Chapter 8
Reproducing Kernel Hilbert Spaces Regression and Classification Methods

8.1 The Reproducing Kernel Hilbert Spaces (RKHS)

One of the main goals of genetic research is accurate phenotype prediction. This goal has largely been achieved for Mendelian diseases with a small number of risk variants (Schrodi et al. 2014). However, many traits (like grain yield) have a complex genetic architecture that is not well understood (Golan and Rosset 2014). Phenotype prediction for such traits remains a major challenge. A key challenge in complex phenotype prediction is accurate modeling of genetic interactions, commonly known as epistatic effects (Cordell 2002). In recent years, there has been mounting evidence that epistatic interactions are widespread throughout biology (Moore and Williams 2009; Lehner 2011; Hemani et al. 2014; Buil et al. 2015). It is well accepted that epistatic interactions are biologically plausible, on the one hand (Zuk et al. 2012), and are difficult to detect, on the other hand (Cordell 2009), suggesting that they may be highly influential in our limited success in modeling complex heritable traits.

Reproducing Kernel Hilbert Spaces (RKHS) regression was one of the earliest statistical machine learning methods suggested for use in plant and animal breeding (Gianola et al. 2006; Gianola and van Kaam 2008) for the prediction of complex traits. An RKHS is a Hilbert space of functions in which all the evaluation functionals are bounded linear functionals. The fundamental idea of RKHS methods is to project the given original input data contained in a finite-dimensional vector space onto an infinite-dimensional Hilbert space. The kernel method consists of transforming the data using a kernel function and then applying conventional statistical machine learning techniques to the transformed data, hoping for better results. Methods based on implicit transformations (RKHS methods) have become very popular for analyzing nonlinear patterns in data sets from various fields of study. Furthermore, the introduction of kernel functions has become an efficient alternative to obtain measures of similarity between objects that do not have a natural vector representation. Although the best known application of kernel methods is

Support Vector Machines (SVM), which is studied in the next chapter, lately it has been shown that any learning algorithm based on distances between objects can be formulated in terms of kernel functions, applying the so-called "kernel trick." However, RKHS methods are not limited to regression; they are also really powerful for classification and data compression problems and theoretically sound for dealing with nonlinear phenomena in general. For these reasons, they have found a wide range of practical applications ranging from bioinformatics to text categorization, from image analysis to web retrieval, from 3D reconstruction to handwriting recognition, and from geostatistics to chemoinformatics. The increase in popularity of kernel-based methods is also due in part to the fact that they provide a rich way to capture nonlinear patterns in data that cannot be captured with conventional linear statistical learning methods. In genomic selection, the application of RKHS methods continues to increase, for example, Long et al. (2010) found better performance of RKHS methods over linear models in body weight of broiler chickens. Crossa et al. (2010) compared RKHS versus Bayesian Lasso and found that RKHS was better than Bayesian Lasso in the wheat data set, but a similar performance of both methods was observed in the maize data set. Cuevas et al. (2016, 2017, 2018) found superior performance of RKHS methods over linear models using Gaussian kernels on data of maize and wheat. Cuevas et al. (2019) also found that when using pedigree, markers, and near-infrared spectroscopy (NIR) data (which is an inexpensive and nondestructive high-throughput phenotyping technology for predicting unobserved line performance in plant breeding trials), kernel methods (Gaussian kernel and arc-cosine kernel) outperformed linear models in terms of prediction performance. However, other authors found minimal differences between RKHS methods and linear models, for example, Tusell et al. (2013) in litter size in swine, Long et al. (2010) and Morota et al. (2013) in progeny tests of dairy sires, and Morota et al. (2014) in phenotypes of dairy cows. These publications have empirically shown equal or better prediction ability of RKHS methods over linear models. For this reason, the applications of kernel methods in GS are expected to continue increasing since they can be implemented in current software of genomic prediction and because they are (a) very flexible, (b) easy to interpret, (c) theoretically appealing for accommodating cryptic forms of gene action (Gianola et al. 2006; Gianola and van Kaam 2008), (d) these methods can be used with almost any type of information (e.g., covariates, strings, images, and graphs) (de los Campos et al. 2010), (e) computation is performed in an n-dimensional space even when the original input information has more columns (p) than observations (n) thus avoiding the $p \gg n$ problem (de los Campos et al. 2010), (f) they provide a new viewpoint whose full potential is still far from our understanding, and (g) they are very attractive due to their computational efficiency, robustness, and stability.

The goal of this chapter is to give the user (student or scientist) a friendly introduction to regression and classification methods based on kernels. We also cover the essentials of kernels methods, and with examples, we show the user how to handcraft an algorithm of a kernel for applications in the context of genomic selection.

8.2 Generalized Kernel Model

Like any regression problem, a generalized kernel model assumes that we have pairs $(y_i, \boldsymbol{x}_i)$ for $i = 1, \ldots, n$, where y_i and $\boldsymbol{x}_i$ are the response variable and the vector of independent variables (pedigree of marker data) measured in individual i, and the relationship between y_i and $\boldsymbol{x}_i$ is given by

$$\text{Distribution} : y_i \sim p(y_i|\mu_i)$$

$$\text{Linear predictor} : \eta_i = f(\boldsymbol{x}_i) = \eta_0 + \boldsymbol{k}_i^{\mathrm{T}}\boldsymbol{\beta}$$

$$\text{Link function} : \eta_i = g(\mu_i)$$

where $g(.)$ is a known link function, $\mu_i = h(\eta_i)$, $h(.)$ denotes the inverse link function, $f(\boldsymbol{x}_i) = \eta_0 + \boldsymbol{k}_i^{\mathrm{T}}\boldsymbol{\beta}, \eta_0$ is an intercept term, $\boldsymbol{k}_i = [K(\boldsymbol{x}_i, \boldsymbol{x}_1), \ldots, K(\boldsymbol{x}_i, \boldsymbol{x}_n)]^{\mathrm{T}}$, $K(.,.)$ is the kernel function, and $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_n)^{\mathrm{T}}$ is an $n \times 1$ vector of coefficients. This generalized kernel model provides a unifying framework for kernel-based analyses for dealing with continuous, binary, categorical, and count data, since with different $p(y_i|\mu_i)$ and $g(.)$, we have different models. It is very interesting to point out that under the kernel framework, the problem is reduced to finding n regression coefficients instead of p, as in conventional regression models, thus avoiding the problem of having to solve a regression problem with $p \gg n$. Also, kernel methods are very useful when genotypes and phenotypes are connected in ways that are not well addressed by the linear additive models that are standard in quantitative genetics.

8.2.1 Parameter Estimation Under the Frequentist Paradigm

Inferring f requires defining a collection (or space) of functions from which an element, $\widehat{f}$, will be chosen via a criterion. Specifically, in RKHS, estimates are obtained by solving the following optimization problem:

$$\underbrace{\min}_{f \in H} \left\{ \frac{1}{n} \sum_{i=1}^{n} L(y_i, f(\boldsymbol{x}_i)) + \lambda \| f \|_H^2 \right\}, \tag{8.1}$$

which mean that the optimization problem is performed within the space of functions H, a RKHS, $f \in H$ and $\|f\|_H$ denotes the norm of f in Hilbert space H; $L(y_i, f(\boldsymbol{x}_i))$ is some measure of goodness of fit, that is, a loss function viewed as the negative conditional log-likelihood, which should be chosen in agreement with the type of response variable. For example, for continuous outcomes, this should be constructed in terms of Gaussian distributions, when the response variable is binary in terms of Bernoulli distributions, when the response is count in terms of Poisson or negative binomial distribution, and when it is categorical in terms of multinomial distributions; λ is a smoothing or regularization parameter that should be positive and should control the trade-off between model goodness of fit and complexity; and $\| f\|_H^2$ is the square of the norm of $f(\boldsymbol{x}_i)$ on H, a measure of model complexity (de los Campos et al. 2010). Hilbert spaces are complete linear spaces endowed with a norm that is the square root of the inner product in the space. The Hilbert spaces that are relevant for our discussion are RKHS of real-valued functions, here denoted as H. Those interested in more technical details of RKHS of real functions should read Wahba (1990). By the representer theorem (Wahba 1990), which tells us that the solutions to some regularization functionals in high or infinite-dimensional spaces fall in a finite-dimensional space, the solution for (8.1) admits a linear representation

$$f(\boldsymbol{x}_i) = \eta_0 + \sum_{j=1}^{n} \beta_j K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = \eta_0 + \boldsymbol{k}_i^{\mathrm{T}} \boldsymbol{\beta}, \tag{8.2}$$

where η_0 is an intercept term, $K(\cdot, \cdot)$ is the kernel function, $k_{\mathrm{i}} = [K(x_i,x_1), \ldots, K(x_i,x_{\mathrm{n}})]^{\mathrm{T}}$ as defined before, and β_j are beta coefficients. Notice that $\| f\|_H^2 = \sum_{l,j=1}^{n} \beta_l \beta_j K\left(\boldsymbol{x}_l, \boldsymbol{x}_j\right)$, and by substituting (8.2) into (8.1), we obtain the minimization problem under a frequentist framework with respect to η_0 and $\boldsymbol{\beta}$, as did Gianola et al. (2006) and Zhang et al. (2011):

$$\underbrace{\min}_{\eta_0, \boldsymbol{\beta}} \left\{ \frac{1}{n} \sum_{i=1}^{n} L\left(y_i, \eta_0 + \boldsymbol{k}_i^{\mathrm{T}} \boldsymbol{\beta}\right) + \frac{\lambda}{2} \boldsymbol{\beta}^{\mathrm{T}} \boldsymbol{K} \boldsymbol{\beta} \right\}, \tag{8.3}$$

where $\boldsymbol{K} = [\boldsymbol{k}_1, \ldots, \boldsymbol{k}_n]$ is the $n \times n$ kernel matrix with $\boldsymbol{k}_i$ as defined above. Since $\boldsymbol{K}$ needs to be symmetric and positive semi-definite, the term $\boldsymbol{\beta}^{\mathrm{T}} \boldsymbol{K} \boldsymbol{\beta}$ is an empirical RKHS norm with regard to the training data, λ is a smoothing or regularization parameter that should be positive and should control the trade-off between model goodness of fit and complexity, and the factor $\frac{1}{2}$ is introduced for convenience. The second term of (8.3) acts as a penalization term that is added to the minus log-likelihood. The goal is to find η_0 and $\boldsymbol{\beta}$, which is equivalent to finding $f(\boldsymbol{x}_i) = \eta_0 + \boldsymbol{k}_i^{\mathrm{T}} \boldsymbol{\beta}$ that minimizes (8.3). $f(\boldsymbol{x}_i)$ is based on a basis expansion of kernel functions and this relationship $f(\boldsymbol{x}) = \eta_0 + \boldsymbol{k}_i^{\mathrm{T}} \boldsymbol{\beta}$ is due to the representer theorem (Wahba 1990). Therefore, model specification under the generalized RKHS methods depends on the choice of loss function L(..), the Hilbert space H to build $\boldsymbol{K}$, and the smoothing parameter λ. The smoothing parameter λ can be chosen by cross-validation or generalized cross-validation under the frequentist framework or by specifying a prior distribution for the $\boldsymbol{\beta}$ coefficients under the Bayesian framework (Gianola and van Kaam 2008). It is important to point out that when the response variable is coded as $y_i \in \{-1, 1\}$ and the hinge function is used as the loss function, the problem to solve is the standard support vector machine (Vapnik 1998), which is studied in the next chapter.

8.2.2 Kernels

A kernel function converts information on a pair of subjects into a quantitative measure representing their similarity with the requirement that the function must create a symmetric positive semi-definite (psd) matrix when applied to any subset of subjects. The psd requirement ensures a statistical foundation for using the kernel in penalized regression models. From a statistical perspective, the kernel matrix can be viewed as a covariance matrix, and we later show how this aids in the construction of kernels. Kernels are used to nonlinearly transform the input data $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \boldsymbol{X}$ into a high-dimensional feature space. Next, we provide a definition of kernel function.

Kernel function. Kernel function K is a "similarity" function that corresponds to an inner product in some expanded feature space that for all $\boldsymbol{x}_i$, $\boldsymbol{x}_j \in X$ satisfies

$$K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = \varphi(\boldsymbol{x}_i)^{\mathrm{T}}\varphi\left(\boldsymbol{x}_j\right),$$

where φ is a mapping (transformation) from $\boldsymbol{X}$ to an (inner product) feature space F, $\varphi : \mathbf{x} \to \varphi(\mathbf{x})$. From this definition, we can see that the kernel has the following properties:

1. It is a symmetric function of its argument so that $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = K(\boldsymbol{x}_j, \boldsymbol{x}_i)$.
2. A necessary and sufficient condition for a function $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ to be a valid kernel (Shawe-Taylor and Cristianini 2004) is that the Gram matrix, also called kernel matrix $\mathbf{K}$, whose elements are given by $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$, should be positive semi-definite for all possible choices of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \boldsymbol{X}$.
3. Kernels are all those functions $K(\boldsymbol{u}, \boldsymbol{v})$ that verify Mercer's theorem, that is, for which

$$\int_{\boldsymbol{u},\boldsymbol{v}} K(\boldsymbol{u}, \boldsymbol{v}) g(\boldsymbol{u}) g(\boldsymbol{v}) d\boldsymbol{u} d\boldsymbol{v} > 0$$

for all $g()$ square-integrable functions.

Mercer's theorem is an equivalent formulation of the finitely positive semi-definite property for vector spaces. The finitely positive semi-definite property suggests that kernel matrices form the core data structure for kernel methods technology. By manipulating kernel matrices, one can tune the corresponding embedding of the data in the kernel-defined feature space.

Next, we give an example of the utility of a kernel function and how this works. We assumed that we measured a sample of n plants with two independent variables (x_1, x_2) and one binary dependent variable (y), that is, $(x_{11}, x_{21}, y_1), \ldots, (x_{1n}, x_{2n}, y_n)$. Then we plotted the observed data in Fig. 8.1 (left panel), and since the response variable is binary, we used triangles for denoting diseased plants and crosses for non-diseased plants. The goal is to build a classifier for unseen data using the data given in Fig. 8.1 as the training set. It is not possible to create a linear decision

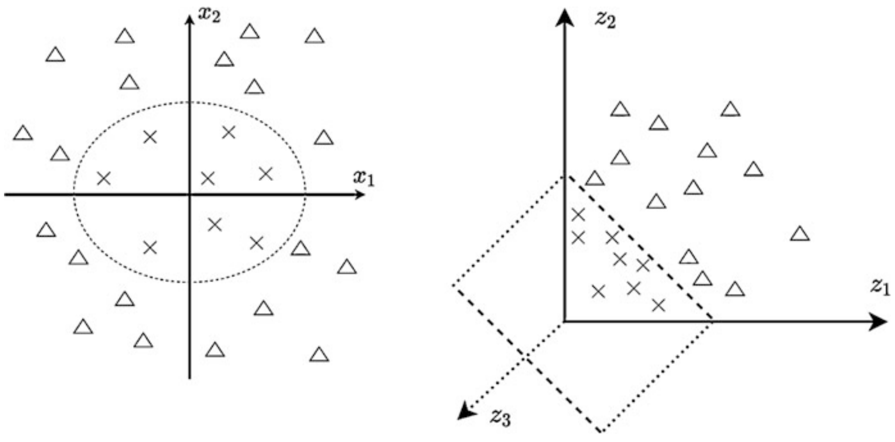

Fig. 8.1 Mapping of the two predictor (x_1, x_2) problems with binary dependent variables (y; crosses = 1 and triangles = 0) where the true decision boundary is an ellipse in predictor space (left panel) to a feature map via nonlinear mapping, $\varphi(\mathbf{x})$. Input space (left panel) and feature space (right panel)

boundary to separate both types of plants (diseased vs. non-diseased) since the true decision boundary is an ellipse in predictor space (Fig. 8.1, left panel). The job of a kernel consists of estimating this boundary by first transforming (mapping) the input information (predictors) via a nonlinear mapping function into a feature map, where the problem can be reduced to estimating a hyperplane (linear boundary) between the diseased and non-diseased plants. We mapped the input information (Fig. 8.1, left panel) to the feature space using the following nonlinear map $\varphi(\mathbf{x}) = \left(z_1 = x_1^2, z_2 = x_2^2, z_3 = \sqrt{2}x_1x_2\right)$ (Fig. 8.1, right panel) and the ellipse became a hyperplane that is parallel to the z_3 axis, which means that all points are plotted on the (z_1, z_2) plane. Therefore, in the feature space, the problem reduces to estimating a hyperplane from the mapped data points.

For this reason, in generalized kernel models, the choice of the kernel function (H) is of paramount importance since it defines the space of functions over which the search for f is performed, and because Hilbert spaces are normed spaces (Akhiezer and Glazman 1963). As mentioned above, by choosing H one automatically defines the reproducing kernel ($\boldsymbol{K}$) which should be at least a psd matrix (de los Campos et al. 2010). There are two main properties that are required for the successful implementation of a kernel function. First, it should capture as precisely as possible the measure of similarity to the particular task and domain, and second, its construction should require significantly less computational resources than would be needed for an explicit evaluation of the corresponding feature mapping, φ.

We will call the original input information ($\boldsymbol{X}$) the input space. Then, with the kernel approach, we define a function for each pair of elements (columns) in this space $\boldsymbol{X}$ that corresponds to a real value. The transformed feature information with the kernel function is called mapped feature space.

8.2.3 Kernel Trick

By kernel trick we mean the use of kernel functions to operate in a high-dimensional space, *implicit* feature space, without ever computing the coordinates of the data in that space, but rather by simply computing the inner products between the images of all pairs of data in the feature space. This operation is often computationally cheaper than the explicit computation of the coordinates. This means that the kernel trick allows you to perform algebraic operations in the transformed data space efficiently and without knowing the transformation φ. For this reason, the kernel trick is a computational trick used to compute inner products in higher dimensional spaces at a low cost. Thus, in principle, any statistical machine learning technique for data in $\boldsymbol{X} \subset \mathbb{R}^n$ that can be formulated in a computational algorithm in terms of dot products can be generalized to the transformed data using the kernel trick. Kernel functions have been introduced for sequence data, graphs, text, images, as well as vectors.

To better understand the kernel trick, we provide an example. Assume that we measure two independent variables (x_1, x_2) in four individuals. In matrix notation, the information of the independent variables (input information) is equal to

$$\boldsymbol{X} = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \\ x_{41} & x_{42} \end{bmatrix}$$

Also, assume we will build a polynomial kernel of degree 2, with

$$\varphi(\mathbf{x}_i)^{\mathrm{T}} = \left(z_1 = x_1^2, z_2 = x_2^2, z_3 = \sqrt{2}x_1x_2\right).$$

Therefore, for building the Gram matrix (kernel matrix), we need to compute

$$\boldsymbol{K} = \begin{bmatrix} \varphi(\mathbf{x}_1)^{\mathrm{T}}\varphi(\mathbf{x}_1) & \varphi(\mathbf{x}_1)^{\mathrm{T}}\varphi(\mathbf{x}_2) & \varphi(\mathbf{x}_1)^{\mathrm{T}}\varphi(\mathbf{x}_3) & \varphi(\mathbf{x}_1)^{\mathrm{T}}\varphi(\mathbf{x}_4) \\ \varphi(\mathbf{x}_2)^{\mathrm{T}}\varphi(\mathbf{x}_1) & \varphi(\mathbf{x}_2)^{\mathrm{T}}\varphi(\mathbf{x}_2) & \varphi(\mathbf{x}_2)^{\mathrm{T}}\varphi(\mathbf{x}_3) & \varphi(\mathbf{x}_2)^{\mathrm{T}}\varphi(\mathbf{x}_4) \\ \varphi(\mathbf{x}_3)^{\mathrm{T}}\varphi(\mathbf{x}_1) & \varphi(\mathbf{x}_3)^{\mathrm{T}}\varphi(\mathbf{x}_2) & \varphi(\mathbf{x}_3)^{\mathrm{T}}\varphi(\mathbf{x}_3) & \varphi(\mathbf{x}_3)^{\mathrm{T}}\varphi(\mathbf{x}_4) \\ \varphi(\mathbf{x}_4)^{\mathrm{T}}\varphi(\mathbf{x}_1) & \varphi(\mathbf{x}_4)^{\mathrm{T}}\varphi(\mathbf{x}_2) & \varphi(\mathbf{x}_4)^{\mathrm{T}}\varphi(\mathbf{x}_3) & \varphi(\mathbf{x}_4)^{\mathrm{T}}\varphi(\mathbf{x}_4) \end{bmatrix}$$

This means that we need to compute each coordinate (cell of $\boldsymbol{K}$) with $\varphi(\mathbf{x}_i)^{\mathrm{T}}\varphi(\mathbf{x}_j)$, with $i, j = 1, 2, 3, 4$. Note that

$$\varphi(\mathbf{x}_i)^{\mathrm{T}}\varphi(\mathbf{x}_j) = \left(x_{i1}^2, x_{i2}^2, \sqrt{2}x_{i1}x_{i2}\right)\begin{bmatrix} x_{j1}^2 \\ x_{j2}^2 \\ \sqrt{2}x_{j1}x_{j2} \end{bmatrix} = x_{i1}^2x_{j1}^2 + 2x_{i1}x_{i2}x_{j1}x_{j2} + x_{i2}^2x_{j2}^2$$

$$= \left(x_{i1}x_{j1} + x_{i2}x_{j2}\right)^2.$$

Therefore,

$$\boldsymbol{K} = \begin{bmatrix} \left(x_{11}^2 + x_{12}^2\right)^2 & \left(x_{11}x_{21} + x_{12}x_{22}\right)^2 & \left(x_{11}x_{31} + x_{12}x_{32}\right)^2 & \left(x_{11}x_{41} + x_{12}x_{42}\right)^2 \\ \left(x_{21}x_{11} + x_{22}x_{12}\right)^2 & \left(x_{21}^2 + x_{22}^2\right)^2 & \left(x_{21}x_{31} + x_{22}x_{32}\right)^2 & \left(x_{21}x_{41} + x_{22}x_{42}\right)^2 \\ \left(x_{31}x_{11} + x_{32}x_{12}\right)^2 & \left(x_{31}x_{21} + x_{32}x_{22}\right)^2 & \left(x_{31}^2 + x_{32}^2\right)^2 & \left(x_{31}x_{41} + x_{32}x_{42}\right)^2 \\ \left(x_{41}x_{11} + x_{42}x_{12}\right)^2 & \left(x_{41}x_{21} + x_{42}x_{22}\right)^2 & \left(x_{41}x_{31} + x_{42}x_{32}\right)^2 & \left(x_{41}^2 + x_{42}^2\right)^2 \end{bmatrix}$$

To compute $\boldsymbol{K}$ we calculated each coordinate using $\varphi(\mathbf{x}_i)^{\mathrm{T}}\varphi(\mathbf{x}_j)$. However, note that

$$\varphi(\mathbf{x}_i)^{\mathrm{T}}\varphi(\mathbf{x}_j) = \left(x_{i1}x_{j1} + x_{i2}x_{j2}\right)^2 = \left(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j + 0\right)^2,$$

where $\boldsymbol{x}_i^{\mathrm{T}} = [x_{i1}, x_{i2}]$ and $\boldsymbol{x}_j = \begin{bmatrix} x_{j1} \\ x_{j2} \end{bmatrix}$.

Hence, the function

$$K(\mathbf{x}_j, \mathbf{x}_j) = \left(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j + 0\right)^2$$

corresponds to a polynomial kernel of degree $d = 2$, and constant $a = 0$, with F, its corresponding feature space. This means that we can compute the inner product between the projections of two points into the feature space without explicitly evaluating the coordinates. In other words, the kernel trick means that we can compute each element (coordinate) of the kernel matrix $\mathbf{K}$, without any knowledge of the true nature of $\varphi(\mathbf{x}_i)$; we only need to know the kernel function $K(\mathbf{x}_j, \mathbf{x}_j)$. This means that the kernel function is a key ingredient for implementing kernel methods in statistical machine learning.

