EVOLUTIONARY OPTIMIZATION OF KERNEL MACHINES

Arjan Gijsberts

EVOLUTIONARY OPTIMIZATION OF KERNEL MACHINES

by

Arjan Gijsberts

A thesis submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE

presented at

Delft University of Technology
Faculty of Electrical Engineering, Mathematics, and Computer Science
Man-Machine Interaction Group

August 2007

Author
Arjan Gijsberts

Title
Evolutionary Optimization of Kernel Machines

MSc presentation
31st August 2007

Man-Machine Interaction Group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

Laboratory for Integrated Advanced Robotics
Department of Communication, Computer, and System Sciences
Faculty of Engineering
University of Genoa
Viale F. Causa, 13
16145 Genoa
Italy

Department of Robotics, Brain, and Cognitive Sciences
Italian Institute of Technology
Via Morego, 30
16163 Genoa
Italy

Keywords
Machine learning, artificial intelligence, kernel machines, support vector machines, least squares support vector machines, evolutionary computation, evolution strategies, genetic programming, robotics.

“Jene neue Partei des Lebens, welche die grösste aller Aufgaben, die Höherzüchtung der Menschheit in die Hände nimmt, eingerechnet die schonungslose Vernichtung alles Entartenden und Parasitischen, wird jenes Zuviel von Leben auf Erden wieder möglich machen, aus dem auch der dionysische Zustand wieder erwachsen muss.” – FRIEDRICH WILHELM NIETZSCHE

ABSTRACT

Machine learning techniques have become increasingly more sophisticated and useful over the last decades. In particular, the class of Kernel Machines has been shown to outperform many other techniques in various classification and regression problems. Its performance, however, depends to a large extent on the kernel function and hyperparameters that are being used. Unfortunately, there is no analytic method to aid the user in finding a proper kernel function and hyperparameters. This selection procedure normally comes down to a “trial-and-error” approach, which limits the user in the range of kernel functions that can be considered. In this report we present an automated approach for finding a good kernel function and optimal hyperparameters using Evolutionary Computation techniques. Evolutionary Computation is class of optimization techniques that is inspired by biological evolution. Potential solutions to the problem under consideration are iteratively generated, mutated, recombined, and evaluated. The search process converges to good solutions, since it biases reproduction toward the fittest individuals (cf. “survival of the fittest”).

Two separate evolutionary models are proposed in this report. The first of these two models uses Evolution Strategies to rapidly find optimal hyperparameters. The second model aims to improve the generalization capacity of the machine by evolving complex kernel functions using Genetic Programming. Empirical studies show that our Evolution Strategies approach is able to find competitive hyperparameters, as compared with traditional methods, in less time for regression problems. Classification problems are problematic, due to the discontinuity of the error surface. Nonetheless, it has to be noted that the approach is still able to find reasonable solutions in a time efficient manner. The Genetic Programming approach, however, is shown to improve the generalization capacity of the machine only marginally. In most practical applications this minor improvement will not justify the high computational requirements of the model.

Lastly, we present a study on the reliability of Kernel Target Alignment as a performance measure for Kernel Machines. The main advantage of Kernel Target Alignment is the computational efficiency, as compared with other performance measures. Nonetheless, our empirical study suggests that Kernel Target Alignment does not reliably approximate the true generalization performance of a Kernel Machine. Therefore, the use of this measure as an objective function should be avoided.

PREFACE

Writing the preface for this Master's thesis is quite frankly a strange sensation. In no other of part this thesis the perspective of the author and the reader will be further apart than during this preface. To the reader it will mark the very beginning of the report, whereas for the author it (usually) is the very last element that has to be written. On part of the author it inherently causes reflections on the many months of hard work that has been invested to produce the work that is in front of you. Of course, these reflections include the gratitude toward many persons that have contributed to this work in one way or another.

First of all, I would like to thank Giulio Sandini, as head of the LiraLab, and Giorgio Metta, as my supervisor, for the wonderful opportunity they offered me to perform my thesis work here in Genoa. Given that I was hoping to combine the concepts "artificial intelligence" and "Italia" in my thesis work, it is not very hard to imagine how grateful I am to them. Of course, many thanks as well to all the others at the LiraLab and the Italian Institute of Technology. It has absolutely been a very pleasant experience. In particular I would like to thank Michael Bucko, Francesco Orabona, and Vadim Tikhanoff, with whom I have shared an apartment. Besides this, Francesco also helped me to grasp certain aspects of machine learning and in particular Kernel Machines. Our discussions on the topic will surely have shaped certain parts of this work. A special word of thanks also to Giorgio Metta (again) and Carlos Beltran for helping me to get all the computational power I needed for my experiments. At the university in Delft I would like to thank Leon Rothkrantz for his support on this "external" thesis and his valuable comments on my report. Especially in these last hectic weeks he has shown me to be very flexible and willing to support his students.

Clearly, I could never forget to mention my family here and to express my gratitude. Especially when it comes to my education no effort has ever been too big for them. Jan, Gerjo and Gert-Jan Gijsberts, many thanks for the everlasting support and care. A very special gratitude I would like to express as well toward my aunts Wil and Magda Rijkeboer and Ger Kroon, who have never stopped to show me their care.

One person that has enriched my stay here in Italy is my girlfriend Roberta Basili. I would like to thank her for the valuable introductions that she gave me into many aspects of the Italian culture and for all the wonderful moments that we have spent together.

Finally, my stay in Genoa would not have been possible without the aid from the following parties:

- Stichting Fundatie van de Vrijvrouwe van Renswoude,
- The LiraLab of the University of Genoa,
- The ERASMUS program of the European Commission.

Arjan Gijsberts

Genoa, Italy

24th August 2007

NOTATION

$y \in \mathcal{Y}$	output and output space
$\mathbf{x} \in \mathcal{X}$	input and input space
n	dimension of the input space
$\langle \mathbf{x}, \mathbf{z} \rangle$	inner product of $\mathbf{x}$ and $\mathbf{z}$
$\phi(\cdot)$	mapping into feature space
N	dimension of feature space
$\mathcal{H}$	feature space
$k(\mathbf{x}, \mathbf{z})$	kernel function $\langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle$
$f(\cdot)$	real valued function
$\hat{y}$	predicted output
$\mathbf{w}$	weight vector
b	bias
R	radius of smallest containing sphere
$\boldsymbol{\alpha}$	model parameters
$\| \cdot \|_p$	p-norm, default is 2-norm
$\mathbb{N}$	natural numbers
$\mathbb{R}$	real numbers
$\mathbf{K}$	kernel matrix
$\mathbf{I}$	identity matrix
ϵ	prediction error
$\mathcal{E}$	estimated error
S	training set
ℓ	training set size
$\boldsymbol{\theta}$	hyperparameters
ξ	slack variable
$\mathcal{F}$	function space
μ	size of ES parent population
λ	size of ES offspring population
$\mathbf{c}$	object vector for chromosome
$F(\cdot)$	objective function
$\mathcal{N}(\mu, \sigma)$	normal distribution with mean μ and standard deviation σ
$\mathcal{O}$	asymptotic complexity
r_P	Pearson product-moment correlation coefficient
τ_K	Kendall's τ rank correlation coefficient

CONTENTS

Chapter 1 INTRODUCTION

Over the last half decade the computer has shown to be a versatile instrument, which can be used for a wide variety of problems. As computers have grown more powerful, we humans have become more ambitious regarding the tasks that we assign to them. Perhaps the most audacious of all is to let machines behave in a human-like, intelligent manner. We desire to move into an era in which the computer is no longer a mere instrument, but instead becomes an autonomous, cooperative companion. This implies that the machine is able to behave in an "intelligent" way and can develop skills on its own[1].

A clear example in which a machine is expected to behave intelligently is a *humanoid robot*, which is a type of robot that is modeled after humans in both the physical and the behavioral sense. Obviously, from a robot that *looks* like a human we expect that it *behaves* like a human as well. A recent approach for making a robot behave like humans is to let it undergo a similar developmental process as we humans do. This means that the machine is not preprogrammed with knowledge and skills, but mostly obtains them by means of interaction with its environment. In order to do so, we need techniques and algorithms that help the machine to "learn" from experience, i.e. to infer knowledge from observations.

The field of *machine learning* is concerned with developing such techniques. A major focus in machine learning research is on automatically extracting useful information from data, a process which is named *pattern recognition*. An important aspect in pattern recognition in general – and within the context of humanoid robotics in particular – is the *generalization capacity* of the machine. Generalization is the act of responding to an unfamiliar situation in a manner similar to responding to a trained situation. One class of machine learning algorithms that has been particularly successful in the last decade, with regard to generalization performance, is that of *Kernel Machines*. The success of its practical application, however, depends to a great extent on certain preliminary settings of the algorithm, which need to be configured by the user.

In this report we propose a model for finding optimal settings for Kernel Machines efficiently and without any human intervention. Our model uses a technique inspired by biological *evolution* to find these

[1]The definition of "intelligence" is point of a perdurable philosophical debate. We will not replicate this debate here, as it is irrelevant in the context of this report.

settings. In short, several possible solutions are randomly generated and tested for their quality, i.e. their *fitness*. Then the process continues by constructing a new generation of solutions based on those solutions that have an above-average fitness. This way the search process iteratively converges to solutions of high quality. Furthermore, we propose a second model that uses similar evolutionary techniques to improve the generalization capacity of Kernel Machines.

We will begin by explaining the context of this study, after which we will briefly introduce the field of machine learning. We will emphasize on the two machine learning techniques that are relevant for this report, namely Kernel Machines and *Evolutionary Computation*. This will bring us to our problem definition. We will conclude this chapter with the outline of this report.

1.1 The RobotCub Project

The work presented in this report has been conducted as part of the *RobotCub* project. This project is a research initiative on developing an open humanoid robotic platform. It is funded by the European Union and is a collaboration of several universities and research institutes spread over the world. The purpose of developing this platform is to facilitate research in embodied cognition, both from a psychological as well as a robotic point of view [72]. One particular aim of the project is to aid research on *epigenetic robotics*, or the much similar field of *developmental robotics* [67]. In this paradigm the cognitive development of the robot is inspired by the development of humans themselves. The belief is that cognition emerges while the subject, either human or robot, has experiences and makes observations. This requires a dynamic interaction with the environment, in forms such as manipulation, locomotion, interpretation, perception, and communication. The cognitive capabilities of the subject then self-organize (i.e. it *learns*), thanks to these experiences and observations. Obviously, interaction with an environment implies that the robot is embodied, otherwise locomotion and manipulation would not be possible. The epigenetic approach to cognitive skills has the following advantages:

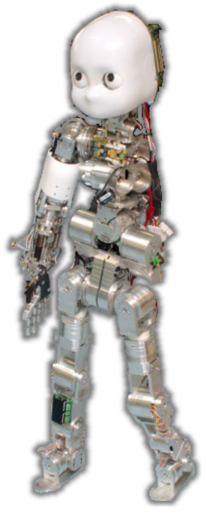
- A robot that is programmed to *learn* will be able to adapt to changing environments. It is intractable to program a priori the appropriate knowledge and skills for every possible situation that a robot might encounter; therefore, it must learn how to deal with unfamiliar situations. This requires the capacity to generalize in the cognitive system of the robot.

- It is less difficult to teach a robot at the hand of examples than it is to literally program this knowledge. A great deal of human knowledge is extremely hard to define in such a precise manner that it is useful for a machine. For instance, try to think of a *precise* definition of the word "game". Note that we humans usually have no problem whatsoever to classify a given activity as a "game", despite not having a precise definition at hand[2].

- The ability to learn from experience means that the machine can continuously improve on itself. Hence, its cognitive capacity is not necessarily limited to the knowledge it has inherited from its creators.

- The study on human development can benefit from letting a robot undergo a similar cognitive development. For instance, a hypothesis on human cognitive development could be confirmed experimen-

[2]This particular thought experiment was put forward by philosopher Ludwig Wittgenstein in his *Philosophische Untersuchungen* [119].


(a) iCub

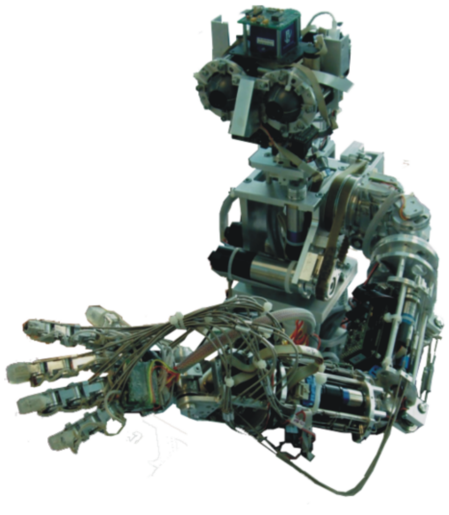

(b) James

Figure 1.1: Two humanoid robots at the Italian Institute of Technology. The *iCub* is currently being developed as part of the RobotCub project. *James* is a predecessor of the iCub developed by the Laboratory for Integrated Advanced Robotics of the University of Genoa. The picture of James is taken from [9].

tally using robotics. In many situation it is not regarded ethical to perform similar experiments on real humans.

The influential paper *"Elephants don't play chess"* marked the beginning of the approach of embodied cognition in robotics [7]. In this paper Brooks claims that physical interaction with an environment is a primary source of constraint on the design of intelligent systems. The traditional approach within the field of Artificial Intelligence and robotics, however, had up until that point in time emphasized on abstract symbolic representations. Since then, the approach of emergent cognition and embodiment has been applied in the development of various (humanoid) robots, e.g. [8, 18].

The particular robot being developed within the RobotCub project is the *iCub*, which is physically modeled after a $3\frac{1}{2}$-years old child. The robot has 53 degrees of freedom and must eventually be able to crawl. The robot is situated at the Italian Institute of Technology, although more copies will be made available to other research institutes after completion. Also situated at the Italian Institute of Technology is *"James"*, which is an upper-torso humanoid robot that predates the iCub. This robot is approximately the size of a ten-years old boy and has 22 degrees of freedom. Both humanoid robots are shown in Fig. 1.1.

The software framework that runs on the robots is *Yet Another Robot Platform* (*YARP*) [71]. The main aim of YARP is to simplify the development of robots by creating an abstraction layer for devices (e.g. a camera or sensor) and other common entities in robotics (e.g. image representations). This is done by encapsulating devices and other modules in so-called *ports*. A port is a transmitter or receiver of information, which can be connected to other ports using the *observer* design pattern [41]. The information that travels over these network channels can be either binary data (e.g. an image stream from a camera) or textual data (e.g. sensor data). The architecture of the robot as a whole can then be defined as a set of ports and their interconnections, which together form a directed graph. This structure increases the modularity of the system and the processes can be arranged over multiple physical machines. Furthermore, YARP is platform independent, thus machines with different operating systems can cooperate in one global system. Lastly, YARP is a publicly available open-source project that is written in the C++ programming language.

1.2 Machine Learning

As explained above, machine learning deals with extracting useful information from data. Usually, we can refer to this data as observations and we wish to recognize one or more patterns. For instance, a pattern found in the data can be used to to predict an properties (e.g. an output value) of new, unseen observations. That is, the machine is said to be able to *generalize*. Usually, three types of learning problems are distinguished, based on the type of feedback [97]. The first and probably most common type of machine learning is *supervised learning*, in which an algorithm creates a function that maps inputs to desired outputs. This requires, of course, that the desired outputs are available to the machine during the training phase. In other words, the training examples must be *labeled*. These labels can be provided either by a teacher or be obtained by means of measurements, depending on the type of problem. For instance, the labels will be provided by a teacher for a task like face recognition. On the other hand, measurements will be more likely in case the machine has to learn a physical phenomenon. Supervised learning is the type of machine learning that we will consider in this report.

Opposed to supervised learning is *unsupervised learning*, in which no labeled examples are available. This technique can be used to model a set of inputs, for instance to cluster the observations. It is important to realize that a purely unsupervised machine cannot have any notion about the correctness of its functioning. Intermediate variants of supervised and unsupervised learning fall in the category of *semi-supervised learning*, in which the set of examples is only partially labeled.

Another popular machine learning technique – especially within the context of developmental robotics – is *reinforcement learning* [110]. In this type of learning the machine has to learn a model by means of action and reaction. The machine itself initiates an action and a corresponding reaction is “given” by the environment in the form of a consequence. This consequence can either be a positive or negative reward. In other words, the machine learns to act in an environment by observing the impact of its actions and trying to maximize its utility. The model that is obtained using reinforcement learning is a *stimulus-response association* [103].

All of these types of machine learning can be encountered in the development and functioning of a humanoid robot. For instance, consider the following situations:

- A robot needs to communicate with people. Therefore, it is important to understand both from visual and auditory cues which person is communicating with the robot. An important visual cue is the gaze direction of the person that is speaking.
- When a human-like robot is developing in an environment with multiple persons, then it must be able to uniquely identify these persons. This can be done, for instance, using face recognition.
- For locomotion and manipulation skills it is necessary that the robot is aware of its morphology. In a robust robotics platform this morphology should be learned – as we humans do – instead of being preprogrammed.
- In order to explore an environment it is necessary that a robot is able to grasp an object, for instance, to see how this affects the object and to learn certain properties (e.g. the weight) of the object. Grasping an object involves locating the object visually, moving toward the object if it is not within reach, then

reaching the hand toward the object, and finally grasping it. These steps can incorporate machine learning at a visual level (e.g. object recognition, object tracking, and location estimation), a motor-sensory level (e.g. moving its hand toward a certain position), and a cognitive level (e.g. predicting certain properties of the object) [78]. A prior belief about certain properties of the object is useful, since these properties determine to a great extent how the object should be approached and which type of grasp should be applied.

- Locomotion of the robot requires that it is able to plan a path from its current position toward the goal position. Often this involves creating a map of the environment and using visual information for object avoidance. Furthermore, the motor control system of the robot needs to take various aspects of the environment into account during locomotion, such as elevation (e.g. stairs or slopes) or the type of ground surface.

These situations are just a small set of examples in which machine learning plays an important role for a humanoid robot. Nonetheless, we wish to emphasize that the field of machine learning is much broader applicable than just robotics. It has been used in many other fields, such as economics, physics, biology, and medicine. In many of these applications the machines are able to recognize patterns that humans would not have been able to identify.

Machine learning techniques can be based on different theories, e.g. based on logical or statistical inference. One statistical method that has been shown to give excellent results is the class of Kernel Machines. A particular instance of this class is the Support Vector Machine, which outperformed many existing techniques on certain tasks over the last decade. The different algorithms within the Kernel Machine paradigm share the fact that similarity between observations is measured by a so-called *kernel function*. The quality of the machine, however, depends heavily on the kernel function and the corresponding parameters (i.e. the hyperparameters). It is therefore important to optimize both the selected kernel function and the hyperparameters. However, selecting a good kernel function can take a serious amount of time, which is why the user usually has to resort to a small set of well-known kernel functions. Results could possibly be improved upon by considering a larger set of kernel functions. This way a kernel function can be selected that is a is suited for the problem under consideration.

Another interesting approach in machine learning are techniques that are inspired by biology. This class of machine learning techniques include Artificial Neural Networks, Swarm Optimization and Evolutionary Computation. The latter is a very versatile paradigm, which can be applied on optimization, classification, regression, and even programming problems. In this paradigm a set of initial potential solutions is randomly generated. Then the quality of these solutions is evaluated and the process converges to good solutions by applying mutation and natural selection, similar to biological evolution.

1.3 Problem Definition

Kernel Machines are a relatively new and powerful class of machine learning algorithms. This makes them very interesting candidates for being applied within a humanoid robot. However, applying the technique is by no means trivial, as it requires a careful selection of the kernel function and hyperparameters. Most commonly this selection is done by an expert and it has been referred to as being more of an “art” than a “science”. It would be desirable to completely automate this selection procedure, so that Kernel Machines can easily be used at their full power by less experienced users. Perhaps more importantly, a fully automated approach also implies that a robot could perform this routine autonomously. A possible scenario in which this could be beneficial is when a robot would wish to retrain one of its systems. With unpredictable environments it is not very unlikely that such a scenario might happen. For instance, the robot could be damaged, after which a new training session could help to regain some its original function. Compare this to humans that have to learn motor skills again after a serious injury.

We believe that Evolutionary Computation would be suitable for optimizing both the kernel function and the hyperparameters. It is a highly generalized paradigm that can be applied to many problems. In this study we want to investigate the following two research questions:

1. *Can we use Evolutionary Computation techniques in order to rapidly optimize the hyperparameters of a Kernel Machine?*
2. *Can we use Evolutionary Computation techniques to evolve an optimal kernel function for a Kernel Machine given a certain problem?*

We want to note that both questions have a slightly different goal. In the first question we aim to find optimal hyperparameters in a way that is computationally less demanding than the common optimization method (i.e. a grid search). Therefore, we do not emphasize on finding necessarily a better solution, but instead on *finding a comparable solution in less time*. In the second research question we do not only consider optimizing the hyperparameters, but also automate the selection of a kernel function. Currently, there is no common way to select a suitable kernel function, other than testing two or three well-known kernel functions by hand. The total class of potential kernel functions is, however, much larger than just these well-known kernels. We wish to investigate if we can exploit this fact and find kernel functions that perform better than these default functions. In short, for the second research question we emphasize on *improving the quality of the solution*, with only a minor priority on the computational expenses.

Previously, we noted that Evolutionary Computation works by evaluating potential individuals and assigning a certain fitness to them, similar to biological evolution. This raises a very important question, namely, *how do we define the quality of a solution*? Not only do we have to define what quality is in the context of kernel functions and hyperparameters, we also need to formalize a quantitative measure for this notion of quality. This is the third (minor) research question that we wish to answer in this report. More formally, we state this question as:

3 *What measure of quality can be used for kernel functions and hyperparameters, in the context of the previous two research questions?*

1.4 Outline

In this report we will propose solutions to the two main research questions that we just described. These solutions will be explained, evaluated, and reflected upon in the course of 5 parts:

- We commence with describing the *Preliminaries* in Part I. we will describe all the theory regarding machine learning that is necessary to come to our model and implementation. Kernel Machines will be explained in detail in Chapter 2, at the hand of Support Vector Machines and the closely related Least Squares Support Vector Machines variant. Evolutionary Computation will be dealt with in

Chapter 3. In that chapter we will learn the basics of Genetic Algorithms, Evolution Strategies and Genetic Programming. The first part will be concluded with Chapter 4, in which we will give a thorough overview of the related work. We will primarily – but not exclusively – describe approaches on hyperparameter and kernel selection using evolutionary techniques.

- In Part II we will explain our *Model and Implementation*. We will propose two models, one for each of the two main research questions posed above. These models will be described in Chapter 5. The implementation phase of these two models will be covered in Chapter 6.

- The *Experimental Results* of our two models will be described in Part III. The primary results of our two proposed models will be dealt with in Chapter 7. Previously, however, we also identified the sub-question of which fitness measure is suitable for use with a Kernel Machine. This sub-question is covered in Chapter 8.

- Part IV will contain our *Conclusions*. These will described in Chapter 9, as well as directions that we have identified for future work.

- Finally, in Part V we present the appendices, which contain documentation on the design of our implementation and example configuration files that have been used for our experiments.

Part I

Preliminaries

Chapter 2

KERNEL MACHINES

The field of machine learning has developed rapidly over the last decades. This field of research deals with learning a certain model from empirical observations. Ideally, the learning is done in such a way that the knowledge can be generalized to predict observations that had not been encountered during the training, much similar to the human capacity to generalize. For instance, a human does not need to have seen each distinct tree in order to classify a certain object as a tree. Instead, the person is able to classify the object by matching it with certain known features of trees (e.g. structure, color, context) that he or she has learned from previous observations. Machine learning aims to replicate this capacity in a computational sense.

Several different machine learning techniques and algorithms have been proposed over the years. Among these the class of *Artificial Neural Networks* is probably the most well-known [45, 91]. This class of techniques is vastly inspired by the physiology of the brain. Since its introduction it has successfully been applied to a plethora of problems, such as control and classification systems. More recently, however, the class of *Kernel Machines* has received a large amount of attention from academics [102]. This class of algorithms, and the subtype of Support Vector Machines in particular, has been shown to outperform Artificial Neural Networks and other techniques in various aspects [99, 11, 16, 122].

In this chapter we will explain the class of Kernel Machines in detail, at the hand of Support Vector Machines and the closely related Least Squares variant. This cannot be done without a sound theoretical introduction into the field of machine learning, which will be presented in Section 2.1. In that section we will formalize the goals of (supervised) machine learning and introduce the related terminology. Thereafter, the Support Vector Machine algorithm will be laid out in Section 2.2. The common aspect of all Kernel Machines is the so-called kernel function, which improves the performance of these machines on non-linear problems. The kernel function will be dealt with in Section 2.3. Once the so-called kernel trick has been explained in detail, we will have a look into the Least Squares variant of Support Vector Machines in Section 2.4. An important phase in applying Kernel Machines to a problem is the proper selection of parameters. This problem will be covered in the final section of this chapter.

2.1 Theory on Machine Learning

In order to describe the theory behind machine learning it is useful to describe this process in mathematical terms. Suppose that we are investigating an actual model that takes a certain input, let us define this input with vectors $\mathbf{x} \in \mathcal{X} \subseteq \mathbb{R}^n$. The n elements of these input vectors are called the *features*. The model assigns a certain output value to these input vectors, which we define as $y \in \mathcal{Y}$. In case $\mathcal{Y}$ denotes a set of discrete classes, e.g. $\mathcal{Y} \subseteq \{-1, 1\}$, then the problem is considered a *classification* problem. On the other hand, if $\mathcal{Y} \subseteq \mathbb{R}$, then we are dealing with a *regression* problem. The input vectors and output values together form observations, or samples, which can be denoted with a tuple $(\mathbf{x}, y)$. We wish to construct a machine that will learn a corresponding function $f : \mathcal{X} \to \mathcal{Y}$ from a – finite – set of ℓ observations (i.e. $S = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_\ell, y_\ell)$). It is important to note that we do *not* want our machine to just learn the observations, otherwise a simple lookup table would suffice. Instead, it should learn the *actual model* underlying these observations, so that it will be able to make predictions for unknown input. This capacity to correctly evaluate unseen samples is called the *generalization* capacity.

We can describe the process of machine learning using three main components [114]:

1. A *generator* of random vectors $\mathbf{x} \in \mathcal{X}$. These samples are drawn independently from a fixed but *unknown* probability distribution function $P(\mathbf{x})$[1].

2. A *supervisor* that returns an output value $y \in \mathcal{Y}$ for any input vector $\mathbf{x}$. This is done according to a conditional probability function $P(y|\mathbf{x})$, which is unknown but fixed as well.

3. The *learning machine* that can implement a set of functions $f(\mathbf{x}, \boldsymbol{\alpha}) \to y$. Here α is a set of parameters for the machine. The function that is implemented by the machine should return a prediction y, given a input vector $\mathbf{x}$ and the parameters $\boldsymbol{\alpha}$. It is common to denote the output of the machine with $\hat{y}$, as to distinguish it from the *desired* output value y. Obviously, the goal is to learn the machine in such a way that the difference between y and $\hat{y}$ is minimal for any input vector $\mathbf{x}$.

Let us consider this process in a less formal way using a simple classification example. Assume we have a machine that needs to determine whether someone is ill, given certain features (e.g. body temperature). The generator determines the distribution of the input vectors and thus the distribution of persons that have certain features. Given the fact that most people are – hopefully – healthy, this means that the generator will be most likely to "generate" people with a body temperature around 37°C. The supervisor will produce a distribution for the output of the system, given an input vector. In other words, the supervisor will take the input vector and will determine the probability that this person is really ill. The learning machine receives a set of samples of body temperatures and has the task to classify these samples in the same way as the supervisor has done.

More formally, the generator and the supervisor determine the model that the learning machine has to learn. The learning machine receives a set of independent and identically distributed (*i.i.d.*) samples from the generator (i.e. $P(\mathbf{x})$), which are supplied with an output value by the supervisor (i.e. $P(y|\mathbf{x})$). This makes up the set of observations $S = (\mathbf{x}_1, y_i), \ldots, (\mathbf{x}_\ell, y_\ell)$, drawn according to $P(\mathbf{x}, y) = P(\mathbf{x})P(y|\mathbf{x})$. From this set of observations the learning machine has to induce a model (i.e. $f(\mathbf{x}, \boldsymbol{\alpha}) = \hat{y}$) with an error that is as small as possible.

[1]The requirement that the probability function must be fixed excludes some types of problem from this theory, such as certain forms of time-series prediction. Nevertheless, Kernel Machines have successfully been used for time-series prediction [77].

The expected value of loss between the response of the supervisor and the learning machine can be formalized by the so-called *risk functional* [114]

$$R(\boldsymbol{\alpha}) = \int_{\mathcal{X}\times\mathcal{Y}} L\left(y, f(\mathbf{x}, \boldsymbol{\alpha})\right) dP(\mathbf{x}, y) \quad , \tag{2.1}$$

where $L(y, f(\mathbf{x}, \boldsymbol{\alpha}))$ is a function that measures the loss for a prediction. For the classification case, one could consider a loss function that returns 1 for a misclassification and 0 otherwise. In case of regression a suitable loss function would be

$$L\left(y, f(\mathbf{x}, \boldsymbol{\alpha})\right) = \left(y - f(\mathbf{x}, \boldsymbol{\alpha})\right)^2 \quad , \tag{2.2}$$

which yields the least-squares optimization problem. Henceforth we will focalize on the regression case, although the theory is easily adapted to the classification case.

The ideal goal of a learning machine is to find a function $f(\mathbf{x}, \boldsymbol{\alpha})$ that minimizes the risk functional described in Eq. 2.1. However, it is impossible to optimize this risk functional, as the underlying model $P(\mathbf{x}, y)$ is unknown. The only information that we have about this model is the set of i.i.d. samples S. This information can be used to define an empirical risk function

$$R_{emp}(\boldsymbol{\alpha}) = \frac{1}{\ell} \sum_{i=1}^{\ell} L\left(y_i, f(\mathbf{x_i}, \boldsymbol{\alpha})\right) \quad . \tag{2.3}$$

If we insert the least squares loss function for the regression case (i.e. Eq. 2.2), we obtain

$$R_{emp}(\boldsymbol{\alpha}) = \frac{1}{\ell} \sum_{i=1}^{\ell} \left(y_i - f(\mathbf{x_i}, \boldsymbol{\alpha})\right)^2 \quad , \tag{2.4}$$

which yields the standard least squares error criterion. One could opt to try to minimize this empirical criterion directly, which is called *empirical risk minimization*. On the other hand, Vapnik proposes *structural risk minimization*, which aims to minimize an upper bound on the actual risk. If we choose some η, so that $0 \leq \eta \leq 1$, then with probability $1 - \eta$ we can state the following bound [114]:

$$R(\boldsymbol{\alpha}) = R_{emp}(\boldsymbol{\alpha}) + \sqrt{\left(\frac{h\left(\log\left(2\ell/h\right) + 1\right) - \log\left(\eta/4\right)}{\ell}\right)} \quad , \tag{2.5}$$

where h is a measure on the *capacity* of the machine (i.e. the *Vapnik Chervonenkis* (VC) dimension). The right part of the right term is denoted the *VC confidence* term. This VC dimension is determined by the class of functions and the parameters that are being used. In most techniques the capacity is controlled by the number of free parameters, such as the number of hidden nodes in Artificial Neural Networks. More theoretically, the measure is the largest number of arbitrary data points that can be *shattered* by the class of functions[2]. If the capacity of the machine is high, it will be able to shatter many data points and the empirical error can be reduced to a minimum. However, this will result in overfitting, as the machine starts to capture all details (e.g. noise) in the input data. On the other hand, a low capacity can prevent the machine from being able to represent the actual model, i.e. underfitting. In this situation the VC confidence term will be low, but the overall risk will be high, due to a high empirical error. Therefore, we need to minimize

[2]A function set can shatter data points if it is able to classify them in any arbitrary way.

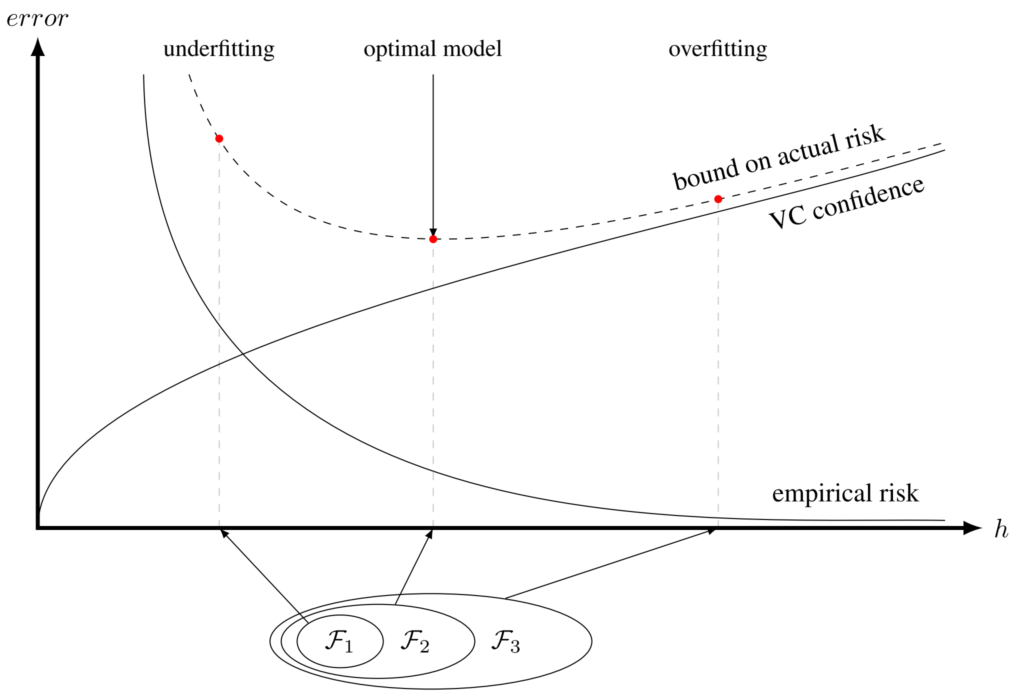


Figure 2.1: A demonstration of structural risk minimization [114]. The optimal model lies at the point where the combination of the capacity of the machine and the empirical error is at a minimum. The sets $\mathcal{F}_1 \subset \mathcal{F}_2 \subset \mathcal{F}_3$ represent three classes of functions. Their capacities h follow the order $h_1 < h_2 < h_3$.

the capacity, in order to prevent overfitting, while maintaining a low empirical error at the same time (i.e. preventing underfitting). This scenario, where there is a balance between the capacity of the machine and the empirical error, is depicted in Fig. 2.1. Interestingly, we can also observe that the limit of the VC confidence term for $\ell \to \infty$ is 0. This confirms our intuition that for an infinite amount of observations the actual risk and the empirical risk are equal. The VC dimension is described in more detail by Vapnik or Burges [114, 10].

However, the VC dimension has two disadvantages that make it less suitable in practical situations. Firstly, it can be very difficult to calculate the VC dimension for certain classes of functions and parameters. Secondly, the VC dimension is a very loose upper bound on the capacity of the machine, which means that the actual risk may be much lower than this bound. Other measures are usually preferred to perform structural risk minimization in a practical context. These measures include various types of cross validation, in which the risk is estimated by an empirical error on a separate validation set. Cross validation will be covered in more detail in Section 8.2.1.

2.2 Support Vector Machines

A machine learning technique that is well suited for structural risk minimization is the Support Vector Machine (*SVM*) [114, 23]. The standard variant of SVM has been proposed for binary classification problems [10]. The idea behind this technique is to classify data by means of a *separating hyperplane* between the positive and negative examples. The samples $\mathbf{x}$ that lie on the hyperplane satisfy the condition $\langle \mathbf{x}, \mathbf{w} \rangle + b = 0$. The weight vector $\mathbf{w}$ is normal to the separating hyperplane and the minimal distance from

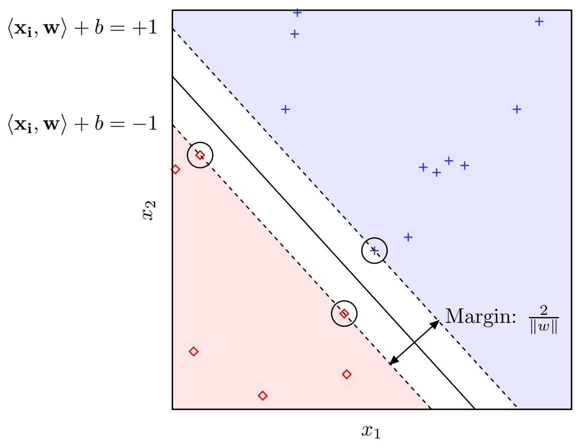

$\langle \mathbf{x_i}, \mathbf{w} \rangle + b = +1$
$\langle \mathbf{x_i}, \mathbf{w} \rangle + b = -1$
x_2
Margin: $\frac{2}{\|w\|}$
x_1

Figure 2.2: The linear hyperplane that separates positive and negative data points with the largest margin. This figure shows the three support vectors and the hyperplanes that they are "supporting".

the origin to the hyperplane is $\frac{|b|}{\|w\|}$. The hyperplane is thus completely described by means of the *weight vector* $\mathbf{w}$ and the *bias* b. Obviously, many different linear hyperplanes may exist that separate the positive from the negative data points.

The SVM algorithm revolves around finding the separating hyperplane that has the *largest margin*. In other words, the hyperplane must be as distant as possible from the data points that have a distinct output value (i.e. either -1 or $+1$). Let us define the distance between the separating hyperplane and the closest positive and negative sample as d_+ and d_-, respectively. The total margin between both classes can then be defined as the sum of both distances, i.e. $d_+ + d_-$. This can be formalized with the following constraints:

$$\langle \mathbf{x_i}, \mathbf{w} \rangle + b \geq +1 \quad \text{for } y_i = +1 \tag{2.6}$$

$$\langle \mathbf{x_i}, \mathbf{w} \rangle + b \leq -1 \quad \text{for } y_i = -1 \tag{2.7}$$

$$y_i \left(\langle \mathbf{x_i}, \mathbf{w} \rangle + b \right) - 1 \geq 0 \quad \text{for } \forall_i \ , \tag{2.8}$$

where the latter is a combination of the first two inequalities[3]. The two constraints in Eq. 2.6 and Eq. 2.7 define two hyperplanes, which can be seen in Fig. 2.2. The distance from the origin to these two parallel hyperplanes is $\frac{|1-b|}{\|w\|}$ and $\frac{|-1-b|}{\|w\|}$, respectively. From this we can conclude that $d_+ = d_- = \frac{1}{\|\mathbf{w}\|}$ and that the margin between the two hyperplanes is thus $\frac{2}{\|\mathbf{w}\|}$. Obviously, the separating hyperplane with the largest margin lies centered between these two hyperplanes. This placement guarantees a maximal distance to positive data points on the one side and negative data points on the other side. The maximum margin hyperplane can thus be found by *minimizing* the norm $\|\mathbf{w}\|$, subject to the constraints in Eq. 2.8.

In Fig. 2.2 we can observe that there are three data points that lie on either of the two hyperplanes. For these vectors the equality $y_i \left(\langle \mathbf{x_i}, \mathbf{w} \rangle + b \right) - 1 = 0$ holds. These vectors are called the *support vectors*, since they "support" the separating hyperplane. In case that one or more of these support vectors would be removed from the data set, then the found solution would change. After the hyperplane with the largest margin has been determined, the weight and bias can be used to predict output values for given input vectors. These predictions can be made using the equation

[3] In this reformulation we make use of the fact that for binary classification problems the output y can take either the value $+1$ or -1.

$$f(\mathbf{x}) = \operatorname{sgn}\left(\langle \mathbf{x}, \mathbf{w} \rangle + b\right) \ . \tag{2.9}$$

The optimization problem, including the constraints in Eq. 2.8, can be described using a Lagrange formulation. The advantage is that this enables the constraints to be directly formulated in the optimization problem. Furthermore, after this reformulation the training data will only appear as inner products between two vectors. We will explore the major advantage of this fact later in this chapter. The optimization problem can be reformulated by subtracting the inequality constraints, multiplied by a positive Lagrange multiplier, from the original optimization problem. These steps yield the following minimization problem[4]:

$$L_P \equiv \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{\ell} \alpha_i y_i \left(\langle \mathbf{x}_i, \mathbf{w} \rangle + b\right) + \sum_{i=1}^{\ell} \alpha_i \qquad \text{for } \alpha_i \geq 0 \ \forall i \ . \tag{2.10}$$

This primal Lagrangian can be rewritten in a so-called dual formulation (i.e. the Wolfe dual [34]). This gives us the following *maximization* problem:

$$L_D \equiv \sum_{i=1}^{\ell} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{\ell} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \qquad \text{for } \alpha_i \geq 0 \ \forall i, \sum_{i,j=1}^{\ell} \alpha_i y_i = 0 \ . \tag{2.11}$$

As we explained before, this formulation contains the input vectors only in the inner product with each other. Although the primal and dual formulations have different constraints, they result in the same unique solution. In this solution the input vectors for which the Lagrange multiplier is non-zero, i.e. $\alpha_i > 0$, will be the support vectors. The other data points, which will have $\alpha_i = 0$, are not strictly necessary for the solution. This fact is also represented in the definitions of the weight vector and the prediction function, i.e.

$$\mathbf{w} = \sum_{i=1}^{\ell} \alpha_i y_i \mathbf{x}_i \tag{2.12}$$

$$f(\mathbf{x}) = \sum_{i=1}^{\ell} \alpha_i y_i \langle \mathbf{x}_i, \mathbf{x} \rangle + b \ . \tag{2.13}$$

We can observe that only the support vectors contribute to the result of these equations, since only for these vectors α_i is non-zero. As a result, the solution is generally *sparse*, as only a fraction of all training samples will become support vectors. Therefore, it suffices to take the summation in Eq. 2.13 only over the set of support vectors.

2.2.1 The Non-Separable Case

For sake of simplicity we have assumed so far that the samples are indeed linearly separable. In many real-life applications this will not be the case, e.g. due to noise present in measurements. As a result, the above algorithm will not find any solution in these situations. Therefore, it is necessary to relax the constraints in

[4]The factor $\frac{1}{2}$ is solely added for mathematical convenience and is not strictly necessary. However, we have opted to follow the common formulation found in related literature.

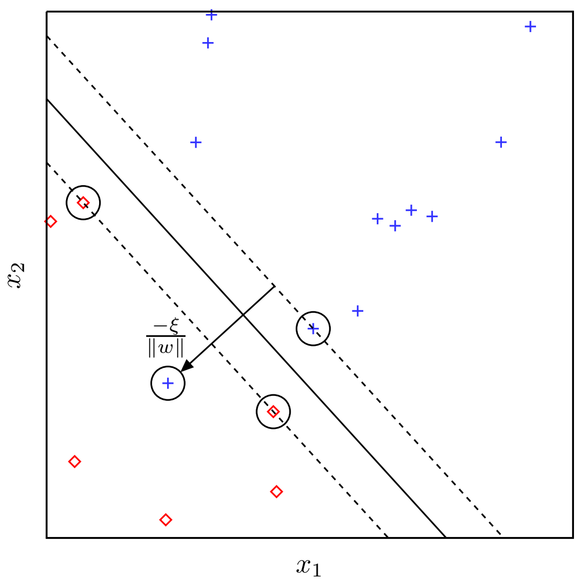


Figure 2.3: An example of data that is not linearly separable. The slack variable ξ measures the degree of misclassification, as demonstrated in the figure.

Eq. 2.6 and Eq. 2.7, as to allow for misclassifications. The amount of misclassifications should be regulated, so that they are only allowed when necessary.

The constraints can be relaxed using a so-called *soft-margin*. This is done by introducing positive slack variables $\xi_i,\ i = 1,\dots,\ell$ in the constraints. These variables measure the degree of misclassification of a certain sample, as is shown in Fig. 2.3. The constraints in Eq. 2.8 with the addition of slack variables becomes

$$y_i\left(\langle \mathbf{x_i}, \mathbf{w}\rangle + b\right) - 1 + \xi_i \geq 0 \qquad \text{where } \xi_i \geq 0\ \forall_i \ . \tag{2.14}$$

The use of the slack variables should be kept at a minimum, which implies that there should be a function that penalizes non-zero ξ_i[5]. This can be done by adding the sum of all slack variables in the objective function, such that the original minimization problem transforms into

$$\text{minimize} \quad \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{\ell}\xi_i \tag{2.15}$$

$$\text{subject to} \begin{cases} y_i(\langle \mathbf{x}_i, \mathbf{w}\rangle + b) \geq 1 - \xi_i\ \forall i \\ \xi_i \geq 0\ \forall i \\ C > 0 \end{cases} .$$

Here we have introduced a new parameter C, which is the regularization – or tradeoff – parameter. This parameter needs to be chosen by the user. A larger C corresponds to assigning a higher penalty to the

[5]Note that a misclassification occurs if and only if $\xi_i > 1$. Therefore, $\sum_{i=1}^{\ell}\xi_i$ is an upper bound on the number of actual training errors.

errors, which will result in overfitting to the data for very high values of C. A lower value for C, on the other hand, corresponds to emphasizing on the minimization of $||w||^2$, i.e. maximizing the margin. This will cause the machine to underfit for very low values of C. The attentive reader may notice the similarity with the situation in structural risk minimization. Indeed, the regularization parameter C allows for controlling the tradeoff between the capacity of the SVM on the one hand and the empirical error on the other hand. Structural risk minimization in the perspective of SVM thus corresponds to finding an optimal value for C [89]. Unfortunately, there is no analytical way to determine what a “high” or a “low” value is for this parameter, as it depends on the training data. It is therefore necessary to find good values for C empirically[6]. We will come back to this issue in Section 2.5.

Perhaps the most interesting aspect is that the slack variables vanish in the Lagrange dual formulation. Instead, the tradeoff parameter C now appears as an upper bound on the Lagrange multipliers. The modified Wolfe dual problem, with the additional constraints on the multipliers, becomes

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{\ell} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{\ell} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \\ \text{subject to} \quad & \begin{cases} 0 \le \alpha_i \le C \; \forall i \\ \sum_{i,j=1}^{\ell} \alpha_i y_i = 0 \end{cases} . \end{aligned} \tag{2.16}$$

This problem is a so-called *quadratic programming* optimization problem. It is possible to modify the primal Lagrange formulation as well, in order to take the slack variables into account. However, that formulation is considerably more complex, since new Lagrange multipliers need to be introduced for the slack variables. Therefore, the optimization problem in SVM is usually solved using the Wolfe dual.

2.2.2 Support Vector Machine for Regression Problems

Thus far we have concentrated on the SVM algorithm for classification problems. Since its introduction the algorithm has been extended for regression problems [29, 106]. This modified algorithm is referred to as *Support Vector Regression* (*SVR*), as opposed to *Support Vector Classification* (*SVC*). Henceforth, the set of both algorithms will be denoted with the global term SVM and the more specific SVC and SVR will be used when applicable.

Recall from Section 2.1 that in regression problems $\mathcal{Y} \subseteq \mathbb{R}$. This means that we wish to predict a real-valued output y with a certain accuracy. In ϵ-SVR the goal is to find a function $f(\mathbf{x_i})$ that predicts the actual output values with an error smaller than or equal to ϵ, for all the training samples. This means that errors that are smaller than or equal to ϵ (i.e. $|f(\mathbf{x_i}) - y_i| \le \epsilon$) are ignored. Furthermore, also for regression it is desirable to obtain the hyperplane with the largest margin. Just as for the classification situation, we wish to apply the soft-margin loss function, as to allow for some errors larger than ϵ. Combining all these elements we can formulate the following optimization problem:

[6]A good value for C is one that does not overfit, nor underfit, to the training data and thus optimizes the generalization performance.

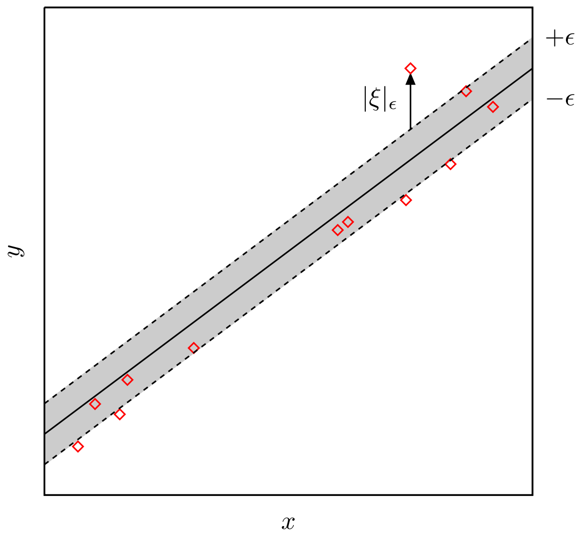


Figure 2.4: The ϵ-insensitive tube in the SVR algorithm with soft-margin loss function.

$$\text{minimize} \quad \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{\ell}(\xi_i + \xi_i^*) \tag{2.17}$$

$$\text{subject to} \begin{cases} y_i - \langle \mathbf{x}_i, \mathbf{w}\rangle - b \leq \epsilon + \xi_i \; \forall i \\ \langle \mathbf{x}_i, \mathbf{w}\rangle + b - y_i \leq \epsilon + \xi_i^* \; \forall i \\ \xi_i, \xi_i^* \geq 0 \; \forall i \\ C > 0 \end{cases} . \tag{2.18}$$

Note that for the regression case we cannot anymore combine the inequality constraints into a single constraint, thus two separate slack variables ξ_i and ξ_i^* are needed for each sample. The two slack variables are used to denote the deviation in either direction (i.e. “upward” or “downward”). The loss function has to be aware of the insensitivity to errors smaller than ϵ, which can be achieved by using an ϵ-insensitive tube. The exact definition of the loss function in case of an ϵ-insensitive tube is simply

$$|\xi|_\epsilon = \begin{cases} 0 & \text{if } |\xi| \leq \epsilon \\ |\xi| - \epsilon & \text{otherwise} \end{cases} . \tag{2.19}$$

The situation is depicted graphically in Fig. 2.4. Only the samples that lie outside the shaded area (i.e. the ϵ-insensitive tube) will contribute to the error penalty. For all samples inside the tube the slack variables $|\xi|_\epsilon$ will be zero.

