TECHNISCHE UNIVERSITÄT
CHEMNITZ

Fast Matrix-Vector Multiplication for the ANOVA Kernel

Master Thesis

submitted by

Theresa Wagner

Professorship of Scientific Computing

Faculty of Mathematics

Chemnitz University of Technology

Supervisors: Prof. Dr. Martin Stoll

Dr. Franziska Nestler

Chemnitz, 08 October 2020

Contents

1 Introduction

In today's world, vast amounts of data are collected and stored everywhere. Information have become one of the most valuable things of our time. However, if we do not know how to interpret them correctly, they are worthless for us. Usually, real world data has huge dimensions and can impossibly be evaluated by a human. This would take years and before having finished, the data would be out of date and useless. Thus, we need computers to automatically recognise patterns in data. Here, by patterns, we understand any relations or structure in some given source of data. The point of detecting significant patterns in data is to be able to make predictions about new data from the same source later. Frequently, one feature of the data is isolated and is intended to be predicted as a function of the other feature values. Then, the system is trained with the given data, so that it learns something about the source, which generated the data. That way, the system acquires generalisation power and is enabled to predict class affiliations [1]. In this thesis, we deal with binary classification problems.

We start with a chapter on two classical learning methods. This theoretical introduction provides the background for all further chapters. We learn that efficiently detecting linear patterns often succeeds with well-known procedures. Usually, real world problems require non-linear methods though. At this point, kernel functions come into play, since they enable us to represent non-linear patterns through linear relations. The underlying theory is covered by Chapter 3. In Chapter 4, we illustrate first results. For that, the prediction quality is examined for exemplary data sets. Both learning methods are performed and the classification rate is compared for several settings. However, the computational complexity of performing the learning methods from Chapter 2 scales bad for high-dimensional data sets. Chapter 5 is devoted to design a solution to this problem.

2 Learning Methods

In this chapter we give an introduction to *learning methods* that utilise positive definite *kernels* [1, 2]. As described above, we aim to detect dependencies to successfully predict class affiliations. The underlying theory of machine learning is well developed for the linear case indeed [1]. Our problem requires typically non-linear methods though. By applying the so-called *kernel trick*, this difficulty can be overcome. It is common practice to embed the data into a high-dimensional *feature space*, where the patterns can be represented through linear relations. This is done by replacing dot products by a *kernel evaluation* and allows us to utilise linear methods in this space without ever explicitly having to compute in it (cf. [1, 2, 3]).

We are given a *data* or *design matrix*

$$X = \begin{bmatrix} \mathbf{x}_1{}^T \\ \vdots \\ \mathbf{x}_N{}^T \end{bmatrix} \in \mathbb{R}^{N \times n}, \tag{2.1}$$

whose rows are represented by *feature vectors* $\mathbf{x}_i \in \mathbb{R}^n$. $N \in \mathbb{N}$ denotes the number of data points and $n \in \mathbb{N}$ the number of features. In the case of binary pattern recognition, each of those vectors possesses an assigned label $y_i \in \{-1, 1\}$. This yields pairs $(\mathbf{x}_i, y_i)$. Usually, the amount of labelled input data is limited. Our aim is to predict corresponding labels for data points, which have not been labelled yet. Generalising to unlabelled data is the heart of *learning*. It can be achieved using the classifications for given labelled data [2].

Remark 2.1. Usually, different entries in the *feature vectors* $\mathbf{x}_i \in \mathbb{R}^n$ do not have the same magnitude. Therefore, it is extremely important to scale the data before starting the learning process. Otherwise, features are weighted differently from the beginning, what leads to distorted results. Of course, different features of $\mathbf{x}_i \in \mathbb{R}^n$ are differently related to the corresponding label y_i. But determining this is the purpose of the *learning method* and has nothing to do with the magnitude of the features per se.

2.1 Kernel Ridge Regression

Kernel Ridge Regression is a fundamental method for detecting dependencies between *features* of given data and *responses* [4]. It combines *ridge regression* with *kernel methods* and has two phases: the training and the prediction process. The training phase builds and solves a least-squares problem from a subset of given data, while the prediction phase uses this model to predict the label of a new data point. Therefore, it stands to reason to divide the set of labelled data points into training data and test data. By doing this, the training data can be used for the training process and the test data helps us determining the prediction quality.

2.1.1 Linear Regression and Ridge Regression

Implementing the simple linear regression

$$\hat{w} = \underset{w \in \mathbb{R}^n}{\arg\, min} \, \|f - Xw\|_2^2 + \lambda \|w\|_2^2, \tag{2.2}$$

yields *weights* $\hat{w} \in \mathbb{R}^n$, with $f \in \mathbb{R}^N$ being a given *response vector* incorporating the labels y_i and $\lambda > 0$ balancing the importance of $\|w\|_2^2$. For $\lambda = 0$ these *weights* obviously minimise

$$\begin{aligned} \sum_{i=1}^{N} r_i^2 &= \sum_{i=1}^{N} \left(f_i - {\mathbf{x}_i}^T w\right)^2 \\ &= \sum_{i=1}^{N} \left(y_i - {\mathbf{x}_i}^T w\right)^2, \end{aligned} \tag{2.3}$$

i. e. the sum of the squared *residuals* r_i. This process is called *training*. Having computed $\hat{w}$, the linear model

$$F\left(\mathbf{x}_{\text{new}}\right) = {\mathbf{x}_{\text{new}}}^T \hat{w} \tag{2.4}$$

yields the *predicted response* for a new point $\mathbf{x}_{\text{new}} \in \mathbb{R}^n$.

Suppose $N \geq n$, i. e. the number of training data is bigger or equal the dimen-

sion of the feature vectors, and $\text{rank}\,(X) = n$. Then we have

$$\begin{aligned}\hat{w} &= \underset{w\in\mathbb{R}^n}{\arg min} \sum_{i=1}^{N} r_i^2 \\ &= \underset{w\in\mathbb{R}^n}{\arg min} \sum_{i=1}^{N} \left(y_i - {\mathbf{x}_i}^T w\right)^2 \\ &= \underset{w\in\mathbb{R}^n}{\arg min} \sum_{i=1}^{N} \left(y_i - \sum_{j=1}^{n} \mathbf{x}_i^j w_j\right)^2 \\ &= \underset{w\in\mathbb{R}^n}{\arg min} \underbrace{\|f - Xw\|_2^2}_{r(w)},\end{aligned} \tag{2.5}$$

with

$$\begin{aligned}r\,(w) &= (f - Xw)^T (f - Xw) \\ &= w^T X^T X w - w^T X^T f - f^T X w + f^T f \\ &= w^T X^T X w - 2w^T X^T f + f^T f.\end{aligned} \tag{2.6}$$

Setting the differentiation of $r\,(w)$ zero

$$\nabla_w r\,(w) = 2X^T X w - 2X^T f \stackrel{!}{=} 0 \tag{2.7}$$

gives

$$\hat{w} = \left(X^T X\right)^{-1} X^T f. \tag{2.8}$$

Obviously, this result for $\hat{w}$ just makes sense if the inverse of $X^T X \in \mathbb{R}^{n\times n}$ actually exists. By assumption, $\text{rank}\,(X) = n$ so that $X \in \mathbb{R}^{N\times n}$ has full rank. Hence,

$$n = \text{rank}\,(X) = \text{rank}\left(X^T\right) = \text{rank}\left(X^T X\right), \tag{2.9}$$

i. e. $X^T X$ has full rank and is invertible. We see that $\hat{w}$ in (2.8) is well-defined for $N \geq n$.

Next, let us assume $N < n$. This time

$$\text{rank}\,(X) = \text{rank}\left(X^T\right) = \text{rank}\left(X^T X\right) \leq N < n, \tag{2.10}$$

such that $X^T X$ has no full rank and is therefore not invertible. Clearly, using Definition (2.8) is no option now. Let us require $\|f - Xw\|_2$ and $\|w\|_2$ to be small. Glancing at (2.2) we define the linear regression problem

$$\hat{w} = \underset{w\in\mathbb{R}^n}{\arg min} \|f - Xw\|_2^2 + \lambda\|w\|_2^2. \tag{2.11}$$

In data science this process is called *ridge regression* with λ the *ridge parameter*. Since the function in (2.11) does not look easy to use yet, we try to simplify it:

$$\begin{aligned} \|f - Xw\|_2^2 + \lambda\|w\|_2^2 &= \left\| \begin{bmatrix} f - Xw \\ \sqrt{\lambda}w \end{bmatrix} \right\|_2^2 \\ &= \left\| \begin{bmatrix} f \\ 0 \end{bmatrix} - \begin{bmatrix} X \\ -\sqrt{\lambda}I_n \end{bmatrix} w \right\|_2^2 \\ &= \left\| \hat{f} - \hat{X}w \right\|_2^2, \end{aligned} \tag{2.12}$$

with $\hat{f} = \begin{bmatrix} f \\ 0 \end{bmatrix} \in \mathbb{R}^{N+n}$, $\hat{X} = \begin{bmatrix} X \\ -\sqrt{\lambda}I_n \end{bmatrix} \in \mathbb{R}^{(N+n)\times n}$ and $\operatorname{rank}\left(\hat{X}\right) = n$. Comparing (2.12) with (2.5) reveals that Definition (2.8) for $\hat{w}$ can be applied to $\hat{f}$ and $\hat{X}$ now. $\hat{X}$ having full rank makes $\hat{X}^T\hat{X}$ invertible. This leads to

$$\begin{aligned} \hat{w} &= \left(\hat{X}^T\hat{X}\right)^{-1}\hat{X^T}\hat{f} \\ &= \left(X^TX + \lambda I_n\right)^{-1}X^Tf. \end{aligned} \tag{2.13}$$

By the Sherman-Morrison-Woodbury formula [3], (2.13) can be rewritten as

$$\begin{aligned} \hat{w} &= \left(X^TX + \lambda I_n\right)^{-1}X^Tf \\ &= X^T\underbrace{\left(XX^T + \lambda I_N\right)^{-1}f}_{\alpha}. \end{aligned} \tag{2.14}$$

Clearly, these calculations work either way, for $N < n$ but also for $N \geq n$. Hence, from now on (2.14) is considered a general method for *training* [3]. The matrix $\left(XX^T + \lambda I_N\right)$ is symmetric and positive definite for $\lambda > 0$. Hence, we can use the *conjugate gradient method* to determine α. This is what we will deal with in Section 5.2.

As mentioned previously, we aim to compute the *predicted response* for a new data point $\mathbf{x}_{\text{new}} \in \mathbb{R}^n$. Bringing (2.14) and (2.4) together yields

$$\begin{aligned} F\left(\mathbf{x}_{\text{new}}\right) &= {\mathbf{x}_{\text{new}}}^T\hat{w} \\ &= \hat{w}^T\mathbf{x}_{\text{new}} \\ &= \left(X^T\alpha\right)^T\mathbf{x}_{\text{new}} \\ &= \alpha^TX\mathbf{x}_{\text{new}} \\ &= \sum_{i=1}^{N}\alpha_i{\mathbf{x}_i}^T\mathbf{x}_{\text{new}}. \end{aligned} \tag{2.15}$$

2.1.2 Kernel Evaluation

Most real world data analysis problems cannot be described by a linear model. They therefore possess poor prediction quality. In data science, it is common practice to map data to a high-dimensional space and learn the model there as the data is easier to separate. This is done by replacing dot products by a non-linear *kernel function*, which describes the similarity of data in the high-dimensional space. This procedure is called the *kernel trick* [4].

Looking closely at (2.15), we recognise the inner product ${\mathbf{x}_i}^T\mathbf{x}_{\text{new}}$ in the last line. Replacing it by a *kernel evaluation*

$$\kappa\left(\mathbf{x}_i, \mathbf{x}_j\right) = {\mathbf{x}_i}^T\mathbf{x}_j \tag{2.16}$$

yields

$$\begin{aligned} F\left(\mathbf{x}_{\text{new}}\right) &= \sum_{i=1}^{N} \alpha_i {\mathbf{x}_i}^T\mathbf{x}_{\text{new}} \\ &= \sum_{i=1}^{N} \alpha_i \kappa\left(\mathbf{x}_i, \mathbf{x}_{\text{new}}\right). \end{aligned} \tag{2.17}$$

Moreover, we remember the definition of the *dual variable*

$$\alpha = \left(XX^T + \lambda I_N\right)^{-1} f \tag{2.18}$$

from (2.14). Since all entries of

$$XX^T = \begin{bmatrix} {\mathbf{x}_1}^T \\ \vdots \\ {\mathbf{x}_N}^T \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 & \dots & \mathbf{x}_N \end{bmatrix} = \begin{bmatrix} {\mathbf{x}_1}^T\mathbf{x}_1 & {\mathbf{x}_1}^T\mathbf{x}_2 & \dots & {\mathbf{x}_1}^T\mathbf{x}_N \\ \vdots & & & \vdots \\ {\mathbf{x}_N}^T\mathbf{x}_1 & {\mathbf{x}_N}^T\mathbf{x}_2 & \dots & {\mathbf{x}_N}^T\mathbf{x}_N \end{bmatrix} \tag{2.19}$$

are inner products, XX^T is a so-called *Gram matrix*. Applying the *kernel trick* again, we obtain the *kernel matrix*

$$K = XX^T = \begin{bmatrix} {\mathbf{x}_1}^T\mathbf{x}_1 & \dots & {\mathbf{x}_1}^T\mathbf{x}_N \\ \vdots & & \vdots \\ {\mathbf{x}_N}^T\mathbf{x}_1 & \dots & {\mathbf{x}_N}^T\mathbf{x}_N \end{bmatrix} = \begin{bmatrix} \kappa\left(\mathbf{x}_1, \mathbf{x}_1\right) & \dots & \kappa\left(\mathbf{x}_1, \mathbf{x}_N\right) \\ \vdots & & \vdots \\ \kappa\left(\mathbf{x}_N, \mathbf{x}_1\right) & \dots & \kappa\left(\mathbf{x}_N, \mathbf{x}_N\right) \end{bmatrix}, \tag{2.20}$$

with $K_{ij} = \kappa\left(\mathbf{x}_i, \mathbf{x}_j\right)$, and the *kernelised* α

$$\alpha = \left(K + \lambda I_N\right)^{-1} f. \tag{2.21}$$

In summary, we need to solve the linear system

$$\left(K + \lambda I_N\right)\alpha = f, \tag{2.22}$$

so that we can use (2.17) to get the *predicted response* $F(\mathbf{x}_{\text{new}})$ for a new unlabelled data point $\mathbf{x}_{\text{new}}$. This process is known as *kernel ridge regression*. The complexity for solving (2.22) directly is $\mathcal{O}(N^3)$. Therefore, we usually do not solve it that way, especially not in high-dimensional cases [3].

Remark 2.2. Above, we introduced the *kernel function* κ as the inner product (2.16). Later, we are going to define more general *kernels*. But designing an appropriate *kernel function* is not easily done. *Kernels* are supposed to reflect prior knowledge about the problem and its solution and hugely influence the quality of prediction [5]. We are going to realise that a universal *kernel function* does not exist. It always depends on the properties of the system and needs to be chosen for each problem individually to perform the *learning methods* optimally. In Chapter 3 we address the design of *kernel functions* in detail. Thereupon, we demonstrate the importance of that choice in Chapter 4.

2.2 Spectral Clustering

Spectral clustering is a clustering technique that divides data points into groups of similar behaviour [6]. Therefore, we define measures w_{ij} of similarity for all combinations of the data points $\mathbf{x}_i, \mathbf{x}_j \in \mathbb{R}^n$. For convenience this type of clustering is based on viewing the data represented in a *similarity graph*. The vertices in this graph correspond to the data points. Two nodes $\mathbf{x}_i$ and $\mathbf{x}_j$ are connected via an edge if $w_{ij} > 0$. In our case this means that we aim to find a partitioning of the graph into two clusters such that the weight of the edges across the clusters is minimal and the weight of the edges within a cluster is maximal. This corresponds to the expectation that data points which are similar to each other are associated with the same cluster.

2.2.1 The Graph Laplacian

Transforming a set of data points with pairwise distances into a graph can be done in several ways. The procedure we present here is based on von Luxburg [6]. We assume the *similarity graph* to be an undirected, fully connected, weighted graph, i. e. each edge between a pair of nodes $\mathbf{x}_i$ and $\mathbf{x}_j$ carries a weight $w_{ij} \geq 0$. We require $w_{ij} = w_{ji}$ and $w_{ii} = 0$ for all $i = 1, \ldots, N$, since data points are not connected to themselves. This yields the dense *adjacency matrix* $W = (w_{ij})_{i,j=1,\ldots,N}$. It is defined in the same manner as the *kernel matrix* K occurring in the *kernel ridge regression*, i. e. $w_{ij} = K_{ij} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$ for all $i, j = 1, \ldots, N$, $i \neq j$. Again, for details concerning

the design of the *kernel function* κ, we refer to Chapter 3. Moreover, we define the *degree* of a vertex corresponding to a data point $\mathbf{x}_i$ as

$$d_i = \sum_{j=1}^{N} w_{ij}, \tag{2.23}$$

such that we can create the diagonal *degree matrix*

$$D = \begin{bmatrix} d_1 & & & \\ & d_2 & & \\ & & \ddots & \\ & & & d_N \end{bmatrix}. \tag{2.24}$$

Now, we are ready to construct the so-called *unnormalised graph Laplacian*

$$L = D - W \in \mathbb{R}^{N \times N}. \tag{2.25}$$

This matrix has the following property which is going to be proved to be of great use.

Lemma 2.3. *The following equation holds for every vector $u \in \mathbb{R}^N$:*

$$u^T L u = \frac{1}{2} \sum_{i,j=1}^{N} w_{ij} \left(u_i - u_j\right)^2. \tag{2.26}$$

Proof. By (2.23) and (2.25) we have

$$\begin{aligned} u^T L u &= u^T D u - u^T W u \\ &= \sum_{i=1}^{N} d_i u_i^2 - \sum_{i,j=1}^{N} w_{ij} u_i u_j \\ &= \frac{1}{2} \left(\sum_{i=1}^{N} d_i u_i^2 - 2 \sum_{i,j=1}^{N} w_{ij} u_i u_j + \sum_{j=1}^{N} d_j u_j^2 \right) \\ &= \frac{1}{2} \left(\sum_{i,j=1}^{N} w_{ij} u_i^2 - 2 \sum_{i,j=1}^{N} w_{ij} u_i u_j + \sum_{i,j=1}^{N} w_{ij} u_j^2 \right) \\ &= \frac{1}{2} \left(\sum_{i,j=1}^{N} w_{ij} \left(u_i^2 - 2 u_i u_j + u_j^2\right) \right) \\ &= \frac{1}{2} \sum_{i,j=1}^{N} w_{ij} \left(u_i - u_j\right)^2. \end{aligned} \tag{2.27}$$

□

After having introduced the *unnormalised graph Laplacian* we define the *normalised graph Laplacian* now as

$$\begin{aligned} L_{\text{sym}} &= D^{-\frac{1}{2}} L D^{-\frac{1}{2}} = I_N - D^{-\frac{1}{2}} W D^{-\frac{1}{2}} \in \mathbb{R}^{N \times N} \\ L_{\text{rw}} &= D^{-1} L = I_N - D^{-1} W \in \mathbb{R}^{N \times N}. \end{aligned} \tag{2.28}$$

The first matrix is denoted by L_{sym} as it is symmetric. The second one is closely related to a random walk, which is why we denote it by L_{rw} [6]. Both matrices are referred to as *normalised graph Laplacians* in the literature. From now on we are going to limit our studies to the matrix L_{sym}. The following lemma is an analogue to Lemma 2.3.

Lemma 2.4. *The following equation holds for every vector $u \in \mathbb{R}^N$:*

$$u^T L_{sym} u = \frac{1}{2} \sum_{i,j=1}^{N} w_{ij} \left(\frac{u_i}{\sqrt{d_i}} - \frac{u_j}{\sqrt{d_j}} \right)^2. \tag{2.29}$$

Proof. This lemma can be proved analogously to Lemma 2.3. □

By construction, the *normalised graph Laplacian* L_{sym} is a symmetric matrix. Furthermore, it would really suit us if L_{sym} was positive semi-definite. This is proven by the following lemma [6].

Lemma 2.5. *The* normalised graph Laplacian L_{sym} *is symmetric and positive semi-definite.*

Proof. Since the matrices W, D and I_N are all symmetric, the symmetry of L_{sym} follows by Definition (2.28). However, its positive definiteness consequences directly from Lemma 2.4. By definition, $w_{ij} \geq 0$ holds for all $i, j = 1, \ldots, N$, $i \neq j$. This makes the right side of equation (2.29) bigger or equal zero and verifies $u^T L_{\text{sym}} u \geq 0$ for all $u \in \mathbb{R}^N$. □

2.2.2 Semi-Supervised Learning

Similar to the linear regression (2.2) we can define the problem

$$\hat{u} = \underset{u \in \mathbb{R}^N}{\arg min} \underbrace{\tfrac{1}{2} \|u - f\|_2^2 + \tfrac{\lambda}{2} u^T L_{\text{sym}} u}_{=l(u)}, \tag{2.30}$$

with $f \in \mathbb{R}^N$ incorporating the known labels y_i for a small number of the data points $\mathbf{x}_i \in \mathbb{R}^n$ and being zero everywhere else [7]. $\lambda > 0$ is conceived as a regularisation parameter. Since only parts of the information are used or known, this

process is called *semi-supervised learning*. Obviously, the first summand makes sure that the deviation between the solution and the given labels is as small as possible. Lemma 2.4 clarifies the relevance of the second summand. It ensures that two similar nodes are assigned to the same cluster and vice versa. This proves the convenience of problem (2.30).

