it. For kernelization, this is problematic as the centers are implicit which makes the distance calculations much more expensive than in regular k-means. Therefore, lowering the iterations through more precision (as centers are more accurate) does not lead to an efficiency gain.

3.3.2 Hartigan-Wong

This algorithm is more sophisticated than both Lloyd's and MacQueen's, as it considers the impact of reassigning a datapoint to another cluster. The problem with kernelization is the same as in MacQueen's, as the distances are recomputed for each datapoint.

3.4 K-means++

A commonly used method to improve k-means is k-means++, which "substantially improves both the running time and the accuracy" [2] by reducing its dependency on being lucky when randomly choosing the initial cluster centers. So instead of doing step 1 in the pseudocode above, kmeans++ instead chooses the center in the following manner:
Let $D(x)$ be the distance from x to its closest cluster.

1. Choose first center μ_1 randomly from $\mathcal{X}$
2. Choose $\hat{x} \in \mathcal{X}$ as μ_i with probability $\frac{D(\hat{x})^2}{\sum_{x \in \mathcal{X}} D(x)^2}$
3. Repeat 2. until k centers are found.

3.5 Quality measurement

The standard quality measurement (which is also used in `sklearn.cluster.KMeans`) is the sum of squared distances (called *variance* or *inertia*). Another commonly used internal indicator to measure the quality of any partitional clustering is the average silhouette of a clustering [22] [8]. As inertia suffers heavily from the curse of dimensionality, it is advisable to use the average silhouette when clustering high-dimensional data or apply some dimensionality reducing technique like PCA.

3.5.1 Silhouette Coefficient

For each $x \in \mathcal{X}$ the following values have to be computed, where C_x is the cluster x is assigned to. For (kernel) k-means, the dissimilarity is given by the distance of the objects.

- $a(x)$: average dissimilarity of x to all other objects of C_x
- $d(x, C)$: average dissimilarity of x to all objects of C
- $b(x)$: $\min_{C \neq C_x} d(x, C)$

The silhouette of x is then given by

$$s(x) = \frac{b(x) - a(x)}{\max(a(x), b(x))}$$

which yields a value between -1 and 1 that indicates how sensible the assignment of x to cluster C_x is. A value close to zero means that there would be no meaningful difference if x were assigned to the next-nearest cluster after C_x and a negative value suggests x should better be assigned to another cluster. To measure the quality of a clustering, the average silhouette of the entire dataset $\frac{1}{|\mathcal{X}|} \cdot \sum_{x \in \mathcal{X}} s(x)$ can be utilized. The same notion also works to check the quality of a single cluster [22].

To determine a reasonable number of clusters, it is possible to run k-means with different k and check the resulting average silhouette of the result This leads to the definition of the *Silhouette Coefficent* $\mathrm{SC} = \max_k(\hat{s}(k))$. Here, $\hat{s}(k)$ is the average silhouette of the entire dataset when clustered to k clusters.

3.6 Elkan's k-means

Probably the most common optimization for Lloyd's algorithm is Elkan's algorithm (often just called Elkan's k-means) [9]. It utilizes the triangle inequality (and saves information through iterations) to avoid unnecessary distance calculations. While it doesn't change the algorithm's complexity, empirically a speedup of $\times 300$ and beyond has been seen on linearly seperable data [9].

3.6.1 Pseudocode

As Elkan is focused rather on *how* calculations should be done rather than *what* to calculate, the pseudocode ended up being more technical and less abstracted.[3]

[3] Also, the algorithm has been slightly rearranged to help with implementation.

Algorithm 2 Elkan's k-means

Input: Data $\mathcal{X}$, number of clusters k
Output: Clustering $\mathcal{C}$

$I \leftarrow \{1, \ldots, k\}$ // For notational simplicity
Initilaize centers $\mu_i \ \forall i \in I$
$\mathrm{L}(x, \mu_i) \leftarrow d(x, \mu_i) \ \forall x \in \mathcal{X} \ \forall i \in I$ // d being the distance
$\mu_x \leftarrow \mathrm{argmin}_{\mu_i} \mathrm{L}(x, \mu_i) \ \forall x \in \mathcal{X}$
while `not Converged` **do** // criteria vary
    $\hat{\mu_i} \leftarrow \mu_i \ \forall i \in I$
    $\mu_i \leftarrow \mathrm{avg}(\{x \mid \mu_x = \hat{\mu_i}\}) \ \forall i \in I$
    $\mathrm{L}(x, \mu_i) \leftarrow max\{L(x, \mu_i) - d(\mu_i, \hat{\mu_i}), 0\} \ \forall x \in \mathcal{X} \ \forall i \in I$
    store $d(\mu_i, \mu_j) \ \forall i, j \in I$
    $\mathrm{U}(x) \leftarrow \mathrm{U}(x) + d(\mu_x, \hat{\mu_x})$
    $\mathrm{S}(\mu_i) \leftarrow \frac{1}{2} \min_{j \neq i} d(\mu_i, \mu_j) \ \forall i \in I$
    for all $x \in X$ **do**
        if $\mathrm{U}(x) \leq \mathrm{S}(\mu_x)$ **then**
            `continue`
        $\mathrm{R} \leftarrow$ `True`
        for all $i \in I$ **do**
            if $\mu_i = \mu_x$ **or** $\mathrm{U}(x) \leq \mathrm{L}(x, \mu_i)$ **or** $\mathrm{U}(x) \leq \frac{1}{2} d(\mu_x, \mu_i)$ **then**
                `continue`
            if R **then**
                $\mathrm{U}(x) = d(x, \mu_x)$
                $\mathrm{R} \leftarrow$ `False`
            if $\mathrm{U}(x) > \mathrm{L}(x, \mu_i)$ **or** $\mathrm{U}(x) > \frac{1}{2} d(\mu_x, \mu_i)$ **then**
                $\mathrm{L}(x, \mu_i) \leftarrow d(x, \mu_i)$
                if $\mathrm{L}(x, \mu_i) < \mathrm{U}(x)$ **then**
                    $\mathrm{U}(x) \leftarrow \mathrm{L}(x, \mu_i)$
                    $\mu_x \leftarrow \mu_i$
$C_i \leftarrow \{x \mid \mu_x = \mu_i\} \ \forall i \in I$
return $\mathcal{C} = \{C_i \mid i \in \{1, \ldots, k\}\}$