The optimization problem in Eq. 2.17 can be reformulated using Lagrange multipliers. This step is nearly identical to the reformulation for SVC, which was described previously. A notable difference is that we need to split up the Lagrange multipliers α_i, which becomes α_i and α_i^*. This is because the soft-margin constraints could not be combined into a single constraint in the regression case, as we have noted before. The optimization problem in the dual Lagrange formulation becomes

$$\text{maximize} \begin{cases} \sum_{i=1}^{\ell} (\alpha_i - \alpha_i^*) \, y_i - \epsilon \sum_{i=1}^{\ell} (\alpha_i + \alpha_i^*) \\ -\frac{1}{2} \sum_{i,j=1}^{\ell} (\alpha_i - \alpha_i^*) \left(\alpha_j - \alpha_j^*\right) \langle \mathbf{x}_i, \mathbf{x}_j \rangle \end{cases} \tag{2.20}$$
$$\text{subject to} \begin{cases} 0 \leq \alpha_i, \alpha_i^* \leq C \, \forall i \\ \sum_{i=1}^{\ell} (\alpha_i - \alpha_i^*) = 0 \end{cases} .$$

The weight vector and prediction function can then be defined in terms of the Lagrange multipliers and the input data, analogous to Eq. 2.12 and Eq. 2.13. For the regression case this yields the following equations:

$$\mathbf{w} = \sum_{i=1}^{\ell} (\alpha_i - \alpha_i^*) \, y_i \mathbf{x}_i \tag{2.21}$$
$$f(\mathbf{x}) = \sum_{i=1}^{\ell} (\alpha_i - \alpha_i^*) \langle \mathbf{x}_i, \mathbf{x} \rangle + b \ . \tag{2.22}$$

2.3 The Kernel Trick

The SVM algorithm, as explained thus far, is not very useful in practical situations, since it is limited to linear problems. Classifying data points that lie within a circle, for instance, is not possible with this algorithm. This problem could be solved by preprocessing the training samples by a function $\phi : \mathcal{X} \rightarrow \mathcal{H}$, as to map them into some feature space $\mathcal{H}$ [1]. Consider the map $\phi : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ that is defined as $\phi(x_1, x_2) = \left(x_1^2, x_2^2, \sqrt{2}x_1 x_2\right)$. A linear SVM trained on the preprocessed features with this map would yield a quadratic function. In Fig. 2.5 we can see that, after preprocessing the samples with this map, the example circle problem becomes linearly separable in the feature space. This demonstrates how non-linear problems can be transformed into linear problems, on which the SVM algorithm can successfully be learned. A projection of the data samples of the circle example on the z-plane is depicted in Fig. 2.6. Note that this situation is completely identical to the linear problems that we have seen previously in Section 2.2. Furthermore, since the SVM algorithm operates after preprocessing the data, it will maximize the margin of the data in the feature space.

In our simplistic example above it is relatively easy to explicitly map all the samples into the feature space $\mathcal{H}$. Unfortunately, more complex mappings may be computationally infeasible, since they result in feature space of a very high dimensionality. This problem can be overcome using an observation that we made previously in Section 2.2. Recall that the SVM algorithm in its Wolfe dual formulation depends only on the inner products of the training samples. If we would map the data points using ϕ, then the Wolfe dual would simply depend on the inner products of the points in the feature space, i.e. $\langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle$. The key observation is that the algorithm depends only on the *inner products* of these points and *not* on the points themselves. Thus, if we define a function that directly calculates these inner products *in the feature space*, i.e.

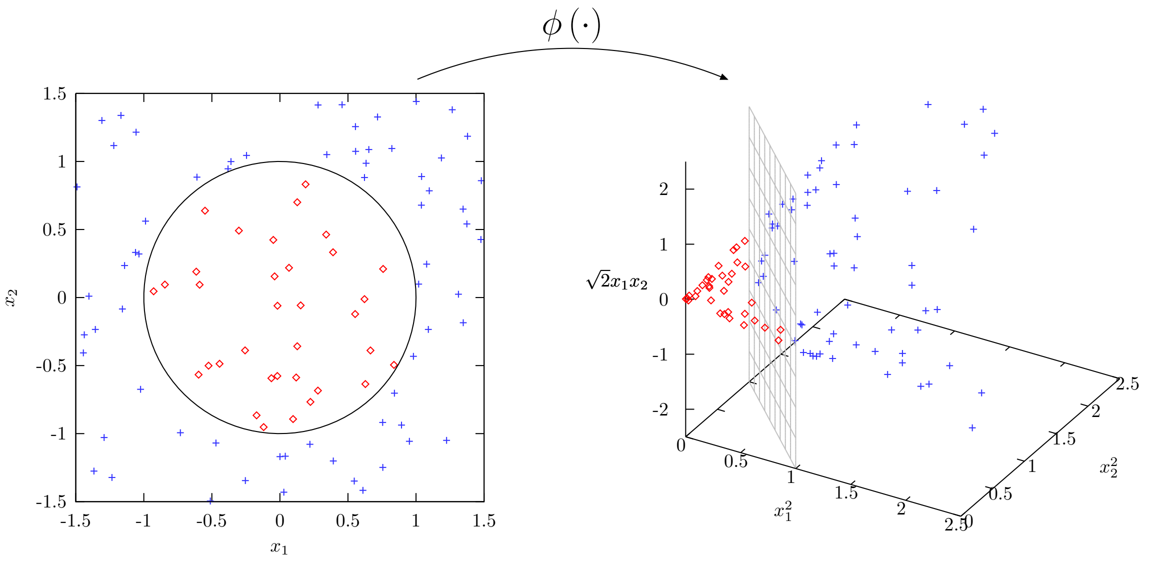


Figure 2.5: Demonstration of the mapping of a two dimensional classification set into a three dimensional feature space. After the mapping the data becomes linearly separable.

$$k(\mathbf{x},\mathbf{z}) = \langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle \quad , \tag{2.23}$$

then we would not need to explicitly map all the data points. Such a function that represents an inner product in a hypothetical feature space is named a *kernel function*. A kernel function that corresponds to our example mapping $\phi(x_1, x_2) = \left(x_1^2, x_2^2, \sqrt{2}x_1x_2\right)$ is given by $k(\mathbf{x},\mathbf{z}) = \langle \mathbf{x}, \mathbf{z} \rangle^2$. The SVM algorithm can make use of these kernels functions by simply substituting the inner product with a kernel function. Obviously, this works only in the Wolfe dual formulation, as in that formulation the input data only appears within inner products. The maximization problem in the dual formulation for the regression situation, i.e. Eq. 2.20, becomes after substitution

$$\begin{aligned} &\text{maximize} \begin{cases} \displaystyle\sum_{i=1}^{\ell}(\alpha_i - \alpha_i^*)\, y_i - \epsilon \sum_{i=1}^{\ell}(\alpha_i + \alpha_i^*) \\ \displaystyle -\frac{1}{2}\sum_{i,j=1}^{\ell}(\alpha_i - \alpha_i^*)\left(\alpha_j - \alpha_j^*\right) k(\mathbf{x}_i, \mathbf{x}_j) \end{cases} \\ &\text{subject to} \begin{cases} 0 \le \alpha_i, \alpha_i^* \le C \; \forall i \\ \displaystyle\sum_{i=1}^{\ell}(\alpha_i - \alpha_i^*) = 0 \end{cases} \quad . \end{aligned} \tag{2.24}$$

The concept of substituting inner products with kernel functions is known as the *kernel trick*. This technique has been applied to a wide variety of learning algorithms and is the “common divisor” of the class of Kernel Machines. It is important to understand that the mapping into the hypothetical feature space is *implicit*, when using the kernel function. Because of this implicit mapping it becomes feasible to use feature spaces of infinite dimensionality.

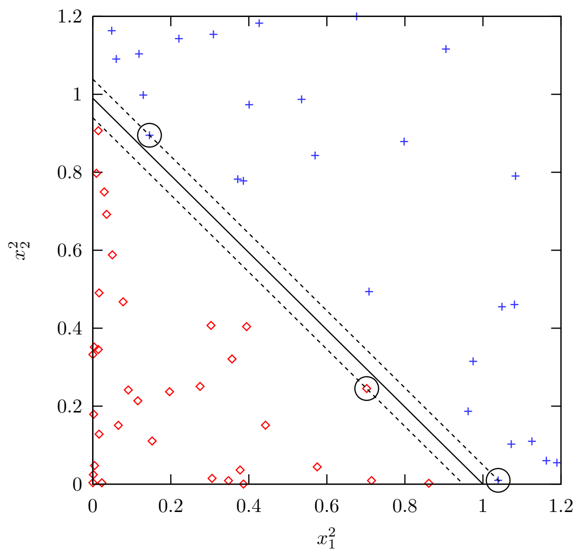


Figure 2.6: Projection on the z-plane of the data after the mapping into feature space.

In Eq. 2.24 we can observe that the kernel function needs to be computed for each possible combination of samples $\mathbf{x_i}$ and $\mathbf{x_j}$. Therefore, it common practice to construct a so-called *kernel matrix* containing all these results. The definition of the kernel matrix is

$$\mathbf{K} = \left(k\left(\mathbf{x}_i, \mathbf{x}_j\right)\right)_{i,j=1}^{\ell} \quad . \tag{2.25}$$

The kernel matrix is also referred to as the *Gram matrix*. In this report, however, we will consistently use the former term.

2.3.1 Conditions for Kernels

We have defined the kernel function as a function that represents an inner product in a certain hypothetical feature space $\mathcal{H}$, where $\mathcal{H}$ is – arguably – a Hilbert space. For this report, it suffices to mention that a Hilbert space is a generalization of Euclidean space. A Hilbert space requires that there is a notion of an inner product in the space. An interesting question that arises is, which kernel functions actually correspond to an inner product in some feature space $\mathcal{H}$. The answer is given by *Mercer's theorem*, which state that the function must be symmetric, continuous, and positive semi-definite [70, 114]. This can be formalized in the following condition (i.e. *Mercer's condition*):

$$\int_{\mathcal{X}\times\mathcal{X}} k\left(\mathbf{x}, \mathbf{z}\right) f\left(\mathbf{x}\right) f\left(\mathbf{z}\right) d\mathbf{x} d\mathbf{z} \geq 0 \qquad \text{for all } f \in L_2\left(\mathcal{X}\right) \quad . \tag{2.26}$$

Kernel functions that satisfy these conditions are referred to as *admissible* kernel functions. Following this condition on the kernel function, we can state the condition that the kernel matrix has to be positive semi-definite [10]. In a positive semi-definite matrix all the eigenvalues are greater than or equal to zero. Unfortunately, it may not be easy to check whether a kernel function satisfies Mercer's conditions, nor whether the kernel matrix is positive semi-definite. There are, however, certain functions that have analytically been proven to be admissible. The most common kernel functions – for classification and regression purposes – are given below.

$$\begin{aligned}
&\textit{Polynomial:} \quad k(\mathbf{x},\mathbf{z}) = \left(\langle \mathbf{x},\mathbf{z}\rangle + c\right)^d && \text{for } d \in \mathbb{N},\ c \geq 0 && (2.27)\\
&\textit{RBF:} \quad k(\mathbf{x},\mathbf{z}) = \exp\left(-\gamma \|\mathbf{x}-\mathbf{z}\|^2\right) && \text{for } \gamma > 0 && (2.28)\\
&\textit{Sigmoid:} \quad k(\mathbf{x},\mathbf{z}) = \tanh\left(\gamma \langle \mathbf{x},\mathbf{z}\rangle + c\right) && \text{for some } \gamma > 0, c \geq 0 && (2.29)
\end{aligned}$$

All these function are parameterized, as to allow for adjustments with respect to the training data. This introduces a problem identical to the one we encountered for the regularization parameter C, namely, which kernel parameters optimizes the generalization performance. Again, the optimal parameter setting depends on the actual training data and commonly needs to be determined empirically. The kernel parameters and the regularization parameter C together are denoted the *hyperparameters*.

Furthermore, we have to note that Mercer's condition is not strictly necessary to obtain good results with SVM. The Sigmoid kernel, for instance, does not satisfy Mercer's conditions [86]. Nonetheless, it has been used successfully in practical settings [98]. Furthermore, for a given training set the kernel matrix can be positive semi-definite, despite the kernel function not being admissible.

2.3.2 Combinations of Kernels

Verifying whether a kernel function satisfies Mercer's condition is not trivial, as we have stated before. However, this condition can be used to infer simple operations for the composition of kernel functions, which then also will be admissible [106]. For instance, the (weighted) linear combination of two admissible kernel functions is also admissible. Assume that k_1 and k_2 are admissible kernel functions, then the following combinations will be admissible as well [102]:

$$\begin{aligned}
k(\mathbf{x},\mathbf{z}) &= c_1 k_1(\mathbf{x},\mathbf{z}) + c_2 k_2(\mathbf{x},\mathbf{z}) && \text{for } c_1, c_2 \geq 0 && (2.30)\\
k(\mathbf{x},\mathbf{z}) &= k_1(\mathbf{x},\mathbf{z})\, k_2(\mathbf{x},\mathbf{z}) && && (2.31)\\
k(\mathbf{x},\mathbf{z}) &= a k_2(\mathbf{x},\mathbf{z}) && \text{for } a \geq 0 && (2.32)
\end{aligned}$$

The interesting aspect is that these operations allow for the construction of kernel functions in a modular way. Let us denote the polynomial, RBF and Sigmoid kernel functions (i.e. Eq. 2.27, Eq. 2.28, and Eq. 2.29, respectively) as *atomic kernel functions*. These atomic functions can be used in an initial set of admissible kernel functions. The combination operators can be applied to the kernel functions in this set, after which the resulting admissible kernel function can be added to the set. This way the combination operators allow us to recursively construct increasingly more complex kernel functions. At the end of this chapter we will see why this modular construction of kernel functions can be interesting.

2.4 Least Squares Support Vector Machines

The *Least Squares Support Vector Machine* (*LS-SVM*) has been introduced – more or less – as a simplification of the SVM [112, 111]. The main change is that the first two inequality constraints in Eq. 2.18 have been substituted with equality constraints. In other words, the notions of the margin and the ϵ-insensitive tube have been substituted with a normal error term. The error for a data sample $\mathbf{x}_i$ is defined as

$$y_i - (\langle \mathbf{x_i}, \mathbf{w} \rangle + b) = \epsilon_i \qquad \text{for } \forall_i \ . \tag{2.33}$$

The goal is to minimize the squared errors, as could be expected from the name of this algorithm. This yields the following optimization problem:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + \frac{1}{2}C\sum_{i=1}^{\ell} \epsilon_i^2 \\ \text{subject to} \quad & y_i = \langle \mathbf{x}_i, \mathbf{w} \rangle + b + \epsilon_i \ \forall i \ . \end{aligned} \tag{2.34}$$

Note that there are only equality constraints in this equation and that the equation is identical for both classification and regression. The advantage of using equality constraints is that the problem is no longer a quadratic programming problem, but instead a linear optimization problem. As we will see later, this simplifies the way in which the solution for this optimization problem can be computed.

It is convenient to reformulate the optimization problem from Eq. 2.34 as a Lagrangian, for the same reasons that applied in case of SVM. We have to note that the Lagrange multipliers are not necessarily positive anymore, since equality constraints appear in this optimization problem. The unconstrained optimization problem that follows is

$$\frac{1}{2}\|\mathbf{w}\|^2 + \frac{1}{2}C\sum_{i=1}^{\ell} \epsilon_i^2 - \sum_{i=1}^{\ell} \alpha_i \left(\langle \mathbf{x}_i, \mathbf{w} \rangle + b + \epsilon_i - y_i \right) \qquad \text{where } \alpha_i \in \mathbb{R} \ \forall i \ . \tag{2.35}$$

The optimality conditions for this problem can be found by setting the derivates[7] to zero, i.e. the saddle-point situation. These conditions can be used to derive the following equality:

$$\sum_{j=1}^{\ell} \alpha_j \langle \mathbf{x}_j, \mathbf{x}_i \rangle + b + C^{-1}\alpha_i = y_i \qquad \text{for } \forall i \ . \tag{2.36}$$

This set of equalities is a system of linear equations, which can be used to find the dual model parameters α_i and b. Obviously, the inner product in this equation can be substituted with any kernel function. Noting that we described a kernel matrix as well, i.e. Eq. 2.25, we can reformulate the system of linear equations in matrix form, which yields

[7]This gives us four different conditions, as the derivates should be taken with respect to (1) the weight vector $\mathbf{w}$, (2) the bias b, (3) the errors ϵ_i, and (4) the Lagrange multipliers α_i.

$$\begin{bmatrix} \mathbf{K}+C^{-1}\mathbf{I} & \mathbf{1} \\ \mathbf{1}^T & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{\alpha} \\ b \end{bmatrix} = \begin{bmatrix} \mathbf{y} \\ 0 \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} \boldsymbol{\alpha} \\ b \end{bmatrix} = \begin{bmatrix} \mathbf{K}+C^{-1}\mathbf{I} & \mathbf{1} \\ \mathbf{1}^T & 0 \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{y} \\ 0 \end{bmatrix} \ . \tag{2.37}$$

In this equation $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_\ell)^T$ and $\mathbf{y} = (y_1, \ldots, y_\ell)^T$. Analogous to SVM, the prediction output of LS-SVM can be described with

$$f(\mathbf{x}) = \sum_{i=1}^{\ell} \alpha_i k(\mathbf{x}_i, \mathbf{x}) + b \ . \tag{2.38}$$

In Eq. 2.37 we can observe that the LS-SVM optimization problem can be solved by means of a matrix inversion. As a result, solving the LS-SVM optimization problem is temporally independent on the regularization parameter C. This is contrary to SVM, where the training time increases as the value for C increases. This advantage comes at a certain cost, as the model in LS-SVM is not sparse by nature of the algorithm. This is because the Lagrange multipliers are proportional to the error ϵ_i, which means that α_i will be zero if and only if the error is zero. One could argue that the name Least Squares Support Vector Machine is slightly misleading, since practically all input vectors will become support vectors.

2.4.1 Efficient LS-SVM using Cholesky Factorization

Solving the LS-SVM optimization problem consists of the inversion of a matrix. This step can be made more efficient by making advantage of *Cholesky factorization* [13]. Unfortunately, the matrix on the left hand side in Eq. 2.37 is not positive semi-definite and cannot be solved directly using Cholesky factorization. Nonetheless, the left hand side of the system of linear equations can be made positive semi-definite by means of a rearrangement of the matrix. Let us define $\mathbf{M} = \mathbf{K} + C^{-1}\mathbf{I}$, then Eq. 2.37 can be reformulated as

$$\begin{bmatrix} \mathbf{M} & \mathbf{0} \\ \mathbf{0}^T & \mathbf{1}^T\mathbf{M}^{-1}\mathbf{1} \end{bmatrix} \begin{bmatrix} \boldsymbol{\alpha} + \mathbf{M}^{-1}\mathbf{1}b \\ b \end{bmatrix} = \begin{bmatrix} \mathbf{y} \\ \mathbf{1}^T\mathbf{M}^{-1}\mathbf{y} \end{bmatrix} \ . \tag{2.39}$$

The revised system of linear equations can be solved by setting

$$\mathbf{M}\boldsymbol{\eta} = \mathbf{1} \quad \text{and} \quad \mathbf{M}\boldsymbol{\nu} = \mathbf{y} \ . \tag{2.40}$$

Hereafter, the model parameters $\boldsymbol{\alpha}$ and b of the LS-SVM can be derived using $\boldsymbol{\eta}$ and $\boldsymbol{\nu}$

$$b = \frac{\mathbf{1}^T\boldsymbol{\nu}}{\mathbf{1}^T\boldsymbol{\eta}} \quad \text{and} \quad \boldsymbol{\alpha} = \boldsymbol{\nu} - \boldsymbol{\eta}b \ . \tag{2.41}$$

The explanation of these rearrangement steps have been kept concise in this report. The reader is referred to the work of Cawley for a detailed overview [13].

2.4.2 Reduced LS-SVM

The full LS-SVM algorithm involves the inversion of a $\ell \times \ell$ matrix. One can imagine that the algorithm becomes computationally intractable for relatively large training sets, due to the complexity with respect to the number of samples of $\mathcal{O}(\ell^3)$ of the inversion step. Furthermore, it is required that the complete kernel matrix is present in memory, which means a space complexity of $\mathcal{O}(\ell^2)$. A logical solution to reduce the temporal and spatial requirements is to approximate the full LS-SVM algorithm.

Approximating the algorithm is usually done by sparsifying the training set by means of creating an approximate low-rank kernel matrix $\tilde{\mathbf{K}}$. An intuitive way to do this is to simply take a subset of m samples of the set S and train the machine on this subset. Nonetheless, it is not hard to imagine that this method discards useful information from the original learning set. A more elaborate low-rank approximation is to minimize the empirical risk over all samples, but to allow the Lagrange multiplier α_i to be non-zero only at a denoted subset of the points [95, 105]. More formally, let us define a subset $S_m \subseteq S$ of the training samples, with size $0 < m \leq \ell$. We can then express the following modified problem

$$\left(\mathbf{K}_{m\ell}\mathbf{K}_{\ell m} + C^{-1}\mathbf{K}_{mm}\right)\boldsymbol{\alpha} = \mathbf{K}_{m\ell}\mathbf{y} \quad \Rightarrow \quad \boldsymbol{\alpha} = \left(\mathbf{K}_{m\ell}\mathbf{K}_{\ell m} + C^{-1}\mathbf{K}_{mm}\right)^{-1}\mathbf{K}_{m\ell}\mathbf{y} \ . \tag{2.42}$$

By taking a subset of only m data points, the matrix inversion is reduced to a $m \times m$ matrix, as opposed to $\ell \times \ell$ in the original algorithm. Furthermore, this approximation has the advantage that all ℓ samples are used for training. The only limitation is that only m samples are used to describe the model. In Eq. 2.42 the bias term has been omitted for illustrative purposes. The variant of the reduced LS-SVM algorithm that includes the bias term is

$$\left(\begin{bmatrix}\mathbf{K}_{m\ell} \\ \mathbf{1}^T\end{bmatrix}\begin{bmatrix}\mathbf{K}_{\ell m} & \mathbf{1}\end{bmatrix} + C^{-1}\begin{bmatrix}\mathbf{K}_{mm} & \mathbf{0} \\ \mathbf{0}^T & 0\end{bmatrix}\right)\begin{bmatrix}\boldsymbol{\alpha} \\ b\end{bmatrix} = \begin{bmatrix}\mathbf{K}_{m\ell} \\ \mathbf{1}^T\end{bmatrix}\mathbf{y} \ . \tag{2.43}$$

An interesting question is how to select the data points that will be in the subset S_m. Several approaches to this problem have been proposed [95]. These approaches generally aim to iteratively reduce some measure of the difference between $\mathbf{K}$ and $\tilde{\mathbf{K}}$, e.g. the trace of $\mathbf{K} - \tilde{\mathbf{K}}$. This can be done by selecting those samples that are the most poorly represented by the current subset [33]. This procedure, however, is somewhat flawed for the RBF kernel, as it is prone to select the outliers in the training set. In fact, the RBF kernel would perform better if those points are added that are centers of clusters of data points. Furthermore, some of the selection procedures may be computationally so demanding, that the time needed for the selection procedure exceeds the time gained by using an approximation strategy. If this is the situation, then it should of course be preferred to simply use the standard "full" LS-SVM algorithm.

It has been shown that selecting the samples at random gives a good balance between the quality of the approximation and the temporal complexity of the selection procedure [95]. Obviously, the random selection of data points can be done in constant time. Furthermore, it seems intuitive that selecting the data points at random from the training set will yield a relatively good approximation of the probability distribution found of the original data set.

2.4.3 Fast Leave-One-Out Cross Validation

Another interesting aspect of LS-SVM is that with this algorithm it is possible to compute the leave-one-out cross validation, with only a negligible additional computational expense. In short, the leave-one-out cross validation error is the error that one obtains if the machine is trained on $\ell - 1$ samples and then tested on the single remaining sample. This procedure is then repeated for each training sample. The leave-one-out cross validation measure will be perused in Section 8.2.1. Obviously, calculating the leave-one-out error requires the machine to be trained ℓ times, which is usually infeasible.

However, with LS-SVM it is possible to analytically compute the leave-one-out cross validation error after just a single training phase on the entire data set [13]. The only requirement is that the inverse of the extended kernel matrix, i.e.

$$\begin{bmatrix} \mathbf{K} + C^{-1}\mathbf{I} & \mathbf{1} \\ \mathbf{1}^T & 0 \end{bmatrix} = \mathbf{D} \ , \tag{2.44}$$

is explicitly calculated during training. This is a reasonable requirement, as the inverse matrix can be obtained at a low expense, even in case the Cholesky factorization method is used. Let us define $\epsilon_i^{(-i)}$ as the error that is obtained by training the machine on all samples, except for x_i. Normally, this error would be obtained by training the machine on all samples, except for x_i, and then predicting the output value y_i. In the fast leave-one-out procedure this error can be obtained directly after training on the entire data set. The error $\epsilon_i^{(-i)}$ is then given by

$$\epsilon_i^{(-i)} = y_i - \hat{y}_i^{(-i)} = \frac{\alpha_i}{(\mathbf{D}^{-1})_{ii}} \ , \tag{2.45}$$

where $\hat{y}_i^{(-i)}$ is the predicted output for sample x_i, y_i is the actual output value of the i^{th} sample, and $\left(\mathbf{D}^{-1}\right)_{ii}$ is the i^{th} value on the diagonal of the inverse of matrix $\mathbf{D}$. There is a similar analytical computation for calculating the leave-one-out cross validation error for the reduced variant of LS-SVM [14].

2.5 Hyperparameter and Kernel Selection

In this chapter we have laid out the theory of Kernel Machines at the hand of SVM and LS-SVM. These algorithms have shown to perform excellent on a wide variety of pattern recognition problems. However, as we have noted, the performance of either of these machines depend to a great extent on the hyperparameters that are used. This means that one needs to optimize these parameters in order to obtain the maximum generalization performance of Kernel Machines.

The hyperparameters are commonly tuned using a grid search, which can be considered a “brute force” search in the parameter space. In grid search the machine needs to be trained for every combination of parameter values, given a range and interval. Needless to say, this method is computationally demanding and scales exponentially with the number of parameters. In Section 4.1 we will explore different approaches for the problem of hyperparameter selection.

Selecting optimal hyperparameters is not the only important decision when training a Kernel Machine. Also the kernel function that is being used has a major influence on the generalization performance that is obtained. Which kernel function will yield the best results depends on the problem at hand. The fact that there is no such thing as a universal kernel function that is suited for every domain is occasionally referred to as the “no free kernel” theorem [22]. This is an analogy of the well-known “no free lunch for search” theorem, which states that there is no search method that will outperform all others for each type of problem [120].

The parameterized kernel function implicitly maps the data into a hypothetical feature space $\mathcal{H}$. The linear separating algorithms SVM and LS-SVM operate in this hypothetical space and depend on the exact way in which the data points are organized in this space. Therefore, we need to select a kernel function that organizes the data points in such a way that the linear algorithms can successfully be applied. In other words, for a given problem we need to find the kernel function that matches the training data best. Unfortunately, there exist no structured approach in selecting a kernel function. It is common that the user simply experiments with a small set of kernel functions (e.g. the RBF and polynomial kernel functions) and selects the one that yields the best results. Perhaps even more frequently, the RBF kernel is chosen by default, leaving other possibly better kernel functions untried. Obviously, this strategy will not be likely to yield the optimal generalization performance.

Chapter 3 Evolutionary Computation

In 1859 Charles Darwin published his famous work "On the Origin of Species", laying out a theory in which nature advances using gradual evolution by means of natural selection. Ever since, this theory has had a major impact on the fields as biology, religion, sociology, and philosophy. What Darwin could not have expected, is that over a century later his work on the evolution of species has inspired even mathematicians and computer scientists. Several techniques for search, optimization, and machine learning have been developed over the years under the collective term *Evolutionary Computation* (*EC*) [117, 26]. These techniques share the common notion that a search process can advance to good solutions by means of the principle of "survival of the fittest".

Over the years the *Evolutionary Algorithms* (*EA*) have been shown to be a very powerful and generalized search and optimization technique. It has been applied to domains as diverse as protein folding, image processing, task scheduling, and quantum algorithm discovery. This marks one of the strengths of EC, namely, that the algorithms are very generalized. As a result, many different types optimization problems may be tackled relatively easily using one of the EC paradigms.

In this chapter we will explain EC at the hand of three well-known algorithms. Section 3.2 will be dedicated to Genetic Algorithms, which is arguably the most used variant of EC. Then we will continue with Evolution Strategies in Section 3.3, after which the slightly different Genetic Programming will be covered in Section 3.4. However, we will start off with a general overview of EC in Section 3.1, in which we will introduce the reader to the relevant terminology.

3.1 Overview and Terminology

The key principle in EC – and evolution theory in general – is that potential solutions are generated, evaluated, and reproduced in an iterative process. During the course of this process, the individuals are subject to certain forms of mutation and can reproduce with a probability proportional to their *fitness*. A selection procedure removes the individuals with a relatively low fitness from the population, so that the more fit ones

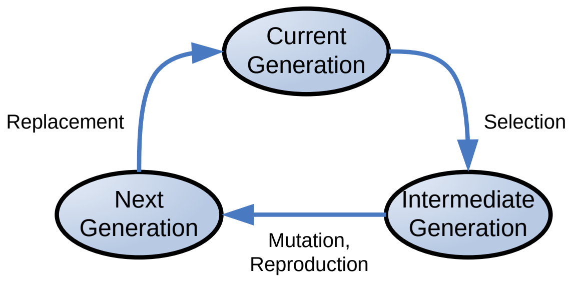


Figure 3.1: Overview of a basic evolution cycle within EC.

are more likely to "survive". This way the individuals become increasingly more fit as the process continues over more iterations.

A high level overview of an iteration cycle in this evolutionary process is shown in Fig. 3.1. The cycle starts with a *current generation*, which contains a certain *population* of *individuals*. The condition that such a population exists can be ensured by providing a *bootstrap* routine, which will generate a random population from scratch. An individual in the EC sense is simply a possible solution to the problem under consideration. The routine proceeds by evaluating all individuals using an *objective function*, which assigns a numeric value to each individual. Obviously, the objective function is highly domain dependent. The outcome of the objective function can be used either directly or indirectly as the *fitness* score of the individual[1]. The cycle continues by selecting a new intermediate "parent" population. Each individual is selected with a probability proportional to its fitness score. This intermediate population is used to create an offspring population, using various mutation and reproduction operators. The final step in the cycle is that the offspring population replaces the current generation. This replacement step could involve pruning back the population to a desired size. At this point the cycle has been completed and algorithm could perform another iteration. This process continues until a given termination condition has been met.

The interesting aspect in this implementation is the balance between two opposite operations. On the one hand, the selection operator aims to reduce the diversity of the population. The mutation and reproduction operators, on the other hand, try to increase the diversity. This is the key to the convergence of good solutions in evolutionary algorithms, as only those mutations that have improved the quality of the solutions persist.

So far a possible solution to a problem has been described as an individual. Individuals can be further specified into a *phenotype* and a *genotype*, analogue to the terminology in biology. The phenotype is the complete physical representation of an individual. In biological sense, this would encompass physical properties such as shape and size (i.e. morphology), behavior, physiology, and so forth. In the computational context of EC, the phenotype depends on the problem domain. As an example, for a parameter optimization problem the phenotype could be a tuple of real-valued parameters. The genotype, on the other hand, is the "blueprint" of the individual. In the case of a human being the genotype is the DNA that is inherited from the parents. In EC the genotype is a computational representation of the phenotype. For instance, we might prefer to encode the parameters in a real-valued optimization problem by means of bitstrings. These bitstrings are then the genotype. The distinction between genotype and phenotype may or may not be clear, depending on the problem.

[1]There is a slight distinction between an objective function and a fitness function. For instance, consider a minimization problem, where a lower value of the objective function means a higher fitness. The fitness function in this situation could assign a fitness value that is the inverse of the score of the objective function.

Strictly speaking, genotypes and phenotypes are type descriptors, whereas *genomes* and *phenomes* are instantiations of genotypes and phenotypes, respectively[2]. Furthermore, a genome consists of a set of *chromosomes*, which encode parts of the genetic material. Nonetheless, the terminology on genetics within the field of EC is commonly not as strict as within the field of biology. We have opted to follow this less strict use of the terminology. The only important distinction that we will use consistently throughout this report is between the genotype and the phenotype.

There are three important preparatory steps to be made when applying an EC algorithm to a problem. These three steps require the user to specify:

1. The *phenotype* and *genotype* representations of the problem.
2. An appropriate *objective function* for the individuals. This objective function determines the measure that is being optimized by the algorithm.
3. The *termination condition* for the evolutionary run. Possible termination conditions include a maximum number of generations or the stalling of the convergence rate.

All different variants of EC share the high level overview as we have just described. More precisely, however, the cycles may proceed in two different manners. The overview as explained so far iterates on the level of generations of populations, i.e. one generation is substituted with another generation. This form is named *generational* EC. Opposed to this is *steady-state* EC, in which the cycle iterates on the level of individuals. This means that every iteration a newly individual replaces an existing one. The generations in steady-state EC are therefore said to overlap each other. Throughout this chapter we will focus on the more standard *generational* EC. The explanations, however, can easily be adapted to the steady-state variant.

As we will see later, the distinction between the different paradigms of EC lies mainly within two aspects. Firstly, different representations may be used for the individuals. Secondly, the algorithms may emphasize on different types of operators. For example, some algorithms may emphasize on mutation, whereas others mainly use recombination of a set of individuals. We wish to emphasize that the "no free lunch" theorem applies also on different variants of EC. Each of the described paradigms has its own strengths and weaknesses.

The class of EC algorithms is "generalized" in the sense that it does not use any domain knowledge. This can be considered both a strength as well as a weakness. On the one hand, this fact means that the algorithms can easily be applied to a wide variety of problem domains, such as numerical and combinatorial optimization problems. On the other hand, however, it also implies that the algorithms will not perform in optimal fashion for many problem domains. Domain specific knowledge and optimizations can help to improve the quality of the solutions and can increase the convergence rate. *Memetic Algorithms* try to exploit this domain specific knowledge by combining Evolutionary Algorithms with other local search methods [76, 61]. This is commonly done by allowing the individuals to perform a local search before they are being evaluated. The analogy with biological evolution is that the fitness of an individual is by no means determined entirely by their genes. Instead, individuals can increase their fitness during a lifetime, for instance by means of exercises or specific nutrition. Although Memetic Algorithms are relatively new as compared to EC in general, they have been shown to outperform EC on specific problems.

[2]This distinction is similar to the distinction between a class and an object in Object-Oriented programming.

Algorithm 3.1 Pseudocode for the standard Genetic Algorithm (GA).

```
Require: $s > 0$   // size s must be larger than 0
1: $P \Leftarrow$ initialize($s$)   // create initial random population
2: $P$.evaluate()
3: while isNotTerminated() do
4:     $O \Leftarrow P$.select()   // create offspring population
5:     $O$.crossover()   // perform sexual reproduction
6:     $O$.mutate()
7:     $O$.evaluate()
8:     $P \Leftarrow O$
9: end while
```

3.2 Genetic Algorithms

Probably the most recognized form of EC is the class of *Genetic Algorithms* (*GA*), which were popularized by Holland [47, 48]. Genetic Algorithms mainly operate in the realm of the genotype. The chromosomes are usually very simple data structures, such as bitstrings. The offspring is generated by means of simple binary operators on one or more individuals. From these operators, sexual reproduction has traditionally been the operator that is emphasized in GA. The pseudocode of the GA algorithm is shown in Algorithm 3.1. The pseudocode clearly shows the evolutionary architecture that we have explained above.

It has been stressed that GA is not strictly an optimization method. Instead, it should be considered a search method that finds competitive solutions for a given problem. This means that there is no guarantee whatsoever that the process will yield globally optimal solutions. We will describe the GA algorithm at the hand of its genotype representation, the mutation and reproduction operators, and the selection procedures that can be used.

3.2.1 Genotype Representation

As noted before, GA often uses very simple data structures for the genotype. The most common genotype representation is to simply use bitstrings for the chromosomes. This can best be explained at the hand of an example. Consider a real-valued optimization $f(x, y, z)$ problem that takes three parameters as its input. An obvious phenotype representation for this specific problem would be a vector with three elements, i.e. (x, y, z). The corresponding genotype would represent the three parameters as bitstrings of a specific length. Suppose that we have the specific phenome $(16.26, 6.8, 0.55)$, then this would yield the chromosome

$$\mathbf{c} = (\underbrace{01000001}_{x}, \underbrace{00011011}_{y}, \underbrace{00000010}_{z}) \ . \tag{3.1}$$

In this example the parameters have been represented with three binary fixed-point numbers. The total resolution of the numbers is 8 bits, of which 2 bits have been reserved for the fraction. As a result, rounding errors may occur, as is the case in this example. GA has typically been used with representations of relatively low precision, such as 10 bits per parameter. The resolution, of course, has an influence on the quality of the solutions and the execution time of the algorithm.

The encoding method of the genotype is another important aspect. The common practice is to use standard binary encoding or closely related variants, such as the fixed-point encoding used above. These encoding schemes, however, have certain disadvantages. The most apparent problem is that the two values with a

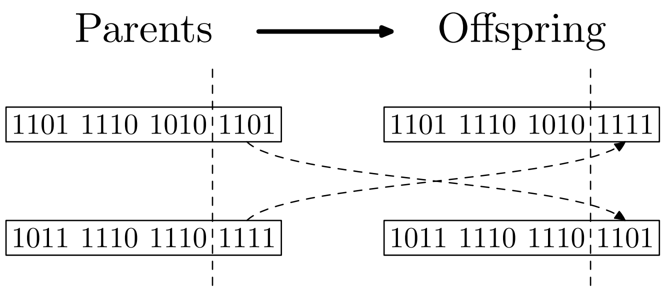


Figure 3.2: Demonstration of the crossover operator in GA applied to two chromosomes. The dashed lines mark the crossover point.

small Hamming distance [3] can represent vastly different values. Due to this fact, a minor mutation in the genome (e.g. a single bit) can mean a drastic mutation in the phenome of an individual. The so-called *Gray codes* solve this problem, as this type of encoding guarantees that adjacent integers in the original space are also adjacent in Hamming space, i.e. they have a Hamming distance equal to 1. A consequence of this connectivity of gray codes is that the number of optima in the Gray coded space must be less than or equal to the number of optima in the original real-valued space of the function. Standard binary encoding, on the other hand, may in fact introduce many new local optima. It has been shown empirically that Gray codes outperform binary representation for most – but not all – types of problems [117]. Besides gray codes, also real-valued presentations have occasionally been used in GA.

3.2.2 Reproduction and Mutation Operators

In Algorithm 3.1 we can observe that two operators are being applied to the population. Recombination is the more emphasized operator in GA. In recombination two – or more – individuals are combined to produce offspring that inherits genetic material from the parents. The idea is that partial solutions to the problem can be exchanged by means of this genetic material. This is to a great extent inspired by the sexual reproduction of human beings. The common way to implement this form of reproduction in GA is by means of *crossover*. In crossover two parents swap a fragment of their chromosome, which results in two offspring. An example of 1-point crossover is depicted in Fig. 3.2. Note that the crossover point is randomly chosen and may ignore the parameter boundaries. The probability with which crossover recombination occurs is usually denoted p_c. The parents that engage in recombination can be selected using various schemes, as we will see shortly. In general, however, individuals with a higher fitness will be more likely to be selected for reproduction. This way the beneficial genetic material of strong individuals is propagated throughout the population and the effect of malicious genetic material is dampened (i.e. the genetic repair effect).

After recombination the mutation operator can be applied to the population. This operator mutates every bit in the population with a low probability p_m. This probability is typically very low, i.e. less than 1%. Mutation aids the algorithm to create new genetic material within the population. A final remark is that not every individual will necessarily be subject to either operator. It is perfectly possible that an individual is copied unchanged into the offspring generation. Some algorithms may even copy the best m individuals directly into the offspring generation on purpose. This strategy is named *m-elitism* and it guarantees that the process cannot diverge into worse solutions.

[3] In information theory, the Hamming distance between two strings of equal length is the number of positions for which the corresponding symbols are different.

3.2.3 Selection Procedure

In addition to the mutation and recombination operators, the other key element in GA is the selection mechanism. In the canonical GA the selection algorithm is applied to the current population, in order to create an intermediate population. The selection procedure thus selects the individuals that will be subject to recombination and mutation. These individuals are selected with a probability that proportional to their fitness (i.e. *fitness proportional selection*). The fitness for individual i can, for example, be defined as $\frac{F_i}{\bar{F}}$, where F_i denotes the objective score of the individual and $\bar{F}$ denotes the average score of all individuals in the population. Individuals for which $\frac{F_i}{\bar{F}}$ is larger than 1 will thus have an above-average fitness. This fitness score can be used to determine the probability in which the individuals will be chosen for reproduction. One common way to do so is to use *roulette-wheel* selection, in which every individual has a space on the wheel that is proportional to its fitness. A common problem with this method is that the selection pressure is not constant over time. In the beginning, selection pressure may be to high, as the population consists of random generated individuals. In a later stage of the process, there tend to be less variation in the fitness of the individuals. The selection pressure then can becomes lower, which results in the search process to stagnate.

Another way to do fitness proportional selection is to uses a rank-based mechanism. The most simple scheme is to assign a fitness bias of $1 < Z \leq 2$ to the top ranked individual and $2 - Z$ to the bottom ranked individual. The other individuals are uniformly distributed between these two individuals, based on their rank. This means that the median ranked individuals will be designated a fitness bias of exactly 1. The advantage in this scheme is that the selection pressure is more constant throughout the search process. *Tournament Selection* is a fast – but noisy – way to implement ranking. In this selection mechanism the intermediate population is constructed by keeping tournaments between two individuals. The competitor with the highest fitness wins the tournament and is reproduced. On average, every individual competes in two tournaments each generation. The top ranked individual is expected to win both tournaments, whereas the bottom ranked individual will win none. The median ranked individual will, on average, win one tournament. In expectation, this produces a linear ranking of the individuals with a bias $Z = 2$. Tournament selection can also be used with competitions in which more than 2 individuals compete. The amount of competitors in tournament selection can be used to control the fitness bias and thus the selection pressure.

3.3 Evolution Strategies

While Holland was working on his GA, Rechenberg and Schwefel developed their *Evolution Strategies* (*ES*) [94, 4]. Interestingly enough, both techniques have emerged rather independently, although they were developed around the same time and both share their inspiration from biological evolution. ES have originally been developed for parameter optimization problems and has therefore taken a different direction from GA. There are two main differences between ES and GA. The first is that ES operates within the phenotype realm, where in GA the operations are done on a genotype representation. This distinction shows in the fact that ES are applied to real-valued representations of the problem. The second distinction between the algorithms is that mutation is the more emphasized operator in ES, in contrast to recombination in GA. There exist recombination mechanisms for ES, but the canonical ES relies on mutation for diversifying the genetic material. A practical distinction between GA and ES lies within the population sizes. ES are commonly applied on relatively small populations, say 1 to 20 individuals, whereas GA performs best on large populations of hundreds or even thousands of individuals.

Algorithm 3.2 Pseudocode for the $(\mu \overset{+}{,} \lambda)$ Evolution Strategy (ES).

```
Require: $\mu > 0$ and $\lambda > \mu$
 1: $P \Leftarrow$ initialize($\mu$)   // create initial random population
 2: $P$.evaluate()
 3: while isNotTerminated() do
 4:     $O \Leftarrow P$.reproduce($\lambda$)   // create offspring population
 5:     $O$.mutate()
 6:     $O$.evaluate()
 7:     if usePlusStrategy() then
 8:         $O \Leftarrow O \cup P$   // combine parent and offspring populations
 9:     end if
10:     $P \Leftarrow O$.select($\mu$)
11: end while
```

The pseudocode of the ES algorithm is shown in Algorithm 3.2. From this pseudocode we can observe the high resemblance of the GA and ES algorithms from a high abstraction level. The two basic types of ES are denoted with $(\mu + \lambda)$-ES and (μ, λ)-ES. The symbol μ refers to the size of the parent population, whereas λ denotes the size of the offspring population. The basic idea behind both variants of ES is that μ individuals are reproduced and mutated into λ offspring individuals. At the end of a generational cycle, the population is pruned back to the best μ individuals. The distinction between both variants lies within the fact that in (μ, λ)-ES only the newly generated offspring population is used to replace the parent population. This means that the lifetime of an individual is limited to a single generation. In contrast to this, in $(\mu + \lambda)$-ES the next generation is chosen from *both* the parent population and the offspring population (cf. lines 7–9 in the pseudocode). This makes it possible that individuals with a high fitness survive multiple generations. For example, in $(1 + 1)$-ES the offspring only replaces the parent if it has a higher fitness[4]. This could be considered as a form of hill-climbing search, which accepts a random change if and only if it is an improvement. It may be intuitive to assume that $(\mu + \lambda)$-ES should be preferred over (μ, λ)-ES, since it will not discard good solutions that have been found so far. Nonetheless, (μ, λ)-ES has empirically been shown to outperform $(\mu + \lambda)$-ES for certain problems [118]. This can be contributed to the higher selective pressure in (μ, λ)-ES.

3.3.1 Genotype Representation and Mutation

Previously, we noted that ES operates in the realm of the phenotype. This means that the operations are done directly on the real-valued parameters, instead of being encoded into a certain genotype representation. Alternatively, one could argue that in ES the genotype and phenotype are simply identical to each other. Recall the example of the real-valued parameter optimization from Section 3.2.1. The specific phenome $(16.26, 6.8, 0.55)$ for this problem can be represented as the – identical – ES chromosome

$$\mathbf{c} = (16.26, 6.8, 0.55) \quad . \tag{3.2}$$

This example makes clear that ES is particularly suited for real-valued optimization problems, for which they were originally developed. Unfortunately, combinatorial optimization problems may be considerably more hard to represent in ES.

[4]Furthermore, the more general $(\mu + 1)$-ES yields the steady-state variant of ES.

Since the representation is real-valued, the mutation operator can be much more simple than in GA. It is typically implemented as a distribution around the individual being mutated. The mutation is done by adding a random value, generated according to the distribution, to the parameter in the chromosome. Obviously, the distribution should have a mean of 0. The equation for a mutation of parameter x_i into a new value x_i' then becomes

$$x_i' = x_i + \mathcal{N}_i\left(0, \sigma_i\right) \quad , \tag{3.3}$$

where $\mathcal{N}\left(0, \sigma\right)$ denotes a logarithmic normal distribution with mean 0 and standard deviation σ. Other probability distribution functions may be used, if desired. Note that this mutation mechanism requires the user to specify a standard deviation σ_i for each parameter in the chromosome. These standard deviations are the *strategy parameters* of the algorithm, as are μ and λ. The value of σ determines the step size and is therefore very important for the convergence rate of the ES algorithm. In the initial phase of the evolution a relatively large step size should be used, which is gradually decreased as the process advances.

