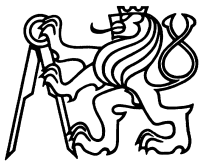
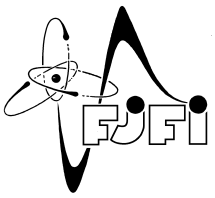


Czech Technical University in Prague
Faculty of Nuclear Sciences and Physical Engineering



Data clustering in Hilbert spaces

Shlukování dat v Hilbertových prostorech

Bachelor's Degree Project

Author: **Diana Varšíková**

Supervisor: **doc. Ing. Jaromír Kukal, Ph.D.**

Academic year: 2022/2023

České vysoké učení technické v Praze, Fakulta jaderná a fyzikálně inženýrská

Katedra: matematiky Akademický rok: 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student:	Diana Varšíková
Studijní program:	Aplikované matematicko-stochastické metody
Název práce (česky):	Shlukování dat v Hilbertových prostorech
Název práce (anglicky):	Data clustering in Hilbert spaces

Pokyny pro vypracování:

1) Seznamte se s principy shlukování v reálných n-dimenzionálních prostorech.
2) Seznamte se s technikami analýzy dat v Hilbertově prostoru.
3) Navrhněte shlukovací algoritmy v Hilbertově prostoru.
4) Implementujte knihovnu umožňující shlukování dat použitím jazyka Python.
5) Proveďte numerické experimenty za použití různých reálných dat.

Doporučená literatura:

1) J. Shawe-Taylor, N. Cristianini, Kernel Methods for Pattern Analysis, Cambridge University Press, 2004.
2) J. Šnor, J. Kukal, Q. V. Tran, SOM in Hilbert Space. Neural Network World, 29(1), 2019, 19-31.
3) R. Hrebik, J. Kukal, J. Jablonsky, Optimal unions of hidden classes. Central European Journal of Operations Research, 27(1), 2019, 161-177.

Jméno a pracoviště vedoucího bakalářské práce:

doc. Ing. Jaromír Kukal, Ph.D.
KM, Fakulta jaderná a fyzikálně inženýrská, ČVUT v Praze, Břehová 78/7, 115 19 Praha 1-Staré Město

Jméno a pracoviště konzultanta:

Datum zadání bakalářské práce: 31.10.2022

Datum odevzdání bakalářské práce: 2.8.2023

Doba platnosti zadání je dva roky od data zadání.

V Praze dne 31.10.2022

.....
garant oboru

.....
vedoucí katedry

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta jaderná a fyzikálně inženýrská
-3-

.....
děkan

Acknowledgment:
I would like to thank my supervisor doc. Ing. Jaromír Kukal, Ph.D. for his invaluable guidance, insightful feedback and humane approach.
I would also like to express my gratitude to my sister for her language assistance.

Author's declaration:
I declare that this Bachelor's Degree Project is entirely my own work and I have listed all the used sources in the bibliography.

Prague, August 2, 2023 Diana Varšíková

Název práce:

Shlukování dat v Hilbertových prostorech

Autor: Diana Varšíková

Obor: Aplikované matematicko-stochastické metody

Druh práce: Bakalářská práce

Vedoucí práce: doc. Ing. Jaromír Kukal, Ph.D.
KM, Fakulta jaderná a fyzikálně inženýrská, ČVUT v Praze, Břehová 78/7, 115 19 Praha 1-Staré Město

Abstrakt: Hledání nelineárních závislostí mezi daty je dlouhodobě studovaným problémem. Za tímto účelem byly vyvinuty jádrové metody, jež převádí problém do Hilbertových prostorů vyšších dimenzí. Tato práce si klade za cíl představit jádrové metody a jejich aplikaci na problém shlukování. Dále práce studuje optimální nastavení parametru pro shlukování s použitím Gaussova jádra. Za tímto účelem bylo zkoumáno 21 souborů dvou-dimenzionálních dat. K řešení problému shlukování se dají využít i heuristické metody, konkrétně metoda náhodného sestupu. Tato metoda je testována na třech různých souborech dat, konkrétně na souborech iris, breast a wine. V rámci její optimalizace byly použity tři různé typy mutací, jež bojují proti problému lokálního minima. Jako součást práce byla vytvořena Python knihovna pro manipulaci s jádry a jádrové shlukování.

Klíčová slova: Gaussovo jádro, heuristika, Hilbertův prostor, jádrové metody, mutace, nelineární závislosti, optimalizace, Python, shlukování

Title:

Data clustering in Hilbert spaces

Author: Diana Varšíková

Abstract: Detecting non-linear patterns among data has been a long-lasting problem. Kernel-based methods have been developed to tackle this issue using Hilbert spaces of higher dimension. This work aims to introduce kernel methods and their application on the clustering problem, all while implementing concise Python library to do so. It further researches the optimal value of the parameter for clustering using the Gaussian kernel while examining 21 two-dimensional datasets. The clustering problem in the feature space can be seen as an optimization problem in $\mathbb{Z}^n$, therefore heuristic methods can be used for its solution. The properties of the random descent method are tested using three widely used datasets - iris, breast and wine. To combat the problem of local minima, mutations originating from the field of genetic algorithms are used. As a part of this work, Python library PyKern for manipulating kernels and performing kernel clustering algorithms was created.

Key words: clustering, Gaussian kernel, heuristics, Hilbert space, kernel, mutation, non-linear relations, optimization, Python

Contents

Introduction

The problem of detecting and characterizing relations between data has been investigated by scientists for decades. Plenty of methods to detect linear relations exist, be it using statistical methods, machine learning or heuristics methods. However, discovering non-linear relations is a much harder task. Kernel-based methods provide an efficient way to tackle this problem, since they are able to identify non-linear relations using the Hilbert spaces of an infinite dimension and the so-called kernel trick. This trick, along with the basic properties of kernels and kernel matrices will be described in the first chapter. One of the biggest advantages of kernel-based methods is their generality, meaning that the input data can have a different form than vectorial. As a result, kernel methods are commonly used with sound or image data. However, this particular property is not a subject of this work, as the input in all of the chapters is assumed to have a vectorial form.

One of the sought-after relations between data is their similarity to each other. The goal is to divide homogenous data into groups with members being more similar to each other than to members of the other groups. This problem is called clustering and it is also one of the most widely used techniques in exploratory data analysis [1]. The formal definition of the clustering problem together with the metrics for evaluating the quality of the solution will be presented in Chapter 2.

The clustering task can be seen as an optimization task in $\mathbb{Z}^n$. One of the algorithms used for this optimization is known as K-Means. The kernel version of this algorithm along with its properties will be described in Chapter 3. Heuristics methods can also be used as an optimization tool for this problem. Chapter 3 further introduces the random descent algorithm as another way to solve this optimization task. Unfortunately, this algorithm suffers from getting trapped at the local minima. To address this issue, mutations originating from the field of genetic algorithms and also described in Chapter 3, can be used. Moreover, their behaviour and influence on the performace of the random descent will be examined in Chapter 5.

To accompany this work, a Python library PyKern was created. It provides the neccessary tools for manipulating kernels, kernel matrices, mutations and is able to perform both K-Means and the random descent optimization for the clustering problem. The implementation details about the library and its structure are described in Chapter 7. Afterwards, this library will be used to perform a series of tests to determine the optimal parameter for one of the most widely used kernels, the Gaussian kernel. The results of these tests and the details can be found in Chapter 4.

Chapter 1

Kernel Methods

This chapter gives an introduction to the main principles of kernel methods and their benefits. Kernel methods are very often used for detecting non-linear relations among data. Each kernel algorithm consists of a given kernel function that transforms input data into a matrix form and of a pattern seeking algorithm that uses this matrix as an input [12]. Thanks to this property, kernel algorithms are very flexible and can work well with different data types, such as vectors, strings, trees etc. All the data that can be transformed into a matrix form by a kernel function can be used as an input. The theory in this work will be built using vectorial input data.

Definition 1 (Pattern set)**.** *Let a pattern be defined as any member of the input space* $\mathbf{x} \in \mathbb{R}^p$*, where* p *is the number of features. Consider a set of* n *patterns denoted by* $\mathcal{S} = \{\mathbf{x}_k \in \mathbb{R}^p : k = 1, \ldots, n\}$*. This set is called a pattern set.*

Definition 2 (Hilbert space)**.** *A Hilbert space* $\mathcal{H}$ *is an inner product space with the additional properties of being separable and complete. Completeness means that every Cauchy sequence in* $\mathcal{H}$ *converges in* $\mathcal{H}$ *and separability means that* $\mathcal{H}$ *contains a countable, dense subset.*

Definition 3 (Kernel function)**.** *Given a non-linear mapping* φ *from the input space* $\mathbb{R}^p$ *to the feature space* $\mathcal{H}$*, i.e* $\varphi : \mathbb{R}^p \to \mathcal{H}$*, a kernel is a function* κ *defined for all* $\mathbf{x}, \mathbf{z} \in \mathbb{R}^p$ *as* $\kappa(\mathbf{x}, \mathbf{z}) = \langle \varphi(\mathbf{x}), \varphi(\mathbf{z}) \rangle \in \mathbb{R}$ *[12].*

The feature space $\mathcal{H}$ is a real Hilbert space. Kernel functions are usually created in a way to represent similarity between data. They transform the finite-dimensional input data into a higher dimensional space, possibly with infinite dimensions. Feature spaces with infinite dimensions are often used. In the case of the infinite dimensions, and usually in kernel algorithms, we cannot calculate the exact coordinates in this new feature space. However, it is still possible to calculate the between-point distances and many useful properties, thanks to the so called 'kernel trick'(more information provided in the following section). Thanks to this transformation, kernel algorithms are able to detect even non-linear relations, as it should always be possible to obtain linearly separable groups in higher dimensions.

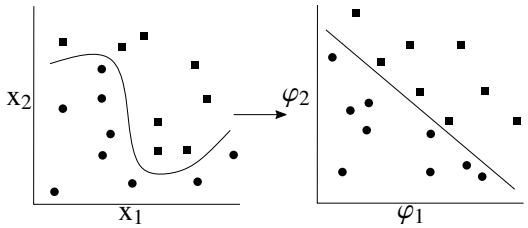


Figure 1.1: Linear separability in $\mathcal{H}$

Definition 4 (Kernel Matrix)**.** *Given the pattern set* $\mathcal{S} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$*, the associated kernel matrix is defined as* $\mathbf{K} = \left(\kappa(\mathbf{x}_i, \mathbf{x}_j)\right)_{i,j=1}^{n} \in \mathbb{R}^{n \times n}$

The matrix K is the Gram matrix with a kernel function κ used to evaluate the inner product, therefore K is symmetric, positive semidefinite and $\mathbf{K}^{\mathrm{T}} = \mathbf{K}$. This kernel matrix acts as an input data type in all kernel-based algorithms. The number of useful properties of the input dataset in a kernel-defined feature space can be obtained from it.

