Model Construction with Support Vector Machines and Gaussian Processes through Kernel Search

Fredrik Hjorth Bentsen

Department of Mathematics

University of Bergen

Master's thesis in Actuarial Science

June 2019

Acknowledgements

I would like to thank my supervisor Yushu Li for her guidance and advice throughout the project.

I also thank fellow students for their help and useful discussions, and my family for their support.

Abstract

In this thesis, we aim at constructing a framework for kernel searching in both Gaussian process and Support Vector Machines. Choosing the right kernels for these two methods often requires expert knowledge and many people who use these methods do not have enough knowledge of making a good choice at first hand. A system which is automating the choice of kernels could be useful for non-experts and this project carries out an experimental study of how the automatic chosen system can be formulated according to the data structure. Based on our system, we have carried out four empirical analyses: two in regression and two in classification. To evaluate the constructed kernels and final models in the empirical cases, we have followed an experimental and innovative approach instead of a traditional approach. The implementation of the kernel search and model evaluation is explained in detail. More concretely, the data sets we have used in regression are time series data and we have focused on finding models which have reasonable extrapolations. In the classification analyses, we have chosen medical data where we want to find out whether a patient has a disease or not. We have trained the classification models to have good accuracy but also to minimize the type II error of not identifying a patient with the disease.

Table of contents

Chapter 1

Introduction

In the field of data science, we have all kinds of data source in today's information age and different types of data become easier to collect and store. For the statisticians, the availability of huge amount of data along with new scientific problems has reshaped statistical thinking and data analysis. However, in order to make predictions, or create interpretable knowledge of the data, experts with cross-sectional skills in statistics, data science, and machine learning are needed. Thus this thesis aim at combining the knowledge of statistical model constructing, data analysis techniques and machine learning methods to train our skills. We will carry out exploratory study of two important kernel based machine learning methods: SVM and Gaussian Processes. We will present a comprehensive study for the theoretic background of both methods. As both methods are kernel based, this thesis will also study an automatic statistical model building process, which will focus on choosing a suitable set of kernels based on the data structure. This topic of the thesis is inspired by the project: The Automatic Statistician [23], where they aim to explore an open-ended space of possible statistical models and return a statistical model with state-of-the-art extrapolation performance evaluated over real data sets from various domains. In this thesis we will try to design our own automatic statistical learning process through a broad kernel searching procedure. The kernel chosen by the searching method should be most suitable to the data structure. We will carry out four empirical analyses: two in regression and two in classification. In the Automatic Statistician project, the framework of the automatic machine learning is done in combination with Python and MATLAB. For our framework, we write our own **R** codes when designing the kernel searching process as well as looking at a different procedure of how analyzing the data can be done. The code we have written can be found in the GitHub repository at `https://github.com/HjorthBe/Master-thesis-2019` . It seems little work has been done on automatic statistical learning. Moreover, it

could be interesting to look at other ways of evaluating the kernel families, as the approximation of the marginal likelihood of a GP with the Bayesian information criterion done by other works, does not guarantee to find the global optimum for the hyperparameters.

In this thesis, we first study the support vector machines (SVM), as it is one of the most successful classification/regression methods for many applications in big data data analysis. Vladimir Vapnik invented the first version of the pattern recognition algorithm termed Generalized Portrait [25] in the early 1960s. This algorithm was a predecessor of the Support Vector Machines, which he co-invented 30 years later. After his invention, Vapnik started to collaborate with Aleksey Chervonenkis to further develop on the framework of the algorithm. Many of the ideas which are now being developed in the framework of Support Vector Machines was first proposed by Vapnik and Chervonenkis in the 1960s [20].

In the early 1990s, the neural-network community at AT&T Bell laboratories was among the leaders of machine learning research driven by prediction problems [7]. Vapnik joined the team and he and his co-workers developed the Support Vector Machine. This started with the influential paper by Boser et al. [1], which introduced the optimal margin classifier. The paper transformed the field of machine learning with the pioneering use of functional analysis and convex optimization. The success of SVMs on a variety of real-world pattern recognition benchmark problems, has attracted many researchers and engineers from various diciplines to the field of statistical learning theory.

There are many advantages with SVM. Firstly, we can avoid overfitting when we adjust the tuning parameter in the model. Secondly, only training observations that end up as support vectors are necessary for predictions to be made, so computations are done on a substantially smaller part of the data set. Lastly, one trick which is applied in SVM lies at the concept of "kernel mapping". This kind of mapping can be used when real-world data is not separable by a linear hyperplane, but may still have a underlying nonlinear separable boundary.

The SVM has been successfully applied to pattern classification and regression tasks that ranges from bioinformatics: microarray cancer diagnosis, gene selection, to computer vision: face recognition and hand-written character recognition. One aim of this project is to apply SVM in data analysis where we have high dimensional input variables or big datasets. Except for SVM, this thesis will also investigate how the automatic kernel searching process can be used in another kernel based machine learning method: Gaussian process.

Gaussian process models are an alternative to classical machine learning and statistical approaches to learning. Instead of assuming a parametric form for functions, Gaussian process assumes a probabilistic prior over functions. With this assumption, the Gaussian Process has the advantage that it can represent the uncertainty associated with their function representation. Other advantages with the GP model is that we can sample from the prior to get intuitions about the assumptions of the model. Moreover, the GP provides rich modeling capacity. For instance, if we use the Matérn class of covariance functions, then a special case of this class is the Ornstein-Uhlenbeck process with applications in financial mathematics and physics. There has been much interest of Gaussian process in research, it was for instance used to automatically tune the Monte Carlo three search hyperparameters for AlphaGO Zero [21]. Gaussian processes have also been successful in biogeophysical parameter retrieval [2]. Unlike SVM, prediction with Gaussian process is not a recent topic and Rasmussen mentions that for time series analysis, the basic theory can be seen in the works of Wiener and Kolmogorov in the 1940s [19].

If we compare the Support Vector Machine and Gaussian process classifier, we will see that there is a connection between the two kernel machines. In chapter 6 of Rasmussens book [19], it is shown that the MAP solution of the GP classifier and the SVM solution have a close correspondence. The difference of GP and SVM, is that for GP the uncertainty in the unknown function is handled by averaging and not by minimization as in SVM. One benefit of the GP classifier over the SVM is that it provides an output with a probabilistic interpretation.

We will develop our thesis through the following chapters: chapter 2 is the comprehensive study of support vector machine with the underlying mathematical derivation, chapter 3 is the introduction of Gaussian Processes in the framework of Bayesian analysis, chapter 4 is the process of how we design the automatic kernel search including important parts of our self-written **R** code with explanations, and in the final chapter we have conclusion and discussion.

Chapter 2

Support Vector Machines

In this chapter, we will have a study of the mathematical framework of support vector machine. The support vector machine is a supervised learning approach for predicting qualitative responses known as classification and can also be used as regression. For classification, it is a generalization of the maximal margin classifier and an extension of the support vector classifier, and here we begin with the definition for maximal margin classifier.

2.1 The maximal margin classifier

Maximal margin classifier is defined in the form of a hyperplane, and a hyperplane is a flat affine subspace of dimension $p-1$ in a p-dimensional space. It is parameterized by a vector $\beta = (\beta_1, \beta_2, ..., \beta_p)$, and a constant β_0, expressed in the equation $f(x) = x^T\beta + \beta_0 = 0$. If we are in $\mathbb{R}^2$ the hyperplane $f(x) = x^T\beta + \beta_0 = 0$ is a line. If a point x^T in p-dimensional space satisfies $x^T\beta + \beta_0 = 0$, then x^T lies on the hyperplane. Suppose we have a training data which consists of N pairs $(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)$, where each pair has inputs $x_i \in \mathbb{R}^p$ and a class label with one of two values $y_i \in \{-1, 1\}$, for $i = 1, ..., N$. Where the values of y_i is chosen -1 and 1, for convenience of further computation. Given that the data is linear separable, and a hyperplane that separates the data, the separating hyperplane has the property that

$$x_i^T\beta + \beta_0 > 0 \text{ if } y_i = 1,$$

and

$$x_i^T\beta + \beta_0 < 0 \text{ if } y_i = -1.$$

Equivalently, a separating hyperplane which correctly classify the training example also satisfy the property

$$y_i(x_i^T\beta + \beta_0) > 0$$

for all $i = 1, ..., N$. This is the reason for why we chose $y_i \in \{-1, 1\}$. The larger $y_i(x_i^T\beta + \beta_0) > 0$ is, the more confident we are in the prediction of x_i. If our data can be perfectly separated by a hyperplane, then there will exist an infinite number of such hyperplanes. In this case we will have infinite ways that we can shift the hyperplane a small amount up or down, and rotate it clockwise or counterclockwise without making it touch any of the observations.

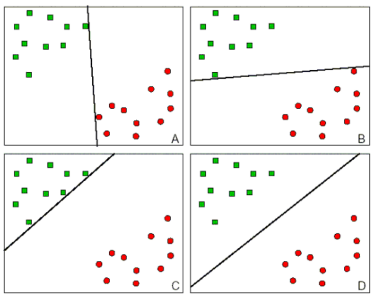

Fig. 2.1 The four plots show observations from two classes and four separating hyperplanes with different decision boundaries. Plot D at the right bottom shows the maximal margin hyperplane.

A good way to decide which of the infinite number of separating hyperplanes to use, is to choose the maximal margin hyperplane. The maximal margin hyperplane separates the two classes and maximizes the distance to the closest point from either class. This provides a unique solution to the separating hyperplane problem. The distance from one point x_i to the hyperplane can be measured by the geometric margin, which is the perpendicular distance from the x_i'th training observation to the separating hyperplane, $\frac{1}{\|\beta\|} y_i f(x_i)$ for all $i = 1, ..., N$, where $\|\beta\|^2 = \sum_{j=1}^{p} \beta_j^2$. We can scale the hyperplane by multiplying any constant C to the parameters (β, β_0), without changing the position of the hyperplane. $f(x) = 0$ for any constant C. By adding the scaling condition $\|\beta\|^2 = \sum_{j=1}^{p} \beta_j^2 = 1$, the geometrical margin then becomes $y_i f(x_i)$ instead. So we have that $m_i = y_i f(x_i)$ is the perpendicular distance of the x_i'th training observation to the separating hyperplane. The margin is then $\mathrm{M} = \min_{i=1,...,\mathrm{N}} m_i$, the shortest distance of the training observation to the hyperplane. Using this notation, the search of the maximal margin hyperplane can be transformed into the optimization problem:

$$\begin{aligned} &\max_{\beta,\beta_0,\|\beta\|=1} \quad \mathrm{M} \\ &\text{subject to} \quad y_i(x_i^T\beta + \beta_0) \geq \mathrm{M}, \quad i = 1, ..., \mathrm{N}. \end{aligned} \tag{2.1}$$

These constraints ensures us that each of the observations is on the correct side of the hyperplane and at least a distance M from the decision boundary. The optimization problem chooses β and β_0 to maximize M. We get rid of the $\|\beta\| = 1$ constraint by replacing the conditions with

$$\frac{1}{\|\beta\|} y_i(x_i^T\beta + \beta_0) \geq \mathrm{M} \tag{2.2}$$

or equivalently

$$y_i(x_i^T\beta + \beta_0) \geq \mathrm{M}\|\beta\|. \tag{2.3}$$

We can arbitrarily set $\|\beta\| = 1/\mathrm{M}$ because any β and β_0 satisfying these inequalities will also have the inequalities satisfied for any positively scaled multiple of them. So now $\mathrm{M} = 1/\|\beta\|$, and we see that $\max \mathrm{M}$ is the same as $\min\|\beta\|$, which again is the same as $\min \|\beta\|^2$. Thus eq.(2.1) is equivalent to

$$\begin{aligned} \min_{\beta,\beta_0} \quad & \frac{1}{2}\|\beta\|^2 \\ \text{subject to } & y_i(x_i^T\beta + \beta_0) \geq 1, \quad i = 1, ..., \mathrm{N}. \end{aligned} \tag{2.4}$$

This is a quadratic optimization problem under linear restriction and for this problem we can construct a Lagrangian [8].

2.2 Lagrange duality

The method of Lagrange multipliers is used to solve an optimization problem subject to equality constraints. Let:

$$\begin{aligned} \min_{\beta} \quad & f(\beta) \\ \text{subject to} \quad & h_i(\beta) = 0, \text{ for} \quad i = 1, ..., l, \end{aligned} \tag{2.5}$$

define the constrained optimization problem of this form. Then the minimum of the constrained function is given by finding the parameter β where $\nabla f(\beta) = -\alpha_i \nabla h_i(\beta)$. We then have $\nabla f(w)$ and $\nabla h_i(w)$ pointing in the same direction and the Lagrange multiplier α_i, which scales the gradient vector to have equal length as the other. Solving this optimization problem we first define the Langrangian to be

$$\mathcal{L}(\beta, \alpha) = f(\beta) + \sum_{i=1}^{l} \alpha_i h_i(\beta). \tag{2.6}$$

We then find and set the $\mathcal{L}$ partial derivatives to zero:

$$\frac{\partial \mathcal{L}}{\partial \beta_i} = 0; \quad \frac{\partial \mathcal{L}}{\partial \alpha_i} = 0, \tag{2.7}$$

and solve for β and α. The method of Lagrange multipliers only allows equality constraints, but if we add the Karush-Kuhn-Tucker (KKT) conditions it can be generalized to allow both inequality and equality constraints:

$$\begin{aligned} \min_{\beta} \quad & f(\beta) \\ \text{subject to} \quad & g_i(\beta) \leq 0, \quad i = 1, ..., k, \\ & h_i(\beta) = 0, \quad i = 1, ..., l. \end{aligned} \tag{2.8}$$

To solve this optimization problem we use the generalized Lagrangian defined as:

$$\mathcal{L}(\beta, \alpha, \lambda) = f(\beta) + \sum_{i=1}^{k} \alpha_i g_i(\beta) + \sum_{i=1}^{l} \lambda_i h_i(\beta)$$

Here, both α_i's and λ_i's are the Lagrange multipliers [16, p. 8]. If we look at the quantity

$$\max_{\alpha:\alpha_i \geq 0} \mathcal{L}(\beta, \alpha, \lambda)$$

for a given β, we see that $\max_{\alpha:\alpha_i \geq 0} \mathcal{L}(\beta, \alpha, \lambda) = \infty$, if β violates any of the primal constraints. This is because the violation now allows $g_i(\beta) > 0$ or $h_i(\beta) \neq 0$. But if, instead the primal constraints are satisfied we see that

$$\max_{\alpha:\alpha_i \geq 0} \mathcal{L}(\beta, \alpha, \lambda) = \max_{\alpha:\alpha_i \geq 0} f(\beta) + \sum_{i=1}^{k} \alpha_i g_i(\beta) + \sum_{i=1}^{l} \lambda_i h_i(\beta) = f(\beta)$$

because then $h_i(\beta) = 0$ and $\max_{\alpha:\alpha_i \geq 0} \mathcal{L}(\beta, \alpha, \lambda)$ is given when $g_i(\beta) = 0$, which is the largest possible value allowed. Hence,

$$\max_{\alpha:\alpha_i \geq 0} \mathcal{L}(\beta, \alpha, \lambda) = \begin{cases} f(\beta)\,, & \text{if } \beta \text{ satisfies the primal constraints} \\ \infty & \text{otherwise.} \end{cases}$$

We can now write $\mathcal{L}_P(\beta) = \max_{\alpha:\alpha_i \geq 0} \mathcal{L}(\beta, \alpha, \lambda)$ and see that minimizing this with respect to β, $\min_{\beta} \mathcal{L}_P(\beta)$, is actually the same primal optimization problem we started with. So the procedure for solving the primal problem is: first we fix β and then we find α to maximize $\mathcal{L}(\beta, \alpha, \lambda)$, defined $\mathcal{L}_p(\beta) = \max_{\alpha:\alpha_i \geq 0} \mathcal{L}(\beta, \alpha, \lambda)$. Then we find β so that $\mathcal{L}_p(\beta)$

is minimized. Moreover, the primal optimization problem can also be viewed from the dual perspective. For the dual problem we instead first fix α and then find β to minimize $\mathcal{L}(\alpha, \beta)$, defined as $\mathcal{L}_D(\alpha) = \min_{\beta} \mathcal{L}(\alpha, \beta)$. The dual optimization problem is then to find α so that $\mathcal{L}_D(\alpha)$ is maximized, $\max_{\alpha:\alpha_i \geq 0} \mathcal{L}_D(\alpha)$. Let $d^* = \max_{\alpha:\alpha_i \geq 0} \mathcal{L}_D(\alpha)$ denote the optimal value of the dual problem and $p^* = \min_{\beta} \mathcal{L}_p(\beta)$, the optimal value of the primal problem. Because "max min" of a function is always less than or equal to the "min max" of a function, the solution of the dual problem provides a lower bound to the solution of the primal problem. So we have that

$$d^* = \max_{\alpha:\alpha_i \geq 0} \mathcal{L}_D(\alpha) = \max_{\alpha:\alpha_i \geq 0} \min_{\beta} \mathcal{L}(w, \alpha, \beta) \leq \min_{\beta} \max_{\alpha:\alpha_i \geq 0} \mathcal{L}(\beta, \alpha, \lambda) = \min_{\beta} \mathcal{L}_P(\beta) = p^*.$$

If the KKT conditions are satisfied, $d^* = p^*$. The KKT conditions are satisfied with the following conditions:

$$\frac{\partial}{\partial \beta_i} \mathcal{L}(\beta^*, \alpha^*, \lambda^*) = 0, \quad i = 1, ..., n \tag{2.9}$$

$$\frac{\partial}{\partial \lambda_i} \mathcal{L}(\beta^*, \alpha^*, \lambda^*) = 0, \quad i = 1, ..., l \tag{2.10}$$

$$\alpha_i^* g_i(\beta^*) = 0 \quad i = 1, ..., k \tag{2.11}$$

$$g_i(\beta^*) \leq 0, \quad i = 1, ..., k \tag{2.12}$$

$$\alpha^* \geq 0, \quad i = 1, ..., k. \tag{2.13}$$

[16].

2.3 Maximal margin classifiers

We can now look at the Lagrange primal for the optimization problem of the maximal margin hyperplane. For the optimization problem:

$$\begin{aligned} &\min_{\beta,\, \beta_0} \quad \frac{1}{2} \|\beta\|^2 \\ &\text{subject to } y_i(x_i^T \beta + \beta_0) \geq 1, \quad i = 1, ..., \mathrm{N}. \end{aligned}$$

we can write the constraint as $g_i(\beta) = -y_i(x_i^T\beta + \beta_0) + 1 \leq 0$. We then have one such constraint for each of the training examples.

We now construct a Lagrangian for this optimization problem and get the Lagrange primal function, to be minimized w.r.t. β and β_0:

$$\mathcal{L}_P(\beta, \beta_0, \alpha) = \frac{1}{2}\|\beta\|^2 - \sum_{i=1}^{N} \alpha_i[y_i(x_i^T\beta + \beta_0) - 1] \tag{2.14}$$

Here we only have "α_i" Lagrange multipliers because the problem only has inequality constraints. Instead of solving (14), the original Lagrange primal of the problem, we find the dual form and solve it.

To find the dual form of the problem we first need to minimize $\mathcal{L}(\beta, \beta_0, \alpha)$ with respect to β and β_0, to get $\mathcal{L}_D$ we then set the derivatives with respect to β and β_0 to zero. We have:

$$\nabla_\beta \mathcal{L}(\beta, \beta_0, \alpha) = \beta - \sum_{i=1}^{N} \alpha_i y_i x_i$$

From this we get

$$\beta = \sum_{i=1}^{N} \alpha_i y_i x_i \tag{2.15}$$

For the derivative with respect to β_0, we obtain

$$\frac{\partial}{\partial \beta_0}\mathcal{L}(\beta, \beta_0, \alpha) = \sum_{i=1}^{N} \alpha_i y_i = 0. \tag{2.16}$$

When substituting the definition of equation (2.15) back into the Lagrangian (2.14), we get

$$\mathcal{L}(\beta, \beta_0, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{k=1}^{N} \alpha_i \alpha_k y_i y_k x_i^T x_k - \beta_0 \sum_{i=1}^{N} \alpha_i y_i$$

But from equation (2.16), the last term must be zero, so we obtain

$$\mathcal{L}(\beta, \beta_0, \alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{k=1}^{N} \alpha_i \alpha_k y_i y_k x_i^T x_k.$$

We now have the Wolfe dual:

$$\begin{aligned} \mathcal{L}_D &= \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{k=1}^{N} \alpha_i \alpha_k y_i y_k x_i^T x_k \\ \text{subject to} \quad \alpha_i &\geq 0 \quad \text{and} \quad \sum_{i=1}^{N} \alpha_i y_i = 0. \end{aligned} \tag{2.17}$$

We finally get the solution for the maximal margin classifier (MMC) by maximimizing $\mathcal{L}_D$, which is a simpler convex optimization problem than the primal. Moreover, if we have found the α's, we see from eq.(17) that all we need in order to make a prediction, is to calculate the inner product between x and the points in the training set. This will be very useful later, when we are able to solve a non-linear classification problem by using kernel functions. The kernel is useful to calculate the inner product in the enlarged feature space.

From the KKT condition (2.11), the solution must satisfy

$$\alpha_i[y_i(x_i^T\beta + \beta_0) - 1] = 0 \quad \forall\, i. \tag{2.18}$$

So we will only have $\alpha_i > 0$ for the training examples that are on the boundary of the slab, which is $y_i(x_i^T\beta + \beta_0) = 1$. We also see that if $y_i(x_i^T\beta + \beta_0) > 1$, then x_i is not on the boundary of the slab and the α_i must be $\alpha_i = 0$. From (2.15) we see that the solution vector β, is a linear combinations of the points x_i that are defined to be on the boundary of the slab, with $\alpha_i > 0$. These points are called support vectors. The coefficient $\beta = \sum_{i=1}^{N} \alpha_i y_i x_i$, is then equal to a much smaller sum $\beta = \sum_{l=1}^{L} \alpha_l y_l x_l$ where x_l is the support vectors and L is the number of support vectors. β_0 is obtained by solving (2.18) for any of the support vectors. Finally the optimal separating hyperplane is the function $\hat{f}(x) = x^T\hat{\beta} + \hat{\beta}_0 = \sum_{l=1}^{L} \alpha_l y_l x_l^T x + \beta_0$ for classying new observations: $\hat{\mathrm{G}}(x) = \text{sign}\, \hat{f}(x)$ [8].