3.3.2 Self Adaptation

One interesting approach to adapt the step size during the course of the evolution is to encode the standard deviations into the chromosome of an individual. This means that these strategy parameters will be optimized by the evolutionary process itself. This mechanism is named *self adaptation* and is commonly applied in ES. An example of a chromosome with three parameters and the additional strategy parameters is

$$\mathbf{c} = \left(x_1, x_2, x_3, \sigma_1, \sigma_2, \sigma_3\right) \quad . \tag{3.4}$$

The standard deviations in this mechanism are usually referred to as the *endogenous strategy parameters* [5]. Opposed to this are the *exogenous strategy parameters*, such as μ and λ. We will refer to the original parameters x_i as the object parameters. Self adaptation has been shown to be an excellent mechanism of adapting the strategy parameters. Nonetheless, it leaves us with an important question that needs to be addressed, namely, how the endogenous strategy parameters themselves are updated. The following two update functions demonstrate how a chromosome $(\mathbf{x}, \boldsymbol{\sigma})$ is mutated into a new chromosome $(\mathbf{x}', \boldsymbol{\sigma}')$:

$$x_i' = x_i + \mathcal{N}_i\left(0, \sigma_i\right) \tag{3.5}$$

$$\sigma_i' = \sigma_i \exp\left(\tau' \mathcal{N}\left(0, 1\right) + \tau \mathcal{N}_i\left(0, 1\right)\right) \quad . \tag{3.6}$$

Note that $\mathcal{N}\left(0, 1\right)$ denotes a random value that is identical for each object parameter in the chromosome. On the other hand, $\mathcal{N}_i\left(0, 1\right)$ is a random value that is specific for each distinct object parameter. In Eq. 3.6 τ and τ' are the so-called learning parameters, which are constants that control the rate and precision of self adaptation. These learning parameters are not subject to the evolutionary process and are thus exogenous strategy parameters. Theoretical as well as empirical investigations suggest that these learning parameters should be chosen according to [100]

$$\tau \propto \frac{1}{\sqrt{2\sqrt{m}}} \tag{3.7}$$

$$\tau' \propto \frac{1}{\sqrt{2m}} \ , \tag{3.8}$$

where m denotes the number of object parameters.

The canonical ES algorithm, as described so far, uses mutations for the object parameters according to a spherical distribution parallel to the axes. The more advanced *covariance matrix adaptation* ES (*CMA-ES*) allows to arbitrarily change the orientation of the mutation by means of a correlation matrix [43, 44]. As a result, the mutations can adapt to the local shape of the fitness landscape. Although more powerful, this drastically increases the size of the chromosomes and the computational load. A full description of CMA-ES is beyond the scope of this report.

3.4 Genetic Programming

A more recent form of EC is the *Genetic Programming* (*GP*) paradigm, developed and popularized by Koza [59]. GP is very different from both GA and ES, as it should be considered rather a form of automated programming than a parameter optimization technique. It aims to solve a problem not by optimizing solely parameters, but by breeding a population of computer programs. These programs, when executed, are direct solutions to the problem. Obviously, this gives much more freedom in the structure of the solutions and therefore it can be applied to wider variety of problem domains. Thus far, GP has been successfully used to duplicate the functionality of various previously patented inventions [60]. These include general purpose controllers that outperform tuned PID controllers and the synthesis of various electronic components. This demonstrates the enormous problem solving potential of GP on non-trivial domains.

The base of the GP algorithm is identical to GA and therefore we refer the reader to Algorithm 3.1 for its pseudocode. The extension of GP lies within the representation of the individuals. Where GA uses fixed length strings to encode an individual, GP instead evolves runnable programs. This major extension of the paradigm requires a well-thought representation for the chromosomes. Furthermore, the mutation and reproduction operators need to be redefined to be used with new type of representation of the individuals. We will focus mainly on the differences between the GP and the GA paradigms.

3.4.1 Genotype Representation

The most common way to represent programs in GP is by means of *syntax trees*. These syntax trees are, regarding the structure, identical to how a compiler parses a high-level programming language. An example of a syntax tree that represents the mathematical function $(3/x) - (y * 5)$ is shown in Fig. 3.3. A syntax tree contains *nodes* and *links*. The nodes indicate the instructions to execute, whereas a link indicate the arguments for each node. The leave nodes, i.e. the nodes without any downward links, are named the *terminals*. The internal nodes, on the other hand, we will refer to as *non-terminals*[5]. These trees can be executed simply by means of a recursive depth-first traversal.

[5]Koza refers to the internal nodes as *functions*. In this report, however, we prefer to use *non-terminals* in order to prevent confusion with other types of functions (e.g. kernel functions). The term non-terminal is more common in the field of compiler theory.

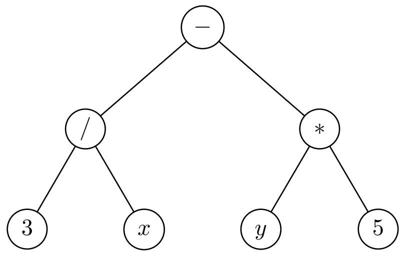


Figure 3.3: An example tree-like program representation as used in Genetic Programming.

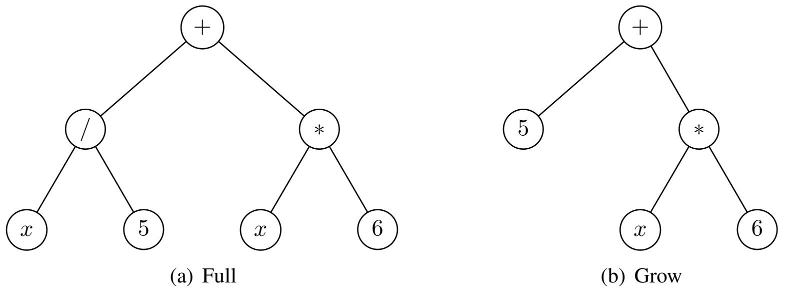


Figure 3.4: Constructing GP trees using (a) the *full method* and (b) the *grow method*.

Obviously, representing a genotype in this manner requires the user to specify which sets of terminals and non-terminals are allowed for a given problem. The set of terminals include the independent variables of the problem, zero argument functions, and constants. *Ephemeral constants* are a special type of constants, which are generated at random during the execution of the evolutionary process. The set of non-terminals (i.e. the *primitive functions*) includes all functions that take at least one argument. Many different types of non-terminals may be used, as this is entirely dependent on the problem domain. Examples include binary operations such as $\{+, -, \times, /, \max, \min, \dots\}$ or unary operations as $\{\sin, \exp, \log, \mathrm{sgn}, \dots\}$. Again, considerably more complex or specialized non-terminals may be necessary, depending on the problem domain under consideration.

Recall that in the primary phase of EC algorithms – and possibly also during later stages – individuals are randomly generated. In case of GA, where chromosomes are simply binary bitstrings, this random generation is of course trivial. Generating random syntax trees is relatively more complex. The individual trees in the initial population are typically recursively generated, using random choices of the non-terminals and terminals. This recursion is continued until a certain limit is reached, such as a maximum depth. In the *full initialization method* the nodes are taken from the set of non-terminals until the desired depth is reached. Beyond that depth only terminals can be chosen. The full initialization method will produce trees that are balanced, in the sense that the path from the root to a terminal is always of the specified depth (cf. Fig. 3.4(a)). The *grow initialization method* is a variant on the full initialization method. In grow initialization nodes are taken at random from both the non-terminals and terminals, until the depth limit is reached. Thereafter, it will behave as the full initialization method. The path from the root node to a terminal may be shorter than the actual depth limit, since terminals may be selected already before the depth limit is reached (cf. Fig. 3.4(b)).

As a final note, we wish to make clear that the GP paradigm is not restricted to tree-like structures, although this representation method is by far the most common. GP has been used with various other types of

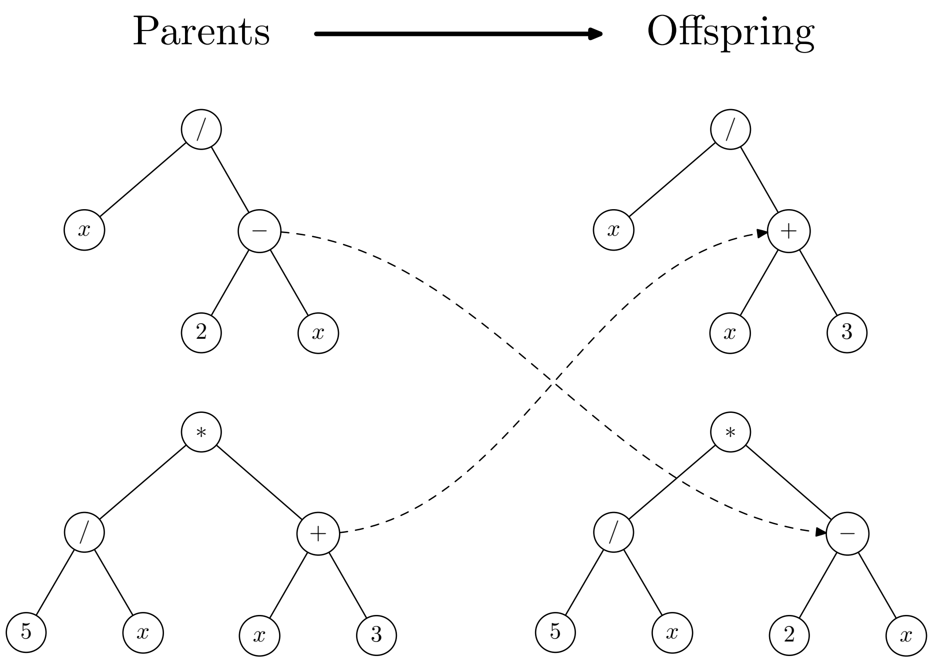


Figure 3.5: The crossover operator performed on two parents and the resulting offspring. Both parents exchange a sub-tree at a random crossover point.

representations, such as graph and linear representation schemes. If desired, several different representation schemes may be used concurrently by means of individuals with multiple chromosomes.

3.4.2 Reproduction and Mutation Operators

The operations in GP include sexual recombination and mutation operators that are much similar to their GA counterparts. In crossover recombination two parents exchange a part of their genetic material. Since the chromosomes in GP are trees, this means that they swap a sub-tree rooted at a random crossover point. Note that a crossover point needs to be selected for both trees separately and that these points do not necessarily need to coincide. The process of crossover recombination is illustrated in Fig. 3.5. The crossover point is often not selected with uniform probability. Instead, non-terminals are preferred over terminals as crossover points (e.g. in a $90\%/10\%$ ratio). The reason for this is that exchanging a terminal value is not very likely to make a large difference. Furthermore, due to the syntax tree structure approximately half of all the nodes will be terminals[6]. A bias toward non-terminals thus helps to increase the effect of crossover recombination.

Traditional mutation in GP consists of randomly selecting a mutation point in the syntax tree and replacing the sub-tree rooted at this point with a randomly generated tree. This process is shown in Fig. 3.6. The mutation point can be selected either at a terminal or a non-terminal. The randomly generated replacement tree can be constructed using either the full or grow initialization methods, as described above.

Another form of mutation in GP is to replace a sub-tree with one of its own sub-trees. This form is known as *shrink mutation*, as it shrinks the total size of the chromosome. An example of shrink mutation is shown in Fig. 3.7. First, a mutation point is randomly selected to mark a sub-tree of the chromosome, let us denote this sub-tree with T_1. Then a second mutation point is chosen somewhere in the marked sub-tree. This

[6]This statement holds only if the average number of branches of the non-terminals is at least 2.

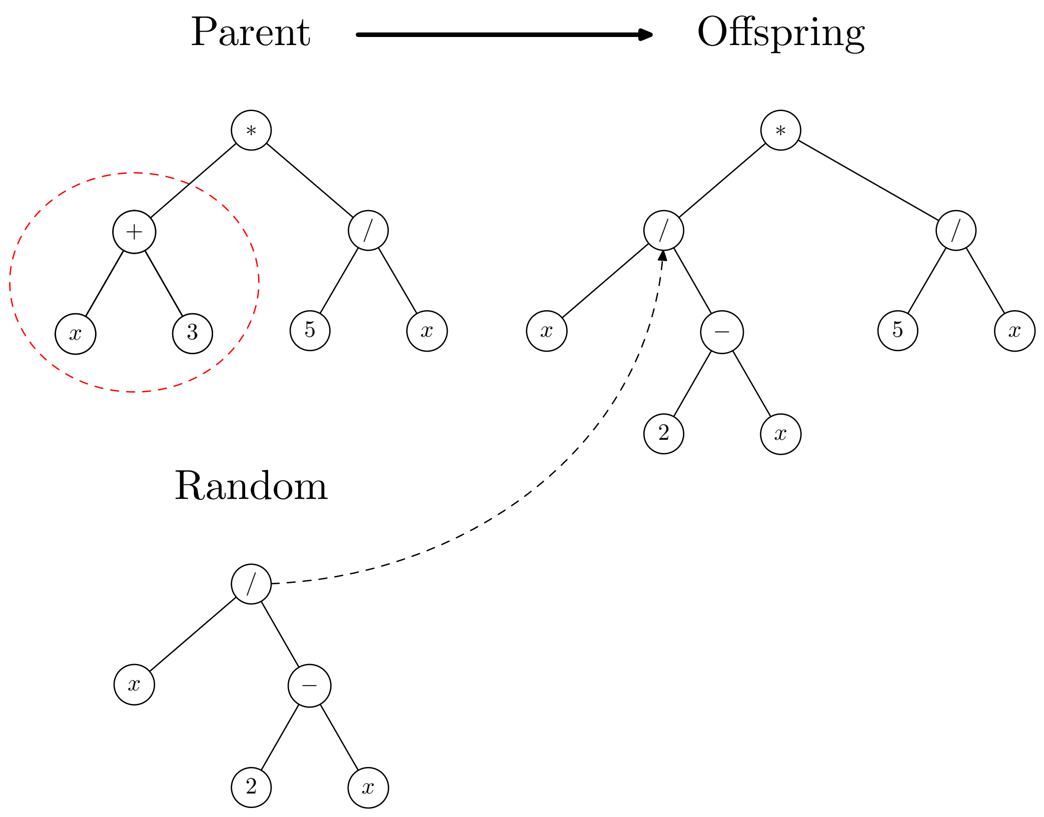


Figure 3.6: Example of sub-tree mutation in GP. The marked sub-tree in the parent is substituted with a randomly generated sub-tree.

second mutation point marks another sub-tree, which we will refer to as T_2. We can observe that both T_1 and T_2 are sub-trees of the same chromosome and that T_2 is a sub-tree of T_1. The mutation is completed by replacing T_1 with T_2 in the original chromosome. This means that the size of the chromosome is reduced, hence the name "shrink" mutation. Therefore, shrink mutation can be considered a rudimentary form of *parsimony pressure*. Parsimony pressure can be further enhanced by biasing toward smaller – and thus less complex – chromosomes in the fitness evaluation [39]. The goal of this is to reduce the complexity of the solutions, as to prevent too much "bloat" and overfitting of the solutions to the exact problem.

3.4.3 Constrained Syntactic Structures

In the canonical GP algorithm there is no way to restrict the structure of the syntax tree. For some problems it may be desirable to impose restrictions, so that non-terminals operate only on appropriate data types. Consider, for instance, a binary equality function, which takes two real values as its children and returns a boolean value. This boolean return value could of course not be used with standard arithmetic operators. On the other hand, the boolean return value could be useful in an if-then-else construction. An example of a syntax tree with typed non-terminals is shown in Fig. 3.8. In short, for some problems it is necessary to impose restrictions on the syntactic structure of the chromosomes.

Strongly Typed Genetic Programming has been proposed as an enhanced version of GP that enforces data type constraints [75]. This influences both the representation of the individuals and the chromosome altering

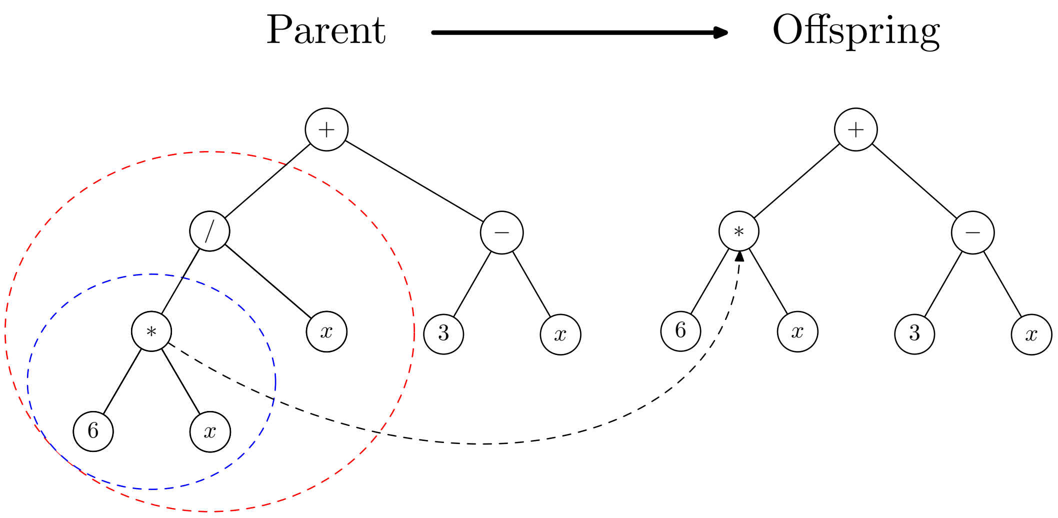


Figure 3.7: Example of shrink mutation in GP. The sub-tree marked with the red dashed line is substituted with one of its own sub-trees, marked with the blue dashed line.

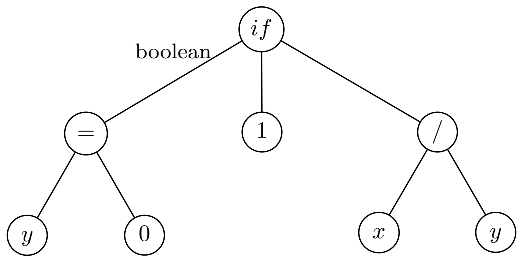


Figure 3.8: A syntax tree that models a protected division operator. This example shows the need of type constraints on the syntax tree. Note that all links are numerically typed, unless stated otherwise.

operators. Firstly, while defining the terminal and non-terminal sets the user also has to specify the types of the terminals and of the parameters. This typing information needs of course to be derived from the problem domain. Furthermore, the random generation must be altered to enforce type checking. We must guarantee that the trees that are randomly generated respect the typing constraints as imposed by the user. Secondly, the same holds for the mutation and recombination operators. Only those modifications are allowed to be made that do not break the validity of the syntactic structure.

Chapter 4

Related Work

Since their introduction, Kernel Machines have been applied successfully to a variety of problems. However, an important aspect of the practical application is the selection of both the hyperparameters and the kernel function. For certain values of the hyperparameters the machine will not be able to train itself on the problem and therefore performs poorly. The situation is similar for the kernel function: certain functions may perform better, or worse, for certain types of problems. Which are the optimal hyperparameters and kernel depends on the problem and the type of Kernel Machine that is used.

The common procedure of hyperparameter optimization is a grid search in the parameter space, as has been described in Section 2.5. This means that the machine is optimized by training it on a predefined range of parameter values. Two major drawbacks in this type search process are that it is extremely time consuming and that the method does not scale well with the number of parameters. Because of these drawbacks, various research has been conducted in finding betters methods to optimize hyperparameters.

Selecting a suitable kernel function is perhaps even more difficult than hyperparameter optimization. Here the common strategy is to just try the kernel machine on various kernels and see how it performs – if kernel selection is done at all. This is not very likely to change soon, as a relatively limited amount of research has been dedicated to the optimization of kernel selection.

In this chapter we will present the scientific advances made for the problems of hyperparameter and kernel selection. In this literature study there will be an emphasis on the application of EC for these problems. We start of with descriptions of both analytical and empirical studies on hyperparameter selection. A separate section has been dedicated to those solutions that make use of EC. Consequently, we will focus on the studies regarding combined kernel functions and lastly we will focus on the approaches that construct kernel functions using GP.

4.1 Hyperparameter Selection

Techniques for optimizing hyperparameters can be subdivided in two distinct groups, where one group comprises the *analytical* methods, whereas the other is based on empirical foundations. Within the group of analytical methods the most elaborate technique that has been used is that of *gradient descent* methods [21, 15, 56]. Gradient descent is an analytical minimization method, in which a local minimum is found by taking steps in the negative gradient direction. The approach is demonstrated on a non-spherical RBF function, which in this context means that features do not all share the same scaling factor, as would be the case for the standard RBF kernel in Eq. 2.28. A non-spherical variant of the RBF kernel that has as many scaling factors as there are features in the data set is given by

$$k(\mathbf{x}, \mathbf{z}) = \exp\left(-\sum_{t=1}^{n} \gamma^t \|x^t - z^t\|^2\right) \quad , \tag{4.1}$$

where γ^t defines the weight for the t^{th} feature. This demonstrates the excellent scaling properties of their approach, as there are more hyperparameters than there are features. Performing a grid search on such an optimization problem would be intractable, due to this large amount of hyperparameters. Furthermore, this non-spherical kernel can be used to perform feature selection, since the scaling factor of features that do not contribute to the solution will approach zero. Some error criteria are proposed for usage in their approach, of which the *radius-margin bound* is shown to perform well. This bound is given by

$$T = \frac{1}{\ell} R^2 \|\mathbf{w}\|^2 \quad . \tag{4.2}$$

Here R stands for the radius of the minimum sphere that contains all the samples in feature space[1]. The margin between the support vectors and the separating hyperplane, i.e. $\frac{2}{\|\mathbf{w}\|}$, we have seen before in Section 2.2. It is shown that the radius-margin is effectively an upper bound for the leave-one-out cross validation error for the classification case [115, 19]. This explains why this bound can be used to approximate the generalization performance of the classifier. Nonetheless, it has to be noted that more direct error criteria can be used, such as the leave-one-out cross validation error itself [57].

Although the gradient descent method is an elaborate and systematic technique, it does impose some strict requirements. The method assumes that the kernel function is continuously differentiable, since the hyperparameters are updated based on the gradient direction of a smoothed leave-one-out error bound. For certain kernel functions it may be impossible to satisfy this requirement, especially so for kernels that operate on non-vectorial data. Besides the kernel function, also the objective function that measures the performance of a set of hyperparameters is assumed to be differentiable. Moreover, the whole class of gradient descent methods has the apparent disadvantage that they can get stuck in local minima. Lastly, their approach is aimed at with SVM, which excludes the – direct – application of the method to other types of Kernel Machines.

An approach based on pattern search has been proposed to relax some of these assumptions [74]. In pattern search the neighborhood of a parameter vector is investigated in order to approximate a gradient [25]. This method thus removes the need for differentiability for determining a path based on the – approximated – gradient direction. The size of the neighborhood is gradually reduced at each iteration, which results in a smaller search steps. On the one hand, this approach solves a main problem in analytical gradient descent

[1]There is no need to explicitly map all samples into feature space to calculate the margin.

methods (i.e. the differentiability assumption). On the other hand, the empirical sampling of the neighborhood makes that the scaling performance, with regards to the number of hyperparameters, is drastically reduced.

Fröhlich and Zell have proposed a variant on the previous method [37]. Their method uses online Gaussian Processes to create a model of the error surface [24, 93]. Gaussian Processes are a machine learning technique not much different from LS-SVM[2]. An important difference is that Gaussian Processes not only give an expected output, but also an expected variance. Fröhlich and Zell make use of this property by sampling systematically at points where the expected improvement is highest. Therefore, the model of the error surface becomes increasingly reliable and the search is directed at interesting regions. One might ask what the gain is in using a learning method – Gaussian Processes – to model the error surface of another learning method – SVM. The answer is that training the Gaussian Process is relatively cheap as compared with training the SVM, since the dimensionality and sample size of the error surface model are generally much smaller.

A vastly different analytical approach is driven by heuristics based on information deduced from the data set. An example of such an approach is based on what is called *meta-learning*, or in other words, learning about learning [107]. The idea is to calculate meta-statistics of the problem data set (e.g. size, mean, variation, correlations among features) and to use this information to predict which hyperparameters would be optimal. This means that machine learning itself is applied on the problem of optimizing hyperparameters. Obviously, such a system first needs to be trained on a labeled training set. The main advantage of these heuristic approaches is that no machine needs to actually be trained. Especially for large data sets this enables significant savings in time when compared with methods that do need to train the Kernel Machine. However, the empirical evidence does not support the feasibility of this approach to a great extent. Its application seems therefore limited in situations where a relatively good set of initial parameters have to be approximated.

The original grid search method has slightly been adapted by Staelin. He proposes a method in which initially a sparse grid search is performed on the search space [108]. Then in the next iteration the resolution of the search is decreased (e.g. halved), after which a new grid search is centered around the currently best hyperparameters are. This means that the grid search is iteratively focused on the best solutions that are found so far, while the resolution is decreased. This method is shown to search more efficiently than the normal grid search. Nonetheless, it may be clear that this method is likely to find suboptimal solutions.

4.2 Evolutionary Hyperparameter Optimization

In Chapter 3 we learned that EC can well be applied to real-valued and combinatorial optimization problems. It is therefore not surprising that these algorithms have been applied to the hyperparameter optimization problem. One of the first mentions can be found in the work of Fröhlich et al. [36], in which GA is primarily used for feature selection. This is done by creating a *mask* for each feature in a binary genotype. The optimization of the regularization parameter C is done in parallel. Several objective functions have been used in their work, among which 4-fold cross validation and the radius-margin bound. The benchmark results are, however, difficult to interpret, since feature selection and optimization of C is done in parallel. Other similar work have focused on empirical error criteria [52, 73, 82], on solely hyperparameter optimization using the radius-margin bound [20] or k-fold cross validation [17, 96]. Furthermore, in some studies

[2]A detailed explanation of Gaussian Processes is outside the scope of this report.

the real-valued variant of GA is applied to the problem [51, 121]. Unfortunately, these studies do not give a clear idea whether the real-valued representation performs significantly better than a binary one. Practically all these studies have focused on the RBF kernel, which seem to have become the standard choice for kernel functions.

The literature in which ES is used for optimization instead of GA is rather sparse. To the best of our knowledge, only the approach proposed by Friedrichs and Igel make use of this method for hyperparameter optimization on a single kernel function [35]. They use ES to not only optimize the scaling, but also the orientation of the RBF kernel. This method is best explained using the appropriate formula. A more general variant of the standard RBF kernel is given by

$$k(\mathbf{x}, \mathbf{z}) = \exp\Big(-(\mathbf{x}-\mathbf{z})^T \mathbf{A} (\mathbf{x}-\mathbf{z})\Big) \quad , \tag{4.3}$$

where $\mathbf{A}$ is a symmetric positive definite matrix. In case that $\mathbf{A} = \gamma\mathbf{I}$, then this yields the standard RBF kernel function (i.e. Eq. 2.28) with one uniform scaling factor for all the features. In case that $\mathbf{A}$ can have independent scalings on the diagonal – and non diagonal values are all zero –, then this yields the non-spherical RBF kernel function, where each feature t is scaled by a corresponding scaling factor γ^t. This is the formula that we saw previously in the approach of Chapelle et al. (i.e. Eq. 4.1). Finally, if we allow arbitrary symmetric positive definite matrices for $\mathbf{A}$, then the input space can both be scaled and rotated. It is obvious that this approach gives an enormous freedom in the feature mapping of the RBF kernel. However, the price one has to pay is that the number of kernel parameters is quadratic with respect to the number of features, as opposed to linear and constant for the scaled and ordinary variants of the RBF kernel, respectively.

This ES approach achieved an improvement over the kernel parameters that were found using grid search. This result should be interpreted with care, as the optimal grid search parameter were used to initialize the evolutionary algorithm. This means that the ES approach was likely to perform *at least* as good as grid search[3], in terms of generalization capacity. The empirical objective function was based on classification errors on partial test sets.

In general the benchmark results in these studies confirms that EC can be applied rather well to the problem of hyperparameter optimization. The generalization performance is generally not much better than the one obtained using grid-search. However, these optimal solutions are usually found in much less iterations and the GA approach is thus less time consuming. An advantage of EC optimization when compared with gradient descent methods is that EC is more able to cope with local minima. Furthermore, it does not impose restrictions on the kernel and error criterion, such as differentiability.

4.3 Combined Kernel Functions

In Section 2.3.2 we demonstrated how single kernel functions can be combined into more complex ones. It seems intuitive that these more complex composite kernels could improve on the generalization performance, as the implicit feature mapping could be more suited for a specific problem. Several methods have been proposed for the composition of kernel functions, which is a form of kernel selection.

The first notions of optimizing a combined kernel were by Lanckriet et al. [62, 63]. In this work the linear

[3]Divergence of the search process could have occurred, since they made use of (μ, λ)-ES. Therefore, their approach could theoretically still have performed worse than grid search.

combinations of kernels are considered, i.e. a kernel matrix $\mathbf{K}$ is defined as

$$\mathbf{K} = \sum_{i=0}^{m} c_i \mathbf{K}_i \qquad \text{for } \mathbf{c} \geq 0 \ , \tag{4.4}$$

where the set $\mathcal{K} = \{\mathbf{K}_i, \ldots, \mathbf{K}_m\}$ is a predefined set of kernel matrices (e.g. initial random guesses) and the weight factors c_i need to be optimized. This optimization is demonstrated by means of Kernel Alignment or simply based on the margin, where a larger margin is considered better. Kernel Alignment, in short, is a quantitive measure that computes to which extent two kernel matrices are aligned with each other. Furthermore, the ideal kernel is assumed to be $\mathbf{y}\mathbf{y}^T$, i.e. the *target* matrix. The reasoning behind this specific ideal kernel is that if we would know a priori the target classification function $y(\mathbf{x})$, then the optimal kernel matrix would be $\mathbf{K}_{ij} = (y(\mathbf{x}_i) == y(\mathbf{x}_j))^4$, which yields the rank 1 matrix $\mathbf{K} = \mathbf{y}\mathbf{y}^T$. The general idea of Kernel Target Alignment is to use the alignment between a kernel matrix $\mathbf{K}$ and this target matrix as a performance measure for a kernel matrix. Kernel Alignment will be dealt with in more detail in Section 8.3.

The optimization of the weight factors $\mathbf{c}$ is done by using techniques from Semidefinite Programming [113]. Semidefinite Programming is an convex optimization that deals with convex functions over the convex cone of positive semi-definite matrices. This optimization method can be applied to this problem, given that the kernel matrices need to be semi-definite to satisfy Mercer's conditions. It is important to note that this optimization method works directly with kernel matrices, instead of kernel functions.

Similar approaches have been presented that use the same kind of linear combination of kernels and error criterion (i.e. Kernel Alignment). The methods differ in the way in which the weight factors are optimized. Ong and Smola use Semidefinite Programming on a different formalized problem, based on so-called hyperkernels [84, 83, 85]. Instead of using Semidefinite Programming, Kandola et al. resolve to the optimization problem by means of the Lagrange multiplier method [55]. This form of optimization is identical to how the SVM model selection problem is solved. Additionally, overfitting of the composite kernel is prevented by constraining the weight vector $||\mathbf{c}||$. Sun et al. optimize the weight factors using a generalized eigenvalue problem [109]. Although all these approaches make use of advanced mathematical foundations to tackle the problem, they fail to provide a considerable amount of benchmark results that support their claims. Furthermore, they rely on the assumption that Kernel Alignment indeed is a reliable predictor for the generalization performance of a Kernel Machine.

It has been argued that during the combination of kernels some potentially useful information is lost. Consider, for example, data sets that contain varying local distributions. Lee et al. proposed a method for combining kernels that does take this aspect into account [64]. Instead of combining various kernel matrices into one, their method creates a large kernel matrix that contains all original kernel matrices and all other possible mixtures of these matrices. Such a mixture is defined as

$$k_{i,j}(\mathbf{x}, \mathbf{z}) = \langle \phi_i(\mathbf{x}), \phi_j(\mathbf{z}) \rangle \ , \tag{4.5}$$

where $1 \leq i, j \leq m$ are indices in the set of kernel functions $\mathcal{K}$, m is the size of $\mathcal{K}$ and ϕ_i is the feature mapping corresponding to the kernel function k_i. These mappings cannot explicitly be calculated, since the RBF kernel implies a mapping into a feature space of infinite dimensionality. Therefore, this inner product

[4]In this equation $a == b$ returns $+1$ if a is equal to b and -1 otherwise.

cannot be calculated in the normal way. This problem is resolved by analytically deriving a function for the mixture of two RBF kernels, so that the kernel trick can still be applied. As a result of this strategy of mixtures, the combined kernel matrix grows m times in both dimensions. The support vector algorithm is now applied on this combined kernel matrix, by substituting the range of the summations from $[1, n]$ (i.e. the original SVM) to $[1, n \times m]$. One main advantage of this scheme is that it does not require any weight factors to be calculated.

Although their method is shown to outperform other combination methods for problems with local distribution, there are several disadvantages. Firstly, it focuses solely on the *combination* of a set of predefined kernel functions. This leaves us with the problem of hyperparameter selection for the set of predefined kernels. Furthermore, the time consumption of the learning algorithm will be increased with $\mathcal{O}\left(m^3\right)$ and the memory consumption with $\mathcal{O}\left(m^2\right)$. This is because the number of samples, from the perspective of the SVM, will increase with a factor m. This limits the application of this methods to relatively small problems. Probably the most apparent limitation in their approach is the dependency on a mixture function for the kernel functions. They managed to derive such a mixture for combining two RBF kernels, but it might prove difficult, if not impossible, to replicate this trick for various other kinds of kernel functions.

4.3.1 Evolutionary Combined Kernel Functions

Various EC inspired approaches have been proposed to combine kernel functions. One of these approaches optimize a linear combination of weighted RBF kernels using ES [88]. This means that for every kernel i in the combination there are two parameters to be optimized, namely the scale γ_i and the corresponding weight c_i within the combination. The used fitness function is based on 5-fold cross validation and is thus computationally intensive. Nonetheless, the results show that with an increased number of kernels in the combination higher generalization performance can be achieved.

This work is very similar to that of Dioşan et al., who proposed a linear combination of three *different* types of kernel functions [28]. She uses the linear, inhomogeneous polynomial, and RBF kernel functions, which all have predefined parameters. This means that only the weights are optimized using GA, which is claimed to be for comparison purposes. Therefore, the approach focuses strictly on the problem of combining kernels functions and not the hyperparameter optimization problem. The test set accuracy on benchmark problems show significant improvements of the combined kernel, when compared to the performance of its components.

So far these approaches have all used exclusively the linear combination method. However, in Section 2.3.2 we have given also other methods for combining kernel functions, such as multiplication. Other EA inspired methods have been proposed that make use of both addition and multiplication in the process of combining kernel methods. Lessmann et al. have proposed such a model based on GA and a set of five kernel functions [65, 66]. The algorithm is used to define the structure of the kernel function (i.e. multiplication or addition between two adjacent components) and the hyperparameters. The formula for the combined function is defined as

$$k(\mathbf{x}, \mathbf{z}) = k_1(\mathbf{x}, \mathbf{z}) \otimes_1 k_2(\mathbf{x}, \mathbf{z}) \otimes_2 k_3(\mathbf{x}, \mathbf{z}) \otimes_3 k_4(\mathbf{x}, \mathbf{z}) \otimes_4 k_5(\mathbf{x}, \mathbf{z}) \quad , \tag{4.6}$$

where the set of kernel functions $\mathcal{K} = \{k_1, \ldots, k_5\}$ is predefined and $\otimes_i \in \{+, \times\}\ \forall i$. The genotype in the GA approach contains the parameters for the five kernel functions and four extra bits that specify the type of the operators (i.e. either $+$ or $\times$). This approach is demonstrated to perform well on classification problems when the 10-fold cross validation is used as the error criterion. Several similar publications support this conclusion [6, 81, 80]. One limitation of these latter approaches, however, is that they are limited in the kernels that can be used. The set of kernel functions is either fixed to one type with various parameters, or to a fixed set of kernels with all distinct types.

These latter approaches suggest one important advantage of applying EC to this problem. The analytical methods, described in the beginning of this section, depended heavily on specially crafted mathematics to optimize the combination of kernels. In general, this decreases the flexibility of these approaches, as it would be difficult to extend the model to other combination methods (e.g. multiplication). Some others rely on specific aspects of the kernel function, which decreases the flexibility in using an array of different kernel functions. The evolutionary approaches do not know these difficulties, as they are generalized search methods. As a consequence, they are very flexible, as they can easily be adapted to handle very distinct approaches. Although we would not want to deny the importance of sound analytical grounds for a model, we would like to emphasize this flexibility offered by generalized search methods – like EC – for this type of problems.

4.4 Kernel Generation by means of Genetic Programming

In Section 3.4 we saw that GP can are well suited to search for mathematical functions, such as symbolic regression. It is therefore not surprisingly that people have tried to incorporate GP techniques to search for kernel functions. Howley and Madden were among the first to propose such a method, in the form of their *Genetic Kernel Support Vector Machine* [49, 50]. In this method a kernel function is evolved for use with an SVM classifier. The genotype used in their GP algorithm is tree structured, with the operators $+$, $-$, and $\times$ as the non-terminals. These operators come in two variants, namely a scalar and a vector version. To demonstrate the difference, we consider the $\times$ operator. The vector variant (i.e. $\times^v$) of this operator will return a vector, where each component is the product of two components of the input vectors. The scalar variant (i.e. $\times^s$) guarantees to return a scalar, so it will return the inner product of the two input vectors. The terminals in their approach are the two vectors $\mathbf{x}$ and $\mathbf{y}$. To guarantee that the kernel is symmetric, i.e. $k(\mathbf{x}, \mathbf{z}) = k(\mathbf{z}, \mathbf{x})$, the kernels are defined as $\langle e(\mathbf{x}, \mathbf{z}), e(\mathbf{z}, \mathbf{x})\rangle$, where e denotes the evaluation function of a tree. An example kernel function that they have found is shown in Fig. 4.1.

The attentive reader will notice that the kernel functions generated using this approach will *not* be guaranteed to satisfy Mercer's conditions. The argument is that it is time consuming to verify whether a generated kernel function satisfies Mercer's conditions and that these conditions are not strictly necessary. Experimentally they decided on a reasonable objective function, which is the classification error on the training set combined with a tiebreaker to limit overfitting. This tiebreaker is a close variant to the radius-margin bound that we described before. Despite the disadvantage that the constructed kernels only use simple arithmetic, the technique keeps up with or outperforms traditional kernels for most data sets. The authors emphasize, however, that a technique as GP requires the availability of a sufficiently large dataset.

Also Gagné et al. have recently used GP for kernel-based learning methods, which resulted in a system named the *Evolutionary Kernel Machine* [40]. This approach is similar to the Genetic Kernel SVM, as it also tries to evolve symmetric kernel functions using arithmetic operators. Nonetheless, the set of operators is extended with some operators, e.g. the exponential function, minimum, and maximum operators. A more notable difference is that this approach is applied to kernel k-nearest neighbors classification technique, instead of SVM. This choice has no influence on the generation of the kernel functions, but it allows for

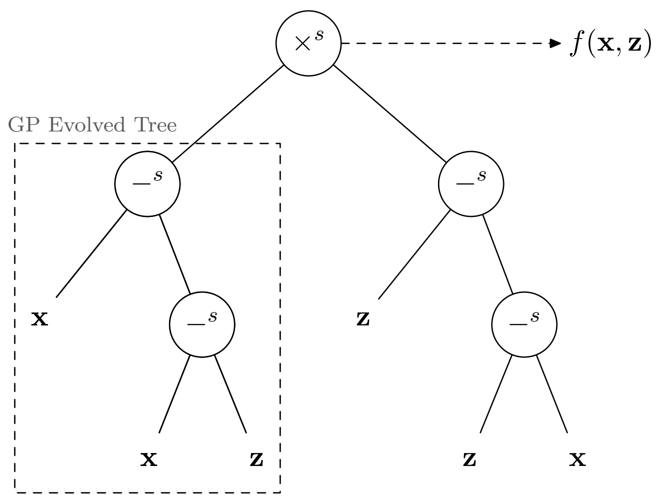


Figure 4.1: A kernel function as evolved by the Genetic Kernel Support Vector Machine approach [49, 50]. Note that the symmetry is guaranteed by taking the inner product of two identical trees, except that $\mathbf{x}$ and $\mathbf{z}$ are substituted.

a different objective function. The approach has been kept tractable by means of *co-evolution*. In short, a co-evolutionary framework evolves multiple – either cooperating or competing – species simultaneously. This technique has been used to determine the subsets that are used for both training and validation. The species that evolve the validation subset are competitive with the kernel function species. In other words, the system tries to find difficult validation subsets for the kernel functions, as the validation species aims to decrease the fitness of the kernels. On the other hand, the training subset species cooperate with the kernel functions and try to increase their fitness. This approach creates an advantageous balance, in which the training subsets are chosen in such a way to maximize the fitness of the kernels, whereas the validation subset is selected with the goal to minimize their fitness. The benchmark results suggest that their approach is able to compete with conventionally tuned SVM and k-NN systems.

Some enhancements to the approach by Madden and Howley have also been proposed by Dioşan et al. Their method differs from the original approach by a richer operator set (e.g. various norms are included) and small changes to certain operators [27]. The objective function used is the classification error on a validation set. In the comparison with the original approach by Madden and Howley the method does not perform significantly better. Furthermore, their results indicate that – generally – the RBF kernel performs at the same level. It has to be noted, however, that a kernel function was evolved using their system on one data set. In the empirical validation *the same* kernel function was tested on all benchmark problems. Therefore, it is likely that this kernel function was adapted to the specific problem on which it had been trained. It would be interesting to see the comparison when a *new* kernel function would be trained for each distinct data set.

Part II

Model and Implementation

Chapter 5

OPTIMIZATION MODEL

The work presented in Chapter 4 show that Evolutionary Computation techniques can help to improve the performance of Kernel Machines. The work presented so far has emphasized on the problem of hyperparameter selection and creating simple ensembles of kernels. In this chapter we present two models that can be used to tackle the problems of hyperparameter and kernel selection. The first model uses evolutionary parameter optimization techniques in order to find optimal hyperparameters for a kernel machine. This model is relatively similar to some prior work that was presented in the previous chapter. The second model uses more advanced EC techniques to search for combinations of kernel functions, driven by the objective to increase the generalization performance of the Kernel Machine for a specific problem.

We believe that it is important to stress the different goals that these models try to achieve. The hyperparameter optimization model aims to use EC techniques to improve on the search speed of hyperparameter selection. As such, this model tries to answer the question whether EC can optimize the phase of parameter selection in terms of computational cost. Generally, a lower computational cost directly relates to having the results available in less time. The search space in which this optimization model operates is *exactly* the same function space $\mathcal{F}$ as would be used for traditional optimization methods (e.g. grid search). Naturally, this function space is bounded by the parameterized kernel function that is under consideration. Since the search spaces for hyperparameter optimization are identical, any search method that converges will *eventually* result in the optimal solution. Therefore we should not expect a significant increase in generalization performance.

The second model that we present aims to increase the generalization performance by means of drastically increasing the search space $\mathcal{F}$. In Section 2.3 we have learned that kernel functions can be constructed in a *modular* fashion. This combination is based on simple operations that allow us to create more complicated kernels from simple building blocks (see Eq. 2.30, Eq. 2.31, and Eq. 2.32). Effectively, this combination of kernels enlarges the search space $\mathcal{F}$. We expect that this increase of search space will result in finding better kernel functions, in terms of a higher generalization performance. However, this enlargement of the search space will come at a higher computational cost and specialization to the training set.

5.1 Evolutionary Hyperparameter Optimization

Finding optimal hyperparameters for Kernel Machines is not essentially different from other real-valued optimization problem[1]. The problem constitutes of finding an object variable vector $\mathbf{c}_{opt}$ that optimizes the objective function $F(\mathbf{c})$. Applied to the case of hyperparameter selection, this means that we are searching for the set of hyperparameters that optimizes the generalization performance. This directs us to the steps that we have to undertake to shape an evolutionary algorithm for our specific problem. Firstly, the hyperparameters must be encoded in an object variable vector. This step ensures that the evolutionary algorithm can deal with the representation of the problem. Secondly, an objective function needs to be defined for our *specific* problem. This objective function needs to evaluate the generalization performance for a given set of hyperparameters. For now it suffices to note that we have used k-fold cross validation as our performance measure. In k-fold cross validation the data set is subdivided in k equal parts. Then the learning machine is trained k times on $k-1$ sets and tested on the single subset that was not included in the training set. It is important to note that the machine is therefore never tested on a subset that was used for training. The problem of finding a suitable objective function will be dealt with in more detail in Chapter 8.

5.1.1 Genotype Representation

Thus far an important model decision has been ignored, namely, the exact type of evolutionary algorithm that we wish to use. In Chapter 3 we have seen that both GA and ES can be used for parameter optimization problems, albeit there are differences among both approaches. Generally, it is safe to assume that ES performs better on real-valued problems, whereas GA excels at combinatorial problems [5]. Given the nature of our problem, which is obviously a real-valued optimization problem, we opt to base our model on the ES approach. An additional minor advantage is that the encoding of the parameters is trivial, as the phenotype and genotype are identical in ES. It therefore suffices to create an object vector containing all the hyperparameters, i.e. the kernel parameters and the regularization parameter C, and the strategy parameters. An example chromosome for the RBF kernel is given by the vector

$$\mathbf{c} = [\gamma, C, \sigma_1, \sigma_2] \quad .$$

The first two elements in this vector are the RBF kernel parameter γ and the machine regularization parameter C. These hyperparameters are the object parameters in the chromosome. The last two elements are the endogenous strategy parameters, which in this case are the standard deviations for the object parameters (cf. Section 3.3.2). Obviously, the chromosome of the polynomial kernel would contain 6 elements; three hyperparameters (i.e. the degree d, the constant c, and C) and three standard deviation parameters, respectively.

5.1.2 Overview of the Hyperparameter Optimization Model

An overview of the model is depicted in Fig. 5.1. This model closely resembles the standard ES algorithm, as has been laid out in Algorithm 3.2. The model starts with an initial phase, or *bootstrap* phase, in which a parent population is randomly generated. This first generation of μ individuals is then evaluated using an objective function, after which fitness functions can be assigned to all individuals. Note that an individual

[1]A minor exception is the *degree* parameter in the polynomial kernel, as it is defined as $d \in \mathbb{N}$.

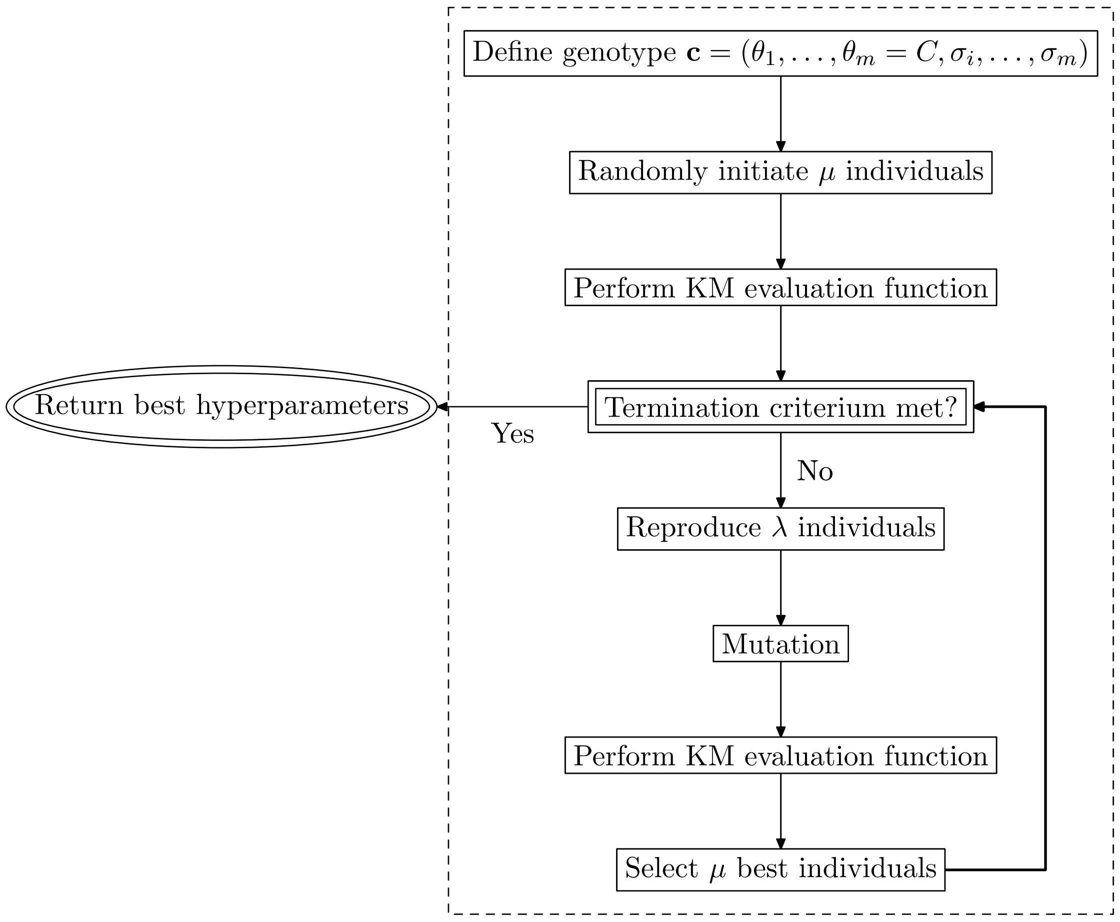

Define genotype $\mathbf{c} = (\theta_1, \ldots, \theta_m = C, \sigma_i, \ldots, \sigma_m)$
Randomly initiate μ individuals
Perform KM evaluation function
Termination criterium met?
Yes
No
Return best hyperparameters
Reproduce λ individuals
Mutation
Perform KM evaluation function
Select μ best individuals

Figure 5.1: Graphical overview of the evolutionary hyperparameter optimization model.

in this context consists of the hyperparameters, which are encapsulated in the genotype. Besides the hyperparameters, the genotype also contains endogenous strategy parameters (see Section 3.3.2). These strategy parameters are the standard deviations that determine the amount of mutation for each hyperparameter.