For solving it, we rewrite

$$\begin{aligned} l(u) &= \tfrac{1}{2}\|u-f\|_2^2 + \tfrac{\lambda}{2}u^T L_{\text{sym}} u \\ &= \tfrac{1}{2}(u-f)^T(u-f) + \tfrac{\lambda}{2}u^T L_{\text{sym}} u \\ &= \tfrac{1}{2}\left(u^T u - 2u^T f + f^T f\right) + \tfrac{\lambda}{2}u^T L_{\text{sym}} u. \end{aligned} \tag{2.31}$$

Setting the differentiation of $l(u)$ zero

$$\begin{aligned} \nabla_u l(u) &= \tfrac{1}{2}(2u-2f) + \tfrac{\lambda}{2}(2L_{\text{sym}}u) \\ &= u - f + \lambda L_{\text{sym}} u \\ &\overset{!}{=} 0 \end{aligned} \tag{2.32}$$

gives

$$(I_N + \lambda L_{\text{sym}})\, u = f. \tag{2.33}$$

Taking the sign of the optimal solution $\hat{u} \coloneqq \operatorname{sign}(u) \in \mathbb{R}^N$ yields the predicted labels for all data points $\mathbf{x}_i \in \mathbb{R}^n$. To construct L_{sym} in accordance with (2.28), it remains to define the similarity function characterising the adjacency matrix W. We make use of *kernel functions* here, regarding that $w_{ii} = 0$ for all $i = 1, \ldots, N$. As already mentioned in Section 2.1.2, the right choice for this function is not obvious. This is what we examine in the next chapter.

3 Kernel Functions

We learned in Chapter 2 that *kernel functions* come into play whenever an inner product occurs and the similarity between data points needs to be measured. Thereby, the data is embedded into a new *feature space* such that non-linear relations between features can be modelled in a linear way as illustrated in Figure 3.1 [1]. This makes computations more efficient.

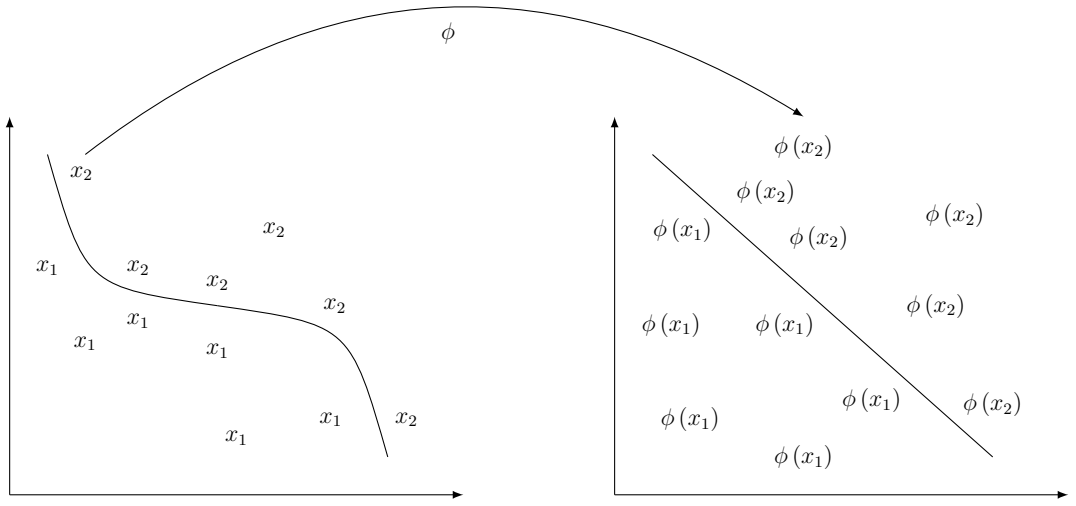

Figure 3.1: Embedding data into a *feature space* by applying an *embedding map* ϕ

But the right choice of a kernel is by no means trivial. In this chapter we present several *kernel functions* [1]. Our aim is to work out their differences. The best choice for a given set of data points is determined in the following chapters. For that, the computational efficiency, the dimension of the input and the quality of the prediction are taken into account.

3.1 Basics

Following Definition (2.1), our data are represented by feature vectors $\mathbf{x}_i \in \mathbb{R}^n$. After introducing an *embedding map*

$$\phi : \mathbf{x} \in \mathbb{R}^n \longmapsto \phi(\mathbf{x}) \in \mathbb{R}^{n'}, \tag{3.1}$$

with $n < n'$, we recode our pairs of data points from $(\mathbf{x}_i, y_i)$ to $(\phi(\mathbf{x}_i), y_i)$. This enables us to transform non-linear relations to linear ones. Remembering now the two learning methods explained in Chapter 2, we substitute the inner products ${\mathbf{x}_i}^T\mathbf{x}_j$ between two feature vectors $\mathbf{x}_i, \mathbf{x}_j \in \mathbb{R}^N$ by a *kernel function* κ with

$$\kappa\left(\mathbf{x}_i, \mathbf{x}_j\right) = \langle \phi(\mathbf{x}_i), \phi\left(\mathbf{x}_j\right)\rangle. \tag{3.2}$$

One of the most basic kernels is the *derived polynomial kernel*

$$\kappa_p^{\text{pol}}\left(\mathbf{x}_i, \mathbf{x}_j\right) = p\left(\kappa\left(\mathbf{x}_i, \mathbf{x}_j\right)\right) \tag{3.3}$$

for a kernel κ as in (3.2), with $p(.)$ being any polynomial with positive coefficients. From there we can define the special case

$$\kappa_d^{\text{pol}}\left(\mathbf{x}_i, \mathbf{x}_j\right) = \left(\langle \mathbf{x}_i, \mathbf{x}_j\rangle + R\right)^d, \tag{3.4}$$

where R and d are chosen parameters. By the binomial theorem, the polynomial kernel κ_d^{pol} can be expanded to

$$\kappa_d^{\text{pol}}\left(\mathbf{x}_i, \mathbf{x}_j\right) = \sum_{s=0}^{d} \binom{d}{s} R^{d-s} \langle \mathbf{x}_i, \mathbf{x}_j\rangle^s. \tag{3.5}$$

With $\alpha_s = \binom{d}{s} R^{d-s}$ and $\hat{\kappa}_s^{\text{pol}}\left(\mathbf{x}_i, \mathbf{x}_j\right) = \langle \mathbf{x}_i, \mathbf{x}_j\rangle^s$, this can be rewritten as

$$\kappa_d^{\text{pol}}\left(\mathbf{x}_i, \mathbf{x}_j\right) = \sum_{s=0}^{d} \alpha_s \hat{\kappa}_s^{\text{pol}}\left(\mathbf{x}_i, \mathbf{x}_j\right). \tag{3.6}$$

Obviously, the features of the kernel κ_d^{pol} are formed from the features of all components in the sum. Here, α_s serves as a reweighting of the polynomial kernels $\hat{\kappa}_s^{\text{pol}}\left(\mathbf{x}_i, \mathbf{x}_j\right) = \langle \mathbf{x}_i, \mathbf{x}_j\rangle^s$ for $s = 0 \ldots, d$. Since α_s is smaller for large values of s, the weight of the higher-order polynomials $\hat{\kappa}_s^{\text{pol}}\left(\mathbf{x}_i, \mathbf{x}_j\right)$ decreases with increasing R.
Next, we consider the feature map

$$\phi_A(\mathbf{x}_i) = \prod_{l \in A} \mathbf{x}_i^l, \tag{3.7}$$

which multiplies the input features for all elements of the subset $A \subseteq \{1, \ldots, n\}$. This yields the embedding

$$\phi : \mathbf{x} \longmapsto \left(\phi_A\left(\mathbf{x}\right)\right)_{A \subseteq \{1,\ldots,n\}}, \tag{3.8}$$

with 2^n possible subsets $A \subseteq \{1, \ldots, n\}$ and the so-called *all-subsets kernel* $\kappa_{\subseteq}\left(\mathbf{x}_i, \mathbf{x}_j\right) = \langle \phi\left(\mathbf{x}_i\right), \phi\left(\mathbf{x}_j\right) \rangle$. Applying the distributive law, we can write

$$\begin{aligned} \kappa_{\subseteq}\left(\mathbf{x}_i, \mathbf{x}_j\right) &= \langle \phi\left(\mathbf{x}_i\right), \phi\left(\mathbf{x}_j\right) \rangle \\ &= \sum_{A \subseteq \{1,\ldots,n\}} \phi_A\left(\mathbf{x}_i\right) \phi_A\left(\mathbf{x}_j\right) \\ &= \sum_{A \subseteq \{1,\ldots,n\}} \prod_{l \in A} \mathbf{x}_i^l \mathbf{x}_j^l \\ &= \prod_{l=1}^{n} \left(1 + \mathbf{x}_i^l \mathbf{x}_j^l\right) \end{aligned} \tag{3.9}$$

Example 3.1. We are given two feature vectors $\mathbf{x}_i = (\mathbf{x}_i^1, \mathbf{x}_i^2)$ and $\mathbf{x}_j = \left(\mathbf{x}_j^1, \mathbf{x}_j^2\right)$. For the subset $A \subseteq \{1, 2\}$ we clearly have $A \in \{\{\varnothing\}, \{1\}, \{2\}, \{1, 2\}\}$, so that

$$\begin{aligned} \kappa_{\subseteq}\left(\mathbf{x}_i, \mathbf{x}_j\right) &= \sum_{A \subseteq \{1,\ldots,n\}} \prod_{l \in A} \mathbf{x}_i^l \mathbf{x}_j^l \\ &= 1 + \mathbf{x}_i^1 \mathbf{x}_j^1 + \mathbf{x}_i^2 \mathbf{x}_j^2 + \mathbf{x}_i^1 \mathbf{x}_j^1 \mathbf{x}_i^2 \mathbf{x}_j^2 \\ &= \left(1 + \mathbf{x}_i^1 \mathbf{x}_j^1\right)\left(1 + \mathbf{x}_i^2 \mathbf{x}_j^2\right) \\ &= \prod_{l=1}^{2} \left(1 + \mathbf{x}_i^l \mathbf{x}_j^l\right). \end{aligned} \tag{3.10}$$

This general introduction to the theory of kernels should be enough for the purpose of this thesis. In the following sections we go into more detail for some selected kernel functions [1].

3.2 The Gaussian Kernel

The *Gaussian kernel* [1] is the most widely used *kernel function*. It is defined as

$$\kappa\left(\mathbf{x}_i, \mathbf{x}_j\right) = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|_2^2}{\sigma^2}\right), \tag{3.11}$$

where $\|\mathbf{x}_i - \mathbf{x}_j\|^2$ is the squared Euclidean distance between the feature vectors. $\sigma > 0$ is a *scaling parameter* that is often tailored to the task by hand. It acts similar as the degree d in (3.4).

It is not immediately obvious, how the *Gaussian kernel* (3.11) fits into the basic

definitions from the previous section. Actually, the exponential function is a limit of kernels, since it can be approximated by polynomials with positive coefficients. Therefore, the exponential of a kernel yields a kernel by Definition (3.3). Then, we obtain the *Gaussian kernel* (3.11) by normalising the kernel

$$\kappa\left(\mathbf{x}_i, \mathbf{x}_j\right) = \exp\left(\langle \tfrac{\sqrt{2}}{\sigma}\mathbf{x}_i, \tfrac{\sqrt{2}}{\sigma}\mathbf{x}_j\rangle\right) = \exp\left(\frac{\langle \mathbf{x}_i, \mathbf{x}_j\rangle}{\frac{1}{2}\sigma^2}\right), \tag{3.12}$$

where $\phi\left(\mathbf{x}\right) = \frac{\sqrt{2}}{\sigma}\mathbf{x}$. We refer to Proposition 3.24 in Shawe-Taylor et al. [1] for a great explanation of the details.

3.3 The ANOVA Kernel

In the previous section we defined the *Gaussian kernel*. We will see in Chapter 4 that it is a good choice for low-dimensional applications. However, it is very expensive for large dimensions. This is due to the complexity $\mathcal{O}\left(N^3\right)$ for solving (2.22) directly. We are in need of a kernel, which is designed in such a way that the linear system can be solved more efficiently. At the same time the prediction quality shall not be (drastically) reduced. Moreover, we wish for more freedom regarding the inclusion of the monomials compared to previous kernels. A remedy is to work with the so-called *ANOVA kernel* [1]. In this chapter we analyse this kernel closely and check in the further course whether it meets our expectations.

In Section 3.1 we defined two of the most basic kernel functions: the *polynomial kernel* and the *all-subsets kernel*. It is possible to compute them recursively indeed. But we are limited regarding the choice of the considered features and the weighting. The *polynomial kernel* (3.4) is restricted to using all monomials of degree d. Since a weighting scheme can just depend on the parameter R, the best we achieve are all monomials of degree up to d. Whereas the *all-subsets kernel* (3.9) uses literally all monomials corresponding to all subsets of the n features in the input space. The *ANOVA kernel* is defined quite similar to this with the difference that the considered subsets are restricted to a given cardinality d [1], the *degree* of the *ANOVA kernel*. It thereby provides more freedom in determining the set of monomials. In comparison to the *polynomial kernel*, repeated coordinates are excluded. This yields the embedding

$$\phi_d : \mathbf{x} \longmapsto \left(\phi_A\left(\mathbf{x}\right)\right)_{|A|=d} \tag{3.13}$$

of the ANOVA kernel of degree d, with

$$\phi_A(\mathbf{x}_i) = \prod_{l \in A} \mathbf{x}_i^l \tag{3.14}$$

and results in the inner product

$$\begin{aligned} \kappa_d(\mathbf{x}_i, \mathbf{x}_j) &= \langle \phi_d(\mathbf{x}_i), \phi_d(\mathbf{x}_j) \rangle \\ &= \sum_{|A|=d} \phi_A(\mathbf{x}_i)\, \phi_A(\mathbf{x}_j) \\ &= \sum_{1 \le l_1 < l_2 < \cdots < l_d \le n} \left(\mathbf{x}_i^{l_1} \mathbf{x}_j^{l_1}\right) \left(\mathbf{x}_i^{l_2} \mathbf{x}_j^{l_2}\right) \ldots \left(\mathbf{x}_i^{l_d} \mathbf{x}_j^{l_d}\right) \\ &= \sum_{1 \le l_1 < l_2 < \cdots < l_d \le n} \prod_{t=1}^{d} \mathbf{x}_i^{l_t} \mathbf{x}_j^{l_t}. \end{aligned} \tag{3.15}$$

It obviously consists of a sum of $\binom{n}{d}$ products, since this is the number of possible d-order subsets of $\{1, \ldots, n\}$. Hence, computing this explicitly requires $\mathcal{O}\left(d\binom{n}{d}\right)$ operations. As motivated before, we wish to evaluate this kernel faster by considering a recursive method of computation. Using the notation $\mathbf{x}_i^{1:m} = (\mathbf{x}_i^1, \ldots, \mathbf{x}_i^m)$, $m \ge 1$, we introduce the *ANOVA kernel* of degree $s \ge 0$ ($|A| = s$) with inputs restricted to the first m coordinates

$$\kappa_s^m(\mathbf{x}_i, \mathbf{x}_j) = \kappa_s\left(\mathbf{x}_i^{1:m}, \mathbf{x}_j^{1:m}\right), \tag{3.16}$$

where κ_s follows (3.15). This can be written recursively as

$$\kappa_s^m(\mathbf{x}_i, \mathbf{x}_j) = \left(\mathbf{x}_i^m \mathbf{x}_j^m\right) \kappa_{s-1}^{m-1}(\mathbf{x}_i, \mathbf{x}_j) + \kappa_s^{m-1}(\mathbf{x}_i, \mathbf{x}_j). \tag{3.17}$$

The idea is to divide the considered subsets of features into two groups: those features that contain $\mathbf{x}_i^m$ and those that do not. The first group includes all subsets of size $s-1$ restricted to $\mathbf{x}_i^{1:m-1}$. It represents all s-order subsets that contain $\mathbf{x}_i^m$. The second one contains all subsets of size s which are restricted to $\mathbf{x}_i^{1:m-1}$. Those obviously do not contain $\mathbf{x}_i^m$. For $m < s$ it is impossible to find a subset of size s. Therefore, $\kappa_s^m(\mathbf{x}_i, \mathbf{x}_j) = 0$ if $m < s$. For $s = 0$ the only valid subset is the empty set, so that $\kappa_0^m(\mathbf{x}_i, \mathbf{x}_j) = 1$. Altogether this yields the *naive ANOVA recursion*

$$\begin{aligned} &\kappa_0^m(\mathbf{x}_i, \mathbf{x}_j) = 1, \text{ if } m \ge 0, \\ &\kappa_s^m(\mathbf{x}_i, \mathbf{x}_j) = 0, \text{ if } m < s, \\ &\kappa_s^m(\mathbf{x}_i, \mathbf{x}_j) = \left(\mathbf{x}_i^m \mathbf{x}_j^m\right) \kappa_{s-1}^{m-1}(\mathbf{x}_i, \mathbf{x}_j) + \kappa_s^{m-1}(\mathbf{x}_i, \mathbf{x}_j). \end{aligned} \tag{3.18}$$

Implementing the Recursion (3.18) seems useful in theory. But it is very inefficient. Let the function $T(m, s)$ denote the cost of calculating $\kappa_s^m(\mathbf{x}_i, \mathbf{x}_j)$ using the *naive*

ANOVA recursion (3.18). Then we can estimate the number of operations as follows:

$$\begin{aligned} T(m,s) &= T(m-1,s) + T(m-1,s-1) + 3 \\ &> T(m-1,s) + T(m-1,s-1). \end{aligned} \tag{3.19}$$

We have $T(m,s) = 1$ for $m < s$ and $T(m,0) = 1$. For both of these special cases the inequality

$$T(m,s) \geq \binom{m}{s} \tag{3.20}$$

holds true. Using this induction hypothesis and applying (3.19) yields

$$\begin{aligned} T(m,s) &> T(m-1,s) + T(m-1,s-1) \\ &\geq \binom{m-1}{s} + \binom{m-1}{s-1} \\ &= \binom{m}{s}. \end{aligned} \tag{3.21}$$

Therefore, computing the kernel $\kappa_d(\mathbf{x}_i, \mathbf{x}_j)$ requires at least $\mathcal{O}\left(\binom{n}{d}\right)$ operations, which is still not really satisfying and far away from best-case complexity. This is due to the fact that many of the same computations are repeated again and again. Thus, saving the values of $\kappa_s^m(\mathbf{x}_i, \mathbf{x}_j)$ as they are computed is the key to success in drastically reducing the overall computational complexity. This process is called *dynamic programming*. It can be realised using a dynamic programming table:

DP	$m=1$	2	...	n
$s=0$	1	1	...	1
1	$\mathbf{x}_i^1\mathbf{x}_j^1$	$\mathbf{x}_i^1\mathbf{x}_j^1 + \mathbf{x}_i^2\mathbf{x}_j^2$	...	$\sum_{l=1}^{n} \mathbf{x}_i^l\mathbf{x}_j^l$
2	0	$\kappa_2^2(\mathbf{x}_i, \mathbf{x}_j)$	...	$\kappa_2^n(\mathbf{x}_i, \mathbf{x}_j)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
d	0	0	...	$\kappa_d^n(\mathbf{x}_i, \mathbf{x}_j)$

Table 3.1: Dynamic Programming Evaluation

Using the *ANOVA recursion* we take each row in turn from left to right. By (3.18) one particular entry depends on two other entries. Both of them are already available in our table: the one diagonally above to its left and the one immediately to its left. The bottom rightmost entry corresponds to our Definition (3.15) of the *ANOVA kernel* of degree d. Moreover, the sum of all entries in the final column yields the *all-subsets kernel*

$$\kappa_{\subseteq}(\mathbf{x}_i, \mathbf{x}_j) = \kappa_{\leq n}(\mathbf{x}_i, \mathbf{x}_j) = \sum_{s=0}^{n} \kappa_s(\mathbf{x}_i, \mathbf{x}_j). \tag{3.22}$$

In comparison with formula (3.9), this method is much less efficient though.