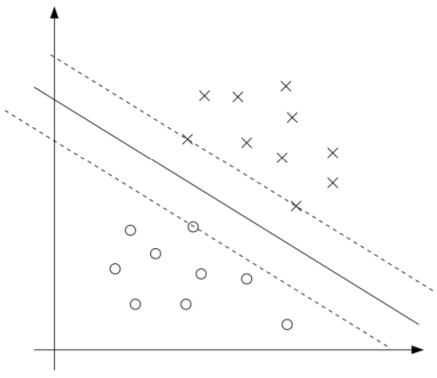

Fig. 2.2 A maximum margin separating hyperplane with three support vectors. Only three α_i's will be used in the optimal solution of the optimization problem. This is much less than the size of the data set [16].

2.4 Support vector classifier

In many cases the data is noisy and not pure linearly separable, so the separating hyperplane does not exist. And even if a separating hyperplane does exist, it may have a margin which is susceptible to outliers. If an observation is added, it will result in a dramatic shift in the maximal margin hyperplane. In this case we can use the support vector classifier, which misclassifies a few training observations in order to do a better job in classifying the remaining observations. We can define the slack variables $\xi = (\xi_1, \xi_2, ..., \xi_N)$ and modify the constraint in eq.(2.1) to $y_i(x_i^T\beta + \beta_0) \geq M(1 - \xi_i)$.

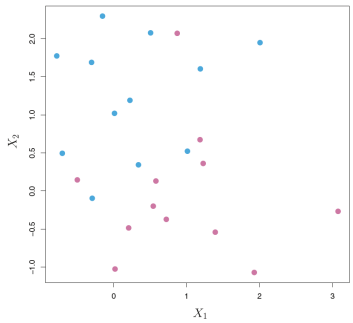

Fig. 2.3 Two classes of observations that are not separable by a separating hyperplane and so the maximal margin classifier can not be used [9].

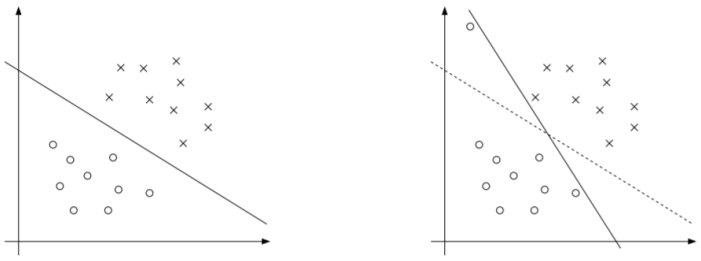

Fig. 2.4 To graphs of almost the same observed inputs except the right graph which has one more extra observation. This new observation changes the maximal margin hyperplane in a very noticeable way [16].

When the slack variables are added, we allow misclassification of difficult examples in a training set which is noisy. The slack variables allow some randomness, so that individual observations can be on the wrong side of the margin and we then avoid overfitting. As a result the model becomes more robust. The slack variables corresponds to the random error in a probability model. The value ξ_i in the constraint, is the proportional amount of the prediction $f(x_i) = x_i^T\beta + \beta_0$ when it is on the wrong side

of the margin. By bounding the sum $\sum \xi_i$, we bound the total proportional amount by which predictions fall on the wrong side of its margin. If $\xi_i > 1$, the ith observations is on the wrong side of the hyperplane. If $\xi_i = 0$, then the ith observation is on the correct side of the margin. If $\xi_i > 0$, then the ith observation is on the wrong side of the margin. Bounding $\sum \xi_i$ at a value K, bounds the total number of training misclassifications at K. When including these slack variables in the optimization problem, we get the support vector classifier with the optimization problem:

$$\begin{aligned} &\max_{\beta,\beta_0,\|\beta\|=1} \quad \mathrm{M} \\ &\text{subject to} \quad y_i(x_i^T\beta + \beta_0) \geq \mathrm{M}(1-\xi_i) \quad \forall\, i, \\ &\xi_i \geq 0, \sum \xi_i \leq \text{constant}. \end{aligned} \tag{2.19}$$

As in the maximal margin optimization problem, we can again get rid of the norm constraint and scale $M = 1/\|\beta\|$ so that the optimization problem is equivalent to

$$\begin{aligned} &\min \|\beta\| \\ &\text{subject to } y_i(x_i^T\beta + \beta_0) \geq 1-\xi_i \quad \forall\, i, \\ &\xi_i \geq 0, \sum \xi_i \leq \text{constant}. \end{aligned} \tag{2.20}$$

It is computationally convient to again re-express the optimization problem as:

$$\begin{aligned} \min_{\beta,\beta_0} \quad & \frac{1}{2}\|\beta\|^2 + C\sum_{i=1}^{N}\xi_i \\ \text{subject to} \quad & \xi_i \geq 0, \quad y_i(x_i^T\beta + \beta_0) \geq 1-\xi_i \quad \forall\, i, \end{aligned} \tag{2.21}$$

where C denotes the constant. As before, the lagrange primal function is then similarly:

$$\mathcal{L}_P = \frac{1}{2}\|\beta\|^2 + C\sum_{i=1}^{N}\xi_i - \sum_{i=1}^{N}\alpha_i[y_i(x_i^T\beta+\beta_0) - (1-\xi_i)] - \sum_{i=1}^{N}\mu_i\xi_i \tag{2.22}$$

which we minimize w.r.t β, β_0 and ξ_i and obtain the Lagrangian dual objective function

$$\mathcal{L}_D = \sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{i'=1}^{N}\alpha_i\alpha_{i'}y_iy_{i'}x_i^Tx_{i'}. \tag{2.23}$$

We maximize $\mathcal{L}_D$ subject to $0 \leq \alpha_i \leq C$ and $\sum_{i=1}^{N}\alpha_i y_i = 0$. The solution for β has the form

$$\hat{\beta} = \sum_{i=1}^{N}\hat{\alpha}_i y_i x_i, \tag{2.24}$$

with nonzero coefficients $\hat{\alpha}_i$ only for those observations i for which the constraints in (9-13) are exactly met. These observations are called the support vectors, since $\hat{\beta}$ is represented in terms of them alone [8]. We can now see that when the dual problem is solved, the following conditions are satisfied:

$$C \geq \alpha_i \geq 0, \tag{2.25}$$
$$y_i(x_i^T\beta + \beta_0) - (1 - \xi_i) \geq 0, \tag{2.26}$$
$$\alpha_i[y_i(x_i^T\beta + \beta_0) - (1 - \xi_i)] = 0, \tag{2.27}$$
$$\mu_i\xi_i = 0, \tag{2.28}$$
$$\xi_i \geq 0, \mu_i \geq 0. \tag{2.29}$$

The support vectors are the observations x_i that either lies on the margin or violates the margin. Only then do we have $\alpha_i > 0$. This follows from the fact that if $\alpha_i > 0$, then from eq.(2.27) we have that $y_i(x_i^T\beta + \beta_0) = 1 - \xi_i$. If $\xi_i = 0$, the i'th observation lie on the margin. If $\xi_i > 0$, the i'th observation lie on the wrong side of the margin. Finally if $\xi_i > 1$, the i'th observation lie on the wrong side of the hyperplane.

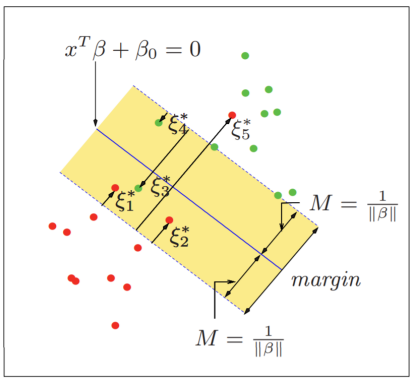

Fig. 2.5 A nonseperable data set and the support vector classifiers. The labeled ξ_i^* points are on the wrong side of their margin. We see that $\xi_3^* > 1$ and $\xi_5^* > 1$, is on the wrong side of the hyperplane. The points which are not labeled are on the correct side with $\xi_i^* = 0$ [8].

2.5 Support Vector Machines

The support vector classifier described so far, finds linear boundaries in the input feature space. This works for linearly separable data sets with some noise, but we get a problem if the data set is very large. At figure 2.3 we see a data set which clearly is not linearly separable, and no linear boundary is possible for any value of C. The cost

parameter C is a tuning parameter, and the value we choose for C decides both the severity and number of violations to the margin, and even to the hyperplane. The basic idea of support vector machine is that the training data of the original input space, with dimension p, can always be mapped to some higher-dimensional feature space where the training set is more linear separable. The data set will be more easily separated in an enlarged feature space that we allow to be larger; however, if the dimension is too large, we will end up with overfitting the data set. If the dimension of the enlarged feature space is $m > \mathrm{N}$, where N is the sample size, we have a totally separated data set with no use of the slack variables. The SVM will find a balance of bias and overfitting when $m \approx \mathrm{N}$ or $m > \mathrm{N}$. The original feature space where the support vector classifier

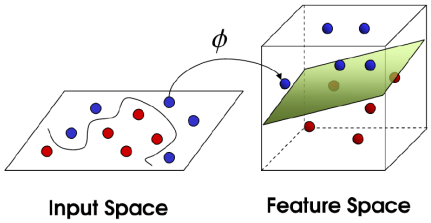

Fig. 2.6 Data in two dimension which is not linearly separable. Applying a transformation to the data, the data becomes linearly separable in three dimensions [10].

finds the optimal hyperplane is spanned by $X = (X_1, X_2, ..., X_p)$. We now instead map the input of the training data into a higher dimension $m > p$ by the transformation $X \rightarrow h(X)$, where $h(X) = (h(X)_1, h(X)_2, ..., h(X)_m)$ denotes the feature mapping. The optimal separating hyperplane is then found in the enlarged feature space spanned by $h(X) = (h(X)_1, h(X)_2, ..., h(X)_m)$. We then have a linear decision boundary in the enlarged space, but a non-linear decision boundary in the original space.

Example: suppose we have a training data which consist of nine pairs $(x_1, y_1), (x_2, y_2), ..., (x_9, y_9)$, where x_i has inputs in the one-dimensional space and y_i belongs to one of two classes, for $i = 1, ..., 9$. Looking at the data to the left in figure 2.7, it is clear that in this situation, even if we allow noise it is not possible to separate the data with SVC. If we let $h_1(x) = x$ and $h_2(x) = x^2$ we can map the input of the training data into a higher dimension $m = 2 > p = 1$ with the transformation $x \rightarrow h(x) = (x, x^2)$. As a result of this mapping, we can in fig. 2.7 see that the data set becomes linear separable in the enlarged future space. The transformation we apply to the observed values are:

$$x = \begin{pmatrix} -6 \\ -5 \\ -4 \\ 1 \\ 2 \\ 3 \\ 5 \\ 6 \\ 7 \end{pmatrix} \rightarrow h(x) = \begin{pmatrix} -6 & 36 \\ -5 & 25 \\ -4 & 16 \\ 1 & 1 \\ 2 & 4 \\ 3 & 9 \\ 5 & 25 \\ 6 & 36 \\ 7 & 49 \end{pmatrix}.$$

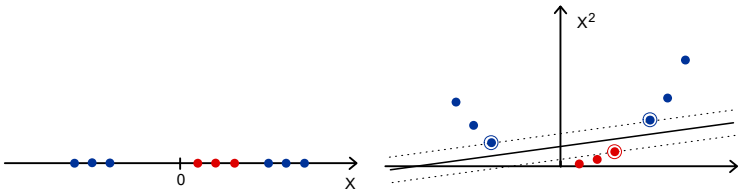

Fig. 2.7 One-dimensional data which becomes linear separable with a nonlinear transformation to two-dimensions.

2.6 Kernel Trick

We have shown that the solution to the support vector classifier problem in eq.(2.21) can be represented by the lagrange dual function, which only involves the inner products of the observations. The inner product of two observations $x_i, x_{i'}$ is given by $\langle x_i, x_{i'} \rangle = \sum_{j=1}^{p} x_{ij} x_{i'j}$. After the transformation, the lagrange dual function has the form

$$\mathcal{L}_D = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{i'=1}^{N} \alpha_i \alpha_{i'} y_i y_{i'} \langle h(x_i), h(x_{i'}) \rangle.$$

From eq.(2.15) we see that the solution function $f(x)$ can be written

$$\begin{aligned} f(x) &= h(x)^T \beta + \beta_0 \\ &= \sum_{i=1}^{N} \alpha_i y_i \langle h(x), h(x_i) \rangle + \beta_0. \end{aligned} \tag{2.30}$$

To compute the coefficients and make a prediction, we only need the inner products $\langle h(x_i), h(x_{i'}) \rangle$. The problem is that the enlarged feature space can be in a very large dimension and in a large dimension the computation of $\langle h(x_i), h(x_{i'}) \rangle$ becomes hard.

In many cases the enlarged feature space is so large that the computations becomes intractable. Fortunately, there is an elegant solution to this problem where we can get the inner product in the enlarge feature space without doing the computations, visiting the enlarged feature space and without knowing the enlarged feature space. The solution is to choose a kernel function $K(x, x') = \langle h(x), h(x') \rangle$. The kernel function is some function that corresponds to an inner product in the enlarged feature space and needs to be a symmetric semi-positive definite function. Some popular choices of kernels to use in SVM are [8]:

$$\begin{aligned} \text{dth-Degree polynomial: } K(x,\, x') &= (1 + \langle x,\, x' \rangle)^d, \\ \text{Radial basis: } K(x,\, x') &= \exp(-\gamma \|x - x'\|^2), \\ \text{Neural network: } K(x,\, x') &= \tanh(\kappa_1 \langle x,\, x' \rangle + \kappa_2). \end{aligned} \tag{2.31}$$

As an example, if we look at the Radial basis kernel, we can show that the kernel corresponds to a inner product in the enlarged feature space and that this space is infinite-dimensional.

Example: we define $\gamma = 1/2$ and look at the Radial basis kernel applied in a one-dimensional space. We then have

$$K(x, x') = \exp\left(-\frac{(x - x')^2}{2} \right) = \exp\left(-\frac{x^2}{2} \right) \exp\left(-\frac{x'^2}{2} \right) \sum_{k=0}^{\infty} \frac{(x)^k (x')^k}{k!},$$

where $\sum_{k=0}^{\infty} \frac{(x)^k (x')^k}{k!}$ is a Taylor expansion of $\exp(xx')$. We can now define the infinite-dimensional feature mapping

$$h(x) = \exp\left(-\frac{x^2}{2} \right) \left(1, x, \frac{x^2}{\sqrt{2!}}, \frac{x^3}{\sqrt{3!}}, \frac{x^4}{\sqrt{4!}}, ... \right)^T.$$

With this mapping we see that

$$K(x, x') = \exp\left(-\frac{(x - x')^2}{2} \right) = \langle h(x), h(x') \rangle.$$

Using the kernel function in SVM, the solution in eq.(2.30) can now be written

$$\hat{f}(x) = \sum_{i=1}^{N} \hat{\alpha}_i y_i K(x, x_i) + \hat{\beta}_0. \tag{2.32}$$

This is known as the kernel trick, where the dot product is simply replaced by the kernel function $K(x, x_i)$.

In SVM, it is much easier in higher dimensions to to find a linearly separable hyperplane. However if the enlarged feature space is too large, we get overfitting which results in high variance for SVM. In this case we can choose a hyperparameter for the given kernel to control the trade-off between variance and bias. For example, the hyperparameter in SVC is C. A good value for C can be choosen with cross validation.

We will now look at how the SVM can be applied to the case of regression and still maintain the properties of the SVM classifier.

2.7 Support Vector Machine for Regression

The Support Vector Machine is not only for classification, it can also be adapted for regression with a quantitative response [26]. Corresponding to the MMC, in ε-SVM regression (SVR) we want to find a linear function in the original space that has at most ε deviation from the observed response values y_i in the training set, and we also want this function to be as flat as possible. Similar to the margin in the classification setting with SVM, we have a ε-tube in SVR, but instead of penalizing observations that are inside the margin, we now penalize errors that are outside the tube. In fig. 2.8, we see the SVR model and observations inside the margin(ε) which are tolerated. With the linear regression model $f(x) = x^T\beta + \beta_0$, flatness is achieved by minimizing $\|\beta\|^2$. This problem can be written as the convex optimization problem [22]:

$$\begin{aligned} &\text{minimize} && \frac{1}{2}\|\beta\| \\ &\text{subject to} && \begin{cases} y_i - x_i^T\beta - \beta_0 \leq \varepsilon \\ x_i^T\beta + \beta_0 - y_i \leq \varepsilon. \end{cases} \end{aligned} \tag{2.33}$$

Similar to the reason for why the slack variables was included in the Support Vector classifier, this optimization problem often does not have a solution or we may want to allow for some errors larger than ε. In the case of no solution, there does not exist any function to satisfy the constraints of residuals having values less than or equal to ε, and so we have to include the slack variables to obtain a solution. The slack variables ξ_i, ξ_i^* allow errors to be larger than ε up to the value of ξ and ξ^*, so the constraints become feasible. In fig. 2.9, we see the SVR model and two observations outside the margin which are allowed by the ξ_i and ξ_i^* variables. The formulation of the optimization

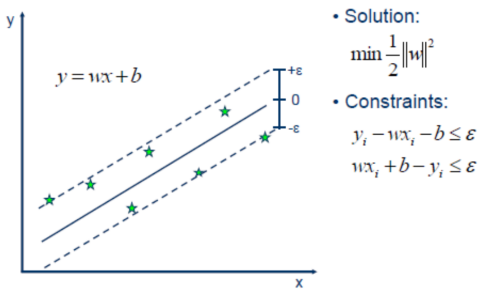

Fig. 2.8 A linear function with all of the observations inside the ε - tube [4].

problem with these slack variables is:

$$\begin{aligned} &\text{minimize} \quad \frac{1}{2}\|\beta\| + C\sum_{i=1}^{l}(\xi_i + \xi_i^*) \\ &\text{subject to} \begin{cases} y_i - x_i^T\beta - \beta_0 \le \varepsilon + \xi_i \\ x_i^T\beta + \beta_0 - y_i \le \varepsilon + \xi_i^* \\ \xi_i, \xi_i^* \ \ge \ 0. \end{cases} \end{aligned} \tag{2.34}$$

Choosing a value for the constant $C > 0$, is a trade off between the flatness of $f(x)$ and the amount of deviations larger than ε which can be tolerated. This means that we have a ε-insensitive loss function L_ε defined as:

$$L_\varepsilon \ = \begin{cases} 0 & \text{if } |y - f(x)| \le \varepsilon \\ |y - f(x)| & \text{otherwise.} \end{cases} \tag{2.35}$$

The ε-loss function sets points within the tube to zero and points outside the tube will be penalized in a linear way.

The optimization problem in ε-SVR can also be formulated as a Lagrange dual and becomes computationally easier to solve. The dual formulation is [22]

$$\begin{aligned} &\text{maximize} \quad -\frac{1}{2}\sum_{i,i'=1}^{N}(\alpha_i - \alpha_i^*)(\alpha_{i'} - \alpha_{i'}^*)\langle x_i, x_{i'}\rangle - \varepsilon\sum_{i=1}^{N}(\alpha_i + \alpha_i^*) + \sum_{i=1}^{N} y_i(\alpha_i - \alpha_i^*) \\ &\text{subject to} \begin{cases} \sum_{i=1}^{N}(\alpha_i - \alpha_i^*) = 0 \\ \alpha_i, \alpha_i^* \ \in \ [0,\, \mathrm{C}]. \end{cases} \end{aligned} \tag{2.36}$$

The solution of the Lagrange dual is $f(x) = \sum_{i=1}^{N}(\alpha_i - \alpha_i^*)\langle x_i, x\rangle + \beta_0$. If there exists obvious nonlinear relationship in the original space, the dot product can be replaced

with a kernel function so that the SV machine is extended for non-linear functions. In fig. 2.10, we see the SVR model which finds a linear relationship of the data in the enlarged feature space, the result is a non-linear function in the original space. Finally, the linear function in the feature space which can be used for a quantitative response, is $f(x) = \sum_{i=1}^{N}(\alpha_i - \alpha_i^*)K(x_i, x) + \beta_0$, where $K(x_i, x) = \langle h(x_i), h(x') \rangle$ is the kernel function.

We will now look at Gaussian processes which is another popular machine learning method. Compared to the SVMs, the Gaussian processes have a more general machine learning methodology in supervised learning.

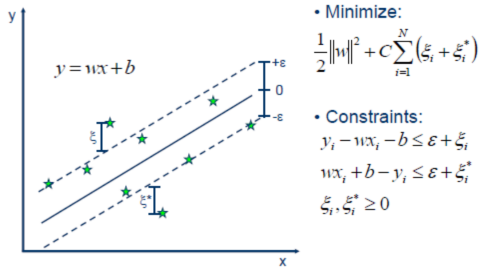

Fig. 2.9 Slack variables are included to allow for some errors with deviation larger than ε, with the result of soft margins [4].

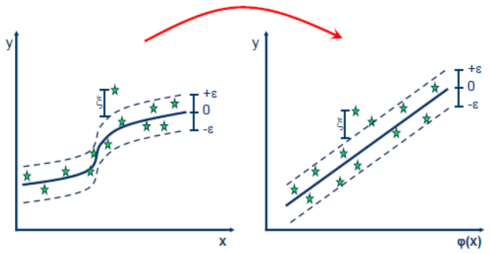

Fig. 2.10 With kernel mapping we can get nonlinear regression. Slack variables are still used in order avoid overfitting [4].