Next, we provide another simple example also using the polynomial kernel of degree 2, with the same two independent variables (x_1, x_2) but with a constant value $a = 1$, that is, $K(\mathbf{x}_j, \mathbf{x}_j) = \left(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j + 1\right)^2$. According to the kernel trick, this means that we do not need knowledge of $\varphi(\mathbf{x}_i)$ to compute all coordinates of the matrix of kernel $\mathbf{K}$, since each coordinate will take values of

$$\begin{aligned}K(\mathbf{x}_j,\mathbf{x}_j) &= \left(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j+1\right)^2 = \left(x_{i1}x_{j1}+x_{i2}x_{j2}+1\right)^2\\ &= \left(x_{i1}x_{j1}+x_{i2}x_{j2}\right)^2+2\left(x_{i1}x_{j1}+x_{i2}x_{j2}\right)+1\\ &= x_{i1}^2x_{j1}^2+2x_{i1}x_{i2}x_{j1}x_{j2}+x_{i2}^2x_{j2}^2+2\left(x_{i1}x_{j1}+x_{i2}x_{j2}\right)+1.\end{aligned}$$

Therefore, the **K** matrix is

$$=\begin{bmatrix}\left(x_{11}^2+x_{12}^2+1\right)^2 & \left(x_{11}x_{21}+x_{12}x_{22}+1\right)^2 & \left(x_{11}x_{31}+x_{12}x_{32}+1\right)^2 & \left(x_{11}x_{41}+x_{12}x_{42}+1\right)^2\\ \left(x_{21}x_{11}+x_{22}x_{12}+1\right)^2 & \left(x_{21}^2+x_{22}^2+1\right)^2 & \left(x_{21}x_{31}+x_{22}x_{32}+1\right)^2 & \left(x_{21}x_{41}+x_{22}x_{42}+1\right)^2\\ \left(x_{31}x_{11}+x_{32}x_{12}+1\right)^2 & \left(x_{31}x_{21}+x_{32}x_{22}+1\right)^2 & \left(x_{31}^2+x_{32}^2+1\right)^2 & \left(x_{31}x_{41}+x_{32}x_{42}+1\right)^2\\ \left(x_{41}x_{11}+x_{42}x_{12}+1\right)^2 & \left(x_{41}x_{21}+x_{42}x_{22}+1\right)^2 & \left(x_{41}x_{31}+x_{42}x_{32}+1\right)^2 & \left(x_{41}^2+x_{42}^2+1\right)^2\end{bmatrix}$$

This implies that we computed each coordinate of **K** without first computing $\varphi(\mathbf{x}_i)$. This trick is really useful since for computing each coordinate of **K** in this example, we only performed dot products with vectors of size two, in the original dimension of the input information, and not dot products of vectors of dimension $\begin{pmatrix}2+2\\2\end{pmatrix}=6$, which is the dimension, in this example, of $\varphi(\mathbf{x}_i)^{\mathrm{T}}=\left[x_{i1}^2,\sqrt{2}x_{i1},\sqrt{2}x_{i1}x_{i2},\sqrt{2}x_{i2},x_{i2}^2,1\right]$. Therefore, this trick facilitates the computation of **K** since it requires less computation resources. The utility of the trick is better appreciated in a large dimensional setting. For example, assume that the input information of each individual ($\mathbf{x}_i$) contains 784 independent variables; this means that to compute matrix **K** we need to compute, for each coordinate, only dot products of vectors of dimension 784 and not of dimension $\begin{pmatrix}784+2\\2\end{pmatrix}=308,505,$ which is the dimension of $\varphi(\mathbf{x}_i)^{\mathrm{T}}$ for the same polynomial kernel with degree 2. For this reason, kernel methods are well suited for handling a massive amount of information, because the computational burden can be proportional to the number of data points rather than to the number of predictor variables (e.g., markers in the context of genomic prediction). This is particularly true if a common weight is assigned to each marker (Morota et al. 2013).

In simple terms, the *kernel trick* makes it possible to perform a transformation from the input data space to a higher dimensional feature space, where the transformed data can be analyzed with conventional linear models and the problem becomes tractable. However, the result highly depends on the considered transformation. If the kernel function is not appropriate for the problem, or the kernel parameters are badly set, the fitted model can be of poor quality. Due to this, special care must be taken when selecting both the kernel function and the kernel parameters to obtain good results.

The kernel trick allows an efficient search in a higher dimensional space, while the related estimation problems are often cast as convex optimization problems that can be solved by many established algorithms and packages. Kernel methods can be applied to all data analysis algorithms whose inputs can be expressed in terms of dot products. If the data in the original space cannot be analyzed satisfactorily with conventional statistical machine learning techniques, the strategy to extend it to nonlinear models using kernel methods is based on the apparently paradoxical idea of transforming the data, by means of a nonlinear function, toward a space with a greater dimension than the space where the data are located and applying any statistical machine learning algorithm to the transformed data.

Therefore, in general terms, the kernelization of an algorithm consists of its reformulation, so that the determination of a pattern or linear regularity in the data can be carried out exclusively from the information collected in the scalar products calculated for all the pairs of elements in the space. Kernel functions are characterized by the property that all finite kernel matrices are positive semi-definite.

8.2.4 Popular Kernel Functions

Next, we provide the most popular kernel methods in statistical machine learning.

Linear Kernel This kernel is defined as $K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j$. For example,

$$K(\boldsymbol{x}, \boldsymbol{z}) = (x_1, x_2)\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = x_1z_1 + x_2z_2 = \varphi(\boldsymbol{x})^{\mathrm{T}}\varphi(\boldsymbol{z}).$$

Next, we provide an *R* function for calculating this kernel that can be used for both single-attribute value vectors and for the whole data set:

```
K.linear=function(x1, x2=x1) {as.matrix(x1)%*%t(as.matrix(x2)) }
```

Next, we simulate a matrix data set:

```
set.seed(3)
X=matrix(round(rnorm(16,2,0.2),2),ncol=8)
X
```

that gives as output:

```
> set.seed(3)
> X=matrix(round(rnorm(16,2,0.2),2),ncol=8)
> X
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,] 1.81 2.05 2.04 2.02 1.76 1.85 1.86 2.03
[2,] 1.94 1.77 2.01 2.22 2.25 1.77 2.05 1.94
```

For individual features in pairs of individuals, this function is used as

```
> K.linear(X[1,1:4],X[2,1:4])
      [,1]   [,2]   [,3]   [,4]
[1,] 3.5114 3.2037 3.6381 4.0182
[2,] 3.9770 3.6285 4.1205 4.5510
[3,] 3.9576 3.6108 4.1004 4.5288
[4,] 3.9188 3.5754 4.0602 4.4844
```

while for the full set of features, it can be used as

```
> K.linear(X)
      [,1]    [,2]
[1,] 29.8212 30.7104
[2,] 30.7104 32.0265
```

This kernel does not overcome the linearity limitation of linear classification and linear regression models in any way since it leaves the original representation unchanged. It is important to point out that linear kernels (such as linear regression, linear support vector machines, and linear support vector regression algorithms) are special cases of more sophisticated kernel-based algorithms.

Polynomial Kernel As mentioned above, this kernel is defined as $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\gamma \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{x}_j + a)^d$, where a is a real scalar and d is a positive integer, and where $\gamma > 0$, $a \geq 0$, and $d > 0$ are parameters. This kernel family makes it possible to easily control the enhanced representation size and degree of nonlinearity by adjusting the d parameter. Positive a can be used to adjust the relative impact of higher order and lower order terms in the resulting polynomial representation. For example, when $\gamma = 1$, $a = 0$, and $d = 2$, we have

$$K(\boldsymbol{x}, \boldsymbol{z}) = \left((x_1, x_2) \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} \right)^2 = (x_1 z_1 + x_2 z_2)^2 =$$

$$x_1^2 z_1^2 + 2 x_1 z_1 x_2 z_2 + x_2^2 z_2^2 = \left[x_1^2, \sqrt{2} x_1 x_2, x_2^2 \right] \begin{bmatrix} z_1^2 \\ \sqrt{2} z_1 z_2 \\ z_2^2 \end{bmatrix} = \varphi(\boldsymbol{x})^{\mathrm{T}} \varphi(\boldsymbol{z}).$$

However, when $\gamma = 1$, $a = 1$, and $d = 2$, we have

$$K(\boldsymbol{x}, \boldsymbol{z}) = \left((x_1, x_2) \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} + 1 \right)^2 = (x_1 z_1 + x_2 z_2 + 1)^2 =$$

$$1 + 2x_1z_1 + 2x_2z_2 + x_1^2z_1^2 + x_2^2z_2^2 + 2x_1z_1x_2z_2$$

$$= \left[1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, \sqrt{2}x_1x_2, x_2^2\right] \begin{bmatrix} 1 \\ \sqrt{2}z_1 \\ \sqrt{2}z_2 \\ z_1^2 \\ \sqrt{2}z_1z_2 \\ z_2^2 \end{bmatrix} = \varphi(\boldsymbol{x})^{\mathrm{T}}\varphi(\boldsymbol{z}).$$

This demonstrates that increasing a increases the coefficients of lower order terms. The dimension of the feature space for the polynomial kernel is equal to $\binom{p+d}{d}$. For example, for an input vector of dimension $p = 10$ and polynomial with degree $d = 3$, the dimension for this polynomial kernel is equal to $\binom{10+3}{3}$ =286, while if $p = 1000$ and $d = 3$, the dimension for this polynomial kernel is equal to $\binom{1000+3}{3}$ =167,668,501. Although convenient to control and easy to understand, the polynomial kernel family may be insufficient to adequately represent more complex relationships.

The R code for calculating this kernel is given next:

```
K.polynomial=function(x1, x2=x1, gamma=1, b=0, d=3){
 (gamma*(as.matrix(x1)%*%t(x2))+b)^d}
```

Now this function can be used as

```
> K.polynomial(X[1,1:4],X[2,1:4])
        [,1]     [,2]     [,3]     [,4]
[1,] 43.29532 32.88180 48.15306 64.87758
[2,] 62.90234 47.77288 69.95999 94.25850
[3,] 61.98630 47.07716 68.94117 92.88582
[4,] 60.18099 45.70607 66.93331 90.18058
```

But for the full set of features, it can be used as

```
> K.polynomial(X)
        [,1]     [,2]
[1,] 26520.11 28963.86
[2,] 28963.86 32849.48
```

Sigmoidal Kernel This kernel is defined as $K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = \tan h\left(\gamma \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{x}_j + b\right)$, where tan$h$ is the hyperbolic tangent defined as $\tan h(z) = \sin h(z)/\cos h(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$. This function is widely used as the activation function for artificial neural networks and deep learning models, and hence has also become popular for kernel methods. If used with properly adjusted parameters, it can represent complex nonlinear relationships. In some parameter settings, it actually becomes similar to the radial kernel (Lin and Lin 2003) described below. However, the sigmoid function may not be positive definite for some parameters, and therefore may not actually represent a valid kernel (Lin and Lin 2003).

Next, we provide an R code for calculating this kernel:

```
K.sigmoid=function(x1,x2=x1, gamma=0.1, b=0)
{ tanh(gamma*(as.matrix(x1)%*%t(x2))+b) }
```

This function is used as

```
> K.sigmoid(X[1,1:4],X[2,1:4])
          [,1]      [,2]      [,3]      [,4]
[1,] 0.3373862 0.3098414 0.3485656 0.3815051
[2,] 0.3779793 0.3477219 0.3902119 0.4260822
[3,] 0.3763152 0.3461650 0.3885066 0.4242635
[4,] 0.3729798 0.3430454 0.3850881 0.4206158
```

For the full set of features, it is used as

```
> K.sigmoid(X)
          [,1]      [,2]
[1,] 0.9948752 0.9957083
[2,] 0.9957083 0.9966999
```

Gaussian Kernel This kernel, also known as the radial basis function kernel, depends on the Euclidean distance between the original attribute value vectors (i.e., the Euclidean norm of their difference) rather than on their dot product, $K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = e^{-\gamma\left\|\boldsymbol{x}_i - \boldsymbol{x}_j\right\|^2} = e^{-\gamma\left[\boldsymbol{x}_i^T \boldsymbol{x}_i - 2\boldsymbol{x}_i^T \boldsymbol{x}_i + \boldsymbol{x}_j^T \boldsymbol{x}_j\right]}$, where γ is a positive real scalar. It is known that the feature vector φ that corresponds to the Gaussian kernel is actually infinitely dimensional (Lin and Lin 2003). Therefore, without the kernel trick, the solution cannot be computed explicitly. This type of kernel tends to be particularly popular, but it is sensitive to the choice of the γ parameter and may be prone to overfitting.

The R code for calculating this kernel is given next:

```
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]) ^2)))}
```

This function is used as

```
> K.radial(X[1,1:4],X[2,1:4])
       [,1]      [,2]      [,3]      [,4]
[1,] 0.9832420 0.9984013 0.9607894 0.8452693
[2,] 0.9879729 0.9245945 0.9984013 0.9715136
[3,] 0.9900498 0.9296938 0.9991004 0.9681193
[4,] 0.9936204 0.9394131 0.9999000 0.9607894
```

while for the full set of features, it can be used as

```
> K.radial(X)
       [,1]      [,2]
[1,] 1.0000000 0.6525288
[2,] 0.6525288 1.0000000
```

The parameter γ controls the flexibility of the Gaussian kernel in a similar way as the degree d in the polynomial kernel. Large values of γ correspond to large values of d since, for example, they allow classifiers to fit any labels, hence risking overfitting. In such cases, the kernel matrix becomes close to the identity matrix. On the other hand, small values of γ gradually reduce the kernel to a constant function, making it impossible to learn any nontrivial classifier. The feature space has infinite dimensions for every value of γ, but for large values, the weight decays very fast on the higher order features. In other words, although the rank of the kernel matrix is full, for all practical purposes, the points lie in a low-dimensional subspace of the feature space.

Exponential Kernel This kernel is defined as $K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = e^{-\gamma\left\|\boldsymbol{x}_i-\boldsymbol{x}_j\right\|}$, which is quite similar to the Gaussian kernel function.

The R code is given below:

```
K.exponential=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
        Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]))))}
```

For individual features in pairs of individuals, it can be used as

```
> K.exponential(X[1,1:4],X[2,1:4])
     [,1]      [,2]      [,3]      [,4]
[1,] 0.8780954 0.9607894 0.8187308 0.6636503
[2,] 0.8958341 0.7557837 0.9607894 0.8436648
[3,] 0.9048374 0.7633795 0.9704455 0.8352702
[4,] 0.9231163 0.7788008 0.9900498 0.8187308
```

while for full set of features, it can be used as

```
> K.exponential(X)
     [,1]      [,2]
[1,] 1.0000000 0.5202864
[2,] 0.5202864 1.0000000
```

Arc-Cosine Kernel (AK) For AK, an important component is the angle between two vectors computed from inputs $\boldsymbol{x}_i$, $\boldsymbol{x}_j$ as

$$\theta_{i,j} = \cos^{-1}\left(\frac{\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j}{\|\boldsymbol{x}_i\|\|\boldsymbol{x}_j\|}\right),$$

where $\|\boldsymbol{x}_i\|$ is the norm of observation i. The following kernel is positive semi-definite and related to an ANN with a single hidden layer and the ramp activation function (Cho and Saul 2009).

$$\mathrm{AK}^1\left(\boldsymbol{x}_i,\boldsymbol{x}_j\right) = \frac{1}{\pi}\|\boldsymbol{x}_i\|\|\boldsymbol{x}_j\|\ J\left(\theta_{i,j}\right), \tag{8.4}$$

where π is the pi constant and $J(\theta_{i,j}) = [\sin(\theta_{i,j}) + (\pi - \theta_{i,j}) \cos(\theta_{i,j})]$. Equation (8.4) gives a symmetric positive semi-definite matrix (AK^1) preserving the norm of the entries such that $\mathrm{AK}(\boldsymbol{x}_i,\boldsymbol{x}_i) = \|\boldsymbol{x}_i\|^2$ and $\mathrm{AK}(\boldsymbol{x}_i, -\boldsymbol{x}_i) = 0$ and models nonlinear relationships.

Note that the diagonals of the AK matrix are not homogeneous and express heterogeneous variances of the genetic value $\boldsymbol{u}$; this is different from the Gaussian kernel matrix, with a diagonal that expresses homogeneous variances. This property could be a theoretical advantage of AK when modeling interrelationships between individuals.

In order to emulate the performance of an ANN with more than one hidden layer (l), Cho and Saul (2009) proposed a recursive relationship of repeating l times the interior product:

$$\mathrm{AK}^{(l+1)}\left(\boldsymbol{x}_i,\boldsymbol{x}_j\right)=\frac{1}{\pi}\left[\mathrm{AK}^{(l)}(\boldsymbol{x}_i,\boldsymbol{x}_i)\mathrm{AK}^{(l)}\left(\boldsymbol{x}_j,\boldsymbol{x}_j\right)\right]^{\frac{1}{2}}J\left(\theta_{i,j}^{(l)}\right), \tag{8.5}$$

where $\theta_{i,j}^{(l)}=\cos^{-1}\left\{\mathrm{AK}^{(l)}\left(\boldsymbol{x}_i,\boldsymbol{x}_j\right)\left[\mathrm{AK}^{(l)}(\boldsymbol{x}_i,\boldsymbol{x}_i)\mathrm{AK}^{(l)}\left(\boldsymbol{x}_j,\boldsymbol{x}_j\right)\right]^{-\frac{1}{2}}\right\}$.

Thus, computing $\mathrm{AK}^{(l+1)}$ at level (layer) $l + 1$ is done from the previous layer $\mathrm{AK}^{(l)}$. Computing a bandwidth (the smoothing parameter that controls variance and bias in the output, e.g., the γ parameter in the Gaussian kernel) is not necessary, and the only computational effort required is to compute the number of hidden layers. Cuevas et al. (2019) described a maximum marginal likelihood method used to select the number of hidden layers (l) for the AK kernel. It is important to point out that this kernel method is like a deep neural network since it allows using more than one hidden layer.

The R code for the AK kernel with one hidden layer is given below:

```
K.AK1_Final<-function(x1,x2){
 n1<-nrow(x1)
 n2<-nrow(x2)
 x1tx2<-x1%*%t(x2)
 norm1<-sqrt(apply(x1,1,function(x) crossprod(x)))
 norm2<-sqrt(apply(x2,1,function(x) crossprod(x)))
 costheta = diag(1/norm1)%*%x1tx2%*%diag(1/norm2)
 costheta[which(abs(costheta)>1,arr.ind = TRUE)] = 1
 theta<-acos(costheta)
 normx1x2<-norm1%*%t(norm2)
 J = (sin(theta)+(pi-theta)*cos(theta))
 AK1 = 1/pi*normx1x2*J
 AK1<-AK1/median(AK1)
 colnames(AK1)<-rownames(x2)
 rownames(AK1)<-rownames(x1)
 return(AK1)
}
```

For the full set of features, it can be used as

```
> K.AK1_Final(x1=X,x2=X)
      [,1]    [,2]
[1,] 0.9709  1.000000
[2,] 1.0000  1.042699
```

Since the K.AK1_Final() kernel function is only useful for one hidden layer, for this reason, the next part of the code extends this to more than one hidden layer.

```
####Kernel Arc-Cosine with deep=4#####
diagAK_f<-function(dAK1)
{
 AKAK = dAK1^2
 costheta = dAK1*AKAK^(-1/2)
 costheta[which(costheta>1,arr.ind = TRUE)] = 1
 theta = acos(costheta)
 AKl  = (1/pi)*(AKAK^(1/2))*(sin(theta)+(pi-theta)*cos(theta))
 AKl
 AKl<-AKl/median(AKl)
}
AK_L_Final<-function(AK1,dAK1,nl){
 n1<-nrow(AK1)
 n2<-ncol(AK1)
 AKl1 = AK1
 for ( l in 1:nl){

  AKAK<-tcrossprod(dAK1,diag(AKl1))

  costheta<-AKl1*(AKAK^(-1/2))
  costheta[which(costheta>1,arr.ind = TRUE)] = 1
  theta <-acos(costheta)

  AKl<-(1/pi)*(AKAK^(1/2))*(sin(theta)+(pi-theta)*cos(theta))
  dAKl = diagAK_f(dAK1)
  AKl1 = AKl
  dAK1 = dAKl
 }
 AKl<-AKl/median(AKl)
 rownames(AKl)<-rownames(AK1)
 colnames(AKl)<-colnames(AK1)
 return(AKl)
}
```

Next, we illustrate how to use this kernel function for an AR kernel with four hidden layers:

```
> AK1=K.AK1_Final(x1=X,x2=X)
> AK_L_Final(AK1= AK1,dAK1=diag(AK1),nl=4)
          [,1]      [,2]
[1,] 0.9649746 1.000000
[2,] 1.0000000 1.036335
```

Hybrid Kernel We understand by hybrid kernels when two or more kernels are combined, since complex kernels can be created by simple operations (multiplication, addition, etc.) that combine simpler kernels. An example of a hybrid kernel can be obtained by multiplying the polynomial kernel and the Gaussian kernel. This kernel is defined as $\left(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j + a\right)^d e^{-\gamma\left\|\boldsymbol{x}_i-\boldsymbol{x}_j\right\|}$. However, other types of kernels can also be combined in the same fashion or with other basic operations, like kernel averaging, which is explained next.

Kernel Averaging Averaging is another way to create hybrid kernels, since kernel methods do not preclude the use of several kernels together (de los Campos et al. 2010). To illustrate the construction of these kernels, we assume that we have three kernels $\boldsymbol{K}_1$, $\boldsymbol{K}_2$, and $\boldsymbol{K}_3$ that are distinct from each other. In this approach, the three kernels are "averaged" to form a new kernel $\boldsymbol{K} = \boldsymbol{K}_1 \frac{\sigma^2_{K_1}}{\sigma^2_K} + \boldsymbol{K}_2 \frac{\sigma^2_{K_2}}{\sigma^2_K} + \boldsymbol{K}_3 \frac{\sigma^2_{K_3}}{\sigma^2_K}$, where $\sigma^2_{K_1}, \sigma^2_{K_2}, \sigma^2_{K_3}$ are variance components attached to kernels $\boldsymbol{K}_1$, $\boldsymbol{K}_2$, and $\boldsymbol{K}_3$, respectively, and σ^2_K is the sum of the three variances. The ratios of the three variance components are tantamount to the relative contributions of the kernels. For instance, the kernels used can be three Gaussian kernels with different bandwidth parameter values, as employed in Tusell et al. (2013), or one can fit several parametric kernels jointly, e.g., the additive (**G**), dominance (**D**), and additive by dominance (**G#D**) kernels, as in Morota et al. (2014). While there are many possible choices of kernels, the kernel function can be estimated via maximum likelihood by recourse to the Matérn family of covariance functions (e.g., Ober et al. 2011) or by fitting several candidate kernels simultaneously through multiple kernel learning.

Hybrid kernels illustrate a general principle of how more complex kernels can be created from simpler ones in a number of different ways. Kernels can even be constructed that correspond to infinite-dimensional feature spaces at the cost of only a few extra operations in the kernel evaluations, like the Gaussian kernel which most often is good enough (Ober et al. 2011). There are many other kernels, however, the above-mentioned kernels are the most popular. For example, Morota et al. (2013) evaluated diffusion kernels for discrete inputs with animal and plant data, and compared these to the Gaussian kernel. Differences in predictive ability were minimal; this is fortunate because computing diffusion kernels is time-consuming.

The bandwidth parameter can be selected based on (1) a cross-validation procedure, (2) restricted maximum likelihood (Endelman 2011), and (3) an empirical Bayesian method such as the one proposed by Pérez-Elizalde et al. (2015). The optimal value of the bandwidth parameter is expected to change with many factors such as (a) distance function, (b) number of markers, allelic frequency, and coding of markers, all markers affecting the distribution of observed distances, and (c) genetic architecture of the trait, a factor affecting the expected prior correlation of genetic values (de los Campos et al. 2010).

As pointed out above, kernel methods only need information of the kernel function $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$, assuming that this has been defined. For this reason, nonvectorial patterns $\mathbf{x}$ such as sequences, trees, and graphs can be handled. That is, kernel functions are not restricted to vectorial inputs: kernels can be designed for objects and structures as diverse as strings, graphs, text documents, sets, and graph nodes. It is important to point out that the kernel trick can be applied in unsupervised methods like cluster analysis, and dimensionality reduction methods like principal component analysis, independent component analysis, etc.

8.2.5 A Two Separate Step Process for Building Kernel Machines

The goal of this section is to emphasize that the building process of kernel machines consists of two general independent steps. The first one consists of calculating the Gram matrix (kernel matrix $\boldsymbol{K}$) using only the information of the independent variables (input). This means that in this process the user needs to define the type of kernel function that he (she) will use in such a way as to capture the hypothesized nonlinear patterns in the input data. Then in the second step, after the kernel is ready, we select the statistical machine learning algorithm that will be used for training the model using the dependent variable, the kernel built in the first step and other available covariates. These two separate steps for building kernel methods for prediction imply that we can use conventional linear statistical machine learning algorithms to accommodate a particular type of kernel function. The only important consideration when choosing the kernel is that it should be suitable for the data at hand. But, if you built the kernel, you can evaluate the performance of this kernel with many other statistical machine learning methods. This illustrates the two separate steps required for training predictive machines using kernel methods where any statistical machine learning method can be combined with any kernel function. It is important to point out that since many machine learning methods are only able to work with linear patterns, using the kernel trick allows you to build nonlinear versions of the linear algorithms, without the need to modify the original machine learning algorithm. The following sections show how the kernel trick works in some standard statistical machine learning models.