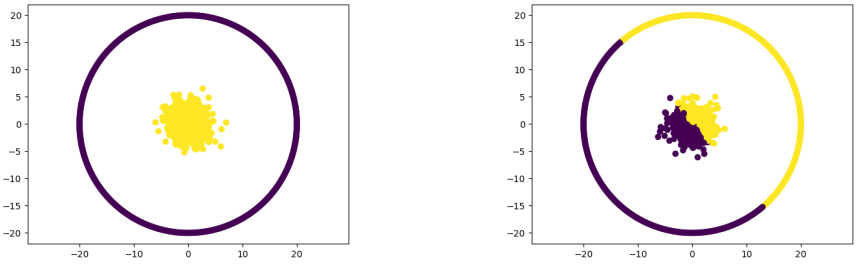
Step 3 can be sped up by utilizing the triangle equality, but the efficiency gain is minimal as it only done once throughout the algorithm.

4 Kernelization

4.1 The problem

As mentioned before, the k-means algorithm is only able to find clusters when the data is linearly separable. Figure 1 illustrates this.

The method this thesis will discuss to resolve this problem is to project the data in some higher-dimensional space in such a manner that a separating hyperplane can be found. In the (highly constructed) illustrated example one



(a) Non linearly-separable Data (b) Example clustering via k-means

Figure 1

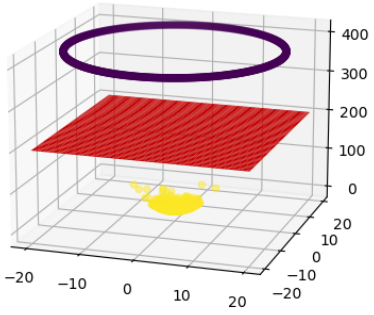


Figure 2: Projected Data with exemplary hyperplane

could utilize a function $\phi((x, y)) \to (x, y, x^2 + y^2)$ to project the data into $\mathbb{R}^3$, where (Figure 2) it is obvious that the data can be clustered.

Here, the projection was simple and for computing k-means the additional dimension adds very little (absolute) computational overhead. But what if the projection needs to be made to a very high (or even infinite) dimensional space to get proper results? The mathematical solution are *kernel functions*.

4.2 Kernel functions

Definition (Kernel Function)**.** [23] A *kernel* is a function κ so that $\forall x, y \in \mathcal{X}$: $\kappa(x, y) = \langle \phi(x), \phi(y) \rangle$, where ϕ is a mapping from input space $\mathcal{X}$ to a hilbert space H with inner product $\langle \cdot, \cdot \rangle$.

Definition (Gram Matrix)**.** [23] Given a set of vectors $S = \{x_1, \ldots, x_l\}$, a *Gram Matrix* is an $l \times l$ matrix whose entries are $G_{i,j} = \langle x_i, x_j \rangle$. When utilizing a

kernel κ to calculate the inner product the matrix is called *Kernel Matrix* and is referred to by $\mathbf{K}$.

A necessary and sufficient condition for a function to be a valid kernel (without having to compute ϕ explicitly, which might be impossible when it maps to an infinite dimensional hilbert space) is if the kernel matrix is positive semidefinite for all choices $S \subseteq \mathcal{X}$ [3][4].

So, kernel functions allow to calculate the inner product of elements from the input space $\mathcal{X}$ in a feature (hilbert-)space H *without having to calculate their representation* ($\phi(x)$) in H. This makes the computation of the scalar products independent from the dimensions ϕ maps to. Calculating the scalar product in this manner is also often referred to as the *kernel trick*. As a scalar-product always induces a distance (Appendix B), using the kernel trick is a way to calculate the distance of two points from the input space in the feature space without having to know (or calculate) the feature map ϕ.

4.2.1 Popular Kernels

The popularity of a kernel was determined based on whether it was implemented in `sklearn.metrics.pairwise` [25]. The formulations used are taken from there.

- Linear Kernel: $\kappa(x, y) = x^\top y$
- Polynomial Kernel: $\kappa(x, y) = (\gamma x^\top y + c_0)^d$
- Sigmoid Kernel: $\kappa(x, y) = \tanh(\gamma x^\top y + c_0)$
- RBF Kernel: $\kappa(x, y) = \exp(-\gamma ||x - y||^2)$
- Laplacian Kernel: $\kappa(x, y) = \exp(-\gamma ||x - y||_1)$
- Chi-squared Kernel: $\kappa(x, y) = \exp(-\gamma \sum_i \frac{(x[i] - y[i])^2}{x[i] + y[i]})$

A special case of the RBF Kernel where γ is $-\sigma^2$ is also called the *Gaussian Kernel* with variance σ^2. When using the Linear Kernel, kernel k-means just becomes a computationally more expensive version of regular k-means.

Despite being used frequently, the parameters for the sigmoid kernel can be choosen so that it is not positive semidefinite [17].