Chapter 3

Gaussian Processes

Together with support vector machines, Gaussian Process models is also a kernel machine which can be used to get models that are both flexible and easy to work with. Compared with the deterministic prediction given by SVM, Gaussian processes can give simple probabilistic representation of random precesses and can be used for many different types of nonparametric estimation. Same as SVM, Gaussian processes can be used in both regression and classification. We will first look at Gaussian process for regression problems where we want to predict continuous quantities.

3.1 Introduction to Gaussian Process

3.1.1 Gaussian Process from linear regression view

In the setting of classic linear regression, we model the output variable y by a function of an input variable x as $y = f(x) + \varepsilon$, where ε is a random error term. We assume that there is a linear relationship, so the function is defined as a linear combination of the inputs $f(x) = \beta_0 + \beta_1 x$. We then use data to estimate the coefficients β_0 and β_1, which is the intercept and slope of the line respectively. The simple linear regression is a parametric model with the advantage that it is easy to implement and interpret. On the other hand, the disadvantage of a parametric model is that we may choose a wrong shape. If the chosen shape of the model is not close to the true form of the unknown f, we will get poor estimates. The simple linear regression model has low flexibility and is not a good model if the relationship between the input and output is not linear.

We also have the Bayesian framework for regression. In this framework we assume that the parameters have a *prior* distribution. The Gaussian Process models can be used to formulate the Bayesian framework for regression, but first we can look at the

Bayesian analysis of the standard linear regression model with Gaussian noise

$$f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w}, \qquad y = f(\mathbf{x}) + \varepsilon, \tag{3.1}$$

where $\mathbf{x}$ is the input vector, $\mathbf{w}$ is a vector of weights, f is the function value and y is the observed target value. The observed values are assumed to differ from the function values by additive noise, which again is assumed to be independent identically distributed as a Gaussian with zero mean and variance σ_n^2

$$\varepsilon \sim \mathcal{N}(0, \sigma_n^2). \tag{3.2}$$

From these assumptions and the linear model we get the *likelihood*, which is the probability density of the observed values given the parameters

$$\begin{aligned} p(y|X, \mathbf{w}) &= \prod_{i=1}^{n} p(y_i|x_i, \mathbf{w}) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_n} \exp\left(-\frac{(y_i - x_i^\top \mathbf{w})^2}{2\sigma_n^2}\right) \\ &= \frac{1}{(2\pi\sigma_n^2)^{n/2}} \exp\left(-\frac{1}{2\sigma_n^2}|y - X^\top \mathbf{w}|^2\right) = \mathcal{N}(X^\top \mathbf{w}, \sigma_n^2 I). \end{aligned}$$

In the Bayesian viewpoint for regression, we express a belief over the parameters before seeing the observations. This is done by defining the *prior* distribution. We set the weights to have a *prior* distribution which is Gaussian with zero mean and covariance matrix Σ_p

$$\mathbf{w} \sim \mathcal{N}(0, \Sigma_p). \tag{3.3}$$

From Bayes' rule we compute the posterior distribution over the weights as

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{marginal likelihood}}, \quad p(\mathbf{w}|\mathbf{y}, X) = \frac{p(\mathbf{y}|X, \mathbf{w})p(\mathbf{w})}{p(\mathbf{y}|X)}. \tag{3.4}$$

We now have the posterior distribution, from this we can do the inference that Bayesian linear model is based on. The marginal likelihood in the denominator is independent of the weights and defined as

$$p(\mathbf{y}|X) = \int p(\mathbf{y}|X, \mathbf{w})p(\mathbf{w})\, d\mathbf{w}. \tag{3.5}$$

Looking at the nominator in eq. (3.4), we see that the posterior contains everything we know about the parameters, combining the likelihood and the prior. We can see that the posterior is Gaussian distributed by writing out the terms from the likelihood

and the prior which depend on the weights:

$$\begin{aligned} p(\mathbf{w}|X,\mathbf{y}) &\propto \exp\Big(-\frac{1}{2\sigma_n^2}(\mathbf{y}-X^\top\mathbf{w})^\top(\mathbf{y}-X^\top\mathbf{w})\Big)\exp\big(-\frac{1}{2}\mathbf{w}^\top\Sigma_p^{-1}\mathbf{w}\big) \\ &\propto \exp\Big(-\frac{1}{2}(\mathbf{w}-\bar{\mathbf{w}})^\top(\frac{1}{\sigma_n^2}XX^\top+\Sigma_p^{-1})(\mathbf{w}-\bar{\mathbf{w}})\Big), \end{aligned}$$

where $\bar{\mathbf{w}} = \sigma_n^{-2}(\sigma_n^{-2}XX^\top + \Sigma_p^{-1})^{-1}X\mathbf{y}$. The Gaussian distribution is recognized with mean $\bar{\mathbf{w}}$ and covariance matrix A^{-1}

$$p(\mathbf{w}|X,\mathbf{y}) \sim \mathcal{N}(\bar{\mathbf{w}} = \frac{1}{\sigma_n^2}A^{-1}X\mathbf{y},\, A^{-1}), \tag{3.6}$$

where $A = \sigma_n^{-2}XX^\top + \Sigma_p^{-1}$. For making predictions, we now average over all the possible parameter values and weight each one with their posterior probability. The predictive distribution is again Gaussian, for f_* at $\mathbf{x}_*$, the prediction is given by

$$\begin{aligned} p(f_*|\mathbf{x}_*,X,\mathbf{y}) &= \int p(f_*|\mathbf{x}_*,\mathbf{w})p(\mathbf{w}|X,\mathbf{y})d\mathbf{w} \\ &= \mathcal{N}(\frac{1}{\sigma_n^2}\mathbf{x}_*^\top A^\top X\mathbf{y},\, \mathbf{x}_*^\top A^{-1}\mathbf{x}_*). \end{aligned} \tag{3.7}$$

We have now shown the Bayesian treatment of the linear model. Next, we show the interpretation of Gaussian process as distribution over functions, and the inference which can be done, directly in the space of functions.

3.1.2 Definition of Gaussian processes

The multivariate Gaussian distribution which has a mean vector $\boldsymbol{\mu}$ and covariance matrix Σ has the joint probability density

$$p(\mathbf{x}|\boldsymbol{\mu}) = (2\pi)^{-D/2}\,|\Sigma|^{-1/2}\,\exp\Big(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})\Big).$$

The Gaussian process is a generalization of this distribution to infinite dimensionality, completely specified by its mean function $m(\mathbf{x})$ and covariance function $k(\mathbf{x},\mathbf{x}')$. Unlike the Gaussian distribution which is a distribution over vectors, the Gaussian process is a distribution over functions

$$f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x},\mathbf{x}')).$$

The indexes of the GP is $\mathbf{x}$. Where the mean function and covariance function of a real process $f(\mathbf{x})$ is defined as

$$\begin{aligned} m(\mathbf{x}) &= \mathbb{E}[f(\mathbf{x})] \\ k(\mathbf{x}, \mathbf{x}') &= \mathbb{E}[(f(\mathbf{x}) - m(\mathbf{x}))(f(\mathbf{x}') - m(\mathbf{x}'))]. \end{aligned}$$

The formal definition of a Gaussian process is

Definition: *A gaussian Process is a collection of random variables, any finite number of which have (consistent) joint Gaussian distributions.*

When we want to compute some quantity that we are interested in, the Gaussian process is an infinite dimensional object and seems uncontrollable, but we actually only need to compute with finite dimensional objects. The following thought is mathematically not correct, but we can think of an infinite long vector as a function. Following this thought the Gaussian Process is a multivariate Gaussian of infinite length.

We now show how we can get an understanding of a GP by going from the process to a distribution, and then draw samples from it. We define a Gaussian process $f \sim \mathcal{GP}(m, k)$ with mean function and covariance function respectively as $m(x) = 0$ and $k(x, x') = \exp(-\frac{1}{2}(x - x')^2)$. Choosing the mean function to be zero is a common choice. We can draw samples from the function f, which is distributed as a GP. The goal of only working with finite quantities, is simply achieved by requiring the values of f at a distinct number of n locations. Given the x-values we can evaluate the GP, which is now reduces to a multivariate Gaussian distribution:

$$\begin{aligned} \mu_i &= m(x_i) = 0, \quad i = 1, ..., n \quad \text{and} \\ \Sigma_{ij} &= k(x_i, x_j) = \exp(-\frac{1}{2}(x_i - x_j)^2), \quad i, j = 1, ..., n. \end{aligned} \tag{3.8}$$

From this distribution we can generate a random vector $\mathbf{f} \sim \mathcal{N}(0, \Sigma)$ and plot the values. Where $\mathbf{f}$ is the finite subset of function values $\mathbf{f} = (f(x_1), f(x_2), ..., f(x_n))^\top$. The reason for why a finite sample of a Gaussian process is a multivariate Gaussian distribution, follows from the marginalization property. We have that if $p(x, y)$ is a joint Gaussian distribution, then $p(x) = \int p(x, y)\, dy$ is also a Gaussian distribution.

We have now seen how to draw random samples from a Gaussian Process, but in real life we want to do something more. We will now look further at how the Gaussian Process can be used for Bayesian regression.

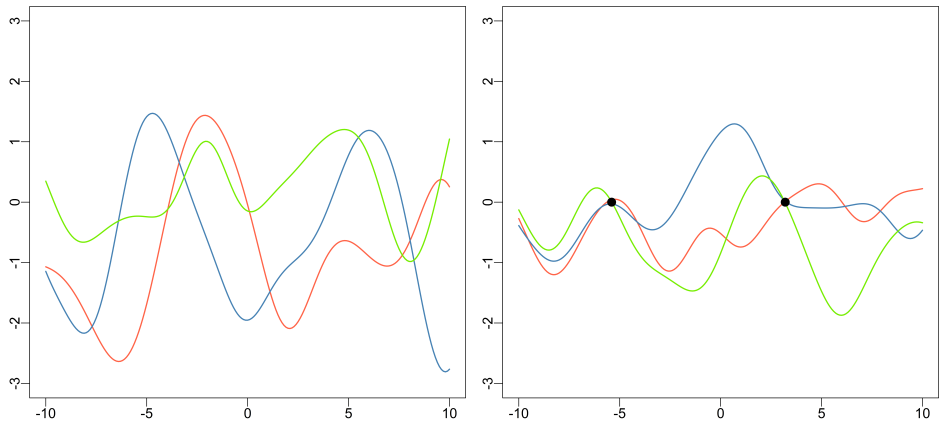

Fig. 3.1 At the plot to the left, we see three functions which are random samples from a GP prior on the finite set of $x = -10$ to $x = 10$. Right: Three functions with the same prior but now conditioned on two observations. These functions are random samples from a posterior distribution.

3.1.3 Prediction with Noise-free observations

Suppose we have a data set of five observed variables. We have the dependent variables $\mathbf{y}$ at five locations of the independent variable $\mathbf{x}$. We now want to estimate a new dependent variable y_* given a new value of x_*. In the most simplest case, we can view the data as noise-free. We then only view the observations as function values $f(\mathbf{x})$, not including the noise that $y = f(\mathbf{x}) + \varepsilon$ has. For the noise-free data, the joint distribution of the training outputs $\mathbf{f}$ and test output $\mathbf{f}_*$ is

$$\begin{bmatrix} \mathbf{f} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K(X,X) & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix} \right).$$

This follows from the idea that our data is a sample from a multivariate Gaussian distribution. We now calculate the covariance function $k(x,x')$ between all possible combinations of the training points and test point. This gives us three matrices

$$K(X,X) = \begin{bmatrix} k(x_1,x_1) & k(x_1,x_2) & \cdots & k(x_1,x_5) \\ k(x_2,x_1) & k(x_2,x_2) & \cdots & k(x_2,x_5) \\ \vdots & \vdots & \ddots & \vdots \\ k(x_5,x_1) & k(x_5,x_2) & \cdots & k(x_5,x_5) \end{bmatrix}$$

$$K(X_*,X) = \begin{bmatrix} k(x_*,x_1) & k(x_*,x_2) & \cdots & k(x_*,x_5) \end{bmatrix} \quad K(X_*,X_*) = k(x_*,x_*).$$

In general, for n training points and n_* testpoints, $K(X,X_*)$ denotes a $n \times n_*$ matrix. For the estimation of the new variable, we are interested in the conditional probability

$p(\mathbf{f}_*|\,\mathbf{f})$, given the observed function values $f(\mathbf{x})$, what is the probability of a prediction $f(\mathbf{x}_*)$? Fortunately, this conditional probability is also Gaussian

$$\mathbf{f}_*|\mathbf{f} \sim \mathcal{N}\Big(K(X_*,X)K(X,X)^{-1}\mathbf{f}, \\ K(X_*,X_*) - K(X_*,X)K(X,X)^{-1}K(X,X_*)\Big).$$

This is the posterior process for the test point x_*. Finally, the best estimate for $f(x_*)$ is the mean from the conditional distribution, $K(X_*,X)K(X,X)^{-1}\mathbf{f}$. Moreover, the uncertainty of the estimate is the variance $K(X_*,X_*)-K(X_*,X)K(X,X)^{-1}K(X,X_*)$. Next, we show how noise is added to the GP regression model.

3.1.4 Prediction using Noisy Observations

In most modelling examples we almost never know the real function values and include noise with the model, $y = f(x) + \varepsilon$, where ε is additive and i.i.d. $\varepsilon \sim \mathcal{N}(0, \sigma_n^2)$. Because the noise is independent, we only need to add a diagonal matrix. With these assumptions we get the joint distribution of the observed target values and the function value at the test data point under the *prior*

$$\begin{bmatrix}\mathbf{y}\\ \mathbf{f}_*\end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix}K(X,X)+\sigma_n^2 I & K(X,X_*)\\ K(X_*,X) & K(X_*,X_*)\end{bmatrix}\right).$$

The equation for prediction in GPR is now the conditional distribution with the noise term

$$\begin{aligned}\mathbf{f}_*|\mathbf{y} &\sim \mathcal{N}\big(\bar{\mathbf{f}}_*,\, \mathrm{cov}(\mathbf{f}_*)\big), \text{ where}\\ \bar{\mathbf{f}}_* &\triangleq \mathbb{E}[\mathbf{f}_*|\mathbf{y}] = K(X_*,X)[K(X,X)+\sigma_n^2 I]^{-1}\mathbf{y},\\ \mathrm{cov}(\mathbf{f}_*) &= K(X_*,X_*) - K(X_*,X)[K(X,X)+\sigma_n^2 I]^{-1}K(X,X_*).\end{aligned} \tag{3.9}$$

The variance in eq. 3.9 does not depend on $\mathbf{y}$ and only $\mathbf{x}$, which is a property of the Gaussian distribution. The variance is expressed as the difference of two terms, where the first term, $K(X_*,X_*)$, is the prior covariance. The second term, which is positive, is representing the information that the observations gives us about the function.

To summarize, in this section we have shown how the Gaussian process method is used for regression problems. Regression and classification problems are problems of supervised learning. Next, we cover the classification problems of supervised learning and describe the Gaussian process for classification.

3.2 Gaussian process in classification

When Gaussian process is used for classification, we want to choose an input pattern $\mathbf{x}$ to one of T classes, $\mathcal{C}_1, ..., \mathcal{C}_T$. Classification problems can be binary where $T = 2$ or multi-class where $T > 2$. We will only focus on the binary classification problem. Moreover, we focus on probabilistic classification where test predictions are class probabilities instead of a guess at the class label.

Looking at the joint probability $p(y, \mathbf{x})$, where y now is the class label, we can discuss different approaches to classification. From Bayes' theorem the joint probability can be written in two ways. We have $p(y)p(\mathbf{x}|y)$, the *generative* approach which models the class conditional distribution. Then we also have $p(\mathbf{x})p(y|\mathbf{x})$, the *discriminative* approach which instead models $p(y|\mathbf{x})$ directly. Both of these approaches have some advantages and disadvantages that can be discussed. However, we will develop the Gaussian process classifier by the *discriminative* approach, which has the advantage of modeling directly what we want, $p(y|\mathbf{x})$.

Somehow we need a way to turn the discriminative approach into a method that is practical and create a model for $p(y|\mathbf{x})$. One way to do this is by using the response function. The response function is a function which we use to give outputs between zero and one for all values of its argument which can lie in the domain $(-\infty, \infty)$. As a result we have outputs that gives us a correct probabilistic interpretation. One common choice for a response function is the linear logistic regression model

$$p(\mathcal{C}_1|\mathbf{x}) = \lambda(\mathbf{x}^\top \mathbf{w}), \quad \text{where} \quad \lambda(z) = \frac{1}{1 + \exp(-z)}. \tag{3.10}$$

Another common choice is the cumulative density function of a standard normal distribution $\Phi(z) = \int_{-\infty}^{z} \mathcal{N}(x|0, 1)dx$. We can now look at a Bayesian approach to logistic regression. Following this we can look at the Gaussian process classifier as a natural next step, similar to what we did for regression.

3.2.1 Linear Models for Classification

We use the same labels as earlier in the SVM chapter, where we labeled $y = +1$ and $y = -1$ for each of the two classes. For the linear model of binary classification the likelihood is

$$p(y = +1|\mathbf{x}, \mathbf{w}) = \sigma(\mathbf{x}^\top \mathbf{w}),$$

where the vector of weight $\mathbf{w}$ is given and $\sigma(z)$ denotes a sigmoid function of any type. We have $p(y = -1|\mathbf{x}, \mathbf{w}) = 1 - p(y = +1|\mathbf{x}, \mathbf{w})$ as the probability of two classes must

sum to one. For a data point $(\mathbf{x}_i, y_i)$, we then see that there are two cases of the likelihood. When $y_i = +1$ the likelihood is $\sigma(\mathbf{x}_i^\top \mathbf{w})$ and for $y_i = -1$ the likelihood is $1 - \sigma(\mathbf{x}_i^\top \mathbf{w})$. The logistic likelihood function is symmetric and can be written more concisely. For the symmetric likelihood functions we have $\sigma(-z) = 1 - \sigma(z)$ and we instead write the likelihood as

$$p(y_i|\mathbf{x}_i, \mathbf{w}) = \sigma(y_i f_i), \tag{3.11}$$

where $f_i \stackrel{\Delta}{=} f(\mathbf{x}_i) = \mathbf{x}_i^\top \mathbf{w}$.

Given a data set $\mathcal{D} = \{(\mathbf{x}_i, y_i)|i = 1, ..., n\}$, the labels are assumed to be generated independently conditional on $f(\mathbf{x})$. When using the same Gaussian prior $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \Sigma_p)$ as we did earlier for regression in eq.(3.3), we get the un-normalized log posterior which is

$$\log p(\mathbf{w}|X, \mathbf{y}) = -\frac{1}{2}\mathbf{w}^\top \Sigma_p^{-1} \mathbf{w} + \sum_{i=1}^{n} \log \sigma(y_i f_i). \tag{3.12}$$

In the case of linear regression with Gaussian noise, the posterior was Gaussian as seen in eq.(3.6). This was because the likelihood times the prior was a Gaussian times a Gaussian which again equals a Gaussian. For classification the likelihood is unfortunately no longer Gaussian, so now the posterior does not have a simple analytic form. With this non-Gaussian likelihood the computations of predictions for Gaussian process classification will not be as straightforward as in the regression case. If we have a training set $\mathcal{D}$ and want to make predictions for a test point $\mathbf{x}_*$, we integrate the prediction $p(y_* = +1|\mathbf{w}, \mathbf{x}_*) = \sigma(\mathbf{x}_*^\top \mathbf{w})$ over the distribution of weights

$$p(y_* = +1|\mathbf{x}_*, \mathcal{D}) = \int p(y_* = +1|\mathbf{w}, \mathbf{x}_*) p(\mathbf{w}|\mathcal{D}) d\mathbf{w}. \tag{3.13}$$

3.2.2 Gaussian Process Classification

In order to get a Gaussian process classification, the following construction is done: we place a GP prior directly on the latent function $f(\mathbf{x})$ and then "squash" the output of the latent function through the logistic function to get a prior on $\pi(\mathbf{x}) \stackrel{\Delta}{=} p(y = +1|\mathbf{x}) = \sigma(f(\mathbf{x}))$. This is the probabilistic classification. We have that π is a deterministic function of f, and from the fact that f is stochastic, π is also stochastic. In fig. 3.2 we see how the latent function is squashed in to the class probability. The GP classification is a generalization of the linear logistic regression model and is similar to the development of linear regression to GP regression that we showed earlier. When

we take the linear function $f(\mathbf{x})$ from the linear logistic model in eq. (3.12), and replace it with a Gaussian process, together with replacing the Gaussian prior on the weights with a Gaussian process prior, we end up with a Gaussian Process classification.