As mentioned earlier, we wish for more freedom concerning the choice of the considered monomials. For this purpose we introduce a weighting factor $a_i \geq 0$, which enables us to downplay or emphasise certain features. In addition, the components $\mathbf{x}_i^l \mathbf{x}_j^l$ occurring in both the *all-subsets* and the *ANOVA kernel* can be extended to a *base kernel*

$$\kappa_l^{\text{base}}\left(\mathbf{x}_i, \mathbf{x}_j\right) = \mathbf{x}_i^l \mathbf{x}_j^l. \tag{3.23}$$

Applying the reweighting scheme to this yields

$$\kappa_l^{\text{base}}\left(\mathbf{x}_i, \mathbf{x}_j\right) = a_l \mathbf{x}_i^l \mathbf{x}_j^l. \tag{3.24}$$

Here, the l-th *base kernel* is only dependent on the l-th feature. But we do not need to restrict ourselves to that. That kernel might also depend on some other coordinate, on a *window* of several features or even on all of them. More generally, this yields *base kernels* $\kappa_1^{\text{base}}\left(\mathbf{x}_i, \mathbf{x}_j\right), \ldots, \kappa_n^{\text{base}}\left(\mathbf{x}_i, \mathbf{x}_j\right)$ and

$$\kappa_d\left(\mathbf{x}_i, \mathbf{x}_j\right) = \sum_{1 \leq l_1 < l_2 < \cdots < l_d \leq n} \prod_{t=1}^{d} \kappa_{l_t}^{\text{base}}\left(\mathbf{x}_i, \mathbf{x}_j\right) \tag{3.25}$$

as a generalised version of the *ANOVA kernel* [1]. In the following subsections, we discuss several ideas for choosing these *base kernels*. Speculating why certain choices might not work well, we construct remedies which promise to perform better in theory. We compare their performance in Chapter 4. Actually, we restrict ourselves to using the *Gaussian kernel* for all the *base kernels*. Our object of investigation is the choice of coordinates, the base kernels depend on, and its influence on the overall performance in prediction.

3.3.1 Simple Base Kernels

We start investigating the *ANOVA kernel* using *base kernels*, which only depend on one coordinate, such that

$$\kappa\left(\mathbf{x}_i, \mathbf{x}_j\right) = \sum_{l=1}^{n} \exp\left(-\frac{\|\mathbf{x}_i^l - \mathbf{x}_j^l\|_2^2}{\sigma^2}\right)^d. \tag{3.26}$$

This kernel describes a very special case of (3.25) and is called the *simple Gaussian ANOVA kernel* from now on [8]. The corresponding kernel matrix

$$K = \begin{bmatrix} \kappa\left(\mathbf{x}_1, \mathbf{x}_1\right) & \ldots & \kappa\left(\mathbf{x}_1, \mathbf{x}_N\right) \\ \vdots & & \vdots \\ \kappa\left(\mathbf{x}_N, \mathbf{x}_1\right) & \ldots & \kappa\left(\mathbf{x}_N, \mathbf{x}_N\right) \end{bmatrix} \tag{3.27}$$

is built using the kernel function (3.26), what requires all features of both data points $\mathbf{x}_i$ and $\mathbf{x}_j$. We will later see, why this leads to a high computational complexity. Since (3.26) sums the exponential up over all features, we can divide the kernel function into n summands, such that

$$\kappa\left(\mathbf{x}_i, \mathbf{x}_j\right) = \kappa_1\left(\mathbf{x}_i^1, \mathbf{x}_j^1\right) + \cdots + \kappa_n\left(\mathbf{x}_i^n, \mathbf{x}_j^n\right) \tag{3.28}$$

with

$$\kappa_l\left(\mathbf{x}_i^l, \mathbf{x}_j^l\right) = \exp\left(-\frac{\|\mathbf{x}_i^l - \mathbf{x}_j^l\|_2^2}{\sigma^2}\right)^d \tag{3.29}$$

and

$$K = K_1 + \cdots + K_n \tag{3.30}$$

with

$$K_l = \begin{bmatrix} \kappa_l\left(\mathbf{x}_1^l, \mathbf{x}_1^l\right) & \dots & \kappa_l\left(\mathbf{x}_1^l, \mathbf{x}_N^l\right) \\ \vdots & & \vdots \\ \kappa_l\left(\mathbf{x}_N^l, \mathbf{x}_1^l\right) & \dots & \kappa_l\left(\mathbf{x}_N^l, \mathbf{x}_N^l\right) \end{bmatrix} \tag{3.31}$$

for $l = 1, \ldots, n$. Representing the kernel matrix K as a sum of n matrices K_l is a nice way to examine relations between the features and their influence on the label. It seems reasonable to take into account all features since incorporating all information can impossibly be detrimental. This can prove to be a logical fallacy for designated data sets. They might possess features, which do not interact at all with the last feature - the label, we aim to predict. If these features are being involved in the learning process though, relations might be detected, which do not exist in reality. This worsens the precision in prediction. Instead, examining the influence of each feature separately might be a good idea. Now, we do not consider the combination of all features as in (3.28), but their individual influence. This yields n 1-*dimensional simple Gaussian ANOVA kernels*

$$\kappa\left(\mathbf{x}_i, \mathbf{x}_j\right) = \kappa_l\left(\mathbf{x}_i^l, \mathbf{x}_j^l\right) = \exp\left(-\frac{\|\mathbf{x}_i^l - \mathbf{x}_j^l\|_2^2}{\sigma^2}\right)^d \tag{3.32}$$

for $l = 1, \ldots, n$. Since these kernels restrict themselves to investigating the relation between one feature and the label, we do not expect a high precision in prediction. We can identify features which perform markedly bad though and take them as irrelevant for predicting the labels.

3.3.2 Windowed Base Kernels

So far it is not clear if the kernel functions we formulated in the previous subsection can produce satisfying results. Taking into account all features might fail due to attributing more influence to some features than they actually have. And examining each feature's importance separately neglects relevant relations between them. Instead, selecting some coordinates and analysing their combined influence sounds more promising. We implement this by selecting 3 coordinates, which embody the input for the kernel function. This results in $\binom{n}{3}$ possibilities for choosing the so-called *window* of 3 features. The motive for using 3-dimensional inputs is down to the *Nonequispaced Fast Fourier Transform*, see Section 5.3. This method runs our computations very efficiently as long as the input dimension is smaller than 4. Since we aim to examine the combined relation of as many coordinates as possible, we choose our *windows* to be 3-dimensional and define the *windowed Gaussian ANOVA kernel*

$$\kappa\left(\mathbf{x}_i, \mathbf{x}_j\right) = \kappa\left(\mathbf{x}_i^{\text{window}}, \mathbf{x}_j^{\text{window}}\right) = \exp\left(-\frac{\|\mathbf{x}_i^{\text{window}} - \mathbf{x}_j^{\text{window}}\|_2^2}{\sigma^2}\right)^d, \tag{3.33}$$

with $\binom{n}{3}$ possibilities for defining $\mathbf{x}_i^{\text{window}} = \begin{bmatrix}\mathbf{x}_i^{w_1} & \mathbf{x}_i^{w_2} & \mathbf{x}_i^{w_3}\end{bmatrix}^T$ and $\mathbf{x}_j^{\text{window}} = \begin{bmatrix}\mathbf{x}_j^{w_1} & \mathbf{x}_j^{w_2} & \mathbf{x}_j^{w_3}\end{bmatrix}^T$ each. Analogously to what we did in (3.28), we can combine these kernels by summarising over several choices of (3.33). For the sake of convenience we demonstrate this for kernel functions with consecutive *windows*. We distinguish two different cases. In the first case the *windows* of consecutive coordinates do not overlap each other. Provided that 3 is a factor of n this yields a sum of $\frac{n}{3}$ matrices

$$K_l = \begin{bmatrix} \kappa\left(\mathbf{x}_1^{\text{window}_l}, \mathbf{x}_1^{\text{window}_l}\right) & \dots & \kappa\left(\mathbf{x}_1^{\text{window}_l}, \mathbf{x}_N^{\text{window}_l}\right) \\ \vdots & & \vdots \\ \kappa\left(\mathbf{x}_N^{\text{window}_l}, \mathbf{x}_1^{\text{window}_l}\right) & \dots & \kappa\left(\mathbf{x}_N^{\text{window}_l}, \mathbf{x}_N^{\text{window}_l}\right) \end{bmatrix} \tag{3.34}$$

with κ as defined in (3.33) and the appropriate *window* $\mathbf{x}_i^{\text{window}_l} = \begin{bmatrix}\mathbf{x}_i^{3l-2} & \mathbf{x}_i^{3l-1} & \mathbf{x}_i^{3l}\end{bmatrix}^T$.

Example 3.2. Let a data set with N data points $\mathbf{x}_i \in \mathbb{R}^6$ be given. Then, the kernel matrix can be represented by a sum of $\frac{6}{3} = 2$ matrices

$$K = K_1 + K_2, \tag{3.35}$$

where

$$K_1 = \begin{bmatrix} \kappa\left(\mathbf{x}_1^{\text{window}_1}, \mathbf{x}_1^{\text{window}_1}\right) & \dots & \kappa\left(\mathbf{x}_1^{\text{window}_1}, \mathbf{x}_N^{\text{window}_1}\right) \\ \vdots & & \vdots \\ \kappa\left(\mathbf{x}_N^{\text{window}_1}, \mathbf{x}_1^{\text{window}_1}\right) & \dots & \kappa\left(\mathbf{x}_N^{\text{window}_1}, \mathbf{x}_N^{\text{window}_1}\right) \end{bmatrix},$$
$$K_2 = \begin{bmatrix} \kappa\left(\mathbf{x}_1^{\text{window}_2}, \mathbf{x}_1^{\text{window}_2}\right) & \dots & \kappa\left(\mathbf{x}_1^{\text{window}_2}, \mathbf{x}_N^{\text{window}_2}\right) \\ \vdots & & \vdots \\ \kappa\left(\mathbf{x}_N^{\text{window}_2}, \mathbf{x}_1^{\text{window}_2}\right) & \dots & \kappa\left(\mathbf{x}_N^{\text{window}_2}, \mathbf{x}_N^{\text{window}_2}\right) \end{bmatrix}, \tag{3.36}$$

with $\mathbf{x}_i^{\text{window}_1} = \begin{bmatrix} \mathbf{x}_i^1 & \mathbf{x}_i^2 & \mathbf{x}_i^3 \end{bmatrix}^T$ and $\mathbf{x}_i^{\text{window}_2} = \begin{bmatrix} \mathbf{x}_i^4 & \mathbf{x}_i^5 & \mathbf{x}_i^6 \end{bmatrix}^T$ for all $i = 1, \dots, N$.

It suggests itself that the *windows* of consecutive coordinates do overlap each other in the second case. This time the kernel matrix K results from a sum of $n-2$ matrices K_l. Their definition follows (3.34) with varying *windows* $\mathbf{x}_i^{\text{window}_l} = \begin{bmatrix} \mathbf{x}_i^l & \mathbf{x}_i^{l+1} & \mathbf{x}_i^{l+2} \end{bmatrix}^T$ though.

Example 3.3. Let a data set with N data points $\mathbf{x}_i \in \mathbb{R}^6$ be given. Then, the kernel matrix can be represented by a sum of $6-2=4$ matrices

$$K = K_1 + K_2 + K_3 + K_4, \tag{3.37}$$

where

$$K_1 = \begin{bmatrix} \kappa\left(\mathbf{x}_1^{\text{window}_1}, \mathbf{x}_1^{\text{window}_1}\right) & \dots & \kappa\left(\mathbf{x}_1^{\text{window}_1}, \mathbf{x}_N^{\text{window}_1}\right) \\ \vdots & & \vdots \\ \kappa\left(\mathbf{x}_N^{\text{window}_1}, \mathbf{x}_1^{\text{window}_1}\right) & \dots & \kappa\left(\mathbf{x}_N^{\text{window}_1}, \mathbf{x}_N^{\text{window}_1}\right) \end{bmatrix},$$
$$\vdots$$
$$K_4 = \begin{bmatrix} \kappa\left(\mathbf{x}_1^{\text{window}_4}, \mathbf{x}_1^{\text{window}_4}\right) & \dots & \kappa\left(\mathbf{x}_1^{\text{window}_4}, \mathbf{x}_N^{\text{window}_4}\right) \\ \vdots & & \vdots \\ \kappa\left(\mathbf{x}_N^{\text{window}_4}, \mathbf{x}_1^{\text{window}_4}\right) & \dots & \kappa\left(\mathbf{x}_N^{\text{window}_4}, \mathbf{x}_N^{\text{window}_4}\right) \end{bmatrix}, \tag{3.38}$$

with $\mathbf{x}_i^{\text{window}_1} = \begin{bmatrix} \mathbf{x}_i^1 & \mathbf{x}_i^2 & \mathbf{x}_i^3 \end{bmatrix}^T$, $\mathbf{x}_i^{\text{window}_2} = \begin{bmatrix} \mathbf{x}_i^2 & \mathbf{x}_i^3 & \mathbf{x}_i^4 \end{bmatrix}^T$,
$\mathbf{x}_i^{\text{window}_3} = \begin{bmatrix} \mathbf{x}_i^3 & \mathbf{x}_i^4 & \mathbf{x}_i^5 \end{bmatrix}^T$ and $\mathbf{x}_i^{\text{window}_4} = \begin{bmatrix} \mathbf{x}_i^4 & \mathbf{x}_i^5 & \mathbf{x}_i^6 \end{bmatrix}^T$ for all $i = 1, \dots, N$.

Comparing the last two cases we realise that they are basically identical. They only differ in the *window* of considered coordinates. Up to now we restricted ourselves to consecutive ones for convenience. But recognising that the kernel function will be the same however the index set for the *window* is chosen, enables us to define the *generalised Gaussian ANOVA kernel*

$$\kappa\left(\mathbf{x}_i, \mathbf{x}_j\right) = \kappa\left(\mathbf{x}_i^I, \mathbf{x}_j^I\right) = \exp\left(-\frac{\|\mathbf{x}_i^I - \mathbf{x}_j^I\|_2^2}{\sigma^2}\right)^d \tag{3.39}$$

for an index set $I = \{w_1, w_2, w_3\} \in \{1, \dots, n\}^3$, with corresponding $\mathbf{x}_i^I = \begin{bmatrix} \mathbf{x}_i^{w_1} & \mathbf{x}_i^{w_2} & \mathbf{x}_i^{w_3} \end{bmatrix}^T$ and $\mathbf{x}_j^I = \begin{bmatrix} \mathbf{x}_j^{w_1} & \mathbf{x}_j^{w_2} & \mathbf{x}_j^{w_3} \end{bmatrix}^T$. Again several choices for these kernels can be combined by summing up kernel matrices

$$K_l = \begin{bmatrix} \kappa\left(\mathbf{x}_1^{I_l}, \mathbf{x}_1^{I_l}\right) & \dots & \kappa\left(\mathbf{x}_1^{I_l}, \mathbf{x}_N^{I_l}\right) \\ \vdots & & \vdots \\ \kappa\left(\mathbf{x}_N^{I_l}, \mathbf{x}_1^{I_l}\right) & \dots & \kappa\left(\mathbf{x}_N^{I_l}, \mathbf{x}_N^{I_l}\right) \end{bmatrix}, \tag{3.40}$$

where $\mathbf{x}_i^{I_l} = \begin{bmatrix} \mathbf{x}_i^{w_{1_l}} & \mathbf{x}_i^{w_{2_l}} & \mathbf{x}_i^{w_{3_l}} \end{bmatrix}^T$ for all $i = 1, \dots, N$.

Example 3.4. Let a data set with N data points $\mathbf{x}_i \in \mathbb{R}^6$ and index sets $I_1 = \{1, 4, 5\}$ and $I_2 = \{2, 3, 6\}$ be given. Then, the kernel matrix can be represented by a sum of 2 matrices

$$K = K_1 + K_2, \tag{3.41}$$

where

$$\begin{aligned} K_1 &= \begin{bmatrix} \kappa\left(\mathbf{x}_1^{I_1}, \mathbf{x}_1^{I_1}\right) & \dots & \kappa\left(\mathbf{x}_1^{I_1}, \mathbf{x}_N^{I_1}\right) \\ \vdots & & \vdots \\ \kappa\left(\mathbf{x}_N^{I_1}, \mathbf{x}_1^{I_1}\right) & \dots & \kappa\left(\mathbf{x}_N^{I_1}, \mathbf{x}_N^{I_1}\right) \end{bmatrix}, \\ K_2 &= \begin{bmatrix} \kappa\left(\mathbf{x}_1^{I_2}, \mathbf{x}_1^{I_2}\right) & \dots & \kappa\left(\mathbf{x}_1^{I_2}, \mathbf{x}_N^{I_2}\right) \\ \vdots & & \vdots \\ \kappa\left(\mathbf{x}_N^{I_2}, \mathbf{x}_1^{I_2}\right) & \dots & \kappa\left(\mathbf{x}_N^{I_2}, \mathbf{x}_N^{I_2}\right) \end{bmatrix}, \end{aligned} \tag{3.42}$$

with $\mathbf{x}_i^{I_1} = \begin{bmatrix} \mathbf{x}_i^1 & \mathbf{x}_i^4 & \mathbf{x}_i^5 \end{bmatrix}^T$ and $\mathbf{x}_i^{I_2} = \begin{bmatrix} \mathbf{x}_i^2 & \mathbf{x}_i^3 & \mathbf{x}_i^6 \end{bmatrix}^T$ for all $i = 1, \dots, N$.

3.3.3 Tuning the Parameters

Recapitulating the kernel functions we defined in the previous sections, we notice that all of them had one attribute in common: the scaling parameter σ. Aside from $\sigma > 0$ we do not know anything about it so far. This subsection shall bring light into the darkness. It serves as an instruction for choosing σ and all other occurring parameters.

We aim to set σ equal to the value, which minimises the prediction error. However, varying choices for this scaling parameter can lead to significantly different results. Hence, failing in identifying the optimal parameter can tremendously decrease the generalisation performance and needs to be prevented. This so-called *model selection* problem can among others be solved using the *cross-validation* based method. It combines a thorough grid search over the parameter space with *cross-validation*

on each candidate parameter [9, 10].
In detail, *k-fold cross-validation* describes a method, which randomly divides the data $\mathcal{D}$ into k mutually exclusive subsets $\mathcal{D}_1, \ldots, \mathcal{D}_k$ - the *folds*. They are of approximately same size. For each *fold* $t \in \{1, \ldots, k\}$ our model is performed on $\mathcal{D} \setminus \mathcal{D}_t$ and validated on $\mathcal{D}_t$, what yields k rounds of execution [11, 12]. Especially for classification problems using several parameters, this method is computationally expensive though. Performing grid search with a 20×20 mesh of parameter combinations for instance requires 400 trials of *cross-validation* [10]. We remember the previous section, where all representations of the *ANOVA kernel* include the degree d. This parameter needs to be chosen in addition to σ. Moreover, we recall λ from Chapter 2, which occurs for both learning methods in the definition of the *predicted response*. Thus, a combination of 3 parameters has to be chosen in total. This being the case, the *model selection* method described above seems slightly too complex and expensive for now.
In Subsection 3.3.2 we defined the *generalised Gaussian ANOVA kernel* (3.39), through which all previous *ANOVA kernels* can be represented. Looking just at the basic structure we have

$$\kappa\left(\mathbf{x}_i, \mathbf{x}_j\right) = \left(\exp\left(-\frac{\|.\|_2^2}{\sigma^2}\right)\right)^d = e^{\left(-\frac{\|.\|_2^2}{\sigma^2}\right)^d} = e^{\left(-\frac{\|.\|_2^2}{\sigma^2}\right)d} = \exp\left(-\frac{d}{\sigma^2}\|.\|_2^2\right) \quad (3.43)$$

by the exponential rules. Obviously, the parameter d does nothing more than scaling σ. Without loss of generality we therefore set $d = 1$ and restrict ourselves to tuning σ. Having given a kernel function κ, we perform our learning model several times with changing values of σ. For the sake of convenience we build a loop over the powers of ten and save the one which yields the smallest prediction error. At this point we set λ to some value, which works well for all choices of σ. For several kernel functions the optimal scaling parameters differ significantly. Therefore, this process is trivial by no means. To the contrary it is crucial. Now only λ remains to be selected appropriately. Having determined the optimal scaling parameter σ we find a good choice for λ by simply trial and error. For sure this is not the best we can do. But it is sufficient enough at this point.

4 The Prediction Quality in Comparison

After having introduced the theory in the previous chapters, we want to demonstrate some results now. As part of this thesis, we created a Python code that implements both *learning methods* from Chapter 2. Applying it to exemplary data, we illustrate both learning processes and emphasise the importance of crucial steps, such as scaling the data or tuning the parameters.
For the kernel matrix K and the weight matrix W occurring in the *kernel ridge regression* and the *semi-supervised learning*, respectively, we determine the following definition:

$$\begin{aligned} K_{ij} &= \begin{cases} \exp\left(-\frac{\|\mathbf{x}_i^I - \mathbf{x}_j^I\|_2^2}{\sigma^2}\right) & i \neq j, \\ 1 & i = j, \end{cases} \\ w_{ij} &= \begin{cases} \exp\left(-\frac{\|\mathbf{x}_i^I - \mathbf{x}_j^I\|_2^2}{\sigma^2}\right) & i \neq j, \\ 0 & i = j, \end{cases} \end{aligned} \tag{4.1}$$

for all $i, j = 1, \ldots, N$, where I is the index set for the chosen window of 3 features. Obviously, all non-diagonal entries are defined as the *generalised Gaussian ANOVA kernel* (3.39).