1.1 Properties in feature space

As mentioned earlier, due to the infinity of the feature space $\mathcal{H}$, the exact coordinates of objects in this space cannot be obtained. However, much useful information, such as the norm of a vector in the feature space and the distances between vectors can be computed just with the knowledge of κ or $\mathbf{K}$. The norm of feature vectors can be computed as

$$\|\varphi(\mathbf{x})\|_2 = \sqrt{\|\varphi(\mathbf{x})\|^2} = \sqrt{\langle \varphi(\mathbf{x}), \varphi(\mathbf{x}) \rangle} = \sqrt{\kappa(\mathbf{x}, \mathbf{x})}, \tag{1.1}$$

and the distance between two vectors $\mathbf{x}$ and $\mathbf{z}$ in the feature space is

$$\begin{aligned} &\|\varphi(\mathbf{x}) - \varphi(\mathbf{z})\|^2 = \langle \varphi(\mathbf{x}) - \varphi(\mathbf{z}), \varphi(\mathbf{x}) - \varphi(\mathbf{z}) \rangle = \\ &= \langle \varphi(\mathbf{x}), \varphi(\mathbf{x}) \rangle - 2 \langle \varphi(\mathbf{x}), \varphi(\mathbf{z}) \rangle + \langle \varphi(\mathbf{z}), \varphi(\mathbf{z}) \rangle = \\ &= \kappa(\mathbf{x}, \mathbf{x}) - 2\kappa(\mathbf{x}, \mathbf{z}) + \kappa(\mathbf{z}, \mathbf{z}). \end{aligned} \tag{1.2}$$

This property is very important for all kernel clustering algorithms, as it allows to measure distances between objects in the feature space implicitly, without the knowledge of their coordinates. It is usually referred to as the 'kernel trick'. Given the pattern set $\mathcal{S}$ and its associated kernel matrix $\mathbf{K}$, these properties can also be rewritten using this matrix as

$$\|\varphi(\mathbf{x}_i)\|_2 = \sqrt{\kappa(\mathbf{x}_i, \mathbf{x}_i)} = \sqrt{k_{ii}}, \tag{1.3}$$

$$\|\varphi(\mathbf{x}_i) - \varphi(\mathbf{x}_j)\|^2 = \kappa(\mathbf{x}_i, \mathbf{x}_i) - 2\kappa(\mathbf{x}_i, \mathbf{x}_j) + \kappa(\mathbf{x}_j, \mathbf{x}_j) = k_{ii} - 2k_{ij} + k_{jj}. \tag{1.4}$$

1.2 Polynomial kernel functions

One of the standard kernel functions is a polynomial kernel function [4]. The polynomial kernel of the degree d is defined for every $\mathbf{x}$ and $\mathbf{z}$ from the input space $\mathbb{R}^p$ as

$$\kappa(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^{\mathrm{T}}\mathbf{z} + c)^d. \tag{1.5}$$

Both the degree d and the offset $c \geq 0$ are user-defined parameters. If $c = 0$, the kernel is called homogenous. The degree d controls the flexibility of the kernel. Low-degree mappings are usually used, so d is often equal to 2 or 3. The dimension of the feature space $\mathcal{H}$ for polynomial kernel is [4]

$$d_{\mathrm{H}} = \binom{n + d}{d}.$$

1.3 Gaussian kernel functions

Perhaps the most popular kernel functions that can be found in all literature([12], [2], [3]) and are widely used are the Gaussian kernels defined as

$$\kappa(\mathbf{x}, \mathbf{z}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{z}\|^2}{2\sigma^2}\right). \tag{1.6}$$

The width parameter $\sigma > 0$ controls the flexibility of the kernel in the same way as d controls the flexibility of the polynomial kernel. For too small σ, the kernel matrix becomes very close to the identity matrix. Otherwise, when the σ value is too large, the kernel function is reduced to constant. Note that $\kappa(\mathbf{x}, \mathbf{x}) = 1$, therefore the norm of all vectors in $\mathcal{H}$ is equal to one. It can also be seen that the distance between two points using the Gaussian kernel is never higher than $\sqrt{2}$. All kernel algorithms using this kernel are very dependent on the choice of the σ parameter. This kernel is also known as the radial basis function (RBF) kernel. The corresponding Hilbert space $\mathcal{H}$ has the dimension $d_{\mathrm{H}} = +\infty$ [12].

Chapter 2

Clustering Problem

2.1 Clustering problem

Definition 5 (Labelled pattern set)**.** *Let $\mathcal{S}$ be a pattern set with n patterns. Let the label vector be $\mathbf{y} \in \mathbb{N}_0^n$. A labelled pattern set is the pair $(\mathcal{S}, \mathbf{y})$. A pattern set without $\mathbf{y}$ is called an unlabelled pattern set.*

Given the unlabelled pattern set $\mathcal{S} = \{\mathbf{x}_1, \mathbf{x}_2, ...\mathbf{x}_n\}$ and a given number of clusters $k \in \mathbb{Z}, k \geq 2$, the ideal partition of data into dissimilar groups (clusters) $\{C_1, \ldots, C_k\}$ is sought. Sometimes, k is not specified beforehand, in which case the optimal k must first be determined. The partition of the data can be represented by the partition vector $\mathbf{p} = \{1, \ldots, k\}^n$, where $\mathbf{p}_j$ is the cluster index of $\mathbf{x}_j$. For measuring the quality of the clustering, the total heterogeneity of a given partition is calculated as the sum of heterogeneity in each class

$$J = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ p_j=i}}^{n} \|\mathbf{x}_j - \boldsymbol{\mu}_i\|^2, \tag{2.1}$$

where $\boldsymbol{\mu}_i$ is the centre of mass (centroid) of the i-th class. The goal of clustering algorithms is to minimize this criterion. In this work, the elements of the real label vector $\mathbf{y}$ will be called classes and the groups obtained from clustering algorithms will be called clusters or hidden clusters.

2.2 Comparing clustering quality

Many different metrics exist for comparing clustering quality. Throughout this work, three main metrics will be used. Those are contingency tables, accuracy and Adjusted Rand Index.

Definition 6 (Contingency table)**.** *Let n be the number of patterns, N the number of output classes and $H \geq N$ the number of hidden clusters obtained from a clustering algorithm. The matrix $\mathbf{C} \in \mathbb{N}_0^{N \times H}$ is called the contingency table (confusion matrix) when*

$$c_{ij} = \sum_{k=1}^{n} \mathrm{I}(\mathbf{x}_k \in C_i, \mathbf{x}_k \in \mathcal{H}_j), \tag{2.2}$$

where I *denotes the indicator function.*

class	$\mathcal{H}_1$	$\mathcal{H}_2$	...	$\mathcal{H}_H$	sum
C_1	c_{11}	c_{12}	...	c_{1H}	$c_{1\bullet}$
C_2	c_{21}	c_{22}		c_{2H}	$c_{2\bullet}$
.	.	.	.	.	.
.	.	.	.	.	.
C_N	c_{N1}	c_{N2}	...	c_{NH}	$c_{N\bullet}$
sum	$c_{\bullet 1}$	$c_{\bullet 2}$	...	$c_{\bullet H}$	n

Table 2.1: Contingency table

Definition 7 (Reduced contingency table)**.** *Reduced contingency table is a contingency table* $\mathbf{T} \in \mathbb{N}_0^{N\times N}$*, which satisfies* $t_{ii} \geq t_{ij}$ *for every* $i, j \in \{0, \ldots, N\}$

Definition 8 (Accuracy)**.** *Let* $\mathbf{T} \in \mathbb{N}_0^{N\times N}$ *be a reduced contingency table and n the sum of all its elements. The accuracy is defined as*

$$acc = \frac{1}{n}\sum_{i=1}^{N} t_{ii}. \tag{2.3}$$

Any contingency table $\mathbf{C} \in \mathbb{N}_0^{N\times H}$ can be converted to $\mathbf{T} \in \mathbb{N}_0^{N\times N}$ using the relationship between the output classes and the hidden clusters which maximizes the accuracy. The hidden cluster $\mathcal{H}_j$ belongs to the output class C_i if $c_{ij} \geq c_{kj}$ for all $k = 1, \ldots, N$. The union of all the hidden clusters belonging to C_i then forms the final cluster $\mathcal{F}_i$. The relationship between the output classes and final clusters then forms the reduced table $\mathbf{T}$ as

$$t_{ij} = \sum_{k=1}^{m} \mathrm{I}(\mathbf{x}_k \in C_i, \mathbf{x}_k \in \mathcal{F}_j). \tag{2.4}$$

The accuracy is maximized when the row maximas c_{ij} are unique. This process is illustrated in Figure 2.1.

	$\mathcal{H}_1$	$\mathcal{H}_2$	$\mathcal{H}_3$	$\mathcal{H}_4$
C_1	1	**100**	20	5
C_2	**10**	3	**13**	10
C_3	1	10	8	**11**

→

	$\mathcal{H}_2$	$\mathcal{H}_1 \cup \mathcal{H}_3$	$\mathcal{H}_4$
C_1	100	21	5
C_2	3	23	10
C_3	10	9	11

→

	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_3$
C_1	100	21	5
C_2	3	23	10
C_3	10	9	11

Figure 2.1: The transformation of a contingency table into a reduced contingency table

This process can be used even for $N = H$, where it serves to permute rows so that the accuracy is maximized. It allows accuracy to be used not only for classification, but also for clustering problems.

Another clustering evaluation metric, Rand Index, was defined in [9]. It is based on combinatorial approach, as it examines the number of pairs of patterns that are clustered similarly in the output classes and calculated clusters. The Rand index can be interpreted as the probability of a pair of points being clustered similarly (together or separately) in two clusterings C and $\mathcal{D}$. The Rand index lies between 0 and 1. When the two clusterings agree perfectly, the value is 1 [10].

Definition 9 (Rand Index)**.** *Let* C *and* $\mathcal{D}$ *be two different clusterings of the same pattern set. The Rand index is defined as*

$$RI = \frac{S_{11} + S_{00}}{S_{11} + S_{00} + S_{01} + S_{01}} = \frac{S_{11} + S_{00}}{\binom{n}{2}}, \tag{2.5}$$

where S_{11} is the number of pairs clustered together in both C and $\mathcal{D}$, S_{00} is the number of pairs clustered separately in C and $\mathcal{D}$, S_{01} is the number of pairs clustered together in C, but not in $\mathcal{D}$ and S_{01} is the number of pairs clustered together in $\mathcal{D}$, but not in C.

The main problem of the Rand index is the fact that its expected value for two random clusterings does not take a constant value (for example zero). The adjusted Rand index, that was published in [7], fixes this issue.

Definition 10 (Adjusted Rand Index)**.** *The general form of the Adjusted Rand Index is*

$$ARI = \frac{RI - \mathrm{E}[RI]}{\max(RI) - \mathrm{E}[RI]} \tag{2.6}$$

Its expected value for two random clusterings is equal to zero. After calculating $\mathrm{E}[RI]$ and $\max(RI)$ ([7][14]), the exact calculation for obtaining ARI using the contingency table is

$$ARI = \frac{\sum_{i,j}\binom{c_{ij}}{2} - \left[\sum_i \binom{c_{i\bullet}}{2}\sum_j \binom{c_{\bullet j}}{2}\right]/\binom{n}{2}}{\frac{1}{2}\left[\sum_i \binom{c_{i\bullet}}{2} + \sum_j \binom{c_{\bullet j}}{2}\right] - \sum_i \binom{c_{i\bullet}}{2}\sum_j \binom{c_{\bullet j}}{2}/\binom{n}{2}}. \tag{2.7}$$

Chapter 3

Clustering as Optimization Task

The clustering problem can be interpreted as an optimization problem with the goal of minimizing J in (2.1). Many heuristic approaches can be used to achieve this. The two approaches that will be discussed in this chapter are random descent with mutations and a greedy algorithm called K-Means.