At this point the continuation of the model depends on one or more termination conditions. In case a termination condition has been met, the model should return the best individual, i.e. the individual with the highest fitness. If no termination condition have been met, then the model continues by applying a set of evolutionary operators on the population. In case of ES these operators are reproduction and mutation. A detailed explanation of these operators was given previously in Section 3.3. Shortly, these operators produce a new mutated offspring population of λ individuals. In case of (μ, λ)-ES this offspring is then pruned back to μ individuals based on their fitness values. The resulting population forms the parent population for the consecutive generation. In case of $(\mu + \lambda)$-ES the offspring is first merged with the original parent population before being pruned to μ individuals. This sequence is repeated until one of the termination conditions is met.

5.1.3 (μ, λ)-ES versus $(\mu + \lambda)$-ES

It is hard to argue whether the model should be based on (μ, λ)-ES or $(\mu + \lambda)$-ES, since we learned from Section 3.3 that both variants have advantages and disadvantages. To motivate our discussion we recapitulate the general heuristic for choosing either of the two variants. The typical application area of $(\mu + \lambda)$-ES are discrete finite size search spaces, such as combinatorial optimization problems [5, 3]. When the problem is an unbounded search space, typically real-valued search spaces, then (μ, λ)-ES should be preferred [101]. Nonetheless, Whitley presents empirical evidence that indicates that (μ, λ)-ES generally performs better than $(\mu + \lambda)$-ES [118]. We prefer to follow both the heuristic and the empirical indications and therefore we adopted (μ, λ)-ES for our model. Unfortunately, because of this decision we cannot guarantee that the search process will converge to a solution, as would have been the case with $(\mu + \lambda)$-ES. Nonetheless, empirical data has shown that for the type of problem under consideration (μ, λ)-ES would be expected to perform better. Furthermore, with (μ, λ)-ES the process is less likely to get stuck in local minima, as it prevents that a group of strong individuals dominate the population for many generations. In this situation the selection pressure on the individuals diminishes and this effectively disables an important Darwinian element of the model.

5.2 Evolving Kernel Functions using Genetic Programming

In order to tackle the second research problem, i.e. increasing the generalization capacity by means of more complex kernel functions, we propose a model based on GP principles. The idea is to construct complex kernel functions as combinations of simpler kernel functions. As noted before, this approach is inspired by the fact that certain operations on admissible kernels (i.e. kernels that satisfy Mercer's conditions) will result in a combined kernel function that is admissible as well. The operations for which this rule holds are the sum of two kernel functions, a weighted scaling of a kernel, and the product of two kernels. These operations were formally described in Eq. 2.30, Eq. 2.31, and Eq. 2.32, respectively. Constructing kernel functions using these operations guarantees us that the combined kernel function is itself a admissible. This means that we can extend the function space $\mathcal{F}$ of the search process as compared to simple parameterized functions, without risking to search for functions that are not a valid inner product in some feature space.

There are several reasons why this approach of combining kernels would make sense. Firstly, without this restriction on the operations we would search in an enormous function space. Suppose that the normal arithmetic operators in scalar and vector forms (i.e. addition, subtraction, multiplication and a protected division) were to be allowed. For *full* trees of depth 3 there are thus 3 non-terminals, which all can take one out of these 4 operators. If we ignore for a moment the terminal nodes, then there are $4^3 = 64$ possible combinations of trees for that scheme. In our model there are only two binary operators, namely addition and multiplication and the weighted scaling is a unary operator. This results in $2 \cdot 3^2 + 1 \cdot 3 = 21$ possible combinations for a full tree of size 3^2. This calculation shows that our approach is more restrictive on the search space. However, we restrict ourselves to admissible kernels and thus the more sensible candidate solutions. This is a problem specific heuristic that should help to keep the model computationally tractable.

[2]If the root node is one of the two binary operators, then there are 3 possibilities for each its two child nodes (i.e. $2 \cdot 3^2$). If the root node is the weighted scaling operator, then its single child can take 3 possible combinations (i.e. $1 \cdot 3$).

⟨*kernel*⟩	$\rightarrow$	⟨*add_kernels*⟩ | ⟨*multiply_kernels*⟩ | ⟨*weighted_kernel*⟩ | ⟨*polynomial*⟩ | ⟨*rbf*⟩	
⟨*add_kernels*⟩	$\rightarrow$	⟨*kernel*⟩ '+' ⟨*kernel*⟩	
⟨*multiply_kernels*⟩	$\rightarrow$	⟨*kernel*⟩ '$\times$' ⟨*kernel*⟩	
⟨*weighted_kernel*⟩	$\rightarrow$	a '$\times$' ⟨*kernel*⟩	for $a \in \mathbb{R}^+$
⟨*polynomial*⟩	$\rightarrow$	'$(\langle \mathbf{x}, \mathbf{z} \rangle + c)^d$'	for $d \in \mathbb{N},\ c \in \mathbb{R}^+$
⟨*rbf*⟩	$\rightarrow$	'$\exp\left(-\gamma \|\|\mathbf{x} - \mathbf{z}\|\|^2\right)$'	for $\gamma \in \mathbb{R}^+$

Figure 5.2: The context-free grammar – in Backus-Naur form – that constrains the generated expressions for the GP model.

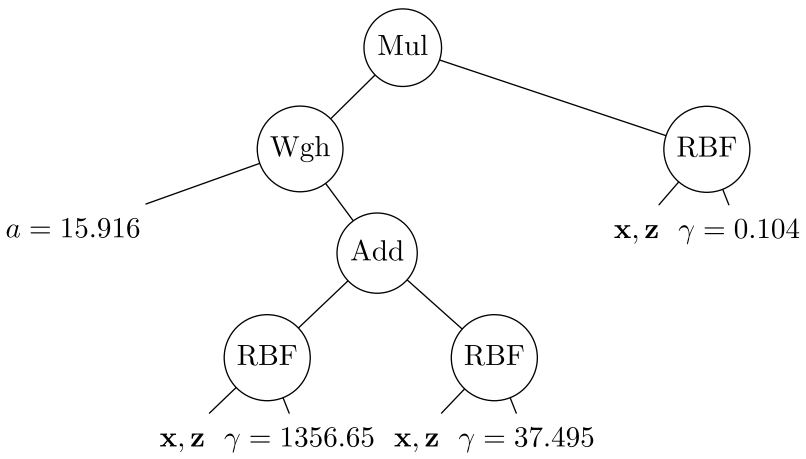

Mul
Wgh
RBF
$a = 15.916$
Add
$\mathbf{x}, \mathbf{z}$ $\gamma = 0.104$
RBF
RBF
$\mathbf{x}, \mathbf{z}$ $\gamma = 1356.65$
$\mathbf{x}, \mathbf{z}$ $\gamma = 37.495$

Figure 5.3: An example of a tree generated by the GP model.

5.2.1 Genotype Representation

The most natural way to represent the described functions in a genotype is to model them as parse trees. An additional grammar constraints the generated functions so that only genotypes can be constructed that adhere to the model that was described in the previous section. This grammar is described in Fig. 5.2. The non-terminals are the operators, i.e. addition, multiplication and weighing, and the various atomic kernel functions. Examples of the latter are the RBF and the polynomial kernels. The set of terminal nodes consists of the various hyperparameters (e.g. γ, d etc.) and the input vectors $\mathbf{x}, \mathbf{z} \in \mathcal{X}$. Clearly, imposing a constraining grammar on the genotypes implies that we need to use *Strongly Typed Genetic Programming*, which was described previously in Section 3.4. An example of a genotype tree based on the grammar is depicted in Fig. 5.3.

The attentive reader will note a minor deviation in the non-terminals, as compared with the operators in the original equations. In this grammar the linear addition operator has no weights, which were c_1 and c_2 in Eq. 2.30. The reason for this is that these weights are superfluous, as there is already a unary weighing operator present in the grammar. The original weighted addition operator can easily be constructed by having two weight scaling operators as child nodes for our simple addition operator, as is demonstrated in Fig. 5.4.

So far we have neglected the method specific parameters in our model, such as the regularization parameter C for SVM and LS-SVM. The most natural way to integrate these in the model is by means of a second genotype. Every individual will thus have two genotypes; the first represents the kernel function including

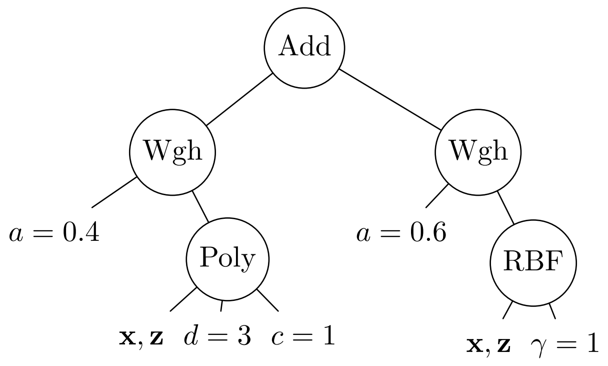

Add
Wgh
Wgh
$a = 0.4$
Poly
$a = 0.6$
RBF
$\mathbf{x}, \mathbf{z}$ $d = 3$ $c = 1$
$\mathbf{x}, \mathbf{z}$ $\gamma = 1$

Figure 5.4: Constructing the original addition operator from Eq. 2.30 using the addition and weighing operators from our grammar.

its parameters, whereas the second genotype contains the parameters for the specific algorithm.

5.2.2 Overview of the Kernel Evolution Model

The architectural overview of the kernel evolution model is very similar to the previous model. This overview is depicted in Fig. 5.5. It is not hard to notice the similarity with the ES model, which obviously can be contributed to the fact that both models are different flavors of EC. The model starts by randomly generating a population of s individuals. These individuals are evaluated according to the objective function, after which the termination conditions are verified. In case it has been met, then the individual with the highest fitness (i.e. the lowest error) is returned. Alternatively, the algorithm enters the main loop, in which various operators are applied to the population, until exactly s new offspring individuals are generated. This offspring is then evaluated, after which the routine continues at the point where the termination conditions are checked.

5.2.3 Operators on the GP Trees

In the overview (i.e. Fig. 5.5) the creation of offspring is depicted as a single step. In reality the situation is more complex, as there are various operators that act on individuals to reproduce offspring. In our model we want to use the most common reproduction and mutation operators for GP, which were previously described in Section 3.4.2. These operators are:

1. A *reproduction* occurs with probability p_r, which means that an individual is directly copied into the offspring population, without any kind of mutation.
2. With probability p_c *crossover* recombination occurs, which means that two parents are chosen that exchange a subtree at a random crossover point. The two new individuals are *both* inserted in the offspring population.
3. A *random mutation* occurs with probability p_m, in which the selected individual is mutated by substituting one of its subtrees with a new random subtree.
4. A *shrink* mutation can happen with probability p_s. This operator replaces a subtree with one of the branches of this subtree and thus reduces the size of the tree.

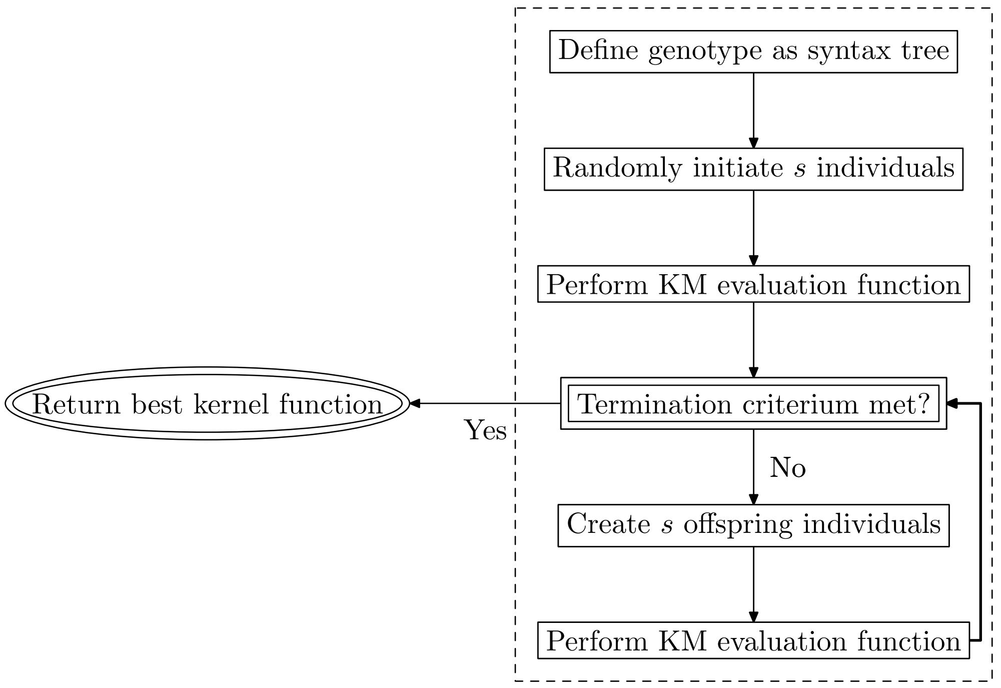

Define genotype as syntax tree
Randomly initiate s individuals
Perform KM evaluation function
Termination criterium met?
Return best kernel function
Yes
No
Create s offspring individuals
Perform KM evaluation function

Figure 5.5: Graphical overview of the evolutionary kernel generation model.

5. A *swap* mutation happens with probability p_w. The swap operator replaces a subtree in the individual with another subtree, effectively swapping two branches of the same tree.
6. A *pdf* parameter mutation happens with probability p_p, in which a real-valued parameter is mutated according to a probability density function.

The first four operators are standard operators in the field of EC. The fifth operator, however, is defined specifically for our type of problem. The process of evolving kernel functions can be considered a super-problem of finding optimal hyperparameters. This can easily be understood by the realization that when a combined kernel has been constructed, we still have to deal with the problem of finding the optimal hyper-parameters for this combined kernel. The parameter mutation operator is included to perform this task of optimizing the hyperparameters for the combined kernels. Generally, mutation of parameters in GP is done randomly, i.e. an ephemeral constant is substituted for a randomly generated new one. This mutation operation was described in Section 3.4.2. However, we believe that for our type of problem a mutation according to a pdf distribution would be more effective. The advantage should come from the fact that knowledge gained during the search process is used instead of discarded. Its functioning is identical to the mutation operator that is used in ES; the only difference is that is has been adapted to work within a GP environment with tree-based genotypes.

Chapter 6

IMPLEMENTATION

In this chapter the implementation of both models will be described in detail. It is safe to say that implementing a Kernel Machine should not be considered trivial. The same goes for systems based on evolutionary computation. One can imagine that implementing a combination of both techniques increases the complexity only further. Firstly, the requirements, functional as well as non-functional, will be laid out. Hereafter the implementation of a Kernel Machine and the two EC-based models will be described.

6.1 Requirements

An obvious requirement for the implementation of our systems is, of course, that it should be able to – empirically – answer the research questions that we put forward in Section 1.3. However, we prefer to state the obvious, as this has been the *primary* requirement for the implementations.

Furthermore, another non-functional requirement is that it should be desirable to implement in such a way that it can be applied in real-life situations. The machine learning techniques implemented for this study should contribute to the RobotCub project. More specifically, the code should – to a certain extent – be integrated in the YARP robotics framework [71]. This requirement imposes some new desired properties of our implementations:

- YARP has been written in C++; our implementation should preferably be written in the same language.
- YARP is an *open source* project released publicly under the *GNU General Public License*. As such, proprietary software or libraries should be avoided.
- Preferably, the implementation should run under both Microsoft Windows and Linux.

The time needed to run the experiments is an important issue, given the research questions under consideration. Training a Kernel Machine can take a considerable amount of time. Consider that we will have to do several experimental runs and that each run a machine is trained up to thousands of times. This raises the requirement that the implementations should be computationally efficient, as to increase the feasibility of empirical validation. The more efficient the implementation is, the larger the data sets that we can use for experiments or the more experimental runs we can perform. This will contribute to the overall quality of our results.

6.2 Least Squares SVM Framework

The two proposed models, as described in Chapter5, can clearly be applied to every type of kernel-based algorithm. Examples of such Kernel Machines are SVM and LS-SVM. Nonetheless, the empirical validation on various kinds of Kernel Machines would be infeasible, due to the vast amount of algorithms available[1]. Therefore we have opted to consider solely one type of Kernel Machine for this study, i.e. the LS-SVM algorithm. The motivations for choosing LS-SVM instead of other algorithms are:

1. The algorithm is identical for both classification and regression problems. This means that a single implementation can be used to test on both type of problems. Furthermore, the results of this study can more easily be applied on real problems, such as those found in the RobotCub project (see Section 1.1).
2. The solution for the optimization problem in LS-SVM can be obtained by solving a system of linear equations. As we will see later, specialized linear algebra libraries can be used to easily create a custom implementation of the algorithm.
3. As a result of the second point, the computation time needed to solve the optimization problem is independent on any of the hyperparameters. This simplifies the comparison of different hyperparameter selection algorithms, as we can assume that each training run for a data set will take constant time, regardless of the chosen hyperparameters.
4. Empirical research has shown that LS-SVM shows similar generalization performance as SVM [95].

One of the few frameworks for LS-SVM has been created by the group of Suykens et al. at the University of Leuven [87]. This toolbox, named *LSSVMLab*, is developed in Matlab. Despite the availability of this toolbox, we have chosen to implement our own LS-SVM framework. Obviously, this is more time consuming than using a publicly available framework. However, there are certain advantages to developing our own customized implementation. Firstly, we have complete control over the architecture and the language that we use. This will prove to be beneficial in later stages, when we have to integrate the framework with the Evolutionary Computation framework. Furthermore, we can design the application in such a way that it can cope well with various kinds of kernel functions, such as the combined kernels described previously. Lastly, by creating a custom framework we can ensure that it can eventually be integrated in a production environment, such as the YARP robotics framework as described in Section 1.1. An external implementation, e.g. LSSVMLab, would most probably conflict with one or more of these objectives. For example, the required integration with YARP, which is an freely distributed open source project, prohibits the use of proprietary software like Matlab.

A high level overview of the LS-SVM framework that we developed is shown in Fig. 6.1. Henceforth we will refer to this framework as LibLSSVM. We refer the reader to Appendix A for a more detailed UML

[1]There exist literally dozens of kernel-based algorithms.

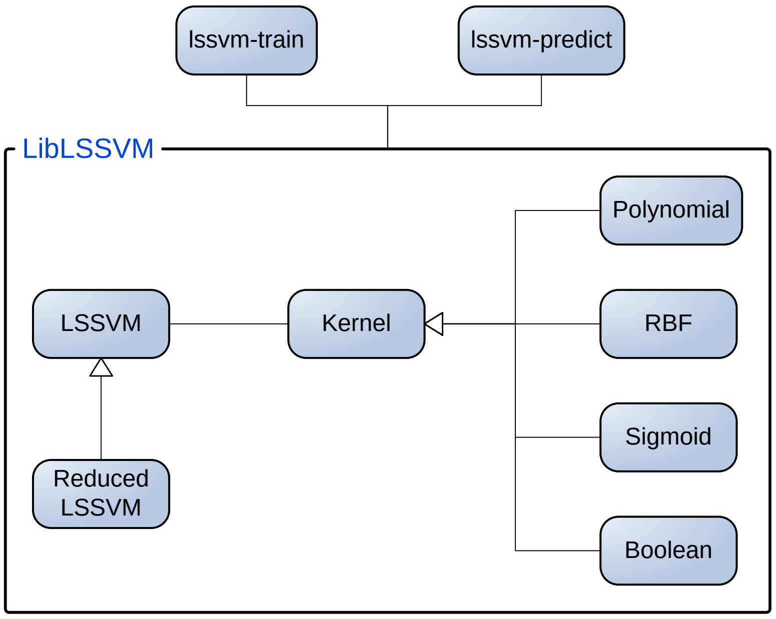


Figure 6.1: Simplified overview of the LibLSSVM architecture.

representation of the architecture. The core of our system revolves around only two classes, namely the *LSSVM* and *Kernel* classes. The former implements the algorithm and thus the routines to solve the linear optimization problem. As we have explained before, the linear optimization problem in LS-SVM is solved by means of a system of linear equations. In order to solve this system, it suffices to do a matrix inversion (see Eq. 2.37). This inversion step can be optimized by exploiting the fact that the Kernel matrix can be rewritten as a positive definite matrix and then solving the system by means of Cholesky decomposition; these steps were described in Section 2.4.1. Our application uses Cholesky decomposition, as to increase the temporal efficiency and numerical stability of the implementation. Furthermore, the LSSVM class is extended by a *ReferenceLSSVM* class. The latter implements the reduced LS-SVM algorithm that has been described in Section 2.4.2. The subset of samples that are used for describing the model is selected randomly from the total training set. Other sparsification schemes exist, but selecting them randomly gives a nice trade off between computational complexity and performance. Finally, the fast leave-one-out algorithms have been implemented for both the standard and the reduced variant of LS-SVM.

The second important class in the system is the *Kernel* class, which is defined as an abstract base class. It defines an interface for a kernel functions and is overwritten by concrete classes (e.g. *RBFKernel* for the RBF kernel function). We have chosen to use this type of *object-oriented* design so that we can easily extend our system with new kernel functions. Any new class that extends *Kernel* can be used as a kernel function within the system. The way in which kernels in the system are created is inspired by the *factory*-pattern [41]. Thanks to this pattern the system does not have to be aware of each possible kernels that could be instantiated. It also helps the serialization of the model by storing information about which kernel has been used to train the system.

The user-interface to the LibLSSVM framework is provided by means of two separate command-line executables. The first, i.e. *lssvm-train*, is used to train the system, after which the model can be saved to a file. The second executable, i.e. *lssvm-predict*, is used to make predictions based on a given model. The tasks of these applications are to parse the command-line parameters, to convert the data sets into a vector notation, and to interact with the LSSVM and Kernel objects. A third small application has been written to normalize data sets. The implementation details of these executables are considered trivial and therefore excluded from this report. The application can handle regression as well as binary classification problems. In the former case the error measure that is returned is the mean-squared-error, whereas in the latter case the classification error measure is used. The type of problem is determined by a simple form of autosensing, which classifies a problem as regression if and only if at least one of the output values is *not* $+1$ or -1. If necessary, this autosensing feature can be overridden by means of a command line parameter.

6.2.1 Library for Linear Algebra: Atlas

Performing Cholesky decomposition and inverting a matrix are relatively simple from an analytical mathematical point of view. From a computational and numerical point of view these operations are, unfortunately, much less trivial. Problems generally encountered with implementations are that the performance is far from optimal, or unexpected results due to numerical difficulties. Reference implementations exist to aid the programmer, such as those found in the excellent *Numerical Recipes* book [90]. Nonetheless, we have chosen to use the publicly available *Atlas* library for the linear algebra routines [116] needed in our system. The Atlas library has the following advantages:

- The quality of the implementation is higher than any implementation that we could do ourselves.
- The library uses platform specific optimizations to increase the performance. This does mean, however, that ideally it should be built on the target platform.
- Atlas can easily be used in C++ projects thanks to the *Boost Numeric Bindings*. These are a set of wrapper functions and classes around the functions of, among others, Atlas.

Basing LibLSSVM on a Linear Algebra library like Atlas gives us the best of both worlds. On the one hand, we can use this library to ensure optimal performance and quality on the critical aspects of our system. On the other hand, we are have full control over the architecture of our system.

6.2.2 Numerical Difficulties due to Singular Matrices

During the implementation and testing phase of the reduced variant of LS-SVM a serious problem became apparent. Recall from Eq. 2.42 in Section 2.4.2 that the model of the reduced LS-SVM is calculated by means of the inverse of the matrix

$$\mathbf{E} = \left(\mathbf{K}_{m\ell}\mathbf{K}_{\ell m} + C^{-1}\mathbf{K}_{mm}\right) \ ,$$

where the matrix is temporarily denoted with $\mathbf{E}$ for convenience. For certain combinations of data sets and kernel functions, however, the inverse of the matrix was unfortunately not correct. This has been validated by analyzing the product $\mathbf{E}\ \mathbf{E}^T$, which obviously should yield an identity matrix $\mathbf{I}$. However, this is not the case, which indicates that the matrix $\mathbf{E}$ is singular in these situations[2]. The result of this problem are very unstable solutions. One of the main causes of this problem is that computers operate with a finite representation for real values. In these numerical calculations an invertible matrix that is close to an

[2]Recall that a matrix $\mathbf{A}$ is *invertible*, or *nonsingular*, if and only if the determinant of the matrix is nonzero, i.e. $|\det(\mathbf{A})| > 0$.

invertible matrix may be problematic. The problem can be resolved by adding a regularization term to the matrix, i.e.

$$\mathbf{E} = \left(\mathbf{K}_{m\ell}\mathbf{K}_{\ell m} + C^{-1}\mathbf{K}_{mm} + q\mathbf{I}_m\right) \quad , \tag{6.1}$$

where q is the regularization term. This enforces the matrix to be full rank and thus invertible, since the term is multiplied with the identity matrix. In the standard variant of LS-SVM C^{-1} is added to the diagonal of the kernel matrix and therefore it is less prone to these numerical difficulties. Note that this regularization term that we add to the formula will have a minor effect on the solution that is found by the algorithm. However, this does not impair the study presented in this report, since the goal is to compare the search performance of multiple methods. All these search methods are equally affected by this minor modification.

An important issue is how to choose q in such a way that it effectively stabilizes the algorithm over a wide range of configurations (i.e. kernel function and hyperparameters). One could opt to set q to a small constant, however, this only partially resolves the problem. Note that the range for C that is commonly used in SVM and LS-SVM can range from up to $\left[2^{-20}, 2^{20}\right]$ in practical settings. Now, suppose that we choose $C = 2^{-20}$, then the inverse C^{-1} will be 2^{20} and the matrix $C^{-1}\mathbf{K}_{mm}$ will most likely "dominate" the matrix $\mathbf{E}$. As we have experienced in practice, the constant regularization term will then not be able to compensate for the large values in the term $C^{-1}\mathbf{K}_{mm}$. The result is that $\mathbf{E}$ will still be an ill-defined matrix and the inversion produces unstable solutions. Our solution to this problem is to set the regularization term proportional to the inverse of C, i.e. $q \propto C^{-1}$. In our preliminary testing of the implementation this has shown to stabilize the solutions over a wide range of configurations.

6.3 Evolutionary Computation Framework

The second important element in the implementation of the two models is the framework for Evolutionary Computation. We have focused on existing EC frameworks, since implementing an EC framework ourselves would take a considerable amount of time[3]. However, there are several frameworks publicly available that fit the requirements and scope of this work. From these options we have chosen the OpenBeagle framework [38], since it seems to fit our purposes very well. An enumeration of the most important motivations to adopt the OpenBeagle framework is given below. Thereafter we give a very brief introduction to OpenBeagle for both general understanding and explanation of the terminology.

- Since it is written in C++ we can easily integrate LibLSSVM into our optimization applications.
- It supports various flavors of EC, among which ES and STGP.
- Due to strong *Object-Oriented* foundations and the use of *design patterns* it is easy to incorporate custom genotype representations, objective functions, and operators. This allows us to create highly customized applications, without having to implement all the basics on our own.
- The evolver model, i.e. the arrangement of operators of the algorithm, can easily be modified by means of configuration files. Combined with the rich set of operators in OpenBeagle this means that we can easily experiment with different configurations.

[3]For instance, version 3.0.1 of OpenBeagle contains more than 30.000 lines of source code. This *excludes* empty lines and comments.

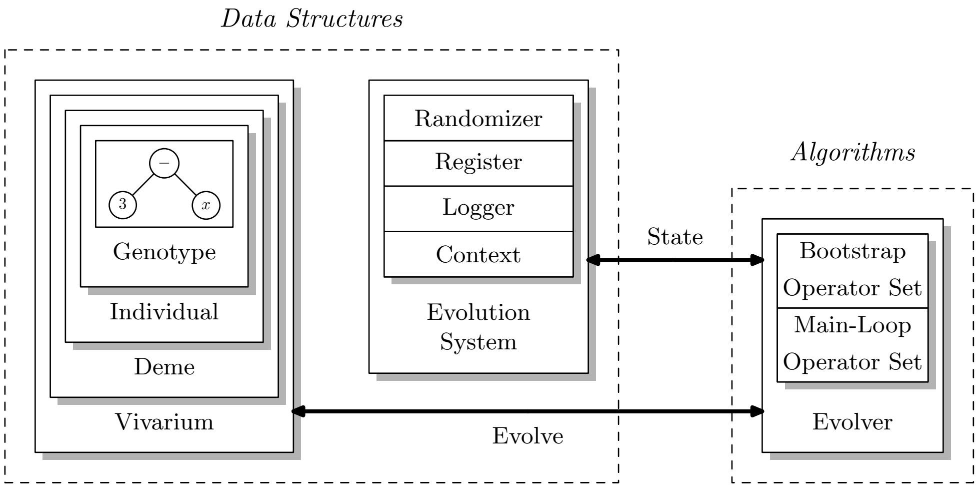


Figure 6.2: Global overview of the OpenBeagle architecture [38].

- OpenBeagle is actively being developed and has been around for many years. In other words, we can prevent ourselves spending much time on testing and fixing bugs.

An overview of the architecture of the OpenBeagle framework is depicted in Fig. 6.2. The framework is integrated in a modular way, which makes that all components shown in this figure can be replaced or specialized independently. This contributes to great extent to the flexibility and extensibility of the framework and in a practical sense simplifies the implementation of various EC flavors.

The architecture can roughly be decomposed in three modules, namely, the *population*, the *evolution system* and the *evolver*. The population comprises four entities that are used to structure the population in a hierarchy. On the top level, the complete population is contained in a *vivarium*. This can be considered a container of all individuals present in the evolutionary system. *Demes* are environments in which groups of individuals evolve independently. Nonetheless, at each generation individuals can migrate between the demes of a vivarium. A vivarium can thus contain one or more demes, whereas a deme contains one or more individuals. Both vivaria and demes keep track of statistics and a so-called *hall-of-fame* (i.e. a list of most fit individuals), respecting the scope imposed by the hierarchy.

Potential solutions to a problem are represented as *individuals*. Individuals consist of a fitness measure and one or more genotypes. Only in particular cases it can be more convenient to use individuals with more than one genotype. A *genotype* contains the genetic description of the individual, which can range from various binary encodings to expression trees or graphs. The organization of individuals, genotypes and fitness measures, which all can be specialized to need, conform to the criteria laid out before.

The *evolution system* that is included in the system has a supportive task. It contains a context, logger, register and a randomizer. The *logger* and *randomizer* do exactly what their name suggests. The *context* in the framework keeps track of the current state of the evolving process. The state is made up by data such as the current generation, the currently processed individuals, etc. Lastly, the *register* is used to centralize information in the evolutionary system. Other entities, such as operators and individuals, can therefore dynamically access a centralized repository of settings.

Finally, the *evolver* captures the evolving processing of the algorithm. It contains two sets of operators that are iteratively applied to the demes of the vivarium. The first set of operators, i.e. the *bootstrap operator set*, is the set of operations that are used to create an initial population for each deme. After the set of bootstrap operations have been finalized, the evolving process enters the *main-loop operator set*. This set of operators is iteratively applied to each deme for each generation, until one of the termination conditions has been met.

Clearly, the *operators* perform a crucial function within OpenBeagle. Recombination, mutation, evaluation and various other kinds of operators have been included in the framework. As explained, the evolver determines the order in which these operators will be applied to each deme. This means that the user of the framework is free to define any preferred order of operations. Furthermore, custom operators can be created using inherited polymorphism so that the not only the order, but also the operators themselves are fully configurable.

6.4 Evolutionary Hyperparameter Optimization Application

The abovementioned OpenBeagle framework has been used to implement an application for the hyperparameter optimization model using ES. We will refer to this application as EvoKM^{ES}, which is an abbreviation for *Evolutionary Kernel Machine using Evolution Strategies*. This application uses the ES implementation in OpenBeagle to optimize the hyperparameters for our LS-SVM implementation LibLSSVM.

The system architecture for EvoKM^{ES} is shown in Fig. 6.3. Again, the architecture shown here is a simplified variant of the real architecture. This will help to make the concepts clearer, without having to deal with an abundance of implementation specific details. The system is organized in two packages, which we will briefly introduce.

LibEvoKM is the package that contains classes that are not dependent on the flavor of ES. This means that these classes are generalized and could easily be used for a similar application based on GA, independent of the chosen representation. Furthermore, this package can be partially reused for the GP application.

EvoKM^{ES} contains the classes that are specific to our ES based model. This package handles the transformation of ES individuals into a generalized format that is more suited to our problem.

It may not be clear how this architecture relates to the hyperparameter optimization model that was proposed in Section 5.1. This is mainly due to the fact that the majority of the model is provided by the OpenBeagle framework. The most relevant aspect in developing EC applications is defining an *Evaluation Operator*. This operator evaluates each individual, in other words, it assigns a fitness value to each individual that it processes. It may be clear that an evaluation operator is completely domain specific, therefore this needs to be defined by the user of the framework. Our evaluation operator is defined as an abstract class, which is extended by three concrete classes. These concrete classes all implement a different objective function:

1. The *k-fold cross validation* measure. A detailed explanation of this performance measure will be given in Chapter 8.
2. The fast *leave-one-out cross validation* measure, as described in Section 2.4.3.
3. A *split training/testing set* performance measure.

Communication with the LibLSSVM framework is encapsulated by means of a wrapper class, i.e. *LibLSSVMWrapper*. Although not strictly necessary, this wrapper provides a single interface with our LS-SVM framework. This helps to centralize certain functionality, such as the conversion of parameters into

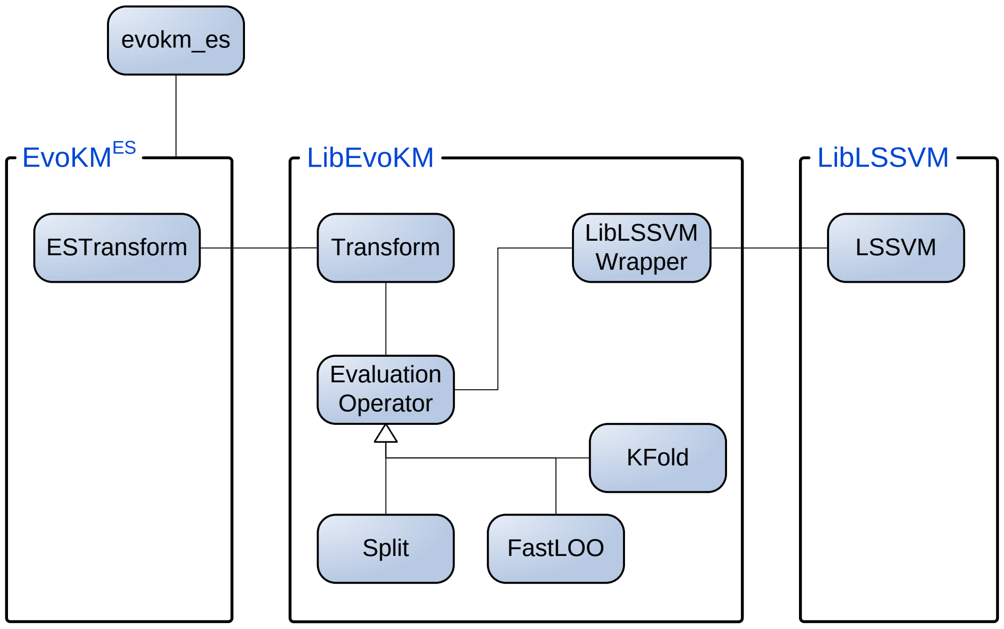


Figure 6.3: Overview of the EvoKM^{ES} system architecture.

logarithmic scale. Furthermore, this class takes responsibly for the proper construction of the learning machine and the kernel functions.

Lastly, the *Transform* and *ESTransform* classes have been designed to deal with the problem of converting the genotype representation into a phenotype. As explained before, the OpenBeagle framework – and EC methods in general – operate on individuals, which on their turn can consist of one or more genotypes. In LibLSSVM, however, the individual must be represented with its phenotype, i.e. a vector of real-valued hyperparameters. The task of the interface class Transform is to convert an individual in such a way that it produces the hyperparameters in the correct format. The idea is that, potentially, the Transform class will be overridden by concrete classes targeted for a specific representation. The ESTransform class is an example of one such concrete classes for the standard ES representation. The advantage of this design structure is that the rest of the architecture is completely independent of the chosen representation scheme. Effectively, the system can completely be reused for various kinds of representations (e.g. ES, real-valued GA, GA with gray encoding). It suffices to supply the Evaluation Operator with an implementation of the Transform class that corresponds to the representation scheme that is used at the moment. Nonetheless, it has to be noted that only the Transform class for ES representations is implemented. This feature is thus reserved for future work.

The user will interface with EvoKM^{ES} using an executable named *evokm_es*. Command line parameters can be used to either load a configuration file or to set configuration settings manually. This functionality is provided by default by the OpenBeagle framework. Due to the large amount of possible configuration options it is most convenient to use a configuration file. As we have seen before, this configuration file can – and should – also be used to define the evolver model. Settings such as the training file, an identifier for used kernel function and parameter ranges are also read from this configuration file. An example configuration file for EvoKM^{ES} is shown in Appendix B.

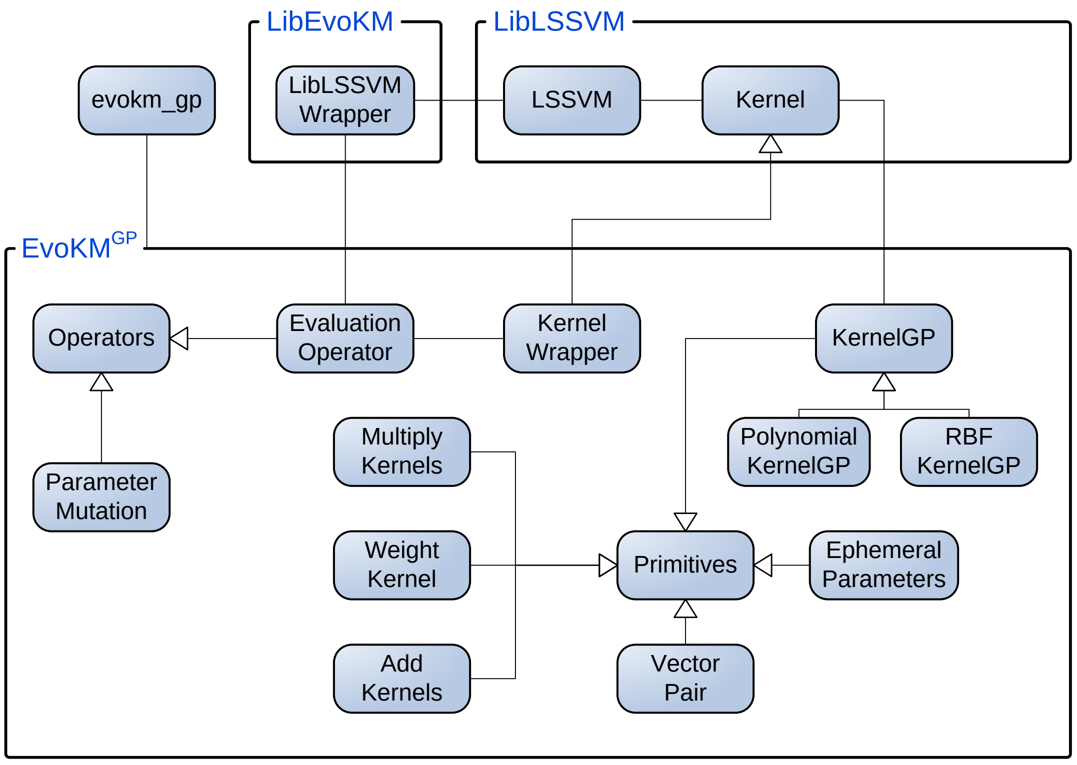


Figure 6.4: Simplified overview of the EvoKM^{GP} system architecture.

6.5 Evolutionary Kernel Generation Application

A second application has been implemented by us for the evolvement of kernel functions using GP. We have named this application EvoKM^{GP}, in tradition with the former one. The architecture of this application, which is shown in Fig. 6.4, is considerably more complex than the previous one. Nonetheless, we were able to implement our model without any major problems.

As with EvoKM^{ES}, also in this architecture the *Evaluation Operator* takes a central place. Evaluating individuals has become more complicated, due to the fact that complete kernel functions are being evaluated instead of only the hyperparameters. Consequently, the Evaluation Operator needs to feed the Kernel Machine (i.e. LibLSSVMWrapper) with the appropriate kernel function that corresponds with an individual. The class *KernelWrapper* has been designed to perform the translation from individual into kernel function. The following detailed explanation of the interactions in the system during the evaluation of an individual clarifies this:

1. The *Kernel* of the LibLSSVMWrapper is set to an instance of KernelWrapper. LibLSSVM can use KernelWrapper as if it were a normal kernel, since the wrapper is a subclass of the *Kernel* class found in LibLSSVM itself.

2. The Evaluation Operator receives an individual and uses this to set the regularization parameter C (i.e. the second genotype in the individual). Then it passes the individual to the KernelWrapper.

3. The Kernel Machine is trained as normal, only that it uses the KernelWrapper as kernel. Nonetheless, this is completely transparent from the perspective of the machine; it uses the normal Kernel interface

and does not notice any difference.

4. Every evaluation for a pair of vectors $\mathbf{x}$ and $\mathbf{z}$ (i.e. to build the kernel matrix) the KernelWrapper sets the vectors in the individual. The entry points for these vectors are the non-terminals of the type *VectorPair*.

5. The individual can be evaluated now the last variables have been filled (i.e. the vectors). The parse tree is evaluated and the result is returned by the wrapper.

6. Once the full Kernel Matrix has been built, then the kernel machine is can start its normal training routine. This results in an error based on the objective function that is being used (e.g. k-fold cross validation). This error is being returned to the Evaluation Operator.

7. The evaluation is completed as the Evaluation Operator returns the calculated fitness for the individual under consideration.

As we can see, the abstraction of the Kernel in LibLSSVM has major advantages here. The LSSVM class can be used with any type of kernel, such as the parse tree that we use here, as long as it implements the right interface. The KernelWrapper class effectively hides all GP specific details from LibLSSVM.

Another very important aspect of this architecture is the representation of the individuals. In Section 5.2.1 we proposed a genotype representation for our kernel functions, which we have followed in our implementation. Terminals and non-terminals, in other words the nodes in the parse tree, share a common superclass *Primitive*[4]. The distinction between terminals and non-terminals, or depending on the terminology branches and leaves, is made by the number of children. Non-terminals have a non-zero amount of children, whereas terminals have no children at all. This means that there is no other distinction between both from a conceptual point of view, other than an attribute that defines the amount of children. In our specific model we can distinguish four different types of primitives:

1. The three operators on kernel functions, i.e. *MultiplyKernels*, *AddKernels* and *WeightKernel*. These take either two kernels as children (MultiplyKernels and AddKernels) or a real value and a kernel (WeightKernel). For all three the type it returns to its parent node is a kernel.

2. Two atomic kernel functions, i.e. *RBFKernel* and *PolyKernel*, which share an abstract superclass *KernelGP*. The RBFKernel takes two child nodes, namely, a VectorPair and an EphemeralParameter for the kernel parameter γ. The PolyKernel takes three child nodes in total, i.e. the two kernel parameters c and d and a VectorPair. Obviously, both atomic kernels return a kernel to the parent node.

3. The VectorPair, which captures the two vectors $\mathbf{x}$ and $\mathbf{z}$ that are passed to a kernel function. A VectorPair is variable in the parse tree, in the sense that it can dynamically be set to any pair of vectors. It does not have any child nodes and can only have one a *KernelGP* primitive as parent.

4. The *EphemeralParameters*, which are the hyperparameters. In the architecture in Fig. 6.4 these are left out for clarity. There are *ParameterDegree* and *ParameterConstant* that belong to PolyKernel, *ParameterGamma* that is used with RBFKernel and *ParameterWeight* that is a child of the WeightKernel operator. Furthermore, *ParameterC* stores the value for the trade-off parameter C. This primitive is only used in the second, dedicated, genotype. All of the EphemeralParameters are terminals and can thus not have any child nodes.

[4]This Primitive interface is defined by the OpenBeagle framework.

The main advantage of modeling the different parameters separately is that this way we can define distinctive properties for each of them. These properties are their range and whether they use a logarithmic scale or not. For instance, for the parameter γ generally it makes sense to use a logarithmic scale and a broad range. On the other hand, for the degree parameter d a limited range on a linear scale makes more sense. Our setup gives us full freedom in defining these properties for each parameter separately.

A last important element in the overview is the *MutationOperator*. This operator implements the ES inspired mutation strategy, as discussed in Section 5.2.3. The functioning of this operator is relatively straight forward in OpenBeagle; the framework feeds with a certain probability individuals to the operator. All the parameters that can be subject to this type of mutation contain a strategy value (i.e. the standard deviation) besides their original value. The mutation operator takes a random step based on this standard deviation and then updates both the value and the strategy value accordingly. The formulae for these updates used in our implementation are the those described in Section 3.3.2.

The interface to the user, i.e. *evokm_gp*, is similar to that of EvoKM$^{\text{ES}}$. Both of these command line applications make use of the functionality offered by OpenBeagle and therefore share a common interface. Also for EvoKM$^{\text{GP}}$ the configuration files are used for defining the parameter settings and the evolver model. An example configuration file for EvoKM$^{\text{GP}}$ has been added in Appendix B.

Part III

Experimental Results

Chapter 7
EXPERIMENTAL RESULTS

In this chapter we will describe the results of the experiments that we have conducted. The goal of our experiments is to empirically verify our two models, which were proposed in Chapter 5. A set of six benchmark data sets have been chosen for our experiments, containing both regression as well as classification problems. The empirical validation can be subdivided in three phases, which are:

1. Optimal hyperparameters have been determined using the traditional grid search method. These results will be used as a reference for our two models.

2. The optimal hyperparameters are again optimized, but now using our EvoKM$^{\text{ES}}$ model. The results of our model are compared with the reference, where we emphasize on the temporal requirements of both techniques. With this experiment we wish to verify our first research question, namely, whether EC can help to find good hyperparameters more quickly than with the traditional optimization method (cf. Section 1.3).

3. In the last phase, both the kernel function and the hyperparameters are optimized using our EvoKM$^{\text{GP}}$ model. The results of this experiment will be compared with the reference, where this time we consider the quality of the solutions. In other words, we want to investigate whether EvoKM$^{\text{GP}}$ can find kernel functions that will reduce the prediction error of the Kernel Machine. This experiments validates our second research question.

This chapter will commence with an overview of the experimental setup in Section 7.1. This overview will list all aspects that hold for all experimental phases. The experimental results using the grid search optimization method will be described in Section 7.2. As explained, these results will be used as a reference for our EvoKM$^{\text{ES}}$ and EvoKM$^{\text{GP}}$ optimization methods. The results obtained using these models will be described in Section 7.3 and Section 7.4, respectively.

7.1 Experimental Setup

All the distinct experimental phases in this study have their own specific setup. However, there are many common aspects in the experimental setup that are identical in all phases. The intrinsic properties of the distinct experiments will be covered in the corresponding sections.

7.1.1 Kernel Machine

In Section 6.2 we explained that we used the LS-SVM algorithm in our implementation of the model. Obviously, our experiments are based on this type of Kernel Machine. More specifically, the experiments have all been conducted with the reduced variant of LS-SVM. The reason for using this approximate variant is to reduce the computational requirements in both temporal and spatial sense. Recall that EC is essentially a stochastic approach, so experiments will have to be repeated multiple times to get statistically significant results. The size of the subset that is used to describe the model is 10% of the total subset, rounded to the nearest integer. We can describe this alternatively as $\frac{m}{\ell} \approx 0.1$ (cf. Section 2.4.2). The subset is selected by taking the first m samples of the data set. This is acceptable, since the data sets are randomly reordered during the preprocessing phase.

In all the experiments we have considered the two most commonly used kernel functions. These kernel functions are the RBF kernel function and the polynomial kernel function, see Eq. 2.28 and Eq. 2.27, respectively. The reason for limiting ourselves to these kernels is to keep the experiments feasible in terms of computational demands.

7.1.2 Objective Function and Error Measure

The performance measure for the data sets is k-fold cross validation, where $k = 5$. The arguments for choosing this performance measure will be given in Chapter 8. Furthermore, we have decided to treat both classification as well as regression problems identically in our experimental framework. The only distinction between both is the error measure. For classification problems, the error measure is simply the classification error, i.e. the percentage of misclassified testing samples. In case of regression problems, the error measure is the *Mean Squared Error* (*MSE*) of all testing samples. This is a common error measure for regression problems in the field of machine learning. Both these error measures are averaged over the 5 folds.