[4] A more in-depth criteria for kernels is provided by *Mercer's Theorem*. Its mathematical scope surpasses the level of this bachelor thesis, but it is included in most of the books on machine learning referenced. A rather conclusive formulation can be found in [28]

4.3 Kernel K-means

Applying the kernel-trick to k-means directly is not possible as, since it is not given that ϕ can be evaluated, there is no way to explicitely calculate the centers in the feature space, which would be necessary for execution. Instead, we will utilize the distance induced by the kernel function and some rearranging to do this calculation. As cluster center in the feature space, the intuition of

$$\mu_i = \frac{\sum_{x \in C_i} \phi(x)}{|C_i|} \text{ is correct, as } \mu_i = \min_z \sum_{x \in C_i} ||\phi(x) - z||^2 .[7]$$

Now, to calculate $||\phi(x) - \mu_i||^2$ the following equation is used:

$$\begin{aligned}
&||\phi(x) - \mu_i||^2 = ||\phi(x) - \frac{\sum_{y \in C_i} \phi(y)}{|C_i|}||^2 \\
&= \langle \phi(x) - \frac{\sum_{y \in C_i} \phi(y)}{|C_i|}, \phi(x) - \frac{\sum_{y \in C_i} \phi(y)}{|C_i|} \rangle \\
&= \langle \phi(x), \phi(x) \rangle - 2\langle \phi(x), \frac{\sum_{y \in C_i} \phi(y)}{|C_i|} \rangle + \langle \frac{\sum_{y \in C_i} \phi(y)}{|C_i|}, \frac{\sum_{y \in C_i} \phi(y)}{|C_i|} \rangle \qquad (1) \\
&= \langle \phi(x), \phi(x) \rangle - \frac{2 \sum_{y \in C_i} \langle \phi(x), \phi(y) \rangle}{|C_i|} + \frac{\sum_{y \in C_i} \sum_{\hat{y} \in C_i} \langle \phi(y), \phi(\hat{y}) \rangle}{|C_i|^2} \\
&= \kappa(x, x) - \frac{2 \sum_{y \in C_i} \kappa(x, y)}{|C_i|} + \frac{\sum_{y, \hat{y} \in C_i} \kappa(y, \hat{y})}{|C_i|^2}
\end{aligned}$$

Algorithm 3 Kernel k-means

Input: Data $\mathcal{X}$, number of clusters k
Output: Clustering $\mathcal{C}$

1: Calculate Kernel-matrix $\mathbf{K}$ // $\mathbf{K}_{ij} = \kappa(x_i, x_j)$
2: Initialize $\mathcal{C}$
3: **while** `not Converged` **do** // criteria vary
4: **for all** $x \in \mathcal{X}$ **do**
5: $\boldsymbol{i}^*(x) \leftarrow \min_i ||\phi(x) - \mu_i||_2^2$ // Utilize Eq. 1
6: **for** i `in` $\{1, \ldots, k\}$ **do**
7: $C_i \leftarrow \{x \mid \boldsymbol{i}^*(x) = i\}$
8: **return** $\mathcal{C} = \{C_i \mid i \in \{1, \ldots, k\}\}$

Calculating the kernel matrix is in $\mathcal{O}(n^2 d)$, d being the dimensionality of the data-points. Calculating step 4 and 5 is in $\mathcal{O}(n^2)$, as the first term ($\kappa(x, x)$) of Eq. 1 is already stored in $\mathbf{K}$ (and could even be omitted as it is a constant for each datapoint and therefore does not affect its assignment to a cluster), calculating the second term is in $\mathcal{O}(n)$ and is calculated n times per iteration and the third term is by itself in $\mathcal{O}(n^2)$ aswell, but is a constant for each cluster

and can therefore be calculated k times and then stored for the iteration. As a result, if the algorithm iterates t times until convergence the total runtime is in $\mathcal{O}(n^2(t+d))$[5] [7].

4.3.1 Initialization and quality

To create the initial clusters the methods for regular k-means cluster initialization require only slight adjustments. For true randomness, each x can be assigned to a random C_i. Kmeans++ can apply Eq.1 naturally, especially as the centers are explicitely given. The distance calculations can be done efficiently via $\kappa(x,x) - 2\kappa(x,\mu) + \kappa(\mu,\mu)$ where both x and μ are elements of the dataset $\mathcal{X}$. Same goes for the original approach of choosing k elements from $\mathcal{X}$.[6]

As calculating the average silhouette would be possible but computationally expensive, a slightly simplified version [13] is more adequate for kernel-k-means. The functions described in subsection 3.5 change to the following:

- $a(x)$: $||\phi(x) - \mu_x||$
- $d(x, C_i)$: $||\phi(x) - \mu_i||$
- $b(x)$: $\min_{C \neq C_x} d(x, C)$

μ_x being the center of C_x.

Obviously, this saves a lot of computation time as for each iteration the distances between each x and μ_i is calculated in any case when using lloyd's algorithm.

4.3.2 Elkan's optimization

Elkan's K-means would adapt very naturally to kernel-methods, but calculating the pairwise center distances is expensive enough to nullify the speedup with

$$d(\mu_i, \mu_j) = \sum_{x,\hat{x} \in C_i} \frac{\kappa(x,\hat{x})}{|C_i|^2} - 2 \sum_{x \in C_i} \sum_{y \in C_j} \frac{\kappa(x,y)}{|C_i| \cdot |C_i|} + \sum_{y,\hat{y} \in C_j} \frac{\kappa(y,\hat{y})}{|C_j|^2} \qquad (2)$$

being the distance-formula for two cluster centers in feature space. For more details, see section 6.3 in measurements.

[7] gives a slightly simplified version of elkan's optimization that results in a speedup on kernel k-means aswell. As the concrete algorithm is not entirely clear from the paper (Appendix A), a slightly deviated version is given here in pseudocode (Algorithm 4) and implemented in `KKMeans`. This works better than Elkan's original algorithm for kernel k-means because the center distances are only calculated for each new to the corresponding old center, and not the pairwise distances between each new center. Nevertheless, it only yields a small

[5] Assuming the full kernel-matrix fits in memory.