3.1 Clustering problem in the feature space

In the feature space, (2.1) can be written in the form

$$J_{\mathrm{f}} = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ p_j=i}}^{n} \|\varphi(\mathbf{x}_k) - \varphi(\boldsymbol{\mu}_i)\|^2 \tag{3.1}$$

Both $\varphi(\mathbf{x}_k)$ and $\varphi(\boldsymbol{\mu}_i)$ have infinite length, so they cannot be used in computations directly. Moreover, the input variable of $\varphi(\boldsymbol{\mu}_i)$, $\boldsymbol{\mu}_i$, does not even have to exist in the original space. Given the pattern set $\mathcal{S}$ with n elements, its centroid (mass) in the feature space is defined as

$$\boldsymbol{\mu}_{\mathrm{s}} = \frac{1}{n} \sum_{i=1}^{n} \varphi(\mathbf{x}_i) \tag{3.2}$$

Its norm can be computed using kernels in the original space[12] as

$$\|\boldsymbol{\mu}_{\mathrm{s}}\|_2^2 = \left\langle \frac{1}{n} \sum_{i=1}^{n} \varphi(\mathbf{x}_i), \frac{1}{n} \sum_{j=1}^{n} \varphi(\mathbf{x}_j) \right\rangle = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \left\langle \varphi(\mathbf{x}_i), \varphi(\mathbf{x}_j) \right\rangle = \frac{1}{n^2} \sum_{i,j=1}^{n} \kappa(\mathbf{x}_i, \mathbf{x}_j) \tag{3.3}$$

with the result being equal to the average of all kernel matrix elements.
Using equations (3.2) and (1.1), the distance of any vector from a given centroid can be expressed as

$$\begin{aligned} &\|\varphi(\mathbf{x}_l) - \varphi(\boldsymbol{\mu}_i)\|^2 = \\ &= \langle \varphi(\mathbf{x}_l), \varphi(\mathbf{x}_l) \rangle + \langle \varphi(\boldsymbol{\mu}_i), \varphi(\boldsymbol{\mu}_i) \rangle - 2 \langle \varphi(\mathbf{x}_l), \varphi(\boldsymbol{\mu}_i) \rangle = \\ &= \kappa(\mathbf{x}_l, \mathbf{x}_l) + \frac{1}{n^2} \sum_{i,j=1}^{n} \kappa(\mathbf{x}_i, \mathbf{x}_j) - \frac{2}{n} \sum_{i=1}^{n} \kappa(\mathbf{x}_l, \mathbf{x}_i) = \\ &= k_{ll} + \frac{1}{n^2} \sum_{i,j=1}^{n} k_{ij} - \frac{2}{n} \sum_{i=1}^{n} k_{li}. \end{aligned} \tag{3.4}$$

It can be seen that all the necessary information to compute J_{f} can be obtained using only the kernel matrix. This is why the kernel matrix plays an important role as the central data structure in the implementation of kernel algorithms.

3.2 Kernel K-Means

K-Means is a widely used method for clustering data [12]. It can be used on pattern sets in both the original and the feature space. It is a simple algorithm with the idea to minimize the sum of distances between vectors and their associated cluster centroids. In the beginning, partition vector $\mathbf{p} \in \mathbb{N}^n$ is initialized randomly. Afterwards, two steps are repeated until convergence - representation step and allocation step [3]. In the representation step, the distances of every pattern to all cluster centroids are computed. The K-Means in original space directly computes the coordinates of all centroids and then computes the distances. In the feature space, the kernel trick (1.2) is used and the distances are obtained directly using (3.4). In the allocation step, the partition vector is updated so that each pattern belongs to the its closest centroid from previous step:

$$\mathbf{p}_i^{\mathrm{new}} = \underset{j \in 1,\ldots,k}{\operatorname{argmin}} \|\varphi(\mathbf{x}_i) - \varphi(\boldsymbol{\mu}_j)\| \tag{3.5}$$

The algorithm ends when the partition vector no longer changes. A convergence to global minima is not guaranteed [12]. As it usually takes only a few iterations to converge, the common practice is to run the clustering multiple times and the best result is then chosen by comparing the respective values of J_{f}.

3.3 Optimization in $\mathbb{Z}^n$

The problem of assigning a cluster to each pattern from a pattern vector of length n can be interpreted as an optimization problem in $\mathbb{Z}^n$. In general, the search space of the optimization problem in $\mathbb{Z}^n$ is defined as $\mathcal{D} = \{\mathbf{x} \in \mathbb{Z}^n : \mathbf{a} \leq \mathbf{x} \leq \mathbf{b}\}$. In this case $a_i = 1$ and $b_i = H$ for each $i \in \{1, \ldots, n\}$, where H is the desired number of clusters. The criterion (3.1) can be looked upon as function $f : \mathcal{D} \to \mathbb{R}$. The goal is to find $\mathbf{x} \in \mathcal{D}$, so that $f(\mathbf{x}) = \min_{\mathbf{x} \in \mathcal{D}} f(\mathbf{x})$.

Definition 11 (Ring neighbourhood in $\mathcal{D}$)**.** *Ring neighbourhood of point* $\mathbf{x} \in \mathcal{D}$ *is defined as*

$$\mathcal{R}(\mathbf{x}, r, p) = \{\mathbf{y} \in \mathcal{D} : 0 < \|\mathbf{x} - \mathbf{y}\|_p \leq r\},$$

where r is the neighbourhood size and p signifies the type of the norm used.

3.4 Random Descent Heuristics

Random descent is a heuristic method for minimizing (3.1) that works iteratively. It starts with an initial solution and then tries to improve it by making random changes. For each solution, it evaluates (3.1) and if its value is lower than before, this solution is considered the new best solution. This heuristic stops when no further improvement can be made.

Its implementation in Python can look like:

```
def repeated_random_descent(max_iter, n)
#max_iter...maximum iterations of RD, n...length of solution
    x_best = np.random.randint(0, max_value, n) #first random vector
    f_best = objective_function(x_best)
    for _ in range(max_iter):
            x_new = x_best.copy()
            new_solution(x_new) # any function which generates different x
            f_new = objective_function(x_new)
            if f_new < f_best: #compare new value against old
                x_best = x_new #set new x_best and f_best
                f_best = f_new
            #else generate different x
    return x_best, f_best
```

The main problem of this method is that it often gets trapped at the local minima. To limit this, certain methods, known as mutations, are used for generating the new solutions.

Algorithm: Random Descent

$\mathbf{x}_0 \sim \mathrm{U}(\mathcal{D}), k = 0$;
while *neccessary* **do**
  $\mathbf{x}_{\text{trial}} = MUTATION(\mathbf{x})$;
  if $f(\mathbf{x}_{\text{trial}}) < f(\mathbf{x}_k)$ **then**
    $\mathbf{x}_{k+1} = \mathbf{x}_{\text{trial}}$;
  else
    $\mathbf{x}_{k+1} = \mathbf{x}_k$;
  end
end

3.5 Mutation Operator

Mutation operators are often used in the field of genetic algorithms. Mutations are random changes that are made to a solution to generate a new solution. They help the algorithms tackle the problem of the local minima and also in the exploration of the whole search space. Mutations can take many forms, such as swapping two elements, moving all elements or permutating a part of the vector[8]. The three mutations used in this work are the Hamming mutation, the wild mutation and the Pareto mutation.

3.5.1 Hamming mutation

The Hamming mutation is a mutation operator that is commonly used in problems involving binary strings. It has one parameter n_{mut}, which represents the number of positions of the vector which values should be changed. The new solution is generated as

$$\mathbf{x}_{\text{trial}} \sim \mathrm{U}(\mathcal{R}(\mathbf{x}_k, r, \mathrm{H})), \tag{3.6}$$

where U is the uniform distribution and

$$\|\mathbf{x} - \mathbf{y}\|_{\mathrm{H}} = \sum_{i=1}^{n} \mathrm{I}(x_i \neq y_i) \tag{3.7}$$

is the number of positions where vectors **x** and **y** differ, also known as the Hamming distance.
Example Python code:

```
def mut_hamming(n_mut, x): # nmut...number of changes to be made
    n = len(x)
    for _ in range(0, nmut):
        i = random.randint(0, n-1)  # random index to mutate
        old_value = x[i]
        new_value = old_value
        while new_value == old_value:
            # choose a new value that is different from the old value
            new_value = (old_value + random.randint(0, b)) % b
        x[i] = new_value # modify element on the chosen index
```

3.5.2 Wild mutation

Another type of mutation is the wild mutation. It has a certain probability p_{wild} of mutating the vector **x** wildly, that is shuffling all its elements randomly. In others cases, it just changes one element by substracting or adding 1 to it. That process itself is called the r-star mutation. New solution is generated as

$$\mathbf{x}_{\text{trial}} \begin{cases} \sim \mathrm{U}(\mathcal{R}(\mathbf{x}_k, 1, 1)) & \text{with probability } 1 - p_{\text{wild}} \\ \sim \mathrm{U}(\mathcal{D}) & \text{with probability } p_{\text{wild}} \end{cases} \tag{3.8}$$

The common choice is to set $p_{\text{wild}} = \mathrm{c}$ for some constant c.
The example of wild mutation in Python:

```
def mutate(pwild, x): #pwild...probability of wild mutation
    if random.uniform(0,1)<pwild: #generate random number
        #mutate wildly
        x = np.random.randint(a, b, len(x))
    else:
        #modify one element of x by adding/substracting
        x = mut_rstar(x, a, b)
```

3.5.3 Pareto mutation

The central principle of the Pareto mutation is to modify a candidate solution by adding a random vector to it. The Pareto mutation takes two parameters, temperature $T > 0$ and shape $\alpha \in (0, 2)$ from the Pareto distribution.

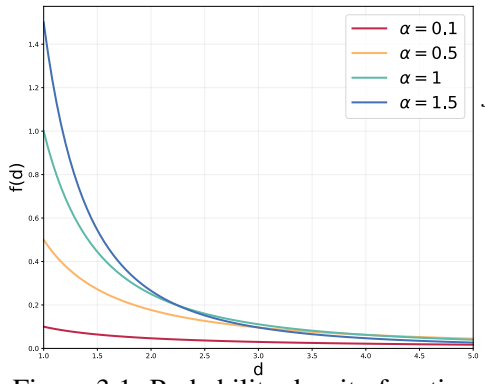


Figure 3.1: Probability density function of Pareto distribution

Definition 12 (Pareto distibution)**.** *Pareto distribution with shape parameter α can be defined by its cumulative density function*

$$\mathrm{F}(d) = \begin{cases} 1 - d^{-\alpha} & \text{for } d \geq 1 \\ 0 & \text{for } d < 1. \end{cases} \tag{3.9}$$

or by its probability density function as

$$\mathrm{f}(d) = \begin{cases} \alpha d^{-(\alpha+1)} & \text{for } d \geq 1 \\ 0 & \text{for } d < 1. \end{cases} \tag{3.10}$$

The random variable from Pareto distribution is generated using a formula:

$$1 - \mathrm{F}(d) = rnd, \text{where } rnd \sim \mathrm{U}([0,1]), \tag{3.11}$$

meaning that $d = rnd^{-1/\alpha} - 1$.