The latent function f acts as a nuisance function, so the values of f itself are not relevant, we do not observe them. The purpose of f is to get a convenient formulation of the model. In order to produce a probabilistic prediction in GP classification, we

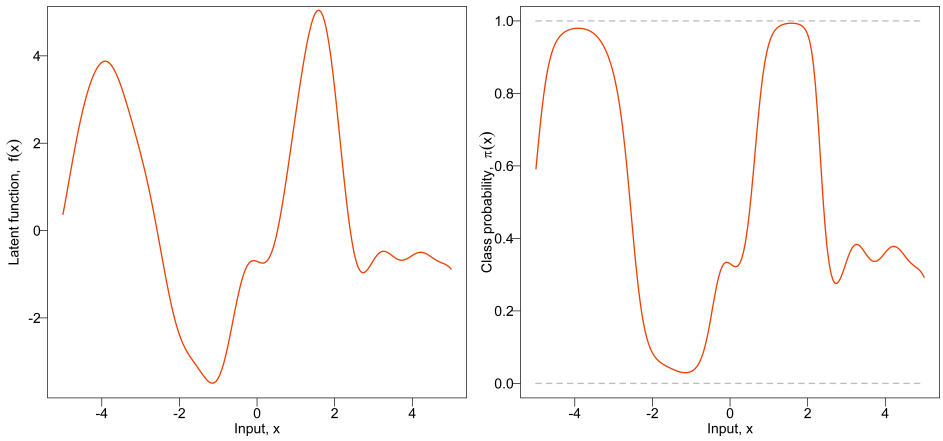

Fig. 3.2 Left: The latent function as a GP which is not constrained to lie between zero and one. Right: The latent function has been squashed using the logistic logit function and gives class probabilities.

first need to compute the distribution of the latent variable corresponding to a test case

$$p(f^*|X, \mathbf{y}, \mathbf{x}_*) = \int p(f^*|X, \mathbf{x}_*, \mathbf{f}) p(\mathbf{f}|X, \mathbf{y})\, d\mathbf{f}, \tag{3.14}$$

where $p(\mathbf{f}|X, \mathbf{y}) = p(\mathbf{y}|\mathbf{f})p(\mathbf{f}|\mathbf{X})/p(\mathbf{y}|X)$ is the posterior over the latent variables. We then use the distribution over the latent f^* which we have computed to produce the probabilistic prediction

$$\bar{\pi}_* \triangleq p(y_* = +1|X, \mathbf{y}, \mathbf{x}_*) = \int \sigma(f_*) p(f_*|X, \mathbf{y}, \mathbf{x}_*)\, df_*. \tag{3.15}$$

In eq.(3.14) there is a non-Gaussian likelihood in the equation which makes the integral intractable. However, we can use analytical approximations of the integral and get the GP classification model with probabilistic prediction.

We will now explain the Laplace approximation which approximate the non-Gaussian joint posterior with a Gaussian one. For the intractable integral, other analytical approximations also exists. However, in the empirical work which is shown later, the

R package which is used for Gaussian process classification uses this method, so here is the explanation for Laplace approximation.

3.2.3 The Laplace Approximation for the Binary GP Classifier

With Laplace's approximation we get the Gaussian approximate $q(\mathbf{f}|X,\mathbf{y})$ to the posterior $p(\mathbf{f}|X,\mathbf{y})$ in the integral eq.(3.14). We obtain the Gaussian approximation by doing a second order Taylor expansion of $\log p(\mathbf{f}|X,\mathbf{y})$ around the maximum of the posterior

$$q(\mathbf{f}|X,\mathbf{y}) = \mathcal{N}(\mathbf{f}|\hat{\mathbf{f}}, A^{-1}) \propto \exp\Big(-\frac{1}{2}(\mathbf{f}-\hat{\mathbf{f}})^\top A(\mathbf{f}-\hat{\mathbf{f}})\Big), \tag{3.16}$$

where $\hat{\mathbf{f}} = \text{argmax}_{\mathbf{f}}\, p(\mathbf{f}|X,\mathbf{y})$ is the maximum of the posterior and $A = -\nabla\nabla \log p(\mathbf{f}|X,\mathbf{y})|_{\mathbf{f}=\hat{\mathbf{f}}}$ is the Hessian of the negative log posterior at that point [19].

We will now explain how to find $\hat{\mathbf{f}}$ and A, which is used in Laplace Approximation. The posterior over the latent variables is from Bayes' rule $p(\mathbf{f}|X,\mathbf{y}) = p(\mathbf{y}|\mathbf{f})p(\mathbf{f}|\mathbf{X})/p(\mathbf{y}|\mathbf{X})$. Since $p(\mathbf{y}|\mathbf{X})$ is independent of $\mathbf{f}$, we do not need it when maximizing w.r.t. $\mathbf{f}$ and we instead consider the un-normalized posterior $p(\mathbf{y}|\mathbf{f})p(\mathbf{f}|\mathbf{X})$. The prior $p(\mathbf{f}|\mathbf{X})$ which is Gaussian, $\mathbf{f}|\mathbf{X} \sim \mathcal{N}(\mathbf{0}, K)$, can be written as

$$\log p(\mathbf{f}|X) = -\frac{1}{2}\mathbf{f}^\top K^{-1}\mathbf{f} - \frac{1}{2}\log|K| - \frac{n}{2}\log 2\pi. \tag{3.17}$$

Taking the logarithm of $p(\mathbf{y}|\mathbf{f})p(\mathbf{f}|\mathbf{X})$ and inserting the expression from eq.(3.17) for the GP prior $p(\mathbf{f}|X)$, we get

$$\begin{aligned}\Psi(\mathbf{f}) &\triangleq \log p(\mathbf{y}|\mathbf{f}) + \log p(\mathbf{f}|X) \\ &= \log p(\mathbf{y}|\mathbf{f}) - \frac{1}{2}\mathbf{f}^\top K^{-1}\mathbf{f} - \frac{1}{2}\log|K| - \frac{n}{2}\log 2\pi\end{aligned} \tag{3.18}$$

Differentiating eq.(3.18) w.r.t. $\mathbf{f}$ we get

$$\nabla\Psi(\mathbf{f}) = \nabla \log p(\mathbf{y}|\mathbf{f}) - K^{-1}\mathbf{f} \tag{3.19}$$

$$\nabla\nabla\Psi(\mathbf{f}) = \nabla\nabla \log p(\mathbf{y}|\mathbf{f}) - K^{-1} = -W - K^{-1}, \tag{3.20}$$

where $W \triangleq -\nabla\nabla \log p(\mathbf{y}|\mathbf{f})$. For the logistic likelihood function, the likelihood $p(\mathbf{y}|\mathbf{f})$ is log concave. In this case the diagonal elements in W are non-negative, and the Hessian in eq.(3.20) is negative definite. This means that $\Psi(\mathbf{f})$ is concave and has a unique maximum which can be found by Newton's method.

At the maximim of $\Psi(\mathbf{f})$, the gradient of $\Psi(\mathbf{f})$ is zero and we can write

$$\nabla\Psi = \mathbf{0} \Rightarrow \hat{\mathbf{f}} = K(\nabla \log p(\mathbf{y}|\hat{\mathbf{f}})). \tag{3.21}$$

Finding the maximum of Ψ, the iteration for Newton's method is

$$\mathbf{f}^{\text{new}} = \mathbf{f} - (\nabla\nabla\Psi)^{-1}\nabla\Psi = \mathbf{f} + (K^{-1} + W)^{-1}(\nabla \log p(\mathbf{y}|\mathbf{f}) - K^{-1}\mathbf{f}) \tag{3.22}$$

With the maximum posterior $\hat{\mathbf{f}}$ being found, we finally have the Laplace approximation to the posterior as a Gaussian with mean $\hat{\mathbf{f}}$ and the covariance matrix as the negative of the inverse Hessian of Ψ from eq.(3.20)

$$q(\mathbf{f}|X, \mathbf{y}) = \mathcal{N}(\hat{\mathbf{f}}, (K^{-1} + W)^{-1}). \tag{3.23}$$

3.2.4 Predictions

The GP predictive mean from eq.(3.9) can be written, with a more compact notation, as

$$\bar{f}_* = \mathbf{k}_*^\top (K + \sigma_n^2 I)^{-1}\mathbf{y}. \tag{3.24}$$

If we combine this equation with eq.(3.21), we can express the posterior mean for f_* under the Laplace approximation as

$$\mathbb{E}_q[f_*|X, \mathbf{y}, \mathbf{x}_*] = k(\mathbf{x}_*)^\top K^{-1}\hat{\mathbf{f}} = k(\mathbf{x}_*)^\top \nabla \log p(\mathbf{y}|\hat{\mathbf{f}}). \tag{3.25}$$

For the variance f_* under the Gaussian approximation, Rasmussen [19] shows that it is

$$\mathbb{V}_q[f_*|X, \mathbf{y}, \mathbf{x}_*] = k(\mathbf{x}_*, \mathbf{x}_*) - \mathbf{k}_*^\top (K + W^{-1})^{-1}\mathbf{k}_*. \tag{3.26}$$

With the mean and variance for f_*, predictions can be made by computing

$$\bar{\pi}_* \simeq \mathbb{E}_q[\pi_*|X, \mathbf{y}, \mathbf{x}_*] = \int \sigma(f_*) q(f_*|X, \mathbf{y}, \mathbf{x}_*)\, df_*, \tag{3.27}$$

where $q(f_*|X, \mathbf{y}, \mathbf{x}_*)$ is Gaussian with mean given by eq.(3.25) and variance given by eq.(3.26). This ends our derivation of the Gaussian process classifier. Compared to the SVM, the GP classifier can give out probabilistic prediction and with this property we are able to analyse the uncertainty of our model.

For both GPs and SVMs, the choice of the kernel is an essential element of the model design. In the next chapter, we present a collection of kernels and discuss different properties they have. Knowing the properties of a kernel function, we will be able to choose a kernel which reflect prior information and captures the structure of the data. We can also view this differently: knowing the properties of a kernel, we can get a better understanding of the data by interpreting the kernel functions which performs good in the model evaluation.

Chapter 4

Automatic machine learning by kernel searching

This chapter will aim at constructing a compositional kernel searching process. The advantage of having a procedure which is able to build kernels at the same level of human experts is the potential to make models such as SVMs and GPs more accessible to non-experts.

4.1 Compositional Kernel Search

In a paper from The Automatic Statistician project by Lloyd et al. [14], a framework for automatic machine learning is presented. Some of the key ideas in this framework are:

1. **An open-ended language of models** expressive enough to capture many of the model composition techniques applied by human statistician to capture real-world phenomena.

2. **A search procedure** to efficiently explore the space of models spanned by the language.

3. **An principled method for evaluating models** in terms of their complexity and their degree of fit to the data.

In our kernel search framework we use the same ideas as above by defining a language of Gaussian process models using a compositional grammar. The space of these models is searched greedily, using a trial and error procedure with the grid search for tuning the hyperparameters while trying to minimize the training and test errors.

4.1.1 Expressing structures through kernels

For the Gaussian process models, the kernel defines the covariance between any two function values: $\mathrm{Cov}(y,\, y') = k(x,\, x')$. The choice of which kernel to use decides which structures that are likely under the GP prior, so choosing a kernel determines the generalization properties of the model. We therefore implicitly define a language of regression models by defining a language of kernels. We will now look at some kernels which are commonly used and go through how these kernels can be composed to express different priors over functions.

For scalar-valued inputs we define the following kernels:

$$k_{\mathrm{SE}}(x, x') = \sigma^2 \exp\Big(- \frac{(x - x')^2}{2\ell^2} \Big) \tag{4.1}$$

$$k_{\mathrm{PER}}(x, x') = \sigma^2 \exp\Big(- \frac{2\sin^2(\pi(x - x')/p)}{2\ell^2} \Big) \tag{4.2}$$

$$k_{\mathrm{LIN}}(x, x') = \sigma^2 (x - \ell)(x' - \ell) \tag{4.3}$$

$$k_{\mathrm{RQ}}(x, x') = \sigma^2 \Big(1 + \frac{(x - x')^2}{2\alpha\ell^2} \Big)^{-\alpha} \tag{4.4}$$

These four kernels are the squared exponential (SE), periodic (PER), linear (LIN) and rational quadratic (RQ) kernels. Each of these kernels give different set of assumptions about the unknown function which we want to model. We will now give an overview of each kernel.

The squared exponential kernel is probably the most widely-used kernel within the kernel machine field [19]. It assumes that the function we model has infinitely many derivatives. It has a lengthscale hyperparameter ℓ which determines the length of the "wiggles" in the function. Moreover, the output variance σ^2 determines the average distance of the function away from its mean. All of the kernels above has this hyperparameter in front as a scale factor. The next kernel is the periodic which has two hyperparameters. The lengthscale determines the length of the "wiggles" in the same way as in the squared exponential kernel. The second hyperparameter p determines the distance between the repetitions of the function. The periodic kernel can be used to model functions which repeat themselves exactly.

The linear kernel is a kernel which is non-stationary and is different from the other three kernels which are stationary. A stationary covariance function is a function

which only depends on the relative positions of its two inputs, and not on their absolute locations. For the non-stationary linear kernel, the hyperparameters are about specifying the origin. The hyperparameter ℓ is an offset which determines the x-coordinate of the point that all the lines in the posterior go through [3].

The last base kernel is the rational quadratic kernel. This kernel can be seen as a scale mixture of the squared exponential kernels with different lengthscales. The limit of the rational quadratic kernel for $\alpha \to \infty$ is the squared exponential kernel [19]. In the rational quadratic kernel the hyperparameter α determines the relative weighting of large-scale and small-scale variations [3]. In fig. 4.1 we see the structures expressible by the four base kernels. We show a plot of each kernel and the function sampled from the GP prior.

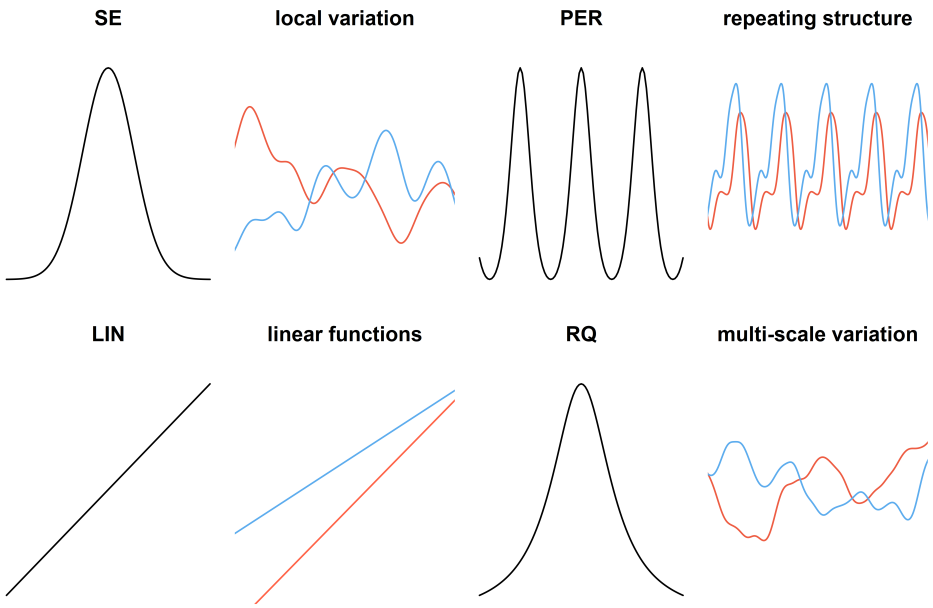

Fig. 4.1 Plot of the base kernel functions with black lines and draws from a the corresponding GP prior with red and blue lines [6].

4.1.2 Combining kernels

For the kernel search, the language of GP models we define has two elements. The first element is the set of base kernels capturing different function properties. The second element is a set of composition rules which combine kernels to provide other valid kernels and express structures that are more rich. These composition rules are

addition and multiplication:

$$(k_1 + k_2)(x, x') = k_1(x, x') + k_2(x, x') \tag{4.5}$$

$$(k_1 \times k_2)(x, x') = k_1(x, x') \times k_2(x, x'). \tag{4.6}$$

We will now give an overview of composite kernels using these operations.

In fig. 4.2 we see examples of different structures expressible by composite kernels. One of the composite kernels is the $k_{\text{SE}} \times k_{\text{PER}}$ kernel. If a squared exponential kernel is multiplied with a periodic kernel we get a periodic function which can slowly vary over time. This composite kernel is useful because most periodic functions does not repeat themselves exactly. When a squared exponential kernel is multiplied or added to a periodic function the result is more flexibility in the model. Moreover, for univariate data multiplying a kernel by a squared exponential gives a way of converting global structure to local structure.

If we multiply a kernel by a linear kernel we get functions with growing amplitude. The third row in fig. 4.2 shows two examples of this. The linear times another linear composite kernel results in quadratic functions. Moreover, the multiplication of linear base kernels can produce Bayesian polynomial regression of any degree. We also have other regression models expressible by sums and products of base kernels. Some of these are listed in table 4.3. In a paper by Duvenaud et al. [6], the following interpretation of how we can view the composite kernels is given. A sum of kernels can be understood as an OR-like operation: two points are considered similar if either kernel has a high value. Similarly, multiplying kernels is an AND-like operation, since two points are considered similar only if both kernels have high values.

We can also construct kernels over multi-dimensional inputs by adding and multiplying between kernels on individual dimensions. The notation for which dimension a kernel operates on is denoted in subscript by an integer. At row four in fig. 4.2 we see examples of composite kernels with multiple dimensions.

The figures and table below gives us instructive ideas of how the kernel should be chosen in the first step when we see the data structure. Thus our method is not totally automatic, the first step is manual searching based on the summaries of the above figures and table. We will now see how we can choose a kernel empirically based on four different data sets: two in regression and two in classification. The code for each of these four analyses can be found in the GitHub repository at `https://github.com/HjorthBe/Master-thesis-2019` .

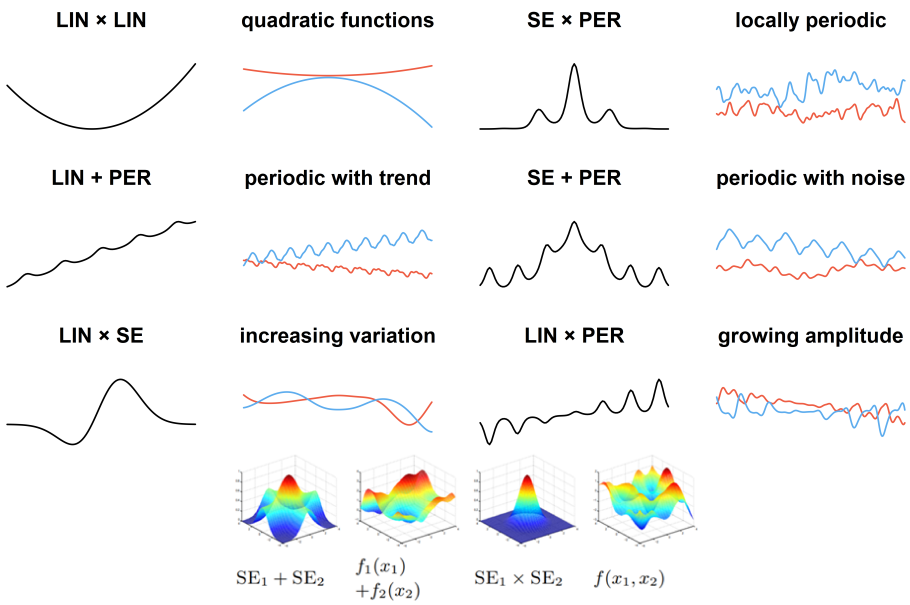

Fig. 4.2 Examples of structures with composite kernels [6]. Similar plot as in fig. 4.1.

Bayesian linear regression	$\textsc{Lin}$
Bayesian polynomial regression	$\textsc{Lin} \times \textsc{Lin} \times \ldots$
Generalized Fourier decomposition	$\textsc{Per} + \textsc{Per} + \ldots$
Generalized additive models	$\sum_{d=1}^{D} \mathrm{SE}_d$
Automatic relevance determination	$\prod_{d=1}^{D} \mathrm{SE}_d$
Linear trend with local deviations	$\textsc{Lin} + \mathrm{SE}$
Linearly growing amplitude	$\textsc{Lin} \times \mathrm{SE}$

Fig. 4.3 Common models that can be expressed by sums and products of one-dimensional base kernels [6].

4.2 Compositional kernel search on Airline passenger data for GP regression

Inspired by Duvenaud et al. [6] paper, we will in this section look at a structural kernel search on the airline passenger data [18] with GP regression. The search is implemented using the kernlab package in **R** for kernel-based machine learning methods [11]. The package has functions for kernel learning algorithms such as SVM and GP, which can be used for both regression and classification. Moreover, the functions implementing the kernel learning algorithms can take kernel functions as an argument. For the kernel functions, the package provides built-in kernels from commonly used kernel families but also the possibility to write user-defined kernel functions. We will now look at the

user-defined kernel functions and go through the framework for the kernel search in detail.

We use the four kernels defined in eq.(4.1) – eq.(4.4) and write a set of user-defined kernel functions. These functions will be the arguments for the gaussian processes function in kernlab, `gausspr`. Each of these four kernel functions has three copies with different indexes on the hyperparameters in each function. In this way a base kernel can be used more than once in the kernel expression and still have different values for the hyperparameters. For instance, given a model with the kernel expression $k_{\text{SE}} + k_{\text{SE}} + k_{\text{SE}}$, the base kernels in the expression can be set to have three different lengthscale values `ell_se_0`, `ell_se_1` and `ell_se_2`, belonging to the kernel function `k_se_0`, `k_se_1` and `k_se_2`, respectively. One thing to note in the code below, is that the notation for x', as in the kernel $k(x, x')$, is notated y, not to be confused with the output variable.

Listing 4.1 Base kernel functions

```
# three squared exponential (SE) base kernel functions:
k_se_0 <- function(x, y){(sigma_var)^2*exp(-0.5*(1/(ell_se_0)^2)*sum((x - y)^2))}
    ⋮
# three periodic (PER) base kernel functions:
k_per_0 <- function(x, y){(sigma_var)^2*exp(-2*
(sin(pi*sum(x-y)/period_0)^2)/ell_per_0^2)}
    ⋮
# three linear (LIN) base kernel functions:
k_lin_0 <- function(x, y){(sigma_var)^2*(sum((x-ell_lin_0)*(y-ell_lin_0)))}
    ⋮
# three rational quadratic (RQ) base kernel functions:
k_rq_0 <- function(x, y){(sigma_var)^2*
(1+sum(x-y)^2/(2*alpha_0*ell_rq_0^2))^(-alpha_0)}
    ⋮
```

In order to compute sums and products of the base kernels we create a set of lists where each list has the operations of summation and multiplication as functions. In **R** it is possible to use the backtick ‘ to refer to functions that have illegal names:

```
operations_1 <- c(‘+‘, ‘*‘)
operations_2 <- c(‘+‘, ‘*‘)
```

By choosing which elements to use from these lists, we also choose which operations to use on the base kernels in the search.

```
> operations_1[[1]](7,13)# summation using the first element
[1] 20
> operations_1[[2]](7,13)# product, now using the second element instead
[1] 91
```

The base kernel functions are also stored in a list with the same purpose, we decide which base kernels to use at the different stages of the search by choosing which element to use from the different lists.