[6] Empirically, assigning to random clusters lead to significantly worse results than choosing random existing data points as centers and calculating the assignment.

speedup compared to lloyd's algorithm (see measurements 6.2) and only under the right conditions.

Algorithm 4 Adjusted Elkan's algorithm

Input: X, C, *labels* // Data X, Clusters C, starting labels
Output: labels
1: $\mathrm{L}[x, c] \leftarrow ||\phi(x) - \mu_c||^2 \; \forall x \in X, c \in C$ // μ_c is center of cluster c
2: $D[] \leftarrow zeros(|X|, |C|)$ // D is a $n \times k$ zero-matrix
3: **while** not Converged **do**
4: $\text{labels}_{old} \leftarrow \text{labels}$
5: $\text{labels}[x] \leftarrow min_c(\mathrm{L}[x, c]) \; \forall x \in X$
6: $\mathrm{D}[:, c] \mathrel{+}= ||\mu_c^n - \mu_c^o|| \; \forall c \in C$ // addition to all of column c
7: **for all** $x \in X$ **do**
8: $c_x \leftarrow \text{labels}[x]$
9: $\mathrm{L}[x, c_x] \leftarrow ||\phi(x) - \mu_{c_x}||^2$ // μ_{c_x} could have moved
10: $\mathrm{D}[x, c] \leftarrow 0$
11: **for all** $c \in C$ **do**
12: **if** $\sqrt{\mathrm{L}[x, c]} - \mathrm{D}[x, c] < \sqrt{\mathrm{L}[x, c_x]}$ **then**
13: $\mathrm{L}[x, c] \leftarrow ||\phi(x) - \mu_c^n||^2$
14: $\mathrm{D}[x, c] \leftarrow 0$

5 Implementation

As independence of other libraries was emphasized, the only external libraries that have been used for implementation are NumPy [11] and Cython [6]. PyTest has been used as testing framework and scikit-learn [21] to validate the results aswell as generating test data.

The docstring in each module provides more details than given here on each function/class implemented.

5.1 General Notes

5.1.1 Structure

The project has been implemented as a package `KKMeans` containing the following modules: `KKMeans.kernels`, `KKMeans.quality`, `KKMeans.KKMeans`, `KKMeans.elkan` and `KKMeans.lloyd`. As core module acts `KKMeans.KKMeans`, which provides the full "framework" (the KKMeans class) and uses and depends on the other modules. For better usability, it has been made possible to import the class via "from KKMeans import KKMeans". Also, `KKMeans.KKMeans` is the only module not implemented in Cython, as all the heavy computations have been outsourced. It has been implemented this manner as on one hand to have high cohesion, and on the other hand each module may provide some utility for which not the whole framework is needed.

There is a small amount of duplicate code to be found. This is on purpose, as for example the first iteration of Elkan's and Lloyd's algorithm are exactly the same, yet have been included in both respective modules to ensure they do not break when one of them changes. The computational functions are included in each file that needs them aswell, also to have little coupling between the modules.

5.1.2 Efficiency

Kernel k-means could be cleanly and easily implemented using just NumPy and the indexing method it provides for its arrays, but the indexing (despite the array being stored contiguously in memory) turned out to be unacceptably slow (see measurements 6.1). To counter this, Cython has been utilized. Cython allows for an easy integration of C code to Python [7] and therefore can yield a significant speed up. All frequently used computations that would require iterating through a multidimensional array (/advanced indexing) have been outsourced to Cython. Cython also allowed for parallelization through OpenMP.

5.1.3 Terminology

For class-docstrings, the terminology used in the NumPy documentation regarding "class-parameters" as class-variables that are initialized with the constructor and "class-attributes" as class variables that store information and are updated during the algorithm has been adopted. Class attributes can be recognized by a trailing underscore.

The mathematical functions have been tried to be named as closely to the computations they provide as possible. The most frequently occurring ones are (for example when calculating the distance between a datapoint and an (implicit) cluster-center $d(x, c_i)$, see 1), $\sum_{y,\hat{y}\in C_i} \kappa(y, \hat{y})$ called an `inner_sum` as it computes the sum of the kernels from all points in a cluster, and for a fixed x: $\sum_{y\in C_i} \kappa(x, y)$ called an `outer_sum`.

5.1.4 Input validation

A validation function which checks for adequate datatypes is called in the constructor of the KKMeans class and throughout the algorithm `ndarrays` with set datatypes are used. Also, Cython code is statically typeset and therefore rejects invalid input.

5.1.5 Public Functions

Functions described here adjust their behaviour accordingly to the parameters given to KKMeans instance the functions are called from.

[7] To be technically precise, it allows to write typeset Python code that then gets transpiled to C code which in turn gets compiled to bytecode for the CPython virtual machine.

calc_q_metric($sq\_dists, labels$) Calculates quality metric for given distances and labels.

calc_silhouette($sq\_dists, labels$) Calculates the average silhouette coefficient for the clustering provided by $labels$.

fit(X) Fits dataset X. Runs the given algorithm n_init times and stores the best one.

elkan($kernel\_matrix, labels$) Executes a run of the algorithm described in 4.3.2.

kernel_wrapper($X, Y = None$) Computes the kernel matrix. If Y is None, the matrix is computed pairwise. Otherwise the matrix is computed so that $K[i, j] = \kappa(X[i], Y[j])$

kmeanspp($X, kernel\_matrix$) Chooses cluster centers via the heuristig described in subsection 3.4 and assigns the datapoints accordingly. Returns the labels.

lloyd($kernel\_matrix, labels$) Executes a run of the algorithm described in subsection 4.3.

predict(X) Assigns each datapoint in X the label corresponding to the closest cluster center.