8.3 Kernel Methods for Gaussian Response Variables

When the response variable is Gaussian, the negative log-likelihood that needs to be used to minimize expression (8.3) belongs to a normal distribution and the expression (8.3) is reduced to

$$\underbrace{\min}_{\eta_0,\boldsymbol{\beta}}\left\{\frac{1}{n}\sum_{i=1}^{n}\left(y_i-\eta_0-\boldsymbol{k}_i^{\mathrm{T}}\boldsymbol{\beta}\right)^2+\frac{\lambda}{2}\boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\beta}\right\}.$$

In matrix notation, the latter expression can be expressed as

$$\underbrace{\min}_{\eta_0,\boldsymbol{\beta}}\left\{\frac{1}{2}(\boldsymbol{y}^*-\boldsymbol{K}\boldsymbol{\beta})^{\mathrm{T}}(\boldsymbol{y}^*-\boldsymbol{K}\boldsymbol{\beta})+\frac{\lambda}{2}\boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\beta}\right\},$$

where $\boldsymbol{y}^*=\boldsymbol{y}-\mathbf{1}\overline{y}$, using $\overline{y}$ as an estimator of the intercept (η_0). The first-order conditions to this problem are familiar to us (see Chap. 3) and are

$$\left[\boldsymbol{K}^{\mathrm{T}}\boldsymbol{K}+\lambda\boldsymbol{K}\right]\boldsymbol{\beta}=\boldsymbol{K}^{\mathrm{T}}\boldsymbol{y}^{*}$$

Further, since $\boldsymbol{K}=\boldsymbol{K}^{\mathrm{T}}$ and $\boldsymbol{K}^{-1}$ exist, pre-multiplication by $\boldsymbol{K}^{-1}$ yields

$$\boldsymbol{\beta}=[\boldsymbol{K}+\lambda\boldsymbol{I}]^{-1}\boldsymbol{y}^{*},$$

where $\boldsymbol{I}$ is an identity matrix of dimension $n \times n$, and to estimate $\boldsymbol{\beta}$, λ must be known. It is important to point out that even in the context of large p and small n, the number of beta coefficients ($\boldsymbol{\beta}$) that need to be estimated is equal to n, which considerably reduces the computation resources in the estimation process. This solution to the beta coefficients obtained under Gaussian response variables is known as kernel Ridge regression in statistical machine learning, and was first obtained by Gianola et al. (2006) and Gianola and van Kaam (2008) in the context of a mixed effects model under a Bayesian treatment. The predicted values in the original scale of the response variables can be obtained as

$$\widehat{\boldsymbol{y}}=\mathbf{1}\overline{y}+\boldsymbol{K}\widehat{\boldsymbol{\beta}}.$$

For a new observation with vector of inputs ($\boldsymbol{x}_{\text{new}}$), the predictions are made using the following expression:

$$\widehat{\boldsymbol{y}}_{\text{new}}=\overline{y}+\sum_{i=1}^{n}\widehat{\beta}_{i}K(\boldsymbol{x}_{i},\boldsymbol{x}_{\text{new}})$$

Next, we provide some examples of Gaussian response variables using different kernel methods.

Example 1 for continuous response variables. The data comprise family, marker, and phenotypic information of 599 lines that were evaluated for grain yield (GY) in four environments. Marker information consisted of 1447 Diversity Array Technology (DArT) markers, generated by Triticarte Pty. Ltd. (Canberra, Australia). Also, this data set contains the pedigree relationship matrix and is preloaded in the BGLR package with the name wheat. We named this data set the wheat599 data set. The GY measured in the four environments was used for single environment analysis using various kernel methods.

The first six observations for trait GY in the four environments (labeled 1, 2, 4, and 5) are given next.

```
> head(y)
             1           2           4          5
775   1.6716295 -1.72746986 -1.89028479  0.0509159
2166 -0.2527028  0.40952243  0.30938553 -1.7387588
2167  0.3418151 -0.64862633 -0.79955921 -1.0535691
2465  0.7854395  0.09394919  0.57046773  0.5517574
```

```
3881 0.9983176 -0.28248062 1.61868192 -0.1142848
3889 2.3360969 0.62647587 0.07353311 0.7195856
```

Also, next are given the first six observations for five standardized markers

```
> head(XF[,1:5])
     wPt.0538   wPt.8463  wPt.6348 wPt.9992  wPt.2838
[1,] -1.3598855 0.2672768 0.772228 0.4419075 0.439209
[2,]  0.7341284 0.2672768 0.772228 0.4419075 0.439209
[3,]  0.7341284 0.2672768 0.772228 0.4419075 0.439209
[4,] -1.3598855 0.2672768 0.772228 0.4419075 0.439209
[5,] -1.3598855 0.2672768 0.772228 0.4419075 0.439209
[6,]  0.7341284 0.2672768 0.772228 0.4419075 0.439209
```

Then with the code given in Appendix 1, that uses the wheat599 data set, the nine kernels explained above were illustrated. We implemented the kernel Ridge regression method using the library glmnet. The results of the nine kernels for GY in each of the four environments are given next.

Table 8.1 indicates that the best predictions were observed in the four environments under the Sigmoid kernel and the worst under the polynomial kernel.

8.4 Kernel Methods for Binary Response Variables

When the response variable is binary, instead of using the sum of squares loss function that was used before for continuous response variables, we now use the negative log-likelihood of the product of Bernoulli distributions, and the expression that needs to be minimized is given next:

$$\underbrace{\min}_{\eta_0,\boldsymbol{\beta}}\left\{\frac{1}{n}\sum_{i=1}^{n}\left(y_i\left(\eta_0+\boldsymbol{k}_i^{\mathrm{T}}\boldsymbol{\beta}\right)+\log\left[1+\exp\left(\eta_0+\boldsymbol{k}_i^{\mathrm{T}}\boldsymbol{\beta}\right)\right]\right)^2+\frac{\lambda}{2}\boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\beta}\right\}$$

Estimation of the parameters η_0 and $\boldsymbol{\beta}$ requires an iterative procedure, and gradient descent methods are used for their estimation, like those explained for logistic regression in Chaps. 3 and 7. Here, for the examples, we will use the glmnet package.

In Table 8.2, we can observe the prediction performance using the binary trait Height of the Data_Toy_EYT.R with nine kernels (linear, polynomial, sigmoid, Gaussian, exponential, AK1, AK2, AK3, and AK4). The Data_Toy_EYT data set contains 160 observations with 40 in each of the four environments that are present. The phenotypic information consists of a column for lines, another for environments and four corresponding to traits, two measured on a categorical scale, one continuous, and the last one binary. The data set also contains a genomic relationship matrix of the 40 lines that were evaluated in each of the four environments. Ten fold cross-

Table 8.1 Prediction performance in terms of the mean square error for GY in the four environments (Env) under nine kernel methods

Env	Linear	Polynomial	Sigmoid	Gaussian	Exponential	AK1	AK2	AK3	AK4
1	1.009	1.001	**0.762**	0.893	0.892	0.882	0.881	0.886	0.891
2	0.916	1.069	**0.772**	0.922	0.905	0.841	0.899	0.901	0.909
4	0.984	1.004	**0.859**	0.940	0.950	0.941	0.936	0.943	0.945
5	1.019	1.002	**0.814**	0.901	0.949	0.868	0.876	0.885	0.894

Table 8.2 Prediction performance in terms of the proportion of cases correctly classified (PCCC) with ten fold cross-validation for the binary trait Height of the Data_Toy_EYT.R data set with nine kernels

Fold	Linear	Polynomial	Sigmoid	Gaussian	Exponential	AK1	AK2	AK3	AK4
1	0.813	0.813	0.688	0.813	0.813	0.813	0.813	0.813	0.813
2	0.688	0.688	0.563	0.688	0.688	0.688	0.688	0.688	0.688
3	0.688	0.750	0.563	0.625	0.625	0.625	0.625	0.625	0.750
4	0.625	0.750	0.375	0.750	0.750	0.625	0.750	0.625	0.750
5	0.750	0.750	0.625	0.750	0.750	0.750	0.688	0.750	0.688
6	0.625	0.688	0.625	0.625	0.625	0.625	0.625	0.625	0.688
7	0.750	0.750	0.500	0.750	0.750	0.750	0.750	0.750	0.750
8	0.813	0.813	0.563	0.813	0.813	0.813	0.813	0.813	0.813
9	0.688	0.688	0.625	0.688	0.688	0.688	0.688	0.688	0.688
10	0.813	0.875	0.688	0.813	0.813	0.813	0.813	0.750	0.813
Average	0.725	0.756	0.581	0.731	0.731	0.719	0.725	0.713	0.744

validation was implemented and the worst performance in terms of the proportion of cases correctly classified (PCCC) was with the sigmoid kernel and the best under the polynomial and AK4 kernels. The R code for reproducing the results in Table 8.2 is given in Appendix 2.

8.5 Kernel Methods for Categorical Response Variables

For categorical response variables, the loss function is the negative log-likelihood of the product of multinomial distributions and the expression that needs to be minimized is given next:

$$\underbrace{\min}_{\eta_0,\boldsymbol{\beta}}\left\{-\frac{1}{n}\left(\sum_{i=1}^{n}\sum_{c=1}^{C}I_{\{y_i=c\}}\left(\eta_{0c}+\boldsymbol{k}_i^{\mathrm{T}}\boldsymbol{\beta}_c\right)-\sum_{i=1}^{n}\log\left[\sum_{l=1}^{C}\exp\left(\eta_{0l}+\boldsymbol{k}_i^{\mathrm{T}}\boldsymbol{\beta}_l\right)\right]\right)+\frac{\lambda}{2}\sum_{l=1}^{C}\boldsymbol{\beta}_l^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\beta}_l\right\}$$

The estimation process does not have an analytical solution, and gradient descent methods are used for the estimation of the required parameters. The optimization process is done with the same methods described in Chaps. 3 and 7 for categorical response variables. The following illustrative examples were implemented using the glmnet library.

Now the Data_Toy_EYT.R data set was used that was also used for illustrating kernels with binary response variables. The nine kernels were implemented but with the categorical response variable days to heading (DTHD). Again, the worst predictions occurred with the sigmoid kernel, but now the best predictions were achieved with the AK2 kernel (Table 8.3). The code given in Appendix 2 can be used for reproducing these results with two small modifications: (a) replace the response variable y2=Pheno$Height with y2=Pheno$DTHD and (b) in the specification of the model in glmnet, replace family='binomial' with family='multinomial'.

8.6 The Linear Mixed Model with Kernels

Under a linear mixed model (LMM) ($\boldsymbol{y} = \boldsymbol{C\theta} + \boldsymbol{K\beta} + \boldsymbol{e}$), every individual i is associated with a genotype vector $\boldsymbol{x}_i^{\mathrm{T}}$ and a covariate vector $\boldsymbol{c}_i^{\mathrm{T}}$ (e.g., gender, age, herd, race, environment, etc.). Given a sample of individuals with a genotyped variants matrix $\boldsymbol{X} = \left[\boldsymbol{x}_1^{\mathrm{T}}, \boldsymbol{x}_2^{\mathrm{T}}, \ldots, \boldsymbol{x}_n^{\mathrm{T}}\right]^{\mathrm{T}}$ and matrix of incidence nuisance variables $\boldsymbol{C} = \left[\boldsymbol{c}_1^{\mathrm{T}}, \boldsymbol{c}_2^{\mathrm{T}}, \ldots, \boldsymbol{c}_n^{\mathrm{T}}\right]^{\mathrm{T}}$, relating some effect ($\boldsymbol{\theta}$) to the phenotype vector $\boldsymbol{y} = [y_1, y_2, \ldots, y_n]^{\boldsymbol{T}}$ that follows a multivariate normal distribution.

$$\boldsymbol{y}|\boldsymbol{X},\boldsymbol{C} \sim N\left(\boldsymbol{C\theta}, \boldsymbol{K} + \boldsymbol{I}\sigma_e^2\right)$$

Table 8.3 Prediction performance in terms of the proportion of cases correctly classified (PCCC) with ten fold cross-validation for the categorical trait days to heading (DTHD) of the Data_Toy_EYT.R data set with nine kernels

Fold	Linear	Polynomial	Sigmoid	Gaussian	Exponential	AK1	AK2	AK3	AK4
1	0.813	0.813	0.688	0.813	0.813	0.813	0.813	0.813	0.813
2	0.813	0.813	0.625	0.813	0.813	0.813	0.813	0.813	0.813
3	0.688	0.688	0.750	0.688	0.688	0.688	0.688	0.688	0.688
4	0.875	0.813	0.750	0.875	0.875	0.875	0.875	0.875	0.875
5	0.688	0.625	0.625	0.688	0.625	0.625	0.688	0.688	0.625
6	0.750	0.750	0.563	0.750	0.750	0.750	0.750	0.750	0.750
7	0.688	0.750	0.563	0.688	0.688	0.688	0.688	0.688	0.688
8	0.875	0.875	0.875	0.875	0.875	0.875	0.875	0.875	0.875
9	0.813	0.813	0.688	0.813	0.813	0.813	0.813	0.813	0.813
10	0.625	0.688	0.500	0.625	0.688	0.625	0.688	0.625	0.625
Average	0.763	0.763	0.663	0.763	0.763	0.756	0.769	0.763	0.756

Here, $\boldsymbol{K}$ is a valid kernel encoding genotypic covariance, as long as it is positive semi-definite and, again, represents similarities between genotyped individuals. Now the nonparametric function is $\boldsymbol{f}(\boldsymbol{X}) = \boldsymbol{K\beta}$ and the nonparametric coefficients, $\boldsymbol{\beta}$, and residuals can be assumed to be independently distributed as $\boldsymbol{\beta} \sim \boldsymbol{N}\left(\boldsymbol{0}, \boldsymbol{K}^{-1}\sigma_\beta^2\right)$ and $\boldsymbol{e} \sim \boldsymbol{N}\left(\boldsymbol{0}, \boldsymbol{I}\sigma_e^2\right)$. $\boldsymbol{\theta}$ is a vector of covariate coefficients (denoted as fixed effects), $\boldsymbol{I}$ is the $n \times n$ identity matrix, and σ_e^2 is the variance of the microenvironmental effects. Now under this LMM approach, the function to be minimized becomes

$$\underbrace{\min}_{\boldsymbol{\theta},\boldsymbol{\beta}} \quad J[\boldsymbol{\theta},\boldsymbol{\beta}|\lambda] = \underbrace{\min}_{\boldsymbol{\theta},\boldsymbol{\beta}} \quad \left\{\frac{1}{2\sigma_e^2}[\boldsymbol{y} - \boldsymbol{C\theta} - \boldsymbol{K\beta}]^{\mathrm{T}}[\boldsymbol{y} - \boldsymbol{C\theta} - \boldsymbol{K\beta}] + \frac{\lambda}{2}\boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{K\beta}\right\}.$$

After setting the gradient of $J(.)$ with respect to $\boldsymbol{\theta}$ and $\boldsymbol{\beta}$ simultaneously to zero (Mallick et al. 2005; Gianola et al. 2006; Gianola and van Kaam 2008), the RKHS regression estimating equations can be formulated in matrix form given σ_e^2 and λ as

$$\begin{bmatrix} \boldsymbol{C}^{\mathrm{T}}\boldsymbol{C} & \boldsymbol{C}^{\mathrm{T}}\boldsymbol{K} \\ \boldsymbol{K}^{\mathrm{T}}\boldsymbol{C} & \boldsymbol{K}^{\mathrm{T}}\boldsymbol{K} + \lambda\boldsymbol{K}\sigma_e^2 \end{bmatrix} \begin{bmatrix} \widehat{\boldsymbol{\theta}} \\ \widehat{\boldsymbol{\beta}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{C}^{\mathrm{T}}\boldsymbol{y} \\ \boldsymbol{K}^{\mathrm{T}}\boldsymbol{y} \end{bmatrix} \tag{8.6}$$

Recall that $\boldsymbol{K}$ is symmetric, so $\boldsymbol{K}^{\mathrm{T}}\boldsymbol{K} = \boldsymbol{K}^2$, and by multiplying the second system of (8.6) by $\boldsymbol{K}^{-1}$ (assuming the inverse exists), we obtain

$$\begin{bmatrix} \boldsymbol{C}^{\mathrm{T}}\boldsymbol{C} & \boldsymbol{C}^{\mathrm{T}}\boldsymbol{K} \\ \boldsymbol{I}^{\mathrm{T}}\boldsymbol{C} & \boldsymbol{K} + \lambda\boldsymbol{I}\sigma_e^2 \end{bmatrix} \begin{bmatrix} \widehat{\boldsymbol{\theta}} \\ \widehat{\boldsymbol{\beta}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{C}^{\mathrm{T}}\boldsymbol{y} \\ \boldsymbol{y} \end{bmatrix} \tag{8.7}$$

This avoids inverting $\boldsymbol{K}$ and forming $\boldsymbol{K}^{\mathrm{T}}\boldsymbol{K}$. Note that the variance of the nonparametric coefficient $\sigma_\beta^2 = \lambda^{-1}$ may be interpreted as variation due to marked additive genomic variation.

The mixed model $\boldsymbol{y} = \boldsymbol{C\theta} + \boldsymbol{K\beta} + \boldsymbol{e}$ (reparametrization I) can be reparametrized as $\boldsymbol{y} = \boldsymbol{C\theta} + \boldsymbol{u} + \boldsymbol{e}$ (reparametrization II), where $\boldsymbol{u} = \boldsymbol{K\beta}$, but with $\boldsymbol{u}$ distributed as $\boldsymbol{u} \sim N\left(\boldsymbol{0}, \boldsymbol{K}\sigma_u^2\right)$, and σ_u^2 is the additive variance due to lines. Both parametrizations produce the same solution since they are equivalent, with the following peculiarities. Parametrization I has two main advantages: (1) kernel matrix $\boldsymbol{K}$ does not need to be inverted. The inverse of kernel matrix $\boldsymbol{K}$ may be time-consuming or unfeasible if the number of genotyped individuals is large, because the matrix is too dense. Currently, there is the need to invert the matrix up to 100,000×100,000. (2) Genome-enabled prediction of breeding values for any t new genotyped individuals ($\widehat{\boldsymbol{u}}_{\text{new}}$) without phenotype can be done using a simple matrix–vector product $\widehat{\boldsymbol{u}}_{\text{new}} = \boldsymbol{K}_s\widehat{\boldsymbol{\beta}}$, where $\widehat{\boldsymbol{\beta}}$ are the n nonparametric coefficients estimated from the n individuals in the training set, $\boldsymbol{K}_s$ is a matrix of dimension ($t \times n$) containing the genomic similarity values between the t new individuals whose direct genomic merits are to be predicted and the individuals in the training set. When $\boldsymbol{K}$ is the genomic relationship matrix calculated as suggested by VanRaden (2008), the kernel is linear, but when $\mathbf{K}$ is calculated with nonlinear kernels such as the Gaussian, exponential, polynomial, arc-cosine, sigmoid, etc., the same model can be used to capture nonlinear patterns better. This means that with the mixed model equations given above, it is possible to implement any of the proposed kernels since the only difference between them is the transformation performed on the input information to obtain a particular kernel. This means that the genomic relationship matrix used in a method known as genomic BLUP (GBLUP) is replaced by a more general kernel matrix that creates similarities among individuals, even if genetically unrelated. However, for a particular data set, some kernels will perform better and others worse, since the performance of the kernels is data-dependent. In our context, a kernel is any smooth function **K** defining a covariance structure among individuals. Also, as mentioned above, we can mix many kernels using a weighted sum or product of them to create new kernels. In general, as mentioned many times in this chapter, there is enough empirical evidence that kernel methods outperform conventional regression methods that are only able to capture linear patterns (Tusell et al. 2013; Long et al. 2010; Morota et al. 2013, 2014).

The solution of the mixed model equations given in (8.6 and 8.7) can be obtained using the rrBLUP package (Endelman 2011). This package is not restricted only to linear kernels since it is also useful for estimating marker effects by Ridge regression, and BLUPs calculated based on an additive relationship matrix. In this package, variance components are estimated by either maximum likelihood (ML) or restricted maximum likelihood (REML; default) using the spectral decomposition algorithm of Kang et al. (2008). The *R* function returns the variance components, the maximized log-likelihood (LL), the ML estimate for $\boldsymbol{\theta}$, and the BLUP solution for $\boldsymbol{u}$.

The basic function for implementing the kernel methods using the rrBLUP package is given next:

```
   mixed.solve(y=y, Z=Z, K=K, X=X, method="REML"),
```

where $\boldsymbol{y}$, $\boldsymbol{Z}$, $\boldsymbol{K}$, and $\boldsymbol{X}$ are the vector of response variables, the design matrix of random effects, the kernel matrix, and the design matrix of fixed effects, respectively. The estimation method by default is REML, but the ML method is also allowed. The kernel matrix is calculated before using the mixed.solve() function. It is important to point out that the function kinship.BLUP allows implementing three kernel methods directly [linear kernel (RR), Gaussian kernel (GAUSS), and Exponential kernel (EXP)] under a mixed model approach.

```
   kinship.BLUP(y=y[trn], G.train=W[trn,], G.pred=W[tst,], X=X
[trn,], K.method="GAUSS", mixed.method="REML"),
```

where **y**[trn] contains the training part of the response variable, **W**[trn,] contains the markers corresponding to the training set, **W**[tst,] contains the testing set of marker data, **X**[trn,] contains the fixed effects corresponding to the training set, the K. method="GAUSS" specifies that the Gaussian kernel will be implemented, and finally, mixed.method="REML" specifies any of the two estimation methods

Table 8.4 Prediction performance in terms of mean square error (MSE) and Pearson's correlation (PC) for each fold for the wheat599 (Environment 4) data set under a mixed model and three kernels: linear, Gaussian, and Exponential. These kernels are the defaults programmed in the rrBLUP library

	Linear	Gaussian	Exponential	Linear	Gaussian	Exponential
Fold	MSE	MSE	MSE	PC	PC	PC
1	0.757	0.694	0.720	0.534	0.592	0.610
2	0.667	0.626	0.630	0.536	0.583	0.590
3	0.774	0.685	0.700	0.435	0.521	0.490
4	0.736	0.609	0.649	0.398	0.535	0.509
5	0.677	0.690	0.685	0.454	0.428	0.426
6	1.139	1.036	1.025	0.309	0.395	0.405
7	1.006	0.966	1.010	0.486	0.523	0.514
8	0.874	0.758	0.752	0.486	0.594	0.616
9	0.683	0.592	0.586	0.526	0.621	0.625
10	0.729	0.683	0.689	0.315	0.366	0.365
Average	0.804	0.734	0.745	0.448	0.516	0.515

REML or ML. As mentioned above, this kinship.BLUP() allows implementing three kernels: linear, Gaussian, and Exponential. Using the wheat599 data set used in the last examples, a ten fold cross-validation using the three default kernels was implemented.

Table 8.4 shows that under a mixed model approach using the default kernels [linear (RR), Gaussian (GAUSS), and Exponential (EXP)] available in the rrBLUP library, the best predictions were observed in Env4 of data set wheat599 with the Gaussian and Exponential kernels under both metrics. The R code for reproducing the results in Table 8.4 is given in Appendix 3.

Table 8.5 provides the results of nine kernels for the response variable of Env4. The building process was manual for the kernel matrices and the data set used for this example was the wheat599 data set. Results for each of the nine kernels are given for each fold and across the 10 folds. Table 8.5 shows that the best prediction performance was observed with the Gaussian kernel and the worst under the polynomial kernel. The R code for reproducing the results in Table 8.5 is given in Appendix 4.

8.7 Hyperparameter Tuning for Building the Kernels

Hand-tuning kernel functions can be time-consuming and requires expert knowledge. A tuned kernel can improve the trained model, if standard kernels are insufficient for achieving a good transformation. In this section, we illustrate how to tune kernels and we compare the standard kernel with a hand-tuned kernel. As pointed out in Chap. 4, one approach to tuning is to divide the data into a training set, a tuning set, and a testing set. The training set is for training the data, the tuning set is for

Table 8.5 Prediction performance in terms of mean square error (MSE) for each fold for the wheat599 (Environment 4) data set under a mixed model, manually building the kernels linear, polynomial, sigmoid, Gaussian, exponential, AK1, AK2, AK3, and AK4

Fold	Linear	Polynomial	Sigmoid	Gaussian	Exponential	AK1	AK2	AK3	AK4
1	0.763	0.848	0.772	0.688	0.691	0.714	0.694	0.693	0.693
2	0.667	0.737	0.664	0.632	0.630	0.645	0.630	0.627	0.626
3	0.780	0.817	0.799	0.701	0.725	0.728	0.713	0.714	0.716
4	0.736	0.672	0.756	0.596	0.655	0.670	0.639	0.634	0.631
5	0.682	0.727	0.703	0.722	0.695	0.681	0.701	0.707	0.710
6	1.139	1.077	1.176	1.015	1.051	1.094	1.057	1.047	1.040
7	1.012	1.078	1.023	0.987	0.986	0.977	0.969	0.971	0.975
8	0.875	0.927	0.899	0.758	0.756	0.805	0.767	0.761	0.758
9	0.684	0.742	0.708	0.590	0.602	0.630	0.608	0.606	0.606
10	0.730	0.696	0.729	0.677	0.701	0.725	0.705	0.700	0.697
Average	0.807	0.832	0.823	0.736	0.749	0.767	0.748	0.746	0.745

choosing the best hyperparameter combination, and the testing set is for evaluating the prediction performance with the best hyperparameters. However, when the data sets are small after selecting the best combination of hyperparameters, the training and tuning sets are joined into one data set and with this data set the model is refitted again with the best combination of hyperparameters, and finally, the prediction performance is evaluated with the testing set.