Remark 4.1. In Chapter 2 we foreshadowed, that we usually solve none of the linear systems

$$\begin{aligned} (K + \lambda I_N)\,\alpha &= f \\ (I_N + \lambda L_{\text{sym}})\,u &= f \end{aligned} \tag{4.2}$$

directly. This is due to its computational complexity of $\mathcal{O}(N^3)$, what scales bad for large $N \in \mathbb{N}$. In this chapter, we want to illustrate first results for small data sets. Therefore, the computational complexity is of little relevance yet. Hence, the linear systems (4.2) are solved as usual within our Python codes. However, the *numpy.linalg.solve* function does not offer itself to be used, because it requires matrices to have full rank. Since we cannot guarantee that for our matrices $K + \lambda I_N$

and $I_N + \lambda L_{\text{sym}}$, we need an alternative and use the *scipy.sparse.linalg.cg* function instead. For details concerning the *CG-method*, we refer to Section 5.2.

We start analysing the *Cryotherapy Data Set*, which "contains information about wart treatment results of 90 patients using cryotherapy" [13]. Each of those 90 patients are assigned 7 attributes, which are represented by a *feature vector* $\mathbf{x}_i \in \mathbb{R}^6$ and the corresponding label $y_i \in \{-1, 1\}$. Figure 4.1 shows the first 10 rows, i. e. the first 10 patients of the *Cryotherapy Data Set*, with "NoW" denoting the number of warts and "Result" indicating if the therapy was successful for the particular patient.

Sex	Age	Time	NoW	Type	Area	Result
1.0	35.0	12.0	5.0	1.0	100.0	-1.0
1.0	29.0	7.0	5.0	1.0	96.0	1.0
1.0	50.0	8.0	1.0	3.0	132.0	-1.0
1.0	32.0	11.75	7.0	3.0	750.0	-1.0
1.0	67.0	9.25	1.0	1.0	42.0	-1.0
1.0	41.0	8.0	2.0	2.0	20.0	1.0
1.0	36.0	11.0	2.0	1.0	8.0	-1.0
1.0	59.0	3.5	3.0	3.0	20.0	-1.0
1.0	20.0	4.5	12.0	1.0	6.0	1.0
2.0	34.0	11.25	3.0	3.0	150.0	-1.0

Table 4.1: First 10 patients in the *Cryotherapy Data Set*

We aim at predicting the success of treating new patients with the *Cryotherapy*. Due to this, we perform a *learning method* on the *Cryotherapy Data Set*, which represents empirical values.

We start with the *kernel ridge regression*. Without having tuned any parameters, we choose $\sigma = 1$, $\lambda = 1$ and have a first look at the performance. The classification rate indicates the amount of test data, which has been correctly classified and is crucial for judging the quality of prediction. Figure 4.1 illustrates the classification rate for different choices of training data and 4 distinct index sets I for the kernel (4.1). Here, the number of training data being 10 means that the information on the first 10 patients serve as training data and the rest as test data.

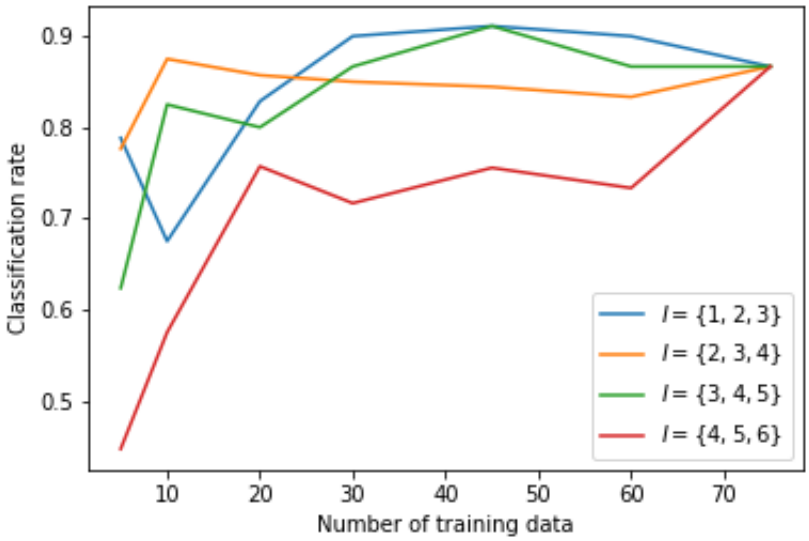

Figure 4.1: *Cryotherapy Data Set* - Classification rate for different choices of consecutive training data, with $\sigma = 1$ and $\lambda = 1$

Figure 4.1 answers our expectation that the classification rate hugely depends on the number of training data. The first value for the number of training data, we examined, is 5. When raising this value to 10, the classification rate goes distinctly up for 3 out of 4 index sets I. Presumably, analysing the information on the first 5 patients is insufficient for reliable predictions. But if having reached a certain number of training data, raising it further does not improve the performance much anymore. Moreover, it can be guessed that certain kernel functions perform better than others. However, we cannot draw general conclusions by Figure 4.1 yet. To raise the significance, we need to tune the parameter σ now. As described in Subsection 3.3.3, we do so by performing our learning model several times for different values of σ. When looking closely at Table 4.1, we notice that 9 out of the 10 first patients in the data set are of the same sex. Therefore, choosing consecutive training data is inappropriate, since the data seems to be sorted. A remedy is the *train_test_split* function from the *sklearn.model_selection* module, which splits data into random training and test subsets. From now on, we use this tool to select training data. We start with a proportion of 0.25 of the data set to be included in the train split. This yields Figure 4.2, where the stars highlight the respective maximum.

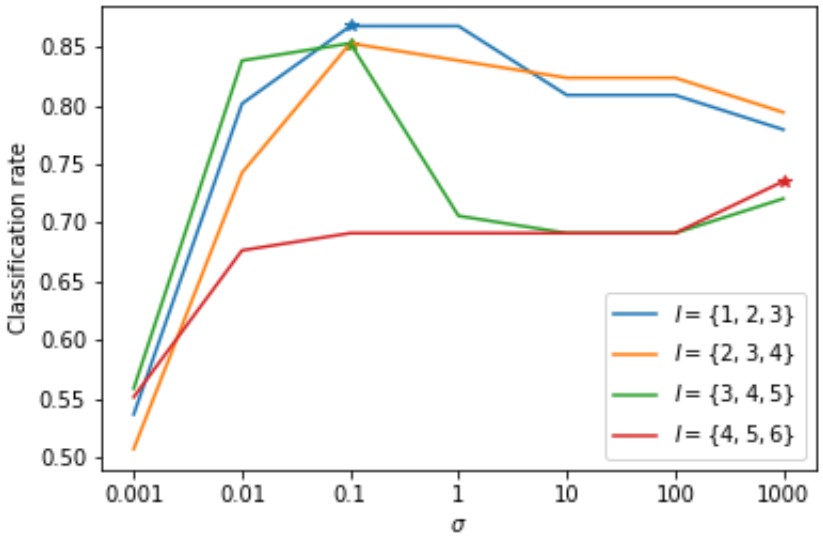

Figure 4.2: *Cryotherapy Data Set* - Classification rate for different values of σ, with $\lambda = 1$

Unambiguously, Figure 4.2 reveals the huge importance of tuning the parameters within the *kernel ridge regression*. All index set's performance is influenced tremendously by the choice of σ. This is particularly recognisable by the index set $I = \{1, 2, 3\}$. For $\sigma = 0.001$, the classification rate comes to 0.537, i. e. labels are misclassified in every second case. This performance is absolutely unsatisfying. We might just as well guess. Especially for medical applications, using the mentioned setting is reckless. As opposed to this, the classification rate for $\sigma = 0.1$ is 0.868. Considering that our learning method was performed on the basis of only 3 empirical features of 22 patients, this is a great result. Combining the kernel for this index set with other kernel matrices in the fashion of Example 3.4, gives hope for promising results. At this point, we refrain from ascertaining the combination of kernel matrices, which yields the best classification rate for the *Cryotherapy Data Set*. This can be caught up on with the attached Python code anytime. In doing so, we need to keep in mind that there exist $\binom{6}{3} = 120$ possibilities for windows of 3 features. Just for the sake of convenience, we restricted our evaluations above on the windows of consecutive features.

We remember that we did not gain much general knowledge by Figure 4.1. After having experienced the huge impact of tuning σ by Figure 4.2, we are interested in how Figure 4.1 looks like if we do not fix σ for all windows and all numbers of training data, but use the optimal σ each. Here, optimal σ denotes the

$\sigma \in \{0.001, 0.01, 0.1, 1, 10, 100, 1000\}$, with which the best classification rate is obtained. This yields Figure 4.3, where the dotted lines correspond with those from Figure 4.1 and the drawn through ones illustrate the classification rate using the optimal σ for the number of consecutive training data and the current window.

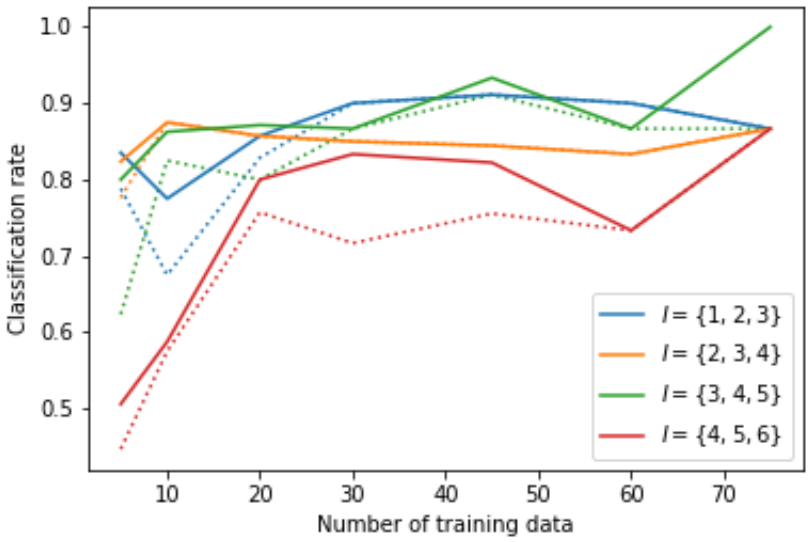

Figure 4.3: *Cryotherapy Data Set* - Classification rate for $\sigma = 1$ (dotted lines) and optimal σ (drawn through lines), with $\lambda = 1$

We perceive that each dotted line lies either underneath or on the corresponding drawn through line. I. e. tuning σ universally improves the performance.
Recalling Remark 2.1, we want to examine the importance of scaling the data. We expect our learning model to perform better if the *Cryotherapy Data Set* is scaled previously. To check whether this is a misjudgement, we perform the *kernel ridge regression* both on unscaled and scaled data. Figure 4.4 highlights the resulting classification rate, where the optimal σ was used each. Again, the training data has a proportion of 0.25 of the data set and is chosen randomly.

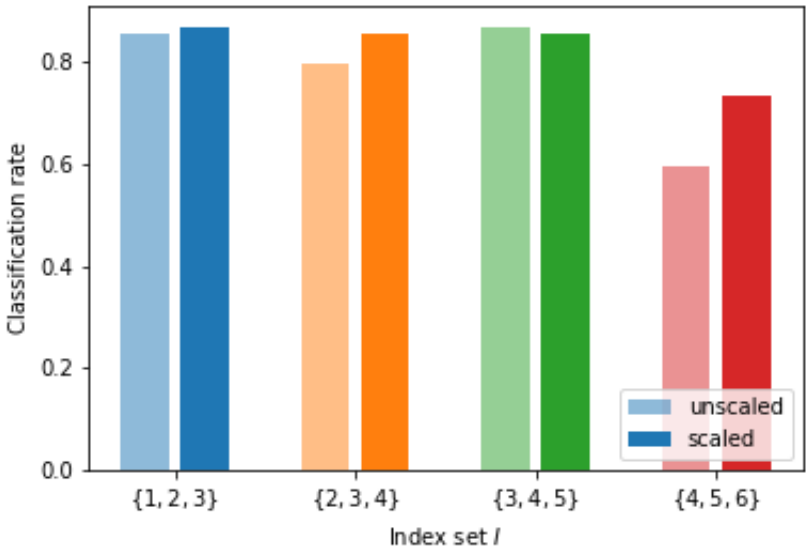

Figure 4.4: *Cryotherapy Data Set* - Classification rate for unscaled and scaled data, with optimal σ and $\lambda = 1$

Figure 4.4 perfectly demonstrates the relevance of scaling. Without scaling the data set first, weightings are assigned just on the basis of the feature's magnitude. Needless to say, this weighting criterion is by no means scientifically justified. This distorts the learning process, what limits our success. In three quarters of all cases, omitting scaling results in a decreasing classification rate.
Moreover, this causes misinterpretations, which can possibly have tremendous consequences. The first red bar in Figure 4.4 makes us believe that the last 3 features, i. e. the number of warts, the type and the area, do not have a high impact on the prospects of success for the *Cryotherapy*. Without the second red bar, which reveals the real impact, we would underestimate the relevance of the last 3 features and draw wrong conclusions. Especially for medical applications, this absolutely needs to be avoided. Consequently, scaling the data is a crucial step, which is definitely necessary to obtain unambiguous results.
Last but not least, we want to compare the classification rate of the windows of consecutive features with the results for the single features. Moreover, we are interested in how the *Gaussian kernel* (3.11) performs against. Figure 4.5 illustrates this comparison, where the optimal σ was used each and the number of training data constitutes 25 per cent of the data set.

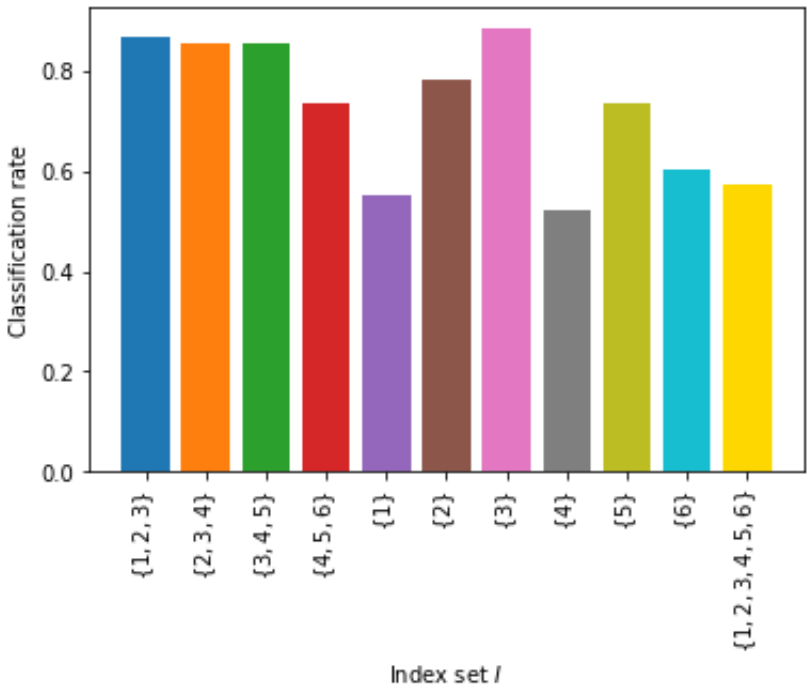

Figure 4.5: *Cryotherapy Data Set* - Classification rate for distinct index sets I, with optimal σ and $\lambda = 1$

In Subsection 3.3.2, we ventured the guess that kernels, which are based on a window of 3 features perform better than the ones based on only one feature. We suppose, this is due to the relations between the features, which are highly relevant and are neglected if just one feature is taken into account. This can just partly be substantiated by Figure 4.5. On the one hand, three quarters of all window-based kernels yield better results than all but one kernel, which are based on a single feature. On the other hand, one window-based kernel performs worse than half of the kernels, which are based on a single feature. Furthermore, certain individual features perform anything but bad. Especially using the kernel based on the third feature, we obtain a remarkable classification rate, which even trumps the results of all window-based kernels. Comparing Definitions (3.11) and (3.39), we realise that choosing a window of all features, i. e. $I = \{1, 2, 3, 4, 5, 6\}$ for the *Cryotherapy Data Set*, the *generalised Gaussian ANOVA kernel* embodies the *Gaussian kernel*. Hence, the yellow bar in Figure 4.5 illustrates the performance of the *Gaussian kernel*. In Chapter 3, we based the necessity of the *ANOVA kernel* on the fact that the input dimension has to be smaller than 4, so that we can perform the *learning methods* fast and efficiently. Now, we realise that searching for alternatives for the *Gaussian kernel* was a good idea regarding the performance as well. Its classification rate

clearly lies below most other kernel's results.

Next, we perform the *semi-supervised learning*. For that, we introduce the variable $n_{\text{train}} \in \mathbb{N}$, which indicates how many *feature vectors* of the labelled data set are taken per class as training data. Let $n_{\text{train}} = 3$ for instance. Then, information on 6 patients serve as training data, where 3 are randomly picked for each label $y_i \in \{-1, 1\}$. Hence, sorted data sets do not pose any problem. But we need to keep in mind that the results differ for every new computation, since the training data is determined each time anew. Figure 4.6 illustrates the average classification rate for different values of n_{train} and optimal σ after 5 computations each.

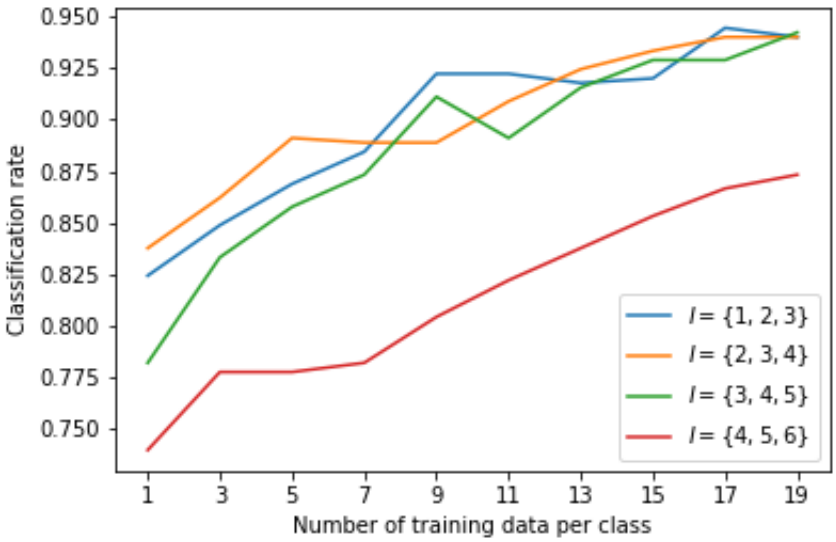

Figure 4.6: *Cryotherapy Data Set* - Average classification rate for different values of n_{train}, with optimal σ and $\lambda = 1$, after 5 computations each

One thing is clearly recognisable: the more training data per class, the higher the classification rate. This agrees with our expectation. It stands to reason that predicting whether a treatment works is more likely to be successful if we are given more empirical information about previous successful and failed treatments. Figure 4.3 could not unequivocally confirm the same. Presumably, this is due to the fact that the training data neither was determined randomly nor contained the same amount of information per class. The perception that the kernel, which is based on the index set $I = \{4, 5, 6\}$, performs worse than the other window-based kernels, is confirmed by all previous figures.
Analogously to Figure 4.5, we want to examine the performance of the window-

based kernels in comparison to the kernels, which are based on only one feature and the *Gaussian kernel*, now. Figure 4.7 compares the performance of the *kernel ridge regression* and the *semi-supervised learning*, where the information on 22 randomly picked patients serve as training data for the former and $n_{\text{train}} = 11$, where the exact same training data is used for all index sets within the *semi-supervised learning*.