5.1.6 `inner_sums` class-attribute

To predict the cluster to which a given (new) sample x belongs to, four terms are necessary:

1. $\kappa(x, x)$, the kernel of the point and itself
2. $\sum_{y \in C_i} \kappa(x, y)\ \forall i$, the `outer_sums` from x to all clusters
3. $\sum_{y, \hat{y} \in C_i} \kappa(y, \hat{y})$, the `inner_sums` for all clusters
4. $|C_i|$, the cluster sizes

For 1. and 2. it is obvious that the kernel-matrix between a dataset $\mathcal{P}$ which should be predicted and the dataset $\mathcal{X}$ that the KKMeans instance is fit to must be computed. As usually $|\mathcal{P}| \ll |\mathcal{X}|$, this is no issue.

To compute 3. on the other hand, the kernel-matrix from $\mathcal{X}$ to itself is necessary. If $\mathcal{X}$ contains 15.000 samples, the corresponding kernel-matrix already takes up 1.8 Gigabytes of memory, so storing it is not an attractive option. As computing the kernel-matrix is time-intensive (and there may be multiple datasets that have to be predicted), recomputing is also not feasible. For this

reason, the `inner_sums` (and the cluster sizes) of each best run of kernel-kmeans are saved as a class-parameter, as they are computed in any case.

5.1.7 General Notes

When computing squared distances via the kernel trick to see which center is the closest, the kernel of the datapoint with itself $\kappa(x, x)$ could be omitted as it is a constant for each datapoint and therefore does not affect the cluster chosen. In practice, the only time this is used is when assigning to explicitly given centers (e.g. when setting the **init**-parameters to an arraylike containing datapoints). This is because if it would be done in the algorithm itself, inertia would not be computed accurately anymore. As kmeans`++` utilizes probabilities it cannot be omitted either.

5.2 Quality

5.2.1 Public Functions

calc_silhouette(*sq_distances*, *labels***)** Returns the silhouette for each datapoint.

avg_silhouette(*sq_distances*, *labels***)** Returns the average silhouette for a clustering.

5.3 Utils

5.3.1 Public Functions

fill_empty_clusters(*labels*, *n_clusters*, *return_sizes*, *rng*) Computes the sizes for each cluster and, if one is empty, assigns a random element to it. Repeats until no empty clusters are left. Returns the resulting labels and if *return_sizes* is true the cluster sizes as ndarrays.

calc_sq_distances(*inner_sums*, *cluster*$_s$*izes*, *kernel_matrix*, *labels*, *n_clusters* Intended to use for the predict function in KKMeans. Uses the kernel-trick to compute and return the square distances implied by the given parameters.

calc_sizes(*labels*, *n_clusters***)** Computes and returns the size of each cluster.

5.4 Elkan and Lloyd

Both provide an **update** function that calculates the next iteration, Elkan also contains a **start** function (which is functionally the same as **update_lloyd**) to calculate exact squared distances to all centers in the first iteration.

5.5 Cython

5.5.1 Compiler issues

Two issues arose due to compiler differences. One problem was that the msvc compiler translates the long datatype to a 4-byte int, while most other compilers translate it to an 8-byte int. This leads to trouble when explicitely specifying which datatype to use (for example by setting the dtype parameter when creating ndarrays) from the Python-side. Luckily NumPy provides the abstracted int_ datatype that translates to the long version used by the compiler, which is possible as NumPy is relies on compiled code aswell.

The other issue was that openMP, which is utilized through Cython's prange, needs to be compiled with different arguments depending on the compiler used, but which compiler is used cannot be known pre-compilation. A clean solution for this has not been found, the current workaround is to provide both a PiPy distribution which is installed without openMP enabled and a github-repository where the setup file can be modified with the adequate compile-args. Other huge libraries (like sklearn or NumPy) solve this issues by providing precompiled binaries for most commonly used systems.[8] Of course, one could *force* setuptools to use a specific compiler, but that seems too restrictive.

5.6 Distribution

As mentioned before, the implementation can be found both on github [15] - installation (after cloning) via "pip install ." - and on PiPy, installation via "pip install KKMeans".

To enable openMP, the implementation needs to be cloned from github and setting the correct compile-args in setup.py before installing with pip. There are default arguments for the msvc and the gcc compiler.

5.6.1 TestPyPi issues

It was impossible to install the project from TestPyPi (without having all dependencies already installed beforehand) because it does not provide distributions for setuptools and Cython. Therefore errors had to be corrected after uploading to the real PyPi, which led to the (unnecessarily) high version number.

5.7 Tests

5.7.1 Strategy

At first, the computational helper-functions have been tested, as they could be unit-tested easily and are the lowest building-blocks of the framework. The

[8] It might be possible that the author missed something here when reading through the ∼1000 LOC setup implementation used by sklearn and there is a possibility to do this without precompilation. Their instruction to install the development-version of sklearn [14] suggests otherwise though.

reason Cython has been used in the first place was that the aforementioned function's implementation with NumPy's boolean indexing mechanism, albeit being clean and functional, was very slow. Because of that there existed working implementations which they could be tested against. The versions using boolean indexing have been tested on simple, manually verified data.

The functions that build upon them (`_calc_center_dists`, ..) have been primarily tested by using them with a linear kernel, because then there is be no projection and the results can be calculated "by hand" and then compared.