This conventional approach to tuning is illustrated next using the wheat599 data set. To illustrate how to choose the hyperparameter, we will work with the arc-cosine kernel where the hyperparameter to be tuned is the number of hidden layers, for which we used a grid of 10 values (1, 2,. . ., 9, 10). To be able to tune the number of hidden layers, first we divided the original data set into four mutually exclusive parts with four fold cross-validation. This is called the outer cross-validation strategy. Then three of these parts were used for training and the remaining for testing. A ten fold cross-validation was performed in each of the outer training sets; this is called inner cross-validation. Nine out of the ten formed the inner training set and the remaining the tuning set. Then for each of the outer folds, the grid was evaluated with 10 values in the grid for the number of hidden layers, with the inner ten fold cross-validation strategy and for each of the 10 tuning sets, the mean square error of prediction was computed for each of the 10 values of the hidden layers in the grid. Then the average mean square error of the 10 inner cross-validations (in each outer fold) was computed for each value in the grid; it was selected as the optimal number of hidden layers in the grid that provides the smallest MSE. Then the inner training and the tuning sets (inner testing) were joined together for refitting the model with the optimal number of hidden layers, and finally, the prediction performance also in terms of MSE was evaluated in the outer testing set. The average of the four values of the outer testing set is reported as the final prediction performance. Under this strategy, the optimal number of hidden layers is different for each fold.

Figure 8.2 shows that the optimal number of hidden layers is different in each fold. In folds 1 and 3 (Fig. 8.2a, c), the optimal number of hidden layers was equal to 3, in folds 3 and 4 (Fig. 8.2b, d), the optimal number of hidden layers was equal to 8. Finally, with these optimal values, the model was refitted with the information of the inner training + tuning set, and then for each fold, the mean square error (MSE) was calculated for each outer testing set; the MSEs were 0.6878 (fold 1), 0.6963 (fold 2), 0.9725 (fold 3), and 0.7212 (fold 4), with an MSE across folds equal to 0.7694. The R code for reproducing these results is given in Appendix 5.

8.8 Bayesian Kernel Methods

For a single environment, the model can be expressed as

$$\boldsymbol{y} = \mu\mathbf{1} + \boldsymbol{u} + \boldsymbol{e}, \tag{8.8}$$

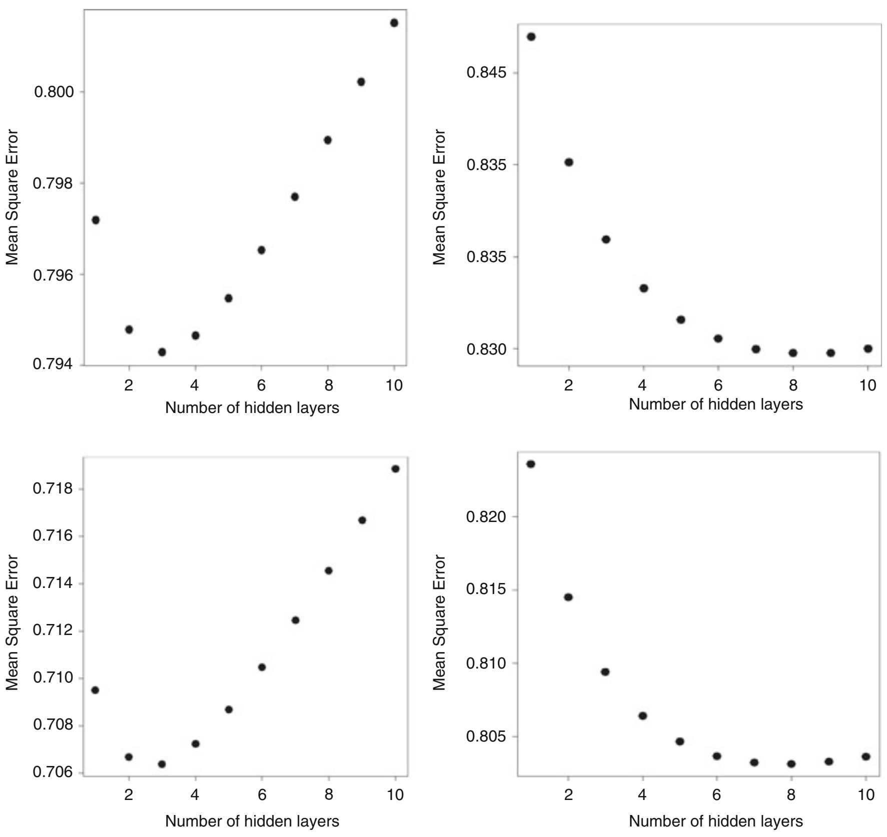

Fig. 8.2 Optimal number of hidden layers in each fold using a grid with values 1 to 10 and an inner ten fold cross-validation

where μ is the overall mean, $\mathbf{1}$ is the vector of ones, and $\boldsymbol{y}$ is the vector of observations of size n. Moreover, $\boldsymbol{u}$ is the vector of genomic effects $\boldsymbol{u} \sim \boldsymbol{N}\left(\mathbf{0}, \sigma_u^2 \boldsymbol{K}\right)$, where σ_u^2 is the genomic variance estimated from the data, and matrix $\boldsymbol{K}$ is the kernel constructed with any of the kernel methods explained above (linear, polynomial, sigmoid, Gaussian, exponential, AK1, AK2, ...). The random residuals are assumed independent with normal distribution $\boldsymbol{e} \sim \boldsymbol{N}\left(\mathbf{0}, \sigma_e^2 \boldsymbol{I}\right)$, where σ_e^2 is the error variance.

Now the kernel Ridge regression is cast under a Bayesian framework with $\lambda = \sigma_e^2/\sigma_u^2$, where σ_e^2 and σ_u^2 are the residual and variance attached to $\boldsymbol{u}$, respectively. With a flat prior to mean parameter (μ), $\sigma_e^2 \sim \chi_{v,S}^{-2}$, and the induced priors $\boldsymbol{u} \mid \sigma_u^2 \sim N_n\left(\mathbf{0}, \boldsymbol{K}\sigma_g^2\right)$ and $\sigma_u^2 \sim \chi_{v_u,\ S_u}^{-2}$, the full conditional posterior distribution of $\boldsymbol{u}$ in model (8.8) is given by

$$\begin{aligned} f(\boldsymbol{u}|-) &\propto L\left(\mu, \boldsymbol{u}, \sigma_e^2; \boldsymbol{y}\right) f\left(\boldsymbol{u}|\sigma_u^2\right) \\ &\propto \frac{1}{\left(2\pi\sigma_e^2\right)^{\frac{n}{2}}} \exp\left[-\frac{1}{2\sigma_e^2}\boldsymbol{y} - 1_n\mu - \boldsymbol{u}^2\right] \frac{1}{\left(\sigma_u^2\right)^{\frac{n}{2}}} \exp\left(\left[-\frac{1}{2\sigma_u^2}\boldsymbol{u}^{\mathrm{T}}\boldsymbol{K}^{-1}\boldsymbol{u}\right]\right) \\ &\propto \exp\left\{-\frac{1}{2}\left[(\boldsymbol{u}-\widetilde{\boldsymbol{u}})^{\mathrm{T}}\widetilde{\boldsymbol{K}}^{-1}(\boldsymbol{u}-\widetilde{\boldsymbol{u}})\right]\right\}, \end{aligned}$$

where $\widetilde{\boldsymbol{K}} = \left(\sigma_u^{-2}\boldsymbol{K}^{-1} + \sigma_e^{-2}\boldsymbol{I}_n\right)^{-1}$ and $\widetilde{\boldsymbol{u}} = \sigma_e^{-2}\widetilde{\boldsymbol{K}}(\boldsymbol{y} - \mathbf{1}_n\mu)$, and from here $\boldsymbol{u} \mid - \sim N_n\left(\widetilde{\boldsymbol{u}}, \widetilde{\boldsymbol{K}}\right)$. Then the mean/mode of $\boldsymbol{u} \mid -$ is $\widetilde{\boldsymbol{u}} = \sigma_e^{-2}\widetilde{\boldsymbol{K}}(\boldsymbol{y} - \mathbf{1}_n\mu)$, which is also the BLUP of $\boldsymbol{u}$ under the mixed model equation of Henderson (1975). For this reason, model (8.8) is often referred to as GBLUP. However, here the genomic relationship matrix (GRM; or pedigree matrix P) was replaced by any kernel $\boldsymbol{K}$; for this reason, under a Bayesian framework, we call this model a Bayesian kernel BLUP, which is reduced to the pedigree (P) or Genomic (G) BLUP when we use the GRM or pedigree matrix as the kernel.

The full conditional posterior of the rest of the parameters is equal to the GBLUP model described in Chap. 6: $\mu \mid - \sim N\left(\widetilde{\mu}, \widetilde{\sigma}_0^2\right)$, where $\widetilde{\sigma}_0^2 = \frac{\sigma^2}{n}$ and $\widetilde{\mu} = \frac{1}{n}\mathbf{1}_n^T(\boldsymbol{y} - \boldsymbol{u})$; $\sigma_e^{-2} \mid - \sim \chi^{-2}_{\widetilde{v},\widetilde{S}}$, where $\widetilde{v} = v + n$ and $\widetilde{S} = S + \|\boldsymbol{y} - \mathbf{1}_n\mu - \boldsymbol{u}\|^2$; and $\sigma_u^2 \mid - \sim \chi^{-2}_{\widetilde{v}_u,\widetilde{S}_u}$, where $\widetilde{v}_u = v_u + n$ and $\widetilde{S}_u = \boldsymbol{u}^T\boldsymbol{K}^{-1}\boldsymbol{u}$. The Bayesian kernel BLUP, like the GBLUP, does not face the large p and small n problem, since due to the kernel trick, a problem of dimensionality p is converted into an n-dimensional problem.

The Bayesian kernel BLUP model (8.8) can also be implemented easily with the BGLR R package, and when the hyperparameters S-v and S_u-v_u are not specified, $v = v_u = 5$ is used by default and the scale parameters are settled as in the BRR. However, a two-step process is required for its implementation: Step 1: Select and compute the kernel matrix to be used. Step 2: Use this kernel matrix to implement the model using the BGLR package.

The BGLR code to fit this model is

```
 ETA = list( list( model = 'RHKS', K = K , df0 = vu, S0 = Su,  R2 = 1-R2)) )
 A   = BGLR(y=y, ETA = ETA, nIter = 1e4, burnIn = 1e3, S0 = S, df0 = v, R2 =
R2)
```

When individuals had more than one replication, or a sophisticated experimental design was used for data collection, the Bayesian kernel BLUP model is specified in a more general way to take into account this structure, as follows:

$$\boldsymbol{Y} = \mathbf{1}_n\mu + \boldsymbol{Z}\boldsymbol{u} + \boldsymbol{e} \qquad (8.9)$$

with $\boldsymbol{Z}$ the incident matrix of the genomic effects. This model cannot be fitted directly in the BGLR and some precalculus is needed first to compute the "covariance" matrix of the predictor $\boldsymbol{Zu}$ in model (8.9): $\mathbf{K}_* = \mathrm{Var}(\boldsymbol{Zu}) = \boldsymbol{ZKZ}^{\mathrm{T}}$. The BGLR code for implementing this model is the following:

```
  Z = model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))
  K_start = Z%*%K%*%t(Z)
  ETA = list( list( model = 'RHKS', K = K_start , df0 = vu, S0 = Su, R2 =
1-R2)) )
  A   = BGLR(y=y, ETA = ETA, nIter = 1e4, burnIn = 1e3, S0 = S, df0 = v, R2 =
R2)
```

To illustrate how to implement the Bayesian kernel BLUP model in BGLR, some examples are provided next.

Example 2 We again consider the prediction of grain yield (tons/ha) based on marker information. The data set used consists of 30 lines in four environments with one and two repetitions, and the genotype information consists of 500 markers for each line. The numbers of lines with one (two) repetition are 6 (24), 2 (28), 0 (30), and 3 (27) in Environments 1, 2, 3, and 4, respectively, resulting in 229 observations. The performance prediction of all these models was evaluated with 10 random partitions using a cross-validation strategy, where 80% of the complete data set was used to fit the model and the rest to evaluate the model in terms of the mean squared error (MSE) of prediction. Nine kernels were evaluated (linear=GBLUP, polynomial, sigmoid, Gaussian, exponential, AK1, AK2, AK3, and AK4). The R code for implementing this model is given in Appendix 6.

The results for all kernels (shown in Table 8.6) were obtained by iterating 10,000 times the corresponding Gibbs sampler and discarding the first 1000 of them, using the default hyperparameter values implemented in BGLR. We can observe that the worst and second worst prediction performances were obtained under the sigmoid and linear (GBLUP) kernels, while the best and second-best predictions were obtained with polynomial and Gaussian kernels. However, it is important to point out that the differences between the best and worst predictions were small.

8.8.1 Extended Predictor Under the Bayesian Kernel BLUP

The Bayesian kernel BLUP method can be extended, in terms of the predictor, to easily take into account the effects of other factors. For example, in addition to the genotype effect, the effects of environments and genotype × environment interaction terms can also be incorporated as

$$\boldsymbol{y} = \mu\mathbf{1} + \boldsymbol{Z}_E\boldsymbol{\beta}_E + \boldsymbol{u}_1 + \boldsymbol{u}_2 + \boldsymbol{\varepsilon}, \tag{8.10}$$

where $\boldsymbol{y} = [\boldsymbol{y}_1, \ldots, \boldsymbol{y}_I]'$ are the observations collected in each of the I sites (or environments). The fixed effects of the environment are modeled with the incidence matrix of environments $\boldsymbol{Z}_E$, where the parameters to be estimated are the intercepts for each environment ($\boldsymbol{\beta}_E$) (other fixed effects can be incorporated into the model). In this model, $\boldsymbol{u}_1 \sim N\left(\mathbf{0}, \sigma^2_{u_1}\boldsymbol{K}_1\right)$ represents the genomic main effects, $\sigma^2_{u_1}$ is

Table 8.6 Mean squared error (MSE) of prediction across 10 random partitions, with 80% for training and the rest for testing, under nine kernel methods

Partition	Linear	Polynomial	Sigmoid	Gaussian	Exponential	AK1	AK2	AK3	AK4
1	0.578	0.569	0.608	0.566	0.568	0.571	0.570	0.565	0.566
2	0.355	0.360	0.371	0.353	0.353	0.357	0.356	0.353	0.354
3	0.443	0.432	0.472	0.428	0.429	0.435	0.433	0.431	0.431
4	0.387	0.392	0.399	0.372	0.374	0.377	0.380	0.377	0.375
5	0.372	0.329	0.416	0.334	0.335	0.351	0.345	0.343	0.342
6	0.810	0.753	0.864	0.789	0.800	0.797	0.794	0.797	0.792
7	0.757	0.740	0.779	0.752	0.756	0.749	0.752	0.751	0.748
8	0.352	0.340	0.362	0.350	0.352	0.352	0.348	0.348	0.348
9	0.297	0.302	0.306	0.294	0.293	0.295	0.295	0.292	0.292
10	0.551	0.565	0.552	0.549	0.550	0.554	0.550	0.550	0.548
Average	0.490	0.478	0.513	0.479	0.481	0.484	0.482	0.481	0.480

the genomic variance component estimated from the data, and $\boldsymbol{K_1} = \boldsymbol{Z}_{u1}\boldsymbol{K}\boldsymbol{Z}'_{u1}$, where $\boldsymbol{Z}_{u1}$ relates the genotypes to the phenotypic observations. The random effect $\boldsymbol{u_2}$ represents the interaction between the genomic effects and environments and is modeled as $\boldsymbol{u_2} \sim N\left(\boldsymbol{0}, \sigma^2_{u_2}\boldsymbol{K_2}\right)$, where $\boldsymbol{K_2} = \left(\boldsymbol{Z}_{u1}\boldsymbol{K}\boldsymbol{Z}'_{u1}\right)^{\circ}\left(\boldsymbol{Z}_E\boldsymbol{Z}_E{'}\right)$, where $^{\circ}$ is the Hadamard product. The BGLR specification for this Bayesian kernel BLUP model with the extended predictor is exactly the same as the GBLUP method studied in Chap. 6, but instead of using the genomic relationship matrix (linear kernel), now any of the kernels mentioned above is specified:

```
  XE = model.matrix(~0+Env,data=dat_F)[,-1]
  K.E=XE%*%t(XE)
  Z_L = model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))
  K_L=Z_L%*%K%*%t(Z_L) ###### K is the kernel matrix
  K_LE= K.E*K_L
  ETA_K=list(list(model='FIXED',X=XE),list(model='RKHS',K=K_L),
list(model='RKHS',K=K_LE))
   y_NA = y
   y_NA[Pos_tst] = NA
  A = BGLR(y=y_NA,ETA=ETA_K,nIter = 1e4,burnIn = 1e3,verbose = FALSE,
nIter = 1e4, burnIn = 1e3, S0 = S, df0 = v, R2 = R^2)
```

Now to illustrate the Bayesian kernel BLUP with the extended predictor described in Eq. (8.10), we used a data set that contains 30 lines in four environments, and the genotyped information is composed of 500 markers for each line. We call this data set dat_ls_E2. Now only the following kernels were implemented: linear, polynomial, sigmoid, Gaussian, exponential, AK1, and AK4. The R code for reproducing the results in Table 8.7 is given in Appendix 7. We can observe that taking into account the genotype by environment interaction in the predictor, the best

Table 8.7 Mean squared error (MSE) of prediction across 10 random partitions, with 80% for training and the rest for testing, under seven kernel methods with the predictor including the effects of environment + genotypes + genotype × environment interaction term. Here we used the dat_ls_E2 data set

Partition	Linear	Polynomial	Sigmoid	Gaussian	Exponential	AK1	AK4
1	0.729	0.662	0.775	0.665	0.930	0.700	0.742
2	0.533	0.559	0.596	0.499	0.573	0.515	0.496
3	0.691	0.629	0.724	0.633	0.634	0.654	0.620
4	0.646	0.621	0.678	0.631	0.748	0.626	0.633
5	0.517	0.550	0.488	0.517	0.490	0.519	0.516
6	0.674	0.650	0.683	0.597	0.586	0.640	0.607
7	0.419	0.435	0.474	0.376	0.558	0.403	0.397
8	0.400	0.409	0.406	0.359	0.349	0.381	0.361
9	0.618	0.611	0.641	0.589	0.586	0.605	0.587
10	0.539	0.494	0.567	0.473	0.493	0.507	0.485
Average	0.576	0.562	0.603	0.534	0.595	0.555	0.544

prediction performance was obtained with the Gaussian kernel while the worst was obtained under the sigmoid kernel.

It is important to point out that in the predictor under a Bayesian kernel BLUP using BGLR, as many terms as desired can be included, and the specification is very similar to how it was done with three terms in the predictor in this example. Using BGLR, the Bayesian kernel BLUP can be implemented for binary and ordinal response variables. Next, we provide one example for binary response variables and one for categorical response variables.

8.8.2 *Extended Predictor Under the Bayesian Kernel BLUP with a Binary Response Variable*

It is important to point out that it is feasible to implement the Bayesian kernel BLUP with binary response variables using the probit link function in BGLR. This implementation first requires calculating the kernel to be used; then with the following lines of code, the Bayesian kernel BLUP can be fitted for binary and categorical response variables:

```
XE = model.matrix(~0+Env,data=dat_F)[,-1]
K.E=XE%*%t(XE)
Z_L = model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))
K_L=Z_L%*%K%*%t(Z_L)
K_LE= K.E*K_L
ETA_K=list(list(model='FIXED',X=XE),list(model='RKHS',K=K_L),
        list(model='RKHS',K=K_LE))
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K, response_type="ordinal",nIter = 1e4,
burnIn = 1e3,verbose = FALSE)
 Probs = A$probs[Pos_tst,]
```

When categorical response variables are used, two different things need to be modified to fit the model in BGLR. The first one is that we need to specify response_type="ordinal" and the other is that the outputs now are the probabilities that can be extracted with A$probs. When response_type="ordinal" is ignored, the response variable is assumed Gaussian by default.

To give an example with a binary response variable, we used the EYT Toy data set (Data_Toy_EYT. RData) that is preloaded in the BMTME library. This data set is composed of 40 lines, four environments (Bed5IR, EHT, Flat5IR, and LHT), and four response variables: DTHD, DTMT, GY, and Height. G_Toy_EYT is the genomic relationship matrix of dimension 40×40. The first two variables are ordinal with three categories, the third is continuous (GY = Grain yield) and the last one (Height) is binary. In this example, we work with only the binary response variable (Height).

Table 8.8 Proportion of cases correctly classified (PCCC) across 10 random partitions, with 80% for training and the rest for testing, under seven kernel methods with the predictor including the effects of environment + genotypes + genotype × environment interaction term with the Data_Toy_EYT with trait Height

Partition	Linear	Polynomial	Sigmoid	Gaussian	Exponential	AK1	AK4
1	0.781	0.781	0.750	0.719	0.625	0.813	0.844
2	0.594	0.625	0.594	0.719	0.750	0.813	0.844
3	0.688	0.625	0.688	0.656	0.719	0.688	0.688
4	0.688	0.656	0.656	0.563	0.656	0.781	0.781
5	0.406	0.500	0.406	0.563	0.531	0.531	0.563
6	0.656	0.656	0.656	0.656	0.688	0.688	0.719
7	0.625	0.625	0.625	0.656	0.688	0.688	0.719
8	0.719	0.688	0.688	0.719	0.719	0.719	0.719
9	0.500	0.563	0.500	0.688	0.719	0.750	0.719
10	0.531	0.563	0.531	0.688	0.594	0.688	0.688
Average	0.619	0.628	0.609	0.663	0.669	0.716	0.728

Table 8.8 gives the results of implementing the Bayesian kernel BLUP method under a binary response variable with seven kernels using the ETY Toy data set with trait Height. The best predictions using the EYT Toy data set were obtained with kernel AK4, and the worst was under kernel sigmoid. Again, we can see that, in general, most kernel methods outperform the linear kernel. The R code for reproducing the results in Table 8.8 is given in Appendix 8.

8.8.3 *Extended Predictor Under the Bayesian Kernel BLUP with a Categorical Response Variable*

The fitting process in BGLR for the categorical response variable is exactly the same as the binary response variable explained above. For this reason, the results given in Table 8.9 for the categorial response variable were obtained with the same R code given in Appendix 8 with the following two modifications: (a) y=dat_F$DTMT instead of y=dat_F$Height and b) yp_ts=apply(Probs,1,which.max) instead of yp_ts=apply(Probs,1,which.max)-1, since now the response variable has levels 1, 2, and 3.

Table 8.9 shows that the best prediction performance for the categorical response variable was observed in the polynomial kernel while the worst was under the AK4 kernel.

Table 8.9 Proportion of cases correctly classified (PCCC) across 10 random partitions, with 80% for training and the rest for testing, under seven kernel methods with the predictor including effects of environment + genotypes + genotype × environment interaction term with the Data_Toy_EYT with the ordinal trait DTMT

Fold	Linear	Polynomial	Sigmoid	Gaussian	Exponential	AK1	AK4
1	0.656	0.656	0.656	0.688	0.688	0.688	0.625
2	0.688	0.688	0.688	0.656	0.688	0.688	0.688
3	0.813	0.813	0.813	0.813	0.813	0.781	0.781
4	0.719	0.719	0.719	0.656	0.656	0.688	0.656
5	0.563	0.625	0.563	0.656	0.656	0.625	0.656
6	0.719	0.750	0.750	0.781	0.750	0.750	0.750
7	0.625	0.656	0.625	0.625	0.594	0.625	0.594
8	0.594	0.594	0.594	0.594	0.594	0.594	0.594
9	0.688	0.688	0.688	0.656	0.656	0.625	0.625
10	0.750	0.719	0.781	0.750	0.719	0.750	0.719
Average	0.681	0.691	0.688	0.688	0.681	0.681	0.669

8.9 Multi-trait Bayesian Kernel

In BGLR, it is possible to fit multi-trait Bayesian kernel BLUP methods, and the fitting process is exactly the same as fitting multi-trait Bayesian GBLUP methods (see Chap. 6). The only difference is that instead of using a linear kernel, any kernel can be used. The basic R code for fitting multi-trait Bayesian kernel methods is given next:

```
  y_NA = data.matrix(y)
  y_NA[Pos_tst,] = NA
  A4= Multitrait(y = y_NA, ETA=ETA_K.Gauss,resCov = list(type ="UN",
S0=diag(4),df0= 5), nIter =10000, burnIn = 1000)
  Me_Gauss= PC_MM_f(y[Pos_tst,],A4$ETAHat[Pos_tst,],Env=dat_F$Env
[Pos_tst])
  A4$ETAHat[Pos_tst,]
```

To illustrate the fitting process of a multi-trait Bayesian kernel BLUP method, we used the EYT Toy data set (Data_Toy_EYT. RData), but using the four response variables simultaneously (even though only GY is Gaussian, we assumed that the four response variables satisfy this assumption). The R code for its implementation is given in Appendix 9. Table 8.10 gives the results of the prediction performance in terms of the mean square error across the 10 random partitions, where the best kernel for prediction differs between environment-trait combinations. The polynomial kernel and the linear kernel were the best in 4 out of 16 environment-trait combinations.