7.1.3 Data Sets

We have selected six different benchmark data sets for our experiments. Four of these data sets are regression problems, whereas the remaining two are binary classification problems. Three of the data sets are well-known benchmark data sets within the machine learning community. The results on these data sets can be used to compare our model with results from other literature. The three *Reaching* data sets are internal to the "LiraLab" and have been generated using the humanoid robot James. These data sets have been included to see how well our method perform in the context of a humanoid robot. Unfortunately, there is no data available to compare our method with. The exact data sets that we have used are:

Diabetes The *Pima Indians Diabetes* data set investigates whether a patient shows signs of diabetes. All

Table 7.1: Basic statistics of the six data sets that have been used in the experiments.

Name	**Type**	**# Samples**	**# Features**	**Balance**	**Source**
Diabetes	classification	768	8	$65.1\%/34.9\%$	UCI Repository [2]
Housing	regression	506	13	n/a	UCI Repository [2]
Reaching 1	regression	1126	4	n/a	Internal
Reaching 2	regression	2534	4	n/a	Internal
Reaching 3	regression	2737	4	n/a	Internal
Wisconsin	classification	449	9	$52.6\%/47.4\%$	UCI Repository [2]

subjects in the data set were females of Pima Indian heritage. The data set includes features such as physiological properties (e.g. diastolic blood pressure), body mass index, and age.

Housing The *Boston Housing* data set concerns housing values in the suburbs of Boston. The features include properties such as crime rate, accessibility to infrastructure, and pollution rates.

Reaching 1,2,3 The *Reaching* data sets concern orienting the head of a humanoid robot in the direction of its reaching arm. The features are the values of 4 arm encoders, whereas the outputs are 3 head joints. This means that there are three separate regression problems. This data set is obtained internally within the "LiraLab".

Wisconsin The *Breast Cancer Wisconsin* data set concerns whether a tumor is benign or malignant. The features include features concerning the tumor, such as uniformity of its shape, size, and so forth.

Basic information and statistics on these data sets is shown in Table 7.1. One can observe that all of the data sets are moderately sized, with respect to machine learning standards. This means that they contain between several hundreds to a few thousands samples each. For our problem this size range is particularly interesting. Smaller data sets are often not very stable, since they do not contain enough information for the machine to recognize patterns in a reliable manner[1]. On the other hand, very large data sets are computationally infeasible. This can be contributed to both the complexity of LS-SVM as well as Evolutionary Computation.

7.1.4 Preprocessing

An important step in applying a machine learning technique to a problem is preprocessing the data sets. In preprocessing the data is reformatted in such a way that the learning algorithm will work better on it. For our specific experiments, we have performed the following preprocessing steps:

1. All the features are (independently) standardized, i.e. rescaling them to have a zero mean and a unit standard deviation. In many machine learning techniques normalization or standardization is essential for obtaining good results. The positive effect of rescaling the features is that all values will fall within a limited range, on which functions (e.g. the Sigmoid activation function in Artificial Neural Networks or the RBF kernel function in Kernel Machines) can operate well. Furthermore, a feature with a relatively large range cannot dominate the other features anymore. The advantage of standardization over normalization is that it is less sensitive to outliers.

[1]Recall from Section 2.1 that the empirical risk functional approaches the actual risk functional as the size of the data set increases.

2. The output values have been standardized in the same manner as the features. However, this has no influence on the quality of the solution whatsoever. The only reason for standardizing the output values is for the purpose of comparing. Since the errors will be measured relatively to the range of the output values, standardizing them will allow us to compare the results of all data set. Obviously, the output values of the classification problems have not been standardized. Instead, when necessary, they have been set to $+1$ for positive labels and -1 for negative labels.

3. Duplicate entries have been removed from the data sets. This reduces the size of the data set by removing redundant information. Furthermore, the samples with missing values have been removed from the data sets as well.

4. The order of the samples in the data set has been randomized. Because of this, each of the k folds will approximate the distribution of the complete data set.

7.2 Ordinary Grid Search Optimization

The first phase of our experiments consists of optimizing the hyperparameters with the traditional grid search method. The results of this phase will later form a baseline for our two models. Grid search is nothing more than a search on a specified set of possible parameter settings. Usually, this set is defined by means of a lower boundary, an upper boundary, and an interval. The parameter settings that lie on the intersections of this grid are tested. An obvious drawback of this method is that it scales exponentially with the number of parameters.

We have defined the lower and upper boundaries and the interval manually for each of the data sets. This was done by first performing a sparse search on a large range. These results were used to identify an interesting region in the parameter space, on which we consecutively performed a dense grid search. The results of this dense grid search will be presented in this section. For SVM and LS-SVM it is common to use a logarithmic scale for most of the hyperparameters [15]. The sizes of the interval have been chosen manually as well, where our main concern was the computational load. In other words, for smaller datasets and fewer number of features we preferred a smaller interval and thus more evaluations. The ranges and intervals for each combination of data set and kernel function is shown in Table 7.2. Note that we have kept the degree of the polynomial kernel fixed at $d = 3$.

7.2.1 Results of Grid Search

The results of the grid search optimization are shown in Table 7.3. We can observe that the results on the Diabetes and Housing data sets are rather poor, although this is in line with the results published in related work [104, 32, 12, 46]. Therefore, it is safe to assume that the data sets are very difficult to learn, which could be caused by noise in the samples or a very complex underlying model. The results on the other data sets are, on the other hand, very good. The MSE on the three Reaching data sets is very low, which confirms our belief that Kernel Machines can be a valuable tool in robotics.

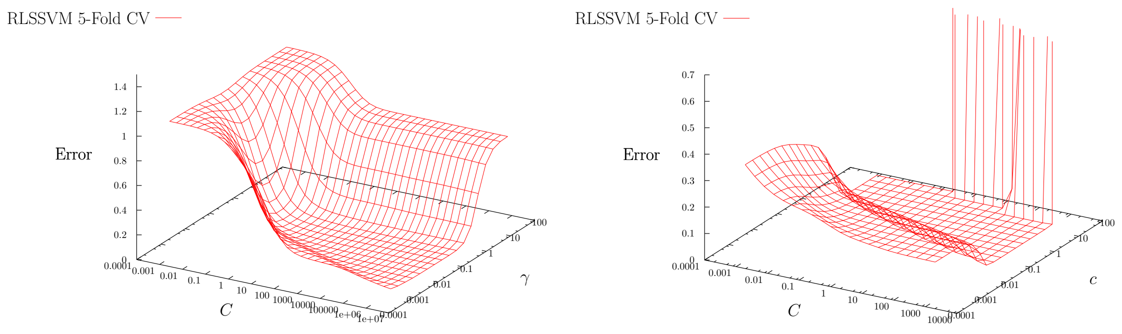
Furthermore, note that both kernel functions have vastly different optimal values for C. This clearly shows that for every change in the machine (e.g. changing kernel function), *all the hyperparameters* will need to be optimized again. We can confirm this statement by considering the error surface of both the kernel functions. As can be seen in Fig. 7.1, these error surfaces are considerably different. The surface of the

Table 7.2: The configuration of our experiments using the grid search method. Note that the ranges are in logarithmic scale. For example, $[-6, 20, 1]$ should be read as a range of $[2^{-6}, 2^{20}]$, where every step the value is increased by a factor of 2^1. The specified kernel parameter is γ in case of the RBF kernel and the constant c in case of the polynomial kernel.

Name	Kernel	Range C	Range γ/c	Evaluations
Diabetes	RBF	[-6, 20, 1]	[-16, 6, 1]	621
	Polynomial 3	[-20, 16, 1]	[-10, 10, 1]	777
Housing	RBF	[-10, 14, 0.5]	[-16, 12, 0.5]	2793
	Polynomial 3	[-21, 2, 0.5]	[-4, 14, 0.5]	1628
Reaching 1	RBF	[-12, 20, 0.5]	[-10, 14, 0.5]	3185
	Polynomial 3	[-10, 10, 0.5]	[-12, 6, 0.5]	1517
Reaching 2	RBF	[-12, 20, 1]	[-10, 6, 1]	561
	Polynomial 3	[-10, 10, 1]	[-12, 6, 1]	399
Reaching 3	RBF	[-12, 20, 1]	[-10, 6, 1]	561
	Polynomial 3	[-10, 10, 1]	[-12, 6, 1]	399
Wisconsin	RBF	[-12, 20, 0.5]	[-10, 14, 0.5]	3185
	Polynomial 3	[-14, 14, 0.5]	[-14, 6, 0.5]	2337

Table 7.3: The optimal solutions and errors as found by the grid search method.

Name	Kernel	ϵ_{min}	C_{min}	γ_{min}/c_{min}
Diabetes	RBF	0.2200	2^{14}	2^{-9}
	Polynomial 3	0.2213	2^{-20}	2^{8}
Housing	RBF	0.1676	2^{14}	2^{-6}
	Polynomial 3	0.1673	2^{-21}	$2^{11.5}$
Reaching 1	RBF	0.0683	2^{20}	2^{-3}
	Polynomial 3	0.0720	2^{-1}	2^{6}
Reaching 2	RBF	0.0042	2^{20}	2^{-2}
	Polynomial 3	0.0063	2^{-6}	2^{5}
Reaching 3	RBF	0.0019	2^{20}	2^{-2}
	Polynomial 3	0.0032	2^{-3}	2^{5}
Wisconsin	RBF	0.0423	2^{3}	2^{-6}
	Polynomial 3	0.0401	2^{-11}	2^{-5}



(a) RBF (b) Polynomial 3

Figure 7.1: The surfaces of the 5-fold cross validation error on the Reaching 2 data set with (a) the polynomial kernel and (b) the RBF kernel function. The extreme values with the polynomial kernel are caused by numerical difficulties.

polynomial kernel seems considerably more smooth, as compared to the RBF kernel. The RBF kernel is commonly preferred over the polynomial kernel, since it performs slightly better in general (cf. Table 7.3). However, the surface of the polynomial kernel is more flat, which means that a short optimization run is more likely to yield reasonable results. For some situations one could argue that the polynomial kernel is therefore the better option. These observations are shared among all the data sets in our study.

7.3 Hyperparameter Optimization using Evolution Strategies

After having determined the reference figures using grid search, we have performed experimental phase with EvoKM$^{\text{ES}}$. As mentioned before, in this phase we investigate whether ES can find good solutions in less time than the traditional grid search. The setup of the experiments is largely similar to the grid search. Mind, however, the following specific settings:

- We have used (μ, λ)-ES instead of $(\mu + \lambda)$-ES, based on empirical observations made in related literature. We have given arguments for this decision in Section 5.1.3.
- The parent and offspring population sizes have been kept relatively low, as is common with ES [118]. The size of the parent population is chosen as $\mu = 3$, whereas the offspring population is chosen $\lambda = 12$.
- The strategy parameters σ_i are initiated to a value of 1. During the course of the search process they are subject to mutation, although a minimum value of 0.1 will be enforced. This lower bound ensures that the search will not stall because the strategy parameters are near zero.
- The termination condition is set to the number of evaluations that were used in the grid search. Consult the last column of Table 7.2 for the exact number of evaluations for each combination of data set and kernel function.
- The parameters are restricted to the same ranges as for the grid search (cf. Table 7.2). The only exception is that for this experiment we have *not* kept the degree of the polynomial kernel function fixed at $d = 3$. Instead, for EvoKM$^{\text{ES}}$ this degree is a parameter with a range of $[1, 8]$ for all data sets.

The reason for this deviation with grid search is that we can obtain information about the scalability of EvoKM^{ES} with respect to the number of object parameters.

- EvoKM^{ES} is a stochastic search process, which is why we have performed the experiments 25 times. This is to ensure that we obtain statistically significant results.

The performance of EvoKM^{ES} will be analyzed at the hand of two measures. Firstly, we will have a look at the minimum error that is found by EvoKM^{ES}, as compared with the grid search. Major improvements should not be expected here, since the search space for both methods is identical – with exception of the polynomial kernel. Instead, the minimum error needs to be considered to verify whether EvoKM^{ES} actually finds good solutions.

The second – more important – aspect of this experiment is to see how many evaluations EvoKM^{ES} needs to come up with a good solution. This will be analyzed by measuring how many evaluations will be needed to reach the "target" solution. Obviously, this target is the best solution as found by the grid search method (cf. Table 7.3). Furthermore, we will consider a second variant, which will measure the number of evaluations that are needed to achieve a solution that is at most 5% worse than the target. This less strict variant gives us more insight in how fast a "reasonable" solution can be reached. One can imagine that there are various settings in which it is useful to quickly approximate an optimal solution.

An important note with this experiment is that we compare the temporal aspect solely in terms of number of evaluations. This simplification is valid, since the model selection in LS-SVM is independent of the parameters. As a result, the experiments could be executed on computers with varying configurations (e.g. CPU, memory, OS, and compiler).

7.3.1 Results of EvoKM^{ES}

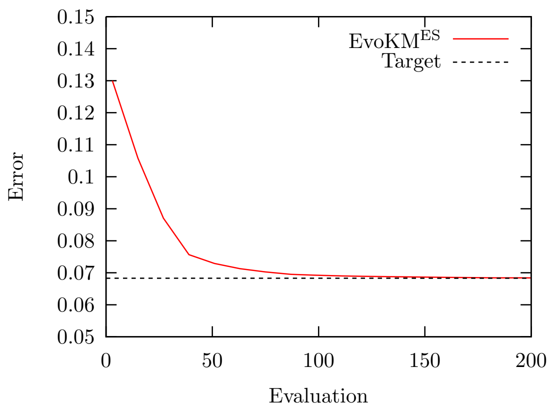
The comparison between grid search and our EvoKM^{ES} is shown in Table 7.4. Note that *ETT* stands for *Evaluations to Target*, which is the quantitative measure described above. From this table we can observe that our EvoKM^{ES} method performs very well on the three Reaching data sets. Only a fraction of the evaluations were needed to find solutions that were at least equally as good as the target. In Fig. 7.2 we can see clearly that EvoKM^{ES} converges rapidly to the target on the Reaching 1 data set. Furthermore, we also see that the polynomial kernel converges faster than the RBF kernel. We contribute this to the fact that the error surface of the polynomial kernel was more smooth. Hence, the search is more likely to find a low-error region in less evaluations.

However, the results on the Diabetes, Housing, and Wisconsin data sets is less positive. EvoKM^{ES} performs poorly on the Housing data set with the polynomial kernel due to an outlier in the results (cf. the corresponding variance in Table 7.4). This is one of the problems when stochastic methods such as ES are applied, namely, the unpredictability of the results. Possibly, the likeliness that this happens could be reduced by experimenting with different exogenous strategy parameters. In particular, increasing the population sizes μ and λ can reduce this probability of outliers. This will, however, have a negative impact on the convergence rate with respect to the number of evaluations.

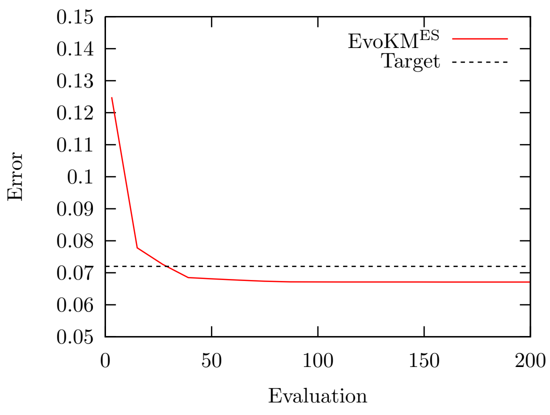
The poor results on the Diabetes and Wisconsin data sets have a completely different cause. The convergence of EvoKM^{ES} with the Diabetes data set is depicted in Fig. 7.3. In particular the combination of EvoKM^{ES} with the RBF kernel function hints us at the problem. We can observe in Fig. 7.3(a) that the

Table 7.4: Comparison between the grid search method and EvoKM^{ES}.

		Grid Search		*EvoKM^{ES}*		
Name	**Kernel**	ϵ_{min}	**Eval.**	$\bar{\epsilon}_{min}$	**ETT$_{0\%}$**	**ETT$_{5\%}$**
Diabetes	RBF	0.2200	621	$0.2238 \pm \mathit{0.0026}$	>621	183
	Polynomial 3	0.2213	777	$0.2228 \pm \mathit{0.0005}$	>777	3
Housing	RBF	0.1676	2793	$0.1674 \pm \mathit{0.0000}$	495	291
	Polynomial 3	0.1673	1739	$0.1714 \pm \mathit{0.0237}$	>1739	963
Reaching 1	RBF	0.0683	3185	$0.0683 \pm \mathit{0.0000}$	327	63
	Polynomial 3	0.0720	1517	$0.0671 \pm \mathit{0.0001}$	39	27
Reaching 2	RBF	0.0042	561	$0.0042 \pm \mathit{0.0000}$	159	111
	Polynomial 3	0.0063	399	$0.0044 \pm \mathit{0.0000}$	27	27
Reaching 3	RBF	0.0019	561	$0.0019 \pm \mathit{0.0000}$	135	87
	Polynomial 3	0.0032	399	$0.0022 \pm \mathit{0.0001}$	39	39
Wisconsin	RBF	0.0423	3185	$0.0454 \pm \mathit{0.0022}$	>3185	>3185
	Polynomial 3	0.0401	2337	$0.0424 \pm \mathit{0.0024}$	>2337	>2337

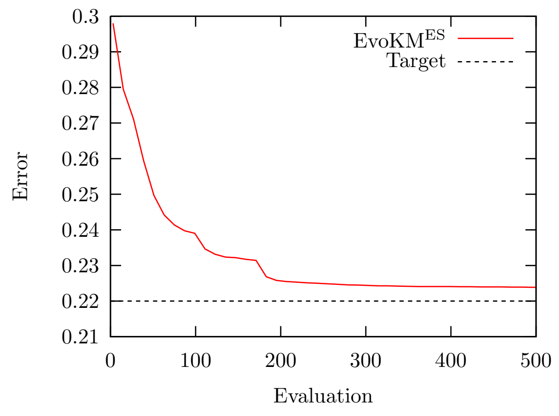


(a) RBF

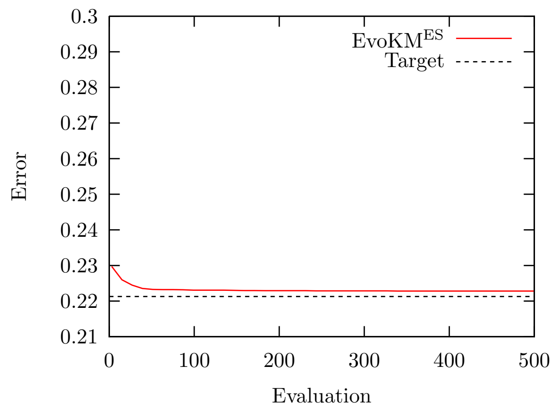


(b) Polynomial

Figure 7.2: The convergence of the minimum error during the EvoKM^{ES} search for the Reaching 1 data set with (a) the RBF kernel function and (b) the polynomial kernel function.



(a) RBF



(b) Polynomial

Figure 7.3: The convergence of the minimum error during the EvoKM$^{\text{ES}}$ search for the Diabetes data set with (a) the RBF kernel function and (b) the polynomial kernel function. Note that the target error is not reached.

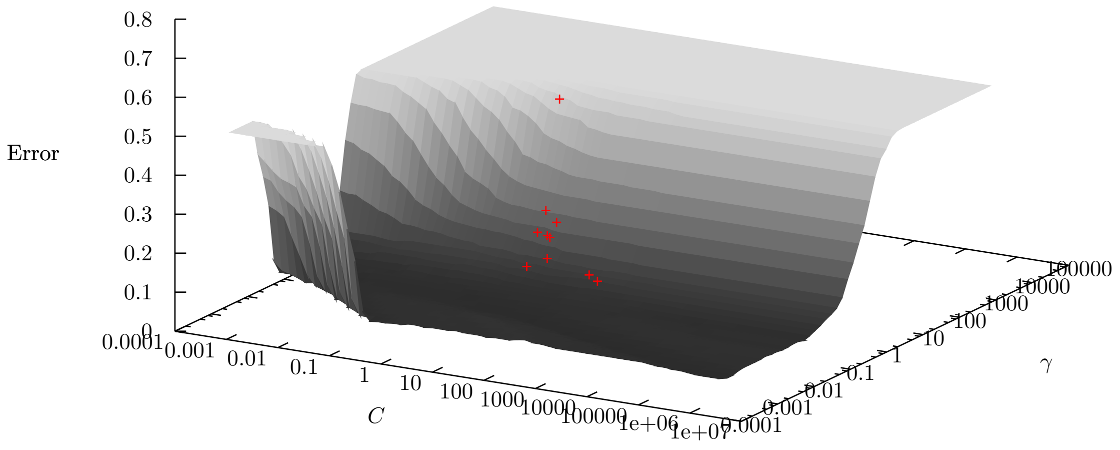
convergence is not completely smooth, but rather stepwise[2]. This is caused by the fact that we consider a classification problem, for which the error surface is not continuous. Instead, the error surface for classification problems is made up out of multiple plateaus[3]. This discontinuity of the error surface makes it very hard for EvoKM$^{\text{ES}}$to find a direction in which the error decreases. In ES the offspring individuals are used to sample the error surface in a certain neighborhood. However, many offspring individuals will have an identical fitness score as the parent, which means that the search process does not receive any “clues” about the shape of the error surface. Smoothness of the fitness landscape may be regarded a prerequisite of efficient optimization using ES [5].

The problem is probably best explained by illustrating the sampling of the individuals. In Fig. 7.4 we have depicted the first 10, 50, and 100 individuals that are evolved by EvoKM$^{\text{ES}}$ on the Wisconsin data set with the RBF kernel function. These individuals – more precisely, the object parameters in their chromosome – are superimposed on the corresponding error surface. In this particular example the search process finds a slope and follows the direction toward a low-error region. However, Fig. 7.5 shows a particular run of EvoKM$^{\text{ES}}$ on the Wisconsin data set where the individuals all remain on the same plateau. This figure clearly demonstrates how EvoKM$^{\text{ES}}$ samples the neighborhood, but does not find any improvements in the offspring. Hence, the search process stalls until an offspring individual will find a region where the error is lower.

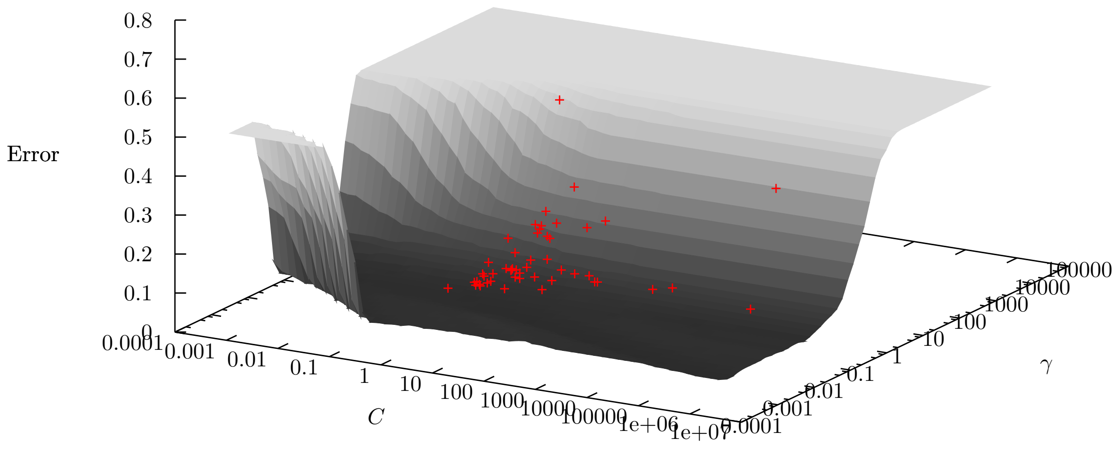
A second reason that explains the poor results on these two data sets is the distribution of the regions with a near-optimal error. We have juxtaposed the error surface of the Reaching 3 and the Wisconsin data sets in Fig. 7.6. These two figures show that the region that yields very low errors is relatively large for the Reaching 3 data set, as compared with the Wisconsin data set. From this we can conclude that finding a good set of hyperparameters for the Reaching 3 data set is considerably more easy than for the Wisconsin data set. The error surface of the Diabetes data set is similar to the error surface of the Wisconsin data set.

[2]Mind that this convergence is the average over 25 runs and therefore already more smooth than a single run.

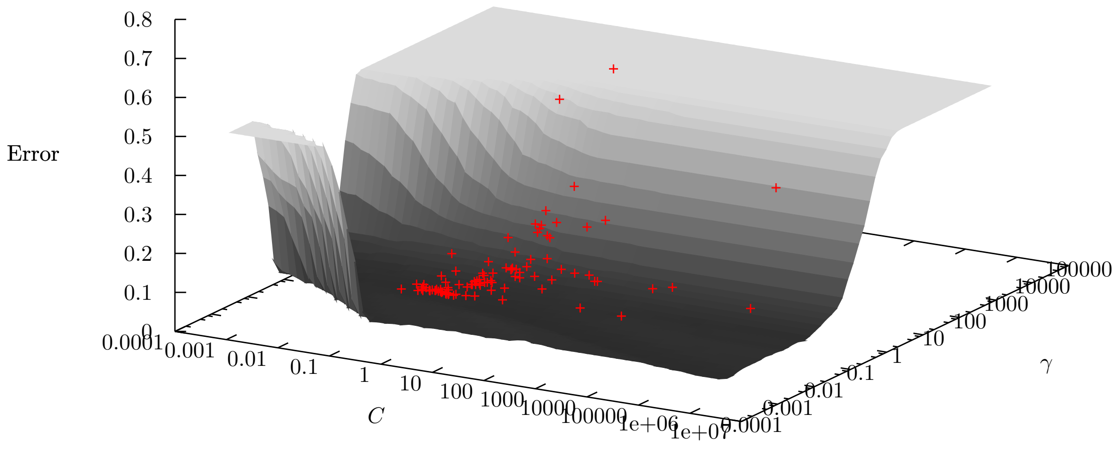
[3]This can very easily be verified by considering a simplified problem. Consider the classification error on a test set of only 4 samples. In this situation the error can only be one of the values $\{0, 0.25, 0.5, 0.75, 1\}$.



(a) 10 individuals

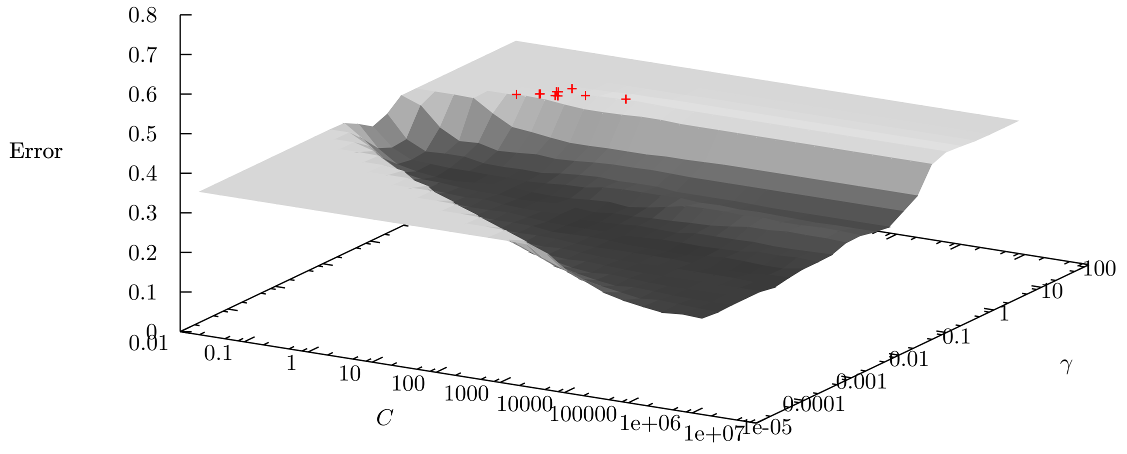


(b) 50 individuals

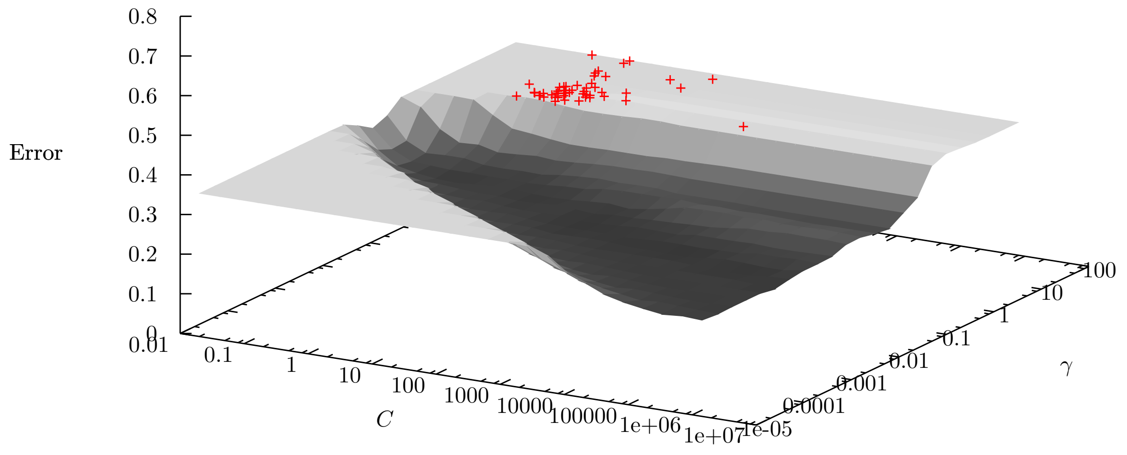


(c) 100 individuals

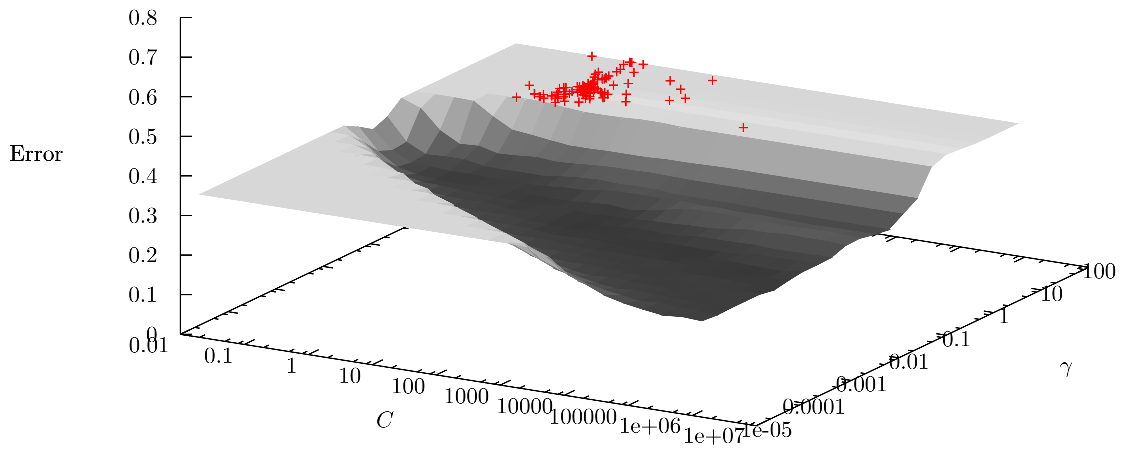
Figure 7.4: Example of a run by EvoKM^{ES} on the Wisconsin data set with the RBF kernel function. The figures show the first (a) 10, (b) 50, and (c) 100 individuals marked in red.



(a) 10 individuals

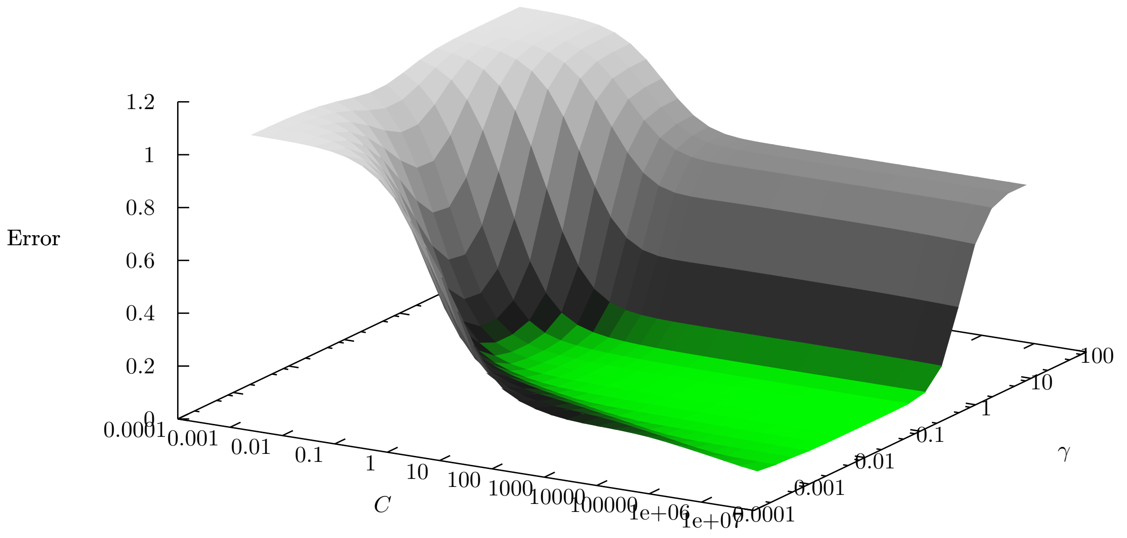


(b) 50 individuals

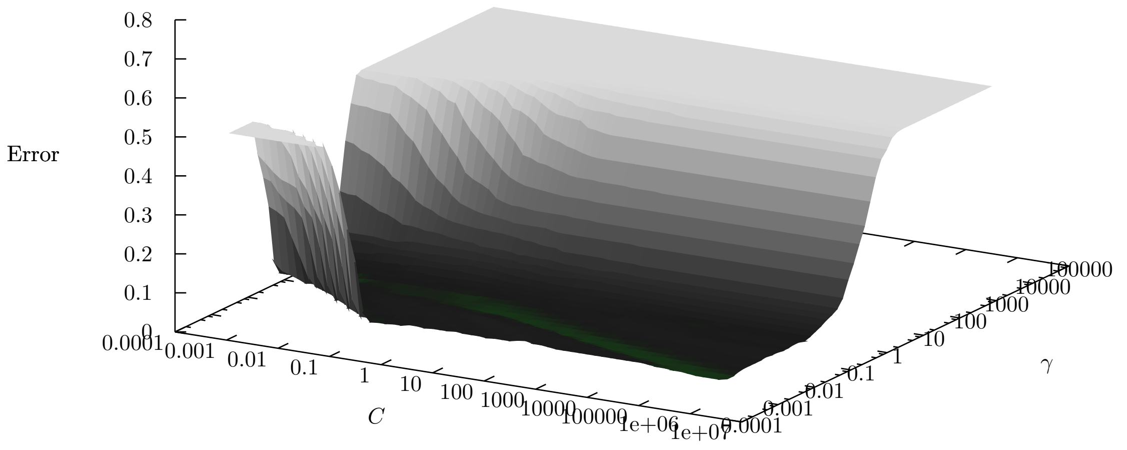


(c) 100 individuals

Figure 7.5: The first (a) 10, (b) 50, and (c) 100 individuals generated by EvoKM$^{\text{ES}}$ for a particular run on the Diabetes data set with the RBF kernel function. The search gets stuck on a plateau, where all individuals have an equal fitness.



(a) Reaching 3



(b) Wisconsin

Figure 7.6: The distribution of the regions with a near-optimal error for the (a) Reaching 3 and (b) Wisconsin data sets. The regions that have a near-optimal error are marked in green.

7.3.2 Conclusions

In this section we presented the results of our empirical study on the EvoKM$^{\text{ES}}$ optimization method. In general, our method was able to find good solutions in only a fraction of the evaluations that were dedicated to the grid search. There are some remarks that need to be considered. Firstly, EvoKM$^{\text{ES}}$ performed rather poorly on the two classification data sets. This can be explained by the fact that ES has difficulties with dealing with plateaus in the fitness landscape. Classification problems, by nature, have a discontinuous error surface that is full of these plateaus. Secondly, the results depend as well on the distribution of the near-optimal error regions. The two classification data sets that were used in this study have only a very small region that yields near-optimal results. Obviously, for any empirical search method it is difficult to find small optimal regions.

Moreover, we must be very careful with our interpretation of these results. ES basically is a stochastic search method, which means that every run will be different. We have seen that this can cause outliers to occur, which in a practical application of the technique is not desirable. More research will have to be dedicated on finding parameters for EvoKM$^{\text{ES}}$ that minimize the probability that this will happen. Furthermore, we must realize that the comparison is not completely fair. EvoKM$^{\text{ES}}$ is an optimization method that – most probably – converges to optimal solutions. This means that the solution will continue to improve as the process runs for more generations. Grid search, on the other hand, does not converge to a solution, it simply finds it by "luck". This means that when 1000 evaluations are dedicated to grid search, the optimal solutions could well be found just after 10 evaluations. It is therefore very hard to make scientifically correct statements on which method is guaranteed to be better. However, in the practical application the important aspect is on which method to spend your "CPU cycles". We believe that the results presented in this section indicate that EvoKM$^{\text{ES}}$ is without a doubt the better choice for regression problems. For classification problems the situation is less obvious. An approximate solution can still be found quickly with EvoKM$^{\text{ES}}$, but we recommend using grid search if the quality of the solution is of great importance.

7.4 Evolved Kernel Functions using Genetic Programming

The last experimental phase in this study constitutes of verifying our EvoKM$^{\text{GP}}$ model. The errors obtained with our GP method will be compared with the minimum error as found using grid search. The number of evaluations are irrelevant in this experiment; the only aspect that we will consider is the quality of the solutions.

One of the difficulties with any GP approach is the wide variety of possible configurations. The configuration includes the exact order of the operators (i.e. the evolver model) and the parameters. Unfortunately, there is not much guidance on optimal parameter settings, other than certain heuristics and empirical evidence. Therefore, the configuration that we have used for EvoKM$^{\text{GP}}$ is mostly done on base of intuition. The evolver model that we used for these experiments is shown in Fig. 7.7[4]. Note that there are in total 9 possible operators that can be applied to an individual – or two individuals in case of crossover. The probabilities are chosen in such a way that we emphasize on the crossover and pdf operators (cf. Section 5.2.3). Our reasoning for this decision is that crossover may be useful for the individuals to exchange a kernel function. The crossover operator will exchange a non-terminal with a probability of 0.99, which means that a terminal will be subject to crossover with a probability of only 0.01. In our specific model this

[4]An example of a configuration file that has been used during the experiments is shown in Fig. B.2.

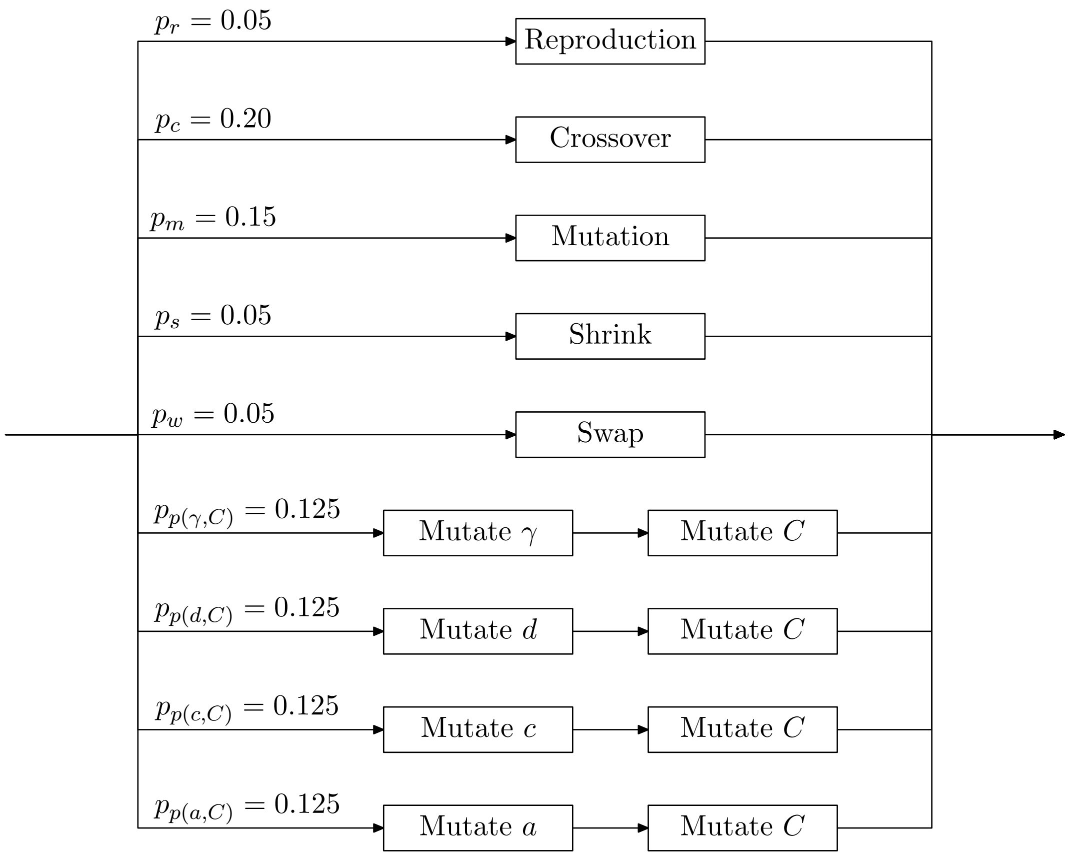


Figure 7.7: The evolver model used for the experiments with EvoKM$^{\text{GP}}$.

means that crossover will most likely exchange a kernel function[5]. We strongly emphasize on exchanging non-terminals, as exchanging a terminal, i.e. parameter, will not be likely to improve the kernel function as a whole.

Defining the structure of a kernel is only the first phase in evolving kernel functions. The second phase is the subproblem of optimizing the hyperparameters. As we could observe from the error surfaces above, the generalization performance – and therefore the fitness – of a kernel function depends strongly on the chosen hyperparameters. We have emphasized on the pdf operator as to deal with this problem. Our approach is to not mutate all the hyperparamers in one operator, as this could lead to too strong mutations on the individuals. Instead, we have created four separate operator sequences, each of which mutate one of the hyperparameters $\{\gamma, d, c, a\}$ *and* the hyperparameter C. This operator does not verify whether one of the former hyperparameters actually occurs in the chromosome. If this is the case, then only the hyperparameter C is mutated.

Furthermore, we have chosen the following settings for our experiments:

- The population size for our experiments was 2000. We have chosen this relatively large, as is common with both GA and GP for non-trivial problems [118].
- The *steady-state* variant of GP has been used, in accordance with suggestions based on empirical

[5]A kernel function in this context is the non-terminal *kernel*, as specified in the context-free grammar from Fig. 5.2. Note that this definition is more broad than just kernel functions from Section 2.3.1.

evidence [118].

- The selection method is *tournament selection*, where the tournament size is 2 [118]. The selection method makes use of parsimony pressure, so that in case of equal fitness the smallest tree wins the tournament.

- The search process is terminated when 12 *generations* have been reached, *excluding* the initial bootstrap generation. This maximum number of generations is relatively low, which is common in both GA and GP with large population sizes.

- All the terminals and non-terminals are chosen with the same probability. In other words, there is *no bias* toward certain primitives.

- The trees for the individuals are initially generated using the *grow method* with a minimum depth of 2 and a maximum depth of 6. The maximum depth of the trees during the course of the GP process is 9.

- The parameters have been limited within a certain range. For the regularization parameter C this range is $\left[2^{-20}, 2^{20}\right]$. The range for the parameter γ for the RBF kernel is set to $\left[2^{-12}, 2^{12}\right]$. The parameters of the polynomial kernel are restricted within the range $[1, 7]$ for the degree d and $\left[2^{-12}, 2^{12}\right]$ for the constant c. Furthermore, the weight parameter a has a range of $[0, 20]$. Note that all parameters have a logarithmic scale, except for the degree d and the weight a. These ranges have been defined based on results with preliminary runs.

- Every experiment has been conducted 10 times, as to compensate for the stochastic behavior of the algorithm. More runs would be desirable to increase the reliability of our results. Unfortunately, this was computationally not tractable in this study.

7.4.1 Results of EvoKM$^{\text{GP}}$

In Table 7.5 we can observe the results of EvoKM$^{\text{GP}}$ on the six data sets. The table shows the average minimum error $\bar{\epsilon}_{min}$ of the 10 runs and the absolute minimum ϵ_{min}. The data shows that the EvoKM$^{\text{GP}}$ approach outperforms grid search in terms of quality of the solutions for each data set. The difference between both methods, however, is only marginal. This means that the method is not improving much as compared with grid search, although at a much higher computational expense.

Another problem that we can observe, is the high heterogeneity of the best solutions found in every run. Some runs result in very complex kernels, such as the kernel function shown in Fig. 7.8. Other runs on the same data set have one of the atomic kernel functions as the optimal result. Nonetheless, the variance of the minimum error is still reasonable. We must not rule out that there may simply exist multiple kernel functions that perform similarly on a data set.

Furthermore, as we noted before, optimizing a kernel function – including its hyperparameters – is far from a trivial problem. It could well be that a different configuration (e.g. the population size and the evolver model) could improve on these results. Another observation that we wish to notify is that for some data sets the error is indeed significantly lower. Consider for instance the Wisconsin data, where the classification error decreased from approximately 4% for grid search to on average 3.6% for EvoKM$^{\text{GP}}$. For instance, in medical settings such a decrease in the classification error could be very desirable.

Table 7.5: The minimum errors as obtained with EvoKM^{GP}. Note that there is only a minor improvement compared with errors obtained using grid search.

	Grid Search	*EvoKM^{GP}*	
Name	ϵ_{min}	$\bar{\epsilon}_{min}$	ϵ_{min}
Diabetes	0.2200	0.2176 ± *0.0032*	0.2096
Housing	0.1673	0.1633 ± *0.0006*	0.1620
Reaching 1	0.0683	0.0592 ± *0.0004*	0.0587
Reaching 2	0.0042	0.0038 ± *0.0000*	0.0037
Reaching 3	0.0019	0.0018 ± *0.0000*	0.0018
Wisconsin	0.0401	0.0358 ± *0.0012*	0.0333

7.4.2 Conclusions

The results for EvoKM^{GP} show that more complex kernel functions can improve on the generalization performance of a Kernel Machine. However, the improvement is for all data sets only marginal and one seriously has to doubt whether this difference is statistically significant. Only in very specific situations, e.g. medical diagnosis, the improvement may outweigh the high computational expense of evolving the kernel function using GP.

One main problem with the method is that if finds heterogeneous optimal solutions. In other words, a particular run of EvoKM^{GP} on a particular data set may yield a drastically different kernel function compared to a previous run. Similar observations were reported with regard to evolutionary hyperparameter optimization [96]. We have to realize, however, that in a sense this is the nature of evolutionary optimization methods such as GA and GP. Instead of finding optimal solutions, they tend to find "good" solutions. Obviously, for any problem there may exist only one optimal solution, but there will surely exist many good solutions.

Another difficulty that we experienced in this study is finding a good configuration for our GP method. There are many parameters that need to be set and one has to find a suitable evolver model. Unfortunately, there is not a structured approach of finding a good configuration. Therefore, it remains mostly a task that has to be solved using loose heuristics or even simply guessing. This problem is especially evident in our context, as the computational demand does not allow for an empirical verification of multiple possible configurations[6].

[6]To give an impression, the experiments of EvoKM^{GP}, as presented in this section, need more than half a year of CPU time on a Pentium 4 class computer running at 3 GHz.

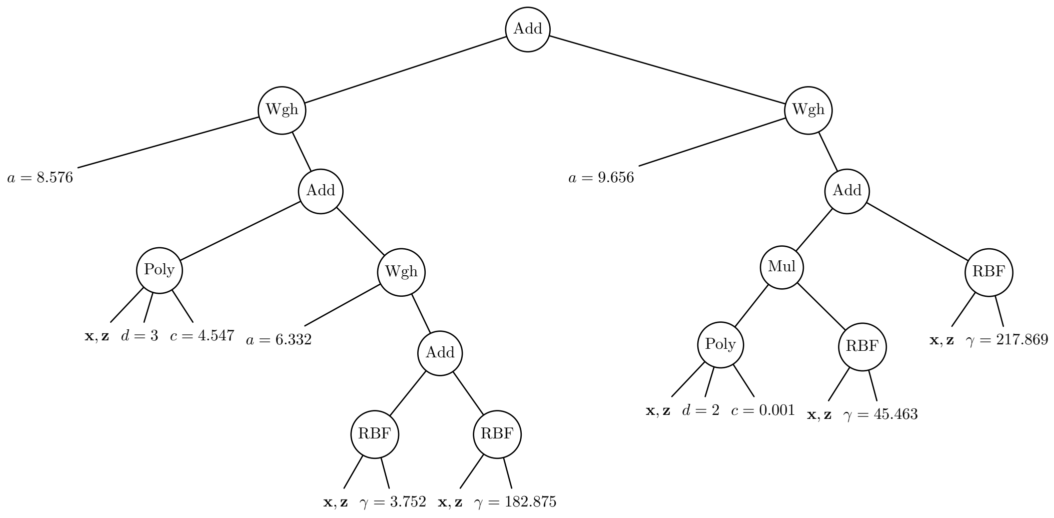

Add
Wgh
Wgh
$a = 8.576$
Add
$a = 9.656$
Add
Poly
Wgh
Mul
RBF
$\mathbf{x}, \mathbf{z}$ $d = 3$ $c = 4.547$
$a = 6.332$
Add
Poly
RBF
$\mathbf{x}, \mathbf{z}$ $\gamma = 217.869$
$\mathbf{x}, \mathbf{z}$ $d = 2$ $c = 0.001$
$\mathbf{x}, \mathbf{z}$ $\gamma = 45.463$
RBF
RBF
$\mathbf{x}, \mathbf{z}$ $\gamma = 3.752$
$\mathbf{x}, \mathbf{z}$ $\gamma = 182.875$

Figure 7.8: An example of a good kernel function for the Reaching 1 data set.