To test the whole algorithm runs, at first `kkmeans.fit` has been tested with a linear kernel against `sklearn.kmeans` with data generated by `sklearn.datasets.make_blobs`, verifying both the computed inertia aswell as the returned labels. These tests have been executed with both random and "true" centers returned by `make_blobs`. As `sklearn.kmeans` has a different method of handling empty clusters than KKMeans, these tests *might* still fail, despite having a huge samples to cluster ratio and being technically correct. To ensure correctness, a "dummy" k-means has been implemented that handles empty clusters in the same manner. It has not been used for all tests as it is very slow. Then, `elkan` has been tested against `lloyd`, utilizing the various kernels and different samples to clusters to dimensions ratios.

The prediction functionality has been tested both in the tests on `lloyd` against `elkan` and in its own testfile, there with the approach of using a linear kernel and verifying the results manually.

To test if data validation works, pytests xfail functionality has been used to see if tests that should fail because of bad input actually fail.

5.7.2 General Notes

As mentioned before, PyTest has been used as testing framework. Each module has its own test folder and for each tested function there is a file containing solely its tests.

6 Measurements

Despite Cython yielding a huge speedoup, KKMeans remains vastly slower than `sklearn.cluster.kmeans`. The measurements here (except when explicitely stated otherwise) have been taken with openMP enabled.

6.1 Cython - boolean indexing

6.1.1 Full run - lloyd

Here, only the execution time of the algorithm is measured. Calculating the kernel-matrix, initializing the labels etc. has been done beforehand. Therefore, the number of features is irrelevant as the computations are only executed through the kernel-matrix.

n_clusters	n_samples	Cython	Boolean
5	3000	0.0043	0.0856
5	5000	0.0108	0.2336
5	10000	0.0398	1.0341
50	3000	0.0378	0.1306
50	5000	0.1088	0.3496
50	10000	0.7411	XXX

Cython achieves between 3 - 20 $\times$ speedup. The boolean indexing method consistently failed in the last test, finishing only once with $\sim$46 seconds. This is because boolean indexing returns a copy of the underlying data, which piles up when using a 10.000 $\times$ 10.000 matrix. It presumably does not fail with 5 clusters as the matrix is indexed for each cluster. The tests have been done seperately, so it is not because the memory was occupied through previous tests.

6.1.2 Inner sums and outer sums

Boolean indexing is only used for the mathematical `inner_sums` and `outer_sums`. Here it is obvious that the major timesave comes from computing the `outer_sums`. All measurements have been done using 10.000 samples

n_clusters	Cython, outer	Boolean, outer	Cython, inner	Boolean, inner
5	0.0351	0.6648	0.0350	0.3
50	0.0350	0.7931	0.0203	0.1815
100	0.0355	0.8071	0.0142	0.1653

Interestingly, the number of clusters does not change the computation time (the small differences occur presumably because the computations have been done on a private machine running Windows, which does not allow for precise resource management). Cython achieves consistently $\sim$20$\times$ for `outer_sums` and $\sim$10$\times$ speedup for `inner_sums`; there seems to be a factor that scales worse when using high n_clusters when compared to the boolean version.

6.1.3 Notes

These calculations have been identified as the measure bottleneck, but the measurements imply some other (albeit, smaller) time-sink as the averaged speedup of $\times$ $\sim$15 is hardly ever achieved when comparing the full runs in the Cython and boolean version.

6.2 Lloyd - Elkan

The results here show that the adapted version of Elkan's is only useful when 1. working with a lot of clusters and 2. working in low dimension. This is because the optimization has some overhead and saves time by saving computing distances to centers. Low dimensions are necessary because through the

course of dimensionality the centers, even when moving relatively small distances, quickly surpass the skipping criteria imposed by the estimated lower bounds. The measurements have been computed on 8.000 samples, taking the average of 20 iterations.

n_clusters	n_features	Elkan	Lloyd
5	5	0.159	0.157
50	5	0.902	0.888
200	5	2.911	3.437
5	50	0.284	0.260
50	50	0.745	0.714
200	50	2.691	2.669

6.3 Real Elkan's kernelized

The problem with kernelizing the real algorithm is that the (implied) distance from each center to each center must be calculated. If there was a way to approximate this, Elkan's algorithm would vastly outclass the current implementation.

When n_clusters is small, vanilla Elkan is still faster, but too heavy penalized when n_clusters grow to be in the final implementation. The actual distance calculations (after computing the pairwise distances) take up about roughly 5% of each iteration.

n_clusters	n_samples	Elkan, vanilla	Lloyd
5	1000	0.018	0.011
5	8000	0.81	1.147
20	1000	0.083	0.022
20	8000	2.218	1.236
50	1000	0.23	0.061
50	8000	6.535	1.374

6.4 OpenMP

To measure the performance boost of using openMP, the average time of 100 runs with set seed, 20 clusters and 5-dimensional data has been taken, once with openMP enabled and once without.

n_samples	openMP	no openMP
5000	0.2528	0.3271
8000	0.7760	0.9038

7 Conclusion and future work

Kernel k-means is a broadly applicable clustering algorithm, that now has with `KKMeans` a publicly available, fast and tested implementation in Python. Still, there are some possibilities to further speed up the algorithm.

As for some kernels explicit feature maps can be computed, the main drawback of the original Elkan's algorithm could be eliminated by computing the centers explicitely. This can even be done for the RBF kernel, for example by using something like a nyström sampler [16].

As the measurements have shown, some of the speedup gained by using Cython is lost when comparing the full runs of Lloyd's algorithm. The result for this discrepany has not been discovered yet and may be the source of some more efficiency gain.