Table 8.10 Mean square error (MSE) of prediction across 10 random partitions, with 80% for training and the rest for testing, under seven kernel methods with the predictor including the effects of environment + genotypes + genotype × environment interaction term with the Data_Toy_EYT assuming that the multi-trait response is Gaussian

Env	Trait	Linear	Polynomial	Sigmoid	Gaussian	Exponential	AK1	AK4
EHT	DTHD	0.302	0.292	0.303	**0.274**	0.268	0.280	0.280
EHT	DTMT	0.164	0.180	0.162	0.154	**0.152**	0.158	0.158
EHT	GY	0.333	0.396	**0.321**	0.342	0.348	0.354	0.354
EHT	Height	0.167	0.149	0.162	0.156	0.150	**0.148**	**0.148**
LHT	DTHD	0.261	0.243	0.259	0.229	**0.220**	0.234	0.234
LHT	DTMT	0.357	0.341	0.360	0.322	**0.310**	0.326	0.326
LHT	GY	0.328	**0.302**	0.322	0.319	0.333	0.318	0.318
LHT	Height	**0.228**	0.230	0.233	0.232	0.230	0.232	0.232
Bed5IR	DTHD	0.150	0.160	0.151	**0.148**	0.153	0.155	0.155
Bed5IR	DTMT	**0.202**	0.228	0.199	0.203	0.210	0.214	0.214
Bed5IR	GY	0.138	0.138	0.136	0.136	0.135	**0.134**	**0.134**
Bed5IR	Height	0.189	**0.178**	0.189	0.185	0.184	0.182	0.182
Flat5IR	DTHD	**0.186**	0.190	0.189	0.194	0.200	0.191	0.191
Flat5IR	DTMT	**0.211**	0.224	**0.211**	0.216	0.226	0.219	0.219
Flat5IR	GY	0.477	**0.465**	0.478	0.478	0.469	0.471	0.471
Flat5IR	Height	0.179	**0.178**	0.180	0.183	0.182	0.183	0.183

8.10 Kernel Compression Methods

By kernel compression methods, we mean those tools that allow us to approximate kernels without affecting the prediction accuracy very much, but gaining a significant reduction in computational resources. There are many methods for compression of kernel methods. However, in this section, we only illustrate the method proposed by Cuevas et al. (2020). The basic idea of this method consists of approximating the original kernel using a small size (m) of the original n observations (lines in the context of GS) available in the training set, which significantly reduces the required computational resources required to build the kernel matrix.

Before giving the details of the compression of kernels proposed by Cuevas et al. (2020), it is important to point out that model (8.8) can be reparametrized as Eq. (8.11) if the eigenvalue decomposition of the kernel matrix $\boldsymbol{K}$ is expressed as $\boldsymbol{US}^{1/2}\boldsymbol{S}^{1/2}\boldsymbol{U}'$,

$$\boldsymbol{y} = \mu 1_n + \boldsymbol{Pf} + \boldsymbol{\varepsilon}, \tag{8.11}$$

where $\boldsymbol{f} \sim N\left(\boldsymbol{0}, \sigma_f^2 \boldsymbol{I}_{r,r}\right)$ (where r is the rank of $\boldsymbol{K}$) and $\boldsymbol{P} = \boldsymbol{US}^{1/2}$. Note that models (8.8) and (8.11) are equivalent. Model (8.11) can be fitted by the conventional Ridge regression model. This Ridge regression reparameterization most of the time is computationally very efficient, since most of the time $r < \min\ (n, p)$, which is common in multi-environment and/or multi-trait models. It should be noted that only r effects can be summarized and projected for $\boldsymbol{P}$ to explain the n effects without any significant loss of precision with the available information.

Next, we describe the Cuevas et al. (2020) method for compressing the kernel matrix $\boldsymbol{K}$. First, the method approximates the original kernel matrix ($\boldsymbol{K}$) using a smaller sub-matrix $\boldsymbol{K}_{m,m}$ ($m < n$) constructed with m out of n lines. The rank of $\boldsymbol{K}_{m,m}$ is m. Under the assumption that the row vectors are linearly independent, Williams and Seeger (2001) showed that the Nyström approximation of the kernel is as follows:

$$\boldsymbol{K} \approx \boldsymbol{Q} = \boldsymbol{K}_{n,m}\boldsymbol{K}_{m,m}^{-1}\boldsymbol{K}'_{n,m},$$

where $\boldsymbol{Q}$ will have the rank of $\boldsymbol{K}_{m,m}$, that is, m. The computation of this kernel is facilitated since it is not necessary to compute and store the original matrix $\boldsymbol{K}$, since only $\boldsymbol{K}_{m,m}$ and $\boldsymbol{K}_{n,m}$ are required.

Therefore, $\boldsymbol{K}_{m,m}$ can be computed with m lines with all the p markers, that is, $\boldsymbol{X}_{m,p}$. For the **linear kernel** (GBLUP), $\boldsymbol{K}_{m,m} = \frac{\boldsymbol{X}_{m,p}\boldsymbol{X}'_{m,p}}{p}$ and $\boldsymbol{K}_{n,m} = \frac{\boldsymbol{X}_{n,p}\boldsymbol{X}'_{m,p}}{p}$ which captures the relationship of all n lines with the m lines. Note that in the construction of $\boldsymbol{Q}$, all the n lines and all the p markers are considered, but not all their relationships are accounted for. For example, relationships $\boldsymbol{K}_{n-m,n-m} = \frac{\boldsymbol{X}_{n-m,p}\boldsymbol{X}'_{n-m,p}}{p}$ are not considered (where $n - m$ represents the complement to the m lines). To try to explain this, we ordered the elements of matrix $\boldsymbol{Q}$ per block, such that $\boldsymbol{Q}_{n,n} = \begin{bmatrix} \boldsymbol{Q}_{m,m} & \boldsymbol{Q}_{m,n-m} \\ \boldsymbol{Q}_{n-m,m} & \boldsymbol{Q}_{n-m,n-m} \end{bmatrix}$.

Rassmussen and Williams (2006) showed that $\boldsymbol{Q}_{m,m} = \boldsymbol{K}_{m,m}$, $\boldsymbol{Q}_{n-m,m} = \boldsymbol{K}_{n-m,m}$, $\boldsymbol{Q}_{m,n-m} = \boldsymbol{K}_{m,n-m}$, and that the difference $\boldsymbol{K}_{n-m,n-m} - \boldsymbol{Q}_{n-m,n-m}$, that is, $\boldsymbol{K}_{n-m,n-m} - \boldsymbol{K}_{n-m,m}\boldsymbol{K}_{m,m}^{-1}\boldsymbol{K}_{m,n-m}$ is known as the Schur complement of $\boldsymbol{K}_{m,m}$ on $\boldsymbol{K}_{n,\ n}$. Then, because it is assumed that $\boldsymbol{K}_{m,m}$ and $\boldsymbol{K}_{n,n}$ are positive semi-definite, their difference is also positive semi-definite: $\boldsymbol{Q}_{n,n} = \begin{bmatrix} \boldsymbol{K}_{m,m} & \boldsymbol{K}_{m,n-m} \\ \boldsymbol{K}_{n-m,m} & \boldsymbol{Q}_{n-m,n-m} \end{bmatrix}$. Assuming the effects of $\boldsymbol{u}_{n\ -\ m} \mid \boldsymbol{u}_m$ are conditional and independent, Snelson and Ghahramani (2006) and Misztal et al. (2014) proposed substituting the diagonal of the differences of $\boldsymbol{Q}_{n\ -\ m,\ n\ -\ m}$ with the diagonal of $\boldsymbol{K}_{n\ -\ m,\ n\ -\ m}$.

In the method called projected process, Seeger et al. (2003) theoretically showed that using all lines and considering the minimum Kullback–Leibler distance $\mathrm{KL}(q(\boldsymbol{u}|\boldsymbol{y})\|p(\boldsymbol{u}|\boldsymbol{y}))$ justify that the matrix $\boldsymbol{K}$ in the prior distribution of $\boldsymbol{u}$ (of model 8.8) can be substituted for the $\boldsymbol{Q}$ approximations from Nyström (Titsias 2009). That is, the random genetic vectors have a normal distribution $\boldsymbol{u} \sim N\left(\boldsymbol{0}, \sigma_u^2\boldsymbol{Q}\right)$, where $\boldsymbol{Q} = \boldsymbol{K}_{n,m}\boldsymbol{K}_{m,m}^{-1}\boldsymbol{K}'_{n,m}$.

With these adjustments in the distribution of the random effects $\boldsymbol{u}$, we used model (8.8) for prediction. It is common to estimate parameters σ_e^2 and σ_u^2 of the model with the marginal likelihood and then predict the random effects using the inversion lemma, which is fast. Furthermore, if matrix $\boldsymbol{Q}$ is directly used in BGLR, there is no advantage in terms of saving computational resources using the approximate kernel. Therefore, an eigen decomposition of $\boldsymbol{K}_{m,m}^{-1} = \boldsymbol{U}\boldsymbol{S}^{-1/2}\boldsymbol{S}^{-1/2}\boldsymbol{U}'$ is used where $\boldsymbol{U}$ are the eigenvectors of order $m \times m$ and $\boldsymbol{S}_{m,m}$ is a diagonal matrix of order $m \times m$ with the eigenvalues ordered from largest to smallest. These values are substituted in $\boldsymbol{Q}$ resulting in $\boldsymbol{u_n} \sim N\left(\boldsymbol{0}, \sigma_u^2 \boldsymbol{K}_{n,m}\boldsymbol{U}\boldsymbol{S}^{-1/2}\boldsymbol{S}^{-1/2}\boldsymbol{U}'\boldsymbol{K}_{n,m}'\right)$, and thus, thanks to the properties of the normal distribution, model (8.8) can be expressed like model (8.11) as

$$\boldsymbol{y} = \mu 1_n + \boldsymbol{P}\boldsymbol{f} + \boldsymbol{\varepsilon} \tag{8.12}$$

Model (8.12) is similar to model (8.11), except that $\boldsymbol{f}$ is a vector of order $m \times 1$ with a normal distribution of the form $\boldsymbol{f} \sim \boldsymbol{N}\left(\boldsymbol{0}, \sigma_f^2 \boldsymbol{I}_{m,m}\right)$, where $\boldsymbol{P} = \boldsymbol{K}_{m,\,n}\boldsymbol{U}\boldsymbol{S}^{-1/2}$ is now the design matrix. This implies estimating only m effects that are projected into the n-dimensional space in order to predict $\boldsymbol{u_n}$ and explain $\boldsymbol{y}_n$. Note that model (8.12) has a Ridge regression solution, and thus available software for Bayesian Ridge regression like BGLR R or software for conventional Ridge regression like glmnet can be used for fitting model (8.12).

In summary, according to Cuevas et al. (2020), the approximation described above consists of the following steps:

Step 1. Computing the following matrices, matrix $\boldsymbol{K}_{m,m}$ from m lines of the training set.
Step 2. Computing matrix $\boldsymbol{K}_{n,m}$.
Step 3. Eigenvalue decomposition of $\boldsymbol{K}_{m,m}$.
Step 4. Computing matrix $\boldsymbol{P} = \boldsymbol{K}_{n,m}\boldsymbol{U}\boldsymbol{S}^{-1/2}$.
Step 5. Fitting the model under a Ridge regression framework (like BGLR or glmnet) and making genomic-enabled predictions for future data.

With the following R code, the $\boldsymbol{P} = \boldsymbol{K}_{n,m}\boldsymbol{U}\boldsymbol{S}^{-1/2}$ (matrix design) can be computed under a linear kernel.

```
##################Linear approximate kernel########################
Sparse_linear_kernel = function(m, X){
m = m
XF = X
p = ncol(XF)
pos_m = sample(1:nrow(XF),m)
####Step 1 compute K_m###############3
X_m = XF[pos_m,]
dim(X_m)
K_m = X_m% * %t(X_m)/p
dim(K_m)
######Step 2 compute K_n_m###########
K_n_m = XF% * %t(X_m)/p
dim(K_n_m)
######Step 3 compute eigenvalue decomposition of K_m######
EVD_K_m = eigen(K_m)
####Eigenvectors
U = EVD_K_m$vectors
###Eigenvalues###
S = EVD_K_m$values
####Square root of the inverse of eigenvalues#####
S_0.5_Inv = sqrt(1/S)
#####Diagonal matrix of square root of inverse of eigenvalues###
S_mat_Inv = diag(S_0.5_Inv)
#####Computing matrix P
P = K_n_m% * %U% * %S_mat_Inv
return(P)}
```

To use this function to create the design matrix $\boldsymbol{P} = \boldsymbol{K}_{n,m}\boldsymbol{U}\boldsymbol{S}^{-1/2}$, you need to provide the standardized matrix of markers $\boldsymbol{X}$, and the number of lines m, to be used for computing the approximate linear kernel. Then with this $\boldsymbol{P}$ you can implement the Ridge regression model under a Bayesian or conventional framework.

Table 8.11 indicates that the lower the value of m for building the approximate kernel, the smaller the time required for its implementation in the four response variables (Env1, . . ., Env4). The table also shows that the lower the value of m, the worse the prediction performance in terms of MSE and PC. However, it is really interesting that with a reduction in the training set from 599 (all data) to 264 (only 44% of the total data set), the implementation time is reduced to almost half without any significant loss in terms of prediction performance. The complete R code to reproduce the results provided in Table 8.11 is given in Appendix 10.

The approximate kernel method can be used for any of the kernels studied before. The construction of the approximate Gaussian kernel matrix can be done with the following R function:

```
####################Gaussian kernel function#####################
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j])^2)))}
K.rad=K.radial(x1=XF,x2=XF, gamma=1/ncol(XF))

######################Approximate Guassian kernel################
Sparse_Gaussian_kernel=function(m,X){
m=m
XF=X
p=ncol(XF)
pos_m=sample(1:nrow(XF),m)
####Step 1 compute K_m###############3
X_m=XF[pos_m,]
dim(X_m)
K_m=K.radial(x1=X_m,x2=X_m, gamma=1/p)

######Step 2 compute K_n_m###########
K_n_m=K.radial(x1=XF,x2=X_m, gamma=1/p)

############Step 3 compute eigenvalues decomposition of
K_m###########
```

Table 8.11 Prediction performance in terms of mean square error (MSE) and average Pearson's correlation (PC) under the approximate linear kernel with the method proposed by Cuevas et al. (2020)

		Env1			Env2			Env3			Env4	
m	MSE	PC	Time	MSE	PC	Time	MSE	PC	Time	MSE	PC	Time
15	0.914	0.293	12.410	0.904	0.312	12.430	0.941	0.248	12.830	0.905	0.310	12.860
32	0.852	0.372	15.860	0.865	0.369	14.480	0.910	0.303	15.220	0.877	0.351	14.760
74	0.804	0.435	18.960	0.810	0.431	20.790	0.873	0.361	20.000	0.824	0.423	20.720
132	0.740	0.494	33.050	0.785	0.458	27.150	0.869	0.369	26.550	0.798	0.453	26.740
264	0.716	0.523	53.250	0.749	0.497	43.720	0.852	0.393	42.090	0.794	0.457	42.630
599	0.713	0.524	98.800	0.741	0.506	86.470	0.844	0.405	86.290	0.785	0.468	87.220

Ten-fold cross-validation was implemented and the prediction performance is only reported for the testing set. Six values of training size *m* were implemented: 15, 32, 74, 132, 264, and 599 (all data). Time reported: the implementation time in seconds. Data wheat599 was used

```
EVD_K_m=eigen(K_m)
####Eigenvectors
U=EVD_K_m$vectors
###Eigenvalues###
S=EVD_K_m$values
####Square root of the inverse of eigenvalues #####
S_0.5_Inv=sqrt(1/S)
#####Diagonal matrix of square root of inverse of eigenvalues###
S_mat_Inv=diag(S_0.5_Inv)
######Computing matrix P
P=K_n_m%*%U%*%S_mat_Inv
return(P)}
```

With this approximate kernel method, an equivalent to Table 8.11 was reproduced, but instead of using a linear kernel, a Gaussian kernel was used. The results for the same values of m as those used in Table 8.11 (m = 15, 32, 74, 132, 264, and 599), with the wheat599 data set for each of the four response variables are given in Table 8.12.

Again, we can see in Table 8.12 that the lower the training set (m) used for approximating the kernel, the lower the prediction performance, but the shorter the implementation time. However, it is very interesting to point out that with this data set, the approximation obtained when 44% (264 lines) of the original lines (599 lines) were used is quite good for the four response variables (E1,. . ., E4). However, the approximation to the full data set was slightly better under the approximate linear kernel (Table 8.11) than under the approximate Gaussian kernel (Table 8.12), but in general, the predictions obtained with the Gaussian kernel (full and approximated) were better than those obtained with the linear kernel (full and approximated). It is really important to point out that the R code given in Appendix 10 can be used for reproducing the results given in Table 8.12, but by replacing the function of the approximate linear kernel with the function for the approximate Gaussian kernel.

8.10.1 Extended Predictor Under the Approximate Kernel Method

Now we will illustrate the approximate kernel using the expanded predictor (8.10) that, in addition to the main effect of lines, contains the main effects of environments and the interaction term between environments and lines. Therefore, the approximate method is similar to the case of a single environment, that is, $\boldsymbol{u_1} \sim N\left(\mathbf{0}, \sigma_{u_1}^2 \boldsymbol{Q}^{u_1}\right)$, where $\boldsymbol{K_1} \approx \boldsymbol{Q}^g = \boldsymbol{Z}_{u1} \left(\boldsymbol{K}_{n,m}\boldsymbol{K}_{m,m}^{-\mathbf{1}}\boldsymbol{K}_{n,m}^T\right)\boldsymbol{Z}_{u1}^T$, whereas for the random interaction $\boldsymbol{u_2} \sim N\left(\mathbf{0}, \sigma_{u_2}^2 \boldsymbol{Q}^{u_2}\right)$, where $\boldsymbol{K_2} \approx \boldsymbol{Q}^{u_2} = \left[\boldsymbol{Z}_{u1} \left(\boldsymbol{K}_{n,m}\boldsymbol{K}_{m,m}^{-\mathbf{1}}\boldsymbol{K}_{n,m}^{\boldsymbol{T}}\right)\boldsymbol{Z}_{u1}^T\right]^{\circ} \left[\boldsymbol{Z}_E\boldsymbol{Z}_E{}^T\right]$.

Also, we decomposed $\boldsymbol{K}_{m,m}^{-\mathbf{1}}$ in such a way that model (8.10) could be approximated as

Table 8.12 Prediction performance in terms of mean square error (MSE) and average Pearson's correlation (PC) under the approximate Gaussian kernel with the method proposed by Cuevas et al. (2020) using the wheat599 data set

	Env1			Env2			Env3			Env4		
m	MSE	PC	Time	MSE	PC	Time	MSE	PC	Time	MSE	PC	Time
15	0.927	0.301	12.780	0.955	0.241	12.710	0.988	0.227	11.120	0.959	0.213	12.700
32	0.845	0.389	14.270	0.911	0.313	16.230	0.937	0.254	12.680	0.931	0.292	14.030
74	0.802	0.448	20.970	0.827	0.412	21.310	0.868	0.365	17.050	0.800	0.453	20.380
132	0.714	0.533	30.200	0.816	0.435	29.680	0.865	0.373	24.770	0.775	0.476	25.250
264	0.677	0.570	51.140	0.769	0.483	48.520	0.854	0.387	39.280	0.745	0.512	38.530
599	0.645	0.595	104.250	0.754	0.501	84.890	0.803	0.454	80.720	0.721	0.538	81.990

Ten-fold cross-validation was implemented and the prediction performance is only reported for the testing set. Six values of training size *m* were implemented: 15, 32, 74, 132, 264, and 599 (all data). Time reported: the implementation time in seconds

$$\boldsymbol{y} = \mu\mathbf{1} + \boldsymbol{Z}_E\boldsymbol{\beta}_E + \boldsymbol{P}^{u_1}\boldsymbol{f} + \boldsymbol{P}^{u_2}\boldsymbol{l} + \boldsymbol{\varepsilon}, \tag{8.13}$$

where $\boldsymbol{P}^{u_1} = \boldsymbol{Z}_{u1}\boldsymbol{P} = \boldsymbol{Z}_{u1}\boldsymbol{K}_{n,m}\boldsymbol{US}^{-1/2}$ of order $n^* \times m$, with $n^* = n_1 + n_2 + \ldots + n_I$, with $\boldsymbol{f}$ a vector of $m \times 1$; $\boldsymbol{P}^{u_2} = \boldsymbol{P}^{u_1} : \boldsymbol{Z}_E$ of order $n^* \times mI$ and the vector $\boldsymbol{l}$ is of order $mI \times 1$, and the notation $\boldsymbol{P}^{u_1} : \boldsymbol{Z}_E$ denotes the interaction term between the design matrix $\boldsymbol{P}^{u_1}$ and $\boldsymbol{Z}_E$.

In summary, the suggested approximate method described above can be summarized as

Step 1. We assume that we have a matrix of markers $\mathbf{X}$ that contains the lines without replication, that is, each row corresponds to a different line. We assume that this matrix contains L lines (rows) and p markers (columns). Also, it is important to point out that this matrix is standardized by columns.
Step 2: We randomly select m lines out of L from the training set $\mathbf{X}$.
Step 3. Next we construct matrices $\boldsymbol{K}_{m,m}$ and $\boldsymbol{K}_{L,m}$, from the matrix of markers as $\boldsymbol{K}_{m,m} = \frac{X_{m,p}X'_{m,p}}{p}$, $\boldsymbol{K}_{L,m} = \frac{X_{L,p}X'_{m,p}}{p}$
Step 4. Eigenvalue decomposition of $\boldsymbol{K}_{m,m}$.
Step 5. Computing matrix $\boldsymbol{P} = \boldsymbol{K}_{n,m}\boldsymbol{US}^{-1/2}$.
Step 6. Computing matrix $\boldsymbol{P}^{u_1} = \boldsymbol{Z}_{u1}\ \boldsymbol{P} = \boldsymbol{Z}_{u1}\boldsymbol{K}_{n,m}\boldsymbol{US}^{-1/2}$.
Step 7. Computing matrix $\boldsymbol{P}^{u_2} = \boldsymbol{P}^{u_1} : \boldsymbol{Z}_E$, where : denotes the interaction between the design matrix $\boldsymbol{P}^{u_1}$ and $\boldsymbol{Z}_E$.
Step 8. Fitting the model under a Ridge regression framework (like BGLR or glmnet) and making genomic-enabled predictions for future data.

It is important to point out that the extension of the approximate kernel method for an extended predictor requires that some lines were studied in some environments but not in all environments. This extended approximate kernel method is expected to be efficient when the number of environments is low and the number of lines is large.

To illustrate the extended approximate kernel method, we used the Data_Toy_EYT that contains four environments, four traits, and 40 observations in each environment. Here we only used the continuous trait (GY) as the response variable. The approximate kernels were built using only the lines (40 lines) from which the training set was built with $m = 4$, 8, 12, 16, 20, and 40 lines. Now instead of using only one kernel, we implemented five (linear, polynomial, sigmoid, Gaussian, and exponential). The R code for reproducing the results in Table 8.13 is given in Appendix 11.

We can see in Table 8.13 that there are differences in the prediction performance using different kernels. However, the approximate kernel, even with $m = 4$, many times outperformed the prediction performance of the exact kernel ($m = 40$), which implies that when the lines are quite correlated even with a small sample size m, approximating the kernel is enough to get good prediction performance. But the time gained using a sample size m, that is less than n total number of lines, significantly reduces the implementation time, and this gain in implementation time is more relevant when the number of lines is very large, as was stated by Cuevas et al. (2020).

Table 8.13 Prediction performance in terms of mean square error (MSE) and average Pearson's correlation (PC) under five approximate Gaussian kernel methods with the method proposed by Cuevas et al. (2020)

Metrics	Kernel	$m = 4$	$m = 8$	$m = 12$	$m = 16$	$m = 20$	$m = 40$
MSE	Linear	0.306	0.325	0.367	0.329	0.327	0.325
PC	Linear	0.939	0.936	0.926	0.935	0.935	0.936
Time	Linear	12.300	17.780	17.090	17.340	19.330	27.240
MSE	Polynomial	0.330	0.343	0.356	0.308	0.325	0.342
PC	Polynomial	0.934	0.931	0.927	0.939	0.935	0.933
Time	Polynomial	17.000	17.030	18.280	20.470	21.180	25.750
MSE	Sigmoid	0.383	0.295	0.335	0.314	0.328	0.324
PC	Sigmoid	0.923	0.941	0.934	0.937	0.935	0.936
Time	Sigmoid	19.400	18.550	17.170	19.770	20.450	23.550
MSE	Gaussian	0.325	0.337	0.351	0.335	0.331	0.340
PC	Gaussian	0.936	0.932	0.929	0.934	0.935	0.933
Time	Gaussian	18.530	19.300	20.090	18.730	22.360	30.630
MSE	Exponential	0.333	0.320	0.339	0.381	0.363	0.348
PC	Exponential	0.932	0.936	0.932	0.924	0.927	0.932
Time	Exponential	17.220	18.860	20.090	21.730	19.200	24.970

Ten fold cross-validation was implemented and the prediction performance is only reported for the testing set. Six values of training size m were implemented: 4, 8, 12, 16, 20, and 40 (all data). Time reported: the implementation time in seconds. The data set used for this example was Data_Toy_EYT with trait GY as the response variable that was used before in Table 8.10

8.11 Final Comments

In the context of genomic prediction, arguably, genotypes and phenotypes may be linked in functional forms that are not well addressed by the linear additive models that are standard in quantitative genetics. Therefore, developing statistical machine learning models for predicting phenotypic values from all available molecular information and that are capable of capturing complex genetic network architectures is of great importance. Kernel Ridge regression methods are nonparametric prediction models proposed for this purpose. Their essence is to create a spatial distance-based relationship matrix called a kernel (Morota et al. 2013). For this reason, many kernel functions have been developed to capture complex nonlinear patterns that many times outperform conventional linear regression models in terms of prediction performance.