Listing 4.2 Lists containing the base kernels functions

```
kernels_0 <- c(‘k_se_0‘, ‘k_per_0‘, ‘k_lin_0‘, ‘k_rq_0‘, ‘k_none‘)
kernels_1 <- c(‘k_se_1‘, ‘k_per_1‘, ‘k_lin_1‘, ‘k_rq_1‘, ‘k_none‘)
kernels_2 <- c(‘k_se_2‘, ‘k_per_2‘, ‘k_lin_2‘, ‘k_rq_2‘, ‘k_none‘)
```

For the air passenger data we will run a search that has two stages. At the first stage we choose two kernels, one from the `kernels_0` list and the second from the `kernels_1` list. At the second stage, we choose a third kernel from the `kernels_2` list. If the third kernel is a periodic kernel, the periodic kernel function is called from the second element in the `kernels_2` list:

```
> kernels_2[[2]]
function (x, y){(sigma_var)^2*
exp(-2 * (sin(pi * sum(x - y)/period_2)^2)/ell_per_2^2)}
attr(,"class")
[1] "kernel"
```

We now have two types of lists, one which has the operations and the other which has the base kernels. In the kernel search we need a main kernel function where the sums and products of different kernels can be chosen. To create this kernel function we write two new kernel functions which will use the elements in the `kernels_` and `operations_` lists. These functions are the `k_composition_1` function and the `k_composition_2` , where `k_composition_1` will be a nested function defined within the `k_composition_2` as the main function. As a result, `k_composition_2` can generate any kernel expressions which is defined by the space of the base kernel grammer and with three base kernels at most.

Listing 4.3 Kernel composition functions

```
# nested kernel functions,
# the first base kernel is multiplied or summed with the second base kernel
k_composition_1 <- function(x, y){
    u = operations_1[[o_1]](kernels_0[[k_0]](x, y), kernels_1[[k_1]](x, y))
```

```
    return(u)}
# main kernel function which is the argument for the gausspr function:
k_composition_2 <- function(x, y){
    u = operations_2[[o_2]](k_composition_1(x, y), kernels_2[[k_2]](x, y))
    return(u)}
```

For the kernel search it is important that we gradually can add a base kernel to the kernel expression. This is because we do not want a model which is too complex with too many base kernel and also not a model which is too simple with too few base kernels. In the kernel lists `kernels_0, kernels_1` and `kernels_2`, the fifth kernel, `k_none`, is used to decide how many base kernels we want in the kernel expression and is not part of the base kernel grammar.

```
k_none <- function(x, y){
  1
}
```

The main kernel function `k_composition_2` gives a kernel expression with three base kernels when the numeric value for the variables `k_0`, `k_1` and `k_2` are any integers from one to four. If we instead set one of these three variables equal to five, the `k_none` function is chosen and the kernel expression will have one base kernel less. In this way we can add or remove a base kernel in the kernel expression.

We will now train a gaussian process for the air passenger data with `k_composition_2` as the kernel function and search for a kernel expression using a greedy search. For our kernel search we will run two sets of for-loops, one for each stage of the search. At the first stage, the first for-loops will find a kernel expression with two base kernels. At the second stage, the next for-loops will expand the kernel expression with a third base kernel, potentially improving the model. Before we run these for-loops we set the following default values for the variables in the `k_composition_2` kernel function:

```
for(i in 1:2){# only products as operations
  assign(paste0("o_", i), 2)}
for(i in 1:3){# base kernels are set to k_none
  assign(paste0("k_", i-1), 5)}
```

This gives the variable outputs:

```
> o_1; o_2# elements in operations lists
[1] 2
[1] 2
> k_0; k_1; k_2# elements in kernel lists
[1] 5
```

```
[1] 5
[1] 5
```

We now have the kernel expression $k(x, x') = 1 \times 1 \times 1$. In order to search through the different combinations of the two kernels at the first stage, we create the table `k_comb` which has a set of numbers representing the different combinations of the kernel pairs we want to evaluate. At each iteration in the for-loop, a new pair of numbers for `k_0` and `k_1` is provided at the current row in the table. The `k_0` and `k_1` variables are the elements in the `kernels_0` and `kernels_1` lists respectively, so the two kernel functions in the `k_composition_1` function are chosen by defining these two variables.

```
unique_b_k <- combn(1:4, 2)# all combinations of the base kernels, first stage
same_b_k <- matrix(c(1:4),c(1:4), nrow = 2, ncol = 4)# identical base kernel pairs
k_comb <- cbind(unique_b_k, same_b_k)
k_comb <- as.data.frame(t(k_comb))
k_comb[4,] <- c(k_comb[4,2], k_comb[4,1])
names(k_comb) <- c("k_0", "k_1")
```

The transpose of the table with all the kernel combinations we search through at stage one is:

```
> t(k_comb)
    [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
k_0    1    1    1    3    2    3    1    2    3     4
k_1    2    3    4    2    4    4    1    2    3     4
```

In the kernel search we will use grid search to perform hyperparameter tuning in order to find the optimal values in the kernel expression. At eq.(4.1) - eq.(4.4), we see the k_{SE} and k_{LIN} kernels have one hyperparameter, while the k_{PER} and k_{RQ} kernel have two. When we combine these kernels at the first stage, the kernel expression can have two, three or four hyperparameters. Before the tuning of the hyperparameters, we need to find the number of hyperparameters in the kernel expression at each iteration in the for-loop, running through the `k_comb` table. We also need a table with the possible combinations of the hyperparameter values for the kernel expression. This table will have different sizes depending on the number of hyperparameters in the kernel expression.

In order to find the number of hyperparameters in the kernel, we create a list with the names of the base kernels and a set of lists with the names of the different hyperparameters for each base kernel.

```
kernel_names <- c("k_se", "k_per", "k_lin", "k_rq")# names of the base kernels
k_se <- c("ell_se"); k_per <- c("ell_per", "period")# names of the hyperparameters
k_lin <- c("ell_lin"); k_rq <- c("ell_rq", "alpha")
```

With these variable definitions we can now find the number of hyperparameters in the kernel. As an example, say we have the pair of kernels k_{SE} and k_{PER} in the kernel expression. The index values for these kernel functions in the `kernels_0` and the `kernels_1` lists are found at the first row in the `h_param_grid` table. The following code shows how the number of hyperparameters are found by using the `get()` function. The `get()` function allows us to call an **R** object using a character string.

```
> k_0 <- k_comb[1,1]; k_1 <- k_comb[1,2]# first row in the k_comb table
> h_param_k_0 <- get(kernel_names[k_0])
> h_param_k_0# hyperparameters in the k_0 kernel
[1] "ell_se"
> h_param_k_1 <- get(kernel_names[k_1])
> h_param_k_1# hyperparameters in the k_1 kernel
[1] "ell_per" "period"
> tot_nr_h_p <- length(c(h_param_k_0, h_param_k_1))
> tot_nr_h_p# total numbers of hyperparameters in the kernel expression
[1] 3
```

For the grid search we train a model on each hyperparameter combination possible. We will now create the tables to be used in the grid search depending on the number of hyperparameters in the kernel. These tables are created using the `expand.grid()` function. Using this function we create a data frame with all possible combinations for a set of values.

```
interval_gpr <- seq(0.55, 3.25, 0.45)# defining the possible h.param. values
two_h_param <- expand.grid(interval_gpr, interval_gpr)# table with values
# for a kernel which has two hyperparameters
three_h_param <- expand.grid(interval_gpr, interval_gpr, interval_gpr)
four_h_param <- expand.grid(interval_gpr, interval_gpr, interval_gpr, interval_gpr)
five_h_param <- expand.grid(interval_gpr, interval_gpr,
                            interval_gpr, interval_gpr, interval_gpr)
six_h_param <- expand.grid(interval_gpr, interval_gpr, interval_gpr, interval_gpr,
                           interval_gpr, interval_gpr)# six hyperparameters
```

Next, we create the variable `search_lengths` which has character strings with the names of the different tables.

```
search_lengths <- c("two_h_param", "three_h_param", "four_h_param",
                    "five_h_param", "six_h_param")
```

We can now find the table to be used in the grid search for a given kernel expression. The following code shows how the correct table is found if we have the k_{SE} and the k_{PER} kernel with three hyperparameters in total.

```
> tot_nr_h_p# total number of hyperparameters for the SE and PER kernel
[1] 3
> h_param_grid <- get(search_lengths[tot_nr_h_p-1])
> h_param_grid# iterate through 343 combinations of values for the hyperparameters
    Var1 Var2 Var3
1   0.55 0.55 0.55
2   1.00 0.55 0.55
       ⋮
297 1.45 0.55 3.25
298 1.90 0.55 3.25
       ⋮
332 1.45 2.80 3.25
333 1.90 2.80 3.25
 [ reached 'max' / getOption("max.print") -- omitted 10 rows ]
```

We have now found the correct table for the grid search with k_{SE} and k_{PER} in the kernel expression and we continue by explaining how the grid search is implemented in the kernel search.

The grid search has a for-loop which runs through each row in the table with the variable i. Inside this for-loop we have a new for-loop which runs trough the base kernels in the kernel expression using the variable j. For the base kernel pair k_{SE} and k_{PER}, when $i = 297$ and $j = 1$ the variable `ell_se_0` is assigned the value 1.45 from the first column in the `h_param_grid` table. Then for the next iteration when $i = 297$ and $j = 2$, the variable `ell_per_1` and `period_1` will be assigned to the values 0.55 and 3.25 respectively, from the second and third column in the `h_param_grid` table. The inner for-loop on the j variable only runs for the number of base kernels in the kernel expression. After the $j = 2$ iteration the inner loop is finished and the grid search continues in the i variable for-loop. Here the 298th model is trained and evaluated with $k(x, x') = \sigma^2 \left(\exp\left(-\frac{(x-x')^2}{2(1.45)^2} \right) + \exp\left(-\frac{2\sin^2(\pi(x-x')/3.25)}{2(0.55)^2} \right) \right)$ as the kernel expression when `o_1` $= 1$.

In the for-loop on the j variable, we did not explain how the variables in the kernel expression was assigned to the values from the columns in the `h_param_grid` table. We will now explain how this was done. Before the for-loops of the grid search start,

we have a boolean vector where the elements in the vector are true if the jth base kernel has two hyperparameters and false otherwise.

```
> two_h_p <- c(F, F)# variable to identify 2-h.param. kernel
> two_h_p[1] <- ifelse((k_0 == 2)|(k_0 == 4), T, F)# k_0 = 1, SE kernel
> two_h_p[2] <- ifelse((k_1 == 2)|(k_1 == 4), T, F)# k_1 = 2, PER kernel
> two_h_p
[1] FALSE  TRUE
```

Next, the variable names of the hyperparameters in the kernel expression are stored in the variable `all_h_p` as a character string.

```
> h_param_k_0 <- get(kernel_names[k_0])# k_0 = 1
> h_param_k_1 <- get(kernel_names[k_1])# k_1 = 2
> all_h_p <- c(h_param_k_0, h_param_k_1)
> all_h_p
[1] "ell_se"  "ell_per" "period"
```

Notice that when $j = 2$, both `ell_per_1` and `period_1` was assigned one value each. This is done using the boolean `two_h_p` vector and the `h_p_count` variable. When the for-loop on the j variable starts, `h_p_count` always has the value one. The character vector `all_h_p` has the names of each hyperparameter variable in the kernel expression. In addition, each of these named variables again has the names of the variables with indexes. These variables are used in the kernel functions.

```
> all_h_p
[1] "ell_se"  "ell_per" "period"
> ell_se
[1] "ell_se_0" "ell_se_1" "ell_se_2"
> ell_per
[1] "ell_per_0" "ell_per_1" "ell_per_2"
> period
[1] "period_0" "period_1" "period_2"
```

The variables above was created in the preparation of the kernel search with the following code:

```
ell_se <- c(); ell_per <- c(); period <- c()
ell_lin <- c(); ell_rq <- c(); alpha <- c()
for(i in 1:3){
    ell_se[i] <- paste0("ell_se_", i-1)
    assign(paste0("ell_se_", i-1), 1)}
    ⋮
```

```
for(i in 1:3){
    alpha[i] <- paste0("alpha_", i-1)
    assign(paste0("alpha_", i-1), 1)}
```

When j and `h_p_count` equals one in the first iteration, we find the variable with the correct index and assign it the value at the first column. This is done using the `assign()` function which allows us to assign a value to a name.

```
> h_p_count <- 1
> variable <- get(all_h_p[h_p_count])[j]# "ell_se" is the character string and j=1
> assign(variable, h_param_grid[i, h_p_count])
> variable# ell_se[1]
[1] "ell_se_0"
> ell_se_0# the variable which is used in the k_se_0 kernel function
[1] 1.45
```

The next variable needs to be assigned the value from the next column. The `h_p_count` variable decides which column in the `h_param_grid` table to be used, so the next column is chosen by adding one to the `h_p_count` variable.

```
h_p_count <- h_p_count + 1
> h_p_count
[1] 2
```

This line of code will always run after a value has been assigned to a variable. For the next iteration when $j = 2$, `h_p_count` also equals two and we find the correct variable with a new index.

```
> variable <- get(all_h_p[h_p_count])[j]# "ell_per" is the character string and j=2
> variable# ell_per[2]
[1] "ell_per_1"
assign(variable, h_param_grid[i, h_p_count])# h_p_count = 2
h_p_count <- h_p_count + 1
```

When $j = 2$ we are at the last iteration in the for-loop. However, we still have one value left at the third column in the table for the `period_1` variable. In order to assign this value we use an if statement with the jth element in the boolean vector explained earlier, `two_h_p[j]` . In the previous iteration, the jth element was false and the statement was skipped. However, with $j = 2$ the second element is true and the statement will be executed.

```
> two_h_p
[1] FALSE  TRUE
```

The following code shows how the if statement assigns an extra value in the jth iteration when the jth base kernel is a kernel with two hyperparameters:

```
if(two_h_p[j]){# TRUE when the jth base kernel has two hyperparameters
          variable <- get(all_h_p[h_p_count])[j]# j = 2, h_p_count = 3
          assign(variable, h_param_grid[i, h_p_count])
          h_p_count <- h_p_count + 1
        }
```

The `period_1` variable has now been assigned to the value from the third column. We have now explained how the values from the columns of the `h_param_grid` table are assigned to the variables in the grid search.

In the grid search, the number of rows in the table for the kernel expression with k_{SE} and k_{PER}, was 343. This is because the interval we have chosen has seven values, so the number of combinations for the hyperparameters is $7^3 = 343$. For each of these 343 combinations we train and evaluate a model. Moreover, after each evaluation we get two values and store them in one list each, `cv_tr` and `mse_te`.

```
l_h_param_grid <- dim(h_param_grid)[1]
cv_tr <- c()
mse_te <- c()
for(i in 1:l_h_param_grid){# grid search for-loop
      ⋮
 eval_res <- model_eval()
 cv_tr[i] <- eval_res[1]
 mse_te[i] <- eval_res[2]
}# grid search end
```

The `model_eval` is a function which trains and evaluates the model. Inside the function a gaussian process is trained using the `gausspr` function, with the `k_composition_2` kernel function as an argument. The model is trained using the first 80% of the data as the training data. In the `model_eval` function, the `gausspr` function also performs a 2-fold CV on the training data. The `model_eval` function returns two values as an evaluation of the model's performance. The CV is the first value and the second value is the MSE. The MSE is computed on the remaining 20% of the data as the test data.

```
model_eval <- function(){
  table <- c()
  set.seed(1202)
  model <- gausspr(x[1:115], y[1:115], kernel = k_composition_2,
                   var = var_noise, cross = 2)
  res_1 <- cross(model)
  pred_model <- predict(model, x[116:144])
  MSE <- mean((y[116:144] - pred_model)^2)
  res_2 <- MSE
  table <- c(round(res_1, 3), round(res_2, 3))
  return(table)}
```

The models trained in the grid search are evaluated by taking the mean of the CV value and the MSE value from the `model_eval` function. To get an optimized model, we choose the model which gives the lowest mean out of all the models trained in the grid search.

For the k_{SE} and k_{PER} kernel, after the grid search is performed we choose one model out of the 343 models in total. This optimized model is stored in a table where it's kernel expression and hyperparameter values can be read. The table is named `res_table_I` and will be printed out when the kernel search at the first stage is finished.

```
    ⋮
}# grid search end
nr <- which.min((cv_tr + mse_te)/2)# choosing the best model
res_table_I[1, count] <- cv_tr[nr]# storing the CV from the best model
res_table_I[2, count] <- mse_te[nr]# storing the MSE from the best model
# grid search results inserted into table:
col_index <- 1
variable <- get(all_h_p[col_index])[1]
assign(variable, h_param_grid[nr, col_index])
res_table_I[4, count] <- get(variable)# storing one hyperparameter value
col_index <- col_index + 1
    ⋮
```

When the kernel search at the first stage is finished, we need a criteria for choosing one kernel structure over the other. For this criteria we again choose the model which gives the lowest mean, but this time we choose among the optimized models stored in the `res_table_I` table.

```
}# kernel search first stage end
var_1_I <- as.numeric(res_table_I[1,])
var_2_I <- as.numeric(res_table_I[2,])
# choosing the model with the best kernel structure:
var_3_I <- (var_1_I + var_2_I)/2# the mean of the models in table
```

```
first_I <- match(sort(var_3_I), var_3_I)[1]
```

At the first stage we evaluate twenty kernel expressions, ten base kernel pairs with summation as the operation and ten kernel pairs with products as the operation. The following code is the kernel search at the first stage. The kernel search has two for-loops, not including the for-loops of the grid search. The outer for-loop runs through the index in the `operation_1` list and the inner for-loop runs through the rows in the `k_comb` table.

Listing 4.4 Kernel search at the first stage

```
count <- 1
for(g in 1:2){# iterates through search operations: sums and products
  o_1 <- g
  for(h in 1:dim(k_comb)[1]){# iterates through table with kernel combinations
    k_0 <- k_comb[h, 1]
    k_1 <- k_comb[h, 2]
    two_h_p <- c(F, F)# variable to identify 2-h.param. kernel
    two_h_p[1] <- ifelse((k_0 == 2)|(k_0 == 4), T, F)
    two_h_p[2] <- ifelse((k_1 == 2)|(k_1 == 4), T, F)
    h_param_k_0 <- get(kernel_names[k_0])
    h_param_k_1 <- get(kernel_names[k_1])
    all_h_p <- c(h_param_k_0, h_param_k_1)
    tot_nr_h_p <- length(c(h_param_k_0, h_param_k_1))
    h_param_grid <- get(search_lengths[tot_nr_h_p-1])
    l_h_param_grid <- dim(h_param_grid)[1]
    cv_tr <- c()
    mse_te <- c()
    for(i in 1:l_h_param_grid){# grid search for-loop
      h_p_count <- 1
      for(j in 1:2){# iterates through the nr. of base kernels
        variable <- get(all_h_p[h_p_count])[j]
        assign(variable, h_param_grid[i, h_p_count])
        h_p_count <- h_p_count + 1
        if(two_h_p[j]){# TRUE when the jth base kernel has two hyperparameters
          variable <- get(all_h_p[h_p_count])[j]
          assign(variable, h_param_grid[i, h_p_count])
          h_p_count <- h_p_count + 1
        }}
      eval_res <- model_eval()
      cv_tr[i] <- eval_res[1]
      mse_te[i] <- eval_res[2]
    }# grid search end
    nr <- which.min((cv_tr + mse_te)/2)# choosing the best model
    res_table_I[1, count] <- cv_tr[nr]# storing the CV from the best model
    res_table_I[2, count] <- mse_te[nr]# storing the MSE from the best model
    # grid search results inserted into table:
    col_index <- 1
    variable <- get(all_h_p[col_index])[1]
    assign(variable, h_param_grid[nr, col_index])
    res_table_I[4, count] <- get(variable)# storing one hyperparameter value
```

```
    col_index <- col_index + 1
        ⋮
    count <- count + 1# next column of the table
  }}# kernel search first stage end
```

For the air passenger data, the number of models trained at the first stage is large and computationally intensive. The `interval_gpr` interval has seven values, so the number of models trained in the grid search for kernels that have two, three and four hyperparameters are 7^2, 7^3 and 7^4 respectively. In addition to this, the number of models which are trained becomes twice as much because of the 2-fold CV. In order to find the total number of models which are trained, we look at the number of hyperparameters in each of the ten kernel pairs from the `k_comb` table. Among these ten kernel pairs we have three pairs with two hyperparameters, four pairs with three hyperparameters and three pairs with four hyperparameters. The number of models which are trained for one search operation is $3 \cdot 7^2 \cdot 2 + 4 \cdot 7^3 \cdot 2 + 3 \cdot 7^4 \cdot 2 = 17\,444$. Including both of the search operators, products and multiplication, the total number of models trained at the first stage of the kernel search is $17\,444 \cdot 2 = 34\,888$.

The code for the second stage of the kernel search is similar to the code for the first stage. There are only some minor parts of the code which are different. When the search at the second stage starts the values for the variables `k_1` , `k_2` and `o_1` are given, so at the second stage we only search over the values for the `k_2` and `o_2` variables. However, the hyperparamaters of the optimized model in the first stage are not used in the second stage, so the model tuning includes the hyperparameters for all of the three base kernels. Because we have three base kernels at the second stage, the following parts of the code are different: `two_h_p` now has three elements, `all_h_p` has an extra variable and the j variable for-loop in the grid search has three iterations instead of two. After the search at the second stage, a table with the results is again printed out. This time with eighth different models.