When handling ndarrays, the implementation always uses c-contiguity, meaning the elements are lined up row-wise in memory as contiguos blocks. In the implementation, NumPy broadcasting is always used column-wise, as the distance matrices are of size $n\_samples \times n\_clusters$ and the additions are done clusterwise. There are some indicators that hint on a speedup when the broadcasting format and the storage format are similiar, meaning either f-contiguity for column wise addition or c-contiguity for row-wise addition. This is not implemented as 1. there is no real literature on this and the speedup therefore not guaranteed and 2. it has been noticed very late in development, and the adjustment would be a lot of very tedious work, for which an uncertain (small) speedup is not enough justification.

Aside from that, there are way more initialization methods than just randomness and Kmeans++ [5], which would offer themselves to be implemented and measured.

References

[1] Tilo Arens et al. *Grundwissen Mathematikstudium*. Springer Spektrum Berlin, Heidelberg, 2013. DOI: 10.1007/978-3-8274-2309-2.

[2] David Arthur and Sergei Vassilvitskii. "K-means++: The Advantages of Careful Seeding". In: *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (Jan. 2007), pp. 1027–1035.

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer New York, NY, 2006. ISBN: 978-0-387-31073-2.

[4] Mathieu Blondel. *kernel_kmeans.py*. https://gist.github.com/mblondel/6230787/revisions. 2013.

[5] Emre M. Celebe, Hassan A. Kingravy, and Patricio A. Vela. "A Comparative Study of Efficient Initialization Methods for the K-Means Clustering Algorithm". In: 40.1 (2013), pp. 200–210. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2012.07.021. URL: https://www.sciencedirect.com/science/article/pii/S0957417412008767.

[6] *Cython*. URL: https://cython.org (visited on 07/05/2023).

[7] Inderjit S. Dhillon, Yuqiang Guan, and Brian Kulis. "Kernel k-means, Spectral Clustering and Normalized Cuts". In: *KDD* (Aug. 2004), pp. 22–25.

[8] Xu Dongkuan and Tian Yingjie. "A Comprehensive Survey of Clustering Algorithms". In: *Annals of Data Science* 2 (2015), pp. 165–193. DOI: 10.1007/s40745-015-0040-1.

[9] Charles Elkan. "Using the Triangle Inequality to Accelerate k-Means". In: *Proceedings of the Twentieth International Conference on Machine Learning* (2003), pp. 147–153.

[10] Jiawei Han, Michelle Kimber, and Jian Pei. *Data Mining: Concepts and Techniques*. Elsevier, 2012.

[11] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[12] John A. Hartigan. *Clustering algorithms*. Wiley, 1975.

[13] Eduardo R. Hruschka, Leandro N. de Castro, and Ricardo J. G. B. Campello. "Evolutionary Algorithms for Clustering Gene-Expression Data". In: *Proceedings of the Fourth International Conference on Data Mining (ICDM'04)* (2004), pp. 403–406. DOI: 10.1109/ICDM.2004.10073.

[14] *Installing scikit-learn development build*. URL: https://scikit-learn.org/dev/developers/advanced_installation.html (visited on 07/29/2023).

[15] *KKMeans github page*. URL: https://github.com/bauxn/kernel-kmeans.

[16] Sanjiv Kumar, Mehryar Mohri, and Ameet Talwalkar. "Sampling Techniques for the Nyström Method". In: *AISTATS* (2009).

[17] Hsuan-Tien Lin and Chih-Jen Lin. "A Study on Sigmoid Kernels for SVM and the Training of non-PSD Kernels by SMO-type Methods". In: *Neural Computation* (June 2003).

[18] Stuart P. Lloyd. "Least Squares Quantization in PCM". In: *IEEE TRANSACTIONS ON INFORMATION THEORY* 28.2 (1982). Originally published in 1957 in an internal paper of the bell-labs. DOI: 10.1109/TIT.1982.1056489.

[19] J. Macqueen. "Some Methods for classification and Analysis of Multivariate Observations". In: *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability* 1 (1967), pp. 281–297.

[20] Mahajan Meenam, Nimbhorkar Prajakta, and Varadarajan Kasturi. "The planar k-means problem is NP-hard". In: *Theoretical Computer Science* 442 (2012), pp. 13–21. DOI: 10.1016/j.tcs.2010.05.034.

[21] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[22] Peter J. Roussew and Leonard Kaufman. *Finding groups in Data - An Introduction to Data Analysis*. John Wiley & Sons, 1990. ISBN: 9780470316801. DOI: 10.1002/9780470316801.

[23] John Shawe-Taylor and Nello Christianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004. ISBN: 978-0-521-81397-6. DOI: 10.1017/CBO9780511809682.

[24] *sklearn.cluster.KMeans docpage*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html (visited on 06/28/2023).

[25] *sklearn.metrics.pairwise.pairwise_kernels docpage*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.pairwise_kernels.html (visited on 07/05/2023).

[26] Noam Slonim, Ehud Aharoni, and Coby Crammer. "Hartigan's K-Means Versus Lloyd's K-Means – Is It Time for a Change?" In: *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence* (2013), pp. 1677–1684.

[27] Romain Tavenard et al. "Tslearn, A Machine Learning Toolkit for Time Series Data". In: *Journal of Machine Learning Research* 21.118 (2020), pp. 1–6. URL: http://jmlr.org/papers/v21/20-091.html.

[28] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recognition*. Academic Press, 2008. ISBN: 978-1-59749-272-0.

A Optimization from Inderjit et al

In the paper from Inderjit et al [7], in section 5 there is a description on how Elkan's optimization has been adjusted and that it lead to a massive speedup. The description is not accurate and (probably because of that) the speedup could not be reproduced.