The kernel trick allows you to build nonlinear versions of any linear algorithms by replacing their independent variables (predictors) with a kernel function, giving them greater advantages.

1. The kernels are interpreted as scalar products in high-dimensional spaces.
2. There are kernels with great versatility and composite kernels can be built; some can be computed in closed form, while others require an iterative process.

3. The number of dimensions increases the complexity, and with it the risk of overfitting.
4. Kernel methods are an alternative to least square methods algorithms that control complexity through regularization.
5. Kernel methods guarantee existence and uniqueness, just like least square methods.
6. Nonlinear versions can be made using the kernel trick, obtaining statistical machines with great expressive capacity, but with training control.
7. Kernel statistical machine learning methods provide promising tools for large-scale and high-dimensional genomic data processing.
8. Kernel methods can also be viewed from a regression perspective and can be integrated with classical methods for gene prioritizing, prediction, and data fusion.
9. Kernel methods allow you to further improve the scalability of conventional machine learning methods and their versatility to work with heterogeneous inputs.
10. Kernel methods are remarkably flexible and elegant, as they are the predictive principle underlying most linear mixed models commonly used in plant breeding, as well as others used in spatial analysis of classification problems.
11. Kernel methods exploit complexity to improve prediction accuracy, but do not help very much to increase the understanding of the complexity.
12. Kernels based on data compression ideas are very promising for dealing with very large data sets, but software is needed, as well as more research to develop new methods and improve the existing methods.

Appendix 1

R code for manually implementing nine kernels (linear, polynomial, sigmoid, Gaussian, exponential, AK1, AK2, AK3, and AK4) with a continuous response variable; the results are provided in Table 8.1.

```
rm(list=ls(all=TRUE))
library(plyr)
library(tidyr)
library(dplyr)
library(glmnet)
library(BGLR)
data(wheat) #Loads the wheat data set
y=wheat.Y
dim(y)
head(y)

XF=scale(wheat.X)
dim(XL)
head(XF[,1:5])
```

```
########Linear Kernel############
K.linear=function(x1, x2=x1,gamma=1) {gamma*(as.matrix(x1)%*%t(as.
matrix(x2))) }
K.lin=K.linear(x1=XF,x2=XF,gamma=1/ncol(XF))
dim(K.lin)
X.lin=t(chol(K.lin))

########Polynomial Kernel############
K.polynomial=function(x1, x2=x1, gamma=1, b=0, p=3){
 (gamma*(as.matrix(x1)%*%t(x2))+b)^p}

K.poly=K.polynomial(x1=XF,x2=XF,gamma=1/ncol(XF))
dim(K.poly)
X.poly=t(chol(K.poly))

########Sigmoid Kernel############
K.sigmoid=function(x1,x2=x1, gamma=1/ncol(XF), b=0)
{ tanh(gamma*(as.matrix(x1)%*%t(x2))+b) }
K.sig=K.sigmoid(x1=XF,x2=XF)
dim(K.sig)
ei=eigen(K.sig)
pos_neg=which(ei$values<0)
Eigenv=ei$value
Eigenv[pos_neg]=0
Eigenv
X.sig=ei$vectors%*%diag(sqrt(Eigenv))%*%t(ei$vectors)

########Radial Kernel############
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j])^2)))}
K.rad=K.radial(x1=XF,x2=XF, gamma=1/ncol(XF))
dim(K.rad)
X.rad=t(chol(K.rad))

########Exponential Kernel############
K.exponential=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]))))}

K.exp=K.exponential(x1=XF,x2=XF, gamma=1/ncol(XF))
dim(K.exp)
X.exp=t(chol(K.exp))

############Arc-cosine kernel with L=1#######
K.AC1<-function(X){
 n<-nrow(X)
 cosalfa<-cor(t(X))
 angulo<-acos(cosalfa)
 mag<-sqrt(apply(X,1,function(x) crossprod(x)))
 sxy<-tcrossprod(mag)
 GC1<-(1/pi)*sxy*(sin(angulo)+(pi*matrix(1,n,n)-angulo)*cosalfa)
 GC1<-GC1/median(GC1)
 colnames(GC1)<-rownames(X)
 rownames(GC1)<-rownames(X)

 return(GC1)
}

####Arc-cosine kernel with L>1########
AK1<-K.AC1(XF)
X.AK1=t(chol(AK1))
K.AC.L<-function(GC,nl){
 n<-nrow(GC)
 GC1<-GC
 for ( l in 1:nl){
  Aux<-tcrossprod(diag(GC))
  cosalfa<-GC*(Aux^(-1/2))
  cosa<-as.vector(cosalfa)
  cosa[which(cosalfa>1)]<-1
  angulo<-acos(cosa)
  angulo<-matrix(angulo,n,n)
  GC<-(1/pi)*(Aux^(1/2))*(sin(angulo)+(pi*matrix(1,n,n)-angulo)
*cos(angulo))
 }
  GC<-GC/median(GC)
  rownames(GC)<-rownames(GC1)
 colnames(GC)<-colnames(GC1)
 return(GC)
}
AK1=AK1
AK2<-K.AC.L(GC=AK1,nl=2)
X.AK2=t(chol(AK2))
AK3<-K.AC.L(GC=AK1,nl=3)
X.AK3=t(chol(AK3))
AK4<-K.AC.L(GC=AK1,nl=4)
.AK4=t(chol(AK4))

##########Cholesky of each kernel#######
X_ker=list(X.lin=X.lin, X.poly=X.poly,X.sig=X.sig,X.rad=X.rad,X.
exp=X.exp,X.AK1=X.AK1, X.AK2=X.AK2,X.AK3=X.AK3,X.AK4=X.AK4)
kernel.name=c
("Linear","Polynomial","Sigmoid","Gaussian","Exponential", "AK1",
"AK2","AK3","AK4")

#####K-fold cross-validation
n=nrow(XF)
No.folds=10
set.seed(10)
Grpv= findInterval(cut(sample(1:n,n),breaks=No.folds),1:n)

Y=y
Env_name=colnames(y)
```

```
rownames(Y)=1:n
Pred_all_kelnels<-data.frame()
for (t in 1:4){
y2=Y[,t]

results<-c() #save cross-validation results
for (k in 1:9){
MSE_Part=c()
for(r in 1:No.folds) {
#r=1
Xstar=X_ker[[k]]
rownames(Xstar)=1:n
X2=Xstar
actual_CV=r
y1=y2

positionTST=which(Grpv==r)

y_tr = y1[-positionTST] ; X_tr =  X2[-positionTST,];
y_tst = y1[positionTST]; X_tst = X2[positionTST,]

A_RR = cv.glmnet(X_tr,y_tr,family='gaussian',
        alpha=0,type.measure='mse')
ypred= as.numeric(predict(A_RR,newx=X_tst,s='lambda.min',
type='class'))
Predicted=ypred
Observed=as.numeric(y_tst)

MSE=mean((Predicted-Observed)^2)
MSE_Part=c(MSE_Part,MSE)
}
MSE_Part
mean(MSE_Part)

results=c(results,mean(MSE_Part))
}
results1=t(results)
colnames(results1)=kernel.name
Pred_all_kelnels=rbind(Pred_all_kelnels,data.frame(Env=Env_name
[t],results1))
}
Pred_all_kelnels
write.csv(Pred_all_kelnels, file ="Kernel_Example_8.1v2.csv")
```

Appendix 2

R code for manually implementing nine kernels (linear, polynomial, sigmoid, Gaussian, exponential, AK1, AK2, AK3, and AK4) with a binary response variable; the results are provided in Table 8.2.

```
rm(list=ls(all=TRUE))
library(plyr)
library(tidyr)
library(dplyr)
library(glmnet)

load("Data_Toy_EYT.RData")
ls()
Pheno=Pheno_Toy_EYT
dim(G_Toy_EYT)
XF=t(chol(G_Toy_EYT))
dim(XF)
head(XF[,1:5])

#######Linear Kernel###########
K.linear=function(x1, x2=x1,gamma=1) {gamma*(as.matrix(x1)%*%t(as.
matrix(x2))) }
K.lin=K.linear(x1=XF,x2=XF,gamma=1/ncol(XF))
dim(K.lin)
X.lin=t(chol(K.lin))

#######Polynomial Kernel###########
K.polynomial=function(x1, x2=x1, gamma=1, b=0, p=3){
 (gamma*(as.matrix(x1)%*%t(x2))+b)^p}

K.poly=K.polynomial(x1=XF,x2=XF,gamma=1/ncol(XF))
dim(K.poly)
X.poly=t(chol(K.poly))

#######Sigmoid Kernel###########
K.sigmoid=function(x1,x2=x1, gamma=1/ncol(XF), b=0)
{ tanh(gamma*(as.matrix(x1)%*%t(x2))+b) }
K.sig=K.sigmoid(x1=XF,x2=XF, gamma=1/1*ncol(XF))
dim(K.sig)
ei=eigen(K.sig)
pos_neg=which(ei$values<0)
Eigenv=ei$value
Eigenv[pos_neg]=0
Eigenv
X.sig=ei$vectors%*%diag(sqrt(Eigenv))%*%t(ei$vectors)

#######Radial Kernel###########
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j])^2)))}
K.rad=K.radial(x1=XF,x2=XF, gamma=1/ncol(XF))
dim(K.rad)
X.rad=t(chol(K.rad))

#######Exponential Kernel###########
K.exponential=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
          Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]))))}

K.exp=K.exponential(x1=XF,x2=XF, gamma=1/ncol(XF))
dim(K.exp)
X.exp=t(chol(K.exp))

############Arc-cosine kernel with L=1#######
K.AC1<-function(X){
 n<-nrow(X)
 cosalfa<-cor(t(X))
 angulo<-acos(cosalfa)
 mag<-sqrt(apply(X,1,function(x) crossprod(x)))
 sxy<-tcrossprod(mag)
 GC1<-(1/pi)*sxy*(sin(angulo)+(pi*matrix(1,n,n)-angulo)*cosalfa)
 GC1<-GC1/median(GC1)
 colnames(GC1)<-rownames(X)
 rownames(GC1)<-rownames(X)
 return(GC1)
}

####Arc-cosine kernel with L>1########
AK1<-K.AC1(XF)
X.AK1=t(chol(AK1))
K.AC.L<-function(GC,nl){
 n<-nrow(GC)
 GC1<-GC
 for ( l in 1:nl){
  Aux<-tcrossprod(diag(GC))
  cosalfa<-GC*(Aux^(-1/2))
  cosa<-as.vector(cosalfa)
  cosa[which(cosalfa>1)]<-1
  angulo<-acos(cosa)
  angulo<-matrix(angulo,n,n)

  GC<-(1/pi)*(Aux^(1/2))*(sin(angulo)+(pi*matrix(1,n,n)-angulo)
*cos(angulo))
 }

 GC<-GC/median(GC)
 rownames(GC)<-rownames(GC1)
 colnames(GC)<-colnames(GC1)
 return(GC)
}
AK1=AK1
AK2<-K.AC.L(GC=AK1,nl=2)
X.AK2=t(chol(AK2))
AK3<-K.AC.L(GC=AK1,nl=3)
X.AK3=t(chol(AK3))
AK4<-K.AC.L(GC=AK1,nl=4)
X.AK4=t(chol(AK4))
```

```
########Design matrices###
ZE=model.matrix(~0+as.factor(Pheno$Env))
ZL=model.matrix(~0+as.factor(Pheno$GID))
cbind(colnames(ZL), colnames(G_Toy_EYT))
###########Cholesky of each kernel#######
X_ker=list(X.lin=cbind(ZE,ZL%*%X.lin), X.poly=cbind(ZE,ZL%*%X.
poly),X.sig=cbind(ZE,ZL%*%X.sig),X.rad=cbind(ZE,ZL%*%X.rad),X.
exp=cbind(ZE,ZL%*%X.exp),X.AK1=cbind(ZE,ZL%*%X.AK1), X.AK2=cbind
(ZE,ZL%*%X.AK2),X.AK3=cbind(ZE,ZL%*%X.AK3),X.AK4=cbind(ZE,ZL%*%X.
AK4))
kernel.name=c
("Linear","Polynomial","Sigmoid","Gaussian","Exponential", "AK1",
"AK2","AK3","AK4")

#####K-fold cross-validation
n=nrow(Pheno)
No.folds=10
set.seed(10)
Grpv= findInterval(cut(sample(1:n,n),breaks=No.folds),1:n)

########Response variable##################################
y2=Pheno$Height
results<-data.frame() #save cross-validation results
for(r in 1:No.folds) {
y1=y2
positionTST=which(Grpv==r)
PCCC_Part=c()
for (k in 1:9){
Xstar=X_ker[[k]]
rownames(Xstar)=1:n
X2=Xstar
y_tr = y1[-positionTST] ; X_tr = X2[-positionTST,];
y_tst = y1[positionTST]; X_tst = X2[positionTST,]

A_RR = cv.glmnet(X_tr,y_tr,family='binomial',
        alpha=0,type.measure='class')
ypred= as.numeric(predict(A_RR,newx=X_tst,s='lambda.min',
type='class'))
Predicted=ypred
Observed=as.numeric(y_tst)

PCCC=1-mean(Predicted!=Observed)
PCCC_Part=c(PCCC_Part,PCCC)
}
names(PCCC_Part)=kernel.name
results=rbind(results,data.frame(fold=r,t(PCCC_Part)))
}
results
apply(results[,-1],2,mean)
write.csv(results, file ="Kernel_Binary_Example_Table_8.2.csv")
```

Appendix 3

R code for implementing three default kernels in rrBLUP (linear, Gaussian, and Exponential), and the results are provided in Table 8.4.

```
library(rrBLUP) #load rrBLUP
library(BGLR) #load BLR
data(wheat) #load wheat data
X=wheat.X
dim(X)
X <- 2*X-1 #recode genotypes
X=scale(X)
t=4
y <-wheat.Y[,t] #yields from E1
yy=y
MSE=function(yobserved,ypredicted){
 MSE=mean((yobserved-ypredicted)^2)
}
n_records=nrow(X)
n_folds=10
set.seed(10)
sets <- findInterval(cut(sample(1:n_records, n_records),
                breaks=n_folds), 1:n_records)
results=data.frame()
for (i in 1:n_folds){
# i=1
trn <- which(sets!=i)
tst <- which(sets==i)
ans.RR<-kinship.BLUP(y=y[trn],
          G.train=X[trn,],G.pred=X[tst,])
#accuracy with RR
Cor_RR=cor(ans.RR$g.pred,yy[tst])
MSE_RR=MSE(ans.RR$g.pred,yy[tst])

ans.GAUSS<-kinship.BLUP(y=y[trn],
           G.train=X[trn,],G.pred=X[tst,],
           K.method="GAUSS")
#accuracy with GAUSS
Cor_GAUSS=cor(ans.GAUSS$g.pred,yy[tst])
MSE_GAUSS=MSE(ans.GAUSS$g.pred,yy[tst])

ans.EXP<-kinship.BLUP(y=y[trn],
           G.train=X[trn,],G.pred=X[tst,],
           K.method="EXP")
#accuracy with EXponential
Cor_EXP=cor(ans.EXP$g.pred,yy[tst])
MSE_EXP=MSE(ans.EXP$g.pred,yy[tst])

results=rbind(results,data.frame(Fold=i, MSE_RR=MSE_RR,
MSE_GAUSS=MSE_GAUSS, MSE_EXP=MSE_EXP,Cor_RR=Cor_RR,
Cor_GAUSS=Cor_GAUSS, Cor_EXP=Cor_EXP))
}
results

write.csv(results,file="Kernel_Mixed_Example_Table_8.4.csv")
```

Appendix 4

R code for manually implementing nine kernels (linear, polynomial, sigmoid, Gaussian, exponential, AK1, AK2, AK3, and AK4); the results are provided in Table 8.5.

```
library(rrBLUP) #load rrBLUP
library(BGLR) #load BLR
data(wheat) #load wheat data
X=wheat.X
dim(X)
X <- 2*X-1 #recode genotypes

XF=scale(wheat.X)
dim(XF)
head(XF[,1:5])

########Linear Kernel############
K.linear=function(x1, x2=x1,gamma=1) {gamma*(as.matrix(x1)%*%t(as.
matrix(x2))) }
K.lin=K.linear(x1=XF,x2=XF,gamma=1/ncol(XF))
dim(K.lin)

########Polynomial Kernel############
K.polynomial=function(x1, x2=x1, gamma=1, b=0, p=3){
 (gamma*(as.matrix(x1)%*%t(x2))+b)^p}

K.poly=K.polynomial(x1=XF,x2=XF,gamma=1/ncol(XF))
dim(K.poly)

########Sigmoid Kernel############
K.sigmoid=function(x1,x2=x1, gamma=1/ncol(XF), b=0)
{ tanh(gamma*(as.matrix(x1)%*%t(x2))+b) }
K.sig=K.sigmoid(x1=XF,x2=XF)
ei=eigen(K.sig)
pos_neg=which(ei$values<0)
Eigenv=ei$value
Eigenv[pos_neg]=0
Eigenv
K.sig=ei$vectors%*%diag((Eigenv))%*%t(ei$vectors)
########Radial Kernel############
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
          Vectorize(function(i, j) l2norm(x1[i,]-x2[,j])^2)))}
K.rad=K.radial(x1=XF,x2=XF, gamma=1/ncol(XF))
dim(K.rad)

########Exponential Kernel############
K.exponential=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
          Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]))))}

K.exp=K.exponential(x1=XF,x2=XF, gamma=1/ncol(XF))
dim(K.exp)
############Arc-cosine kernel with L=1#######
K.AC1<-function(X){
 n<-nrow(X)
 cosalfa<-cor(t(X))
 angulo<-acos(cosalfa)
 mag<-sqrt(apply(X,1,function(x) crossprod(x)))
 sxy<-tcrossprod(mag)
 GC1<-(1/pi)*sxy*(sin(angulo)+(pi*matrix(1,n,n)-angulo)*cosalfa)
 GC1<-GC1/median(GC1)
 colnames(GC1)<-rownames(X)
 rownames(GC1)<-rownames(X)

 return(GC1)
}

####Arc-cosine kernel with L>1########
AK1<-K.AC1(XF)
K.AC.L<-function(GC,nl){
 n<-nrow(GC)
 GC1<-GC

 for ( l in 1:nl){
  Aux<-tcrossprod(diag(GC))
  cosalfa<-GC*(Aux^(-1/2))
  cosa<-as.vector(cosalfa)
  cosa[which(cosalfa>1)]<-1
  angulo<-acos(cosa)
  angulo<-matrix(angulo,n,n)
  GC<-(1/pi)*(Aux^(1/2))*(sin(angulo)+(pi*matrix(1,n,n)-angulo)
*cos(angulo))
 }
 GC<-GC/median(GC)
 rownames(GC)<-rownames(GC1)
 colnames(GC)<-colnames(GC1)
 return(GC)
}
AK1=AK1
AK2<-K.AC.L(GC=AK1,nl=2)
AK3<-K.AC.L(GC=AK1,nl=3)
AK4<-K.AC.L(GC=AK1,nl=4)
```

```
###########Cholesky of each kernel#######
K_ker=list(K.lin=K.lin, K.poly=K.poly,K.sig=K.sig,K.rad=K.rad,K.
exp=K.exp,K.AK1=AK1, K.AK2=AK2,K.AK3=AK3,K.AK4=AK4)
kernel.name=c
("Linear","Polynomial","Sigmoid","Gaussian","Exponential", "AK1",
"AK2","AK3","AK4")

#####K-fold cross-validation
n=nrow(XF)
No.folds=10
set.seed(10)
Grpv= findInterval(cut(sample(1:n,n),breaks=No.folds),1:n)
Env_name=colnames(wheat.Y)
t=4
 y2=wheat.Y[,t]
 results_ker=data.frame() #save cross-validation results

  for (k in 1:9){
  Kstar=K_ker[[k]]
  rownames(Kstar)=1:n
  results=data.frame()
  for(r in 1:No.folds) {
   y1=y2
   positionTST=which(Grpv==r)
   y1[positionTST]=NA
    I=diag(n)
    fit_Mix <- mixed.solve(y=y1,Z=I,K=Kstar)
    Pred_y=c(fit_Mix$beta)*rep(1,nrow(Kstar))+c(fit_Mix$u)
    MSE=mean((Pred_y[positionTST]-y2[positionTST])^2)
    results=rbind(results,data.frame(MSE=MSE))
   }
  results
   results_ker=rbind(results_ker,data.frame(Kernel=kernel.name[k],t
(results)))
   }
  results_ker
  colnames(results_ker)=c("Kernel",
"1","2","3","4","5","6","7","8","9","10")
  results_ker
write.csv(results_ker, file ="Example_Mixed_Table_8.5.csv")
```

Appendix 5

R code for tuning the number of hidden layers in the arc-cosine kernel (Fig. 8.2).

```
library(rrBLUP) #load rrBLUP
library(BGLR) #load BLR
data(wheat) #load wheat data
X=wheat.X
```

```
dim(X)
X <- 2*X-1 #recode genotypes
XF=scale(wheat.X)
rownames(XF)=1:599
dim(XF)
head(XF[,1:5])

############Arc-cosine kernel with L=1#######
K.AC1<-function(X){
 n<-nrow(X)
 cosalfa<-cor(t(X))
 angulo<-acos(cosalfa)
 mag<-sqrt(apply(X,1,function(x) crossprod(x)))
 sxy<-tcrossprod(mag)
 GC1<-(1/pi)*sxy*(sin(angulo)+(pi*matrix(1,n,n)-angulo)*cosalfa)
 GC1<-GC1/median(GC1)
 colnames(GC1)<-rownames(X)
 rownames(GC1)<-rownames(X)
 return(GC1)
}

####Arc-cosine kernel with L>1########
AK1<-K.AC1(XF)
K.AC.L<-function(GC,nl){
 n<-nrow(GC)
 GC1<-GC

 for ( l in 1:nl){

  Aux<-tcrossprod(diag(GC))
  cosalfa<-GC*(Aux^(-1/2))
  cosa<-as.vector(cosalfa)
  cosa[which(cosalfa>1)]<-1

  angulo<-acos(cosa)
  angulo<-matrix(angulo,n,n)
  GC<-(1/pi)*(Aux^(1/2))*(sin(angulo)+(pi*matrix(1,n,n)-angulo)
*cos(angulo))
  }
 GC<-GC/median(GC)
 rownames(GC)<-rownames(GC1)
 colnames(GC)<-colnames(GC1)
 return(GC)
}
AK1=AK1
AK2<-K.AC.L(GC=AK1,nl=2)
dim(AK2)

#####K-fold cross-validation
n=nrow(XF)
No.folds=4
set.seed(10)
```

```
Grpv= findInterval(cut(sample(1:n,n),breaks=No.folds),1:n)
Env_name=colnames(wheat.Y)

t=4
rownames(wheat.Y)=1:599
y2=wheat.Y[,t]
results=data.frame()
for(r in 1:No.folds) {
#  r=4
  y1=y2
  positionTST=which(Grpv==r)
  positionTRN=which(Grpv!=r)
  n_inner=length(positionTRN)
  ICV= findInterval(cut(sample(1:length(positionTRN),n_inner),
breaks=No.folds),1:n_inner)
Lvec=1:10
Ave_MSE_L=c()
for (L in 1:length(Lvec)){
  results_Inner=c()
for (j in 1:No.folds){
  y1_trn=y1[positionTRN]
  y1_trnI=y1_trn
  AKL<-K.AC.L(GC=AK1,nl=L)
  Kstar_inner=AKL[positionTRN,positionTRN]
  pos_TST=which(ICV==j)
  y1_trnI[pos_TST]=NA
  length(y1_trn)
  I=diag(length(y1_trn))
  fit_MixI <- mixed.solve(y=y1_trnI,Z=I,K=Kstar_inner)
  Pred_yI=c(fit_MixI$beta)*rep(1,nrow(Kstar_inner))+c(fit_MixI$u)
  MSE_Inner=mean((Pred_yI[pos_TST]-y1_trn[pos_TST])^2)
  results_Inner=c(results_Inner,MSE_Inner)
  }
MSE_L=mean(results_Inner)
MSE_L
Ave_MSE_L=c(Ave_MSE_L,MSE_L)
opt_mse=which.min(Ave_MSE_L)
L_opt=Lvec[opt_mse]
}

AKL_opt<-K.AC.L(GC=AK1,nl=L_opt )
Kstar=AKL_opt
y1[positionTST]=NA
I=diag(n)
   fit_Mix <- mixed.solve(y=y1,Z=I,K=Kstar)
   Pred_y=c(fit_Mix$beta)*rep(1,nrow(Kstar))+c(fit_Mix$u)
   MSE=mean((Pred_y[positionTST]-y2[positionTST])^2)
   results=rbind(results,data.frame(MSE=MSE))}
 results
apply(results,2,mean)
par(mar=c(4,6,6,4))
plot(1:10,Ave_MSE_L, ylab="Mean Square Error",xlab="Number of hidden
layers",lwd=12, cex.axis =1.5, cex.lab = 2)
```