Chapter 8 Objective Functions for Kernel Machines

One of the main difficulties in hyperparameter selection is finding an appropriate objective function. As we argued before, the performance of a Kernel Machine is heavily dependent on a careful selection of an appropriate kernel function and good values for the hyperparameters. Obviously, this careful selection requires an objective function that is capable of estimating the true generalization capacity of our machine. An error criterion, or objective function, that does exactly this is hard to find. One of the main issues is the problem of *overfitting*. This occurs when the machine is starting to learn the specifics of the data on which it is trained, while reducing the generalization performance. An example of this problem would be a machine that starts to learn the noise that is present in the actual training data. In particular in our GP kernel evolvement method overfitting could be a serious problem, because we are not only tuning the hyperparameters, but the kernel function as well. One could argue that this gives a *double* risk on overfitting. Furthermore, we generally impose certain requirements on the objective function. These requirements may include:

- The most exact approximation of the true generalization error.
- Good scalability, i.e. computational tractable also on large data sets.
- Good performance in situations where a limited amount of data is available for training and validation.
- Applicability for different types of machine learning techniques.

It may be clear that these requirements are potentially conflicting. Therefore, one has to prioritize certain requirements over others. In this chapter we will describe the specific details regarding our decision on an appropriate objective function for our study. We will start with a theoretical base on generalization errors and the estimation thereof, after which some commonly used error criteria will be presented. Then we will cover an interesting potential candidate, which is Kernel Alignment, with an argumentation of its qualities. Lastly, an experimental validation is presented to verify whether this objective function indeed met our expectations.

8.1 Theory on Generalization Errors

In Section 2.1 we explained the difference between the *actual* model and the model that our machine creates based on *empirical* observations. As a short recapitulation, the actual model is defined as $P(x, y) = P(\mathbf{x})P(y|\mathbf{x})$ and the empirical model of our machine as $f(\mathbf{x}, \boldsymbol{\alpha}) = \hat{y}$. When considering the generalization error, it is important to note the difference between these two models. The more identical the models are, the lower our generalization error will be. The main problem is that we need an estimator that reliably measures this generalization error.

Let us assume that the machine has learned a model based on a finite amount of samples, i.e. $S = (\mathbf{x}_1, y_i), \ldots, (\mathbf{x}_\ell, y_\ell)$. Then we could describe the estimated error for a sample $\mathbf{x}$ as [42]:

$$\begin{aligned}\mathcal{E}(\mathbf{x}, \boldsymbol{\alpha}; S) &= E\left[(y - f(\mathbf{x}, \boldsymbol{\alpha}; S))^2 \,|\mathbf{x}, S\right] \\ &= E\left[(y - E[y|\mathbf{x}])^2 \,|\mathbf{x}, S\right] + (f(\mathbf{x}, \boldsymbol{\alpha}; S) - E[y|\mathbf{x}])^2 \quad ,\end{aligned} \tag{8.1}$$

where we included the S to emphasize that these terms depend on the training data. Furthermore, $E[\cdot]$ is taken with respect to the unknown probability distribution $P(y|\mathbf{x})$[1].

The term $E\left[(y - E[y|\mathbf{x}])^2 \,|\mathbf{x}, S\right]$ can be explained as the residual between the actual output value y and the most likely output value according to the probability distribution returned by the supervisor. This error is the variance of $P(y|\mathbf{x})$ and therefore does *not* depend on the training set S. The only relevant term, from the perspective of our machine, is the second term, i.e. $(f(\mathbf{x}, \boldsymbol{\alpha}; S) - E[y|\mathbf{x}])^2$. This squared distance measures the residual between the output of our machine and the estimated output of the supervisor. This term measures the effectiveness of our machine, so we would like to have this term as small as possible. Obviously, the expected error given a data set S is given by

$$\mathcal{E}(\mathbf{x}, \boldsymbol{\alpha}) = E_S\left[(f(\mathbf{x}, \boldsymbol{\alpha}; S) - E[y|\mathbf{x}])^2\right] \quad . \tag{8.2}$$

One can notice that this equation is highly related to the empirical error that we described in Eq. 2.4. The difference is that here we consider the expected output of the supervisor, instead of the actual output y. In other words, we measure the *expected* error, instead of the *empirical* error. The empirical error is a noisy variant of the expected error, since the expected output $E[y|\mathbf{x}]$ not always equals the empirical output y [69].

It is useful to decompose Eq. 8.2 into separate *bias* and *variance* terms [42], which yields

$$\begin{aligned}\mathcal{E}(\mathbf{x}, \boldsymbol{\alpha}) &= E_S\left[(f(\mathbf{x}, \boldsymbol{\alpha}; S) - E[y|\mathbf{x}])^2\right] \\ &= \underbrace{(E_S[f(\mathbf{x}, \boldsymbol{\alpha}; S)] - E[y|\mathbf{x}])^2}_{\text{bias}} + \underbrace{E_S\left[(f(\mathbf{x}, \boldsymbol{\alpha}; S) - E_S[f(\mathbf{x}, \boldsymbol{\alpha}; S)])^2\right]}_{\text{variance}} \quad .\end{aligned} \tag{8.3}$$

The squared bias term represents the degree to which the empirical error *systematically* differs from the expected error, i.e. $f(\mathbf{x}, \boldsymbol{\alpha}; S)$ is different from $E[y|\mathbf{x}]$, respectively. The variance term captures the sensitivity

[1]Mind that it is necessary to use the *expected* output value, as the supervisor gives an probability distribution for y, not directly a value.

of the machine to the data S on which it is being evaluated. When the size ℓ of the data set is increased, then the variance can be expected to decrease. Obviously, both the bias and the variance terms can contribute to poor performance. Often there is even a tradeoff between both terms. For instance, consider that we smoothen the data, as to reduce the variance. This will introduce a bias, since details of the original model will get lost (e.g. sharp peaks). This tradeoff is referred to as the *bias-variance dilemma*.

An important aspect for the objective function for our kernel machine is that it tries to minimize the true test error, not just the empirical test error. Minimizing the latter, i.e. the empirical test error, is very likely to lead to overfitting to the training data. Given the theory that we described in this section, we can reformulate this requirement. The objective function should be based on a method that guarantees an unbiased machine, combined with a low variance.

8.2 Feasible Objective Functions

In the related work described in Chapter 4 we saw many different kinds of objective functions, such as different types of cross validation and Kernel Alignment. All these methods might satisfy different requirements, from those we described in the beginning of this chapter. Therefore, it is important for us to consider these different aspects when deciding upon an objective function to use for our study. The requirements that we impose on the objective functions are – in order of priority – as follows:

1. A good approximation of the true generalization error, i.e. the *expected* error as described in Section 8.1.

2. Applicable to different types of Kernel Machines.

3. Good scalability with respect to the size of the data set.

Duan et al. have presented an extensive juxtaposition of various objective functions for SVM [30]. However, most of these objective functions are specifically targeted at SVM and thus not applicable for our study. Furthermore, the measure of Kernel Alignment has, unfortunately, been proposed after that the study was published. In this section we will describe the two measures that fit our requirements, i.e. k-fold and leave-one-out cross validation. Additionally, we present the Kernel Alignment measure and its interesting aspects.

8.2.1 Cross Validation

Probably the most used evaluation method for various types of machine learning techniques is cross validation [31]. In cross validation the data set is partitioned into two parts. The first of these two parts is used to train the machine, whereas the second is used to validate the performance. It is essential that the *validation subset* does not include any of the samples that were in the *training subset*. This method, i.e. *holdout validation*, reduces the risk of overfitting, as the machine is validated on different samples than it has been trained on. Nonetheless, a deficiency in this method is that it is highly depended on the partitioning of the data set.

k-Fold Cross Validation

A related form of cross validation is the so-called k-fold cross validation [31]. In this variant the training set S is subdivided in k disjoint subsets of equal size $\frac{\ell}{k}$, i.e. $S = \{S_1, \ldots, S_k\}$. Then the machine is trained on $k - 1$ subsets and validated on the single remaining subset. This procedure is repeated k times, where every time a different set is held out as the validation set. The estimated error using this method is defined as the mean of the k errors obtained during the iterations. This reduces the dependency on the partitioning of the data set, as the final error is the average error on k different partitions.

One of the main advantages of k-Fold cross validation is that it can be applied to virtually any kind of machine learning technique. For example, in the field of *Artificial Neural Networks* this validation method has successfully been used for several years. Also for training SVMs it has been shown to be one of the most reliable objective functions [30]. The method has been shown to practically be unbiased and to have a low variance [58]. The latter is a very desirable aspect, as it provides certainty that we have a good approximation for the expected error, based on the average of empirical errors.

Nonetheless, the method does have a disadvantage that may prevent it from being used in some settings. The machine has to be trained to measure the performance for a specific set of hyperparameters $\boldsymbol{\theta}$. Moreover, this has to be done k times, i.e. once for each fold. This can be computationally intractable in case of very large data sets. However, for data sets that are not that large the number of folds k gives a very basic form of regularization of the computational load. Typical values for k range from as low as 4 till around 20, of which $k = 10$ is commonly advised as a good tradeoff [58].

Leave-One-Out Cross Validation

Leave-one-out cross validation is an exception to this heuristic. In this specialized method k is chosen equal to the to the size of the data set ℓ. This is named leave-one-out cross validation, since every iteration only one validation sample is left out of the training set. It has been shown that also leave-one-out cross validation is approximately unbiased [68][2]. However, the variance is generally higher than is the case with k-fold cross validations [58].

Explicitly calculating the leave-one-out cross validation error is usually computationally infeasible for data sets of reasonable size. For certain algorithms, however, there exist some "shortcuts" to calculate the error more efficiently. An example is the fast leave-one-out calculation for LS-SVM that was described in Section 2.4.3. For the case of SVM, one would only need to conduct the leave-one-out procedure for the support vectors, since removing non support vectors would not affect the decision function. Furthermore, for some algorithms there exists algorithms to calculate a *bound* on the leave-one-out error (e.g. the radius-margin bound for certain types of SVM, see Section 4.1).

8.3 Kernel Alignment

Kernel Alignment uses a drastically different approach for approximating the generalization performance of the kernel function. In this method the generalization performance is estimated by measuring the similarity between the kernel matrix and a so-called target [22]. The target matrix, which is based on information from the data set, should be interpreted as an "ideal" kernel matrix. This approach makes the method very

[2]Approximately in the sense that the proof holds for $\ell - 1$ samples instead of ℓ.

different from cross validation, which predicts the generalization performance based solely on empirical errors.

The principal idea behind Kernel Alignment is to calculate the degree of agreement between two kernel matrices. Intuitively this makes sense, since the kernel matrix K captures information about both the kernel function and the training data. Moreover, the crucial element in Kernel Machines is the mapping of the data into a hypothetical feature space. If this mapping coincides with the problem at hand (e.g. makes classification data linearly separable), then we would expect the corresponding kernel function to match the data set quite well.

The similarity between two – normalized – kernel matrices in Kernel Alignment with respect to a training set S is defined as

$$A(S, \mathbf{K_1}, \mathbf{K_2}) = \frac{\langle \mathbf{K_1}, \mathbf{K_2} \rangle_F}{\sqrt{\langle \mathbf{K_1}, \mathbf{K_1} \rangle_F \langle \mathbf{K_2}, \mathbf{K_2} \rangle_F}} \ . \tag{8.4}$$

In Eq. 8.4 $\langle K_1, K_2 \rangle_F$ denotes the Frobenius inner product of matrices $\mathbf{K_1}$ and $\mathbf{K_2}$, which is defined as

$$\langle \mathbf{K_1}, \mathbf{K_2} \rangle_F = \sum_{i,j=1}^{m} k_1(x_i, x_j) k_2(x_i, x_j) \ . \tag{8.5}$$

This quantity is 1 for identical matrices and 0 for orthogonal ones. We can thus state that a higher Frobenius inner product between two matrices means that both are more aligned (i.e. similar) to each other.

However, in practical situations we wish to quantify the alignment between a kernel matrix – or kernel function – and the training data. This can be done if there is a way to transform the data set into a suitable kernel matrix. A kernel matrix can then be compared to this reference (i.e. an ideal kernel matrix). The proposed definition to create such a target matrix is $K_2 = \mathbf{y}\mathbf{y}^T$ [22]. The intuition behind this definition is that for samples with similar output y we would expect that their inner products are large. If we insert this ideal matrix in Eq. 8.4 we obtain the Kernel Target Alignment, i.e.

$$A(S, \mathbf{K}, \mathbf{y}\mathbf{y}^T) = \frac{\langle \mathbf{K}, \mathbf{y}\mathbf{y}^T \rangle_F}{\sqrt{\langle \mathbf{K}, \mathbf{K} \rangle_F \langle \mathbf{y}\mathbf{y}^T, \mathbf{y}\mathbf{y}^T \rangle_F}} = \frac{\mathbf{y}^T \mathbf{K} \mathbf{y}}{m||\mathbf{K}||_F} \ . \tag{8.6}$$

So far these definitions have assumed that we are dealing with a classification problem, i.e. $y_i \in \{-1, +1\}$. It can however easily be modified for the regression case [54, 53]. The only modification that is required to do this is

$$y_i = y_i - \bar{y} \ , \tag{8.7}$$

where $\bar{y}$ represents the mean over the output values of the data set.

Kernel Target Alignment has several advantages as a performance estimator. Firstly, it can be applied to virtually any type of Kernel Machine, due to its nature to estimate the performance based on the kernel matrix. It is desirable that a method gives us information about how well suited a kernel function is for a given data set, without making any assumption about the exact algorithm that is using this kernel function. Furthermore, the computational complexity of Kernel Target Alignment is $\mathcal{O}\left(n^2\right)$, which is relatively low as compared to actually training most forms of Kernel Machines. For example, the computational complexity of both LS-SVM and SVM is (approximately) $\mathcal{O}\left(n^3\right)$. This means that for relatively large data sets there is a considerable gain to be made.

An apparent disadvantage of this method is that there are no guarantees whatsoever about the relation with the generalization performance. There is some empirical evidence that optimizing the alignment increases the generalization performance [22], but this information is rather sparse. Furthermore, other studies have shown opposite results, i.e. that Kernel Target Alignment is not really practically useful [107, 79]. It is therefore difficult to estimate the usability of this method in practical situations.

8.4 Experimental Validation

The properties of the Kernel Target Alignment seem very interesting with regards to our evolutionary optimization method. Especially the fact that it has a low complexity could make it very suitable in an evolutionary context, in which many potential solutions are probed for their fitness. Furthermore, it does satisfy our requirement of applicability for multiple variants of Kernel Machines, since it only considers the kernel matrix. The kernel matrix is obviously a common aspect among all algorithms that make use of a kernel function.

Unfortunately, the empirical results presented in the literature did not convince us regarding a reliable estimation of the generalization error. This reliability is a crucial factor for our two models, of which the GP model in particular. This can be explained by two main arguments. Firstly, there is an increased risk of overfitting in this model, since both the hyperparameters and the kernel function will be adapted to the problem at hand. An objective function that is shown to be unbiased and has a low variance is therefore needed to prevent this. Secondly, the goal of the GP approach is to improve on the generalization performance, as compared to traditional methods. This generalization performance depends to a great degree on the objective function that is used, as that is the actual measure that is being optimized. Also from this perspective it is crucial that our objective function gives a reliable estimation of the true generation error.

For these reasons we have opted to conduct our own – empirical – validation to see how well Kernel Target Alignment performs in this important aspect. Besides Kernel Target Alignment, we have also included the leave-one-out cross validation method. This latter could be interesting, as there exist routines to calculate this measure efficiently. Performing this experiment in a completely statistically sound way is, unfortunately, a very tedious and difficult task. Because the decision on an objective function is only a subquestion in our study, we have decided to use a more "common sense" comparison methodology. This means that we will analyze the results of our study in three different ways, which are:

1. An analysis of the *correlation coefficients* of the different methods, with regard to the generalization error.
2. A human interpretation of the visualization of the *error surfaces*.
3. The generalization error that would have been obtained if one of the methods would have been used for optimization.

It may be clear that it is impossible to perform these experiments exactly as we have described them, as the true generalization error is unknown in real-life data sets. Therefore, we have made the assumption that the k-fold cross validation error is a good approximation of this error, which can be justified by findings in related literature [15, 58]. This means that Kernel Target Alignment and the leave-one-out cross validation methods will be compared, where the k-fold cross validation error is the reference.

8.4.1 Experimental Setup

The setup for these experiments is identical to the experimental setup described in Section 7.2[3]. A grid search has been performed using LibLSSVM for the six data sets on a predefined range of the parameters. This search has been performed using k-fold cross validation, fast leave-one-out cross validation, and Kernel Target Alignment. A discrepancy from the previous experiments is that here we have used the full LS-SVM variant and *not* the reduced variant. We have chosen $k = 5$, as to have a trade-off between computational load and the reliability of the error. Furthermore, the kernel matrix that we have applied in Kernel Target Alignment is the matrix that is obtained *after* adding C^{-1} on the diagonal. This is done so that the hyperparameter C is included in the optimization using Kernel Target Alignment. For large values of C, however, the function approximates the original definition of Kernel Target Alignment.

The grid search has been performed on the two most used kernel functions, i.e. the RBF kernel and the inhomogeneous polynomial kernel. These kernels have been chosen because they are shown to perform well and they are common in the related literature. The polynomial kernel has been restricted to the degree $d = 3$, as to reduce the search to only two parameters (i.e. C and the constant c). This limitation is done for two reasons. Firstly, grid search scales exponentially, which makes it impracticable for more than two parameters. Secondly, it would be very hard to visually represent a problem with three parameters, as we would need four dimensions (i.e. three parameters and the error).

Furthermore, the leave-one-out and 5-fold cross validation measures give an estimate by calculating an empirical error. This empirical error is the *Mean Squared Error* (*MSE*) in case of regression problems and the normal classification error for classification errors. Kernel Target Alignment, on the other hand, does not give a direct error estimate. Instead it returns a measure of agreement between the kernel function and the target problem. This alignment we want to *maximize*, whereas the empirical error need to be *minimized*. Because of this we have taken the inverse of the Kernel Target Alignment, so that it is transformed into a minimization problem. Furthermore, the value have been decreased by a factor of 100, so that its range will be similar to that of the errors. This simplifies the interpretation of the graphical representation of the surfaces. Values higher than 100 have been filtered from the results of all methods, as to remove any extreme outliers from the comparison.

8.4.2 Correlations between the Various Methods

The first step in comparing the accuracy of Kernel Target Alignment and leave-one-out is done by measuring the correlation with 5-fold cross validation. If the two measures are reliable estimators for the generalization error, then we expect there to be a high correlation with the reference (i.e. 5-fold cross validation) measure. Two different types of correlation have been calculated, namely *Pearson product-moment correlation coefficient* and Kendall's τ rank correlation coefficient. The former, which we will refer to as ρ_P, indicates the degree of linear dependence between two variables. It is obtained by dividing the covariance of the two variables by the product of their standard deviations. A value of $+1$ denotes a perfect positive linear relationship between the variables, whereas -1 indicates a perfect negative relationship. If the coefficient is 0, then there is no linear relationship at all.

The Pearson correlation coefficient may not be entirely suited for our situation. Firstly, it assumes normality of both random variables, which we cannot ensure. More importantly, the coefficient measures linear dependence, which intuitively does not need to hold for Kernel Target Alignment. Therefore we have chosen

[3]We would like to refer the reader to that section for the details that we have omitted here.

Table 8.1: Correlation coefficients between Kernel Target Alignment (*KTA*) and leave-one-out cross validation (*LOO*) and the reference 5-fold cross validation. In the table ρ_P denotes the Pearson product-moment correlation coefficient and τ_K stands for the Kendall's τ rank correlation coefficient.

Name	Kernel	ρ_P KTA	τ_K KTA	ρ_P LOO	τ_K LOO
Diabetes	RBF	0.0115	0.3072	0.9838	0.8337
	Polynomial 3	-0.0451	0.0483	0.8981	0.6077
Housing	RBF	0.1351	0.3806	0.9996	0.9635
	Polynomial 3	0.0155	0.0945	0.6339	0.6535
Reaching 1	RBF	0.2161	0.6205	0.9894	0.9413
	Polynomial 3	0.5545	0.2413	0.9991	0.9443
Reaching 2	RBF	0.4413	0.7819	0.9961	0.7588
	Polynomial 3	0.3654	0.0262	0.9996	0.9719
Reaching 3	RBF	0.4599	0.8352	0.9991	0.7951
	Polynomial 3	0.3918	0.4437	0.9994	0.9818
Wisconsin	RBF	-0.0242	-0.0349	0.9956	0.9401
	Polynomial 3	-0.1054	-0.2438	0.9693	0.8403

to include a second correlation measure, i.e. Kendall's rank correlation coefficient. Henceforth we will refer to this coefficient as τ_K. The intuition behind including this correlation measure is that the actual relation between the two variables is not necessarily important. Consider for instance that there would be a quadratic dependence between the 5-fold cross validation error and Kernel Target Alignment. Although the value for ρ_P would be very low, indicating no linear correlation, the method would still be perfectly applicable for our problem. In other words, our main interest is that the ranking of the hyperparameters is similar. Kendall's τ measures the similarity in the ranking of the variables by dividing the number of concordant pairs in the ranking by the total number of pairs. A value of $+1$ thus indicates that all pairs are concordant, -1 indicates that all pairs are discordant, and a value of 0 means that half of the pairs are concordant – the other half is discordant. Both coefficients have been calculated using the *R* software environment for statistical computing [92].

In short, an interpretation of *both* correlation coefficients should give us insight into the reliability of the estimators. The shortcomings of either correlation coefficient are covered by the other method. The results that we have obtained for the various data sets are shown in Table 8.1. These figures show that Kernel Target Alignment performs rather disappointing. For practically all combinations of data sets and kernel functions there is a very low correlation. This situation is roughly identical for both types of correlation coefficients. These results suggest that Kernel Target Alignment is not a suitable estimator for the generalization error.

On the other hand, the leave-one-out cross validation performs as we would expect. In nearly all cases both types of correlation coefficients are very near the optimal value of 1. The only exception is when the Housing data set is used in combination with the polynomial kernel. Later we will investigate this discrepancy in the visualization of the error surfaces.

8.4.3 Graphical Interpretation

The correlations between the two measures and the reference gives us an idea about how well suited both methods are. However, all information that we obtained is captured in a two scalar values. However

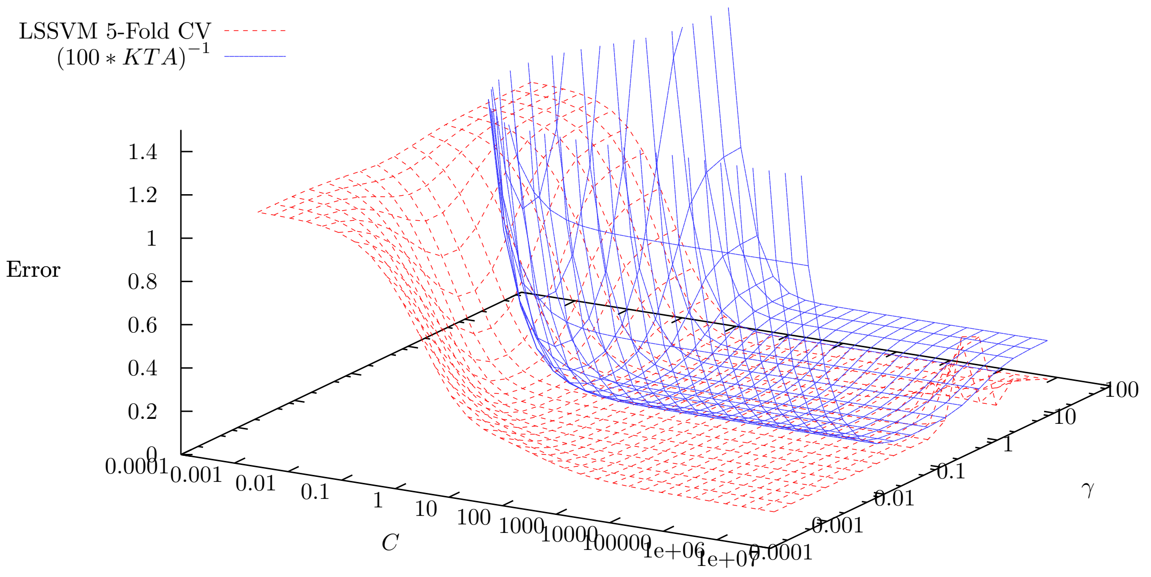


Figure 8.1: The Kernel Target Alignment and 5-fold cross validation error surfaces for the Reaching 2 data set in combination with the RBF kernel.

statistically correct this may be, it does not give that much insight in the actual problem and important information is lost in the process. For this reason we have visualized the error surfaces for all methods, as to increase the – human – understanding in this matter. Nonetheless, we acknowledge to the fullest extent that no decisions should be based *solely* on this human interpretation. The aim of this visualization is only to increase our understanding and possibly to put the analytical results, i.e. the correlation values, into perspective.

In Fig. 8.1 we see the surface of the Kernel Target Alignment score on the Reaching 2 data set and with the RBF kernel function. The figure features the reference 5-fold cross validation error surface as well. It is hard to notice a clear similarity between these two surfaces, even though the correlation coefficient for this combination of data set and kernel function was relatively high. Nonetheless, we prefer to note that minimizing the Kernel Target Alignment would – in this case – result give a relatively low error for the 5-fold cross validation error as well. In the next section we will investigate this aspect in more detail. An observation that needs to be mentioned is that the surface of Kernel Target Alignment is monotonically decreasing as the value for C increases. This confirms that the measure is not suited for the optimization of this tradeoff parameter. The situation for the other data sets and kernel functions is similar or slightly worse in all aspects that have been mentioned.

The surface for the leave-one-out error is expected to resemble the reference surface closely, given the high correlation values that we saw in the previous section. The figures of both surfaces support this expectation completely, see for example Fig. 8.2. In nearly all cases there was only a minimal discrepancy between the error surfaces. However, previously we noted that for the Housing data set, in combination with the polynomial kernel, the correlation was significantly lower. Both error surfaces have been depicted in Fig. 8.3. This figure clearly shows that there is indeed a local discrepancy between both surfaces. The measures give a different error estimation for situations where c is small and C is large. An interesting observation is that the surface of the leave-one-out measure seems – to the human observer – more consistent.

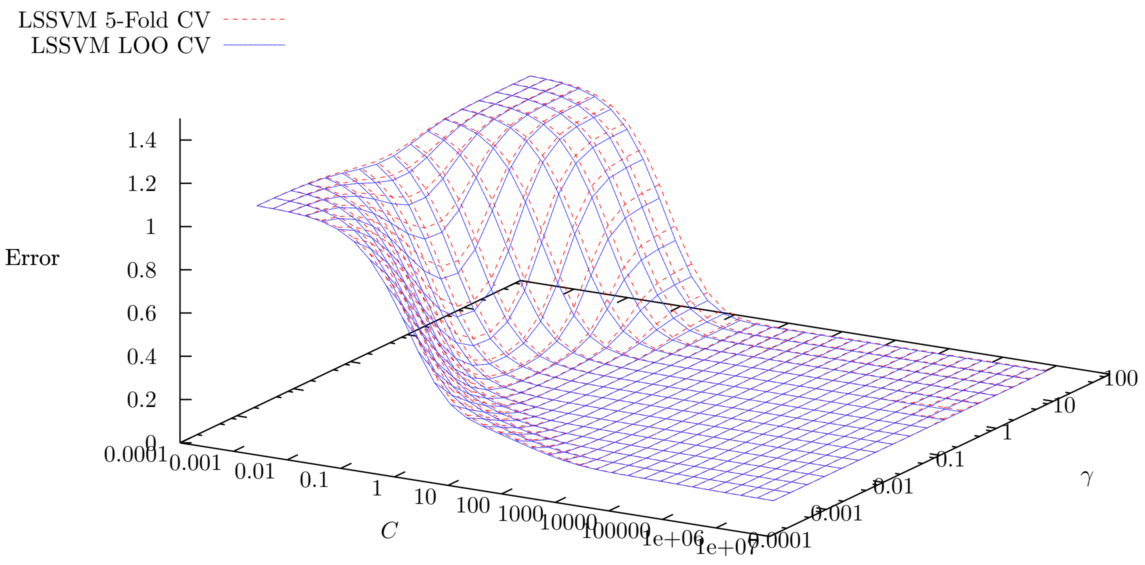


Figure 8.2: The leave-one-out and 5-fold cross validation error surfaces for the Reaching 3 data set in combination with the RBF kernel.

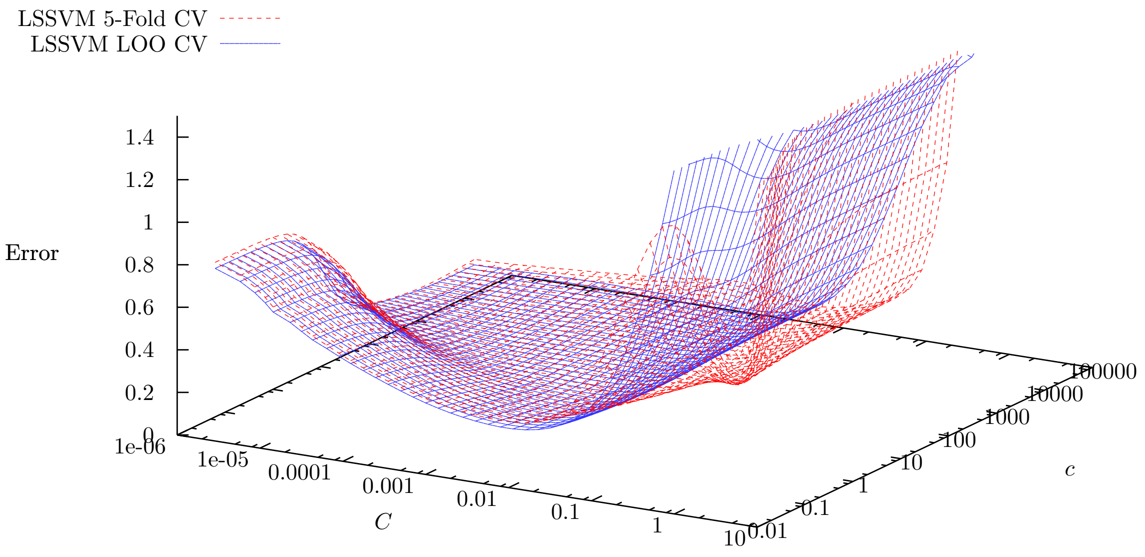


Figure 8.3: The leave-one-out and 5-fold cross validation error surfaces for the Housing data set in combination with the polynomial kernel with degree 3.

Table 8.2: The minimum 5-fold cross validation errors that correspond to the optimal hyperparameters found using the various objective functions.

Name	Kernel	ϵ_{5CV} **5CV**$_{min}$	ϵ_{5CV} **KTA**$_{min}$	ϵ_{5CV} **LOO**$_{min}$
Diabetes	RBF	0.2187	0.2356	0.2356
	Polynomial 3	0.2213	0.2435	0.2265
Housing	RBF	0.1172	0.3233	0.1187
	Polynomial 3	0.1419	8.4269	0.1570
Reaching 1	RBF	0.0473	0.2019	0.0484
	Polynomial 3	0.0721	0.0723	0.0721
Reaching 2	RBF	0.0013	0.0013	0.0102
	Polynomial 3	0.0063	0.0063	0.0063
Reaching 3	RBF	0.0008	0.0008	0.0033
	Polynomial 3	0.0032	0.0032	0.0032
Wisconsin	RBF	0.0400	0.1315	0.0867
	Polynomial 3	0.0445	0.1959	0.0467

8.4.4 “What if...” Interpretation

The last section of the juxtaposition of the two error estimators we have named – rather cryptically – the “what if…” interpretation. Herewith we try to answer the simple question: *What if we would optimize the hyperparameters using either of these methods?* In other words, which 5-fold cross validation error would be obtained if the optimization was done using the leave-one-out or Kernel Target Alignment measures. Despite the obvious simplicity of this question, we believe that answering this question can give us useful information. It captures the essence of our goal, which is optimizing an efficient measure that so that the generalization performance is maximized. Naturally, this statement relies on the assumption that 5-fold cross validation is a reliable estimator for the true generalization performance.

In Table 8.2 the results are shown which errors would have been obtained using the various measures. Obviously, the third column (i.e. ϵ_{5CV} 5CV$_{min}$) is the lower bound, as it simply lists the minimal 5-fold cross validation error. It is interesting to notice that for some data sets and kernels Kernel Target Alignment performs in line with the other two methods. In some situations it even performs better than the leave-one-out measure. We would not have expected this, given the results from the previous two interpretations. Nonetheless, the leave-one-out measure performs overall still better than Kernel Target Alignment. One exceptional outlier in the case of Kernel Target Alignment is for the Housing data set, in combination with the polynomial kernel. In this case the optimal hyperparameters according to Kernel Target Alignment produce an extremely high error when used with the 5-fold cross validation measure. The explanation is that the Kernel Target Alignment decreases monotonically as the value for C increases. Therefore the optimal value is found at the maximum value for C in our range. The 5-fold cross validation surface for this data set and kernel, on the other hand, at a certain point increases drastically as the value for C increases (see Fig. 8.3).

The impressions regarding leave-one-out cross validation are mixed. Overall the optimal errors found are close to those as found with 5-fold cross validation. Unfortunately, there are some occasions where the discrepancy is significant. For instance, for the Wisconsin data set with the RBF kernel the obtained error is more than twice as large as the optimal error. This puts the capacity of leave-one-out as a reliable estimator slightly in a different perspective.

8.5 Conclusion

An important element in the evolutionary models, as presented in this report, is the objective function. In the case of machine learning, this objective function ideally should quantify the true generalization capacity of the machine and its hyperparameters. The problem lies within the fact that the true generalization error, i.e. the expected error, can not be calculated, as the real underlying probability distribution of the problem is unknown. Instead, the error must be approximated by means of empirical validation on a data set with finite size. Finding a measure that does this reliably in an efficient manner is not trivial, as the different measures have very different characteristics. In this chapter we tried to answer this subquestion for our study, i.e. which measure we could best use in our models. We considered three different measures, namely 5-fold cross validation, leave-one-out cross validation and Kernel Target Alignment.

The first, i.e. 5-fold cross validation is shown to be an unbiased estimator of the generalization error, with a low variance. Therefore, this measure is considered to be a relatively reliable estimator of the true generalization error. We have chosen to use this measure as the reference in our comparison, because of this low variance. The disadvantage of this method, however, is that it is computationally expensive. Kernel Target Alignment, on the other hand, can be calculated much more efficiently. Also the leave-one-out cross validation routine is, due to certain “shortcuts”, more efficient than 5-fold cross validation.

The aim of this chapter was to investigate whether either of these two measures could give reliability, similar to that of 5-fold cross validation, combined with a reduced computational load. Regarding Kernel Target Alignment we can be short; the results show that this measure should not be considered a very reliable estimator of the true generalization error. This statement is supported in practically all our interpretations of the problem and for all data sets and kernels. Potentially Kernel Target Alignment could have other applications (e.g. optimizing solely the kernel parameter), but it did not fit the requirements presented in our study.

The situation for leave-one-out cross validation is somewhat more complex. In general, this measure exhibits a very high correlation with the 5-fold cross validation. This is not surprisingly, as both methods are very closely related to each other. Also the error surfaces of both measures seem to match nearly perfectly. Despite this fact, we have opted to *not* use the leave-one-out measure in our study. Our reasons for this decision are two-fold. Most importantly, we have shown that in some situations optimizing the leave-one-out measure results in a relatively high error. The overall generalization performance of our model, obviously, depends to a great extent on the objective function that is being used. Therefore, we prefer to use the best possible approximation for the true generalization error, which is k-fold cross validation. We believe that using $k = 5$ gives us a reasonable tradeoff between computational performance and reliability.

The intuition regarding computational efficiency may erroneously be biased toward leave-one-out cross validation. One would expect this measure, given that it is calculated using the efficient method, to be k times as fast as k-fold cross validation. However, the fact is overseen that the leave-one-out measure trains on the *complete* data set, whereas the k-fold cross validation uses only a fraction of $\frac{k-1}{k}$. For $k = 5$ this means that the learning set is 1.25 times as large for the leave-one-out measure. Given that the complexity of LS-SVM is $\mathcal{O}\left(n^3\right)$ with respect to the size of the dataset, this means that the learning process will take approximately $(1.25)^3$ times as long. Conclusively, the 5-fold cross validation measure will take only approximately $\frac{5}{(1.25)^3} = 2.56$ times as long.

A – less important – second reason is that the efficiency of the leave-one-out is given by *specialized* routines. These routines are dependent on the actual type of Kernel Machine that we are using, i.e. LS-SVM in our case. Making use of these specialized routines would, in our opinion, decrease the general applicability of our model. We prefer to keep the model as general as possible, also if this comes as a computational cost.

Part IV

Conclusion

Chapter 9

Conclusions and Future Work

In the introduction of this report we have stated two main research questions and a third related subquestion. Let us briefly recapitulate our research questions:

1. *Can we use Evolutionary Computation techniques in order to rapidly optimize the hyperparameters of a Kernel Machine?*
2. *Can we use Evolutionary Computation techniques to evolve an optimal kernel function for a Kernel Machine given a certain problem?*
3. *What measure of quality can be used for kernel functions and hyperparameters, in the context of the previous two research questions?*

Based on these questions, we have defined and implemented two models. The implementations have been used to verify empirically whether our goals and objectives could be met. Furthermore, an experimental study has been conducted to investigate our third research question. In this concluding section, a summary will be given of our observations with regard to our experimental validations. Furthermore, we will briefly discuss the practical contributions of our study and give suggestions for future work.

9.1 Evaluation of the Models

In response to the first research question, we have presented a model based on Evolution Strategies that optimizes the hyperparameters. Our experiments suggest that Evolutionary Computation is a good, generalized optimization method for the hyperparameters, as long as the problem at hand is a regression problem. Our ES model converges to good solutions by sampling the fitness landscape and moving into regions where the fitness is high. This approach works well on the continuous error surfaces of regression problems, which results in a high convergence rate. Classification problems intrinsically have a discontinuous fitness landscape. This imposes difficulties for the surface sampling approach of our model.

Nonetheless, the results of our model are still promising, despite this deficiency on classification problems. The main issue is that ES may not find competitive solutions – as compared to the traditional grid search – in a timely fashion. Nonetheless, it was still able to rapidly come up with solutions that were reasonable. This suggests that ES is a method capable of obtaining an approximation of the minimum error for any type of problem. For regression problems in specific, the results suggest that ES can find equally good solutions as the traditional grid search using only a fraction of the computational requirements. An interesting advantage is that ES scales much better with the number of parameters than grid search.

The results of our Genetic Programming based model are less positive. This model was proposed in response to our second research question. Our experiments suggest that there is only a modest improvement in generalization performance to be made by evolving complex kernel functions. In most circumstances this slight improvement will not justify the high computational demands of our model.

It is difficult to determine the exact cause of these results. The fact that we have not found kernel functions that considerably improve on the generalization performance does not necessarily mean that such kernel functions will not exist at all. One thing that is clear is that evolving kernel functions for a Kernel Machine is far from trivial. The configuration of GP, in terms of the evolver model and the parameters, can determine to a great extent the results. However, the vast amount of options make it very difficult to find a configuration that works the best.

The last results that we have presented in this report were a minor empirical study we have conducted in response to the third research question. Presumably, Kernel Target Alignment is a quantitative measure for the agreement between a kernel function and a data set. It would make a very interesting objective function in the context of our main study, given the temporal and spatial efficiency of the algorithm. However, we have empirically shown that Kernel Target Alignment is not a very reliable estimator of the generalization performance. Therefore, we have opted to use 5-fold cross validation for our experiments instead of Kernel Target Alignment. We recommend that Kernel Target Alignment is avoided as a performance measure in situations where the quality of the solutions is relevant.

9.2 Practical Contributions

During the course of this study certain practical contributions have been made, besides the scientific contributions described above. The most relevant of these is the implementation of a complete LS-SVM framework, i.e. LibLSSVM. This framework implements the normal as well as a reduced variant of the LS-SVM algorithm. It has been developed with two primary objectives, namely, *efficiency* and *extensibility*. The former objective has been met by using highly optimized libraries for the linear algebra routines, whereas the latter has been dealt with by using the Object-Oriented programming paradigm. We have exploited the extensibility in our studies for using automatically generated kernel functions. In our experience, this extensibility make that the framework is well suited for studies on non-standard kernel functions. Furthermore, an integration of LibLSSVM in the YARP robotics platform has been planned for the future. This would allow for the usage of a sophisticated machine learning technique in the field of robotics by less experienced users.

We believe that our implementation of EvoKM$^{\text{ES}}$ can be useful for the optimization of the hyperparameters of Kernel Machines. The implementation can be used with any kind of kernel function and is not restricted to any number of hyperparameters. One disadvantage in its current form is that it is based on the OpenBeagle framework, which is relatively large. A minimal ES implementation, specially crafted for the task of hyperparameter optimization, could be developed and integrated into LibLSSVM.

9.3 Future Work

One particular phenomenon in scientific research is that the process of solving questions in itself raises many new questions. Our personal experience during the course of this study has been no different. The results from our study have given us certain directions for future research. Regarding the ES model that we proposed, we believe that it would be interesting to investigate how the results on classification problems can be improved. Possibly, the problem could be solved by using GA instead of ES, which is known to perform better on discontinuous fitness functions. Another possible solution could lie in smoothening the surface, as is done with certain gradient descent methods. Moreover, it would be interesting to juxtapose our EvoKM$^{\text{ES}}$ method with some of the hyperparameter optimization methods described in related work (see Section 4.1). In particular, a comparison with the gradient descent methods would be highly interesting.

Although EvoKM$^{\text{ES}}$ performed very well for the regression data sets, there are still improvements that can be made. Firstly, the exogenous strategy parameters could empirically be optimized for the problem of hyperparameter optimization. The objective is to find the parameters that guarantee a high convergence rate on the one hand and a stable and robust search process on the other hand. The probability that an outlier occurs should be minimized, as to ensure that a single search will yield globally optimal results. Another possible improvement is to make use of the more advanced covariance matrix adaptation variant of ES.

Further experiments and analysis of EvoKM$^{\text{GP}}$ could give more insight on whether complex kernel functions can or cannot improve the generalization performance. As we have stated before, the fact that EvoKM$^{\text{GP}}$ was not able to make substantial improvements does not necessarily mean that we can state that combining kernel functions will never make improvements. Given the vast complexity of the problem, much more experiments with different configurations will be needed to support such a statement. One potential model could extricate the kernel evolution and hyperparameter optimization problems by means of a hybrid approach. The kernel function could be evolved using GP, after which its parameters are optimized using a different technique (e.g. ES or grid search). This would prevent the fact that potentially good kernel functions will be lost due to the selection pressure, only because their parameterization is far from optimal.

There are many open questions regarding Kernel Target Alignment. The main questions are what Kernel Target Alignment exactly measures and how useful this is as an estimator of the generalization error. Furthermore, recall that the efficiency of Kernel Target Alignment is caused by the fact that it only considers the kernel matrix. It seems very likely that the kernel matrix can indeed give valuable information on its suitability for a given data set. Potentially, more sophisticated methods could be devised that measure this agreement based on the kernel matrix and the data set that measure this agreement.

BIBLIOGRAPHY

[1] Mark A. Aizerman, Emmanuel M. Braverman, and Lev I. Rozoner. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25:821–837, 1964.

[2] Arthur Asuncion and David J. Newman. UCI machine learning repository, 2007.

[3] Hans-Georg Beyer. Some aspects of the evolution strategy for solving tsp-like optimization problems. In *Parallel Problem Solving from Nature*, volume 2, page 361370. Elsevier, 1992.

[4] Hans-Georg Beyer. *The theory of evolution strategies*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[5] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies a comprehensive introduction. *Natural Computing: an international journal*, 1(1):3–52, 2002.

[6] Thomas Briggs and Tim Oates. Discovering domain-specific composite kernels. In *Proceedings of the Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference*, pages 732–738, July 2005.

[7] Rodney A. Brooks. Elephants don't play chess. *Robotics and Autonomous Systems*, 6(1&2):3–15, June 1990.

[8] Rodney A. Brooks, Cynthia Breazeal, Matthew Marjanovic, Brian Scassellati, and Matthew M. Williamson. The cog project: Building a humanoid robot. *Lecture Notes in Computer Science*, 1562:52–87, 1999.

[9] Michael C. Bucko. Autonomous hand localization and detection in a humanoid robot. Master's thesis, Universität Karlsruhe, Lehrstuhl Industrielle Anwendungen der Informatik und Mikrosystemtechnik, Mikrosystemtechnik, May 2007.

[10] Christopher J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

[11] Evgeny Byvatov, Uli Fechner, Jens Sadowski, and Gisbert Schneider. Comparison of support vector machine and artificial neural network systems for drug/nondrug classification. *Journal of Chemical Information and Computer Sciences*, 43(6):1882–1889, 2003.

[12] Colin Campbell and Nello Cristianini. Simple learning algorithms for training support vector machines, 1998.

[13] Gavin C. Cawley. Leave-one-out cross-validation based model selection criteria for weighted ls-svms. In *IJCNN-2006: Proceedings of the International Joint Conference on Neural Networks*, pages 1661–1668, Vancouver, BC, Canada, July 2006.

[14] Gavin C. Cawley and Nicola L. C. Talbot. Fast exact leave-one-out cross-validation of sparse least-squares support vector machines. *Neural Networks*, 17(10):1467–1475, 2004.

[15] Olivier Chapelle, Vladimir N. Vapnik, Olivier Bousquet, and Sayan Mukherjee. Choosing multiple parameters for support vector machines. *Machine Learning*, 46(1–3):131–159, 2002.

[16] Datong Chen and Jean-Marc Odobez. Comparison of support vector machine and neural network for text texture verification. Technical Report 19, IDIAP, Martigny, April 2002.

[17] Peng-Wei Chen, Jung-Ying Wang, and Hahn-Ming Lee. Model selection of svms using ga approach. *Proceedings of the 2004 IEEE International Joint Conference on Neural Networks*, 3(2):2035–2040, July 2004.

[18] Gordon Cheng, Sang-Ho Hyon, Jun Morimoto, Aleš Ude, Joshua G. Hale, Glenn Colvin, Wayco Scroggin, and Stephen C. Jacobsen. Cb: a humanoid research platform for exploring neuroscience. *Advanced Robotics*, 21(10):1097–1114, 2007.

[19] Kai-Min Chung, Wei-Chun Kao, Chia-Liang Sun, Li-Lun Wang, and Chih-Jen Lin. Radius margin bounds for support vector machines with the rbf kernel. *Neural Computation*, 15(11):2643–2681, 2003.

[20] Zheng Chunhong and Jiao Licheng. Automatic parameters selection for svm based on ga. In *WCICA 2004: Fifth World Congress on Intelligent Control and Automation*, volume 2, pages 1869–1872, June 2004.

[21] Nello Cristianini, Colin Campbell, and John Shawe-Taylor. Dynamically adapting kernels in support vector machines. In *Proceedings of the 1998 conference on Advances in neural information processing systems II*, pages 204–210, Cambridge, MA, USA, 1999. MIT Press.

[22] Nello Cristianini, Andre Elisseeff, John Shawe-Taylor, and Jaz Kandola. On kernel target alignment. In *Proceedings Neural Information Processing Systems*, 2001.

[23] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, March 2000.

[24] Lehel Csato and Manfred Opper. Sparse online gaussian processes. *Neural Computation*, 14(3):641–669, 2002.

[25] John E. Dennis and Virginia J. Torczon. Derivative-free pattern search methods for multidisciplinary design problems. In *Proceedings of the Fifth AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, pages 992–993, 1994.

[26] Mehrdad Dianati, Insop Song, and Mark Treiber. An introduction to genetic algorithms and evolution strategies. Technical report, University of Waterloo, July 2002.

[27] Laura Dioşan and Mihai Oltean. Evolving kernel function for support vector machines. In Cagnoni C., editor, *The 17th European Conference on Artificial Intelligence, Evolutionary Computation Workshop*, pages 11–16, 2006.