Algorithm: Obtaining new solution using Pareto mutation with α and T

Generate $rnd \sim \mathrm{U}([0,1])$;
$d = rnd^{-1/\alpha}$ -1;
Generate $\boldsymbol{\eta} \sim \mathrm{N}(\mathbf{0}, \mathbb{I})$;
Get random direction $\boldsymbol{\xi} \in \mathbb{R}^n$, $\|\boldsymbol{\xi}\| = 1$ as $\boldsymbol{\xi} = \boldsymbol{\eta}/\|\boldsymbol{\eta}\|$;
$\mathbf{y} = \mathbf{x} + Td\boldsymbol{\xi}$;
$\mathbf{x}_{\text{trial}} = \mathcal{P}(\mathbf{y})$, where $\mathcal{P}$ is a perturbation function.

Example implementation in Python can look like this:

```
def mut_pareto(x, t_mut, alpha): #t_mut mutation step size
    #alpha controls the shape of Pareto distribution
    n = len(x)
    #random vector of length n from N(0,1) distribution:
    eta = np.random.normal(0,1,n)
    #random number from U(0,1) raised to -1/alpha
    delta = random.uniform(0,1)^(-1/alpha)
    #create random vector xi from Pareto distribution
    xi = eta/np.norm(eta)*delta
    #create new x by adding t_mut*xi, add 1/2 for good rounding
    xtrial = np.floor(x + t_mut*xi + 1/2)
    #perturbate the solution
    xnew = perturbation(xtrial)
    if xnew == x: #if the vector didnt change
    #randomly add or substract 1 to one of its elements
        xnew = mut_rstar(x)
    return xnew
```

Chapter 4

Optimal Width Parameter in Gaussian Kernel

This chapter aims to answer the question of the optimal choice of a width parameter for clustering using the Gaussian kernel. All the clustering results using this kernel are highly dependent on the choice of the width parameter σ. Currently, no method for determining its optimal value exists. Because of this, the optimal σ value usually has to be obtained empirically. In this chapter, the optimal σ will be obtained empirically for 21 aritificial pattern sets and compared to σ_{ref} obtained by method from [2].

4.1 Dataset Description

For the purpose of this chapter, a group of 21 pattern sets, called *Clustering excercises*, was used. It is a public, artificially created dataset and can be found online. Popular python library scikit-learn uses this type of pattern sets for comparing their different clustering algorithms [11]. Each of these pattern sets contains 2D vectorial patterns with various lengths. Only the first thousand patterns of each set were used, as more patterns did not prove to have a significant effect on the results and just raised the computational complexity. All of these pattern sets were standardized to zero expected value and standard deviation equal to one. The number of classes ranged from two to five. The original *Clustering excercises* contains 30 different pattern sets. Homogenous pattern sets, pattern sets with only one class and pattern sets with more than ten classes were omitted.

4.2 Optimization of Width Parameter

All the clusterings were performed using kernel K-Means from Section 3.2 with the Gaussian kernel and a known number of classes. All the tests were run using the Python kernel library PyKern, which was created as a part of this work. Because all of the pattern sets were standardized, the optimal σ was expected to have a lower value, close to one. All the clusterings were tested for 15 different values of σ, ranging from 0.1 to 2.512. with higher resolution in lower values. K-Means was run for each σ value 25 times and the best result was selected. The best parameter σ_{opt} was then chosen by comparing the respective values of the Adjusted Rand Index of all the different σ values.

Based on experimental evidence, ref. [2] suggests the use of values between 0.1 and 0.9 quantiles of $\|\mathbf{x}_l - \mathbf{x}_k\|^2, k \neq l$ as the optimal value of the width parameter in the Gaussian kernel. The mean of 0.1 and 0.9 quantiles was used as σ^2_{ref} and in the end, the results obtained using σ_{ref} and σ_{opt} were compared.

This work had two main goals. The first one was to find out how close on average the referential sigma parameter σ_{ref} would be to the best sigma value σ_{opt} found in the experiments. Another investigated aspect was whether any global trends could be identified for all the pattern sets, or if the choice of the optimal width parameter is pattern set dependent.

4.3 Results

The dataset can be visually divided into three smaller datasets, based on the clustering difficulty. Pattern sets that are linearly separable in two dimensions can be considered easy and those that cannot can be further divided into medium and hard, depending on the complexity of their clusters:

Easy - basic1, basic2, basic3, basic4, basic5, lines, network, sparse, triangle
Medium - chrome, dart, spiral2, network, outliers, spirals, lines2
Hard - dart2, face, isolation, un, un2, wave

The overall results for all pattern sets can be seen in the Appendix section. The results obtained from Figure 4.1 show that the best average Adjusted Rand Index (ARI) score of 0.81 was obtained for a value of $\sigma = 0.501$. The best σ values were 0.501, 0.631 and 0.398, for easy, medium and hard pattern sets, respectively. Additionally, Figure 4.2 shows that the highest ARI scores were achieved for the easy dataset and the lowest for the hard one, as expected.

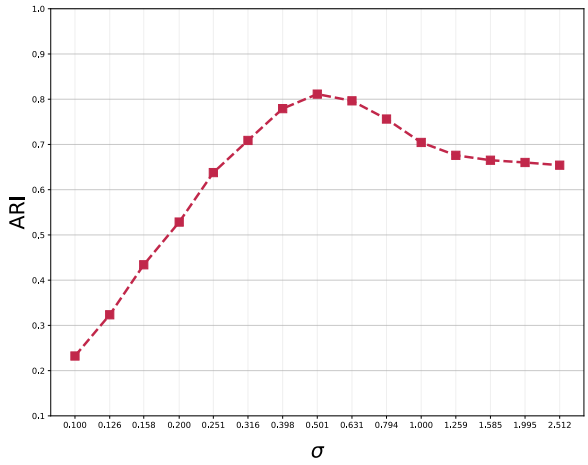


Figure 4.1: Average ARI for all pattern sets

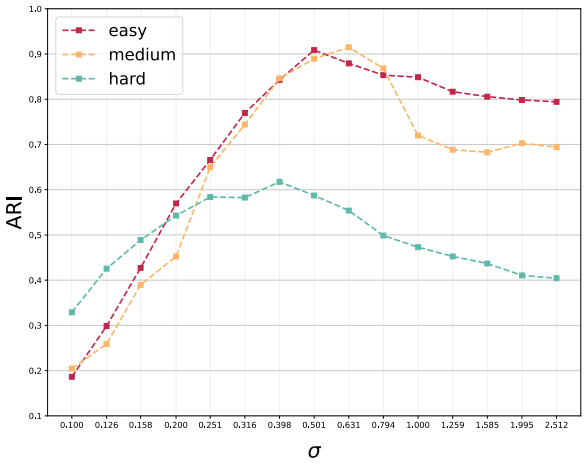


Figure 4.2: Average ARI by datasets

Both easy and medium datasets achieved very good clustering results. The development of ARI score for all the easy and medium pattern sets can be seen in Figures 4.3 and 4.4. It can be seen that ARI values did not change as much for different values of σ. For the easy dataset, the results suggest that the optimization of the width parameter is not that important, perhaps because the pattern sets are linearly separable in the original space. However, the optimization still managed to improve the results. The less linear the clusters are, the more tuning of σ is necessary. For example the *dart* pattern set showed a great jump from ARI equal to zero to ARI equal to one for a well-chosen σ. It shows that kernel K-means has no problem with clustering linearly separable pattern sets, but that it can also generalize and find even non-linear patterns when the width parameter is chosen carefully. The best resulting clusterings of the easy and medium pattern sets are shown in Figures 4.5 and 4.6. The average ARI of those clusterings is 0.91 for the easy dataset and 0.98 for the medium one.

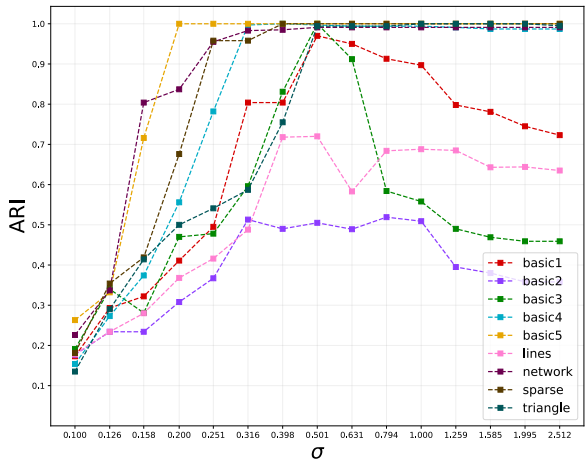


Figure 4.3: ARI scores for easy pattern sets

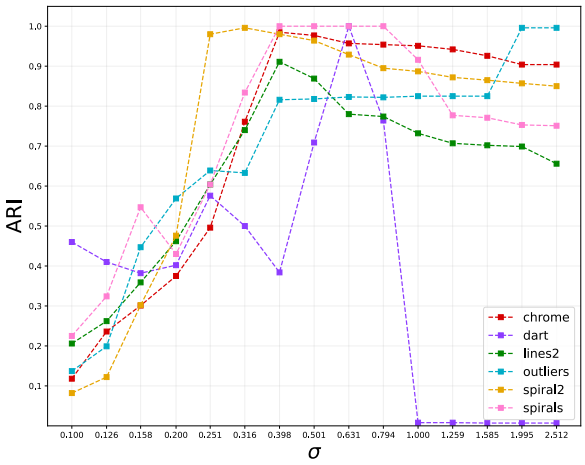
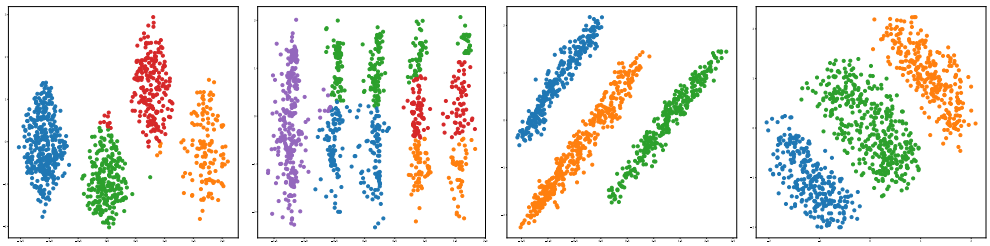
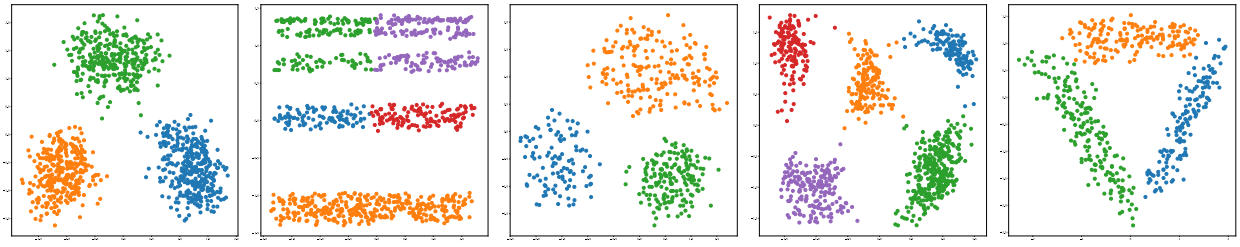