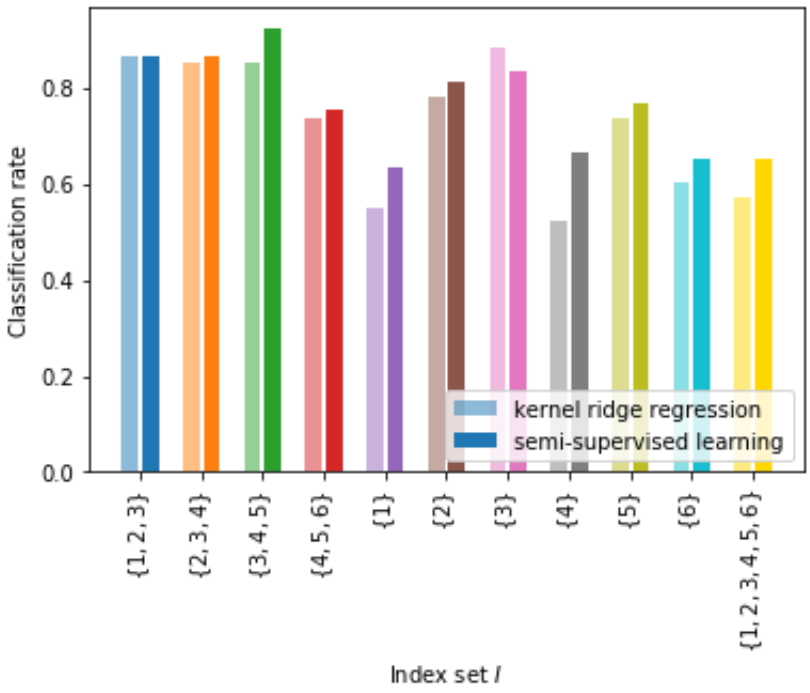

Figure 4.7: *Cryotherapy Data Set* - Classification rate for distinct index sets I using the *kernel ridge regression* and the *semi-supervised learning*, with optimal σ and $\lambda = 1$

Figure 4.7 reveals that the *semi-supervised learning* almost always yields better results than the *kernel ridge regression*. Even though we picked the training data for the *kernel ridge regression* randomly, so that the performance is not distorted in case of a sorted data set, the deviations are occasionally tremendous. Especially kernels, which are based on a single feature mostly report marked differences. The results for the window-based kernels are predominantly more balanced. Predicting class affiliations seemingly is more successful when using the same number of training data per class. This makes sense.
Obviously, we could go on examining the classification rate for a variety of settings forever. This would go beyond the constraints of this thesis. Since we turned our back on the parameter λ up to now, tuning λ is the last aspect to be considered in this chapter. After having demonstrated the huge importance of tuning σ in Figure 4.3,

we wonder how different values of λ affect the classification rate. Investigating this requires a multi-staged process. First, we set $\lambda = 1$. After having tuned σ, we fix the optimal σ and tune λ subsequently in the same fashion. Figure 4.8 shows the results for different values of n_{train}, where the drawn through lines represent the results for randomly picked training data with optimal parameters σ and λ. By way of comparison, the dotted lines show the corresponding classification rate for optimal σ and $\lambda = 1$, where the exact same training data is used.

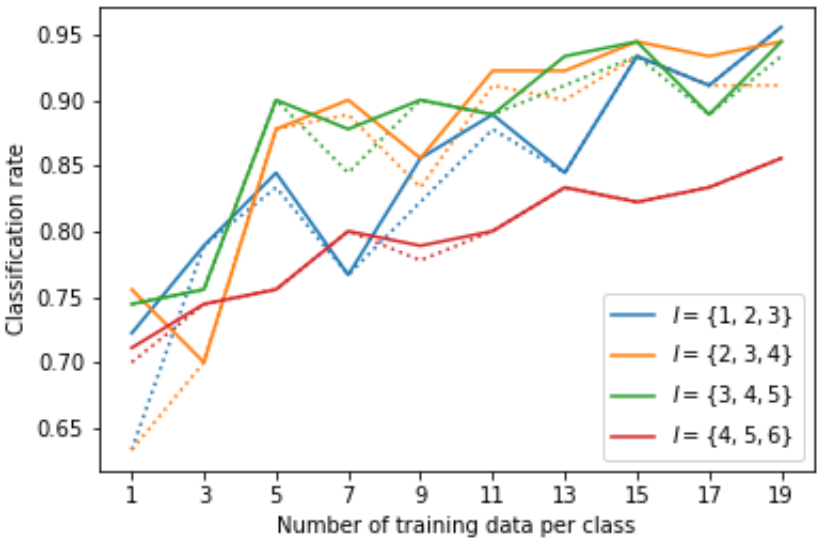

Figure 4.8: *Cryotherapy Data Set* - Classification rate for different values of n_{train}, with optimal σ and λ (drawn through lines) and optimal σ and $\lambda = 1$ (dotted lines)

Looking at Figure 4.8, we notice that the dotted lines lie close to the corresponding drawn through lines. It seems that tuning λ does not highly affect the classification rate.

Finally, we want to confirm the knowledge gained above by performing the learning processes again on an arbitrary other data set. For that, we choose to analyse the *Titanic Data Set* and create a predictive model that detects, "what sorts of people were more likely to survive" the Titanic shipwreck [14]. This is done using passenger data such as name, age, gender and socio-economic class. But not all data included in the *Titanic Data Set* is relevant for the *learning process*. Therefore, features such as PassengerID, Name and Cabin are excluded. Moreover, samples with missing values are sorted out and categorical attributes are transformed to numerical values. Table 4.2 shows the features, which are included in the learning process for the

first 10 passengers. Overall, our data set contains 712 samples, having assigned 5 features and a class label ($y_i = -1$ for deceased, $y_i = 1$ for survived), each.

Survived	Pclass	Sex	Age	Fare	Embarked
-1.0	3.0	1.0	22.0	7.25	2.0
1.0	1.0	0.0	38.0	71.2833	0.0
1.0	3.0	0.0	26.0	7.925	2.0
1.0	1.0	0.0	35.0	53.1	2.0
-1.0	3.0	1.0	35.0	8.05	2.0
-1.0	1.0	1.0	54.0	51.8625	2.0
-1.0	3.0	1.0	2.0	21.075	2.0
1.0	3.0	0.0	27.0	11.1333	2.0
1.0	2.0	0.0	14.0	30.0708	0.0
1.0	3.0	0.0	4.0	16.7	2.0

Table 4.2: First 10 passengers in the *Titanic Data Set*

Once more, we start with examining the performance of the *kernel ridge regression*. Above, we already illustrated that the number of train data highly impacts the prediction quality, until a certain level is reached, see Figure 4.3. Moreover, we emphasised by Figures 4.2 and 4.3 that scaling the parameter σ is a crucial step. These findings make perfectly sense and answered our expectations, so that we do not demonstrate them again. However, we want to re-examine how the design of the kernel influences the prediction quality.
For choosing the train and test samples, we use the *train_test_split* function with *train_size* $= 0.25$, again. This yields Figure 4.9, where the classification rates are given for several kernels and for unscaled and scaled samples. We are not surprised that scaling does not highly affect kernels, which are based on a single feature. In these cases, the input data consists of only 1 column of values of same magnitude. Then, scaling is not as crucial as for window-based kernels. However, it is confirmed that scaling makes an appreciable difference, when using kernels which are based on several features. We recognise that the *Gaussian kernel* yields clearly worse results in comparison to specific window-based kernels, anew. Additionally, Figure 4.9 nicely shows, that the design of the kernel is crucial. We analyse that the kernel which is based on the second feature yields by far the best classification rate among the kernels based on a single feature. This results in the window-based kernels containing the second feature performing considerably better than the others.

Figure 4.9 excellently reveals these relations.

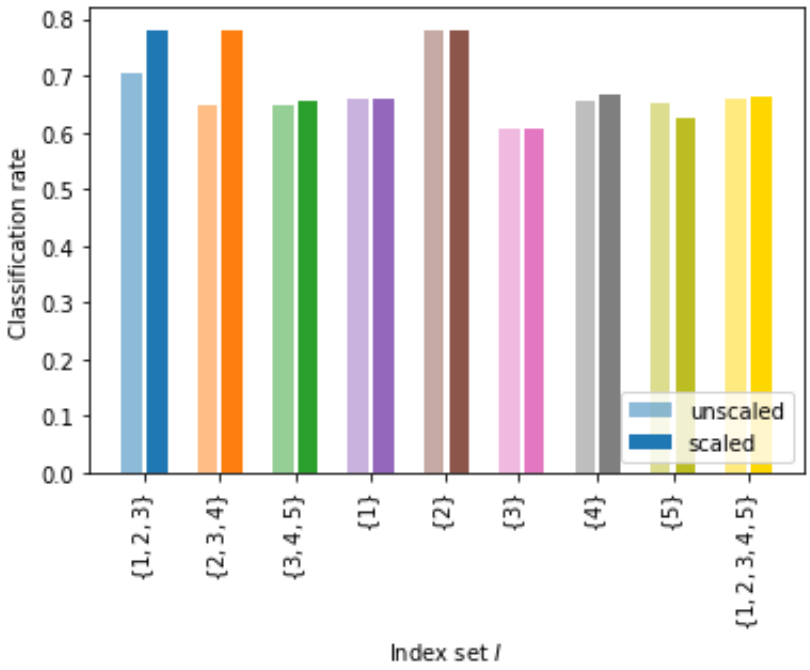

Figure 4.9: *Titanic Data Set* - Classification rate for unscaled and scaled data and distinct index sets I, with optimal σ and $\lambda = 1$

Next, we generate the analogue to Figure 4.7 for the *Titanic Data Set*. For this, we illustrate the results for scaled samples from Figure 4.9 in comparison with the corresponding classification rates obtained by the *semi-supervised learning*. For the *kernel ridge regression*, we use *train_size* $= 0.25$. Since $712 \cdot 0.25 = 178$, we choose $n_{\text{train}} = 89$ for the *semi-supervised learning*. This gives Figure 4.10, which reinforces the takeaway from Figure 4.7 that class affiliations are more likely to be predicted correctly by the *semi-supervised learning*. This pertains for all considered kernels without exception. The deviations range between 0.048 and 0.172. Clearly, 17.2 per cent make a big difference. Accordingly, whether the prediction is classified as successful can even depend on the *learning method* chosen. Among others, this is due to the fact that the train and test data are chosen differently. For both *learning methods*, the *model selection* is randomly done. However, the *semi-supervised learning* is performed on train data with the same number of samples per class. This is not guaranteed by the *train_test_split* function, which is our method of choice for the *kernel ridge regression*.

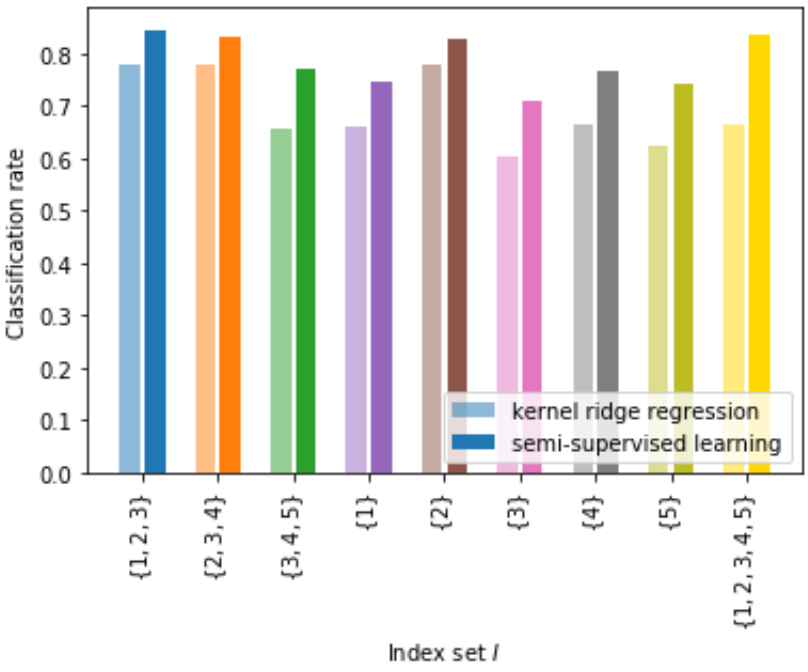

Figure 4.10: *Titanic Data Set* - Classification rate for distinct index sets I using the *kernel ridge regression* and the *semi-supervised learning*, with optimal σ and $\lambda = 1$

By the *Titanic Data Set*, we could confirm our intuitions and expectations concerning the results, anew. Our findings for this data set match the ones for the *Cryotherapy Data Set*.
For the *Cryotherapy Data Set*, the execution time for performing both *learning methods* is neglectable. All results are available immediately. However, N is only 90. That is different for the *Titanic Data Set*, with $N = 712$. Here, execution times of several seconds occur. This shows, what we already announced previously. The computational complexity for solving the linear systems (4.2) scales bad for large N. Developing a strategy for performing *learning methods* on high-dimensional data fast, is subject of the next chapter.

5 Fast Matrix-Vector Multiplication

In Chapter 2 we elaborated on two different learning methods. Both of them work well for teaching programs how to recognise binary patterns, provided that we do not fail in two steps. The first one refers to determining a kernel function with a good generalisation performance. This is what we dealt with in Chapters 3 and 4. The second crucial step relates to the linear systems, which need to be solved no matter which of both learning methods we pick. In the previous chapter we used small data sets for examining the prediction quality for several kernel functions. Solving a linear system is certainly not a big deal for such data sets. But in reality we mostly come across large data sets. This turns solving a linear system into a huge challenge concerning computational complexity. We therefore seek for a way of performing these matrix-vector multiplications particularly for high-dimensional data fast and efficiently. It is the last aspect which is left to be considered in this thesis.

5.1 The Cholesky Decomposition

For reasons of complexity linear systems as (2.22) from Subsection 2.1.2 are usually not solved directly. Compared to other direct methods of solving linear systems the so-called *Cholesky decomposition* [15] provides a huge improvement in computational complexity though. It is the factorisation of choice for symmetric, positive definite matrices. The *Cholesky decomposition* is based on the following theorem presented in Bornemann [15].

Theorem 5.1. *Every symmetric, positive definite matrix $A \in \mathbb{R}^{N \times N}$ can be written as*

$$A = LL^T, \tag{5.1}$$

where L is a lower triangular matrix and has a positive diagonal. This representation of A is unique.

Proof. The approach of constructing the decomposition (5.1) is to build up the factors row-wise for the principal submatrices

$$A_i = L_i L_i^T \tag{5.2}$$

with

$$A_i = \left(\begin{array}{c|c} A_{i-1} & a_i \\ \hline a_i^T & \alpha_i \end{array}\right), \quad L_i = \left(\begin{array}{c|c} L_{i-1} & \\ \hline l_i^T & \lambda_i \end{array}\right), \quad L_i^T = \left(\begin{array}{c|c} L_{i-1}^T & l_i \\ \hline & \lambda_i \end{array}\right). \tag{5.3}$$

In each step a new row of L is attached, what can easily be seen above. Hence, it is reasonable to prove inductively that this partitioning works. The equations from (5.3) yield

$$\begin{aligned} A_{i-1} &= L_{i-1} L_{i-1}^T, \\ l_i^T L_{i-1}^T &= a_i^T, \\ L_{i-1} l_i &= a_i, \\ l_i^T l_i + \lambda_i^2 &= \alpha_i. \end{aligned} \tag{5.4}$$

The first equation shows the factorisation of the previous step, i. e. the induction hypothesis. The second one is the transposed of the third one. It therefore is equivalent and redundant. This leaves us with the last two equations. The third one can be solved by

$$l_i = L_{i-1}^{-1} a_i. \tag{5.5}$$

By the previous step, L_{i-1} is known and a_i can be read from the matrix A. If L_{i-1}^{-1} is defined, we accept this so-called *forward substitution*. Since L_{i-1} is uniquely defined as a lower triangular matrix and has a positive diagonal by the induction hypothesis, the inverse L_{i-1}^{-1} exists. The last equation from (5.4) yields

$$\lambda_i = \sqrt{\alpha_i - l_i^T l_i}, \tag{5.6}$$

what is slightly trickier to validate. So far there is no evidence that a positive square root as above can be calculated. We must verify that $\alpha_i - l_i^T l_i > 0$. Let us suppose that z_i is the solution to $L_{i-1}^T z_i = -l_i$. Then,

$$\begin{aligned} 0 &< \begin{pmatrix} z_i \\ 1 \end{pmatrix}^T A_i \begin{pmatrix} z_i \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} z_i^T & 1 \end{pmatrix} \left(\begin{array}{c|c} L_{i-1} L_{i-1}^T & L_{i-1} l_i \\ \hline l_i^T L_{i-1}^T & \alpha_i \end{array}\right) \begin{pmatrix} z_i \\ 1 \end{pmatrix} \\ &= \underbrace{z_i^T L_{i-1} L_{i-1}^T z_i}_{=l_i^T l_i} + \underbrace{z_i^T L_{i-1} l_i}_{=-l_i^T l_i} + \underbrace{l_i^T L_{i-1}^T z_i}_{=-l_i^T l_i} + \alpha_i \\ &= \alpha_i - l_i^T l_i \end{aligned} \tag{5.7}$$

can be derived from the fact that A_i is positive definite. This completes the proof. □

After having discussed the theory, we apply the *Cholesky decomposition* to our problems. When performing *kernel ridge regression* according to Section 2.1 we aim to solve the linear system

$$(K + \lambda I_N)\,\alpha = f. \tag{5.8}$$

To identify the method from linear algebra which is best suited, we need to take a closer look at the matrix $K + \lambda I_N$. We know that kernels typically satisfy the properties

$$\kappa\left(\mathbf{x}_i, \mathbf{x}_j\right) = \kappa\left(\mathbf{x}_j, \mathbf{x}_i\right) \tag{5.9}$$

and

$$\kappa\left(\mathbf{x}_i, \mathbf{x}_j\right) \geq 0 \tag{5.10}$$

for all $\mathbf{x}_i$, $\mathbf{x}_j \in \mathbb{R}^n$ [1]. Hence, we have

$$\begin{aligned} K + \lambda I_N &= \begin{bmatrix} \kappa(\mathbf{x}_1, \mathbf{x}_1) + \lambda & \kappa(\mathbf{x}_1, \mathbf{x}_2) & \dots & \kappa(\mathbf{x}_1, \mathbf{x}_N) \\ \kappa(\mathbf{x}_2, \mathbf{x}_1) & \kappa(\mathbf{x}_2, \mathbf{x}_2) + \lambda & \dots & \kappa(\mathbf{x}_2, \mathbf{x}_N) \\ \vdots & \vdots & \ddots & \vdots \\ \kappa(\mathbf{x}_N, \mathbf{x}_1) & \kappa(\mathbf{x}_N, \mathbf{x}_2) & \dots & \kappa(\mathbf{x}_N, \mathbf{x}_N) + \lambda \end{bmatrix} \\ &= \begin{bmatrix} \kappa(\mathbf{x}_1, \mathbf{x}_1) + \lambda & \kappa(\mathbf{x}_1, \mathbf{x}_2) & \dots & \kappa(\mathbf{x}_1, \mathbf{x}_N) \\ \kappa(\mathbf{x}_1, \mathbf{x}_2) & \kappa(\mathbf{x}_2, \mathbf{x}_2) + \lambda & \dots & \kappa(\mathbf{x}_2, \mathbf{x}_N) \\ \vdots & \vdots & \ddots & \vdots \\ \kappa(\mathbf{x}_1, \mathbf{x}_N) & \kappa(\mathbf{x}_2, \mathbf{x}_N) & \dots & \kappa(\mathbf{x}_N, \mathbf{x}_N) + \lambda \end{bmatrix}, \end{aligned} \tag{5.11}$$

i. e. $K+\lambda I_N$ is symmetric and positive definite for $\lambda > 0$. This enables us to compute the *Cholesky decomposition*, i. e. to rewrite $K + \lambda I_N$ as

$$K + \lambda I_N = LL^T, \tag{5.12}$$

where L is a lower triangular matrix. Having calculated L, we solve

$$Lc = f \tag{5.13}$$

for c by *forward substitution* and

$$L^T\alpha = c \tag{5.14}$$

for α by *back substitution*. Then,

$$f = Lc = L\left(L^T\alpha\right) = LL^T\alpha = \left(K + \lambda I_N\right)\alpha \tag{5.15}$$

yields (5.8).
This works analogously for the linear system (2.33) occurring in the *spectral clustering* learning method from Section 2.2. The symmetry and positive definiteness of $I_N + \lambda L_{\text{sym}}$ are guaranteed by Lemma 2.5.
Even though the *Cholesky decomposition* performs better than other direct methods, such as the LU decomposition, for solving linear systems, its complexity is still cubic. We abstain from the proof here and refer to the literature. As a result, we now consider an iterative solver.