Appendix 6

R code for implementing nine kernels under a Bayesian kernel BLUP approach with only genotypic effects in the predictor (Table 8.6).

```
rm(list=ls())
library(BGLR)
load('dat_ls_E1.RData',verbose=TRUE)
#Phenotypic data
dat_F = dat_ls$dat_F
head(dat_F)
#Marker data
dat_M = dat_ls$dat_M
dim(dat_M)

dat_F = transform(dat_F, GID = as.character(GID))
head(dat_F,5)
dim(dat_F)

#Matrix design of markers
Pos = match(dat_F$GID,row.names(dat_M))
XM = dat_M[Pos,]
XM = scale(XM)
dim(XM)

n = dim(dat_F)[1]
y = dat_F$y

#10 random partitions
K = 10
set.seed(1)
PT = replicate(K,sample(n,0.20*n))

Tab = data.frame(PT = 1:K,MSEP = NA)
set.seed(1)

#GBLUP=Linear kernel
dat_M = scale(dat_M)
G = tcrossprod(dat_M)/dim(dat_M)[2]
dim(G)
#Matrix design of GIDs
Z = model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))
Ga = Z%*%G%*%t(Z)
ETA_GB = list(list(model='RKHS',K = Ga))
#Tab = data.frame(PT = 1:K,MSEP = NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_GB,nIter = 1e4,burnIn = 1e3,verbose = FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_GB[k] = mean((y[Pos_tst]-yp_ts)^2)
}

mean(Tab$MSEP_GB)
sd(Tab$MSEP_GB)

#######Polynomial Kernel###########
K.polynomial=function(x1, x2=x1, gamma=1, b=0, p=3){
 (gamma*(as.matrix(x1)%*%t(x2))+b)^p}

K.poly=K.polynomial(x1=dat_M,x2=dat_M,gamma=1/ncol(dat_M))
dim(K.poly)
K.poly.Exp= Z%*%K.poly%*%t(Z)
ETA_K.poly = list(list(model='RKHS',K=K.poly.Exp))

for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.poly,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_Poly[k] = mean((y[Pos_tst]-yp_ts)^2)
}

mean(Tab$MSEP_Poly)
sd(Tab$MSEP_Poly)
cor(y[Pos_tst],yp_ts)
plot(y[Pos_tst],yp_ts);abline(a=0,b=1)

#######Sigmoid Kernel###########
K.sigmoid=function(x1,x2=x1, gamma=1, b=0)
{ tanh(gamma*(as.matrix(x1)%*%t(x2))+b) }
K.sig=K.sigmoid(x1=dat_M,x2=dat_M,gamma=1/ncol(dat_M))
#K.sig=K.sig+diag(599)*0.1

K.sig.Exp= Z%*%K.sig%*%t(Z)
ETA_K.sig = list(list(model='RKHS',K=K.sig.Exp))
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.sig,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_sig[k] = mean((y[Pos_tst]-yp_ts)^2)
}
```

```
mean(Tab$MSEP_sig)
sd(Tab$MSEP_sig)
cor(y[Pos_tst],yp_ts)
plot(y[Pos_tst],yp_ts);abline(a=0,b=1)

#######Gaussian or Radial Kernel###########
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j])^2)))}
K.rad=K.radial(x1=dat_M,x2=dat_M, gamma=1/ncol(dat_M))
dim(K.rad)

K.Gauss.Exp= Z%*%K.rad%*%t(Z)
ETA_K.Gauss = list(list(model='RKHS',K=K.Gauss.Exp))
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.Gauss,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_Gauss[k] = mean((y[Pos_tst]-yp_ts)^2)
}

mean(Tab$MSEP_Gauss)
sd(Tab$MSEP_Gauss)
cor(y[Pos_tst],yp_ts)
plot(y[Pos_tst],yp_ts);abline(a=0,b=1)

#######Exponential Kernel###########
K.exponential=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]))))}

K.exp=K.exponential(x1=dat_M,x2=dat_M, gamma=1/ncol(dat_M))
dim(K.exp)

K.Expo.Exp= Z%*%K.exp%*%t(Z)
ETA_K.Expo = list(list(model='RKHS',K=K.Expo.Exp))
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.Expo,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_Expo[k] = mean((y[Pos_tst]-yp_ts)^2)
}
```

```
mean(Tab$MSEP_Expo)
sd(Tab$MSEP_Expo)
cor(y[Pos_tst],yp_ts)
plot(y[Pos_tst],yp_ts);abline(a=0,b=1)

############Arc-cosine kernel with deep=1#######
K.AK1<-function(X){
 n<-nrow(X)
 cosalfa<-cor(t(X))
 angulo<-acos(cosalfa)
 mag<-sqrt(apply(X,1,function(x) crossprod(x)))
 sxy<-tcrossprod(mag)
 GC1<-(1/pi)*sxy*(sin(angulo)+(pi*matrix(1,n,n)-angulo)*cosalfa)
 GC1<-GC1/median(GC1)
 colnames(GC1)<-rownames(X)
 rownames(GC1)<-rownames(X)

 return(GC1)
}

AK1<-K.AK1(dat_M)

K.AK1.Exp= Z%*%AK1%*%t(Z)
ETA_K.AK1 = list(list(model='RKHS',K=K.AK1.Exp))
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.AK1,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_AK1[k] = mean((y[Pos_tst]-yp_ts)^2)
}

mean(Tab$MSEP_AK1)
sd(Tab$MSEP_AK1)
cor(y[Pos_tst],yp_ts)
plot(y[Pos_tst],yp_ts);abline(a=0,b=1)

####Arc-cosine kernel with deep=2#####
AK_L<-function(GC,nl){
 n<-nrow(GC)
 GC1<-GC

 for ( l in 1:nl){

  Aux<-tcrossprod(diag(GC))
  cosalfa<-GC*(Aux^(-1/2))
  cosa<-as.vector(cosalfa)
  cosa[which(cosalfa>1)]<-1
  angulo<-acos(cosa)
  angulo<-matrix(angulo,n,n)

  GC<-(1/pi)*(Aux^(1/2))*(sin(angulo)+(pi*matrix(1,n,n)-angulo)
*cos(angulo))

 }

 GC<-GC/median(GC)

 rownames(GC)<-rownames(GC1)
 colnames(GC)<-colnames(GC1)
 return(GC)
}

AK2<-AK_L(GC=AK1,nl=2)
K.AK2.Exp= Z%*%AK2%*%t(Z)
ETA_K.AK2 = list(list(model='RKHS',K=K.AK2.Exp))
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.AK2,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_AK2[k] = mean((y[Pos_tst]-yp_ts)^2)
}

mean(Tab$MSEP_AK2)
sd(Tab$MSEP_AK2)
cor(y[Pos_tst],yp_ts)
plot(y[Pos_tst],yp_ts);abline(a=0,b=1)

#######Arc-cosine kernel with deep=3
AK3<-AK_L(GC=AK1,nl=3)
K.AK3.Exp= Z%*%AK3%*%t(Z)
ETA_K.AK3 = list(list(model='RKHS',K=K.AK3.Exp))
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.AK3,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_AK3[k] = mean((y[Pos_tst]-yp_ts)^2)
}

mean(Tab$MSEP_AK3)
sd(Tab$MSEP_AK3)
```

```
cor(y[Pos_tst],yp_ts)
plot(y[Pos_tst],yp_ts);abline(a=0,b=1)

#######Arc-cosine kernel with deep=4
AK4<-AK_L(GC=AK1,nl=4)
K.AK4.Exp= Z%*%AK4%*%t(Z)
ETA_K.AK4 = list(list(model='RKHS',K=K.AK4.Exp))
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.AK4,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_AK4[k] = mean((y[Pos_tst]-yp_ts)^2)
}

mean(Tab$MSEP_AK4)
sd(Tab$MSEP_AK4)
cor(y[Pos_tst],yp_ts)
plot(y[Pos_tst],yp_ts);abline(a=0,b=1)

write.csv(Tab,file='Tab_MSEP-Ex1-Kernels.csv',row.names = FALSE)
```

Appendix 7

R code for implementing seven kernels under a Bayesian kernel BLUP approach with effects of environment + genotype + genotype ×environment interaction in the predictor (Table 8.7).

```
rm(list=ls())
library(BGLR)
library(BMTME)
load('dat_ls_E2.RData',verbose=TRUE)
#Phenotypic data
dat_F = dat_ls$dat_F
head(dat_F)
dim(dat_F)
#Marker data
dat_M = dat_ls$dat_M
dim(dat_M)

dat_F = transform(dat_F, GID = as.character(GID))
head(dat_F)

head(dat_F,5)
#Matrix design for markers
```

```
Pos = match(dat_F$GID,row.names(dat_M))
XM = dat_M[Pos,]
dim(XM)
XM = scale(XM)
#Environment design matrix
XE = model.matrix(~0+Env,data=dat_F)[,-1]
K.E=XE%*%t(XE)
dim(K.E)

#GID design matrix and Environment-GID design matrix
#for RKHS models
Z_L = model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))

n=dim(dat_F)[1]
y=dat_F$y

#Number of random partitions
K=10
set.seed(1)
PT = replicate(K,sample(n,0.20*n))

########Linear kernel=GBLUP
dat_M=scale(dat_M)
G=tcrossprod(dat_M)/dim(dat_M)[2]
dim(G)
#Covariance matrix for Zg
K_L=Z_L%*%G%*%t(Z_L)
#Covariance matrix for random effects ZEg
K_LE= K.E*K_L

ETA_K.Linear=list(list(model='FIXED',X=XE),list(model='RKHS',
K=K_L),
        list(model='RKHS',K=K_LE))

#######Polynomial Kernel############
K.polynomial=function(x1, x2=x1, gamma=1, b=0, p=3){
 (gamma*(as.matrix(x1)%*%t(x2))+b)^p}

K.poly=K.polynomial(x1=dat_M,x2=dat_M,gamma=1/ncol(dat_M))
dim(K.poly)
K.poly.Exp=Z_L%*%K.poly%*%t(Z_L)

#Covariance matrix for random effects ZEg
K.GE.poly= K.E*K.poly.Exp
ETA_K.poly = list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
poly.Exp),
          list(model='RKHS',K=K.GE.poly))

#######Sigmoid Kernel############
K.sigmoid=function(x1,x2=x1, gamma=1, b=0)
```

```
{ tanh(gamma*(as.matrix(x1)%*%t(x2))+b) }
K.sig=K.sigmoid(x1=dat_M,x2=dat_M,gamma=1/ncol(dat_M))

K.sig.Exp= Z_L%*%K.sig%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.sig= K.E*K.sig.Exp
ETA_K.sig = list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
sig.Exp),
          list(model='RKHS',K=K.GE.sig))

########Gaussian or Radial Kernel############
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
          Vectorize(function(i, j) l2norm(x1[i,]-x2[,j])^2)))}
K.rad=K.radial(x1=dat_M,x2=dat_M, gamma=1/ncol(dat_M))
dim(K.rad)

K.Gauss.Exp= Z_L%*%K.rad%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.Gauss=K.E*K.Gauss.Exp
ETA_K.Gauss=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
Gauss.Exp),
         list(model='RKHS',K=K.GE.Gauss))

########Exponential Kernel############
K.exponential=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
          Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]))))}

K.exp=K.exponential(x1=dat_M,x2=dat_M, gamma=1/ncol(dat_M))
dim(K.exp)
K.Expo.Exp= Z_L%*%K.exp%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.Exp=K.E*K.Expo.Exp
ETA_K.Exp=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
Expo.Exp),
        list(model='RKHS',K=K.GE.Exp))
############Arc-cosine kernel with deep=1#######
K.AK1<-function(X){
 n<-nrow(X)
 cosalfa<-cor(t(X))
 angulo<-acos(cosalfa)
 mag<-sqrt(apply(X,1,function(x) crossprod(x)))
 sxy<-tcrossprod(mag)
 GC1<-(1/pi)*sxy*(sin(angulo)+(pi*matrix(1,n,n)-angulo)*cosalfa)
 GC1<-GC1/median(GC1)
 colnames(GC1)<-rownames(X)
 rownames(GC1)<-rownames(X)

 return(GC1)
}
```

```
AK1<-K.AK1(dat_M)

K.AK1.Exp= Z_L%*%AK1%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.AK1=K.E*K.AK1.Exp
ETA_K.AK1=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
AK1.Exp),
        list(model='RKHS',K=K.GE.AK1))
####Arc-cosine kernel with deep=4#####
AK_L<-function(GC,nl){
 n<-nrow(GC)
 GC1<-GC

 for ( l in 1:nl){
  Aux<-tcrossprod(diag(GC))
  cosalfa<-GC*(Aux^(-1/2))
  cosa<-as.vector(cosalfa)
  cosa[which(cosalfa>1)]<-1
  angulo<-acos(cosa)
  angulo<-matrix(angulo,n,n)
  GC<-(1/pi)*(Aux^(1/2))*(sin(angulo)+(pi*matrix(1,n,n)-angulo)
*cos(angulo))
 }
  GC<-GC/median(GC)

 rownames(GC)<-rownames(GC1)
 colnames(GC)<-colnames(GC1)
 return(GC)
}

AK4<-AK_L(GC=AK1,nl=4)
K.AK4.Exp= Z_L%*%AK4%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.AK4=K.E*K.AK4.Exp
ETA_K.AK4=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
AK4.Exp),
        list(model='RKHS',K=K.GE.AK4))

Tab1_m = data.frame(PT = 1:K,MSEP = NA)
Tab1_MSE = data.frame(PT = 1:K,MSEP = NA)
Tab1_Cor = data.frame(PT = 1:K,MSEP = NA)
for(k in 1:K)
{
 set.seed(1)
 Pos_tst =PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.Linear,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
  yp_ts = A$yHat[Pos_tst]
  Tab1_MSE$Linear[k] = mean((y[Pos_tst]-yp_ts)^2)
  Tab1_Cor$Linear[k] = cor(y[Pos_tst],yp_ts)
  A = BGLR(y=y_NA,ETA=ETA_K.poly,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
  yp_ts = A$yHat[Pos_tst]
  Tab1_MSE$poly[k] = mean((y[Pos_tst]-yp_ts)^2)
  Tab1_Cor$poly[k]  = cor(y[Pos_tst],yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.sig,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
  yp_ts = A$yHat[Pos_tst]
  Tab1_MSE$sig[k] = mean((y[Pos_tst]-yp_ts)^2)
  Tab1_Cor$sig[k]  = cor(y[Pos_tst],yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.Gauss,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
  yp_ts = A$yHat[Pos_tst]
  Tab1_MSE$Gauss[k] = mean((y[Pos_tst]-yp_ts)^2)
  Tab1_Cor$Gauss[k]  = cor(y[Pos_tst],yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.Exp,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
  yp_ts = A$yHat[Pos_tst]
  Tab1_MSE$Exp[k] = mean((y[Pos_tst]-yp_ts)^2)
  Tab1_Cor$Exp[k]  = cor(y[Pos_tst],yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.AK1,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
  yp_ts = A$yHat[Pos_tst]
  Tab1_MSE$AK1[k] = mean((y[Pos_tst]-yp_ts)^2)
  Tab1_Cor$AK1[k]  = cor(y[Pos_tst],yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.AK4,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
  yp_ts = A$yHat[Pos_tst]
  Tab1_MSE$AK4[k] = mean((y[Pos_tst]-yp_ts)^2)
  Tab1_Cor$AK4[k]  = cor(y[Pos_tst],yp_ts)
 }
Tab1_MSE
apply(Tab1_MSE[,-c(1:2)],2,mean)
write.csv(Tab1_MSE,file="Tab_MSEP.Ex2_kernels_New.csv")
```

Appendix 8

R code for implementing seven kernels under a Bayesian kernel BLUP with a binary response variable with effects of environment + genotype + genotype ×environment interaction in the predictor (Table 8.8).

```
rm(list=ls())
library(BGLR)
library(BMTME)
load('Data_Toy_EYT.RData',verbose=TRUE)
ls()
#Phenotypic data
dat_F =Pheno_Toy_EYT
head(dat_F)
dim(dat_F)
#Marker data
dat_M =t(chol(G_Toy_EYT))
dim(dat_M)

dat_F = transform(dat_F, GID = as.character(GID))
head(dat_F)

head(dat_F,5)
#Matrix design for markers
Pos = match(dat_F$GID,row.names(dat_M))
XM = dat_M[Pos,]

XM =XM
#Environment design matrix
XE = model.matrix(~0+Env,data=dat_F)[,-1]
K.E=XE%*%t(XE)
dim(K.E)

#GID design matrix for lines
Z_L = model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))
n=dim(dat_F)[1]
y=dat_F$Height

#Number of random partitions
K=10
set.seed(1)
PT = replicate(K,sample(n,0.20*n))

########Linear kernel=GBLUP
G=G_Toy_EYT
dim(G)
#Covariance matrix for lines
K_L=Z_L%*%G%*%t(Z_L)
#Covariance matrix for GE term
K_LE= K.E*K_L
##########Predictor
ETA_K.Linear=list(list(model='FIXED',X=XE),list(model='RKHS',
K=K_L),
        list(model='RKHS',K=K_LE))

########Polynomial Kernel############
K.polynomial=function(x1, x2=x1, gamma=1, b=0, p=3){
 (gamma*(as.matrix(x1)%*%t(x2))+b)^p}
```

```
K.poly=K.polynomial(x1=dat_M,x2=dat_M,gamma=1/ncol(dat_M))
dim(K.poly)
K.poly.Exp=Z_L%*%K.poly%*%t(Z_L)

#Covariance matrix for GE term
K.GE.poly= K.E*K.poly.Exp
ETA_K.poly = list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
poly.Exp),
          list(model='RKHS',K=K.GE.poly))

########Sigmoid Kernel############
K.sigmoid=function(x1,x2=x1, gamma=1, b=0)
{ tanh(gamma*(as.matrix(x1)%*%t(x2))+b) }
K.sig=K.sigmoid(x1=dat_M,x2=dat_M,gamma=1/ncol(dat_M))

K.sig.Exp= Z_L%*%K.sig%*%t(Z_L)
#Covariance matrix for GE term
K.GE.sig= K.E*K.sig.Exp
ETA_K.sig = list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
sig.Exp),
         list(model='RKHS',K=K.GE.sig))

########Gaussian or Radial Kernel############
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j])^2)))}
K.rad=K.radial(x1=dat_M,x2=dat_M, gamma=1/ncol(dat_M))
dim(K.rad)

K.Gauss.Exp= Z_L%*%K.rad%*%t(Z_L)
#Covariance matrix for GE term
K.GE.Gauss=K.E*K.Gauss.Exp
ETA_K.Gauss=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
Gauss.Exp),
        list(model='RKHS',K=K.GE.Gauss))

########Exponential Kernel############
K.exponential=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]))))}

K.exp=K.exponential(x1=dat_M,x2=dat_M, gamma=1/ncol(dat_M))
dim(K.exp)
K.Expo.Exp= Z_L%*%K.exp%*%t(Z_L)
#Covariance matrix for GE term
K.GE.Exp=K.E*K.Expo.Exp
ETA_K.Exp=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
Expo.Exp),
        list(model='RKHS',K=K.GE.Exp))
############Arc-cosine kernel with deep=1#######
K.AK1<-function(X){
 n<-nrow(X)
 cosalfa<-cor(t(X))
 angulo<-acos(cosalfa)
 mag<-sqrt(apply(X,1,function(x) crossprod(x)))
 sxy<-tcrossprod(mag)
 GC1<-(1/pi)*sxy*(sin(angulo)+(pi*matrix(1,n,n)-angulo)*cosalfa)
 GC1<-GC1/median(GC1)
 colnames(GC1)<-rownames(X)
 rownames(GC1)<-rownames(X)
 return(GC1)
}

AK1<-K.AK1(dat_M)
K.AK1.Exp= Z_L%*%AK1%*%t(Z_L)
#Covariance matrix for GE term
K.GE.AK1=K.E*K.AK1.Exp
ETA_K.AK1=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
AK1.Exp),
        list(model='RKHS',K=K.GE.AK1))
####Arc-cosine kernel with deep=4#####
AK_L<-function(GC,nl){
 n<-nrow(GC)
 GC1<-GC

 for ( l in 1:nl){

  Aux<-tcrossprod(diag(GC))
  cosalfa<-GC*(Aux^(-1/2))
  cosa<-as.vector(cosalfa)
  cosa[which(cosalfa>1)]<-1
  angulo<-acos(cosa)
  angulo<-matrix(angulo,n,n)
  GC<-(1/pi)*(Aux^(1/2))*(sin(angulo)+(pi*matrix(1,n,n)-angulo)
*cos(angulo))
 }

 GC<-GC/median(GC)

 rownames(GC)<-rownames(GC1)
 colnames(GC)<-colnames(GC1)
 return(GC)
}

AK4<-AK_L(GC=AK1,nl=4)
K.AK4.Exp= Z_L%*%AK4%*%t(Z_L)
#Covariance matrix for GE term
K.GE.AK4=K.E*K.AK4.Exp
ETA_K.AK4=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
AK4.Exp),
        list(model='RKHS',K=K.GE.AK4))
```

```
Tab1_PCCC = data.frame(PT = 1:K,PCCC = NA)

for(k in 1:K) {
 set.seed(1)
 Pos_tst =PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.Linear,response_type="ordinal",nIter =
1e4,burnIn = 1e3,verbose = FALSE)
 Probs = A$probs[Pos_tst,]
 yp_ts = apply(Probs,1,which.max)-1
  Tab1_PCCC$Linear[k] = 1-mean(y[Pos_tst]!=yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.poly,response_type="ordinal",nIter =
1e4,burnIn = 1e3,verbose = FALSE)
  Probs = A$probs[Pos_tst,]
  yp_ts = apply(Probs,1,which.max)-1
  Tab1_PCCC$poly[k] = 1-mean(y[Pos_tst]!=yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.sig,response_type="ordinal",nIter = 1e4,
burnIn = 1e3,verbose = FALSE)
  Probs = A$probs[Pos_tst,]
  yp_ts = apply(Probs,1,which.max)-1
  Tab1_PCCC$sig[k] = 1-mean(y[Pos_tst]!=yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.Gauss,response_type="ordinal",nIter =
1e4,burnIn = 1e3,verbose = FALSE)
  Probs = A$probs[Pos_tst,]
  yp_ts = apply(Probs,1,which.max)-1
  Tab1_PCCC$Gauss[k] = 1-mean(y[Pos_tst]!=yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.Exp,response_type="ordinal",nIter = 1e4,
burnIn = 1e3,verbose = FALSE)
  Probs = A$probs[Pos_tst,]
  yp_ts = apply(Probs,1,which.max)-1
  Tab1_PCCC$Exp[k] = 1-mean(y[Pos_tst]!=yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.AK1,response_type="ordinal",nIter = 1e4,
burnIn = 1e3,verbose = FALSE)
  Probs = A$probs[Pos_tst,]
  yp_ts = apply(Probs,1,which.max)-1
  Tab1_PCCC$AK1[k] = 1-mean(y[Pos_tst]!=yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.AK4,response_type="ordinal",nIter = 1e4,
burnIn = 1e3,verbose = FALSE)
  Probs = A$probs[Pos_tst,]
  yp_ts = apply(Probs,1,which.max)-1
  Tab1_PCCC$AK4[k] = 1-mean(y[Pos_tst]!=yp_ts)
 }
Tab1_PCCC
apply(Tab1_PCCC[,-c(1:2)],2,mean)

write.csv(Tab1_PCCC,file="Tab_PCCC.Ex3_kernels_Final.csv")
```