Figure 4.4: ARI scores for medium pattern sets

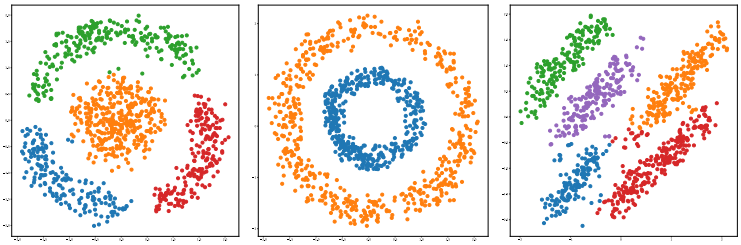


(a) basic1(0.970) (b) basic2(0.519) (c) basic3(1.000) (d) basic4(1.000)

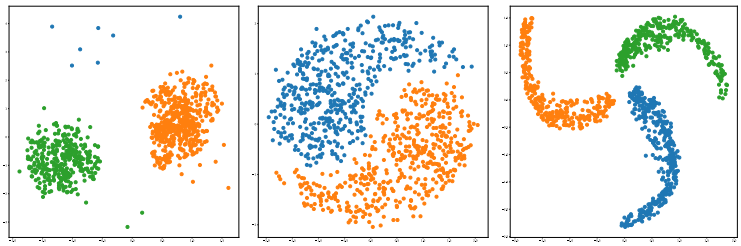


(e) basic5(1.000) (f) lines(0.720) (g) sparse(1.000) (h) network(0.991) (i) triangle(1.000)

Figure 4.5: Best results for easy pattern sets with given ARI score



(a) chrome(0.985) (b) dart(1.000) (c) lines2(0.911)



(d) outliers(0.996) (e) spiral2(0.996) (f) spirals(1.000)

Figure 4.6: Best results for medium pattern sets with given ARI score

The results for the hard dataset were not as good as for the medium and easy ones. The development of the ARI scores is depicted in Figure 4.7. The average ARI for the best clusterings is 0.68, which is approximately 0.2 less than for the easy and medium. Compared to the previous two datasets, the importance of choosing the right σ can be clearly seen in this case. The best clusterings can be seen in Figure 4.8.

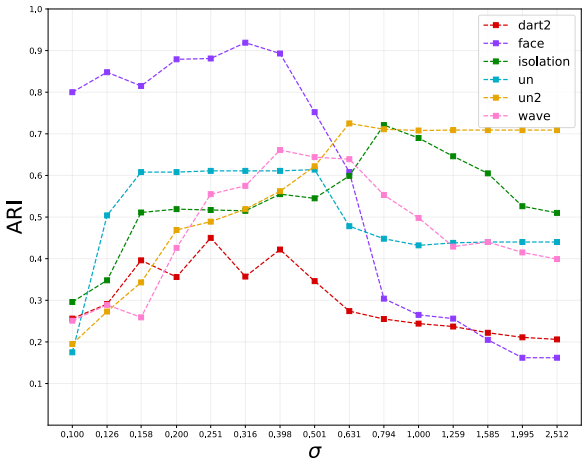
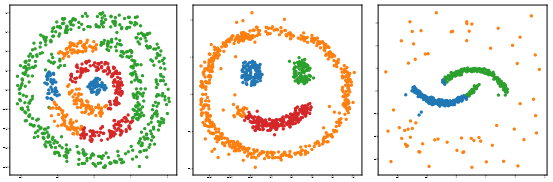
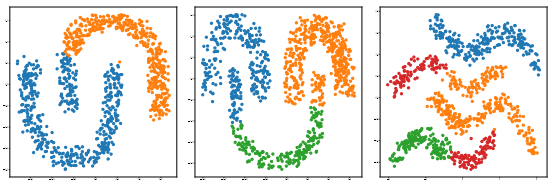


Figure 4.7: ARI scores for hard pattern sets



(a) dart2(0.450) (b) face(0.919) (c) isolation(0.721)



(d) un(0.614) (e) un2(0.725) (f) wave(0.661)

Figure 4.8: Best results for hard pattern sets with given ARI score

Table 4.1: Table with best ARI results for σ_{opt} and σ_{ref}

Pattern set	N	σ_{opt}	ARI_{opt}	σ_{ref}	ARI_{ref}
basic1	4	0.501	0.970	2.169	0.785
basic2	5	0.794	0.519	2.159	0.390
basic3	3	0.501	1.000	2.139	0.460
basic4	3	0.398	1.000	2.272	0.987
basic5	3	0.251	1.000	2.040	1.000
chrome	4	0.398	0.985	2.246	0.904
dart	2	0.631	1.000	2.195	0.007
dart2	4	0.251	0.450	2.185	0.202
face	4	0.316	0.919	2.181	0.162
isolation	3	0.794	0.721	2.263	0.505
lines	5	0.501	0.720	2.101	0.644
lines2	5	0.398	0.911	2.139	0.635
network	5	0.631	0.991	2.141	0.991
outliers	3	1.995	0.996	2.196	0.996
sparse	3	0.398	1.000	2.106	1.000
spiral2	2	0.316	0.996	2.147	0.854
spirals	3	0.398	1.000	2.183	0.751
triangle	3	1.000	1.000	2.079	1.000
un	2	0.501	0.614	2.185	0.440
un2	3	0.631	0.725	2.172	0.709
wave	4	0.398	0.661	2.160	0.428

4.4 Optimal Union of Hidden Classes

The optimal union of hidden classes is a clustering-based technique used for solving classification problems with multiple classes. A general classification task distributes n patterns into N classes, but the optimal union method is based on preprocessing, which places them into H hidden classes [6]. To construct these hidden classes, various clustering techniques can be used. In this case, the output of the kernel K-Means is used to construct them. The main idea is that the hidden classes can capture more subtle differences between the data points. Afterwards, the H hidden classes are merged into N classes. This merging process can be constructed in a way that maximizes the clustering/classification accuracy. It has been described in Section 2.2 in more detail.

The optimal union was performed on all datasets with ARI score lower than 0.9. Those pattern sets were *basic2*, *dart2*, *isolation*, *lines*, *un*, *un2* and *wave*, most of them from the hard dataset. All of them were clustered into $2N, 3N, \ldots$ classes, until either ARI equal to one was achieved or until it was not possible to obtain that many clusters from K-Means. In some cases, none of 100 K-Means iterations was able to create the clustering into a higher number of clusters. That was caused by one of the problems of the K-Means algorithm from section 3.2. The problem being that it can unintentionally merge two or more clusters together while assigning the patterns to their closest centroid. This happens more often with higher number of clusters. Possible fixes could involve adding more patterns or introducing a normalization constraint.

Table 4.2: Table with the best ARI results for different number of hidden classes

Pattern set	N	H						
		N	$2N$	$3N$	$4N$	$5N$	$6N$	$7N$
basic2	5	0.519	0.55	0.758	-	-	-	-
dart2	4	0.450	0.413	0.471	0.806	0.674	0.855	0.837
isolation	3	0.721	0.966	0.978	0.985	0.978	-	-
lines	5	0.720	0.812	0.819	-	-	-	-
un	2	0.614	0.617	0.689	1.000	1.000	-	-
un2	3	0.725	0.668	0.728	0.863	0.960	-	-
wave	4	0.661	0.827	0.990	-	-	-	-

X means that it was not possible to divide this pattern set into that many clusters.

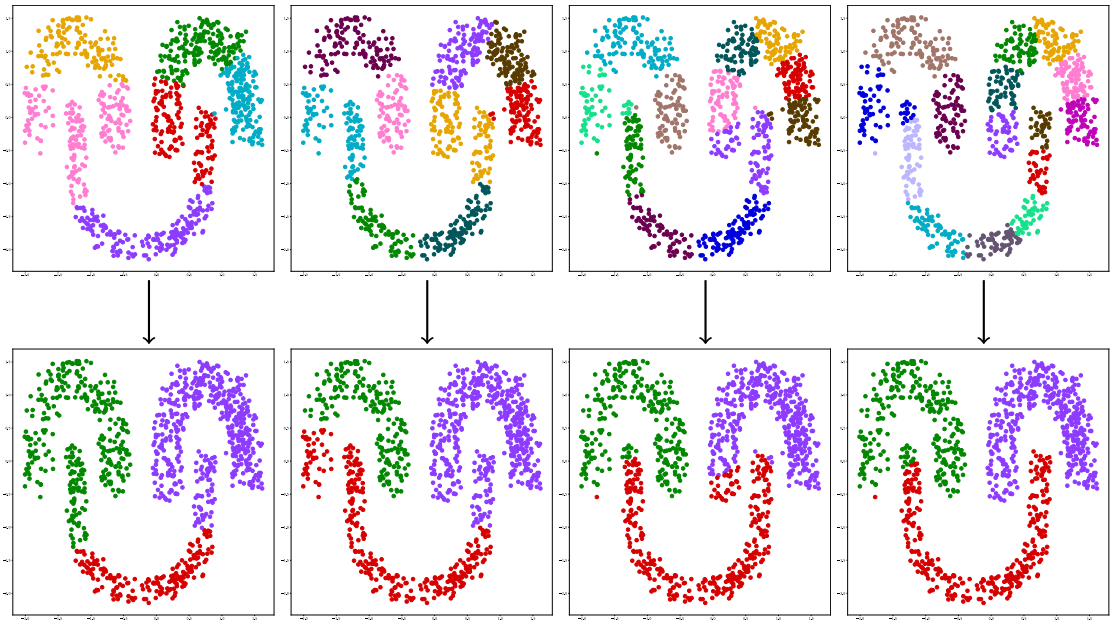


Figure 4.9: Pattern set *un2* before and after optimal union

Chapter 5

Influence of Mutations on the Performance of Random Descent

The goal of this chapter is to compare the performance of the random descent with different mutations from Section 3.5, and compare it with the performance of the standard K-Means from Section 3.2.

5.1 Pattern Set Description

Three pattern sets from [5] were chosen for this test. The first was the wine pattern set with 178 patterns and 13 different features, such as alcohol content, magnesium content and hue. There were three output classes. The second was the breast cancer pattern set with 683 patterns, 9 features and 2 output classes (malignant and benign). It has features such as cell nucleus radius, perimeter and texture. The third was the iris pattern set, which has 150 patterns, 4 features and 3 possible classes. All three pattern sets were standardized before the tests and optimal sigma was chosen empirically. All of this information is depicted in the following table.

Table 5.1: Pattern set properties

pattern set	patterns	features	classes	σ
wine	178	13	3	5.17
breast	683	9	2	4.71
iris	150	4	3	4.80

5.2 Testing Strategy

All of the three mutations defined in Section 3.5 were tested, each of them with a few different settings of parameters. Each configuration was run for 100 attempts and each attempt was capped at 100000 iterations of the Descent algorithm from Section 3.4. The optimal number of clusters N_{opt} was chosen by comparing the results obtained for different values and choosing the best accuracy acc_{opt}. The optimal union from Section 2.2 was performed afterwards. The convergence accuracy $\text{acc}_{\text{thresh}}$ was defined for each pattern set by lowering the acc_{opt} accordingly. The measured metrics were the percentage of convergence cases and the average number of iterations in these cases.