5.2 The Conjugate Gradient Method

The *conjugate gradient method* [16, 17] or *CG-method* for short is an iterative method for solving a system of linear equations

$$Az = b, \tag{5.16}$$

where $A \in \mathbb{R}^{N\times N}$ and $b \in \mathbb{R}^N$ are given and $z \in \mathbb{R}^N$ is unknown. It is applied to systems that are too large to be solved by direct methods as the *Cholesky method* from the previous section. A needs to be symmetric and positive definite.
It is known that given a symmetric and positive definite matrix A, the solution $\hat{z}$ of a linear system (5.16) is equal to the argument that minimises the quadratic form

$$g(z) = \tfrac{1}{2}z^TAz - b^Tz + c, \tag{5.17}$$

where c is a scalar constant. Setting the derivation of (5.17) to zero

$$\nabla_z g(z) = Az - b \stackrel{!}{=} 0 \tag{5.18}$$

provides the explanation. This implies that we can solve system (5.16) by solving the optimisation problem corresponding to (5.17). The *CG-method* is derived from the method of *steepest descent*. Its idea is to start with an initial guess z_0 for the solution $\hat{z}$ of (5.18). From there several steps along the steepest descent are taken until a good estimate of the solution $\hat{z}$ is obtained. New estimates of $\hat{z}$ are always closer to the solution than the previous one. The direction taken at each step is the one in which g decreases most rapidly, so that the negative gradient at this point is chosen. The difference

$$r_i = b - Az_i \tag{5.19}$$

is called the *residual* of z_i as an estimate of $\hat{z}$. It is computed at each step i. Recapitulating what we just said about the directions at each step, we realise that $r_i = -g'(z_i)$. From now on, we think of the *residual* as the direction of steepest descent. The method of *steepest descent* is designed in such a way that successive directions are obliged to be orthogonal to each other. This yields a zigzag path along which we converge towards the solution. However, it implies that our path runs towards wrong directions in the meantime. Moreover, steps are taken in directions which were already pursued before. It is easy to imagine that the resulting path is often far from the optimal path towards the solution, what leads to an increase in the number of required steps.
This problem is addressed by the *CG-method*. Here, a set of orthogonal search directions, the so-called *conjugate directions* p_i, is chosen. Now, exactly one step is taken in each direction. This implies that after this one step, we need to be lined up evenly with the solution $\hat{z}$. It may happen that rounding-off errors are encountered and the residual r_N after N steps is still too large. In this case it might help to continue the iteration. Doing so can yield better estimates of $\hat{z}$. However, we should not go too far beyond z_N. A remedy is to start all over again with taking the last estimate as the new initial guess, as to reduce the effects of rounding-off errors. This can actually be done at every single step, what provides great flexibility, when using this method.
The following algorithm describes the *conjugate gradient method*.

Algorithm 5.2. First, z_0 is set to an initial estimate for the solution. Alternatively, it can be set to 0. Then, we perform the first step

$$\begin{aligned} r_0 &= b - Az_0 \\ p_0 &= r_0 \\ i &= 0 \end{aligned} \tag{5.20}$$

and repeat for all subsequent steps i

$$\begin{aligned} \alpha_i &= \frac{r_i^T r_i}{p_i^T A p_i} \\ z_{i+1} &= z_i + \alpha_i p_i \\ r_{i+1} &= r_i - \alpha_i A p_i \\ \beta_i &= \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i} \\ p_{i+1} &= r_{i+1} + \beta_i p_i \\ i &= i + 1 \end{aligned} \tag{5.21}$$

until we obtain an estimate z_j with $r_j = b - Az_j \approx b - A\hat{z} = 0$, i. e. $\hat{z} \approx z_j$ and r_j is sufficiently small [16].

At this point, we do not go into further detail. Missing details regarding the mathematical theory can be consulted in [16] and [17].
Due to the matrix-vector multiplications, performing the *CG-method* requires $\mathcal{O}(N^2)$ operations. Even though that is a huge improvement in comparison to cubic complexities, it still does not scale well to large N.

5.3 The Nonequispaced Fast Fourier Transform

In many applications coordinate transformations can make our lives a lot easier. Expressions can be simplified and get amenable for computations. The so-called *Fourier transform* [18] describes a concept, where the orthogonal basis of the coordinate system for equations is represented by sine and cosine functions of increasing frequency. This works analogously to regular vector spaces except there are infinitely many directions. We are going to use this to approximate expressions quickly with minimal error.
First, we introduce the *Hermitian inner product* for functions $f(z)$ and $g(z)$, which are defined for $z \in [a, b]$, as

$$\langle f(z), g(z) \rangle = \int_a^b f(z)\, \overline{g}(z)\, dz, \tag{5.22}$$

with $\overline{g}$ denoting the complex conjugate. When discretising these functions into data vectors $\mathbf{f} = \begin{bmatrix} f_1 & f_2 & \dots & f_n \end{bmatrix}^T$ and $\mathbf{g} = \begin{bmatrix} g_1 & g_2 & \dots & g_n \end{bmatrix}^T$, the resulting normalised inner product

$$\tfrac{b-a}{n-1}\langle f, g \rangle = \tfrac{b-a}{n-1} g^T f = \tfrac{b-a}{n-1} \textstyle\sum_{k=1}^{n} f_k \overline{g}_k = \tfrac{b-a}{n-1} \sum_{k=1}^{n} f(z_k)\, \overline{g}(z_k) \tag{5.23}$$

is the Riemann approximation to the inner product of the continuous function. Taking the limit of $n \to \infty$ with $\frac{b-a}{n-1} \to 0$, (5.23) converges to (5.22). Let $f(z)$ be L-periodic on $[0, L)$ and piecewise smooth now. Then, it can be written by means of the *Fourier series* [18]

$$f(z) = \frac{a_0}{2} + \sum_{k=1}^{\infty} \left(a_k \cos\left(\frac{2\pi k z}{L}\right) + b_k \sin\left(\frac{2\pi k z}{L}\right) \right). \tag{5.24}$$

The coefficients a_k and b_k are given by

$$\begin{aligned} a_k &= \frac{2}{L} \int_0^L f(z) \cos\left(\frac{2\pi k z}{L}\right) dz \\ b_k &= \frac{2}{L} \int_0^L f(z) \sin\left(\frac{2\pi k z}{L}\right) dz. \end{aligned} \tag{5.25}$$

We remember that our coordinate system has the orthogonal basis $\{\cos\left(\frac{2\pi kz}{L}\right), \sin\left(\frac{2\pi kz}{L}\right)\}_{k=0}^{\infty}$. Analogously to the change of basis in finite-dimensional vector spaces, the inner product is used to project a function onto the orthogonal basis of the new coordinate system here. This becomes visible when rewriting (5.25) as

$$\begin{aligned} a_k &= \frac{1}{\| \cos\left(\frac{2\pi kz}{L}\right) \|^2} \langle f(z), \cos\left(\frac{2\pi kz}{L}\right) \rangle \\ b_k &= \frac{1}{\| \sin\left(\frac{2\pi kz}{L}\right) \|^2} \langle f(z), \sin\left(\frac{2\pi kz}{L}\right) \rangle. \end{aligned} \tag{5.26}$$

Example 5.3. Let $\vec{f}$ be a vector in the $(\vec{u_1}, \vec{v_1})$ coordinate system. Then, $\vec{f}$ can be written using the projections onto the orthogonal bases $\vec{u_1}$ and $\vec{v_1}$, i. e.

$$\vec{f} = \langle \vec{f}, \vec{u_1} \rangle \frac{\vec{u_1}}{\|\vec{u_1}\|^2} + \langle \vec{f}, \vec{v_1} \rangle \frac{\vec{v_1}}{\|\vec{v_1}\|^2}. \tag{5.27}$$

This way, a change of basis as in Figure 5.1 by Brunton et al. [18] can be performed easily by

$$\vec{f} = \langle \vec{f}, \vec{u_2} \rangle \frac{\vec{u_2}}{\|\vec{u_2}\|^2} + \langle \vec{f}, \vec{v_2} \rangle \frac{\vec{v_2}}{\|\vec{v_2}\|^2}. \tag{5.28}$$

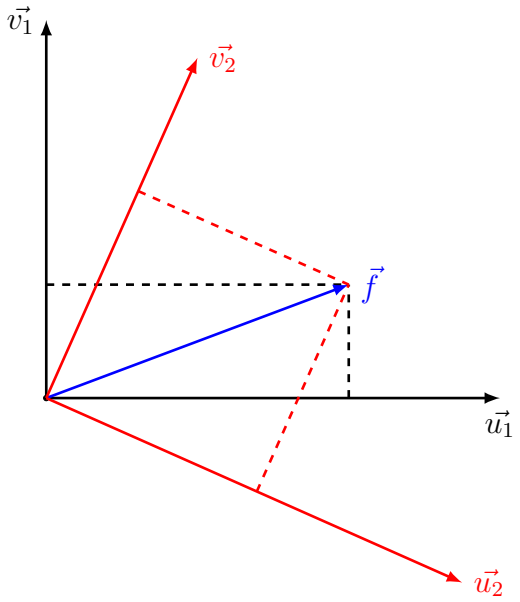

Figure 5.1: Change of coordinates of a vector in two dimensions

Above, the *Fourier series* for L-periodic functions on $[0, L)$ is defined in such a way that the function repeats itself forever outside the domain. This representation is

not always reasonable. Thus, the *Fourier transform* is introduced. Let us consider a function $f(z)$ on a domain $z \in [-L, L)$ now. For convenience, this function is $2L$-periodic. By (5.24) and Euler's formula $e^{ikz} = \cos(kz) + i\sin(kz)$ its *Fourier series* is

$$\begin{aligned} f(z) &= \frac{a_0}{2} + \sum_{k=1}^{\infty} \left(a_k \cos\left(\frac{k\pi z}{L}\right) + b_k \sin\left(\frac{k\pi z}{L}\right) \right) \\ &= \sum_{k=-\infty}^{\infty} c_k e^{ik\pi z/L} \end{aligned} \tag{5.29}$$

with coefficients

$$c_k = \frac{1}{2L} \langle f(z), e^{ik\pi z/L} \rangle = \frac{1}{2L} \int_{-L}^{L} f(z)\, e^{-ik\pi z/L} dz. \tag{5.30}$$

Apparently, the sine and cosine basis functions have a discrete set of frequencies $\{\omega_k = k\pi/L\}_{k=-\infty}^{\infty}$. For details concerning the remodelling (5.29) of the *Fourier series* into the complex form, we refer to [18]. As mentioned above, we aim to find a valid representation of generic non-periodic functions on $(-\infty, \infty)$. Therefore, we take the limit as $L \to \infty$ and $\Delta\omega = \pi/L \to 0$, so that the length of the domain runs towards infinity and discrete frequencies turn into a continuous range of frequencies. This yields

$$f(z) = \lim_{\Delta\omega \to 0} \sum_{k=-\infty}^{\infty} \frac{\Delta\omega}{2\pi} \underbrace{\int_{-\pi/\Delta\omega}^{\pi/\Delta\omega} f(\xi)\, e^{-ik\Delta\omega\xi} d\xi}_{\langle f(\xi), e^{ik\Delta\omega\xi} \rangle} e^{ik\Delta\omega z}, \tag{5.31}$$

where taking the limit of the expression $\langle f(\xi), e^{ik\Delta\omega\xi} \rangle$ leads to the *Fourier transform* $\hat{f}(\omega) \triangleq \mathcal{F}(f(z))$ of $f(z)$. Altogether, this results in the *Fourier transform pair*

$$\begin{aligned} f(z) &= \mathcal{F}^{-1}\left(\hat{f}(\omega)\right) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{f}(\omega)\, e^{i\omega z} d\omega \\ \hat{f}(\omega) &= \mathcal{F}(f(z)) = \int_{-\infty}^{\infty} f(z)\, e^{-i\omega z} dz. \end{aligned} \tag{5.32}$$

These transformations are extremely useful and widely applied in practice. We refer to Brunton and Kutz [18] for some nice examples. This great applicability is due to the following powerful properties, which are stated without proof.

Theorem 5.4. *Let $\hat{f} = \mathcal{F}(f)$ and $\hat{g} = \mathcal{F}(g)$ be the* Fourier transforms *of functions f and g and α and β be scalars. Then the following properties hold.*

1. *Derivatives of functions:*

$$\mathcal{F}\left(\frac{d}{dz} f(z)\right) = i\omega \mathcal{F}(f(z)) \tag{5.33}$$

2. Linearity of Fourier transforms:

$$\begin{aligned} \mathcal{F}\left(\alpha f\left(z\right)+\beta g\left(z\right)\right) &= \alpha\mathcal{F}\left(f\right)+\beta\mathcal{F}\left(g\right) \\ \mathcal{F}^{-1}\left(\alpha\hat{f}\left(\omega\right)+\beta\hat{g}\left(\omega\right)\right) &= \alpha\mathcal{F}^{-1}\left(\hat{f}\right)+\beta\mathcal{F}^{-1}\left(\hat{g}\right) \end{aligned} \tag{5.34}$$

3. Parseval's theorem:

$$\int_{-\infty}^{\infty}|\hat{f}\left(\omega\right)|^{2}d\omega = 2\pi\int_{-\infty}^{\infty}|f\left(z\right)|^{2}dz \tag{5.35}$$

4. Convolution:

$$\begin{aligned} \left(f*g\right)\left(z\right) &\coloneqq \int_{-\infty}^{\infty}f\left(z-\xi\right)g\left(\xi\right)d\xi \\ &= \mathcal{F}^{-1}\left(\hat{f}\hat{g}\right)\left(z\right) \\ &= \left(g*f\right)\left(z\right) \end{aligned} \tag{5.36}$$

The *Fourier transform* is a linear operator, that allows us to compute derivatives and convolutions easily. Moreover, it preserves the L_2-norm except for a constant. Consequently, several challenging calculations in the spatial domain are very simple to implement in the Fourier domain. This makes the *Fourier transform* such a powerful tool not only in mathematics but in all sciences.
Above, we restricted our considerations to continuous functions $f\left(z\right)$. When applying this theory to real world problems, performing the *Fourier transform* on discrete data is crucial though. This is done approximately. So let vectors of data $\mathbf{f} = \begin{bmatrix} f_1 & f_2 & \dots & f_n \end{bmatrix}^T$ be given such that $f\left(z\right)$ is discretised at a regular spacing Δz. Then, the *discrete Fourier transform (DFT)* [18] denotes the discretised version of the *Fourier series*. It is given by

$$\hat{f}_k = \sum_{j=0}^{n-1} f_j e^{-2\pi ijk/n}, \tag{5.37}$$

whereas

$$f_k = \frac{1}{n}\sum_{j=0}^{n-1}\hat{f}_j e^{2\pi ijk/n} \tag{5.38}$$

is referred to as the *inverse discrete Fourier transform*. According to this, the data in $\mathbf{f}$ is mapped to $\hat{\mathbf{f}}$ in the frequency domain. Therefore, the *discrete Fourier transform*

can be seen as a linear operator, such that

$$\underbrace{\begin{bmatrix}\hat{f}_1\\ \hat{f}_2\\ \hat{f}_3\\ \vdots\\ \hat{f}_n\end{bmatrix}}_{\hat{\mathbf{f}}} = \underbrace{\begin{bmatrix}1 & 1 & 1 & \dots & 1\\ 1 & w_n & w_n^2 & \dots & w_n^{n-1}\\ 1 & w_n^2 & w_n^4 & \dots & w_n^{2(n-1)}\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ 1 & w_n^{n-1} & w_n^{2(n-1)} & \dots & w_n^{(n-1)^2}\end{bmatrix}}_{\mathbf{F}_n} \underbrace{\begin{bmatrix}f_1\\ f_2\\ f_3\\ \vdots\\ f_n\end{bmatrix}}_{\mathbf{f}}, \tag{5.39}$$

where $w_n = e^{-2\pi i/n}$ is the fundamental frequency and the matrix $\mathbf{F}_n$ is a Vandermonde matrix.

Remark 5.5. The values (5.37) and (5.38) can be computed for all $k \in \mathbb{Z}$. Due to

$$e^{-2\pi ij(k+n)/n} = w_n^{j(k+n)} = w_n^{jk} \cdot 1 = w_n^{jk} = e^{-2\pi ijk/n}, \tag{5.40}$$

$k \in \mathbb{Z}$, the sequence $(\hat{f}_k)_{k\in\mathbb{Z}}$ is n-periodic. In the same fashion, the periodicity of $(f_k)_{k\in\mathbb{Z}}$ with period n can be shown.
Let n be even. Then, we can form the *DFT* of length n of $(f_j)_{j\in\mathbb{Z}}$ by any of its n-dimensional subvectors [19]. So let us assume to choose $(f_j)_{j=-n/2}^{n/2-1}$. Since $(f_j)_{j\in\mathbb{Z}}$ is n-periodic,

$$\begin{aligned}\sum_{j=-n/2}^{n/2-1} f_j w_n^{jk} &= \sum_{j=1}^{n/2} f_{n-j} w_n^{(n-j)k} + \sum_{j=0}^{n/2-1} f_j w_n^{jk}\\ &= \sum_{j=0}^{n-1} f_j w_n^{jk}\\ &= \hat{f}_k,\end{aligned} \tag{5.41}$$

$k \in \mathbb{Z}$. Consequently, the values $\hat{f}_k$ are independent of the chosen subvector, indeed.

Remark 5.6. By taking a closer look at $w_n \in \mathbb{C}$, we observe that $w_n^n = 1$ and $w_n^k \neq 1$ for $k = 1, \dots, n-1$. Hence, w_n is a *primitive n-th root of unity*. Moreover, w_n^k is a *n-th root of unity* for all $k = 0, \dots, n-1$, because

$$\left(w_n^k\right)^n = \left(e^{-2\pi ik/n}\right)^n = e^{-2\pi ik} = 1. \tag{5.42}$$

Due to these properties, the *Fourier matrix* $\mathbf{F}_n$ can be rewritten as

$$\mathbf{F}_n = \begin{bmatrix}1 & 1 & \dots & 1\\ 1 & w_n & \dots & w_n^{n-1}\\ \vdots & \vdots & & \vdots\\ 1 & w_n^{n-1} & \dots & w_n\end{bmatrix}. \tag{5.43}$$

Obviously, $\mathbf{F}_n$ is symmetric and has only n distinct entries [19].

The matrix-vector multiplication (5.39) obviously requires $\mathcal{O}(n^2)$ operations, what scales poorly for large n. The so-called *fast Fourier transform (FFT)* [18] was developed to overcome this difficulty. Its approach is to rather solve multiple *DFT* computations of smaller dimensions instead of an n-dimensional one. For that, we require the number n of data in $\mathbf{f}$ to be a power of 2, so that (5.39) can be rewritten as

$$\hat{\mathbf{f}} = \mathbf{F}_n \mathbf{f} = \begin{bmatrix} \mathbf{I}_{n/2} & \mathbf{D}_{n/2} \\ \mathbf{I}_{n/2} & -\mathbf{D}_{n/2} \end{bmatrix} \begin{bmatrix} \mathbf{F}_{n/2} & \mathbf{0} \\ \mathbf{0} & \mathbf{F}_{n/2} \end{bmatrix} \begin{bmatrix} \mathbf{f}_{\text{odd}} \\ \mathbf{f}_{\text{even}} \end{bmatrix}, \tag{5.44}$$

where $\mathbf{f}_{\text{odd}}$ are all elements of $\mathbf{f}$ with odd index, $\mathbf{f}_{\text{even}}$ with even index respectively, $\mathbf{I}_{n/2}$ is the $(n/2) \times (n/2)$ identity matrix and

$$\mathbf{D}_{n/2} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & w_n & 0 & \dots & 0 \\ 0 & 0 & w_n^2 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & w_n^{n/2-1} \end{bmatrix}. \tag{5.45}$$

(5.44) is derived by reorganising (5.39) and (5.37). This process is repeated again and again, such that $\mathbf{F}_{n/2}$ is expressed by $\mathbf{F}_{n/4}$, which is expressed by $\mathbf{F}_{n/8}$ and so on. This leaves us performing several 2×2 *DFT* computations, which is a lot less complex than implementing the original n-dimensional one. In doing so, the algorithm scales as $\mathcal{O}(n \log(n))$, what nearly meets a linear scaling. Actually, there is no linear algorithm that can evaluate the *DFT* of length n with a smaller computational cost [19].

Even if n is no power of 2, this process can be applied. In this case, the number of data points in $\mathbf{f}$ is made a power of 2 by padding with zeros. The *fast Fourier transform* is so efficient, that this is still cheaper than performing an n-dimensional *DFT* computation [18]. Above, we gave a rough overview of the main idea of the *FFT*. However, since there are different representations of the *DFT*, there exist different possibilities to describe the *FFT*. We refer to Plonka et al. [19] for the details.