Listing 4.5 Kernel search at the second stage

```
count <- 1
for(g in 1:2){# iterates through search operations: sums and products
  o_1 <- which(res_table_I[,first_I][6] == operation_sign)
  o_2 <- g
  for(h in 1:4){# iterates through base kernels to expand on kernel
    k_0 <- which(res_table_I[,first_I][3] == kernel_names)
    k_1 <- which(res_table_I[,first_I][7] == kernel_names)
    k_2 <- h
    two_h_p <- c(F, F, F)# variable to identify 2-h.param. kernel
    two_h_p[1] <- ifelse((k_0 == 2)|(k_0 == 4), T, F)
```

```
      two_h_p[2] <- ifelse((k_1 == 2)|(k_1 == 4), T, F)
      two_h_p[3] <- ifelse((k_2 == 2)|(k_2 == 4), T, F)
      h_param_k_0 <- get(kernel_names[k_0])
      h_param_k_1 <- get(kernel_names[k_1])
      h_param_k_2 <- get(kernel_names[k_2])
      all_h_p <- c(h_param_k_0, h_param_k_1, h_param_k_2)
      tot_nr_h_p <- length(c(h_param_k_0, h_param_k_1,
                             h_param_k_2))
      h_param_grid <- get(search_lengths[tot_nr_h_p-1])
      l_h_param_grid <- dim(h_param_grid)[1]
      cv_tr <- c()
      mse_te <- c()
      for(i in 1:l_h_param_grid){# grid search for-loop
        h_p_count <- 1
        for(j in 1:3){# iterates through the nr. of base kernels
              ⋮
        }}
        eval_res <- model_eval()
        cv_tr[i] <- eval_res[1]
        mse_te[i] <- eval_res[2]
      }# grid search end
      nr <- which.min((cv_tr + mse_te)/2)# choosing the best model
      res_table_II[1, count] <- cv_tr[nr]# storing the CV from the best model
      res_table_II[2, count] <- mse_te[nr]# storing the MSE from the best model
      # grid search results inserted into table:
          ⋮
      count <- count + 1# next column of the table
    }}# kernel search second stage end
```

We are now done with the explanation of the framework for the structural kernel search on the air passenger data. Next we will look at the results from the search.

In fig 4.4 we see the output of both tables which are printed out from the first and second stage of the search. At the first stage, the kernel expression $k_{\text{SE}} + k_{\text{PER}}$ gives the best model. As shown in figure 4.2, this kernel expresses structure which is periodic with noise. At the second stage, the kernel expression is expanded with the periodic kernel and product as the operation to $(k_{\text{SE}} + k_{\text{PER}}) \times k_{\text{PER}}$ as the new kernel expression. As expected, the expansion of the kernel from stage one improved the model and captures more of the structure in the data.

Tables with results from the first and second stage of the kernel search

	1	2	3	4	5	6	7	8	9	10
CV	617.386	1275.012	1273.352	727.152	615.96	1273.111	1273.066	1360.558	1323.704	1273.29
MSE test	2320.717	4941.413	4904.677	2755.025	2322.641	4929.184	4908.336	41538.051	5448.306	4926.924
k_0	k_se	k_se	k_se	k_lin	k_per	k_lin	k_se	k_per	k_lin	k_rq
h_param_1	3.25	1	1	0.55	0.55	0.55	1	0.55	3.25	1.45
h_param_2	-	-	-	-	1.45	-	-	3.25	-	1.9
o_1	+	+	+	+	+	+	+	+	+	+
k_1	k_per	k_lin	k_rq	k_per	k_rq	k_rq	k_se	k_per	k_lin	k_rq
h_param_3	0.55	0.55	1.45	0.55	3.25	1.45	1	2.8	0.55	1.45
h_param_4	1.45	-	1.45	1.45	1	1	-	2.8	-	1

```
> res_table_I[11:20]
```

	11	12	13	14	15	16	17	18	19	20
CV	584.882	1470.711	1273.143	3004.747	583.737	1454.994	1273.06	546.238	4139.601	1274.191
MSE test	3787.629	6326.552	4921.263	14137.51	4294.33	5322.013	4926.81	16326.858	37764.023	4958.675
k_0	k_se	k_se	k_se	k_lin	k_per	k_lin	k_se	k_per	k_lin	k_rq
h_param_1	3.25	1	1	0.55	1	0.55	1.9	3.25	3.25	1.45
h_param_2	-	-	-	-	1.45	-	-	1.9	-	1.9
o_1	×	×	×	×	×	×	×	×	×	×
k_1	k_per	k_lin	k_rq	k_per	k_rq	k_rq	k_se	k_per	k_lin	k_rq
h_param_3	1	0.55	1.9	1.45	3.25	1.45	1	1	0.55	1.45
h_param_4	1.45	-	3.25	0.55	3.25	1	-	1.45	-	1.9

```
> res_table_II
```

	1	2	3	4	5	6	7	8
CV	619.205	577.635	618.295	618.048	543.805	377.338	985.841	541.349
MSE test	2355.089	2220.389	2338.206	2338.219	2394.74	950.497	2104.452	2668.434
k_0{+,×}k_1	(k_se+k_per)	(k_se+k_per)	(k_se+k_per)	(k_se+k_per)	(k_se+k_per)	(k_se+k_per)	(k_se+k_per)	(k_se+k_per)
h_param_1	3.25	3.25	1.9	3.25	1.9	3.25	1.45	1.9
h_param_2	-	-	-	-	-	-	-	-
h_param_3	0.55	0.55	0.55	0.55	0.55	0.55	0.55	0.55
h_param_4	1.45	1.45	1.45	1.45	1.45	0.55	2.35	1.45
o_2	+	+	+	+	×	×	×	×
k_2	k_se	k_per	k_lin	k_rq	k_se	k_per	k_lin	k_rq
h_param_5	1.9	0.55	2.8	3.25	3.25	1.45	0.55	3.25
h_param_6	-	1.45	-	0.55	-	1	-	3.25

Fig. 4.4 At column 1, $k_{\text{SE}} + k_{\text{PER}}$ gives the best score for the given `interval_1` interval. At column 6, $(k_{\text{SE}} + k_{\text{PER}}) \times k_{\text{PER}}$ gives the best score for the given `interval_1` interval.

4.2.1 Hyperparameter tuning

The hyperparameter values tuned by the grid search for the $(k_{\text{SE}} + k_{\text{PER}}) \times k_{\text{PER}}$ kernel does not guarantee to find the best solution. In order to search for values which are not specified in the interval, we have created a function named `adj_hyperparameter`. The `adj_hyperparameter` function takes a hyperparameter in the kernel and tries to increase or decreases the value to a new value which improves the model. This adjustment is done by a fixed amount and for a fixed number of times. A new model is trained after each adjustment with two outcomes. At the first outcome, if the new model is improved by lowering the mean of the CV and MSE values, the adjustment continues. At the second outcome, if the new model is not an improvement, the adjustments stops and the hyperparameter value is changed back to the previous value.

Listing 4.6 Adjust hyperparameter function

```
search_operator <- c('+', '-')
adj_hyperparameter <- function(s, adj, itr, hyperparameter){
  improved <- 0
  eval_res <- model_eval()
```

```
  CV_0 <- (eval_res[1] + eval_res[2])/2
  print(paste("The initial score is", CV_0))
  for(g in 1:itr){
    assign(hyperparameter,
           search_operator[[s]](get(hyperparameter), adj),
           envir = .GlobalEnv)
    if(get(hyperparameter) <= 0){
      assign(hyperparameter,
             search_operator[[s%%2+1]](get(hyperparameter),
                                       adj), envir = .GlobalEnv)
      print(paste("Error", hyperparameter, "is negative"))
      break
    }
    eval_res <- model_eval()
    CV_1 <- (eval_res[1] + eval_res[2])/2
    if(CV_1 >= CV_0){
      assign(hyperparameter,
             search_operator[[s%%2+1]](get(hyperparameter),
                                       adj), envir = .GlobalEnv)
      print(paste("the score", CV_1, "is not an improvement"))
      break
    }
    print("improved")
    improved <- 1
    CV_0 <- CV_1
  }
  print(paste("Resulting score is", CV_0, "with hyperparameter",
              hyperparameter, "=", get(hyperparameter)))
  table <- list(CV_0, hyperparameter, improved)
  return(table)}
```

The `adj_hyperparameter` function has four arguments. The first argument, `s`, takes the number one or two. When `s` is one, the value of the hyperparameter increases and when `s` is two it will decrease instead. The second argument, `adj`, is the amount we choose to adjust the variable. At each adjustment a new model is trained. The third argument, `itr`, is the number of times we set the variable to be adjusted. If the variable is sucessfully adjusted each time, the `adj_hyperparameter` function will stop after the `itr` number of adjustments. The fourth and last argument, `hyperparameter`, is a character string with the name of the chosen variable in the kernel. The `adj_hyperparameter` function will only adjust one hyperparameter in the kernel, so the function needs to run more than once if we want to adjust all of the hyperparameters in a composite kernel.

In the `adj_hyperparameter` function, the `search_operator` list is used to add or subtract on the hyperparameter.

```
search_operator <- c('+', '-')
```

If the new value does not improve the model, we use the modulo operation to set the value back to the previous value. In this way the value of the hyperparameter only changes when the model is improved. The following code gives an example of the modulo operation usage.

```
> s <- 1
> value <- 5
> assign("value", search_operator[[s]](value,3))
> value
[1] 8
> assign("value", search_operator[[s%%2+1]](value,3))# modulo operation
> value
[1] 5
```

When the function adjusts one hyperparameter, the adjustment may not improve the model at first. However, the successful adjustment of another hyperparameter in the kernel may lead to the case that the adjustment of the first hyperparameter is improving the model at the second attempt. In order to adjust for all hyperparameters in the kernel one after the other, we have written the following while-loop:

```
nr_c_h <- length(current_h_par)
iter <- 1
stop_adj <- rep(F, nr_c_h)
while(iter < 9999){
  add_sub <- 2
  attempt <- 0
  first_attempt <- 0
  for(i in 1:5){
    index <- (iter+nr_c_h-1)%%nr_c_h+1
    continue <- adj_hyperparameter(add_sub, 0.1, 5, current_h_par[index])[3]
    if(continue == 1){
      first_attempt <- 1
      stop_adj[index] <- F
    }
    if(continue == 0 && attempt == 1){
      if(first_attempt == 0){
        stop_adj[index] <- T
      }
      break
    }
    if(continue == 0){
      add_sub <- add_sub%%2 + 1
      attempt <- attempt + 1
    }
  }
  if(all(stop_adj)){
    break
  }
  iter <- iter + 1
```

```
}
```

We will now explain how the code above works. The list `current_h_par` has the names of the hyperparameters in the composite kernel.

```
> current_h_par
[1] "ell_se_0"  "ell_per_1" "period_1"  "ell_per_2" "period_2"  "sigma_var"
```

The while-loop will continue to adjust all of the hyperparameters in this list one after the other until the exit of the while-loop. The while-loop exits when none of the hyperparameters can be adjusted to improve the model. When this happens, all of the elements in the boolean vector `stop_adj` are true as the exit-condition. The `stop_adj` vector has length equal to the length of the `current_h_par` list. All of the elements in the vector are set to false before the while-loop starts.

```
> stop_adj <- rep(F, nr_c_h)
> stop_adj
[1] FALSE FALSE FALSE FALSE FALSE FALSE
```

In the while-loop, a for-loop is used to perform both type of adjustments, increasing or decreasing one hyperparameter in the `current_h_par` list. In this for-loop, the `adj_hyperparameter` function runs and we also have three if-statements. These if-statements are used to distinguish between different events that occur when adjusting the hyperparameter. At one event, the adjustments of the hyperparameter have failed in the first attempt for both type of adjustments. When this is the case, we want to break out of the for-loop and continue in the while-loop to adjust a different hyperparameter. At a different event, the first type of adjustment is not sucessfull, but the second type of adjustment is. We also have the event when the first type of adjustment is sucessfull. In both of these events, the for-loop will continue and the `adj_hyperparameter` function will run again. We will now explain what happens in the first type of event, when the for-loop breaks.

When the `adj_hyperparameter` function runs in the for-loop, the function will return the value zero or one. This value is assigned to the `continue` variable which is used in the if-statements. If the `adj_hyperparameter` function returns the value zero, the adjustment failed to improve the model at the first iteration and stopped. If the `adj_hyperparameter` function returns the value one, the adjustment improved the model at least once and continued.

In the for-loop, the `add_sub` variable decides the adjustment in the `adj_hyperparameter` function, taking the value zero or one. After the `adj_hyperparameter` function has

run, if the `continue` variable is set to zero, an if-statement will change the `add_sub` variable to a different value providing a different adjustment. As a result, when the for-loop continues and the `adj_hyperparameter` function runs for a second time, the function changes the hyperparameter with a new type of adjustment. If this second attempt again fails at the first iteration, so the `continue` is set to zero twice, then an element in the `stop_adj` is set as true by an if-statement. Furthermore, the for-loop breaks and the while-loop continues, choosing a new hyperparameter to run the for-loop on. The while-loop breaks when this event from the for-loop occurs successively for all hyperparameters in the list.

For the air passenger data, we took the $(k_{\text{SE}} + k_{\text{PER}}) \times k_{\text{PER}}$ kernel from the kernel search and ran the while-loop, tuning all of the hyperparameter values from the grid search. In addition to these values, we also tuned the output variance `sigma_var`. In the kernel search, this variable is not tuned and has the default value one. The while-loop was run four times with a smaller adjustment at each time. These four values for `adj` was 1, 0.1, 0.01 and 0.001. The result from running the while-loops was an improvement of the model, lowering the mean of the CV and the MSE as the model evaluation. From the grid search, the model had the mean 663.918 and improved to 323.851 using the while-loop. The improved model can be seen at the right plot in fig. 4.5 and is shown in red.

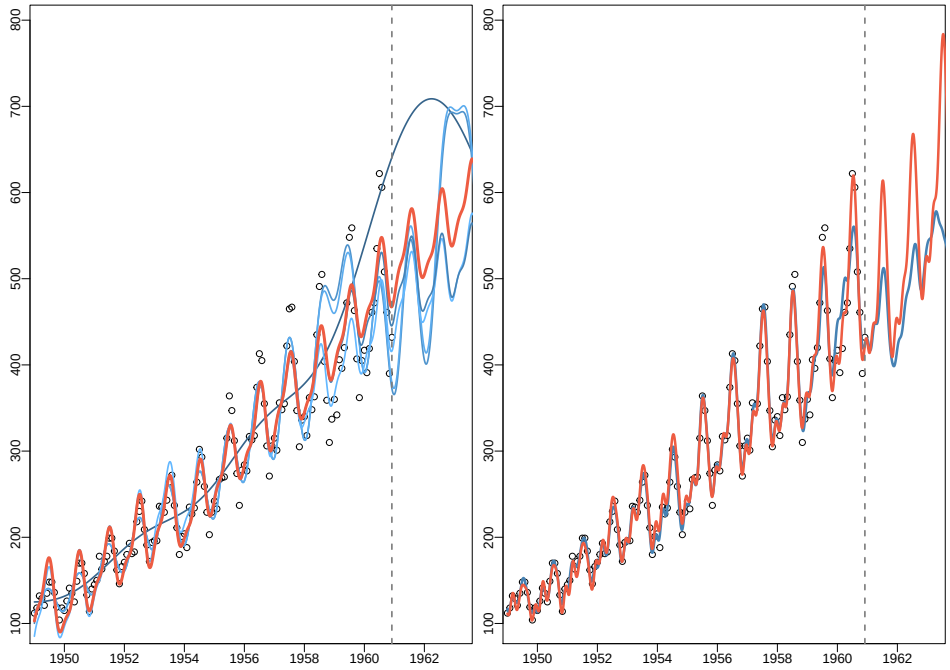

Fig. 4.5 Models from the kernel search on the air passenger data. Left: The six best models from the first stage with two base kernels. The best model is shown in red with the $k_{\text{SE}} + k_{\text{PER}}$ kernel. Right: The best model from the second stage is shown in blue with the $(k_{\text{SE}} + k_{\text{PER}}) \times k_{\text{PER}}$ kernel. The improved model with hyperparameter values found from the while-loop is shown in red.

4.3 SVR on Mauna Loa Atmospheric CO_2 Concentration

We will now look at the CO2 data set [18] and perform a kernel search with support vector regression. The CO2 data has monthly observations of atmospheric CO2 concentrations from 1959 to 1997. For the kernel search with SVR, we have chosen to only use the SE and PER kernels and the LIN and RQ kernels are removed. Compared to the kernel search on the air passenger data, the base kernel grammar is smaller. However, the search will be less computationally intensive. The framework for the code on this search is similar to the code of the previous search with GP. We will now go through some of the main differences in the code.

Since the base kernel grammar is different for the CO2 data, we now have the kernel lists:

```
kernels_0 <- c(‘k_se_0‘, ‘k_per_0‘, ‘k_none‘)
kernels_1 <- c(‘k_se_1‘, ‘k_per_1‘, ‘k_none‘)
kernels_2 <- c(‘k_se_2‘, ‘k_per_2‘, ‘k_none‘)
```

This kernel search will also have two stages and the table with kernel combinations we search through at stage one is:

```
> t(k_comb)
    1 3 4
k_0 1 1 2
k_1 1 2 2
```

In kernlab, the kernel learning algorithm for SVM is `ksvm`. For the CO2 data, the `ksvm` function gives support vector regression by default because `y` is not a factor. The following code shows the `model_eval` function which trains and evaluates the models in the search. The `model_eval` function returns the RMSE of the training set and the RMSE of the test set.

```
model_eval <- function(){
  table <- c()
  model <- ksvm(x[1:374], y[1:374], epsilon = epsilon_value, C = cost,
                kernel = k_composition_2)
  pred_model <- predict(model, x[1:374])
  mse <- mean((pred_model - y[1:374])^2)
  res_1 <- sqrt(mse)
  pred_model <- predict(model, x[375:468])
  res_2 <- mean((pred_model - y[375:468])^2)
  res_2 <- sqrt(res_2)
  table <- c(round(res_1, 3), round(res_2, 3))
  return(table)}
```

The results of the kernel search on the CO2 data is shown in fig. 4.6, with one table from the first stage and one table from the second stage. The hyperparameter values are optimized by the grid search and we have used the same evaluation method as earlier: the hyperparameter values for the kernel expressions in the table, are the values which give the lowest evaluated mean. The mean is computed from the values of the RMSE train and the RMSE test, returned by the `model_eval` function. We also use the same evaluation method when we choose the kernel expression. From the second stage of the search, $(k_{\text{SE}} + k_{\text{PER}}) + k_{\text{SE}}$ is the best kernel but does not improve the model from the first stage very much. The $k_{\text{SE}} + k_{\text{PER}}$ kernel almost gives the same model. We will now choose the $(k_{\text{SE}} + k_{\text{PER}}) + k_{\text{SE}}$ kernel and look more detailed at the models trained in the grid search and our evaluation method.

In the grid search for the $(k_{\text{SE}} + k_{\text{PER}}) + k_{\text{SE}}$ kernel, the interval which is used has five values. With four hyperparameters in the kernel, the number of combinations for the hyperparameters is $5^4 = 625$, so 625 models are trained. In fig. 4.7, twelve of the best models from the grid search are shown. Out of the 625 models, these twelve models have the lowest evaluated mean. In fig. 4.7, we can see that the extrapolation from the models over the dotted line varies greatly. The plot gives us a better idea

	1	2	3	4	5	6
RMSE train	2.084	0.931	0.974	2.078	2.05	2.062
RMSE test	2.387	2.176	6.449	2.525	2.597	6.55
k_0	k_se	k_se	k_per	k_se	k_se	k_per
h_param_1	5.5	3.7	1.9	7.3	3.7	1.9
h_param_2	-	-	7.3	-	-	7.3
o_1	+	+	+	×	×	×
k_1	k_se	k_per	k_per	k_se	k_per	k_per
h_param_3	5.5	0.1	0.1	5.5	7.3	1.9
h_param_4	-	1.9	1.9	-	0.1	7.3

	1	2	3	4
RMSE train	0.93	0.92	0.554	0.898
RMSE test	2.153	2.174	2.851	2.193
k_0{+,×}k_1	(k_se+k_per)	(k_se+k_per)	(k_se+k_per)	(k_se+k_per)
h_param_1	3.7	3.7	3.7	3.7
h_param_2	-	-	-	-
h_param_3	0.1	0.1	0.1	0.1
h_param_4	1.9	1.9	1.9	1.9
o_2	+	+	×	×
k_2	k_se	k_per	k_se	k_per
h_param_5	3.7	0.1	7.3	7.3
h_param_6	-	1.9	-	0.1

Fig. 4.6 Results from the first and second stage of the kernel search. The best kernel expression at each stage is marked with a red rectangle.

of what results we get by the chosen interval for the hyperparameters and our model evaluation approach.

In order to compare our model evaluation with a more traditional approach, we will also look at the models which give the lowest RMSE on the training set. The following code shows how we extract the row numbers of the `h_param_grid` table which have the hyperparameter values for the models with the twelve lowest RMSE train values. The code is run after the kernel search and the `h_param_grid` table is set to have all the hyperparameter combinations for the grid search with the $(k_{\text{SE}} + k_{\text{PER}}) + k_{\text{SE}}$ kernel.

```
ordinal_numbers2 <- c("first2", "second2", "third2", "fourth2", "fifth2",
                      "sixth2", "seventh2", "eighth2", "ninth2", "tenth2",
                      "eleventh2", "twelfth2")
for(i in 1:12){
  sort_rmse_tr <- sort(rmse_tr)
  assign(ordinal_numbers2[i], match(unique(sort_rmse_tr), rmse_tr)[i])}
ord_n_val2 <- t(sapply(ordinal_numbers2, function(x) get(x)))
```

The vector `ord_n_val2` now have the values for the indexes in the `rmse_tr` vector with the lowest RMSE train. The indexes of the `rmse_tr` vector corresponds to the rows of the `h_param_grid` table. A similar code extracts the row numbers from the `h_param_grid` table with the twelve lowest mean of the RMSE train and RMSE test. We will now look at the indexes from both of these vectors.