[28] Laura Dioşan, Mihai Oltean, Alexandrina Rogozan, and Jean Pierre Pecuchet. Improving svm performance using a linear combination of kernels. In *ICANNGA '07: International Conference on Adaptive and Natural Computing Algorithms*, number 4432 in LNCS, pages 218–227. Springer, 2007.

[29] Harris Drucker, Chris J. C. Burges, Linda Kaufman, Alex Smola, and Vladimir Vapnik. Support vector regression machines. In *Advances in Neural Information Processing Systems*, volume 9, page 155. The MIT Press, 1996.

[30] Kaibo Duan, S. Sathiya Keerthi, and Aun Neow Poo. Evaluation of simple performance measures for tuning svm hyperparameters. *Neurocomputing*, 51:41–59, 2003.

[31] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. Wiley-Interscience, 2001.

[32] Jeroen Eggermont, Joost N. Kok, and Walter A. Kosters. Genetic programming for data classification: partitioning the search space. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1001–1005, New York, NY, USA, 2004. ACM Press.

[33] Shai Fine and Katya Scheinberg. Efficient svm training using low-rank kernel representations. *Journal on Machine Learning Research*, 2:243–264, 2002.

[34] Roger Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, New York, second edition, 1987.

[35] Frauke Friedrichs and Christian Igel. Evolutionary tuning of multiple svm parameters. In *ESANN 2004: Proceedings of the 12th European Symposium on Artificial Neural Networks*, pages 519–524, April 2004.

[36] Holger Fröhlich, Olivier Chapelle, and Bernhard Schölkopf. Feature selection for support vector machines by means of genetic algorithms. In *ICTAI '03: Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*, page 142, Washington, DC, USA, 2003. IEEE Computer Society.

[37] Holger Fröhlich and Andreas Zell. Efficient parameter selection for support vector machines in classification and regression via model-based global optimization. In *Proceedings of the International Joint Conference on Neural Networks*, pages 1431–1438, 2005.

[38] Christian Gagné and Marc Parizeau. Genericity in evolutionary computation software tools: Principles and case study. *International Journal on Artificial Intelligence Tools*, 15(2):173–194, April 2006. 22 pages.

[39] Christian Gagné, Marc Schoenauer, Marc Parizeau, and Marco Tomassini. Genetic programming, validation sets, and parsimony pressure. In *EuroGP 2006: Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *LNCS*, pages 109–120. Springer, April 10-12 2006.

[40] Christian Gagné, Marc Schoenauer, Michele Sebag, and Marco Tomassini. Genetic programming for kernel-based learning with co-evolving subsets selection. In *Parallel Problem Solving from Nature - PPSN IX*, volume 4193 of *LNCS*, pages 1008–1017, Reykjavik, Iceland, September 2006. Springer-Verlag.

[41] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[42] Stuart Geman, Elie Bienenstock, and René Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4(1):1–58, 1992.

[43] Nikolaus Hansen and Andreas Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In *International Conference on Evolutionary Computation*, pages 312–317, 1996.

[44] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

[45] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1994.

[46] Luc Hoegaerts, Johan A. K. Suykens, Joost Vandewalle, and Bart De Moor. Subset based least squares subspace regression in rkhs. *Neurocomputing*, 63:293–323, 2005.

[47] John H. Holland. Outline for a logical theory of adaptive systems. *Journal of the ACM*, 9(3):297–314, 1962.

[48] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[49] Tom Howley and Michael G. Madden. The genetic evolution of kernels for support vector machine classifiers. In *Proceedings of 15th Irish Conference on Artificial Intelligence and Cognitive Science*, September 2004.

[50] Tom Howley and Michael G. Madden. The genetic kernel support vector machine: Description and evaluation. *Artificial Intelligence Review*, 24(3-4):379–395, 2005.

[51] Chin-Chia Hsu, Chih-Hung Wu, Shih-Chien Chen, and Kang-Lin Peng. Dynamically optimizing parameters in support vector regression: An application of electricity load forecasting. In *HICSS '06: Proceedings of the 39th Annual Hawaii International Conference on System Sciences*, page 30.3, Washington, DC, USA, 2006. IEEE Computer Society.

[52] Cheng-Lung Huang and Chieh-Jen Wang. A ga-based feature selection and parameters optimization for support vector machines. *Expert Systems with Applications*, 31(2):231–240, 2006.

[53] Jaz Kandola and John Shawe-Taylor. Refining kernels for regression and uneven classification problems. In *Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics*, 2003.

[54] Jaz Kandola, John Shawe-Taylor, and Nello Cristianini. On the extensions of kernel alignment. Technical report, Department of Computer Science, Royal Holloway, University of London, UK, January 2002.

[55] Jaz Kandola, John Shawe-Taylor, and Nello Cristianini. Optimizing kernel alignment over combinations of kernels. Technical Report 121, Department of Computer Science, Royal Holloway, University of London, UK, 2002.

[56] S. Sathiya Keerthi, Vikas Sindhwani, and Olivier Chapelle. An efficient method for gradient-based adaptation of hyperparameters in svm models. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 673–680. MIT Press, Cambridge, MA, USA, 2007.

[57] Kenji Kobayashi, Daisuke Kitakoshi, and Ryohei Nakano. Yet faster method to optimize svr hyperparameters based on minimizing cross-validation error. In *IJCNN '05: Proceedings of the IEEE International Joint Conference on Neural Networks*, volume 2, pages 871– 876, August 2005.

[58] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International Joint Conference on Artificial Intelligence*, pages 1137–1145, 1995.

[59] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[60] John R. Koza and Riccardo Poli. Genetic programming. In Edmund K. Burke and Graham Kendall, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, chapter 5. Springer, 2005.

[61] Natalio Krasnogor and Jim E. Smith. A tutorial for competent memetic algorithms: model, taxonomy and design issues. *IEEE Transactions on Evolutionary Computation*, 9(5):474–488, 2005.

[62] Gert R. G. Lanckriet, Nello Christianini, Peter L. Bartlett, Laurent El Ghaoui, and Michael I. Jordan. Learning the kernel matrix with semi-definite programming. In *ICML '02: Proceedings of the Nineteenth International Conference on Machine Learning*, pages 323–330, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[63] Gert R. G. Lanckriet, Nello Cristianini, Peter L. Bartlett, Laurent El Ghaoui, and Michael I. Jordan. Learning the kernel matrix with semidefinite programming. *Journal of Machine Learning Research*, 5:27–72, 2004.

[64] Wan-Jui Lee, Sergey Verzakov, and Robert P. W. Duin. Kernel combination versus classifier combination. In *MCS 2007: Proceedings of the 7th International Workshop on Multiple Classifier Systems*, pages 22–31, May 2007.

[65] Stefan Lessmann, Robert Stahlbock, and Sven F. Crone. Optimizing hyperparameters of support vector machines by genetic algorithms. In *ICAI 2005: International Conference on Artificial Intelligence*, volume 1, pages 74–82, Las Vegas, Nevada, USA, June 2005.

[66] Stefan Lessmann, Robert Stahlbock, and Sven F. Crone. Genetic algorithms for support vector machine model selection. In *IJCNN '06: International Joint Conference on Neural Networks*, pages 3063–3069. IEEE Press, July 2006.

[67] Max Lungarella, Giorgio Metta, Rolf Pfeifer, and Giulio Sandini. Developmental robotics: a survey. *Connection Science*, 15(4):151–190, December 2003.

[68] Alexander Luntz and Victor Brailovsky. On estimation of characters obtained in statistical procedure of recognition (in russian). *Technicheskaya Kibernetica*, 3, 1969.

[69] Ronny Meir. Bias, variance and the combination of estimators; the case of linear least squares. *Advances in Neural Information Processing Systems*, 7, 1995.

[70] John Mercer. Functions of positive and negative type, and their connection with the theory of integral equations. *Philosophical Transactions of the Royal Society, London*, 209:415–446, 1909.

[71] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. YARP: yet another robot platform. *International Journal on Advanced Robotics Systems, Special Issue on Software Development and Integration in Robotics.*, March 2006.

[72] Giorgio Metta, Giulio Sandini, David Vernon, Darwin Caldwell, Nikolaos Tsagarakis, Ricardo Beira, José Santos-Victor, Auke Ijspeert, Ludovic Righetti, Giovanni Cappiello, Giovanni Stellin, and Francesco Becchi. The robotcub project - an open framework for research in embodied cognition. In *Humanoids Workshop, Proceedings of the IEEE–RAS International Conference on Humanoid Robots*, December 2006.

[73] Sung-Hwan Min, Jumin Lee, and Ingoo Han. Hybrid genetic algorithms and support vector machines for bankruptcy prediction. *Expert Systems with Applications*, 31(3):652–660, 2006.

[74] Michinari Momma and Kristin P. Bennett. A pattern search method for model selection of support vector regression. In *Proceedings of the Second SIAM International Conference on Data Mining*. SIAM, April 2002.

[75] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[76] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Technical Report Caltech Concurrent Computation Program, Report. 826, California Institute of Technology, Pasadena, California, USA, 1989.

[77] Klaus-Robert Müller, Alexander J. Smola, Gunnar Rätsch, Bernhard Schölkopf, Jens Kohlmorgen, and Vladimir Vapnik. Predicting time series with support vector machines. In *ICANN '97: Proceedings of the 7th International Conference on Artificial Neural Networks*, volume 1327, pages 999–1004, Berlin, 1997. Springer Lecture Notes in Computer Science.

[78] Lorenzo Natale, Giorgio Metta, and Giulio Sandini. A developmental approach to grasping. In *Developmental Robotics: A 2005 AAAI Spring Symposium*, March 2005.

[79] Canh Hao Nguyen and Tu Bao Ho. Kernel matrix evaluation. In *IJCAI 2007: Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 987–992, January 2007.

[80] Syng-Yup Ohn, Ha-Nam Nguyen, and Sung-Do Chi. Evolutionary parameter estimation algorithm for combined kernel function in support vector machine. In *AWCC 2004: Advanced Workshop on Content Computing*, pages 481–486, ZhenJiang, JiangSu, China, November 2004.

[81] Syng-Yup Ohn, Ha-Nam Nguyen, and Woo-Jin Choi. Combined kernel function for support vector machine and learning method based on evolutionary algorithm. In *ICONIP 2004: 11th International Conference on Neural Information Processing*, pages 1273–1278, Calcutta, India, November 2004.

[82] Syng-Yup Ohn, Ha-Nam Nguyen, Dong Seong Kim, and Jong Sou Park. Determining optimal decision model for support vector machine by genetic algorithm. In *CIS 2004: First International Symposium on Computational and Information Science*, pages 895–902, Shanghai, China, December 2004.

[83] Cheng S. Ong and Alexander J. Smola. Machine learning using hyperkernels. In *Proceedings of the International Conference on Machine Learning*, 2003.

[84] Cheng S. Ong, Alexander J. Smola, and Robert C. Williamson. Hyperkernels. In *Neural Information Processing Systems*, volume 15. MIT Press, 2002.

[85] Cheng S. Ong, Alexander J. Smola, and Robert C. Williamson. Learning the kernel with hyperkernels. *Journal of Machine Learning Research*, 6:1043–1071, 2005.

[86] Zoltán L. Óvári. Kernels, eigenvalues and support vector machines. Honours thesis, Australian National University, Canberra, 2000.

[87] Kristiaan Pelckmans, Johan A. K. Suykens, Tony Van Gestel, Jos De Brabanter, Lukas Lukas, Bart Hamers, Bart De Moor, and Joost Vandewalle. Ls-svmlab: a matlab/c toolbox for least squares support vector machines.

[88] Tanasanee Phienthrakul and Boonserm Kijsirikul. Evolutionary strategies for multi-scale radial basis function kernels in support vector machines. In *GECCO '05: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, pages 905–911, New York, NY, USA, 2005. ACM Press.

[89] Tomaso Poggio and Steve Smale. The mathematics of learning: dealing with data. *American Mathematical Society*, 50(5):537–544, 2003.

[90] William H. Press, William T. Vetterling, Saul A. Teukolsky, and Brian P. Flannery. *Numerical Recipes in C++: the art of scientific computing*. Cambridge University Press, 2002.

[91] Jose C. Principe, Neil R. Euliano, and W. Curt Lefebvre. *Neural and Adaptive Systems: Fundamentals through Simulations*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[92] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2007. ISBN 3-900051-07-0.

[93] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[94] Ingo Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. frommann-holzbog, Stuttgart, 1973. German.

[95] Ryan Rifkin, Gene Yeo, and Tomaso Poggio. Regularized least squares classification. In *Advances in Learning Theory: Methods, Model and Applications*, volume 190, pages 131–154, Amsterdam, 2003. VIOS Press.

[96] Sergio A. Rojas and Delmiro Fernandez-Reyes. Adapting multiple kernel parameters for support vector machines using genetic algorithms. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 1, pages 626–631, Edinburgh, Scotland, UK, September 2005. IEEE Press.

[97] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[98] Bernhard Schölkopf. *Support Vector Learning*. R. Oldenbourg Verlag, Munich, 1997.

[99] Bernhard Schölkopf and Alexander J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, USA, 2001.

[100] Hans-Paul Schwefel. *Numerical Optimization of Computer Models*. John Wiley & Sons, Inc., New York, NY, USA, 1981.

[101] Hans-Paul Schwefel. Collective phenomena in evolutionary systems. In *Problems of Constancy and Change – The Complementarity of Systems Approaches to Complexity*, volume 2, pages 1025–1033. International Society for General System Research, 1987.

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

[103] Burrhus Frederic Skinner. *Science and Human Behavior*. Macmillan, New York, USA, 1953.

[104] Marina Skurichina, Ludmila Kuncheva, and Robert P. W. Duin. Bagging and boosting for the nearest mean classifier: Effects of sample size on diversity and accuracy. In *MCS '02: Proceedings of the Third International Workshop on Multiple Classifier Systems*, pages 62–71, London, UK, 2002. Springer-Verlag.

[105] Alexander J. Smola and Bernhard Schökopf. Sparse greedy matrix approximation for machine learning. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 911–918, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[106] Alexander J. Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14(3):199–222, 2004.

[107] Carlos Soares and Pavel B. Brazdil. Selecting parameters of svm using meta-learning and kernel matrix-based meta-features. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 564–568, New York, NY, USA, 2006. ACM Press.

[108] Carl Staelin. Parameter selection for support vector machines. Technical Report HPL-2002-354, HP Laboratories, Israel., 2003.

[109] Jian-Tao Sun, Ben-Yu Zhang, Zheng Chen, Yu-Chang Lu, Chun-Yi Shi, and Wei-Ying Ma. Ge-cko: A method to optimize composite kernels for web page classification. In *WI '04: Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*, pages 299–305, Washington, DC, USA, 2004. IEEE Computer Society.

[110] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1998.

[111] Johan A. K. Suykens, Tony Van Gestel, Jos De Brabanter, Bart De Moor, and Joost Vandewalle. *Least Squares Support Vector Machines*. World Scientific Publishing Co., Pte, Ltd., Singapore, 2002.

[112] Johan A. K. Suykens and Joost Vandewalle. Least squares support vector machine classifiers. *Neural Processing Letters*, 9(3):293–300, 1999.

[113] Lieven Vandenberghe and Stephen Boyd. Semidefinite programming. *SIAM Review*, 38(1):49–95, 1996.

[114] Vladimir N. Vapnik. *The nature of statistical learning theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[115] Vladimir N. Vapnik and Olivier Chapelle. Bounds on error expectation for support vector machines. *Neural Computation*, 12(9):2013–2036, 2000.

[116] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005.

[117] Darrell Whitley. An overview of evolutionary algorithms: practical issues and common pitfalls. *Information and Software Technology*, 43(14):817–831, 2001.

[118] Darrell Whitley, Marc Richards, Ross Beveridge, and Andre' da Motta Salles Barreto. Alternative evolutionary algorithms for evolving programs: evolution strategies and steady state gp. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 919–926, New York, NY, USA, 2006. ACM Press.

[119] Ludwig Wittgenstein. *Philosophische Untersuchungen (in German)*. Suhrkamp (Edition 2003 by Joachim Schulte), September 1953.

[120] David H. Wolpert and William G. Macready. No free lunch theorems for search. Technical report, Santa Fe Institute, 1995.

[121] Chih-Hung Wu, Gwo-Hshiung Tzeng, Yeong-Jia Goo, and Wen-Chang Fang. A real-valued genetic algorithm to optimize the parameters of support vector machine for predicting bankruptcy. *Expert Systems with Applications*, 32(2):397–408, 2007.

[122] Wen Zhang, Xijin Tang, and Taketoshi Yoshida. Text classification with support vector machine and back propagation neural network. In *ICCS 2007: Proceedings of the 7th International Conference on Computational Science*, volume 4490 of *Lecture Notes in Computer Science*, pages 150–157. Springer, May 2007.

Part V

Appendices

Appendix A

UML Diagrams

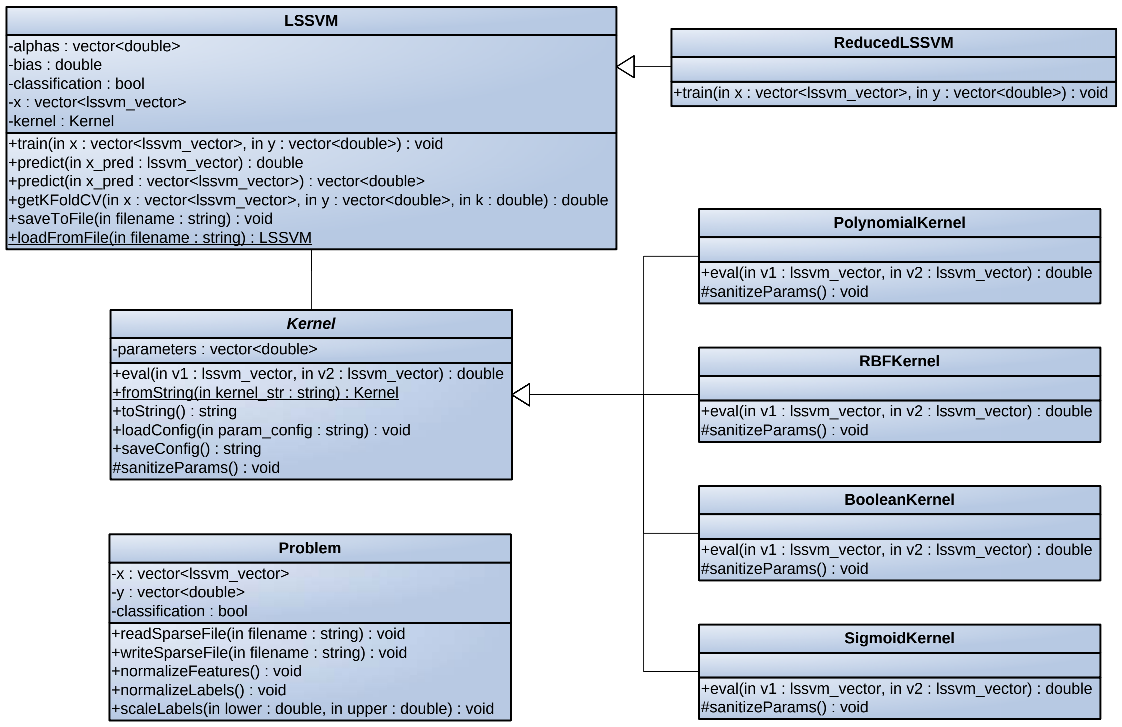

LSSVM
-alphas : vector<double>
-bias : double
-classification : bool
-x : vector<lssvm_vector>
-kernel : Kernel
+train(in x : vector<lssvm_vector>, in y : vector<double>) : void
+predict(in x_pred : lssvm_vector) : double
+predict(in x_pred : vector<lssvm_vector>) : vector<double>
+getKFoldCV(in x : vector<lssvm_vector>, in y : vector<double>, in k : double) : double
+saveToFile(in filename : string) : void
+loadFromFile(in filename : string) : LSSVM
ReducedLSSVM
+train(in x : vector<lssvm_vector>, in y : vector<double>) : void
Kernel
-parameters : vector<double>
+eval(in v1 : lssvm_vector, in v2 : lssvm_vector) : double
+fromString(in kernel_str : string) : Kernel
+toString() : string
+loadConfig(in param_config : string) : void
+saveConfig() : string
#sanitizeParams() : void
PolynomialKernel
+eval(in v1 : lssvm_vector, in v2 : lssvm_vector) : double
#sanitizeParams() : void
RBFKernel
+eval(in v1 : lssvm_vector, in v2 : lssvm_vector) : double
#sanitizeParams() : void
BooleanKernel
+eval(in v1 : lssvm_vector, in v2 : lssvm_vector) : double
#sanitizeParams() : void
Problem
-x : vector<lssvm_vector>
-y : vector<double>
-classification : bool
+readSparseFile(in filename : string) : void
+writeSparseFile(in filename : string) : void
+normalizeFeatures() : void
+normalizeLabels() : void
+scaleLabels(in lower : double, in upper : double) : void
SigmoidKernel
+eval(in v1 : lssvm_vector, in v2 : lssvm_vector) : double
#sanitizeParams() : void

Figure A.1: UML class diagram of the LibLSSVM framework.

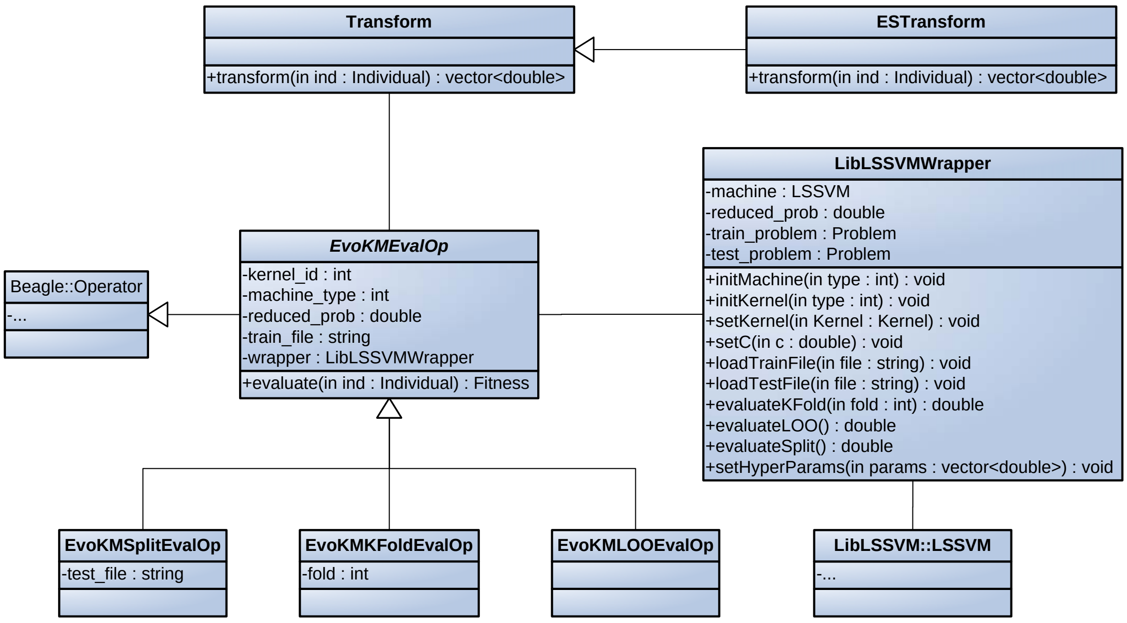

Transform
+transform(in ind : Individual) : vector<double>
ESTransform
+transform(in ind : Individual) : vector<double>
LibLSSVMWrapper
-machine : LSSVM
-reduced_prob : double
-train_problem : Problem
-test_problem : Problem
+initMachine(in type : int) : void
+initKernel(in type : int) : void
+setKernel(in Kernel : Kernel) : void
+setC(in c : double) : void
+loadTrainFile(in file : string) : void
+loadTestFile(in file : string) : void
+evaluateKFold(in fold : int) : double
+evaluateLOO() : double
+evaluateSplit() : double
+setHyperParams(in params : vector<double>) : void
EvoKMEvalOp
-kernel_id : int
-machine_type : int
-reduced_prob : double
-train_file : string
-wrapper : LibLSSVMWrapper
+evaluate(in ind : Individual) : Fitness
Beagle::Operator
-...
EvoKMSplitEvalOp
-test_file : string
EvoKMKFoldEvalOp
-fold : int
EvoKMLOOEvalOp
LibLSSVM::LSSVM
-...

Figure A.2: UML class diagram of the EvoKM^{ES} framework.

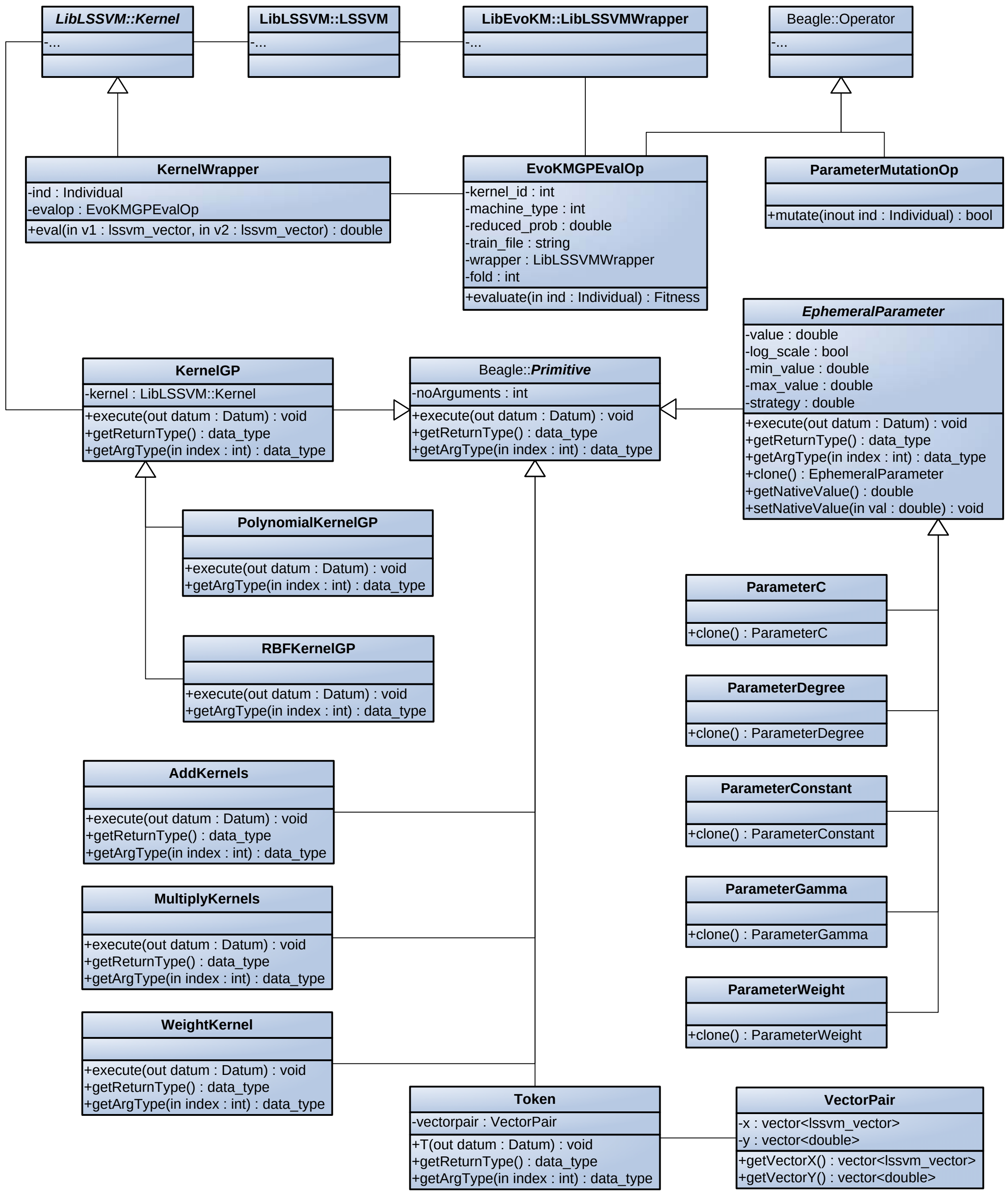

LibLSSVM::Kernel
-...
LibLSSVM::LSSVM
-...
LibEvoKM::LibLSSVMWrapper
-...
Beagle::Operator
-...
KernelWrapper
-ind : Individual
-evalop : EvoKMGPEvalOp
+eval(in v1 : lssvm_vector, in v2 : lssvm_vector) : double
EvoKMGPEvalOp
-kernel_id : int
-machine_type : int
-reduced_prob : double
-train_file : string
-wrapper : LibLSSVMWrapper
-fold : int
+evaluate(in ind : Individual) : Fitness
ParameterMutationOp
+mutate(inout ind : Individual) : bool
EphemeralParameter
-value : double
-log_scale : bool
-min_value : double
-max_value : double
-strategy : double
+execute(out datum : Datum) : void
+getReturnType() : data_type
+getArgType(in index : int) : data_type
+clone() : EphemeralParameter
+getNativeValue() : double
+setNativeValue(in val : double) : void
KernelGP
-kernel : LibLSSVM::Kernel
+execute(out datum : Datum) : void
+getReturnType() : data_type
+getArgType(in index : int) : data_type
Beagle::Primitive
-noArguments : int
+execute(out datum : Datum) : void
+getReturnType() : data_type
+getArgType(in index : int) : data_type
PolynomialKernelGP
+execute(out datum : Datum) : void
+getArgType(in index : int) : data_type
RBFKernelGP
+execute(out datum : Datum) : void
+getArgType(in index : int) : data_type
ParameterC
+clone() : ParameterC
ParameterDegree
+clone() : ParameterDegree
ParameterConstant
+clone() : ParameterConstant
ParameterGamma
+clone() : ParameterGamma
ParameterWeight
+clone() : ParameterWeight
AddKernels
+execute(out datum : Datum) : void
+getReturnType() : data_type
+getArgType(in index : int) : data_type
MultiplyKernels
+execute(out datum : Datum) : void
+getReturnType() : data_type
+getArgType(in index : int) : data_type
WeightKernel
+execute(out datum : Datum) : void
+getReturnType() : data_type
+getArgType(in index : int) : data_type
Token
-vectorpair : VectorPair
+T(out datum : Datum) : void
+getReturnType() : data_type
+getArgType(in index : int) : data_type
VectorPair
-x : vector<lssvm_vector>
-y : vector<double>
+getVectorX() : vector<lssvm_vector>
+getVectorY() : vector<double>

Figure A.3: UML class diagram of the EvoKM^{GP} framework.

Appendix B
CONFIGURATION FILES

```xml
<?xml version="1.0"?>
<Beagle>
  <Evolver>
    <BootStrapSet>
      <IfThenElseOp parameter="ms.restart.file" value="">
        <PositiveOpSet>
          <GA-InitESVecOp/>
          <EvoKMKFoldEvalOp/>
          <LogIndividualOp/>
          <StatsCalcFitnessSimpleOp/>
        </PositiveOpSet>
        <NegativeOpSet>
          <MilestoneReadOp/>
        </NegativeOpSet>
      </IfThenElseOp>
      <TermMinFitnessOp/>
      <TermMaxEvalsOp/>
      <TermMaxGenOp/>
      <MilestoneWriteOp/>
    </BootStrapSet>
    <MainLoopSet>
      <MuCommaLambdaOp>
        <LogIndividualOp>
          <EvoKMKFoldEvalOp>
            <GA-MutationESVecOp>
              <SelectRandomOp/>
            </GA-MutationESVecOp>
          </EvoKMKFoldEvalOp>
        </LogIndividualOp>
      </MuCommaLambdaOp>
      <StatsCalcFitnessSimpleOp/>
      <TermMinFitnessOp/>
      <TermMaxEvalsOp/>
      <TermMaxGenOp/>
      <MilestoneWriteOp/>
    </MainLoopSet>
  </Evolver>
  <Register>
    <Entry key="ec.term.maxgen">1000</Entry>
    <Entry key="ec.mulambda.ratio">4</Entry>
    <Entry key="ec.pop.size">3</Entry>
    <Entry key="es.init.strategy">1</Entry>
    <Entry key="es.mut.minstrategy">0.1</Entry>
    <Entry key="ec.repro.prob">1.0</Entry>
    <Entry key="lg.console.level">3</Entry>
    <Entry key="lg.file.level">3</Entry>
    <!-- Custom Termination Conditions -->
    <Entry key="ec.term.maxevals">3185</Entry>
    <!-- Minimum and Maximum values, order: kernel parameter(s), C -->
    <Entry key="es.value.min">-10;-12</Entry>
    <Entry key="es.value.max">14;20</Entry>
    <!-- Machine Settings -->
    <Entry key="evokm.kernel">1</Entry>
    <Entry key="evokm.machine.prob">0.1</Entry>
    <Entry key="evokm.machine.type">2</Entry>
    <Entry key="evokm.fold">5</Entry>
    <!-- Datasets -->
    <Entry key="evokm.train.file">../dataset/reaching_1.dat.scaled</Entry>
    <!-- Output Options -->
    <Entry key="lg.file.name">results/raw/log/reaching_1_rbf.log</Entry>
    <Entry key="lg.individual.file.name">results/raw/ind/reaching_1_rbf_ind.log</Entry>
    <Entry key="ms.write.prefix">results/raw/ms/reaching_1_rbf</Entry>
    <!-- Temporary -->
    <Entry key="ms.write.over">1</Entry>
    <Entry key="ms.write.perdeme">0</Entry>
    <Entry key="ms.write.interval">1</Entry>
  </Register>
  <Experiment>
    <Tries>25</Tries>
  </Experiment>
</Beagle>
```

Listing B.1: Example configuration file for the EvoKM^{ES} application.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<Beagle>
 <Evolver>
  <BootStrapSet>
   <IfThenElseOp parameter="ms.restart.file" value="">
    <PositiveOpSet>
     <GP-InitGrowConstrainedOp/>
     <EvoKMGPEvalOp/>
     <LogIndividualOp/>
     <GP-StatsCalcFitnessSimpleMinOp/>
     <GP-PrimitiveUsageStatsOp/>
    </PositiveOpSet>
    <NegativeOpSet>
     <MilestoneReadOp/>
    </NegativeOpSet>
   </IfThenElseOp>
   <TermMaxGenOp/>
   <MilestoneWriteOp/>
  </BootStrapSet>
  <MainLoopSet>
   <SteadyStateOp>
    <LogIndividualOp>
     <EvoKMGPEvalOp>
      <GP-CrossoverConstrainedOp>
       <SelectParsimonyTournOp/>
       <SelectParsimonyTournOp/>
      </GP-CrossoverConstrainedOp>
     </EvoKMGPEvalOp>
    </LogIndividualOp>

    <LogIndividualOp>
     <EvoKMGPEvalOp>
      <GP-MutationStandardConstrainedOp>
       <SelectParsimonyTournOp/>
      </GP-MutationStandardConstrainedOp>
     </EvoKMGPEvalOp>
    </LogIndividualOp>

    <LogIndividualOp>
     <EvoKMGPEvalOp>
      <GP-MutationShrinkConstrainedOp>
       <SelectParsimonyTournOp/>
      </GP-MutationShrinkConstrainedOp>
     </EvoKMGPEvalOp>
    </LogIndividualOp>

    <LogIndividualOp>
     <EvoKMGPEvalOp>
      <GP-MutationSwapConstrainedOp>
       <SelectParsimonyTournOp/>
      </GP-MutationSwapConstrainedOp>
     </EvoKMGPEvalOp>
    </LogIndividualOp>

   <LogIndividualOp>
     <EvoKMGPEvalOp>
      <MutationGammaOp>
      <MutationCOp>
       <SelectParsimonyTournOp/>
      </MutationCOp>
      </MutationGammaOp>
     </EvoKMGPEvalOp>
    </LogIndividualOp>

    <LogIndividualOp>
     <EvoKMGPEvalOp>
```

```
          <MutationWeightOp>
          <MutationCOp>
            <SelectParsimonyTournOp/>
          </MutationCOp>
          </MutationWeightOp>
        </EvoKMGPEvalOp>
      </LogIndividualOp>

      <LogIndividualOp>
        <EvoKMGPEvalOp>
          <MutationConstantOp>
          <MutationCOp>
            <SelectParsimonyTournOp/>
          </MutationCOp>
          </MutationConstantOp>
        </EvoKMGPEvalOp>
      </LogIndividualOp>

      <LogIndividualOp>
        <EvoKMGPEvalOp>
          <MutationDegreeOp>
          <MutationCOp>
            <SelectParsimonyTournOp/>
          </MutationCOp>
          </MutationDegreeOp>
        </EvoKMGPEvalOp>
      </LogIndividualOp>

      <SelectParsimonyTournOp/>
    </SteadyStateOp>
    <GP-StatsCalcFitnessSimpleMinOp/>
    <GP-PrimitiveUsageStatsOp/>
    <TermMaxGenOp/>
    <MilestoneWriteOp/>
  </MainLoopSet>
</Evolver>
<System>
  <PrimitiveSuperSet>
    <PrimitiveSet>
      <Primitive name="ADD_KERNELS" bias="1"/>
      <Primitive name="MULTIPLY_KERNELS" bias="1"/>
      <Primitive name="WEIGHT_KERNEL" bias="1"/>
      <Primitive name="RBF_KERNEL_GP" bias="1"/>
      <Primitive name="POLYNOMIAL_KERNEL_GP" bias="1"/>
      <Primitive name="WEIGHT" bias="1"/>
      <Primitive name="GAMMA" bias="1"/>
      <Primitive name="CONSTANT" bias="1"/>
      <Primitive name="DEGREE" bias="1"/>
      <Primitive name="XY" bias="1"/>
    </PrimitiveSet>
    <PrimitiveSet>
      <Primitive name="DUMMY_C" bias="1"/>
      <Primitive name="C" bias="1"/>
    </PrimitiveSet>
  </PrimitiveSuperSet>
  <Register>
    <Entry key="ec.term.maxgen">12</Entry>
    <Entry key="ec.pop.size">2000</Entry>
    <Entry key="ec.repro.prob">0.05</Entry>
    <Entry key="ec.sel.tournsize">2</Entry>
    <Entry key="ec.elite.keepsize">3</Entry>
    <Entry key="ec.hof.vivasize">5</Entry>

    <Entry key="lg.console.level">3</Entry>
    <Entry key="lg.file.level">3</Entry>

    <Entry key="gp.init.mindepth">2</Entry>
    <Entry key="gp.init.maxdepth">6</Entry>
    <Entry key="gp.tree.maxdepth">9</Entry>
    <Entry key="gp.mutshrink.indpb">0.05</Entry>
    <Entry key="gp.mutstd.indpb">0.15</Entry>
    <Entry key="gp.mutswap.indpb">0.05</Entry>
    <Entry key="gp.mutswap.distrpb">1.00</Entry>
    <Entry key="gp.cx.indpb">0.20</Entry>
    <Entry key="gp.cx.distrpb">0.99</Entry>
    <Entry key="gp.init.mintree">2</Entry>
    <Entry key="gp.init.maxtree">2</Entry>

    <!-- Machine Settings -->
    <Entry key="evokm.machine.prob">0.1</Entry>
    <Entry key="evokm.machine.type">2</Entry>
    <!-- C Parameter Settings -->
    <Entry key="evokm.machine.c.min">-20.0</Entry> <!-- log scale -->
    <Entry key="evokm.machine.c.max">20.0</Entry> <!-- log scale -->
    <Entry key="evokm.machine.c.mutpb">1.0</Entry>
    <Entry key="evokm.machine.c.primitname">C</Entry>
    <!-- K-fold CV Settings -->
    <Entry key="evokm.fold">5</Entry>
    <!-- Kernel Parameter Settings -->
    <Entry key="evokm.kernel.gamma.min">-12.0</Entry> <!-- log scale -->
    <Entry key="evokm.kernel.gamma.max">12.0</Entry> <!-- log scale -->
    <Entry key="evokm.kernel.gamma.mutpb">0.125</Entry>
    <Entry key="evokm.kernel.gamma.primitname">GAMMA</Entry>

    <Entry key="evokm.kernel.degree.min">1</Entry>
    <Entry key="evokm.kernel.degree.max">7</Entry>
    <Entry key="evokm.kernel.degree.mutpb">0.125</Entry>
    <Entry key="evokm.kernel.degree.primitname">DEGREE</Entry>

    <Entry key="evokm.kernel.constant.min">-12.0</Entry> <!-- log scale -->
    <Entry key="evokm.kernel.constant.max">12.0</Entry> <!-- log scale -->
    <Entry key="evokm.kernel.constant.mutpb">0.125</Entry>
    <Entry key="evokm.kernel.constant.primitname">CONSTANT</Entry>

    <Entry key="evokm.kernel.weight.min">0.0</Entry>
    <Entry key="evokm.kernel.weight.max">20.0</Entry>
    <Entry key="evokm.kernel.weight.mutpb">0.125</Entry>
    <Entry key="evokm.kernel.weight.primitname">WEIGHT</Entry>
    <!-- Datasets -->
    <Entry key="evokm.train.file">../dataset/reaching_1.dat.scaled</Entry>

    <!-- Logfile information -->
    <Entry key="lg.file.name">results/raw/log/reaching_1.log</Entry>
    <Entry key="lg.individual.file.name">results/raw/ind/reaching_1_ind.log</Entry>