Example 5.7. Let $\mathbf{f} = \begin{bmatrix} f_1 & \dots & f_6 \end{bmatrix}^T$ be a data vector. Then, its *discrete Fourier transform* $\hat{\mathbf{f}} = \mathbf{F}_6\mathbf{f}$ can be rewritten as

$$\begin{aligned}
\hat{\mathbf{f}} &= \underbrace{\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & w_6 & w_6^2 & w_6^3 & w_6^4 & w_6^5 \\ 1 & w_6^2 & w_6^4 & w_6^6 & w_6^8 & w_6^{10} \\ 1 & w_6^3 & w_6^6 & w_6^9 & w_6^{12} & w_6^{15} \\ 1 & w_6^4 & w_6^8 & w_6^{12} & w_6^{16} & w_6^{20} \\ 1 & w_6^5 & w_6^{10} & w_6^{15} & w_6^{20} & w_6^{25} \end{bmatrix}}_{\mathbf{F}_6} \mathbf{f} \\
&= \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & w_6 & w_6^2 & w_6^3 & w_6^4 & w_6^5 \\ 1 & w_6^2 & w_6^4 & w_6^6 & w_6^8 & w_6^{10} \\ 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -w_6 & w_6^2 & -w_6^3 & w_6^4 & -w_6^5 \\ 1 & -w_6^2 & w_6^4 & -w_6^6 & w_6^8 & -w_6^{10} \end{bmatrix} \mathbf{f} \\
&= \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & w_3 & w_3^2 & w_6 & w_3w_6 & w_3^2w_6 \\ 1 & w_3^2 & w_3^4 & w_6^2 & w_3^2w_6^2 & w_3^4w_6^2 \\ 1 & 1 & 1 & -1 & -1 & -1 \\ 1 & w_3 & w_3^2 & -w_6 & -w_3w_6 & -w_3^2w_6 \\ 1 & w_3^2 & w_3^4 & -w_6^2 & -w_3^2w_6^2 & -w_3^4w_6^2 \end{bmatrix} \begin{bmatrix} f_1 \\ f_3 \\ f_5 \\ f_2 \\ f_4 \\ f_6 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & w_6 & 0 \\ 0 & 0 & 1 & 0 & 0 & w_6^2 \\ 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -w_6 & 0 \\ 0 & 0 & 1 & 0 & 0 & -w_6^2 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & w_3 & w_3^2 & 0 & 0 & 0 \\ 1 & w_3^2 & w_3^4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & w_3 & w_3^2 \\ 0 & 0 & 0 & 1 & w_3^2 & w_3^4 \end{bmatrix} \begin{bmatrix} f_1 \\ f_3 \\ f_5 \\ f_2 \\ f_4 \\ f_6 \end{bmatrix}
\end{aligned} \tag{5.46}$$

by rearranging columns and applying $w_n = e^{-2\pi i/n}$, $e^{-2\pi i} = 1$ and $e^{-\pi i} = -1$. This shows the validity of (5.44).

FFT computations require the input data to be equispaced. However, this condition is a significant drawback concerning the width of applications. Thus, we are in need of the *nonequispaced fast Fourier transform (NFFT)* [19]. Let
$\mathcal{I}_n = \{k \in \mathbb{Z} : -\frac{n}{2} \leq k < \frac{n}{2}\}$ be an index set for large $n \in \mathbb{N}$.
First, we introduce the *NFFT* for nonequispaced nodes $x_j \in [-\pi, \pi)$, $j \in \mathcal{I}_m$, in the space domain and given equispaced data $\hat{f}_k \in \mathbb{C}$, $k \in \mathcal{I}_n$, in the frequency domain.

We desire the fast evaluation of the 2π-periodic *trigonometric polynomial*

$$f(x) \coloneqq \sum_{k \in \mathcal{I}_n} \hat{f}_k \, e^{ikx} \tag{5.47}$$

at arbitrary nodes x_j, $j \in \mathcal{I}_m$, for given arbitrary coefficients $\hat{f}_k \in \mathbb{C}$, $k \in \mathcal{I}_n$. I. e. we need an efficient algorithm for computing the values

$$f_j \coloneqq f(x_j) = \sum_{k \in \mathcal{I}_n} \hat{f}_k \, e^{ikx_j}, \tag{5.48}$$

$j \in \mathcal{I}_m$. The approach is to approximate f by a linear combination s_1 of translates of a 2π-periodic window function $\tilde{\varphi}$ and to compute this approximation at the nodes x_j, $j \in \mathcal{I}_m$.
Let $\varphi \in L_1(\mathbb{R}) \cap L_2(\mathbb{R})$ be a convenient window function. Then, we define

$$\tilde{\varphi}(x) \coloneqq \sum_{r \in \mathbb{Z}} \varphi(x + 2\pi r x), \tag{5.49}$$

which is 2π-periodic and has the uniformly convergent *Fourier series*

$$\tilde{\varphi}(x) \coloneqq \sum_{k \in \mathbb{Z}} c_k(\tilde{\varphi}) \, e^{ikx} \tag{5.50}$$

with *Fourier coefficients*

$$\begin{aligned} c_k(\tilde{\varphi}) &\coloneqq \frac{1}{2\pi} \int_{-\pi}^{\pi} \tilde{\varphi}(x) \, e^{-ikx} dx \\ &= \frac{1}{2\pi} \int_{\mathbb{R}} \varphi(x) \, e^{-ikx} dx \\ &= \frac{1}{2\pi} \hat{\varphi}(k), \end{aligned} \tag{5.51}$$

$k \in \mathbb{Z}$.

Remark 5.8. Popular choices for window functions φ are for instance a *Gaussian function*, the *Bessel window* or the *centered cardinal B-spline*. We refer to Plonka et al. [19] for a thorough consideration of approximation errors for these window functions.

We pick an *oversampling factor* $\delta \geq 1$ such that $\delta n \in \mathbb{N}$ is even. Remembering that s_1 shall approximate the *trigonometric polynomial* f, we determine the coefficients g_l, $l \in \mathcal{I}_{\delta n}$, such that

$$s_1(x) \coloneqq \sum_{l \in \mathcal{I}_{\delta n}} g_l \tilde{\varphi}\left(x - \frac{2\pi l}{\delta n}\right), \tag{5.52}$$

next. (5.52) is a *discrete convolution*. The subsequent calculations show, that the respective *Fourier coefficients* are multiplied by each other in that case.
Computing the *Fourier series* of s_1 yields

$$\begin{aligned} s_1(x) &= \sum_{k\in\mathbb{Z}} c_k(s_1)\, e^{ikx} \\ &= \sum_{k\in\mathbb{Z}} \hat{g}_k c_k(\tilde{\varphi})\, e^{ikx}, \end{aligned} \tag{5.53}$$

where

$$\hat{g}_k \coloneqq \sum_{l\in\mathcal{I}_{\delta n}} g_l e^{-ik2\pi l/(\delta n)} \tag{5.54}$$

are the *discrete Fourier coefficients* as introduced above, since

$$\begin{aligned} c_k(s_1) &= \frac{1}{2\pi}\int_{-\pi}^{\pi} s_1(x)\, e^{-ikx} dx \\ &= \frac{1}{2\pi}\int_{-\pi}^{\pi} \sum_{l\in\mathcal{I}_{\delta n}} g_l \tilde{\varphi}\left(x - \frac{2\pi l}{\delta n}\right) e^{-ikx} dx \\ &= \frac{1}{2\pi}\sum_{l\in\mathcal{I}_{\delta n}} g_l \underbrace{\int_{-\pi}^{\pi} \tilde{\varphi}\left(x - \frac{2\pi l}{\delta n}\right) e^{-ikx} dx}_{=\int_{-\pi}^{\pi}\tilde{\varphi}(y)e^{-iky}dy\cdot e^{-ik2\pi l/(\delta n)}} \\ &= \sum_{l\in\mathcal{I}_{\delta n}} g_l c_k(\tilde{\varphi})\, e^{-ik2\pi l/(\delta n)}. \end{aligned} \tag{5.55}$$

By assumption, φ is a convenient window function, i. e. φ is well-localised in the space domain and $\tilde{\varphi}$ in the frequency domain. Thus, these functions have a very small support in the respective domain, so that they do not have many translates. Hence, (5.53) can be rewritten as

$$s_1(x) = \sum_{k\in\mathcal{I}_{\delta n}} \hat{g}_k c_k(\tilde{\varphi})\, e^{ikx} + \sum_{r\in\mathbb{R}\setminus\{0\}} \sum_{k\in\mathcal{I}_{\delta n}} \hat{g}_k c_{k+\delta nr}(\tilde{\varphi})\, e^{i(k+\delta nr)x}. \tag{5.56}$$

Suppose $c_k(\tilde{\varphi}) \neq 0$ for all $k \in \mathcal{I}_n$ and $|c_k(\tilde{\varphi})| \ll 1$ for $|k| \geq \delta n - \frac{n}{2}$. Then, comparing (5.47) with (5.56) yields

$$\hat{g}_k = \begin{cases} \hat{f}_k / c_k(\tilde{\varphi}) & k \in \mathcal{I}_n, \\ 0 & k \in \mathcal{I}_{\delta n} \setminus \mathcal{I}_n, \end{cases} \tag{5.57}$$

what enables us to compute the coefficients g_l in (5.52). Following the *inverse discrete Fourier transform* (5.38), we obtain

$$g_l = \frac{1}{\delta n} \sum_{k\in\mathcal{I}_{\delta n}} \hat{g}_k e^{2\pi ikl/(\delta n)}, \tag{5.58}$$

$l \in \mathcal{I}_{\delta n}$. Moreover, φ can be approximated by its truncation on $Q \coloneqq [-\frac{2\pi m_T}{\delta n}, \frac{2\pi m_T}{\delta n}]$

$$\psi(x) \coloneqq \begin{cases} \varphi(x) & x \in Q, \\ 0 & x \in \mathbb{R} \setminus Q, \end{cases} \tag{5.59}$$

where $2m_T \ll \delta n$ and $m_T \in \mathbb{N}$. Analogously to (5.49), we define

$$\tilde{\psi}(x) \coloneqq \sum_{r \in \mathbb{Z}} \psi(x + 2\pi r x) \in L_2([-\pi, \pi)). \tag{5.60}$$

Then,

$$s(x) \coloneqq \sum_{l \in \mathcal{I}_{\delta n}} g_l \tilde{\psi}\left(x - \frac{2\pi l}{\delta n}\right) = \sum_{l \in \mathcal{I}_{\delta n, m_T}(x)} g_l \tilde{\psi}\left(x - \frac{2\pi l}{\delta n}\right) \tag{5.61}$$

is an approximation of s_1, where

$$\mathcal{I}_{\delta n, m_T}(x) \coloneqq \{l \in \mathcal{I}_{\delta n} : \frac{\delta n}{2\pi} x - m_T \leq l \leq \frac{\delta n}{2\pi} x + m_T\}. \tag{5.62}$$

Consequently, (5.61) contains at most $2m_T + 1$ nonzero terms for each fixed $x_j \in [-\pi, \pi)$. Altogether, we achieved

$$f(x_j) \approx s_1(x_j) \approx s(x_j), \tag{5.63}$$

by truncating first in the frequency domain and then in the space domain. This enables us to approximately evaluate the trigonometric polynomial f for all $x_j \in [-\pi, \pi)$, $j \in \mathcal{I}_m$. The so-called *NFFT of type I* [19] requires $\mathcal{O}(n \log n + m_T m)$ operations, what is much faster than computing the values (5.48) directly in $\mathcal{O}(nm)$ operations. Summarised, the *NFFT of type I* consists of 3 steps. First, the *discrete Fourier coefficients* $\hat{g}_k$ need to be calculated according to (5.57). Thereupon, we perform the *inverse FFT* to obtain g_l from (5.58). Finally, (5.61) is evaluated at the given nodes x_j, $j \in \mathcal{I}_m$. Just a few terms of the sums in (5.61) are non-zero. This is due to the well-localisation of the window function.

Next, we introduce the *NFFT* for arbitrary, nonequispaced nodes $x_j \in [-\pi, \pi)$, $j \in \mathcal{I}_m$, in the frequency domain and given equispaced data $f_j \in \mathbb{C}$, $j \in \mathcal{I}_m$ in the space domain. We want to evaluate the values

$$h(k) \coloneqq \sum_{j \in \mathcal{I}_m} f_j e^{ikx_j}, \tag{5.64}$$

$k \in \mathcal{I}_n$, now. We begin by introducing the 2π-periodic function

$$\tilde{g}(x) \coloneqq \sum_{j \in \mathcal{I}_m} f_j \tilde{\varphi}(x + x_j), \tag{5.65}$$

where $\tilde{\varphi}$ follows Definition (5.49). Then by (5.51) and (5.64),

$$\begin{aligned} c_k(\tilde{g}) &= \frac{1}{2\pi} \int_{-\pi}^{\pi} \tilde{g}(x)\, e^{-ikx} dx \\ &= \frac{1}{2\pi} \int_{-\pi}^{\pi} \sum_{j \in \mathcal{I}_m} f_j \tilde{\varphi}(x + x_j)\, e^{-ikx} dx \\ &= \sum_{j \in \mathcal{I}_m} f_j e^{ikx_j} c_k(\tilde{\varphi}) \\ &= h(k)\, c_k(\tilde{\varphi}), \end{aligned} \tag{5.66}$$

$k \in \mathbb{Z}$, holds for the *Fourier coefficients* of $\tilde{g}$. Accordingly, we can compute the values $h(k)$, $k \in \mathcal{I}_n$, if the *Fourier coefficients* $c_k(\tilde{\varphi})$ and $c_k(\tilde{g})$ are available for all $k \in \mathcal{I}_n$. Applying the trapezoidal rule, we can approximate

$$c_k(\tilde{g}) \approx \frac{1}{\delta n} \sum_{l \in \mathcal{I}_{\delta n}} \sum_{j \in \mathcal{I}_m} f_j \tilde{\varphi}\left(x_j - \frac{2\pi l}{\delta n}\right) e^{2\pi i k l/(\delta n)}. \tag{5.67}$$

Again, φ is well-localised in the space domain, such that its truncation ψ, see (5.59), is a good approximation. Thus, $\tilde{\varphi}$ can be approximated by $\tilde{\psi}$. The method described above is called *NFFT of type II* [19] and has a computational cost of $\mathcal{O}(n \log n + m_T m)$ operations. It is also known as the *adjoint NFFT*. Understanding the sums (5.48) and (5.64) as matrix-vector products, we introduce the vectors

$$\begin{aligned} \hat{\mathbf{f}} &\coloneqq \left(\hat{f}_k\right)_{k \in \mathcal{I}_n} \in \mathbb{R}^n, \\ \mathbf{f} &\coloneqq (f_j)_{j \in \mathcal{I}_m} \in \mathbb{R}^m \end{aligned} \tag{5.68}$$

and the *nonequispaced Fourier matrix*

$$\mathbf{A} \coloneqq \left(e^{ikx_j}\right)_{j \in \mathcal{I}_m, k \in \mathcal{I}_n} \in \mathbb{R}^{m \times n}. \tag{5.69}$$

Then, calculating (5.48) for $j \in \mathcal{I}_m$ corresponds to the computation of the matrix-vector product $\mathbf{A}\hat{\mathbf{f}}$. Moreover, the values $h(k)$ in (5.64) are computed by the matrix-vector multiplication

$$(h(k))_{k \in \mathcal{I}_n} = \mathbf{A}^T (f_j)_{j \in \mathcal{I}_m}, \tag{5.70}$$

where $\mathbf{A}^T$ is the transposed matrix of $\mathbf{A}$. Hence, the *NFFT of type I* and the *NFFT of type II* are closely related. In principle, they proceed in reverse order.

Remark 5.9. Different from the *FFT*, both the *NFFT of type I* and the *NFFT of type II* are approximate algorithms. Therefore, we need to keep in mind that *approximation errors* occur. In this thesis, we refrain from addressing this in greater detail and refer to Plonka et al. [19]. Furthermore, in contrast to the *FFT* the *NFFT* is not easy to invert. If only because the matrix is usually not square.

5.4 Performing Matrix-Vector Multiplications Fast

After having introduced several promising methods in the previous sections, we want to develop a strategy for a fast and efficient computation of the two linear systems

$$(K + \lambda I_N)\,\alpha = f \tag{5.71}$$

and

$$(I_N + \lambda L_{\text{sym}})\,u = f \tag{5.72}$$

now. The *Cholesky decomposition* possesses a cubic computational complexity, whereas the *CG-method* requires $\mathcal{O}(N^2)$ operations. Both methods can be applied to both linear systems. Therefore, we orientate ourselves by the computational complexity and exclude the use of the *Cholesky decomposition*. Still, $\mathcal{O}(N^2)$ does not scale well to large $N \in \mathbb{N}$. When taking a closer look at Algorithm 5.2, computing the matrix-vector product Ap_i is by far the most expensive step within the *CG-method*. Thus, not solving this multiplication directly but replacing it by an efficient method will reduce the computational complexity tremendously. To this end, the *NFFT* is applied.

We introduced the *NFFT* as a method for evaluating the *discrete Fourier transform* for nonequispaced data fast. The connection to performing matrix-vector multiplications is not immediately clear. Actually, this *NFFT* approach is by no means reasonable for general matrices. Here, we benefit from the specific structure of our matrices $K + \lambda I_N$ and $I_N + \lambda L_{\text{sym}}$. Just like the *Fourier matrix* $\mathbf{F}_N$, both matrices are formed on the basis of the exponential function. This enables us to perform our matrix-vector products fast based on the *NFFT*. It is important to keep in mind that this technique is limited to a very small number of *kernel functions*. In fact, we are restricted to the *kernel functions* that can be well approximated by a trigonometric polynomial. For details on the so-called *NFFT-based fast summation method*, we refer to Alfke et al. [3].

It has been experienced, that the *NFFT* runs computations very efficiently as long as the input dimension is smaller than 4. Recalling our thoughts from Subsection 3.3.2,

we aim to examine the combined relation of as many coordinates as possible. Therefore, we are in need of the 3-*variate NFFT*. For dimensions larger than 1, the *NFFT* is based on a tensor product approach. Then, for instance, the window function is simply a product of univariate window functions for the particular dimensions. For details concerning the *multivariate nonequispaced fast Fourier transform*, we refer to Plonka et al. [19].
Alfke et al. [3] originated a "Python extension to compute fast approximate multiplications with Gaussian adjacency matrices" [20]. We apply this *NFFT*-based package for solving the linear systems (5.71) and (5.72) fast and efficiently. As motivated above, this is achieved by solving all matrix-vector products Ap_i within the *CG-method* no longer directly but by means of the so-called *FastAdjacency* package [20]. The approximation error ranges around 10^{-5}, so that it does not have a perceptible impact on the prediction quality. This software is targeted at the case of very large N and small n, which is why we decided to consider windows of 3 features.

After having found a theoretical approach for performing matrix-vector multiplications for the *ANOVA kernel* fast and efficiently, we want to analyse its practicality. First, we install the *FastAdjacency* package following the instructions in [20]. Then, we import the *fastadj* module in the header of our code and are ready to start. Now, instead of computing Ap_i within the *CG-algorithm* as a dot product, we approximate the kernel matrices by the function call *fastadj.AdjacencyMatrix* and compute the approximate results of the product with a vector using *apply*. For the *kernel ridge regression*,

$$\begin{aligned} Ap_i &= (K + \lambda I_N)\,\alpha_i \\ &= K\alpha_i + \lambda\alpha_i, \end{aligned} \tag{5.73}$$

where K might represent a sum of several kernel matrices, see Chapter 3. Hence, we apply the *NFFT-based fast summation method* to perform the matrix-vector product $K\alpha_i$ fast without ever forming the whole matrix K. As part of this thesis, we created a Python code, which uses the *NFFT-based fast summation* to perform the *kernel ridge regression* fast and efficiently. Comparing the problem set description in [20] with our setting, we stick to Definition (4.1) for the kernel matrix K. As in Chapter 4, we illustrate our results by the *Cryotherapy Data Set*. Again, we use the *train_test_split* function to randomly select the training data and choose $train\_size = 0.25$.
First, we want to check if the results of both Python codes, the one using the ordinary *CG-method* and the one combining the *CG-method* with the *NFFT-based fast summation*, are consistent with each other. Figure 5.2 compares the results for

the windows of consecutive features. Now, including the kernels, which are based on a single feature or the *Gaussian kernel* does not make sense, since our *NFFT-based* approach requires the input to be 3-dimensional. Still, we have $\binom{n}{3} = \binom{6}{3} = 120$ possibilities for choosing such a window. Moreover, we can combine several kernels as described in Chapter 3.