Table 5.2: Clustering tasks

Task	N_{opt}	acc_{opt}	acc_{kmeans}	acc_{thresh}
wine	3	0.994	0.971	0.95
breast	4	0.980	0.972	0.95
iris	6	0.966	0.933	0.90

5.3 Testing Results

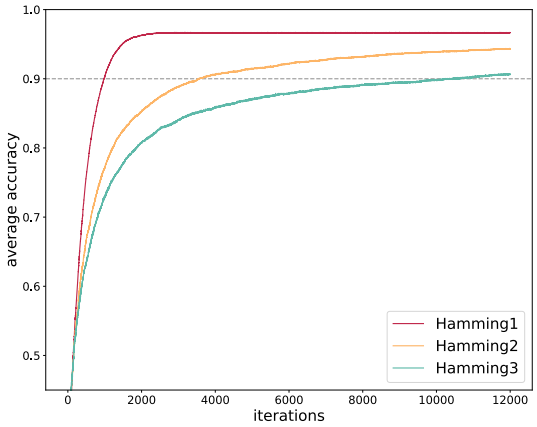
5.3.1 Wine pattern set

The results for the wine pattern set can be seen in Table 5.3.

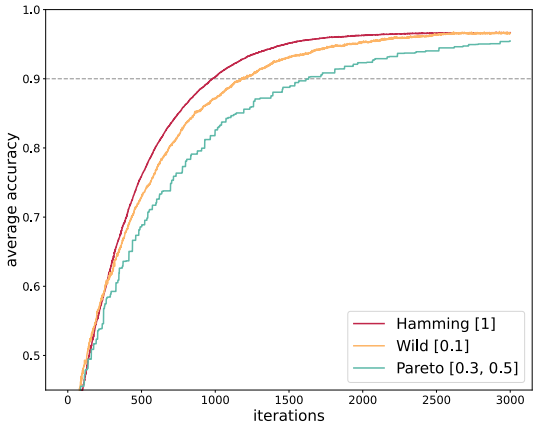
Table 5.3: Mutation Comparison on Wine Dataset

mutation	parameters	reliability[%]	ne (mean ± sd)
Hamming	1	100	1347 ± 214
	2	100	9156 ± 3277
	3	100	26917 ± 12413
Pareto	0.1,0.5	12	1328 ± 130
	0.1,1	12	1002 ± 123
	0.1,1.5	8	1168 ± 211
	0.3,0.5	26	2087 ± 740
	0.3,1	25	1452 ± 248
	0.3,1.5	12	1317 ± 520
	0.5,0.5	19	1984 ± 381
	0.5,1	15	1496 ± 347
	0.5,1.5	15	1356 ± 384
wild	0.02	14	1297 ± 520
	0.05	11	1451 ± 563
	0.1	19	1546 ± 465
	0.2	14	2004 ± 918

The only mutation that achieved 100% reliability was the Hamming mutation. It successfully converged for all three parameters, with the number of iterations increasing as the parameter value increased. This convergence behavior can be seen in Figure 5.1(a). This figure shows accuracy development for all the Hamming mutations. The development is calculated as the average across all 100 runs. By contrast, both the Pareto and wild mutations converged only in 10 to 25% cases. However, the number of iterations needed to converge remained low, averaging around 1000 to 2000 iterations.



(a) Average development of accuracy for different Hamming mutations



(b) Average development of accuracy for one candidate of every mutation

Figure 5.1: Development of accuracy for Wine pattern set

Figure 5.1(b) depicts the accuracy for all the successful runs for one candidate of each mutation. The accuracy is averaged by all the successful runs for each mutation. The accuracy development is very similar, with the Pareto mutation having the lowest values in general. Besides the Hamming mutation, the second most successful mutation was the Pareto mutation with parameters [0.1, 1]. It converged in 26 out of 100 cases. Figure 5.2 shows the development of accuracy for 6 random runs of the Pareto mutation that failed to converge. It can be observed that the accuracy stops improving as it nears 5000 to 10000 iterations, meaning that the lack of iterations is not the reason for failing to converge.

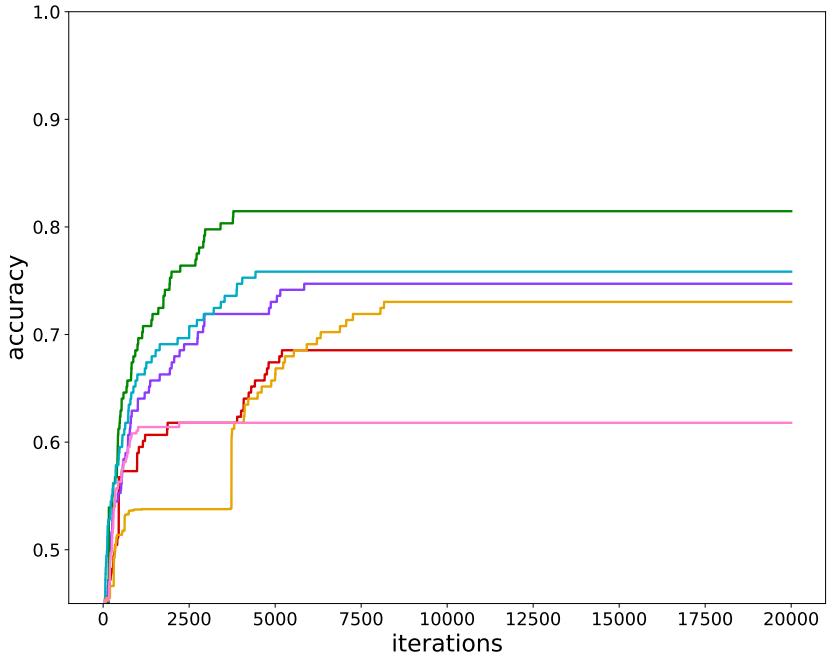


Figure 5.2: Six random unsucessful runs of Pareto(0.1, 1) mutation

5.3.2 Breast pattern set

All the methods achieved a 100% convergency rate on this pattern set. The maximal accuracy was 98% for the Hamming mutation with parameter 3. However, the difference in accuracies between different mutations was not very significant. The mutation with the least iterations to achieve convergency was the wild mutation with parameters equal to 0.02. The results can be seen in Table 5.4.

Table 5.4: Mutation Comparison on the Breast Dataset

mutation	parameters	reliability[%]	ne (mean ± sd)
Hamming	1	100	1283 ± 85
	2	100	3003 ± 347
	3	100	2826 ± 344
Pareto	0.1,0.5	100	1524 ± 135
	0.1,1	100	1326 ± 47
	0.1,1.5	100	1310 ± 57
	0.3,0.5	100	1724 ± 173
	0.3,1	100	1329 ± 69
	0.3,1.5	100	1336 ± 76
	0.5,0.5	100	1871 ± 122
	0.5,1	100	1428 ± 67
	0.5,1.5	100	1351 ± 109
wild	0.02	100	1033 ± 93
	0.05	100	1333 ± 90
	0.1	100	1422 ± 99
	0.2	100	1596 ± 117

The optimization criterion for the random descent with mutation is the criterion J_f defined using Equation (3.1) . Figure 5.3 shows the development of J_f and accuracy for one run of the Hamming mutation with parameter 1. It shows that the accuracy consistently improves as J_f decreases its value.

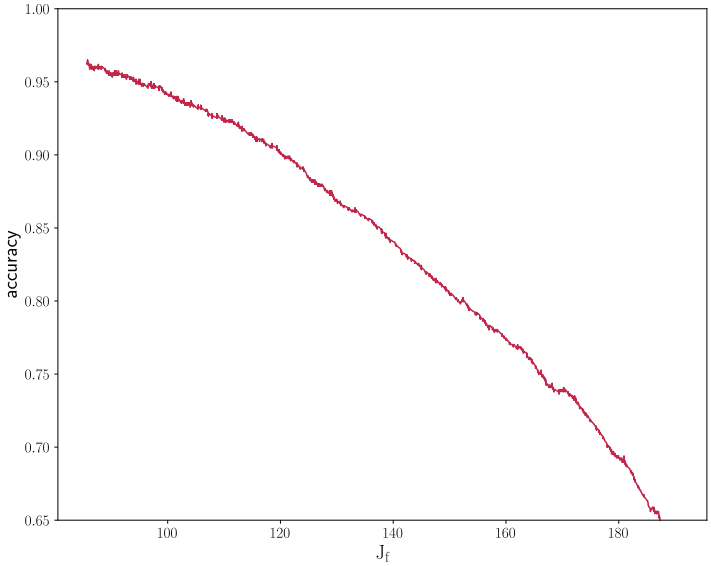


Figure 5.3: Development of accuracy and J_f for one run of wild(0.02) mutation

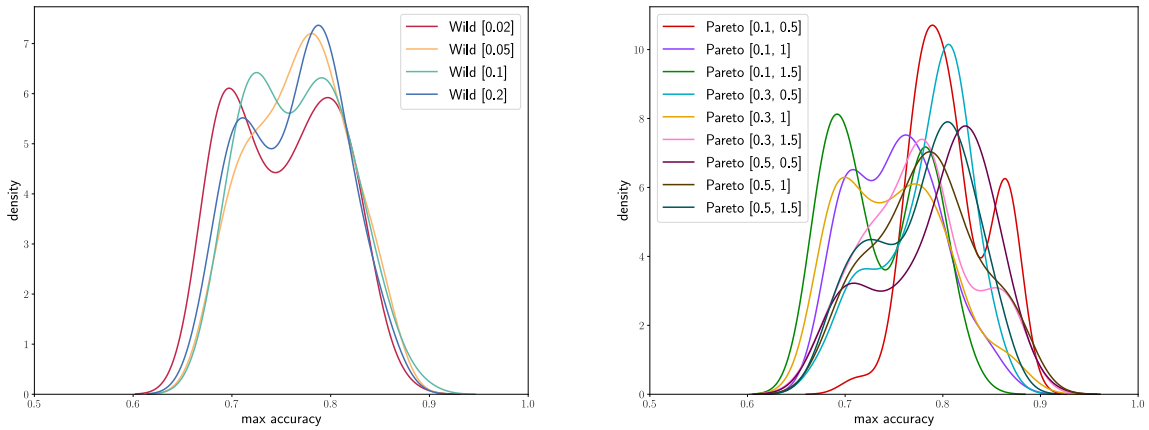
5.3.3 Iris pattern set

There were problems with achieving convergency for this pattern set as the only mutation that was able to achieve convergency was the Hamming mutation. Its results can be seen in Table 5.5.

Table 5.5: Hamming mutation Comparison on the Iris Dataset

mutation	parameters	reliability[%]	ne (mean ± sd)
Hamming	1	23	2121 ± 814
	2	23	9925 ± 7972
	3	31	28481 ± 17204

To further analyze the behaviour of different mutations and the resons of failure, the density plots and accuracy development plots can be used. The goal is to determine whether the lack of convergence was due to an insufficient number of iterations or if the accuracy reached its limit before achieving convergency. Analyzing the density estimates of maximum accuracy for each mutation will help identify any distinct patterns or trends. Additionally, plotting the accuracy development of random samples will provide a closer look at the progression of the accuracy over iterations.