The rows in the `h_param_grid` table with the twelve lowest RMSE train, ordered from least to greatest are:

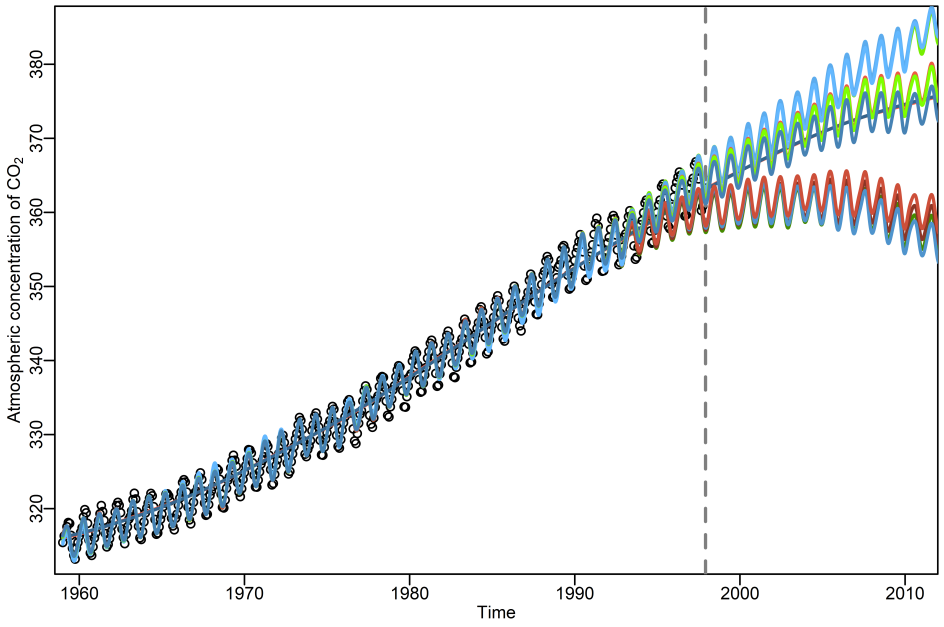

Fig. 4.7 Twelve SVR models with the $(k_{\text{SE}} + k_{\text{PER}}) + k_{\text{SE}}$ kernel. These models had the lowest evaluated mean from the grid search.

```
> ord_n_val2
     first2 second2 third2 fourth2 fifth2 sixth2 seventh2 eighth2
[1,]     26      27    279     278    152    153       154     404
     ninth2 tenth2 eleventh2 twelfth2
[1,]    155    405        530       177
```

When we look at the indexes for the twelve lowest mean values, we see that many of the indexes are the same indexes as in the `ord_n_val2` vector, but in a different order:

```
> ord_n_val
     first second third fourth fifth sixth seventh eighth
[1,]   278    280   279    530   405   404      153    152
     ninth tenth eleventh twelfth
[1,]   154   155       274      264
```

These row numbers of the `h_param_grid` table gives the hyperparameter values for the models shown in fig. 4.7. We did a comparison of both types of model evaluations and in fig. 4.9 we see the models from the row numbers in the `ord_n_val` and `ord_n_val2` vector. The models with the lowest evaluated mean, all have low RMSE test values. The best models with the lowest RMSE train have a high RMSE test and will extrapolate poorly.

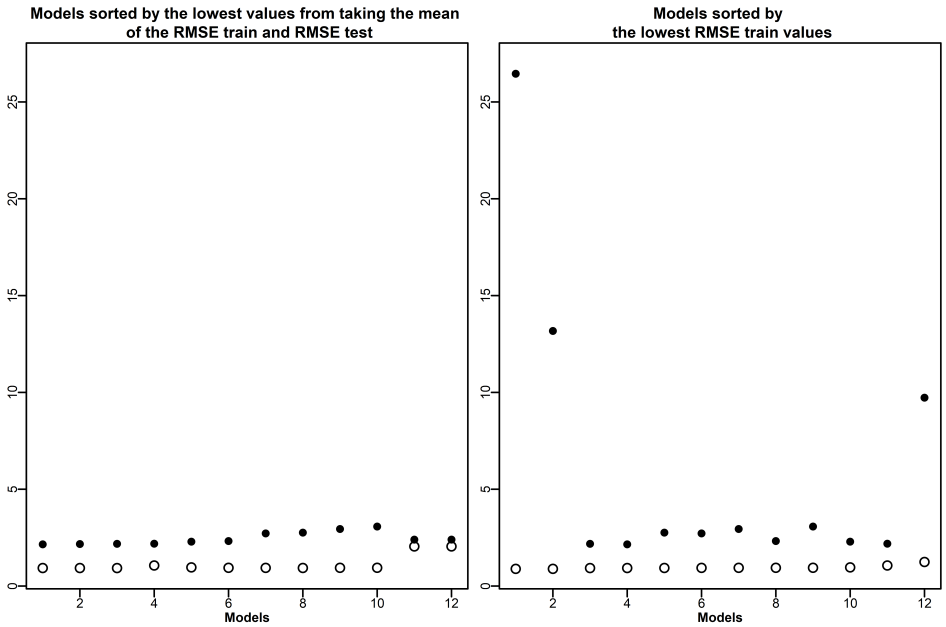

Fig. 4.8 Comparing two types of model evaluation. The white circles is the RMSE on the training set and the black dots is the RMSE for the test set. Left: the twelve models plotted in figure 4.6. Right: two of the best models with the lowest RMSE train, have high RMSE test values.

We will now choose the model with the lowest evaluated mean and again tune the hyperparameters with the while loop and the `adj_hyperparameter` function. The result from running the while loops was an improvement of the evaluated mean. From the grid search, the model had a mean value equal to 1.5415 and improved to 0.5 using the while-loop with `adj` equal to 0.008. The improved model can be seen in fig. 4.9 as the red line.

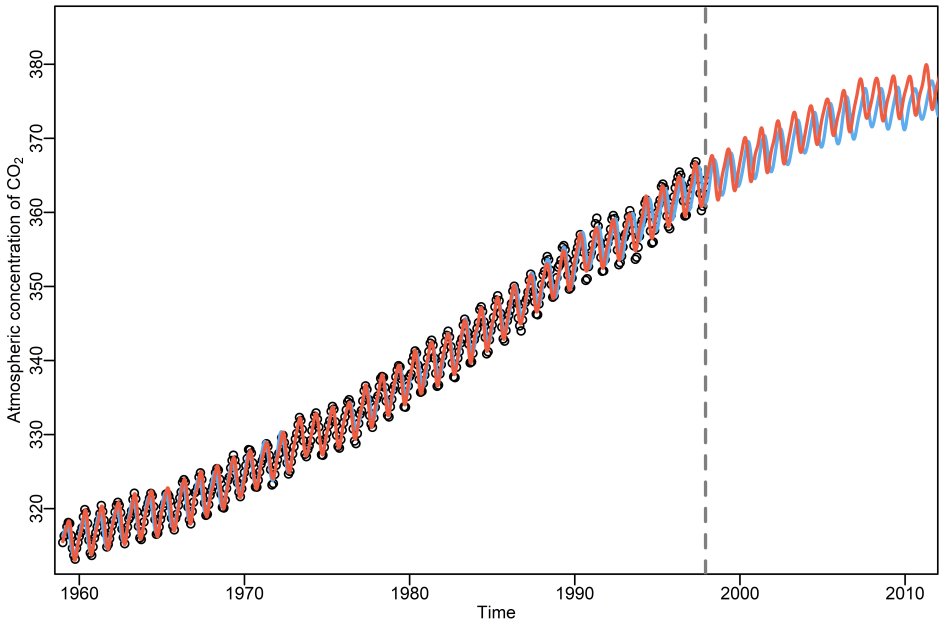

Fig. 4.9 SVR models with the $(k_{\text{SE}} + k_{\text{PER}}) + k_{\text{SE}}$ kernel. The blue line represents the best model from the grid search. The red line is the improvement of the model by the use of the `adjust_hyperparameter` function and the while-loop.

4.4 Gaussian Processes Classification on Pima Indians Diabetes Data

In this section we use the kernel search procedure for classification and the data sets we have used are multidimensional. Compared to the kernel search for regression, the kernel search for binary classification is more difficult. The labels of the data we look at are binary values, so we have less information. Moreover, finding patterns to interpret and discover structure is hard. The results from classification are not as easy to plot as in the case of regression on time series data.

When we define the kernel grammar for classification we do not include the periodic kernel. As discussed in Mrkšić [15], the periodic kernel is suited to one-dimensional time series data and makes less sense in classification. This is because features in classification often are discrete values which does not have a periodic structure. Moreover, if the features are not discrete we only have two class labels, so detecting periodicity is difficult.

For the kernel search with GP classification, we only perform the kernel search with the SE kernel. Moreover, we consider the Pima indians diabetes data set [12] and perform a kernel search with three stages instead of two. The Pima indians diabetes

data had many missing values, so we used the KNN algorithm with the DMwR [24] package to fill in values where the data was missing. For the Pima indians diabetes data, we want to find a model with good performance in identifying the patients who has diabetes. As a consequence, we have not used the classification accuracy as the guiding criteria for the kernel search. The guiding criteria we have used instead, hopefully gives a model with high accuracy but also a model with less false negatives than false positives misclassifications.

The following code shows the `metric_eval` function and is used in the model evaluation for the kernel search:

```
metric_eval <- function(){
  table <- c()
  model <- gausspr(diabetes ~ ., data = train_data, kernel = k_comp_3)
  res_1 <- confusionMatrix(predict(model, train_data[,-9]),
                           train_data$diabetes, positive = "pos")$byClass[1]
  res_2 <- confusionMatrix(predict(model, test_data[,-9]),
                           test_data$diabetes, positive = "pos")$byClass[1]
  res_3 <- confusionMatrix(predict(model, test_data[,-9]),
                           test_data$diabetes, positive = "pos")$overall[1]
  table <- c(round(res_1, 3), round(res_2, 3), round(res_3, 3))
  return(table)}
```

The `metric_eval` function returns three variables: `res_1` , `res_2` and `res_3` . The `res_1` and `res_2` variables, have the values for the sensitivity measure on the training set and the test set respectively. The sensitivity is the true positive rate and measures the patients who actually have diabetes and are correctly identified with the disease. The third variable `res_3` , is the accuracy measure of the model on the test set. In the kernel search for classification, the models are evaluated by adding these three measurements and the model with the maximum sum is the optimal model. In fig. 4.10, we see the results of each stage in the kernel search with one table from each stage. In the first stage of the search, the best kernel $k_{\text{SE}} + k_{\text{SE}}$ provides a model with the maximum sum equal to 2.5. In the third stage of the search, the improved model with the $(k_{\text{SE}} + k_{\text{SE}}) + k_{\text{SE}}$ kernel has the evaluation score equal to 2.615. Comparing these two models, we see that both models have the same sensitivity measure on the test set, but the model from the third stage gives a better train sensitivity and test accuracy measure.

We will now compare the composite $(k_{\text{SE}} + k_{\text{SE}}) + k_{\text{SE}}$ kernel which we have found with the built-in RBF kernel in the kernlab package. The RBF kernel in kernlab [11] is defined as $k_{\text{RBF}}(x, x') = \exp(-\sigma\|x - x'\|^2)$. When the RBF kernel is used, the package automatically tunes the hyperparameter σ. This tuning is done by the `sigest()` function, which estimates the range of values for the σ hyperparameter which would

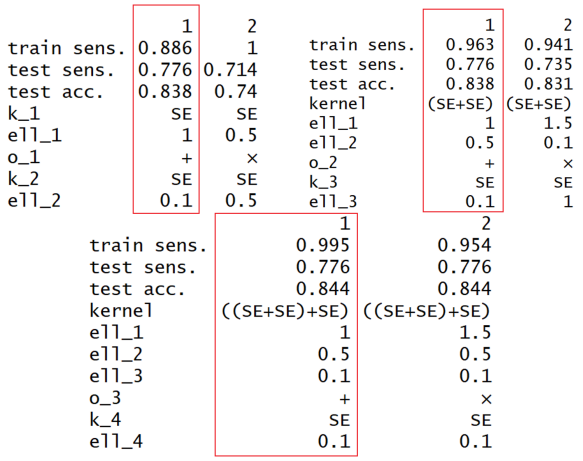

Fig. 4.10 Kernel search result on Pima diabetes data set

provide good results when SVM(`ksvm`) is used with it. Moreover, this estimation is based on the 0.1 and 0.9 quantile of $\|x - x'\|^2$. In fig. 4.11 we see the confusion matrix of the two models on the training set and test set. For the performance on the training set, the model with the $(k_{\text{SE}} + k_{\text{SE}}) + k_{\text{SE}}$ kernel performs better. Out of the 219 patients who actually have diabetes, 218 patients are correctly identified with the disease. The other model fails to identify 83 of the patients with diabetes and out of the 395 patients who does not have the disease, 47 patients are wrongly identified with diabetes. For the performance on the test set, the two models almost have an equal performance, and both models have less false negatives than false positives.

We will now look at another medical data set, and run two kernel searches with different base kernel grammar in each search for comparisons.

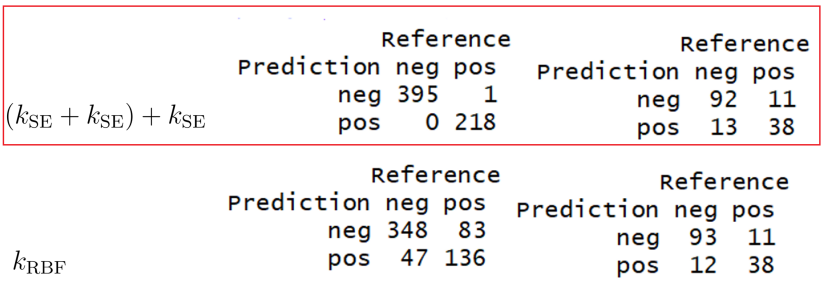

Fig. 4.11 Confusion matrix on the training set and test set. Top: Performance of the model from the kernel search. Bottom: Performance of the model with the RBF kernel from kernlab.

4.5 SVM on Heart Disease Data

For the kernel search with SVM classification, we will use the heart disease data from the UCI machine learning repository [5] and try to predict which patients who has the disease. For this data set, some assumptions of the data was done in order to create a binary classification problem. See [1] or our source code to look at the assumptions and data preparations we used.

The data set has fourteen variables and the goal for this kernel search, was to perform a search with one dimensional base kernels and find a composite kernel across different dimensions. With this in mind a different structure of the search operators was used and we got a different set of kernels. In the previous kernel search, expressions such as $(\mathrm{SE}_1 + \mathrm{SE}_2) \times \mathrm{SE}_3$ was possible. With the new grammar these expressions are no longer possible and we only have expressions which are sums of products such as $\mathrm{SE}_1 \times \mathrm{SE}_3 + \mathrm{SE}_2$ or $\mathrm{SE}_1 + \mathrm{SE}_2 \times \mathrm{SE}_3$. This sums of products grammar allows for more interpetability of the constructed kernels [15] but has less predictive power than the other procedure, so there is a trade-off. We will now explain how this sums of product grammar was implemented.

We first created a set of infix functions for each operation used in the kernel. With infix functions the function name comes in between its arguments and not before which is more common. In **R** the user-defined infix functions starts and ends with `%`.

```
`%op_I%` <- function(a, b){
  operations[[o_1]](a, b)}
`%op_II%` <- function(a, b){
  operations[[o_2]](a, b)}
`%op_III%` <- function(a, b){
  operations[[o_3]](a, b)}
```

In the above code, `operations` is a list which stores the summation and operation function and is used in the same way as we have seen before:

```
operations <- c(`+`, `*`)
```

The following main kernel function which is used in the search, uses the infix functions and the `kern_` lists, where the `kern_` lists have the base kernel functions, same as before. By setting different values for the `o_` and `k_` variables, the main kernel

[1] R document by Brigitte Mueller at RPubs: `https://rpubs.com/mbbrigitte/heartdisease` .

function can generate kernel expressions which have one or up to four base kernels at most.

```
k_comp <- function(x, y){
  res <- kern_1[[k_1]](x, y)%op_I%kern_2[[k_2]](x, y)%op_II%
    kern_3[[k_3]](x, y)%op_III%kern_4[[k_4]](x, y)
  return(res)}
```

We now have the `k_comp` kernel function which provide the sums of product grammar for the kernel search. In order to achieve composite kernels with base kernels across different dimensions, we tried to change the base kernels in the following way but it did not work as planned.

```
d_var <- 2
k_se_1 <- function(x, y){(sigma_var)^2*
exp(-0.5*(1/(ell_se_1)^2)*sum((x[,d_var] - y[,d_var])^2))}
```

The idea was that setting `d_var` , for instance as two, would give a one-dimensional base kernel on the second dimension. For the kernlab package, operating across different dimensions on the base kernels does not seem to be possible. However, we still used the sums of product grammar on the kernel search for the hearth disease data with SVM.

We performed two different kernel search which compared two different base kernel grammars. We first performed a search with the base kernels SE and LIN and then a new search which instead only used the SE kernel. For the LIN kernel, we chose to remove its hyperparameter and the kernel function which we used instead was:

```
k_lin_0 <- function(x, y){(sigma_var)^2*(sum((x)*(y)))}
```

This change made the grid search less computationally intensive and faster to run. In fig. 4.12 we see the results from each of the three stages in the kernel search with the SE and LIN kernels. The evaluation method of the models is the same as in the previous search. We want to find a model with high accuracy and less false negatives than false positives misclassifications. Comparing the best models from the first and third stage, the model from the first stage has the sum of the evaluation measures equal to 2.676. The model from the third stage has a sum equal to 2.68 which is not a big improvement. However, the test sensitivity measure has an improvement which is more significant, going from 0.854 to 0.902.

Looking at the results from the kernel search which only used the SE kernel in fig. 4.13, we see that the search performs better than the SE and LIN search at stage

	1	2	3	4	5	6
train sens.	1	0.979	0.896	0.938	0.927	0.938
test sens.	0.805	0.854	0.854	0.829	0.878	0.756
test acc.	0.831	0.843	0.73	0.843	0.764	0.775
k_1	k_se	k_se	k_lin	k_se	k_se	k_lin
ell_1	1	0.5	-	2	2	-
o_1	+	+	+	×	×	×
k_2	k_se	k_lin	k_lin	k_se	k_lin	k_lin
ell_2	0.1	-	-	2	-	-

	1	2	3	4
train sens.	0.99	0.917	1	0.927
test sens.	0.829	0.951	0.854	0.902
test acc.	0.843	0.697	0.809	0.764
kernel	k_se+k_lin	k_se+k_lin	k_se+k_lin	k_se+k_lin
ell_1	2.5	3	1.5	1
ell_2	-	-	-	-
o_2	+	+	×	×
k_3	k_se	k_lin	k_se	k_lin
ell_3	0.1	-	1.5	-

	1	2	3	4
train sens.	0.958	1	0.948	0.969
test sens.	0.902	0.829	0.927	0.878
test acc.	0.82	0.809	0.775	0.831
kernel	k_se+k_lin×k_se	k_se+k_lin×k_se	k_se+k_lin×k_se	k_se+k_lin×k_se
ell_1	0.1	0.1	0.5	2
ell_2	-	-	-	-
ell_3	2	3	2.5	2
o_3	+	+	×	×
k_4	k_se	k_lin	k_se	k_lin
ell_4	0.5	-	2.5	-

Fig. 4.12 Kernel search result on the Heart Disease data set. The SE and LIN kernels are used in the search.

two and three. Comparing the best models from each search, the model with the $k_{\text{SE}} + k_{\text{SE}} + k_{\text{SE}} \times k_{\text{SE}}$ kernel performs best with an evaluation sum equal to 2.719. This model has a higher sensitivity measure on the training set and a higher accuracy on the test set. However, the model with the $k_{\text{SE}} + k_{\text{LIN}} \times k_{\text{SE}} + k_{\text{SE}}$ kernel has a higher test sensitivity and its evaluation sum of 2.68 is close to 2.719.

	1	2
train sens.	1	0.938
test sens.	0.805	0.829
test acc.	0.831	0.843
k_1	k_se	k_se
ell_1	1	2
o_1	+	×
k_2	k_se	k_se
ell_2	0.1	2

	1	2
train sens.	1	1
test sens.	0.829	0.805
test acc.	0.854	0.854
kernel	k_se+k_se	k_se+k_se
ell_1	1.5	1.5
ell_2	0.1	0.1
o_2	+	×
k_3	k_se	k_se
ell_3	0.1	1.5

	1	2
train sens.	1	1
test sens.	0.854	0.854
test acc.	0.843	0.865
kernel	k_se+k_se+k_se	k_se+k_se+k_se
ell_1	1.5	2.5
ell_2	0.5	1
ell_3	0.5	0.1
o_3	+	×
k_4	k_se	k_se
ell_4	0.1	1.5

Fig. 4.13 Results from the second search on the data set using SE kernels only.

To visualize the performance of the best models from each search, we used a ROC curve. In fig. 4.14 and fig. 4.15, we see each model compared with the built-in RBF kernel in kernlab. On the training set, both of the constructed kernels performs better than the RBF kernel. The AUC for the $k_{\text{SE}} + k_{\text{LIN}} \times k_{\text{SE}} + k_{\text{SE}}$ kernel and the $k_{\text{SE}} + k_{\text{SE}} + k_{\text{SE}} \times k_{\text{SE}}$ kernel, is 0.986 and 1 respectively. The RBF has an lower AUC equal to 0.962. Next, we look at the test set. On the test set, the RBF kernel performs better than both of the other two models. The RBF model has an AUC value which equals 0.956, and the model with the $k_{\text{SE}} + k_{\text{SE}} + k_{\text{SE}} \times k_{\text{SE}}$ kernel has an lower AUC value equal to 0.891. In contrast, the model with the $k_{\text{SE}} + k_{\text{LIN}} \times k_{\text{SE}} + k_{\text{SE}}$ kernel has

an AUC value equal to 0.892 and performs a little better than the $k_{\text{SE}} + k_{\text{SE}} + k_{\text{SE}} \times k_{\text{SE}}$ kernel.