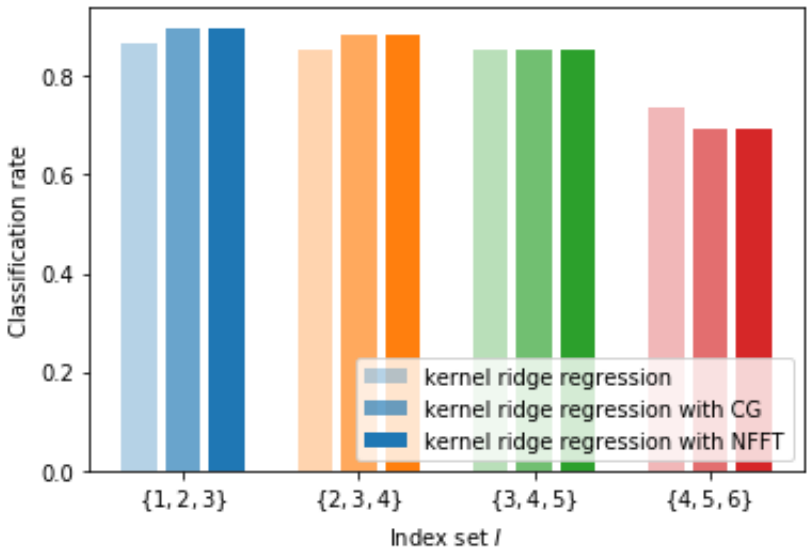

Figure 5.2: *Cryotherapy Data Set* - Classification rate for distinct index sets I using the ordinary *kernel ridge regression* with *scipy.sparse.linalg.cg*, the *kernel ridge regression* with a self-built *CG-method* and the *kernel ridge regression* with *NFFT* approach, with optimal σ and $\lambda = 1$

For our self-built *CG-method*, we chose the tolerance for convergence to be 10^{-5}, so that the algorithm stops if $\|r_i\|^2 = \langle r_i, r_i \rangle < 10^{-5}$. It was applied for the respective second and third bar. Figure 5.2 shows, that the *scipy.sparse.linalg.cg* function has another default value for that tolerance. Indeed, this algorithm's stopping criterion is slightly different, namely $\|r_i\| \leq 10^{-5} \cdot \|f\|$. This explains the deviations between the respective first bars and the other two. Evidently, the *NFFT-based fast summation* yields hyper-accurate approximations in such a way that applying the *NFFT* approach does not affect the classification rate at all. The respective prediction qualities match exactly. For the index set $I = \{3, 4, 5\}$, the classification rate coincides for all 3 computation methods. By contrast, the results deviate by 0.04 for $I = \{4, 5, 6\}$ and by 0.03 for the other two windows. These deviations keep within limits. Besides, they only come into existence, because the stopping criterion for the *CG-methods* do not agree. The main takeaway from Figure 5.2 is that our *NFFT* approach approximates matrix-vector products excellently per se.

Next, we want to examine the number of required iterations by the self-built *CG-method* with and without the *NFFT-based fast summation* and compare the residual norms. This yields Table 5.1.

Index set	*CG* iteration i	$\lvert\lVert r_i^{CG}\rVert_2^2 - \lVert r_i^{NFFT}\rVert_2^2\rvert$
$I = \{1,2,3\}$	1	$6.220286707048217e-06$
	2	$4.839732689276843e-06$
	3	$6.425458556609358e-11$
	4	$1.1459281152267947e-13$
$I = \{2,3,4\}$	1	$9.53729475838827e-07$
	2	$5.426287970067278e-07$
	3	$2.5469859246237607e-08$
	4	$2.957958648893702e-14$
$I = \{3,4,5\}$	1	$3.1825768348370254e-05$
	2	$6.17600354206084e-07$
	3	$3.6416455621491217e-09$
	4	$6.88424171531731e-11$
$I = \{4,5,6\}$	1	$2.4157464520023098e-02$
	2	$1.198812684528261e-03$
	3	$2.292740398947002e-04$
	4	$5.368536653147307e-05$
	5	$1.5825118318456505e-06$
	6	$3.59507655486551e-07$
	7	$1.802773232507821e-08$

Table 5.1: *Cryotherapy Data Set* - Deviation of the residual norms for the self-built *CG-method* with and without the *NFFT-based fast summation*, with optimal σ and $\lambda = 1$

We ascertain, that the number of iterations required by the self-built *CG-method* equals the number of taken iterations, when applying the *NFFT* approach. The deviation of the residual norms is small from the beginning and decreases with every iteration. The kernel, which is based on the index set $I = \{4,5,6\}$, consistently produces the worst results. Moreover, it requires the most iterations within the *CG-method* by far. I. e. for this kernel, the *CG-method* converges clearly slower than for the others.

We have already observed, that applying our *NFFT* approach does not perceptibly affect the *CG-method*. The number of iterations is equal and the residual norms

are nothing but subtly different, so that the *CG-method* converges equally fast. Furthermore, we obtain the exact same classification rates. All that remains to show is the improvement in computational complexity. That would confirm the success of our approach.
Therefore, we measure the execution time for all 3 computation methods. Figure 5.3 illustrates the results for distinct index sets I.

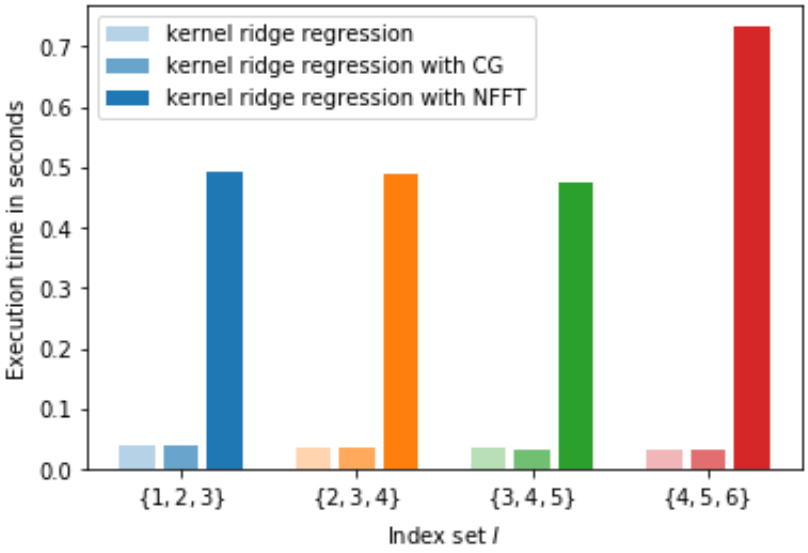

Figure 5.3: *Cryotherapy Data Set* - Execution time for distinct index sets I using the ordinary *kernel ridge regression* with *scipy.sparse.linalg.cg*, the *kernel ridge regression* with a self-built *CG-method* and the *kernel ridge regression* with *NFFT* approach, with optimal σ and $\lambda = 1$

We must find that using the *FastAdjacency* package for approximating the kernel matrix K and products with vectors does not reduce the execution time in our setting. To the contrary, this procedure takes up to twenty times as long as the other two. We wonder, what slows this process down so heavily. Therefore, we measure the time needed to compute or approximate the kernel matrix K, now.

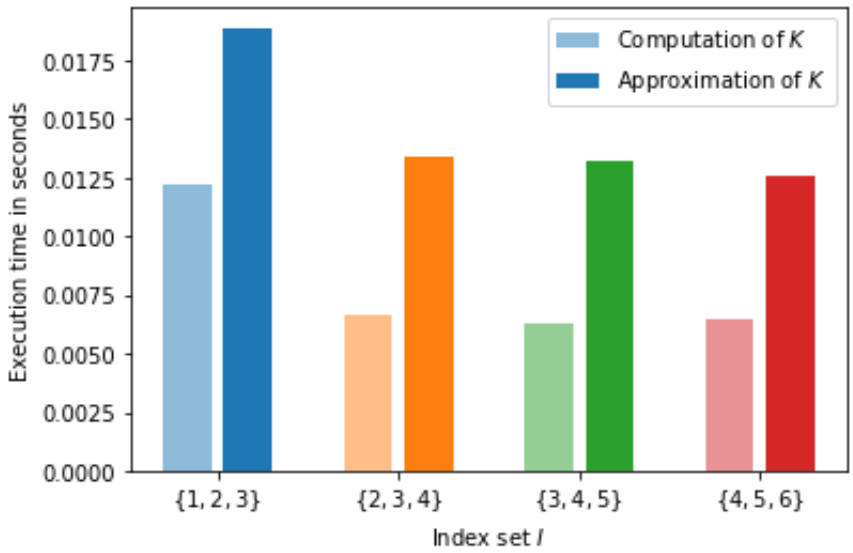

Figure 5.4: *Cryotherapy Data Set* - Execution time for computing or approximating the kernel matrix K for distinct index sets I

Figure 5.4 illustrates that the approximation of K using the *FastAdjacency* package is not solely responsible for the huge deviations in execution time. Actually, performing the approximation of K takes maximally twice as long as computing it, where both procedures run fairly quickly.
According to the problem set description in [20], the *FastAdjacency* package is targeted at the case of large N. For the *Cryotherapy Data Set*, $N = 90$. This is by far no large dimension. Hence, we cannot play to the strengths of the *NFFT-based fast summation*. Moreover, the *curse of dimensionality* does not arise when computing K. Because of that, our expectations regarding improved execution times with the *NFFT* approach are not satisfied for the *Cryotherapy Data Set*.

To finally ascertaining the qualities of our *NFFT* approach, we analyse the *Skin Segmentation Data Set* with $N = 245057$ samples and $n = 3$ features, now. It fits perfectly in the target group of the *FastAdjacency* software. The data set "is collected by randomly sampling B, G, R values from face images of various age groups (young, middle and old), race groups (white, black and asian), and genders" [21]. The class labels $y_i \in \{-1, 1\}$ indicate, if an entry corresponds to a skin ($y_i = 1$) or a non-skin sample ($y_i = -1$). Table 5.2 illustrates the first and the last 5 samples.

B	G	R	Label
74	85	123	1
73	84	122	1
72	83	121	1
70	81	119	1
70	81	119	1

B	G	R	Label
163	162	112	-1
163	162	112	-1
163	162	112	-1
163	162	112	-1
255	255	255	-1

Table 5.2: First and last 5 entries in the *Skin Segmentation Data Set*

As mentioned above, the total learning sample size is 245057, consisting of 50859 skin samples and 194198 non-skin samples. Performing the *kernel ridge regression* for such giant data sets requires powerful computers. For convenience, we reduce the number of samples before analysing it, with maintaining the proportion of skin samples to non-skin samples and selecting the samples randomly using the *train_test_split* function with $train\_size = 0.25$, once again. Since the *Skin Segmentation Data Set* possesses 3 features, the kernel matrix is based on the index set $I = \{1, 2, 3\}$. For details on the design of the downsized data sets S_i, we refer to Table 5.3.

We already demonstrated in Figure 5.2 and Table 5.1, that the *NFFT-based fast summation* yields extremely accurate approximations. This is confirmed by Figure 5.5. Again, the *kernel ridge regression* with self-built *CG-method* and the *kernel ridge regression* with *NFFT* approach yield exactly the same classification rate for all data sets S_i. Furthermore, we achieve an excellent prediction quality, even though we have not even tuned any parameter. The maximal classification rate is obtained at 0.97 for the data set S_6, where $N_6 = 5000$. I. e. our system predicts class affiliations correctly in 97 per cent of all cases. This performance is highly satisfying, considering that we might even have the potential to improve this result by tuning the parameters.

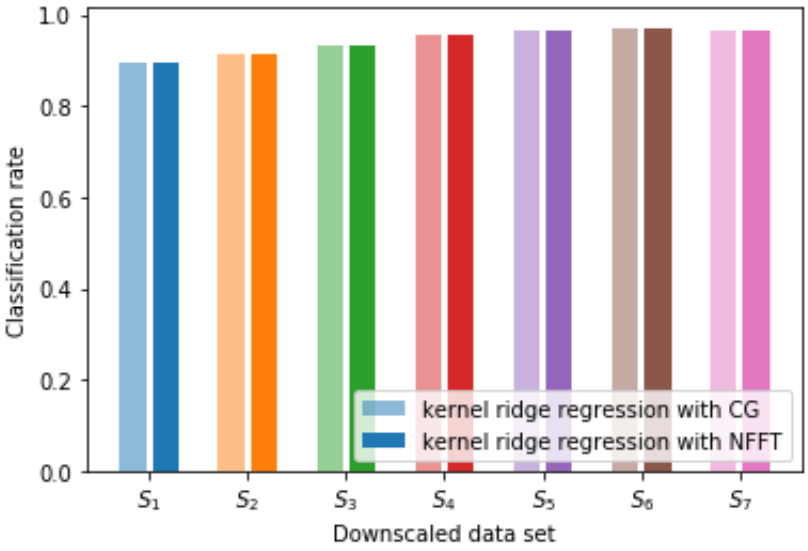

Figure 5.5: *Skin Segmentation Data Set* - Classification rate for distinct dowscaled data sets S_i using the *kernel ridge regression* with a self-built *CG-method* and the *kernel ridge regression* with *NFFT approach*, with $\sigma = 1$ and $\lambda = 1$

Next, we measure the execution time, when performing the *kernel ridge regression* with self-built *CG-method* and with the *NFFT-based fast summation* for differently downscaled data sets. The results are illustrated in Table 5.3 and Figure 5.6.

Data set	Number of samples		Execution time in seconds	
	Skin	Non-skin	*CG*	*NFFT*
S_1	25	100	0.0772240161895752	0.5077226161956787
S_2	50	200	0.29552197456359863	0.7269303798675537
S_3	100	400	1.1209490299224854	1.4271478652954102
S_4	250	1000	6.874807119369507	6.042292833328247
S_5	500	2000	27.057647705078125	21.784718990325928
S_6	1000	4000	106.40446710586548	84.62689971923828
S_7	1500	6000	238.30217266082764	189.48373556137085

Table 5.3: *Skin Segmentation Data Set* - Execution time for distinct downscaled data sets S_i using the *kernel ridge regression* with a self-built *CG-method* and the *kernel ridge regression* with *NFFT* approach, with $\sigma = 1$ and $\lambda = 1$

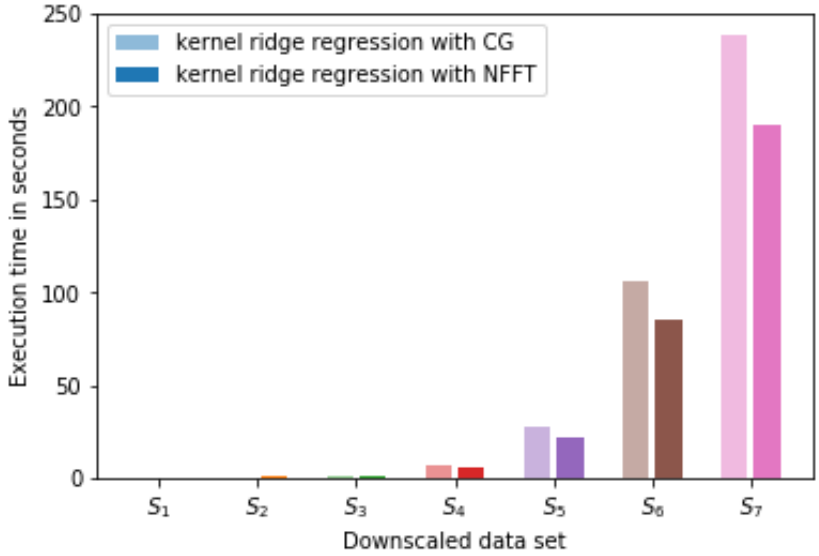

Figure 5.6: *Skin Segmentation Data Set* - Execution time for distinct downscaled data sets S_i using the *kernel ridge regression* with a self-built *CG-method* and the *kernel ridge regression* with *NFFT* approach, with $\sigma = 1$ and $\lambda = 1$

Table 5.3 and Figure 5.6 verify our explanation from above. The results in Figure 5.3 were only different than expected, because the *FastAdjacency* package targets data sets with a large number of samples. Thus, the *Cryotherapy Data Set* with $N = 90$ is just not suitable for this application.
By Figure 5.5, we can confirm yet again that the *NFFT-based fast summation* yields hyper-accurate approximations. Table 5.3 and Figure 5.6 clarify, that the *NFFT* approach gathers strength with increasing numbers of samples, whereas the *curse of dimensionality* arises with the ordinary *CG-method*.

So far, we applied the *NFFT-based fast summation* only on the *kernel ridge regression*. Realising this approach on the *semi-supervised learning* will be the subject of future research.

6 Conclusion

In this thesis, we have successfully reduced the computational complexity of performing learning methods on high-dimensional data. This was possible due to the computational power of *NFFT-based fast summation*. Applying the *FastAdjacency* package, we compute hyper-accurate approximations of the matrix-vector products $K\alpha_i$, without ever setting up the full matrix. In doing so, we nearly meet a linear scaling, while the ordinary procedure requires $\mathcal{O}(N^2)$ computations.
Basis for this approach is a favourably designed kernel matrix K, though. Thus, we deeply delved into the theory of kernels and defined a suitable kernel function.
In our numerical experiments, we compared the classification results yielded by the ordinary *kernel ridge regression* and the *semi-supervised learning*. We found that the *semi-supervised learning* predominantly achieves better prediction qualities. Moreover, we demonstrated the necessity of crucial steps, such as scaling the data and tuning the parameters. However, to these ordinary procedures are set limits. With increasing number of samples, the *curse of dimensionality* arises, what motivates the demand for speeding up the learning processes for large data. We developed a *kernel ridge regression* method, which follows the *NFFT* approach. It yields exact same classification results as the ordinary *kernel ridge regression*. Its execution time is unconvincing for small data sets, though. In this case, the ordinary *kernel ridge regression* clearly is the method of choice. But the *NFFT* approach gathers strength with increasing number of data. Accordingly, this method is only targeted at high-dimensional cases.
During originating this thesis, new research questions arose again and again. We could not include all of them in this thesis. For instance, we did not examine the classification results for combining several kernel matrices. Moreover, we did not apply the *NFFT-based fast summation* on the *semi-supervised learning* either. Addressing these questions will be subject of future research.

References

[1] John Shawe-Taylor, Nello Cristianini, et al. *Kernel methods for pattern analysis*. Cambridge university press, 2004.

[2] Thomas Hofmann, Bernhard Schölkopf, and Alexander J Smola. Kernel methods in machine learning. *The annals of statistics*, pages 1171–1220, 2008.

[3] Dominik Alfke, Daniel Potts, Martin Stoll, and Toni Volkmer. NFFT meets Krylov methods: Fast matrix-vector products for the graph Laplacian of fully connected networks. *Frontiers in Applied Mathematics and Statistics*, 4:61, 2018.

[4] Yang You, James Demmel, Cho-Jui Hsieh, and Richard Vuduc. Accurate, fast and scalable kernel ridge regression on parallel and distributed systems. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 307–317, 2018.

[5] Bernhard Scholkopf and Alexander J Smola. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. Adaptive Computation and Machine Learning series, 2018.

[6] Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.

[7] Matthias Hein, Simon Setzer, Leonardo Jost, and Syama Sundar Rangapuram. The total variation on hypergraphs-learning on hypergraphs revisited. In *Advances in Neural Information Processing Systems*, pages 2427–2435, 2013.

[8] César R. Souza. Kernel Functions for Machine Learning Applications. `http://crsouza.blogspot.com/2010/03/kernel-functions-for-machine-learning.html`, 2010 (accessed: 14 September 2020).

[9] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012.

[10] Yaohua Tang, Weimin Guo, and Jinghuai Gao. Efficient model selection for support vector machine with Gaussian kernel function. In *2009 IEEE Symposium on Computational Intelligence and Data Mining*, pages 40–45. IEEE, 2009.

[11] Tie Liang and Alina A von Davier. Cross-validation: An alternative bandwidth-selection method in kernel equating. *Applied Psychological Measurement*, 38(4):281–295, 2014.

[12] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[13] University of California Irvine. Center for Machine Learning and Intelligent Systems - Cryotherapy Data et. `https://archive.ics.uci.edu/ml/datasets/Cryotherapy+Dataset+`, 2018 (accessed: 28 September 2020).

[14] Kaggle. Titanic: Maachine Learning from Disaster. `https://www.kaggle.com/c/titanic/`, accessed: 06 October 2020.

[15] Folkmar Bornemann. *Numerical linear algebra: a concise introduction with MATLAB and Julia*. Springer, 2018.

[16] Magnus R Hestenes, Eduard Stiefel, et al. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49(6):409–436, 1952.

[17] Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[18] Steven L Brunton and J Nathan Kutz. *Data-driven science and engineering: Machine learning, dynamical systems, and control*. Cambridge University Press, 2019.

[19] Gerlind Plonka, Daniel Potts, Gabriele Steidl, and Manfred Tasche. *Numerical Fourier Analysis*. Springer, 2018.

[20] Dominik Alfke. FastAdjacency: Python extension to compute fast approximate multiplications with Gaussian adjacency matrices. `https://github.com/dominikalfke/FastAdjacency`, 2018 (accessed: 24 September 2020).

[21] University of California Irvine. Center for Machine Learning and Intelligent Systems - Skin Segmentation Data Set. https://archive.ics.uci.edu/ml/datasets/Skin+Segmentation, 2017 (accessed: 05 October 2020).

List of Figures

List of Tables

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Studentenservice – Zentrales Prüfungsamt

Selbstständigkeitserklärung

Name: Vorname: geb. am: Matr.-Nr.:	**Bitte beachten:** 1. Bitte binden Sie dieses Blatt am Ende Ihrer Arbeit ein.

Selbstständigkeitserklärung*

Ich erkläre gegenüber der Technischen Universität Chemnitz, dass ich die vorliegende selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch nicht als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

Datum: ……. Unterschrift: ……

* Statement of Authorship

I hereby certify to the Technische Universität Chemnitz that this thesis is all my own work and uses no external material other than that acknowledged in the text.

This work contains no plagiarism and all sentences or passages directly quoted from other people's work or including content derived from such work have been specifically credited to the authors and sources.

This paper has neither been submitted in the same or a similar form to any other examiner nor for the award of any other degree, nor has it previously been published.