Decsription

"To speed up the distance computation in our weighted kernel k-means algorithm, we can adapt the pruning procedure used in [ELKAN].
...
We compute the distances between corresponding new and old centers, $||\mathbf{m}_j^n - \mathbf{m}_j^o||$ for all j, and store the information in a $k \times k$ matrix D. Similarly, we keep a $k \times n$ matrix L that contains lower bound for the distance from each point to each center. The distance from a point to its cluster center is exact in L. After the centers are updated, we estimate the lower bound from each point a to new cluster center, say $\mathbf{m}_j^n$, to be the difference between the lower bound from $\mathbf{a}$ to $\mathbf{m}_j^o$ and $||\mathbf{m}_j^n - \mathbf{m}_j^o||$. We actually compute the distance from $\mathbf{a}$ to $\mathbf{m}_j^n$ only if the estimation is smaller than distance from $\mathbf{a}$ to its cluster center."

Pseudocode

The resulting pseudocode would then look something like this:

Input: X, C, $labels$ // Data X, Clusters C
1: $L[x, c] \leftarrow ||\phi(x) - m_c|| \; \forall x \in X, c \in C$ // m_c is center of cluster c
2: **while** not converged **do**
3: $labels_{old} \leftarrow labels$
4: $labels[x] \leftarrow min_c(L[x, c]) \; \forall x \in X$
5: $D[c] \leftarrow ||m_c^n - m_c^o|| \; \forall c \in C$ // m_c^n from new labels, m_c^o from old
6: **for all** $x \in X$ **do**
7: $c_x \leftarrow labels[x]$
8: $L[x, c_x] \leftarrow ||\phi(x) - m_{c_x}||$ // m_{c_x} could have moved
9: **for all** $c \in C$ **do**
10: **if** $|L[x, c] - D[c]| < L[x, c_x]$ **then**
11: $L[x, c] \leftarrow ||\phi(x) - m_c^n||$

Issues

From the description, it is confusing why the distances between the new and old centers need to be stored in a $k \times k$ matrix, as there are k centers and it is not done pairwise. So, a k-array would be sufficient. If it is meant to be calculated pairwise, the algorithm would suffer from the same drawback as the original elkan (which the paper says that they have adapted so that seems unlikely).

In the algorithm itself, the center movements are lost at the end of each iteration, which would lead to errors. For example, let x be assigned to a cluster C with center μ which is a distance d apart from x. Let $\hat{C}$ be a cluster whose center $\hat{\mu}$ is a distance $\hat{d}$ apart from x so that $\hat{d} = 3 \cdot d$ Now, when $\hat{c}$ moves each iteration d units closer to x, $L[x, \hat{c}]$ is never being recalculated, despite being as close to x as c after only three iterations .

On a sidenote, in the paper $||\phi(a) - \mathbf{m}_j||^2$ is referred to as the euclidean distance in the feature space between point $\mathbf{a}$ and center $\mathbf{m}_j$. This is not technically correct, as those are squared distances and when for example utilizing the RBF kernel it is debatable if the result is the squared euclidian (as the feature space is infinite-dimensional).

B Definitions and distances in featurespace

The definitions used here (along with more background) can be found in [1].

Inner Product Space

An *inner product* over a real vector space V is defined as a function $\langle \cdot, \cdot \rangle \colon V \times V \to \mathbb{R}$ that satisfies the following properties:

(S1) $\forall x \in V \colon \langle x, x \rangle \geq 0 \wedge (\langle x, x \rangle = 0 \iff x = 0)$

(S2) $\forall x, y \in V \colon \langle x, y \rangle = \langle y, x \rangle$

(S3) $\forall x, y, z \in V \ \forall \lambda, \mu \in \mathbf{R} \colon \langle \lambda x + \mu y, z \rangle = \lambda \langle x, z \rangle + \mu \langle y, z \rangle$

A real vector space equipped with such an inner product is called *inner product space.*

Norm

A *norm* over a real vector Space V is a function $|| \cdot || \colon V \to \mathbb{R}$ that satisfies:

(N1) $\forall x \in V \colon ||x|| = 0 \iff x = 0$

(N2) $\forall x \in V \ \forall \lambda \in \mathbb{R} \colon ||\lambda x|| = |\lambda| \ ||x||$

(N3) $\forall x, y \in V \colon ||x + y|| \leq ||x|| + ||y||$

An inner product induces a norm via $||x|| \coloneqq \sqrt{\langle x, x \rangle}$.

Metric

A *metric* over a set X is a map $d \colon X \times X \to \mathbb{R}$ that satisfies:

(M1) $\forall x, y \in X \colon d(x, y) = 0 \iff x = y$

(M2) $\forall x, y \in X \colon d(x, y) = d(y, x)$

(M3) $\forall x, yz \in X : d(x, z) \leq d(x, y) + d(y, z)$

$d(x, y)$ is called the distance between x and y. A norm induces a metric via $d(x, y) = ||x-y||$ and therefore is also induced by an inner product via $d(x, y) = \sqrt{\langle x - y, x - y \rangle}$.

Hilbert Space

A *Hilbert Space* is an inner product space that is complete with regards to the distance induced by the inner product.[9]

Distance calculation

As the feature map ϕ maps by definition to a hilbert space H and the kernel function κ is a "shortcut" to calculating the inner product in H, the distance from two vectors x, y of the input space $\mathcal{X}$ in the feature space can be calculated via

$$\begin{aligned} d(\phi(x), \phi(y)) &= \sqrt{\langle \phi(x) - \phi(y), \phi(x) - \phi(y) \rangle} \\ &= \sqrt{\langle \phi(x), \phi(x) \rangle - 2\langle \phi(x), \phi(y) \rangle + \langle \phi(y), \phi(y) \rangle} \\ &= \sqrt{\kappa(x, x) - 2\kappa(x, y) + \kappa(y, y)} \end{aligned}$$

[9] There are at least two other definitions for Hilbert Spaces, see [23][28]. [28] states that a finite Hilbert Space is also called an Euclidean Space.