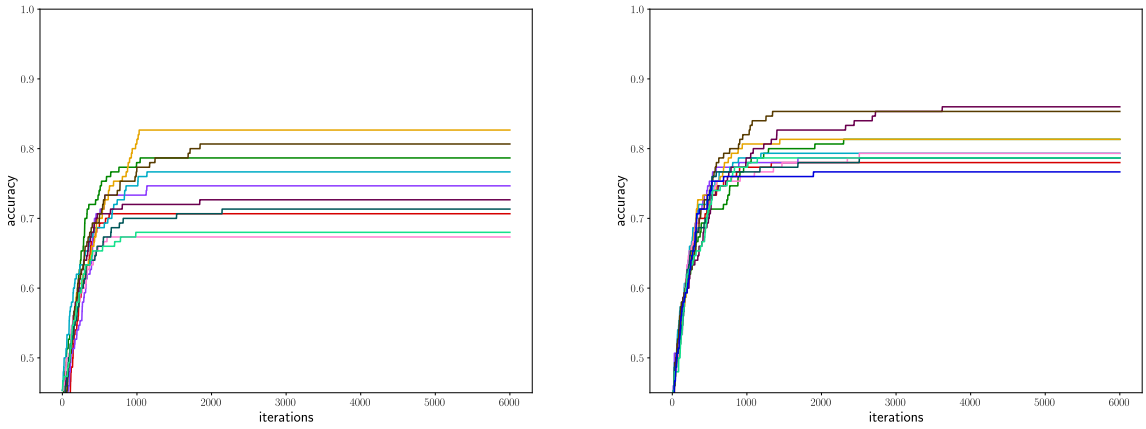
(a) wild mutations

(b) Pareto mutations

Figure 5.4: Density estimates of maximum accuracy

Figure 5.4 displays the kernel density estimates of maximum accuracy for all wild and Pareto mutations.

To produce smooth probability density estimates, kernel density estimation was used. Its basic principle is explained in [13]. The estimated densities were obtained using Python library *sklearn*. All the wild mutations show very similar trends, usually achieving accuracy between 70 and 85%. In contrast, the Pareto mutations show different behaviours according to different parameters used. Specifically, the Pareto mutation with parameters [0.1, 0.5] stands out as having the biggest peaks in the highest accuracy values. However, it can be seen that the probability of overcoming 90% accuracy is very low for both of the mutations.

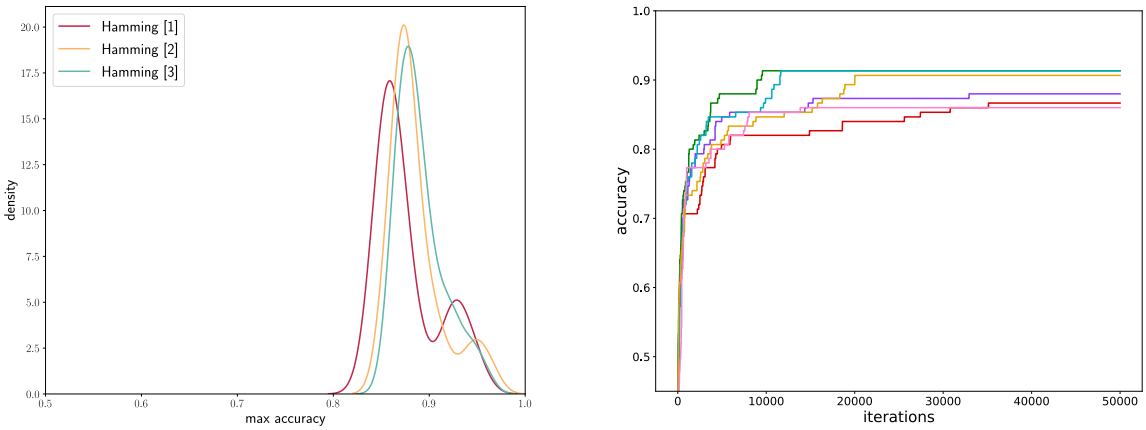


(a) wild (0.2) mutation

(b) Pareto (0.1, 0.5) mutation

Figure 5.5: Accuracy development of 10 random samples

To see whether the lack of iterations was the reason of failing to converge, accuracy development of random samples from each mutation can be used. Figure 5.5 depicts 10 random samples from the wild [0.2] mutation and the Pareto [0.1, 0.5] mutation. This figure shows that both the mutations follow a similar trend of rapid attainment of peak accuracy, at approximately 1500 iterations. It can also be seen that the wild mutation produces lower values of accuracy in general.



(a) Density estimates of maximum accuracy for different Hamming mutations

(b) Accuracy development of 6 random samples from wild (3) mutation

Figure 5.6: Behaviour of Hamming mutation for Iris pattern set

Figures 5.6(a) and 5.6(b) both suggest very different qualities for the Hamming mutation. Notably, the density estimates have peaks in higher values, approaching 90%. The peak value gradually increases with an increasing parameter of the Hamming mutation. Contrary to the wild and Pareto mutations, the Hamming mutation also exhibits a contrasting behavior in terms of achieving the peak accuracy. As observed in Figure 5.6(b), the accuracy of the Hamming mutation experiences a rapid increase, reaching the value of 70%. However, unlike the previous mutations, the Hamming mutation continues to gradually improve its accuracy over the next 20,000 iterations and beyond.

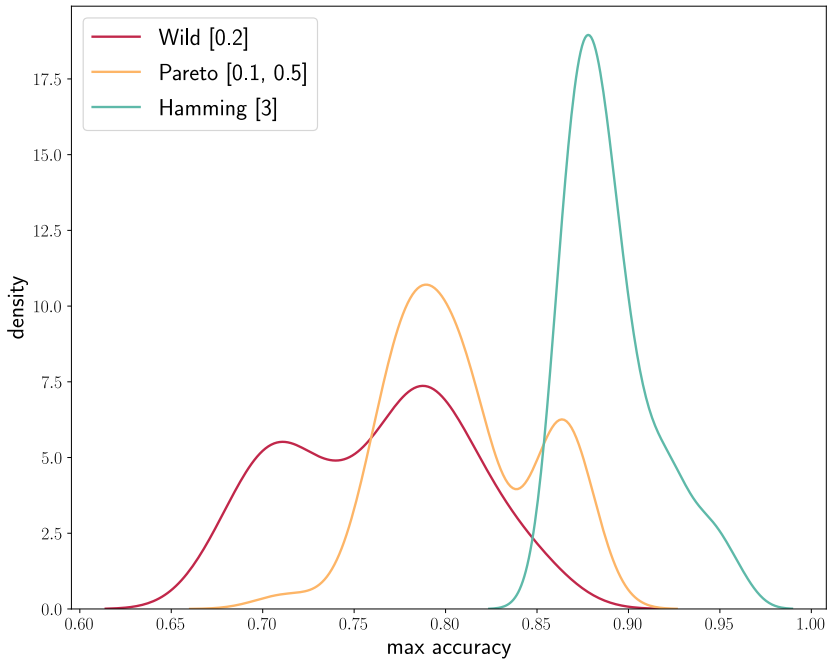


Figure 5.7: Density estimates for candidates from each mutation

In conclusion, Figure 5.7 provides a comparison of the best candidates from each mutation type. It is apparent that the Hamming mutation outperforms the other mutations for this pattern set. The comparison also shows that the Pareto mutation performed better that the wild one. The Hamming mutation demonstrates the highest consistency among the three, indicating that the optimization process avoids local minima, and all the runs yield similar results, as indicated by Figure 5.7. Additionally, it also achieves the highest average maximum accuracy. The only disadvantage of the Hamming mutation is that the number of iterations required to obtain these results is slightly higher, averaging at 28481 iterations for the Hamming mutation with parameter 3.

Chapter 6

Implementation of Kernel Library in Python

As a part of this work, a Python library for working with kernel algorithms, called KernPy, was created. It can be found online: https://github.com/xdxdhh/kernel_lib. It serves as a base library that provides different kernel functions, kernel matrices and kernel versions of standard machine learning algorithms for classification and clustering tasks. The whole library was written using Python 3.9 and it currently consists of four modules.

Modules kernel.py and kernel_matrix.py are the building blocks of all, seeing as they cover basic utilities for working with kernel algorithms, such as different kernels and kernel matrix class. The third module is cluster.py, which should serve as the main envelope for kernel clustering algorithms. It currently supports two clustering methods, the first one being Kernel K-Means from Section 3.2 and the second one being Random Descent with optimization criterion in the feature space, as described in Section 3.4. In the future, the KernPy library could be expanded to have more clustering methods. A classification module could also be added easily. The last module, mutation.py, implements different mutation operators, which are used in Random Descent optimization and other heuristics algorithms. All of the modules will be described in more detail in the following sections.

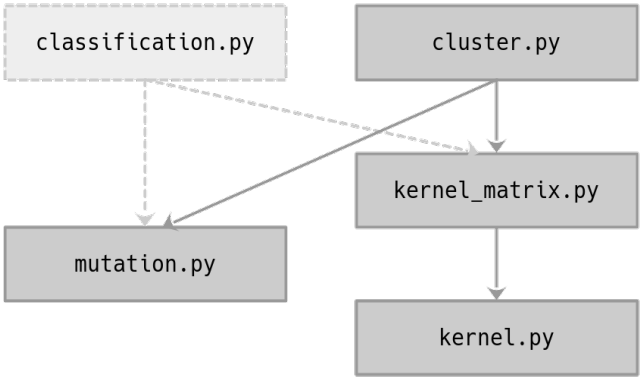


Figure 6.1: The structure of KernPy library

6.1 Kernel module

This module should contain implementations of different kernel functions for different data types. Currently supported kernels are the Gaussian (RBF) kernel and the polynomial kernel, as defined in 1.3 and 1.2. Each kernel has an *eval*() function, which evaluates its value for any two given vectors. Parameters for each kernel should be passed as an array, even if the kernel only has one parameter. The main reason for this is the unified callability of all the kernels. All kernel classes should inherit from the kernel_base class, which defines the required methods.

class kernel_base()

The base kernel class which defines all kernel classes.

Parameters:

None

Methods:

kernel.eval(x, y)
Evaluate kernel function on two given vectors **x** and **y**.
kernel.optimal_params(X)
Estimate optimal kernel parameters for given data from data matrix **X**.

class gauss_kernel(params)

Class which implements the gaussian kernel defined as $\kappa(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{\|\mathbf{x}-\mathbf{y}\|^2}{2\sigma^2}\right)$.

Parameters:

params : array containing σ parameter

Methods:

gauss_kernel.eval(x, y)
Evaluate Gaussian kernel function on vectors **x** and **y**.
gauss_kernel.optimal_params(X)
Estimate optimal σ parameter from data matrix **X**.
The optimal σ^2 is estimated as an average of 0.1 and 0.9 quantiles of $\|\mathbf{x}_l - \mathbf{x}_k\|^2, k \neq l$.

class polynom_kernel(params)

Class which implements the polynomial kernel defined as $\kappa(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^{\mathrm{T}}\mathbf{y} + c)^d$.

Parameters:

params : array containing degree parameter d and shift parameter c
If only the degree is given, **c** is assumed as zero.

Methods:

polynom_kernel.eval(x, y)
Evaluate polynomial kernel function on vectors **x** and **y**.

6.2 Kernel Matrix module

class Kernel_matrix(X, kernel, params)

This class should serve as an input for all kernel algorithms from KernLib. It implements basic utilities such as access operators and string representiation.

Attributes:

n : dimension of kernel matrix
shape : 2-dimensional array [n,n]
K : contents of kernel matrix, stored as *numpy* 2-dimensional array

Parameters:

X : array-like, matrix of input data, with one record stored in one row
kernel : name of kernel from kernel module that will be used to create the matrix
{"gauss", "polynomial", "identity"}
"identity" kernel can be used to create kernel matrix from already existing square matrix
params : array of parameters for the given kernel(e.g σ for Gaussian)
value "auto" can be used to estimate the parameters using *kernel.optimal_params*() method

Methods:

Kernel_matrix.submatrix(index_list)
Obtain submatrix while leaving only rows and columns from index list.
Resulting matrix is a kernel matrix for those rows from input data X that are on those indexes.