    <Entry key="ms.write.prefix">results/raw/ms/reaching_1</Entry>
    <Entry key="ms.write.over">1</Entry>
    <Entry key="ms.write.perdeme">0</Entry>
    <Entry key="ms.write.interval">1</Entry>
   </Register>
  </System>
  <Experiment>
   <Tries>10</Tries>
  </Experiment>
</Beagle>
```

Listing B.2: Example configuration file for the EvoKM^{GP} application.

Appendix

C

Paper Version

Evolutionary Optimization of Kernel Machines

Arjan Gijsberts
Man-Machine Interaction Group
Faculty of Electrical Engineering,
Mathematics, and Computer Science
Delft University of Technology
Email: a.gijsberts@student.tudelft.nl

***Abstract*—Kernel Machines are a class of machine learning algorithms that have been shown to outperform many other techniques in various classification and regression problems. This class relies on the fact that non-linear problems can be transformed into linear problems by means of the so-called *kernel trick*, which maps the input space onto a hypothetical, high dimensional feature space. The performance of these algorithms, however, depends to a large extent on the kernel function and hyperparameters that are being used. The selection is traditionally done by an expert using a "trial-and-error" approach. In this paper two automated approaches are presented for the selection of a suitable kernel function and optimal hyperparameters. The first of these two models uses Evolution Strategies to rapidly find optimal hyperparameters. The second model aims to improve the generalization capacity of the machine by evolving complex kernel functions using Genetic Programming. Empirical studies show that our Evolution Strategies approach is able to find competitive hyperparameters, as compared with traditional methods, in less time for regression problems. Classification problems are problematic, due to the discontinuity of the error surface. The Genetic Programming approach, however, is shown to improve the generalization capacity of the machine only marginally. In most practical applications this minor improvement will not justify the high computational requirements of the model.**

I. Introduction

The class of *Kernel Machines* has received a large amount of attention from academics over the last decade. These machine learning techniques are interesting, as they allow the construction of powerful, non-linear classifiers using relatively simple mathematical and computational techniques [1]. It has successfully been applied in fields as diverse as economics, biology, medicine, and robotics. One of the main aspects that contribute to the success of the Kernel Machines is the so-called *kernel trick*. The kernel trick can best be described as an implicit mapping of the input data onto a high dimensional feature space. It allows the machine to operate in a high dimensional space, without the need to explicitly map the data points onto this space. This implicit mapping is done by means of a kernel function, which represents the inner product for the specific hypothetical feature space.

However, the performance of Kernel Machines is highly dependent on a good choice for both the kernel function and its parameters. Unfortunately, there are no analytical methods that can guide the user in selecting an appropriate kernel function and good parameter values. In this paper we propose two models for the automated selection of the parameters and the kernel function itself. These two models are based on techniques that fall in the class of Evolutionary Computation. This class of optimization and learning techniques is inspired by neo-Darwinian evolution [2]. The idea is that the mutation and reproduction of solutions, combined with fitness proportionate selection, will converge toward optimal solutions.

Our first model uses Evolution Strategies to optimize the hyperparameters of a Kernel Machine in a time-efficient manner. The second model tries to increase the generalization performance of a Kernel Machine by constructing complex, problem specific kernel functions using Genetic Programming. Both models will be validated on six benchmark data sets, on which the traditional grid search will be used as a reference.

This paper is organized as follows. In Section II we give an introduction into Kernel Machines and the kernel trick. We will emphasize on one particular type of Kernel Machine, namely the Least Squares Support Vector Machine. In Section III an introduction will be given into Evolutionary Computation in general, and Evolution Strategies and Genetic Programming in particular. A review of related work on hyperparameter optimization and kernel construction is given in Section IV. The two models will be presented in Section V, after which we will present the experimental results in Section VI. The paper is finalized with the conclusions in Section VII.

II. Kernel Machines

The common aspect of all Kernel Machines is the so-called kernel function, which has as goal to improve the performance on non-linear problems. The idea is that a non-linear problem can be made linear by mapping the input data onto a hypothetical, high dimensional feature space. This mapping is not done explicitly, as only the inner product for corresponding feature space is necessary. The kernel functions represent an inner product in a certain feature space. We will demonstrate this so-called kernel-trick at the hand of *Least-Squares Support Vector Machines*, which is one particular type of Kernel Machine.

A. *Least-Squares Support Vector Machines*

Assume that we have a set of ℓ labeled training samples, i.e. $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^{\ell}$, where $\mathbf{x} \in \mathcal{X} \subseteq \mathbb{R}^n$ is an input vector of n features and $y \in \mathcal{Y}$ is the corresponding label. In case $\mathcal{Y}$ denotes a set of discrete classes, e.g. $\mathcal{Y} \subseteq \{-1, 1\}$, then the problem is considered a *classification* problem. On the other hand, if $\mathcal{Y} \subseteq \mathbb{R}$, then we are dealing with a *regression* problem.

The Least Squares Support Vector Machine (*LS-SVM*) aims to construct a linear model [3]

$$f(\mathbf{x}) = \langle \mathbf{x}, \mathbf{w} \rangle + b \ , \tag{1}$$

which is able to predict the output values y. Note that for binary classification purposes it is necessary to additionally apply the sign function. The error in the prediction for each sample is defined as

$$y_i - (\langle \mathbf{x_i}, \mathbf{w} \rangle + b) = \epsilon_i \qquad \text{for } \forall_i \ . \tag{2}$$

The optimization problem in LS-SVM is nearly identical to SVM [4]. The goal is to minimize both the norm of the weight vector $\mathbf{w}$ (i.e. maximizing the margin for a smoother solution) and the sum of the squared errors. In contrast with SVM, LS-SVM uses *equality constraints* for the errors instead of *inequality constraints*. Combining the optimization problem with the equality constraints for the errors in (2), we obtain

$$\begin{aligned} &\text{minimize} \quad \frac{1}{2}\|\mathbf{w}\|^2 + \frac{1}{2} C \sum_{i=1}^{\ell} \epsilon_i^2 \\ &\text{subject to} \quad y_i = \langle \mathbf{x}_i, \mathbf{w} \rangle + b + \epsilon_i \ \forall i \ , \end{aligned} \tag{3}$$

where C is a regularization parameter. This parameter is used to control the emphasis on minimizing either the norm or the errors. A large value C corresponds to assigning a high penalty to the errors, which may result in overfitting. A low value for C may have the opposite effect, i.e. underfitting.

Reformulating this optimization problem as a Lagrangian gives the unconstrained minimization problem

$$\frac{1}{2}\|\mathbf{w}\|^2 + \frac{1}{2} C \sum_{i=1}^{\ell} \epsilon_i^2 - \sum_{i=1}^{\ell} \alpha_i \left(\langle \mathbf{x}_i, \mathbf{w} \rangle + b + \epsilon_i - y_i \right) \ , \tag{4}$$

where $\alpha_i \in \mathbb{R} \ \forall i$. Note that the Lagrange multipliers α_i can be either positive or negative, due to the equality constraints in the LS-SVM algorithm. The optimality conditions for this problem can be obtained by setting all derivates equal to zero. This yields the following set of linear equations:

$$\sum_{j=1}^{\ell} \alpha_j \langle \mathbf{x}_j, \mathbf{x}_i \rangle + b + C^{-1}\alpha_i = y_i \qquad \text{for } \forall i \ . \tag{5}$$

B. Kernel Functions

We can observe that the training samples are only present within the inner products in (5). The so-called kernel trick is that we substitute the inner product with a kernel function, which is defined as

$$k(\mathbf{x}, \mathbf{z}) = \langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle \ . \tag{6}$$

In this definition $\phi(\mathbf{x})$ is the mapping of the input samples into a feature space. However, this mapping does not need to be calculated explicitly, as only the inner product in this feature space is relevant. For example, the inner product that belongs to the mapping $\phi(x_1, x_2) = \left(x_1^2, x_2^2, \sqrt{2}x_1x_2\right)$ is given by $k(\mathbf{x}, \mathbf{z}) = \langle \mathbf{x}, \mathbf{z} \rangle^2$. Kernel functions thus enable us to perform the LS-SVM algorithm – and many other algorithms – in a high dimensional space, without the need to explicitly calculate the position of the training samples in that space.

If we substitute the standard inner product with a kernel function in (5), we obtain the "kernelized" variant

$$\sum_{j=1}^{\ell} \alpha_j k(\mathbf{x}_j, \mathbf{x}_i) + b + C^{-1}\alpha_i = y_i \qquad \text{for } \forall i \ . \tag{7}$$

Usually it is convenient to define a kernel matrix as $\mathbf{K} = \left(k(\mathbf{x}_i, \mathbf{x}_j)\right)_{i,j=1}^{\ell}$, so that the system of linear equations can be rewritten as

$$\begin{bmatrix} \mathbf{K} + C^{-1}\mathbf{I} & \mathbf{1} \\ \mathbf{1}^T & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{\alpha} \\ b \end{bmatrix} = \begin{bmatrix} \mathbf{y} \\ 0 \end{bmatrix} \ . \tag{8}$$

The advantageous aspect of LS-SVM is that the optimization problem can be solved using a matrix inversion. Furthermore, the algorithm is nearly identical (except for the sign function) for both regression and classification problems. After the optimal Lagrange multipliers and bias have been obtained using (8), unseen samples can be predicted using the function

$$f(\mathbf{x}) = \sum_{i=1}^{\ell} \alpha_i k(\mathbf{x}_i, \mathbf{x}) + b \ . \tag{9}$$

1) Conditions for Kernels: An important question is which functions actually correspond to an inner product in some feature space, i.e. which are valid kernel functions. The answer is given by *Mercer's theorem*, which state that the function must be symmetric, continuous, and positive semi-definite [4]. This can be formalized in the following condition (i.e. *Mercer's condition*):

$$\int_{\mathcal{X}\times\mathcal{X}} k(\mathbf{x}, \mathbf{z}) f(\mathbf{x}) f(\mathbf{z})\, d\mathbf{x} d\mathbf{z} \geq 0 \quad \text{for all } f \in L_2(\mathcal{X}) \ . \tag{10}$$

Kernel functions that satisfy these conditions are referred to as *admissible* kernel functions. Following this condition on the kernel function, we can state that the kernel matrix has to be positive semi-definite [5]. Unfortunately, it may not be easy to verify whether a kernel function satisfies Mercer's conditions, nor whether the kernel matrix is positive semi-definite. There are, however, certain functions that have analytically been proven to be admissible. Common kernel functions – for classification and regression purposes – include the polynomial function (11), the RBF function (12), and the Sigmoid function (13). It has to be noted that the Sigmoid kernel function is only admissible for certain parameter values.

$$k(\mathbf{x}, \mathbf{z}) = \left(\langle \mathbf{x}, \mathbf{z} \rangle + c\right)^d \qquad \text{for } d \in \mathbb{N}, \ c \geq 0 \tag{11}$$

$$k(\mathbf{x}, \mathbf{z}) = \exp\left(-\gamma\|\mathbf{x} - \mathbf{z}\|^2\right) \qquad \text{for } \gamma > 0 \tag{12}$$

$$k(\mathbf{x}, \mathbf{z}) = \tanh\left(\gamma \langle \mathbf{x}, \mathbf{z} \rangle + c\right) \qquad \text{for some } \gamma > 0, c \ \geq 0 \tag{13}$$

All these function are parameterized, as to allow for adjustments with respect to the training data. The kernel parameter(s), together with the regularization parameter C, are the *hyperparameters*. We will denote the hyperparameters with $\boldsymbol{\theta}$. The performance of an LS-SVM (or an SVM, for that matter) is *critically dependent* on the choice of good values for the hyperparameters.

Mercer's condition can be used to infer simple operations for the combination of kernel functions, which then also will be admissible. For instance, the (weighted) linear combination of two admissible kernel functions is also admissible. Assume that k_1 and k_2 are admissible kernel functions, then the following combinations will be admissible as well [1]:

$$k(\mathbf{x},\mathbf{z}) = c_1 k_1(\mathbf{x},\mathbf{z}) + c_2 k_2(\mathbf{x},\mathbf{z}) \quad \text{for } c_1, c_2 \geq 0 \tag{14}$$

$$k(\mathbf{x},\mathbf{z}) = k_1(\mathbf{x},\mathbf{z})\, k_2(\mathbf{x},\mathbf{z}) \tag{15}$$

$$k(\mathbf{x},\mathbf{z}) = a k_2(\mathbf{x},\mathbf{z}) \quad \text{for } a \geq 0 \tag{16}$$

The interesting aspect is that these operations allow for the modular construction of kernel functions. In other words, we can construct increasingly more complex kernel functions by recursively applying these operations.

C. Reduced LS-SVM

The full LS-SVM algorithm involves the inversion of a $\ell \times \ell$ matrix. One can imagine that the algorithm becomes computationally intractable for relatively large training, due to the cubic complexity of the inversion step with respect to ℓ. A logical solution is to approximate the full LS-SVM algorithm.

Approximating the algorithm is usually done by sparsifying the training set by means of creating an approximate low-rank kernel matrix $\tilde{\mathbf{K}}$. An elaborate low-rank approximation is to minimize the empirical risk over all samples, but to allow the Lagrange multiplier α_i to be non-zero only at a denoted subset of the training samples [6]. More formally, let us define a subset $S_m \subseteq S$ of the training samples, with size $0 < m \leq \ell$. We can then express the following modified problem:

$$\left(\begin{bmatrix} \mathbf{K}_{m\ell} \\ \mathbf{1}^T \end{bmatrix} \begin{bmatrix} \mathbf{K}_{\ell m} & \mathbf{1} \end{bmatrix} + C^{-1} \begin{bmatrix} \mathbf{K}_{mm} & \mathbf{0} \\ \mathbf{0}^T & 0 \end{bmatrix} \right) \begin{bmatrix} \boldsymbol{\alpha} \\ b \end{bmatrix} = \begin{bmatrix} \mathbf{K}_{m\ell} \\ \mathbf{1}^T \end{bmatrix} \mathbf{y} \ . \tag{17}$$

By taking a subset of only m data points, the matrix for the costly inversion step is reduced to $m \times m$. Furthermore, this specific approximation has the advantage that all ℓ samples are used for training. The only limitation is that only m samples are used to describe the model. It has been shown that selecting the samples at random gives a good balance between the quality of the approximation and the temporal complexity of the subset selection procedure [6]. Furthermore, it seems intuitive that selecting the data points at random from the training set will yield a relatively good approximation of the probability distribution from the original data set.

III. Evolutionary Computation

Several biological inspired techniques have been developed over the years for search, optimization, and machine learning under the collective term *Evolutionary Computation* (*EC*) [2]. The key principle in EC – and evolution theory in general – is that potential solutions are generated, evaluated, and reproduced in an iterative process. During the course of this process, the individuals are subject to certain forms of mutation and can reproduce with a probability proportional to their *fitness*. A selection procedure removes the individuals with a relatively low fitness from the population, so that the more fit ones are more likely to “survive”. This way the process converges to good solutions. Three main branches within EC are *Genetic Algorithms* (*GA*), *Evolution Strategies* (*ES*), and *Genetic Programming* (*GP*). In this paper we will consider the latter two.

EC techniques work with a *population* of *individuals*, where an individual is a potential solution to a problem. This solution is represented in the *genotype* of the individual, which contains one or more *chromosomes*. The physical representation of the genotype in the context of the problem domain is the *phenotype*. Compare this with the distinction between the appearance of a human (i.e. phenotype) and its DNA (i.e. genotype). Over the course of multiple generations the individuals are subject to several operations, which are traditionally mutation, reproduction, and selection. In every generation the selection operation is performed on the parent population, in order to create an intermediate population. The selection of individuals is usually proportional to their fitness, so that strong individuals are more likely to survive. The mutation and reproduction operations are then performed on this intermediate population, in order to create an offspring population. The individuals in the offspring population are then evaluated, i.e. assigned a fitness score according to problem specific fitness function. The last phase in the evolutionary cycle is that the offspring population replaces the current parent population. This routine is continued until a termination condition has been reached.

When applying an EC technique to a problem, it is necessary to decide on a suitable genotype representation, fitness function, and termination condition. Typically, all these are highly dependent on the problem domain.

A. Evolution Strategies

Evolution Strategies (*ES*) are one of the main branches of Evolutionary Computation [7]. ES operates in the realm of the phenotype and uses real-valued representations for the individuals. A possible chromosome for an optimization problem with three parameters could thus be $\mathbf{c} = (x_1, x_2, x_3)$, where the parameters $x_i \in \mathbb{R}$ are the *object parameters*. In contrast, the canonical GA operates in realm of the genotype and usually represents all parameters in a binary bitstring.

There are two main types of ES, namely $(\mu + \lambda)$-ES and (μ, λ)-ES. In these notations μ is the size of the parent population and λ is the size of the offspring population. In $(\mu + \lambda)$-ES the new parent population is chosen from *both* the current parent population and the offspring. In contrast, in (μ, λ)-ES the new parent population is chosen only from the offspring population, which requires that $\lambda \geq \mu$.

The canonical ES relies on the mutation operation for diversifying the genetic material. The mutation operation is typically implemented as a distribution around the individual being mutated. The mutation is done by adding a random value, generated according to the distribution, to the parameter in the chromosome. More formally,

$$x_i' = x_i + \mathcal{N}_i(0, \sigma_i) \ , \tag{18}$$

where $\mathcal{N}$ denotes a logarithmic normal distribution. Note that this mutation mechanism requires the user to specify a standard deviation σ_i for each parameter in the chromosome. These standard deviations are the *strategy parameters* of

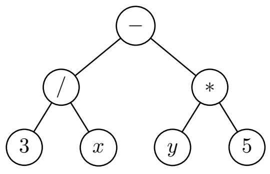


Fig. 1. An example tree representation of a function in GP.

the algorithm. The common approach is to not define these standard deviations explicitly, but to integrate them in the chromosome. This is known as *self adaptation*, as certain parameters of the algorithm are subject to the algorithm itself. An example of a chromosome with three parameters and the additional strategy parameters is $\mathbf{c} = (x_1, x_2, x_3, \sigma_1, \sigma_2, \sigma_3)$. The standard deviations in this mechanism are usually referred to as the *endogenous strategy parameters* [8]. These endogenous strategy parameters are updated according to

$$\sigma_i' = \sigma_i \exp\left(\tau' \mathcal{N}(0,1) + \tau \mathcal{N}_i(0,1)\right) \quad , \tag{19}$$

where $\tau \propto \frac{1}{\sqrt{2\sqrt{m}}}$ and $\tau' \propto \frac{1}{\sqrt{2m}}$ [9]. Note that m is the number of object parameters.

B. Genetic Programming

A vastly different paradigm within EC is that of *Genetic Programming* (*GP*) [10]. GP should be considered rather a form of automated programming than a parameter optimization technique. It aims to solve a problem by breeding a population of computer programs, which – when executed – are direct solutions to the problem. Obviously, this gives much more freedom in the structure of the solutions and it can therefore be applied to wide variety of problems. The common way to represent programs in GP is by means of *syntax trees*. An example of a syntax tree that represents the mathematical function $(3/x) - (y * 5)$ is shown in Fig. 1. Other types of genotype representations, e.g. graphs or linear structures, may be preferred for certain problem domains.

The operations in GP include recombination and mutation operators that are much similar to their GA counterparts. In crossover recombination two parents exchange a part of their genetic material. If the chromosomes are trees, then this means that they swap a sub-tree rooted at a random crossover point. Traditional mutation in GP consists of randomly selecting a mutation point in the tree and replacing the sub-tree rooted at this point with a randomly generated tree.

In the canonical GP algorithm it is not possible to restrict the structure of the syntax tree. For some problems it may be desirable to impose restrictions, so that non-terminals operate only on appropriate data types. Consider, for instance, a binary equality function, which takes two real values as its children and returns a boolean value. *Strongly Typed Genetic Programming* has been proposed as an enhanced version of GP that enforces data type constraints [11]. This influences both the representation of the individuals and the chromosome altering operators. Firstly, while defining the terminal and non-terminal sets the user also has to specify the types of the terminals and of the parameters. Furthermore, the recombination and mutation operators must be altered in such a way that it respects the typing specification, i.e. the syntax structure.

IV. Related Work

A. Hyperparameter Optimization

The common procedure of hyperparameter optimization is a grid search in the parameter space. This means that the machine is optimized by training it on a predefined range of parameter values. The two major drawbacks in this type search process are that it is extremely time consuming and that the method scales exponentially with the number of parameters. Because of these drawbacks, various research has been conducted on efficient optimization of the hyperparameters.

An elaborated analytical technique that has been proposed for hyperparameter optimization is that of *gradient descent* methods [12], [13]. Gradient descent is an analytical minimization method, in which a local minimum is found by taking steps in the negative gradient direction. The approach is demonstrated on a non-spherical RBF function, which means that each features has a distinct scaling factor. Because of this, there will be more hyperparameters than there are features, which allows for demonstrating the scalability of the approach. The gradient descent method is shown to be able to find reasonable hyperparameters more efficiently than grid search. However, the method does require a continuous differentiable kernel function and objective function, which may not be satisfiable for specific types of problems (e.g. non-vectorial kernel functions). Approaches based on pattern search have been proposed to overcome this problem [14]. In pattern search the neighborhood of a parameter vector is investigated in order to *approximate* the gradient empirically. Moreover, the whole class of gradient descent methods has the apparent disadvantage that they can get stuck in local minima.

One of the first mentions of using Evolutionary Computation for hyperparameter optimization can be found in the work of Fröhlich et al. [15], in which GA is primarily used for feature selection. However, the optimization of the regularization parameter C is done in parallel. Other GA-based approaches focus mainly on the optimization of the hyperparameters. These approaches use as objective function either the error on a validation set [16]–[18], the radius-margin bound [19], or k-fold cross validation [20], [21]. Some studies make use of a real-valued variant of GA [22], [23], although it does not become clear whether the real-valued representation performs significantly better than a binary representation. All these studies suggest that GA can easily be applied for hyperparameter optimization. However, there are some caveats, such as heterogeneity of the solutions and the selection of a reliable *and* efficient objective function.

Evolution Strategies have only scarcely been used for hyperparameter optimization [24]. In this approach ES optimizes not only the scaling, but also the orientation of the RBF kernel. An improvement on the generalization performance is achieved over the kernel parameters that were found using grid search. This result should be interpreted with care, as the optimal grid search parameters were used as the initial solutions for the evolutionary algorithm. The classification error on separate test sets was used as the empirical objective function.

The main advantages of the Evolutionary Algorithms is that it usually finds the optimal parameters time efficiently (as compared with grid search) and that the technique scales well with the number of hyperparameters. An advantage of EC optimization when compared with gradient descent methods is that it is more able to cope with local minima. Furthermore, it does not impose restrictions on the kernel and objective functions, such as differentiability.

B. *Combined Kernel Functions*

It seems intuitive that combined kernel functions could improve on the generalization performance, as the implicit feature mapping could be more suited for a specific problem. Several methods have been proposed for the composition of kernel functions. One of the first notions of optimizing a combined kernel were by Lanckriet et al. [25]. This work considers linear combinations of kernels, i.e. $\mathbf{K} = \sum_{i=0}^{m} a_i \mathbf{K}_i$ for $a > 0$ and $\mathbf{K}_i$ chosen from a predefined set of kernel functions. The optimization of the weight factors $\mathbf{a}$ is done using semidefinite programming, which is an optimization method that deals with convex functions over the convex cone of positive semi-definite matrices. This method can be applied to the combination of kernel matrices, since these need to be semi-definite to satisfy Mercer's conditions. However, other methods may be used for the optimization of the weights, such as so-called hyperkernels [26], the Lagrange multiplier method [27], or using a generalized eigenvalue problem [28].

It has been argued that during the combination of kernels some potentially useful information is lost. Lee et al. propose a method for combining kernels that aims to prevent this loss of information [29]. Instead of combining various kernel matrices into one, their method creates a large kernel matrix that contains all original kernel matrices and all other possible mixtures of kernel functions, e.g. $k_{i,j}(\mathbf{x}, \mathbf{z}) = \langle \phi_i(\mathbf{x}), \phi_j(\mathbf{z}) \rangle$, where ϕ_i is the mapping that belongs to kernel function k_i. This removes the requirement to optimize the weight factors, as this is done implicitly by the SVM algorithm. However, special mixture functions need to be provided for the combination of two kernel functions. Furthermore, the spatial and temporal requirements of the algorithm increases drastically, as the kernel matrix is enlarged in both dimensions with the number of kernels in the combination.

Other EC inspired approaches have been proposed to combine kernel functions. Most of these approaches optimize a linear combination of weighted kernels using either GA or ES. The distinction between the methods is in the set of kernel functions that is used and the type of operators between these functions. Some only consider linear combinations (i.e. the addition operator) [30], [31], whilst others may allow both addition and multiplication [32]. The results from these studies suggest that combining kernel functions can improve on the generalization performance of the machine. A limitation of these approaches, however, is that the search is restricted in structure and to a certain number of kernels in the composition.

Howley and Madden propose a method that constructs complete kernel functions using Genetic Programming [33]. In this method a kernel function is evolved for use with an SVM classifier. They use a tree structured genotype, with the operators $+$, $-$, and $\times$ in both scalar and vector variants as the non-terminals. The terminals in their approach are the two vectors $\mathbf{x}$ and $\mathbf{y}$. Since the kernels are constructed using simple arithmetic, they are not guaranteed to satisfy Mercer's condition. Nonetheless, the technique still keeps up with or outperforms traditional kernels for most data sets. The authors emphasize, however, that a technique as GP requires the availability of a sufficiently large dataset. Some enhancements have been proposed by Dioşan et al. Their method differs from the original approach by a richer operator set (e.g. various norms are included) and small changes to certain operators [34]. Similar modifications are presented by Gagné et al., who also use co-evolution to keep the approach computationally tractable [35]. Besides the kernel functions, there are two other species for the training and validation sets. The training set species aims to cooperate with the kernel function on minimizing the error and thus maximizing the fitness, whereas the species for the validation set is competitive and tries to maximize the error of the kernel functions.

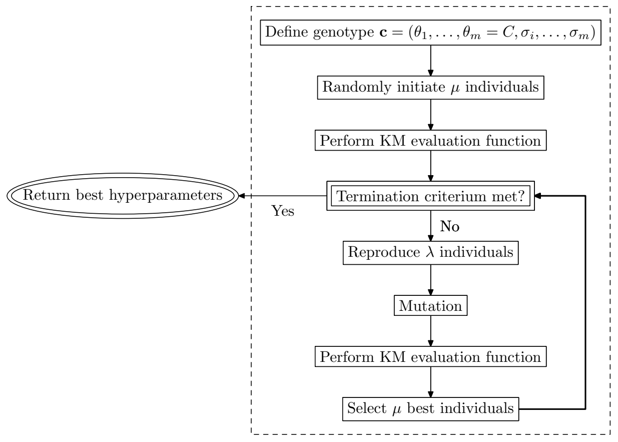
V. Evolutionary Optimization of Kernel Machines

We propose two different models for the evolutionary optimization of the hyperparameters and the kernel function. The first, i.e. EvoKM$^{\text{ES}}$, uses ES to optimize only the hyperparameters for a certain kernel function. The aim is to find optimal hyperparameters more efficiently than using the traditional grid search. Our second model, i.e. EvoKM$^{\text{GP}}$, uses GP to evolve complete kernel functions, with the aim to increase the generalization performance.

A. *EvoKM^{ES}*

In the EvoKM$^{\text{ES}}$ model ES is used to optimize a vector of hyperparameters. Both GA and ES could be used for this problem, but the real-valued genotype representation and operators of ES seem more suited for our type of problem [8]. The chromosomes contain the hyperparameters $\boldsymbol{\theta}$ and the corresponding endogenous strategy parameters $\boldsymbol{\sigma}$, which yields for the RBF kernel the chromosome $\mathbf{c} = [\gamma, C, \sigma_1, \sigma_2]$. ES – and EC techniques in general – are highly generalized algorithms and we can nearly literally employ the technique for our specific problem. This can be confirmed by the overview of the EvoKM$^{\text{ES}}$ model in Fig. 2.

An interesting issue is whether to use $(\mu + \lambda)$-ES or (μ, λ)-ES in the model. Both types have their own specific advantages and disadvantages. Typical application areas of $(\mu + \lambda)$-ES are discrete finite size search spaces, such as combinatorial optimization problems [8]. When the problem is an unbounded search space, typically real-valued search spaces, then (μ, λ)-ES should be preferred [36]. Furthermore, Whitley presents empirical evidence that indicates that (μ, λ)-ES generally



Define genotype $\mathbf{c} = (\theta_1, \ldots, \theta_m = C, \sigma_i, \ldots, \sigma_m)$
Randomly initiate μ individuals
Perform KM evaluation function
Return best hyperparameters
Yes
Termination criterium met?
No
Reproduce λ individuals
Mutation
Perform KM evaluation function
Select μ best individuals

Fig. 2. Graphical overview of the evolutionary hyperparameter optimization model.

$\langle$*kernel*$\rangle$	$\rightarrow$	$\langle$*add_kernels*$\rangle$ | $\langle$*multiply_kernels*$\rangle$ | $\langle$*weighted_kernel*$\rangle$ | $\langle$*polynomial*$\rangle$ | $\langle$*rbf*$\rangle$	
$\langle$*add_kernels*$\rangle$	$\rightarrow$	$\langle$*kernel*$\rangle$ '+' $\langle$*kernel*$\rangle$	
$\langle$*multiply_kernels*$\rangle$	$\rightarrow$	$\langle$*kernel*$\rangle$ '$\times$' $\langle$*kernel*$\rangle$	
$\langle$*weighted_kernel*$\rangle$	$\rightarrow$	a '$\times$' $\langle$*kernel*$\rangle$	for $a \in \mathbb{R}^+$
$\langle$*polynomial*$\rangle$	$\rightarrow$	'$(\langle \mathbf{x}, \mathbf{z} \rangle + c)^d$'	for $d \in \mathbb{N},\ c \in \mathbb{R}^+$
$\langle$*rbf*$\rangle$	$\rightarrow$	'$\exp\left(-\gamma \vert\vert \mathbf{x} - \mathbf{z} \vert\vert^2\right)$'	for $\gamma \in \mathbb{R}^+$

Fig. 3. The context-free grammar – in Backus-Naur form – that constrains the generated expressions for the GP model.

performs better than $(\mu + \lambda)$-ES [37]. We prefer to follow both the heuristic and the empirical indications and adopted (μ, λ)-ES for our model. Unfortunately, because of this decision we cannot guarantee that the search process will converge to a solution, as would have been the case with $(\mu + \lambda)$-ES.

B. EvoKM^{GP}

Our second optimization method constructs complete kernel functions using GP. In this model the functions are represented using syntax trees. The syntactic structure of the trees is based on the combination operations that guarantee admissible kernel functions, c.f. (14), (15), and (16) as the non-terminals and the polynomial and RBF kernels as terminals. This is formalized in the context-free grammar that is shown in Fig. 3. The model makes use of Strongly Typed GP, as it needs to ensure that the syntactic structure is enforced on all the individuals. An example chromosome of a kernel function using the tree representation is shown in Fig. 4. Note that the regularization parameter C is omitted in this figure; it is included in an additional real-valued chromosome.

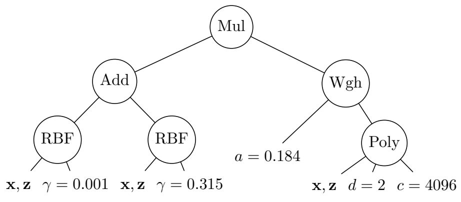
The global overview of EvoKM$^{\text{GP}}$ is shown in Fig. 5. One may notice the similarity – from a high level of abstraction – with the EvoKM$^{\text{ES}}$ model. The first main distinction is the genotype representation, whereas the second lies within the operators that are being used. The following operators are used within the EvoKM$^{\text{GP}}$ model:

1) *Reproduction* occurs with probability $p_r = 0.05$, which means that an individual is directly copied into the offspring population, without any kind of mutation.
2) With probability $p_c = 0.20$ *crossover* recombination occurs, which means that two parents are chosen that exchange a subtree at a random crossover point. The two new individuals are *both* inserted in the offspring population.
3) A *random mutation* occurs with probability $p_m = 0.15$, in which the selected individual is mutated by substituting one of its subtrees with a new random subtree.
4) A *shrink* mutation can happen with probability $p_s = 0.05$. This operator replaces a subtree with one of the branches of this subtree and thus reduces the size of the tree.
5) A *swap* mutation happens with probability p_w. The swap operator replaces a subtree in the individual with another subtree, effectively swapping two branches of the same tree.
6) A *pdf* parameter mutation happens with probability $p_p = 0.5$, in which a real-valued parameter is mutated according to a probability density function.

The latter operator, i.e. the *pdf* mutation, is specially crafted for our specific model. Besides evolving the structure of the kernel functions, it is also important to find appropriate hyperparameters. The common GP operators are only able to mutate these parameters by substituting them for another randomly selected parameter. This strategy may not yield sensible mutations. Mutating these parameters using a probability density function is inspired by ES and guarantees that the parameters are mutated with gradual changes.

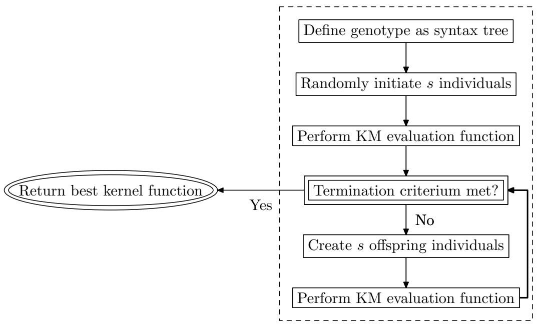
C. Fitness Function

An important decision when applying EC techniques is which fitness function is being used, as this is the actual measure that is being optimized. It should, therefore, measure



Mul
Add
Wgh
RBF
RBF
$a = 0.184$
Poly
$\mathbf{x}, \mathbf{z}$ $\gamma = 0.001$
$\mathbf{x}, \mathbf{z}$ $\gamma = 0.315$
$\mathbf{x}, \mathbf{z}$ $d = 2$ $c = 4096$

Fig. 4. An example of a tree generated by the GP model.



Define genotype as syntax tree
Randomly initiate s individuals
Perform KM evaluation function
Return best kernel function
Yes
Termination criterium met?
No
Create s offspring individuals
Perform KM evaluation function

Fig. 5. Schematic overview of the evolutionary kernel generation model.

the actual concept of "quality" of a solution as close as possible for the given domain. In the context of this study the quality is best described as the generalization performance of the machine. A very important aspect is that the fitness function must prevent overfitting of the machine to the training data. This holds especially for EvoKM^{GP}, as this models adapts not only the hyperparameters, but also the kernel function itself to the data set. There are several methods to estimate this generalization performance, of which cross validation is known to be very generalized. This has as advantage that the same fitness function can be used for various types of Kernel Machines. Furthermore, it can also be used for both classification and regression problems. In *k-fold cross validation* the data set is subdivided in k equal parts. Then the learning machine is trained k times on $k-1$ sets and tested on the single subset that was not included in the training set. The final error is the mean of the errors obtained in the k steps. *Leave-one-out cross validation* is a special case of k-fold cross validation, where k is chosen equal to the size of the data set ℓ. Leave-one-out cross validation is computationally demanding in its standard form, as the machine has to be trained ℓ times on $\ell-1$ samples. However, specialized routines exist for LS-SVM to obtain the leave-one-out error after only a single training on the *entire* data set [38]. Both methods, i.e. k-fold and leave-one-out cross validation, have been shown to be approximately unbiased [39]. However, the k-fold cross validation error usually exhibits a lower variance than the leave-one-out measure. For this reason we have used the k-fold cross validation measure as our fitness function for both the models. Furthermore, the k-fold cross validation measure does not depend on specialized routines for its computational feasibility.

VI. Results

The two models that were described above have been validated experimentally on a set of benchmark problems. The LS-SVM part of the models has been implemented in C++ using the efficient Atlas library for Linear Algebra [40]. The evolutionary aspects of the models, i.e. ES and GP, have been implemented using the OpenBeagle framework for Evolutionary Computation [41].

A. Experimental Setup

All the experiments have been performed with the reduced variant of the LS-SVM kernel machine, which was described in Section II-C. Although this is only one specific type of kernel machine, all the relevant aspects have been kept generalized, as to allow for extension to other types of kernel machines (e.g. SVM). The size of the subset that is used to describe the model is kept at 10% of the total data set. The fitness function is, as explained previously, k-fold cross validation with $k = 5$. This value gives a decent tradeoff between reliability and the computational expenses. For classification problems the error measured is the classification error, which obviously will lie between 0 and 1. The error measure in case of regression problems is the *mean-squared-error* (*MSE*). Two kernel functions have been considered in these experiments. The first is the RBF kernel function (12), which is commonly regarded the "default" choice for kernel machines. The second kernel function is the polynomial function (11).

TABLE I
Basic characteristics of the data sets.

Name	Type	# Samples	# Feat.	Balance
Diabetes	classification	768	8	65.1%/34.9%
Housing	regression	506	13	n/a
Reaching 1	regression	1126	4	n/a
Reaching 2	regression	2534	4	n/a
Reaching 3	regression	2737	4	n/a
Wisconsin	classification	449	9	52.6%/47.4%

B. Data Sets

Six different benchmark data sets have been selected for our experiments. Four of these data sets are regression problems, whereas the remaining two are binary classification problems. Three of the data sets, i.e. *Diabetes*, *Housing*, and *Wisconsin*, are well-known benchmark data sets obtained from the UCI Machine Learning repository [42]. The remaining three data sets, i.e. *Reaching* $1, 2, 3$, are obtained internally from the LiraLab of the University of Genoa. These data sets concern orienting the head of a humanoid robot in the direction of its reaching arm. The features are the values of 4 arm encoders, whereas the outputs are actuator values for 3 head joints. Table I shows certain standard characteristics of the data sets after preprocessing. The exact preprocessing steps that have been performed on the data sets are the following:

1) All the features have been (independently) standardized, i.e. rescaling them to have a zero mean and a unit standard deviation.
2) For regression problems the output values have been standardized in the same manner as the features. For classification problems the labels have been set to $+1$ for positive labels and -1 for negative labels.
3) Duplicate entries have been removed from the data sets.
4) The order of the samples in the data set has been randomized.

C. Results for EvoKM^{ES}

The EvoKM^{ES} model has been verified using the following scenario. First, a very coarse grid search has been performed to identify an interesting region for the parameter ranges for each data set and kernel function. Consecutively, a very dense grid search is performed on this region to establish a reference for our model. For the polynomial kernel function, which has 2 parameters, we have kept the degree fixed at $d = 3$ in order to keep the search tractable. This reference contains a number of evaluations that have been dedicated to grid search[1] and the corresponding minimum error. This minimum error is the

[1]Note that the number of evaluations directly translates into time, as solving the LS-SVM problem is independent of the chosen parameters.

TABLE II
COMPARISON BETWEEN THE GRID SEARCH METHOD AND EVOKM^{ES}. NOTE THAT THE RESULTS OF EVOKM^{ES} ARE AVERAGED OVER 25 RUNS. THE COLUMN *ETT* (*Evaluations To Target*) DENOTES THE NUMBER OF EVALUATIONS THE AVERAGE RUN NEEDS TO REACH THE TARGET ERROR.

		Grid Search		*EvoKM^{ES}*	
Name	**Kernel**	ϵ_{min}	**Eval.**	$\bar{\epsilon}_{min}$	**ETT**
Diabetes	RBF	0.2200	621	0.2238 ± *0.0026*	>621
	Poly	0.2213	777	0.2228 ± *0.0005*	>777
Housing	RBF	0.1676	2793	0.1674 ± *0.0000*	495
	Poly	0.1673	1739	0.1714 ± *0.0237*	>1739
Reaching 1	RBF	0.0683	3185	0.0683 ± *0.0000*	327
	Poly	0.0720	1517	0.0671 ± *0.0001*	39
Reaching 2	RBF	0.0042	561	0.0042 ± *0.0000*	159
	Poly	0.0063	399	0.0044 ± *0.0000*	27
Reaching 3	RBF	0.0019	561	0.0019 ± *0.0000*	135
	Poly	0.0032	399	0.0022 ± *0.0001*	39
Wisconsin	RBF	0.0423	3185	0.0454 ± *0.0022*	>3185
	Poly	0.0401	2337	0.0424 ± *0.0024*	>2337

target for our EvoKM^{ES} model. The aim of these experiments is to investigate how efficient EvoKM^{ES} converges toward the target error.

The EvoKM^{ES} model has been used on the same parameter ranges as the grid search. The only exception to this rule is that for the polynomial kernel we have *not* kept the degree fixed at $d = 3$; instead we have used the range $d = [1, 8]$. This exception is allowed to investigate the scaling properties of the ES-based approach. The size of the parent population has been set to $\mu = 3$, whereas the size of the offspring population is $\lambda = 12$. It has been shown that ES performs best with small population sizes [37]. The evolutionary search is terminated after the same number of evaluations have as used for the grid search.

A juxtaposition of the results of the grid search and EvoKM^{ES} is shown in Table II. These results show that for the three Reaching data sets only a fraction of the evaluations were needed to reach the target error. Furthermore, the polynomial kernel function converges very quickly. This suggests that EvoKM^{ES} scales well with the number of parameters and that it probably “uses” the extra degree of freedom to decrease the error. However, the results on the Diabetes, Housing, and Wisconsin data sets are less positive. EvoKM^{ES} performs poorly on the Housing data set with the polynomial kernel due to an outlier in the results (cf. the variance). Unpredictability of the results is one of the problems when using stochastic optimization methods, such as ES.

The poor results on the Diabetes and Wisconsin data sets have a completely different cause. The convergence of EvoKM^{ES} with the Diabetes data set and the RBF kernel is depicted in Fig. 6. It can be observed that the convergence is not completely smooth, but rather stepwise[2]. This is caused by the fact that this is a classification problem, for which the error surface is *not* continuous, but instead contains plateaus.

[2]Mind that this convergence is the average over 25 runs and therefore already smoother than a single run.

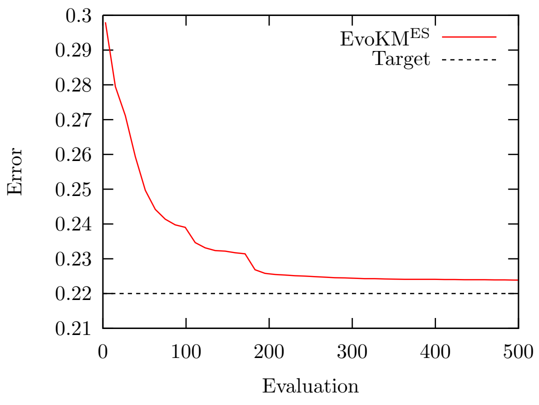


Fig. 6. The convergence of the minimum error during the EvoKM^{ES} search for the Diabetes data set with the RBF kernel function.

Discontinuous error surface are known to be hard for ES-based methods, as it tries to find higher fitness regions (i.e. a lower error) by sampling the error surface in a certain neighborhood. However, many samples (i.e. the offspring individuals) will have a fitness score identical to that of the parent, which means that the search process does not receive any “clues” about the shape of the error surface and starts a random walk on the plateau. Smoothness of the fitness landscape may be regarded a prerequisite of efficient optimization using ES [8]. Nonetheless, we wish to point out that the process still rapidly converges to reasonable solutions, although the target is not reached. Another reason that the target is not reached is that the region with near-optimal solutions is very small for the Diabetes and Wisconsin data sets. This makes it very hard in general to find these regions.

D. *Results for EvoKM^{GP}*

The results from grid search have been used as a reference for EvoKM^{GP} as well. However, for this model we consider solely the quality of the solution and ignore the temporal aspect (i.e. number of evaluations). EvoKM^{GP} has been configured with a population size of 2000 and a maximum of 12 generations. In Table III we can consult the results for both grid search and EvoKM^{GP}. We can observe that the errors obtained using EvoKM^{GP}are only slightly lower than those obtained with grid search. This indicates that the combined kernel functions perform only slightly better than single kernel functions. It is difficult to give any strict interpretations, since the fact that we have not found any combined kernel functions that drastically improve the generalization performance does not necessarily mean that they will not exist at all. We can state, however, that evolving kernel functions is far from a trivial.

One main problem that we encountered during these experiments is the heterogeneity of the optimal solutions. In other words, a particular run of EvoKM^{GP} may yield a drastically different optimal kernel function as compared to a previous run. This seems to be the nature of evolutionary optimization methods such as GA and GP. Instead of finding optimal solutions, they tend to find “good” solutions. Obviously, for

TABLE III
THE MINIMUM ERRORS AS OBTAINED WITH EVOKM^{GP}. NOTE THAT $\bar{\epsilon}_{min}$ INDICATES THE AVERAGE MINIMUM ERROR OVER 10 RUNS, WHEREAS ϵ_{min} INDICATES THE ABSOLUTE MINIMUM ERROR.

	Grid Search	*EvoKM^{GP}*	
Name	ϵ_{min}	$\bar{\epsilon}_{min}$	ϵ_{min}
Diabetes	0.2200	0.2176 ± *0.0032*	0.2096
Housing	0.1673	0.1633 ± *0.0006*	0.1620
Reaching 1	0.0683	0.0592 ± *0.0004*	0.0587
Reaching 2	0.0042	0.0038 ± *0.0000*	0.0037
Reaching 3	0.0019	0.0018 ± *0.0000*	0.0018
Wisconsin	0.0401	0.0358 ± *0.0012*	0.0333

any problem there may exist only one optimal solution, but there can exist many good solutions.

Another difficulty that we experienced in this study is finding a good configuration for our GP method. There are many parameters that need to be set and one has to find a suitable evolver model. Unfortunately, there is no structured approach for optimizing the configuration. Therefore, it remains mostly a task that has to be solved using loose heuristics or even guessing. This problem is especially evident in our context, as the computational demand does not allow for an empirical verification of multiple possible configurations[3].

VII. CONCLUSIONS

In this paper we have presented two models for the evolutionary optimization of Kernel Machines. The distinction between both models was that the first aimed to find optimal hyperparameters more efficiently than traditional methods (i.e. grid search) and the second aimed to increase the generalization performance by means of combined kernel functions. The first model, based on Evolution Strategies, has shown to be an efficient and generalized method of performing hyperparameter optimization for regression problems. Our ES-based model was able to find solutions comparable with optimal grid search solutions in only a fraction of the computational demands. Furthermore, the method scales very well with the number of parameters. Classification problems, on the other hand, are more challenging for our model, because the error surface is discontinuous for this type of problems. ES uses the offspring individuals to sample the neighborhood in order to find a direction that minimizes the error. The plateaus found in the error surface of classification problem interfere with this strategy, as the offspring is likely to have a fitness that is identical to that of the parent. Nonetheless, also for classification problems the model shows a decent convergence rate, which unfortunately halts at a certain non-optimal level.

The Genetic Programming approach for the generation and selection of kernel functions increases the generalization performance of the Kernel Machine only marginally. This suggests that combined kernel functions may not improve the performance as much as one may expect. In most circumstances this slight improvement will not justify the high computational demands of this model. It is difficult to determine the exact cause of these results. The fact that we have not found kernel functions that considerably improve on the generalization performance does not necessarily mean that such kernel functions will not exist at all. It is clear that evolving kernel functions for a Kernel Machine is far from trivial. The configuration of GP, in terms of the evolver model and the parameters, can determine to a great extent the results. However, the vast amount of options make it very difficult to find a configuration that works the best.

[3]The experiments that we presented for EvoKM^{GP} need more than half a year of CPU time on a Pentium 4 class computer running at 3 GHz.

ACKNOWLEDGMENT

The study presented in this paper has been conducted at the Laboratory for Integrated Advanced Robotics of the University of Genoa and the Department of Robotics, Brain, and Cognitive Sciences of the Italian Institute of Technology as part of the *RobotCub* project. This project is funded by the European Commission FP6 Project IST-004370. The author gratefully acknowledges Giorgio Metta and Leon Rothkrantz for their useful comments and supervision.

REFERENCES

[1] J. Shawe-Taylor and N. Cristianini, *Kernel Methods for Pattern Analysis*. Cambridge University Press, June 2004.

[2] D. Whitley, "An overview of evolutionary algorithms: practical issues and common pitfalls," *Information and Software Technology*, vol. 43, no. 14, pp. 817–831, 2001.

[3] J. A. K. Suykens, T. V. Gestel, J. D. Brabanter, B. D. Moor, and J. Vandewalle, *Least Squares Support Vector Machines*. Singapore: World Scientific Publishing Co., Pte, Ltd., 2002.

[4] V. N. Vapnik, *The nature of statistical learning theory*. New York, NY, USA: Springer-Verlag New York, Inc., 1995.

[5] C. J. C. Burges, "A tutorial on support vector machines for pattern recognition," *Data Mining and Knowledge Discovery*, vol. 2, no. 2, pp. 121–167, 1998.

[6] R. Rifkin, G. Yeo, and T. Poggio, "Regularized least squares classification," in *Advances in Learning Theory: Methods, Model and Applications*, vol. 190. Amsterdam: VIOS Press, 2003, pp. 131–154.

[7] H.-G. Beyer, *The theory of evolution strategies*. New York, NY, USA: Springer-Verlag New York, Inc., 2001.

[8] H.-G. Beyer and H.-P. Schwefel, "Evolution strategies a comprehensive introduction," *Natural Computing: an international journal*, vol. 1, no. 1, pp. 3–52, 2002.

[9] H.-P. Schwefel, *Numerical Optimization of Computer Models*. New York, NY, USA: John Wiley & Sons, Inc., 1981.

[10] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.

[11] D. J. Montana, "Strongly typed genetic programming," *Evolutionary Computation*, vol. 3, no. 2, pp. 199–230, 1995.

[12] O. Chapelle, V. N. Vapnik, O. Bousquet, and S. Mukherjee, "Choosing multiple parameters for support vector machines," *Machine Learning*, vol. 46, no. 1–3, pp. 131–159, 2002.

[13] S. S. Keerthi, V. Sindhwani, and O. Chapelle, "An efficient method for gradient-based adaptation of hyperparameters in svm models," in *Advances in Neural Information Processing Systems 19*, B. Schölkopf, J. Platt, and T. Hoffman, Eds. Cambridge, MA, USA: MIT Press, 2007, pp. 673–680.

[14] M. Momma and K. P. Bennett, "A pattern search method for model selection of support vector regression," in *Proceedings of the Second SIAM International Conference on Data Mining*. SIAM, April 2002.

[15] H. Fröhlich, O. Chapelle, and B. Schölkopf, "Feature selection for support vector machines by means of genetic algorithms," in *ICTAI '03: Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*. Washington, DC, USA: IEEE Computer Society, 2003, p. 142.

[16] C.-L. Huang and C.-J. Wang, "A ga-based feature selection and parameters optimization for support vector machines," *Expert Systems with Applications*, vol. 31, no. 2, pp. 231–240, 2006.

[17] S.-H. Min, J. Lee, and I. Han, "Hybrid genetic algorithms and support vector machines for bankruptcy prediction," *Expert Systems with Applications*, vol. 31, no. 3, pp. 652–660, 2006.

[18] S.-Y. Ohn, H.-N. Nguyen, D. S. Kim, and J. S. Park, "Determining optimal decision model for support vector machine by genetic algorithm," in *CIS 2004: First International Symposium on Computational and Information Science*, Shanghai, China, December 2004, pp. 895–902.

[19] Z. Chunhong and J. Licheng, "Automatic parameters selection for svm based on ga," in *WCICA 2004: Fifth World Congress on Intelligent Control and Automation*, vol. 2, June 2004, pp. 1869–1872.

[20] P.-W. Chen, J.-Y. Wang, and H.-M. Lee, "Model selection of svms using ga approach," *Proceedings of the 2004 IEEE International Joint Conference on Neural Networks*, vol. 3, no. 2, pp. 2035–2040, July 2004.

[21] S. A. Rojas and D. Fernandez-Reyes, "Adapting multiple kernel parameters for support vector machines using genetic algorithms," in *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, vol. 1. Edinburgh, Scotland, UK: IEEE Press, September 2005, pp. 626–631.

[22] C.-C. Hsu, C.-H. Wu, S.-C. Chen, and K.-L. Peng, "Dynamically optimizing parameters in support vector regression: An application of electricity load forecasting," in *HICSS '06: Proceedings of the 39th Annual Hawaii International Conference on System Sciences*. Washington, DC, USA: IEEE Computer Society, 2006, p. 30.3.

[23] C.-H. Wu, G.-H. Tzeng, Y.-J. Goo, and W.-C. Fang, "A real-valued genetic algorithm to optimize the parameters of support vector machine for predicting bankruptcy," *Expert Systems with Applications*, vol. 32, no. 2, pp. 397–408, 2007.

[24] F. Friedrichs and C. Igel, "Evolutionary tuning of multiple svm parameters." in *ESANN 2004: Proceedings of the 12th European Symposium on Artificial Neural Networks*, April 2004, pp. 519–524.

[25] G. R. G. Lanckriet, N. Cristianini, P. L. Bartlett, L. E. Ghaoui, and M. I. Jordan, "Learning the kernel matrix with semidefinite programming," *Journal of Machine Learning Research*, vol. 5, pp. 27–72, 2004.

[26] C. S. Ong, A. J. Smola, and R. C. Williamson, "Learning the kernel with hyperkernels," *Journal of Machine Learning Research*, vol. 6, pp. 1043–1071, 2005.

[27] J. Kandola, J. Shawe-Taylor, and N. Cristianini, "Optimizing kernel alignment over combinations of kernels," Department of Computer Science, Royal Holloway, University of London, UK, Tech. Rep. 121, 2002.

[28] J.-T. Sun, B.-Y. Zhang, Z. Chen, Y.-C. Lu, C.-Y. Shi, and W.-Y. Ma, "Ge-cko: A method to optimize composite kernels for web page classification," in *WI '04: Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 299–305.

[29] W.-J. Lee, S. Verzakov, and R. P. W. Duin, "Kernel combination versus classifier combination," in *MCS 2007: Proceedings of the 7th International Workshop on Multiple Classifier Systems*, May 2007, pp. 22–31.

[30] T. Phienthrakul and B. Kijsirikul, "Evolutionary strategies for multi-scale radial basis function kernels in support vector machines," in *GECCO '05: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*. New York, NY, USA: ACM Press, 2005, pp. 905–911.

[31] L. Dioşan, M. Oltean, A. Rogozan, and J. P. Pecuchet, "Improving svm performance using a linear combination of kernels," in *ICANNGA '07: International Conference on Adaptive and Natural Computing Algorithms*, ser. LNCS, no. 4432. Springer, 2007, pp. 218–227.

[32] S. Lessmann, R. Stahlbock, and S. F. Crone, "Genetic algorithms for support vector machine model selection," in *IJCNN '06: International Joint Conference on Neural Networks*. IEEE Press, July 2006, pp. 3063–3069.

[33] T. Howley and M. G. Madden, "The genetic kernel support vector machine: Description and evaluation," *Artificial Intelligence Review*, vol. 24, no. 3-4, pp. 379–395, 2005.

[34] L. Dioşan and M. Oltean, "Evolving kernel function for support vector machines," in *The 17th European Conference on Artificial Intelligence, Evolutionary Computation Workshop*, C. C., Ed., 2006, pp. 11–16.

[35] C. Gagné, M. Schoenauer, M. Sebag, and M. Tomassini, "Genetic programming for kernel-based learning with co-evolving subsets selection," in *Parallel Problem Solving from Nature - PPSN IX*, ser. LNCS, vol. 4193. Reykjavik, Iceland: Springer-Verlag, September 2006, pp. 1008–1017.

[36] H.-P. Schwefel, "Collective phenomena in evolutionary systems," in *Problems of Constancy and Change – The Complementarity of Systems Approaches to Complexity*, vol. 2. International Society for General System Research, 1987, pp. 1025–1033.

[37] D. Whitley, M. Richards, R. Beveridge, and A. da Motta Salles Barreto, "Alternative evolutionary algorithms for evolving programs: evolution strategies and steady state gp," in *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*. New York, NY, USA: ACM Press, 2006, pp. 919–926.

[38] G. C. Cawley, "Leave-one-out cross-validation based model selection criteria for weighted ls-svms," in *IJCNN-2006: Proceedings of the International Joint Conference on Neural Networks*, Vancouver, BC, Canada, July 2006, pp. 1661–1668.

[39] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *International Joint Conference on Artificial Intelligence*, 1995, pp. 1137–1145.

[40] R. C. Whaley and A. Petitet, "Minimizing development and maintenance costs in supporting persistently optimized BLAS," *Software: Practice and Experience*, vol. 35, no. 2, pp. 101–121, February 2005.

[41] C. Gagné and M. Parizeau, "Genericity in evolutionary computation software tools: Principles and case study," *International Journal on Artificial Intelligence Tools*, vol. 15, no. 2, pp. 173–194, April 2006, 22 pages.

[42] A. Asuncion and D. J. Newman, "UCI machine learning repository," 2007. [Online]. Available: http://www.ics.uci.edu/~mlearn/MLRepository.html