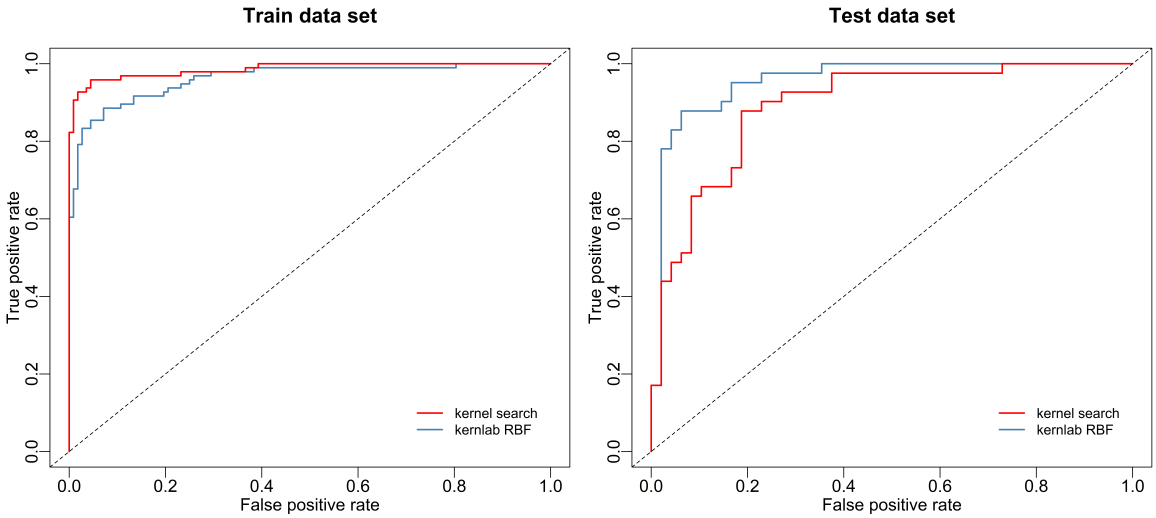

Fig. 4.14 ROC curves comparing the $k_{\text{SE}} + k_{\text{LIN}} \times k_{\text{SE}} + k_{\text{SE}}$ kernel and the RBF kernel on the heart disease data. The $k_{\text{SE}} + k_{\text{LIN}} \times k_{\text{SE}} + k_{\text{SE}}$ kernel performs better than the RBF kernel on the training set but not on the test set.

We will now summarize the four analyses we have done. For the regression data, both the kernel search on the air passenger data and the CO2 data had two stages and a table with the results from both stages, with two and three base kernels was presented.

For the GP kernel search on the air passenger data, the best model with two base kernels seems to give good extrapolation, but does not capture the growing amplitude in the data structure. Consequently, the model is overfitting in the first years of the time period and underfitting in the last years. Next, the best model with three base kernels manage to capture the growing amplitude of the observations over time. Moreover, the model also captures the local periodic structure in the time series. However, the extrapolation of the model shows that it does not capture the linearity of the long term trend. Comparing the two models, the model with two base kernels is the best at capturing this linearity. We also applied a procedure for tuning the hyperparameters in the kernel. Tuning the hyperparameters for the model from the second stage, improved the model and gave better extrapolations.

For the kernel search on the CO2 data, we used a smaller base kernel grammar. Unlike the previous kernel search, the best model from the second stage of the search, only improved the model from the first stage a little. After the search, because our

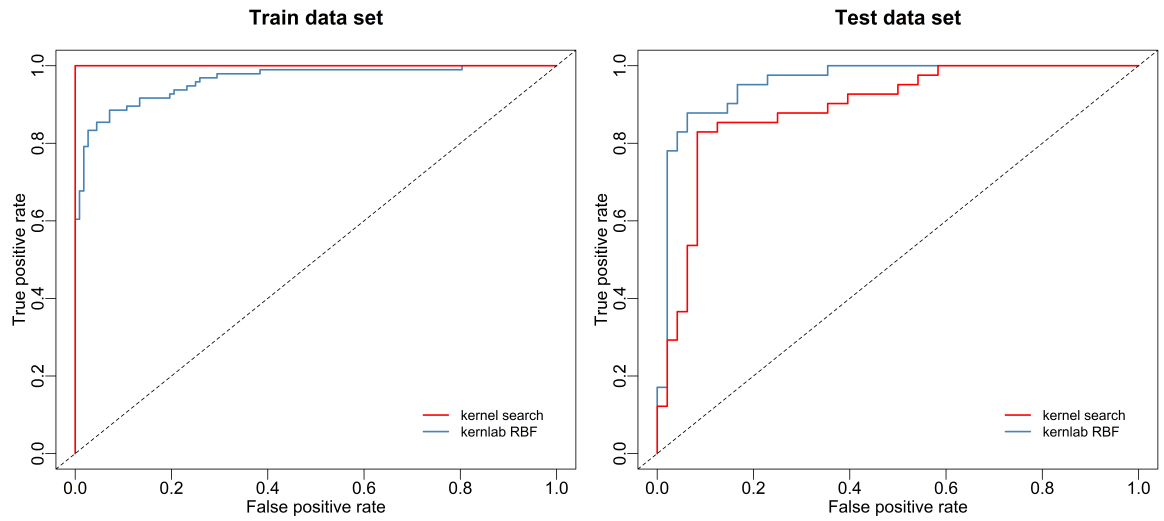

Fig. 4.15 ROC curves comparing the $k_{\text{SE}} + k_{\text{SE}} + k_{\text{SE}} \times k_{\text{SE}}$ kernel and the RBF kernel. ROC curves on the training and test set of the heart disease data.

model evaluation technique is non-traditional, the model evaluation technique was examined by comparing it to models which had the lowest RMSE. Most of the models which had the lowest evaluated mean, corresponded to the models which had the lowest RMSE. Finally, when applying the procedure of hyperparameter tuning, the model again improved and was able to capture more of the upward trend in the data.

Next, we looked at classification problems with two medical data sets and performed kernel searches with three stages. For both data sets, the kernel search was guided by a search criteria to improve the sensitivity measure. In the kernel search on the Pima diabetes data, we only used the SE kernel as the base kernel grammar. The best model from the search, performed better than the other models in both sensitivity and accuracy measure. Hence, the kernel search was successful in the construction of a model adapted to our need, which was identifying ill patients.

For the kernel search on the heart disease data, we did an attempt to perform a kernel search across different dimensions on the base kernels. With that in mind, we implemented the sums of products grammar [15], which provides more interpretation of the kernel expressions which have base kernels across different dimensions. Defining the base kernel functions across different dimensions failed. However, we still used the sums of products grammar on the two kernel searches we applied: one search with the SE and LIN kernels, and one with SE kernel only. We used ROC curves to visualize the performance of the best models from each search. Looking at the ROC curves, we saw that both models from each search gave a performance which was good.

Chapter 5

Topics for further investigation

5.1 Marginal likelihood

The kernel search we have performed in this thesis uses grid search to train the models. For Gaussian process, an alternative would be to train the models by maximizing the marginal likelihood. Since the marginal likelihood may have multiple local optima and does not necessarily provide a better model than grid search, it would be interesting to compare and run two searches in parallel: one search using grid search and the other search using the marginal likelihood.

The marginal likelihood is given by the integral of the likelihood times the prior with a marginalization over the function values $\mathbf{f}$:

$$p(\mathbf{y}|X) = \int p(\mathbf{y}|\mathbf{f}, X)p(\mathbf{f}|X)\, d\mathbf{f}. \tag{5.1}$$

We know that the likelihood is Gaussian distributed, $\mathbf{y}|\mathbf{f} \sim \mathcal{N}(\mathbf{f}, \sigma_n^2 I)$ and that the prior is also Gaussian, $\mathbf{f}|X \sim \mathcal{N}(\mathbf{0}, K)$, so by properties of the multivariate normal distribution the log marginal likelihood can be written

$$\log p(\mathbf{y}|X) = -\frac{1}{2}\mathbf{y}^\top (K_y)^{-1}\mathbf{y} - \frac{1}{2}\log|K_y| - \frac{n}{2}\log 2\pi. \tag{5.2}$$

where $K_y = K_f + \sigma_n^2 I$. The derivatives of the log marginal likelihood have a nice form, which can be seen by using the matrix identities $\frac{\partial}{\partial\theta}K^{-1} = -K^{-1}\frac{\partial K}{\partial\theta}K^{-1}$, and $\frac{\partial}{\partial\theta}\log|K| = \mathrm{tr}\Big(K^{-1}\frac{\partial K}{\partial\theta}\Big)$, where $\frac{\partial K}{\partial\theta}$ is a matrix of elementwise derivatives. The partial derivatives of the marginal likelihood w.r.t. the hyperparameters is

$$\frac{\partial}{\partial\theta_j}\log p(\mathbf{y}|X, \boldsymbol{\theta}) = \frac{1}{2}\mathbf{y}^\top K^{-1}\frac{\partial K}{\partial\theta_j}K^{-1}\mathbf{y} - \frac{1}{2}\mathrm{tr}\Big(K^{-1}\frac{\partial K}{\partial\theta_j}\Big). \tag{5.3}$$

We can now find the values of the hyperparameters which optimizes the marginal likelihood based on the partial derivatives in the above equation. Numerical optimization routines such as conjugate gradient are often used with eq.(5.3) to find good hyperparameters.

Learning the hyperparameter values with the marginal likelihood has an automatic trade off between penalty and data-fit of the GP model. This automatic trade off, which simplifies training a lot, is explained by two of the three terms in the marginal likelihood: the first term $-\frac{1}{2}\mathbf{y}^\top(K_y)^{-1}\mathbf{y}$, which is a data-fit measure and the second term is $-\frac{1}{2}\log|K_y|$, which measures and penalizes the complexity of the model [19].

5.2 Feature selection

In the analyses of the Pima Indians diabetes data and the heart disease data, the data sets was multidimensional and the model we build used all of the different features. However, there is no guarantee that all of the features we used gave a significant contribution to the prediction of the disease. Hence, we will now discuss how feature selection with support vector machine can be done.

In a paper by Li et al. [13], a new approach for a kernel machine with feature scaling technique is presented. This new approach is named Feature Vector Machine (FVM) and reformulates the Lasso regression into a form which is similar to SVM. By introducing kernels, the FVM can perform feature selection with non-linear models and generate sparse solutions in the non-linear feature space. We will now present the Lasso regression model and show how the FVM is derived.

The Lasso regression model is often used for shrinkage and feature selection. The loss function of Lasso regression is

$$L = \sum_i (y_i - \sum_i \beta_p x_{ip})^2 + \lambda \sum_p \|\beta_p\|_1. \tag{5.4}$$

Looking at the last term, $\lambda \sum_p \|\beta_p\|_1$ is a shrinkage penalty term and has the effect of shrinking the estimates of β_p towards zero. When the tuning parameter λ is sufficiently large, Lasso forces some of the coefficient estimates to be exactly equal to zero. As a result, Lasso regression often leads to sparse solution in the feature space with the most irrelevant features removed.

The Lasso regression is limited by its assumption of linearity in the feature space and does not capture non-linear dependencies which we may have between features and output variables. Deriving the FVM from Lasso regression, the goal is to: get

a model which guarantees a sparse solution in the feature space, is able to capture both linear and non-linear relationships between features and the output variables, and which does not involve parameter optimization inside of kernel functions. We will now show how the Lasso regression can be re-formulated and extended into a form which is similar to SVM. We first present some definitions. Let $\mathbf{X} = [x_1, ..., x_N]$ denote a sample matrix, where each column $x_i = (x_1, ..., x_K)^T$ represents a sample vector which have K features. Next, we define the transposed row in the sample matrix as a feature vector $f_q = (x_{1q}, ..., x_{Nq})^T$, where q is the qth row of $\mathbf{X}$. Moreover, we can write $\mathbf{X}^T = [f_1, ..., f_K] = \mathbf{F}$. Lastly, let $y = (y_1, ..., y_n)^T$ denote a response vector where the responses in the vector corresponds to all the samples. We now consider an example space where each basis is represented by an x_i in our sample matrix. Under this example space, both the features f_q and the response vector y can be viewed as a point in the space. For the Lasso regression, the model has an intuitive meaning in this space: the regression coefficients can be regarded as the weights of features vectors. Moreover, all the non-zero weighted feature vectors are on two parallel hyperplanes in the example space. These feature vector, together with the response variables, determine the directions of the two hyperplanes [13]. With the following change of form on the Lasso regression, shown by Perkins et al. [17], we get the geometric point of view explained above:

$$\begin{aligned} & |\sum_i (y_i - \sum_p \beta_p x_{ip}) x_{iq}| \leq \frac{\lambda}{2}, \quad \forall q \\ \Rightarrow \quad & |f_q(y - [f_1, ..., f_K]\beta)| \leq \frac{\lambda}{2}, \quad \forall q. \end{aligned} \tag{5.5}$$

In the above equation, $y - [f_1, ..., f_K]\beta$ defines the orientation of a separating hyperplane and the inequality of the equation only holds for non-zero weighted features. In SVM, the separating hyperplane is defined in the feature space instead of the example space. However, the separating hyperplane in SVM have similar properties to the regression hyperplane we described above. A re-formulation of the Lasso regression with similar character to SVM, is stated as the following optimization problem:

$$\begin{cases} \min\limits_{\beta} & \frac{1}{2} \sum_i (\sum_p \beta_p x_{ip})^2 \\ \text{s.t.} & |\sum_i (y_i - \sum_p \beta_p x_{ip}) x_{iq}| \leq \frac{\lambda}{2}, \quad \forall q \end{cases} \tag{5.6}$$

This equation can be rewritten in the following linear algebra format:

$$\begin{cases} \min\limits_{\beta} & \frac{1}{2}\|[f_1^T, ..., f_K^T]\beta_p\|^2 \\ \text{s.t.} & |f_q(y - [f_1, ..., f_K]\beta)| \leq \frac{\lambda}{2}, \quad \forall q. \end{cases} \tag{5.7}$$

The results we have shown until now, are results from other work which are previous to Li et al. [13] paper. Following this, Li et al. [13] shows that the solution to eq.(5.6) is the exact same of standard Lasso regression. Then, based on the reformulation in eq.(5.7), kernels are introduced to allow feature selection under a non-linear Lasso regression. As a result, the optimization problem defined in eq.(5.7), and its kernelized extensions is refered to as the feature vector machine (FVM).

In Li et al. [13] paper, the following propositions and theorem is stated.
Proposition 1: For a lasso regression problem $\min\limits_{\beta} \sum_i (\sum_p x_{ip}\beta_p - y_i)^2 + \lambda \sum_p |\beta_p|$, if we have β such that: if $\beta_q = 0$, then $|\sum_i (\sum_p \beta_p x_{ip} - y_i)x_{iq}| < \frac{\lambda}{2}$; and if $\beta_q < 0$, then $\sum_i (\sum_p \beta_p x_{ip} - y_i)x_{iq} = \frac{\lambda}{2}$; and if $\beta_q > 0$, then $\sum_i (\sum_p \beta_p x_{ip} - y_i)x_{iq} = -\frac{\lambda}{2}$, then β is the solution of the Lasso regression defined above. For convenience, we refer to the aforementioned three conditions on β as the Lasso sandwich.
Proposition 2: For problem in eq.(5.7), its solution β satisfies the Lasso sandwich.
Theorem 3: Problem in eq.(5.7) $\equiv$ Lasso regression.

For the kernelized extension of eq.(5.7), the $K(f_p, f_q) = \phi(f_p)^T\phi(f_q)$ kernel on the feature vectors are introduced. When f is replaced with $\phi(f)$ in eq.(5.7), we get the following optimization problem for FVM:

$$\begin{cases} \min\limits_{\beta} & \frac{1}{2}\sum_{p,q} \beta_p\beta_q K(f_p, f_p) \\ \text{s.t.} & \forall q, \ |\sum_p \beta_p K(f_q, f_p) - K(f_q, y)| \leq \frac{\lambda}{2}. \end{cases} \tag{5.8}$$

We have now presented the FVM, using this method we have a feature selection algorithm for nonlinear features. As in SVM, slack variables can also be introduced into FVM to get a more robust model. Applying the FVM to the Pima Indians dibates data and the hearth disease data, feature selection can be done by solving a standard SVM problem in feature space and we will get an optimal vector β where some of its elements are zero.

5.3 Conclusion

In this thesis, we have shown how a kernel search in **R** for SVMs and GPs can be implemented. The procedure for the kernel search was inspired by previous work on structural kernel search for regression [6]. We have looked at four real-world data sets and successfully performed a kernel search on both regression and classification tasks. The code we have written to implement the search, have been explained in detail. In our implementation, the search identifies the hyperparameters in the current kernel expression at each iteration of the search. It also identifies the number of hyperparameters in the kernel and performs a grid search accordingly. If we include more stages in the kernel search, the number of possible combinations for the hyperparameters in the kernel increases at each stage. The possible number of hyperparameters also increases, so the implementation of a search with many stages poses a challenge in programming. In our implementation, we can easily expand the kernel search to have many stages and consider composite kernels which have many base kernels. For the real-world data set analyses, we have only used three stages at most and a search with more depth in exploring base kernel combinations can be done. With grid search as the optimization method for the hyperparameters, a kernel stage with many stages will be more computationally intensive. However, it will then be possible to redefine the number of hyperparameters in some of the base kernels and choose an interval with less amount of values. Moreover, with our while-loop and the `adj_hyperparameter` function, a smaller interval for the grid search does not necessarily become a problem: when running the while-loop we can find better hyperparameter values which lie between the values specified in the interval. For the `adj_hyperparameter` function and while-loop procedure, further experimentation can be done by randomly sampling the order of which hyperparameters we adjust, one by one. Further investigation which can be done, is to use different values for the regularization parameter, C, in the kernel search with SVM and SVR, and see how they perform differently.

The strengths of the grid search as the optimization method, is that it easily adapts to both SVMs and GPs and the kernel is chosen by data structure. This chosen process can also be combined with the expertise knowledge in practice to achieve the best result.

References

[1] Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 144–152, New York, NY, USA. ACM.

[2] Camps-Valls, G., Verrelst, J., Munoz-Mari, J., Laparra, V., Mateo-Jimenez, F., and Gomez-Dans, J. (2016). A survey on gaussian processes for earth-observation data analysis: A comprehensive investigation. *IEEE Geoscience and Remote Sensing Magazine*, 4(2):58–78.

[3] David Duvenaud (2019). The kernel cookbook: Advice on covariance functions. website:. https://www.cs.toronto.edu/~duvenaud/cookbook/. Online; accessed 20-May-2019.

[4] Dr. Saed Sayad (2010-2019). Support vector machine - regression (svr). http://www.saedsayad.com/support_vector_machine_reg.htm. Online; accessed 20-May-2019.

[5] Dua, D. and Graff, C. (2017). UCI machine learning repository.

[6] Duvenaud, D., Lloyd, J. R., Grosse, R., Tenenbaum, J. B., and Ghahramani, Z. (2013). Structure discovery in nonparametric regression through compositional kernel search. In *Proceedings of the 30th International Conference on Machine Learning*, pages 1166–1174.

[7] Efron, B. and Hastie, T. (2016). *Computer Age Statistical Inference: Algorithms, Evidence, and Data Science*. Cambridge University Press, New York, NY, USA, 1st edition.

[8] Hastie, T., Tibshirani, R., and Friedman, J. (2001). *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA.

[9] James, G., Witten, D., Hastie, T., and Tibshirani, R. (2014). *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated.

[10] Jeremy Jordan (2018). Support vector machines. https://www.jeremyjordan.me/support-vector-machines/. Online; accessed June-2019.

[11] Karatzoglou, A., Smola, A., Hornik, K., and Zeileis, A. (2004). kernlab – an S4 package for kernel methods in R. *Journal of Statistical Software*, 11(9):1–20.

[12] Leisch, F. and Dimitriadou, E. (2010). *mlbench: Machine Learning Benchmark Problems*. R package version 2.1-1.

[13] Li, F., Yang, Y., and Xing, E. P. (2006). From lasso regression to feature vector machine. In Weiss, Y., Schölkopf, B., and Platt, J. C., editors, *Advances in Neural Information Processing Systems 18*, pages 779–786. MIT Press.

[14] Lloyd, J. R., Duvenaud, D., Grosse, R., Tenenbaum, J. B., and Ghahramani, Z. (2014). Automatic construction and natural-language description of nonparametric regression models. *CoRR*, abs/1402.4304.

[15] Mrkšić, N. (2014). Kernel structure discovery for Gaussian process classification. Master's thesis, Computer Laboratory, University of Cambridge.

[16] Ng, A. (2000). Cs229 lecture notes, part v support vector machines.

[17] Perkins, S., Lacker, K., and Theiler, J. (2003). Grafting: Fast, incremental feature selection by gradient descent in function space. *Journal of Machine Learning Research*, 3:1333–1356.

[18] R Core Team (2018). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

[19] Rasmussen, C. E. (2006). Gaussian processes for machine learning. MIT Press.

[20] Schölkopf, B., Luo, Z., and Vovk, V. (2013). *Empirical Inference - Festschrift in Honor of Vladimir N. Vapnik*. Springer.

[21] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. (2017). Mastering the game of go without human knowledge. *Nature*, 550:354–.

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

[23] The Automatic Statistician (2019). Website:. https://automaticstatistician.com.

[24] Torgo, L. (2016). *Data Mining with R, learning with case studies, 2nd edition*. Chapman and Hall/CRC.

[25] Vapnik, V. and Lerner, A. (1963). Pattern recognition using generalized portrait method. *Automation and Remote Control*, 24:774–780.

[26] Vapnik, V. N. (1995). *The Nature of Statistical Learning Theory*. Springer-Verlag, Berlin, Heidelberg.