6.3 Mutation module

This module contains different mutations that can be used in heuristic methods in clustering tasks. Currently, there are three available mutations: Pareto, Hamming and Wild. Each specialized mutation class must inherit from the base mutation class and thus implement the mutate method.

class Mutation()

The base mutation class which defines all mutation classes. It also implements a perturbation function and the random_point() function which can be shared by all the mutations. This class should not be used itself, its child classes should be used instead.

Parameters:

None

Methods:

mutation.mutate(x):
Change the value of vector **x** by applying mutation to it.
Note that this function creates the new **x** inplace.
mutation.perturbation(x, a, b)
Change the value of vector **x** by applying perturbation to it.
The boundaries of the perturbation are defined using **a** for lower bound, **b** for upper bound.
mutation.random_point(a, b)
Generate random point between **a** and **b**.
mutation.mut_rstar()
Small helper mutation that can be used by other mutations.
Changes one element by adding 1 to it or substracting 1 from it.

class Hamming_mutation(a, b, params)

This class implements the Hamming mutation, which creates new solution by changing n_{mut} elements of a given vector.

Parameters:

a : lower bound for the Hamming mutation
b : upper bound for the Hamming mutation
params : array containing n_{mut} parameter that defines the number of positions to be changed when mutating

Methods:

Hamming_mutation.mutate(x)
Change the value of **x** using the Hamming mutation.

class Wild_mutation(a, b, params)

This class implements the wild mutation, which has certain probability p_{wild} of shuffling all the elements of **x** randomly. In other cases, it just changes one element by adding 1 to it or substracting 1 from it.

Parameters:

a : lower bound for the wild mutation
b : upper bound for the wild mutation
params : array containing p_{wild} parameter that defines the probability of random shuffle

Methods:

mutation.mutate()
Change the value of **x** using Wild mutation.

class Pareto_mutation(a, b, params)

This class implements the Pareto mutation, which modifies the input vector by adding a random vector to it. For obtaining the random vector, the Pareto distribution is used. This mutation has two arguments, the temperature $T > 0$ and the shape of the Pareto distribution $\alpha \in (0, 2)$.

Parameters:

a : lower bound for the Pareto mutation
b : upper bound for the Pareto mutation
params : array containing T parameter and α parameter

Methods:

mutation.mutate()
Change the value of **x** using the Pareto mutation.

6.4 Clustering module

This module should encompass different kernel clustering methods. Currently, there are two implemented ways to obtain clustering of the given data. The first one uses standard K-means algorithm as described in 3.2. The second one uses the Random Descent from 3.4 together with mutations.

class KMeans(n_clusters, max_iter, verbose)

K Means clustering class.

Parameters:

n_clusters : number of desired clusters
max_iter : maximal number of k-means iterations

Methods:

kmeans.fit_one(K)
Perform one run of k-means algortihm.
kmeans.fit(K, n)
Perform n runs of k-means algortihm and choose the best one.
The best one is chosen as the one with the smallest value of the objective function.
kmeans.obj_function(K, partition)
Return the value of the objective function for a given partition of the data.

class Heuristics(n_clusters, max_iter, verbose)

Class which implements the random descent heuristic method in order to obtain the best solution.

Attributes:

mutation_dict : dictionary of all the possible mutations to be used from mutation.py module

Parameters:

n_clusters : number of desired clusters
max_iter : maximal number of random descent iterations

Methods:

heuristics.fit(K, mutation, params = "default")
Perform Random descent with a given mutation.
Params should be the parameters of the mutation.
If "default" is used, the default mutation params will be used.
Default parameters are [0.1, 1] for Pareto, 0.1 for wild and 1 for Hamming mutation.

heuristics.obj_function(K, partition)
Return the value of the objective function for a given partition of the data.

Conclusion

This work explained the underlying principles of kernel clustering methods and the basic properties in the feature space, all using vectorial input data. It presented kernel matrices and introduced two widely used kernels, polynomial kernel and Gaussian kernel.

The clustering problem in the feature space was defined, along with metrics for comparing clustering results, namely the Rand index, Adjusted Rand index and accuracy. Two possible ways of performing clustering in the feature space were explained. The first one was the kernel version of a popular clustering algorithm K-Means. This algorithm was afterwards used to perform series of tests with the goal of determining the optimal σ parameter for the Gaussian kernel. The tests involved 21 different two-dimensional pattern sets. The optimal σ values found were later compared to σ_{ref} values obtained using formula from [2], showing that it is possible to obtain better results by further optimizing the parameter. In the end, the optimal union of hidden classes from [6] was used to fine-tune the results even more.

The second approach for solving the clustering problem involved heuristic methods, specifically random descent. Three different mutations were used to tackle the issue of local minima - Pareto, Hamming and wild mutation. Their performance was compared in the last chapter, using three known pattern sets from [5] - iris, wine and breast. The results showed that even such a simple mutation as the Hamming mutation can produce very good results.

Concise Python library PyKern was created and used to conduct all the experiments. This library provides tools for working with kernels, kernel matrices and can perform both the kernel K-Means and random descent algorithms.

Appendix

Table 6.1: Table showing values of Adjusted Rand Index for different σ values

Pattern set	N	σ							
		0.100	0.126	0.158	0.200	0.251	0.316	0.398	0.501
basic1	4	0.173	0.293	0.322	0.411	0.495	0.804	0.804	0.970
basic2	5	0.177	0.234	0.234	0.308	0.367	0.513	0.490	0.505
basic3	3	0.191	0.34	0.281	0.47	0.478	0.596	0.831	1.000
basic4	3	0.154	0.273	0.374	0.556	0.782	0.997	1.000	0.997
basic5	3	0.263	0.332	0.716	1.000	1.000	1.000	1.000	1.000
chrome	4	0.118	0.236	0.301	0.375	0.496	0.761	0.985	0.977
dart	2	0.460	0.410	0.382	0.402	0.576	0.500	0.384	0.709
dart2	4	0.256	0.291	0.396	0.356	0.450	0.357	0.422	0.346
face	4	0.800	0.848	0.815	0.879	0.881	0.919	0.893	0.752
isolation	3	0.296	0.348	0.511	0.519	0.517	0.515	0.555	0.545
lines	5	0.178	0.234	0.280	0.368	0.416	0.488	0.718	0.720
lines2	5	0.206	0.262	0.359	0.462	0.604	0.740	0.911	0.869
network	5	0.226	0.337	0.804	0.837	0.955	0.983	0.985	0.991
outliers	3	0.137	0.199	0.447	0.569	0.639	0.633	0.816	0.818
sparse	3	0.181	0.354	0.419	0.676	0.958	0.958	1.000	1.000
spiral2	2	0.082	0.122	0.302	0.476	0.980	0.996	0.980	0.964
spirals	3	0.225	0.324	0.547	0.430	0.604	0.834	1.000	1.000
triangle	3	0.135	0.290	0.414	0.500	0.541	0.587	0.755	0.994
un	2	0.175	0.504	0.608	0.608	0.611	0.611	0.611	0.614
un2	3	0.195	0.273	0.343	0.469	0.489	0.519	0.562	0.622
wave	4	0.251	0.289	0.259	0.426	0.555	0.575	0.661	0.644

Pattern set	N	σ						
		0.631	0.794	1.000	1.259	1.585	1.995	2.512
basic1	4	0.950	0.913	0.897	0.798	0.781	0.745	0.723
basic2	5	0.489	0.519	0.509	0.395	0.380	0.358	0.358
basic3	3	0.912	0.584	0.558	0.49	0.469	0.459	0.459
basic4	3	0.994	0.994	0.994	0.991	0.987	0.987	0.987
basic5	3	1.000	1.000	1.000	1.000	1.000	1.000	1.000
chrome	4	0.957	0.954	0.951	0.942	0.926	0.904	0.904
dart	2	1.000	0.764	0.008	0.008	0.007	0.007	0.007
dart2	4	0.274	0.255	0.244	0.237	0.222	0.211	0.206
face	4	0.608	0.304	0.265	0.256	0.205	0.162	0.162
isolation	3	0.599	0.721	0.690	0.646	0.605	0.526	0.510
lines	5	0.583	0.684	0.688	0.685	0.643	0.644	0.635
lines2	5	0.78	0.774	0.732	0.707	0.702	0.699	0.656
network	5	0.991	0.991	0.991	0.991	0.991	0.991	0.991
outliers	3	0.823	0.822	0.825	0.825	0.825	0.996	0.996
sparse	3	1.000	1.000	1.000	1.000	1.000	1.000	1.000
spiral2	2	0.929	0.895	0.887	0.872	0.865	0.857	0.850
spirals	3	1.000	1.000	0.916	0.777	0.771	0.753	0.751
triangle	3	0.994	0.994	1.000	1.000	1.000	1.000	0.995
un	2	0.478	0.448	0.432	0.438	0.440	0.440	0.440
un2	3	0.725	0.711	0.708	0.709	0.709	0.709	0.709
wave	4	0.639	0.553	0.498	0.429	0.44	0.415	0.399

Bibliography

[1] Gérard Biau, Luc Devroye, and Gábor Lugosi. On the performance of clustering in hilbert spaces. *IEEE Transactions on Information Theory*, 54(2):781–790, 2008.

[2] Barbara Caputo, K Sim, Fredrik Furesjo, and Alex Smola. Appearance-based object recognition using svms: which kernel should i use? In *Proc of NIPS workshop on Statistical methods for computational experiments in visual processing and computer vision, Whistler*, volume 2002, 2002.

[3] Francisco Carvalho, Eduardo Simões, Lucas Santana, and Marcelo Ferreira. Gaussian kernel c-means hard clustering algorithms with automated computation of the width hyper-parameters. *Pattern Recognition*, 79, 02 2018.

[4] Yin-Wen Chang, Cho-Jui Hsieh, Kai-Wei Chang, Michael Ringgaard, and Chih-Jen Lin. Training and testing low-degree polynomial data mappings via linear svm. *Journal of Machine Learning Research*, 11(48):1471–1490, 2010.

[5] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[6] Radek Hrebik, Jaromir Kukal, and Josef Jablonsky. Optimal unions of hidden classes. *Central European Journal of Operations Research*, 27:161–177, 2019.

[7] Lawrence J. Hubert and Phipps Arabie. Comparing partitions. *Journal of Classification*, 2:193–218, 1985.

[8] Abdoun Otman, Jaafar Abouchabaka, and Chakir Tajani. Analyzing the performance of mutation operators to solve the travelling salesman problem. *Int. J. Emerg. Sci.*, 2, 03 2012.

[9] William M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(336):846–850, 1971.

[10] Andrew Rosenberg and Julia Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. pages 410–420, 01 2007.

[11] scikit-learn developers.

[12] John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.

[13] Weglarczyk, Stanislaw. Kernel density estimation and its application. *ITM Web Conf.*, 23:00037, 2018.

[14] Ka Yee Yeung and Walter Ruzzo. Details of the adjusted rand index and clustering algorithms supplement to the paper "an empirical study on principal component analysis for clustering gene expression data" (to appear in bioinformatics). *Science*, 17, 01 2001.