Appendix 9

R code for implementing seven kernels under a multi-trait Bayesian kernel BLUP with a Gaussian response variable with effects of environment + genotype + genotype ×environment interaction in the predictor (Table 8.10).

```
rm(list=ls())
library(BGLR)
library(BMTME)
library(BGLR)
library(plyr)
library(tidyr)
library(dplyr)
load('Data_Toy_EYT.RData',verbose=TRUE)
ls()
#Phenotypic data
dat_F =Pheno_Toy_EYT
head(dat_F)
dim(dat_F)
#Marker data
dat_M =t(chol(G_Toy_EYT))
dim(dat_M)

dat_F = transform(dat_F, GID = as.character(GID))
head(dat_F)

head(dat_F,5)
#Matrix design for markers
Pos = match(dat_F$GID,row.names(dat_M))
XM = dat_M[Pos,]
dim(XM)
XM =XM
#Environment design matrix
XE = model.matrix(~0+Env,data=dat_F)[,-1]
K.E=XE%*%t(XE)

#GID design matrix of lines
Z_L = model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))

n=dim(dat_F)[1]
head(dat_F)
y=dat_F[,3:6]
head(y)

#Number of random partitions
K=10
set.seed(1)
PT = replicate(K,sample(n,0.20*n))
```

```
########Linear kernel=GBLUP
G=G_Toy_EYT
dim(G)
#Covariance matrix for Zg
K_L=Z_L%*%G%*%t(Z_L)
#Covariance matrix for random effects ZEg
K_LE= K.E*K_L

ETA_K.Linear=list(list(model='FIXED',X=XE),list(model='RKHS',
K=K_L),
        list(model='RKHS',K=K_LE))

########Polynomial Kernel############
K.polynomial=function(x1, x2=x1, gamma=1, b=0, p=3){
 (gamma*(as.matrix(x1)%*%t(x2))+b)^p}

K.poly=K.polynomial(x1=dat_M,x2=dat_M,gamma=1/ncol(dat_M))
dim(K.poly)
K.poly.Exp=Z_L%*%K.poly%*%t(Z_L)

#Covariance matrix for GE
K.GE.poly= K.E*K.poly.Exp
ETA_K.poly = list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
poly.Exp),
          list(model='RKHS',K=K.GE.poly))

########Sigmoid Kernel############
K.sigmoid=function(x1,x2=x1, gamma=1, b=0)
{ tanh(gamma*(as.matrix(x1)%*%t(x2))+b) }
K.sig=K.sigmoid(x1=dat_M,x2=dat_M,gamma=1/ncol(dat_M))

K.sig.Exp= Z_L%*%K.sig%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.sig= K.E*K.sig.Exp
ETA_K.sig = list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
sig.Exp),
         list(model='RKHS',K=K.GE.sig))

########Gaussian or Radial Kernel############
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j])^2)))}
K.rad=K.radial(x1=dat_M,x2=dat_M, gamma=1/ncol(dat_M))
dim(K.rad)

K.Gauss.Exp= Z_L%*%K.rad%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.Gauss=K.E*K.Gauss.Exp
ETA_K.Gauss=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
Gauss.Exp),
        list(model='RKHS',K=K.GE.Gauss))
```

```
#######Exponential Kernel###########
K.exponential=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
          Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]))))}

K.exp=K.exponential(x1=dat_M,x2=dat_M, gamma=1/ncol(dat_M))
dim(K.exp)
K.Expo.Exp= Z_L%*%K.exp%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.Exp=K.E*K.Expo.Exp
ETA_K.Exp=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
Expo.Exp),
         list(model='RKHS',K=K.GE.Exp))
###########Arc-cosine kernel with deep=1#######
K.AK1<-function(X){
 n<-nrow(X)
 cosalfa<-cor(t(X))
 angulo<-acos(cosalfa)
 mag<-sqrt(apply(X,1,function(x) crossprod(x)))
 sxy<-tcrossprod(mag)
 GC1<-(1/pi)*sxy*(sin(angulo)+(pi*matrix(1,n,n)-angulo)*cosalfa)
 GC1<-GC1/median(GC1)
 colnames(GC1)<-rownames(X)
 rownames(GC1)<-rownames(X)

 return(GC1)
}

AK1<-K.AK1(dat_M)

K.AK1.Exp= Z_L%*%AK1%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.AK1=K.E*K.AK1.Exp
ETA_K.AK1=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
AK1.Exp),
        list(model='RKHS',K=K.GE.AK1))
####Arc-cosine kernel with deep=4#####
AK_L<-function(GC,nl){
 n<-nrow(GC)
 GC1<-GC

 for ( l in 1:nl){
  Aux<-tcrossprod(diag(GC))
  cosalfa<-GC*(Aux^(-1/2))
  cosa<-as.vector(cosalfa)
  cosa[which(cosalfa>1)]<-1

  angulo<-acos(cosa)
  angulo<-matrix(angulo,n,n)
  GC<-(1/pi)*(Aux^(1/2))*(sin(angulo)+(pi*matrix(1,n,n)-angulo)
*cos(angulo))
 }
 GC<-GC/median(GC)
 rownames(GC)<-rownames(GC1)
 colnames(GC)<-colnames(GC1)
 return(GC)
}

AK4<-AK_L(GC=AK1,nl=4)
K.AK4.Exp= Z_L%*%AK4%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.AK4=K.E*K.AK4.Exp
ETA_K.AK4=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
AK4.Exp),
       list(model='RKHS',K=K.GE.AK4))

source('PC_MM.R')#See below

Tab1_Metrics = data.frame()

for(k in 1:K) {
#k=1
 set.seed(1)
 Pos_tst =PT[,k]
 y_NA = data.matrix(y)
 y_NA[Pos_tst,] = NA

 A1= Multitrait(y = y_NA, ETA=ETA_K.Linear,resCov = list(type ="UN",
S0=diag(4),df0= 5),
        nIter =10000, burnIn = 1000)
  Me_linear= PC_MM_f(y[Pos_tst,],A1$ETAHat[Pos_tst,],Env=dat_F$Env
[Pos_tst])
  Me_linear

  A2= Multitrait(y = y_NA, ETA=ETA_K.poly,resCov = list(type ="UN",
S0=diag(4),df0= 5),
         nIter =10000, burnIn = 1000)
  Me_poly= PC_MM_f(y[Pos_tst,],A2$ETAHat[Pos_tst,],Env=dat_F$Env
[Pos_tst])
  Me_poly

  A3= Multitrait(y = y_NA, ETA=ETA_K.sig,resCov = list(type ="UN",
S0=diag(4),df0= 5),
         nIter =10000, burnIn = 1000)
   Me_sig= PC_MM_f(y[Pos_tst,],A3$ETAHat[Pos_tst,],Env=dat_F$Env
[Pos_tst])
  Me_sig

  A4= Multitrait(y = y_NA, ETA=ETA_K.Gauss,resCov = list(type ="UN",
S0=diag(4),df0= 5),
         nIter =10000, burnIn = 1000)
  Me_Gauss= PC_MM_f(y[Pos_tst,],A4$ETAHat[Pos_tst,],Env=dat_F$Env
[Pos_tst])
  Me_Gauss
 A5= Multitrait(y = y_NA, ETA=ETA_K.Exp,resCov = list(type ="UN",
S0=diag(4),df0= 5),
         nIter =10000, burnIn = 1000)
 Me_Exp= PC_MM_f(y[Pos_tst,],A5$ETAHat[Pos_tst,],Env=dat_F$Env
[Pos_tst])
 Me_Exp

 A6= Multitrait(y = y_NA, ETA=ETA_K.AK1,resCov = list(type ="UN",
S0=diag(4),df0= 5),
         nIter =10000, burnIn = 1000)
 Me_AK1= PC_MM_f(y[Pos_tst,],A6$ETAHat[Pos_tst,],Env=dat_F$Env
[Pos_tst])
 Me_AK1

 A7= Multitrait(y = y_NA, ETA=ETA_K.AK4,resCov = list(type ="UN",
S0=diag(4),df0= 5),
         nIter =10000, burnIn = 1000)
 Me_AK4= PC_MM_f(y[Pos_tst,],A7$ETAHat[Pos_tst,],Env=dat_F$Env
[Pos_tst])
 Me_AK4
 Tab1_Metrics=rbind(Tab1_Metrics, data.frame(Fold=k,
Trait=Me_linear[,1],Env=Me_linear[,2],MSE_linear=Me_linear[,4],
MSE_poly=Me_poly[,4],MSE_sig=Me_sig[,4],MSE_Gauss=Me_Gauss[,4],
MSE_Exp=Me_Exp[,4],MSE_AK1=Me_AK1[,4],MSE_AK4=Me_AK4[,4]))
 }
Tab1_Metrics
Tab_R = Tab1_Metrics%>%group_by(Env,Trait)%>%select(MSE_linear,
MSE_poly,MSE_sig,MSE_Gauss,MSE_Exp,MSE_AK1,MSE_AK4)%>%summarise
(Linear= mean(MSE_linear),
   Polynomial= mean(MSE_poly),Sigmoid= mean(MSE_sig),Gaussian= mean
(MSE_Gauss),Exponential= mean(MSE_Exp),AK1= mean(MSE_AK4),AK4= mean
(MSE_AK4))
Tab_R = as.data.frame(Tab_R)
Tab_R

write.csv(Tab_R, file="Tab_R_MSE_C.Ex5_kernels_multi_trait_Final.
csv")
```

Appendix 10

R code for implementing the approximate kernel method proposed by Cuevas et al. (2020) with the wheat599 data set (Table 8.11).

```
rm(list=ls())
library(rrBLUP) #load rrBLUP
library(BGLR) #load BLR
data(wheat) #load wheat data
X=wheat.X
```

```
XF=scale(wheat.X)
rownames(XF)=1:599
dim(XF)
head(XF[,1:5])

############Linear kernel########################
Sparse_linear_kernel=function(m,X){
m=m
XF=X
p=ncol(XF)
pos_m=sample(1:nrow(XF),m)
####Step 1 compute K_m###############3
X_m=XF[pos_m,]
dim(X_m)
K_m=X_m%*%t(X_m)/p
dim(K_m)
######Step 2 compute K_n_m###########
K_n_m=XF%*%t(X_m)/p
dim(K_n_m)
######Step 3 compute eigenvalue decomposition of K_m######
EVD_K_m=eigen(K_m)
####Eigenvectors
U=EVD_K_m$vectors
###Eigenvalues###
S=EVD_K_m$values
####Square root of the inverse of eigenvalues #####
S_0.5_Inv=sqrt(1/S)
#####Diagonal matrix of square root of inverse of eigenvalues###
S_mat_Inv=diag(S_0.5_Inv)
######Computing matrix P
P=K_n_m%*%U%*%S_mat_Inv
return(P)}

#####K-fold cross-validation
n=nrow(XF)
No.folds=10
set.seed(2)
Grpv= findInterval(cut(sample(1:n,n),breaks=No.folds),1:n)
Grpv
Env_name=colnames(wheat.Y)
results_all_traits=data.frame()
for (t in 1:4){
t=t
rownames(wheat.Y)=1:599
y2=wheat.Y[,t]
mvec=c(15,32,74,132,264,599)
results_all=data.frame()

for (j in 1:6){
m=mvec[j]
P=Sparse_linear_kernel(m,X=XF)
results=data.frame()
start_time <- proc.time()
for(r in 1:No.folds) {
  y1=y2
  positionTST=which(Grpv==r)
  positionTRN=which(Grpv!=r)
  y1[positionTST] = NA
  ETA=list(list(model='BRR',X=P))
  A=BGLR(y=y1,ETA=ETA,nIter = 1e4,burnIn = 1e3,verbose = FALSE)
  yHat= A$yHat[positionTST]
  MSE=mean((y2[positionTST]-yHat)^2)
  Cor=cor(y2[positionTST],yHat)
  results=rbind(results,data.frame(MSE=MSE, Cor=Cor))
  }
Summary=apply(results,2,mean)
end_time <- proc.time()
Time=c(end_time[1] - start_time[1])
Time
results_all=rbind(results_all,data.frame(m=m, MSE=Summary[1],
Cor=Summary[2], Time=Time))
}
results_all
results_all_traits=rbind(results_all_traits,data.frame
(results_all))
}
results_all_traits
write.csv(results_all_traits,file="Table8.11_Final.csv")
```

Appendix 11

R code for implementing the approximate kernel method with five kernels with the Data_Toy_EYT data set (Table 8.13).

```
rm(list=ls())
library(BGLR)
library(BMTME)
load('Data_Toy_EYT.RData',verbose=TRUE)
ls()
#Phenotypic data
dat_F =Pheno_Toy_EYT
head(dat_F)
dim(dat_F)
#Marker data as Cholesky of the genomic relationship matrix
dat_M =t(chol(G_Toy_EYT))
dim(dat_M)

dat_F = transform(dat_F, GID = as.character(GID))
head(dat_F)

head(dat_F,5)
#Matrix design for markers
```

```
Pos = match(dat_F$GID,row.names(dat_M))
XM = dat_M[unique(Pos),]
dim(XM)

#######Gaussian Kernel function###########
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
          Vectorize(function(i, j) l2norm(x1[i,]-x2[,j])^2)))}

#######Polynomial Kernel###########
K.polynomial=function(x1, x2=x1, gamma=1, b=0, d=3)
 { (gamma*(as.matrix(x1)%*%t(x2))+b)^d}

#######Sigmoid Kernel###########
K.sigmoid=function(x1,x2=x1, gamma=1, b=0)
{ tanh(gamma*(as.matrix(x1)%*%t(x2))+b) }

#######Exponential Kernel###########
K.exponential=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
          Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]))))}

##########Approximate Guassian Kernel#######################
Sparse_kernel=function(m,X,name){
 m=m
 XF=X
 p=ncol(XF)
 pos_m=sample(1:nrow(XF),m)
 ####Step 1 compute K_m##############3
 X_m=XF[pos_m,]
 dim(X_m)
 if (name=="Linear") {
  K_m=X_m%*%t(X_m)/p
  ######Step 2 compute K_n_m###########
  K_n_m=XF%*%t(X_m)/p
 } else if (name=="Polynomial") {
  K_m=K.polynomial(x1=X_m,x2=X_m,gamma=1/p)
  ######Step 2 compute K_n_m###########
  K_n_m=K.polynomial(x1=XF,x2=X_m,gamma=1/p)
 } else if  (name=="Sigmoid") {
  K_m=K.sigmoid(x1=X_m,x2=X_m,gamma=1/p)
  ######Step 2 compute K_n_m###########
  K_n_m=K.sigmoid(x1=XF,x2=X_m,gamma=1/p)
 }else if  (name=="Gaussian") {
  K_m=K.radial(x1=X_m,x2=X_m,gamma=1/p)
  ######Step 2 compute K_n_m###########
  K_n_m=K.radial(x1=XF,x2=X_m,gamma=1/p)
 } else {
  K_m=K.exponential(x1=X_m,x2=X_m,gamma=1/p)
  ######Step 2 compute K_n_m###########
```

```
   K_n_m=K.exponential(x1=XF,x2=X_m,gamma=1/p)
  }

  ######Step 3 compute eigenvalue decomposition of K_m######
  EVD_K_m=eigen(K_m)
  ####Eigenvectors
  U=EVD_K_m$vectors
  ###Eigenvalues###
  S=EVD_K_m$values
  ####Square root of the inverse of eigenvalues #####
  S_0.5_Inv=sqrt(1/S)
  #####Diagonal matrix of square root of inverse of eigenvalues###
  S_mat_Inv=diag(S_0.5_Inv)
  ######Computing matrix P
  P=K_n_m%*%U%*%S_mat_Inv
  return(P)}

#Environment design matrix
XE = model.matrix(~0+Env,data=dat_F)

#####Design matrix of lines
Z_L=model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))
dim(Z_L)
############Total observations in the data set and response variable
n=dim(dat_F)[1]
y=dat_F$GY

#Number of random partitions
K=10
set.seed(1)
PT = replicate(K,sample(n,0.20*n))
####Trainig sample size of lines m that will be used for training the
model
mvec=c(round(40*0.1),round(40*0.2),round(40*0.3),round(40*0.4),
round(40*0.5),round(40*1))
mvec
kernel_name=c("Linear","Polynomial", "Sigmoid", "Gaussian",
"Exponential")
results_all_kernels=data.frame()
for (i in 1:5) {
results_all=data.frame()
for (j in 1:6){
m=mvec[j]
P_Lines=Sparse_kernel(m=m,X=XM,name=kernel_name[i])
Z_Lines_Sparse=Z_L%*%P_Lines
#####Design matrix of lines x Environment interaction
Z_LE = model.matrix(~0+Z_Lines_Sparse:Env,data=dat_F)

ETA=list(list(model='FIXED',X=XE[,-1]),list(model='BRR',
X=Z_Lines_Sparse),
    list(model='BRR',X=Z_LE))
```

```
Tab1_Metrics= data.frame(PT = 1:K,MSE = NA)
start_time <- proc.time()
for(k in 1:K) {
 Pos_tst =PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA,nIter = 1e4,burnIn = 1e3,verbose = FALSE)
 yp_ts = A$yHat
  Tab1_Metrics$MSE[k] = mean((y[Pos_tst]-yp_ts[Pos_tst])^2)
  Tab1_Metrics$Cor[k] = cor(y[Pos_tst],yp_ts[Pos_tst])
}
end_time <- proc.time()
Time=c(end_time[1] - start_time[1])
Metrics=apply(Tab1_Metrics[,-c(1)],2,mean)
results_all=rbind(results_all,data.frame(m=m, MSE=Metrics[1],
Cor=Metrics[2], Time=Time))
}
results_all_kernels=rbind(results_all_kernels,data.frame
(kernel=kernel_name[i], t(results_all)))

}
results_all_kernels
write.csv(results_all_kernels,
file="Table_8.13_results_kernels_Final.csv")
```

References

Akhiezer NI, Glazman IM (1963) Theory of linear operators in Hilbert Space (Teoriia lineikykh operatorov v Gil'bertovom prostranstve), vol 1. M. Nestell, trans. from Russian. Frederick Ungar, New York

Buil A, Brown AA, Lappalainen T, Viñuela A, Davies MN, Zheng HF, Richards JB, Glass D, Small KS, Durbin R et al (2015) Gene-gene and gene-environment interactions detected by transcriptome sequence analysis in twins. Nat Genet 47:88–91

Cho Y, Saul LK (2009) Kernel methods for deep learning. In: NIPS'09 proceedings of the 22nd international conference on neural information processing systems, pp 342–350

Cordell HJ (2002) Epistasis: what it means, what it doesn't mean, and statistical methods to detect it in humans. Hum Mol Genet 11:2463–2468

Cordell HJ (2009) Detecting gene-gene interactions that underlie human diseases. Nat Rev Genet 10:392–404

Crossa J, de los Campos G, Pérez P, Gianola D, Burgueño J, Araus JL et al (2010) Prediction of genetic values of quantitative traits in plant breeding using pedigree and molecular markers. Genetics 186:713–724. https://doi.org/10.1534/genetics.110.118521

Cuevas J, Crossa J, Soberanis V, Pérez-Elizalde S, Pérez-Rodríguez P, de los Campos G, Montesinos-López OA, Burgueño J (2016) Genomic prediction of genotype × environment interaction kernel regression models. Plant Genome 9(3):1–20

Cuevas J, Crossa J, Montesinos-López OA, Burgueño J, Pérez-Rodríguez P, de los Campos G (2017) Bayesian Genomic prediction with genotype × environment kernel models. G3 7(1):41–53

Cuevas J, Granato I, Fritsche-Neto R, Montesinos-Lopez OA, Burgueño J, Bandeira e Sousa M, Crossa J (2018) Genomic-enabled prediction kernel models with random intercepts for multi-environment trials. G3 8(4):1347–1365

Cuevas J, Montesinos-López OA, Juliana P, Guzmán C, Pérez-Rodríguez P, González-Bucio J, Burgueño J, Montesinos-López A, Crossa J (2019) Deep kernel for genomic and near infrared predictions in multi-environment breeding trials. G3 9(9):2913–2924

Cuevas J, Montesinos-Lopez OA, Martini JW, Pérez-Rodríguez P, Lillemo M, Crossa J (2020) Approximate genome-based kernels models for large data sets including main effects and interactions. Front Genet 11:567757. https://doi.org/10.3389/fgene.2020.567757

de los Campos G, Gianola D, Rosa GJ, Weigel KA, Crossa J (2010) Semi-parametric genomic-enabled prediction of genetic values using reproducing kernel Hilbert spaces methods. Genet Res (Camb) 92:295–308. https://doi.org/10.1017/S0016672310000285

Endelman JB (2011) Ridge regression and other kernels for genomic selection with R package rrBLUP. Plant Genome 4:250–255. https://doi.org/10.3835/plantgenome2011.08.0024

Gianola D, van Kaam JBCHM (2008) Reproducing kernel Hilbert spaces regression methods for genomic assisted prediction of quantitative traits. Genetics 178:2289–2303. https://doi.org/10.1534/genetics.107.084285

Gianola D, Fernando RL, Stella A (2006) Genomic-assisted prediction of genetic value with semi parametric procedures. Genetics 173:1761–1776. https://doi.org/10.1534/genetics.105.049510

Golan D, Rosset S (2014) Effective genetic-risk prediction using mixed models. Am J Hum Genet 95:383–393

Hemani G, Shakhbazov K, Westra HJ, Esko T, Henders AK, McRae AF, Yang J, Gibson G, Martin NG, Metspalu A et al (2014) Detection and replication of epistasis influencing transcription in humans. Nature 508:249–253

Henderson C (1975) Best linear unbiased estimation and prediction under a selection model. Biometrics 31(2):423–447. https://doi.org/10.2307/2529430

Kang HM, Zaitlen NA, Wade CM, Kirby A, Heckerman D, Daly MJ, Eskin E (2008) Efficient control of population structure in model organism association mapping. Genetics 178:1709–1723

Lehner B (2011) Molecular mechanisms of epistasis within and between genes. Trends Genet 27: 323–331

Lin HT, Lin CJ (2003) A study on sigmoid kernels for SVM and the training of non-PSD kernels by SMO-type methods. Neural Comput 3:1–32

Long N, Gianola D, Rosa GJ, Weigel KA, Kranis A, González- Recio, O. (2010) Radial basis function regression methods for predicting quantitative traits using SNP markers. Genet Res 92: 209–225. https://doi.org/10.1017/S0016672310000157

Mallick BK, Ghosh D, Ghosh M (2005) Bayesian classification of tumours by using gene expression data. J R Stat Soc B 67:219–234

Misztal I, Legarra A, Aguilar I (2014) Using recursion to compute the inverse of the genomic relationship matrix. J Dairy Sci 97:3943–3952

Moore JH, Williams SM (2009) Epistasis and its implications for personal genetics. Am J Hum Genet 85:309–320

Morota G, Koyama M, Rosa GJM, Weigel KA, Gianola D (2013) Predicting complex traits using a diffusion kernel on genetic markers with an application to dairy cattle and wheat data. Genet Sel Evol 45:17. https://doi.org/10.1186/1297-9686-45-17

Morota G, Boddhireddy P, Vukasinovic N, Gianola D, DeNise S (2014) Kernel-based variance component estimation and whole-genome prediction of pre-corrected phenotypes and progeny tests for dairy cow health traits. Front Genet 5:56. https://doi.org/10.3389/fgene.2014.00056

Ober U, Erbe M, Long N, Porcu E, Schlather M, Simianer H (2011) Predicting genetic values: a kernel-based best linear unbiased prediction with genomic data. Genetics 188:695–708. https://doi.org/10.1534/genetics.111.128694

Pérez-Elizalde S, Cuevas J, Pérez-Rodríguez P, Crossa J (2015) Selection of the bandwidth parameter in a Bayesian kernel regression model for genomic-enabled prediction. J Agric Biol Environ Stat 20:512–532. https://doi.org/10.1007/s13253-015-0229-y

Rassmussen CE, Williams CK (2006) Gaussian processes for machine learning. MIT Press, Cambridge, MA. ISBN 0-262-18253-X

Schrodi SJ, Mukherjee S, Shan Y, Tromp G, Sninsky JJ, Callear AP, Carter TC, Ye Z, Haines JL, Brilliant MH et al (2014) Genetic-based prediction of disease traits: prediction is very difficult, especially about the future. Front Genet 5:162

Seeger M, Williams CKI, Lawrence N (2003) Fast forward selection to speed up sparse gaussian process regression. In: Bishop C, Frey BJ (eds) Proceedings of the ninth international workshop on artificial intelligence and statistics. Society for Artificial Intelligence and Statistics

Shawe-Taylor J, Cristianini N (2004) Kernel methods for pattern analysis. University Press, Cambridge, UK

Snelson E, Ghahramani Z (2006) Local and global sparse Gaussian process approximations. In: Meilia M, Shen X (eds) Proceedings of the eleven international workshop on artificial intelligence and statistics, Society for Artificial Intelligence and Statistics. Omnipress

Titsias MK (2009) Variational learning of inducing variables in sparse Gaussian Processes. In: van Dyk D, Welling M (eds) Proceedings of the eleven international workshop on artificial intelligence and statistics, Clearwater Beach, FL, 16-18 April 2009, vol 5, pp 567–574. JMLR W&CP 5

Tusell L, Pérez-Rodríguez P, Wu SF-L, Gianola D (2013) Genome-enabled methods for predicting litter size in pigs: a comparison. Animal 7:1739–1749. https://doi.org/10.1017/S1751731113001389

VanRaden PM (2008) Efficient methods to compute genomic predictions. J Dairy Sci 91:4414–4423. https://doi.org/10.3168/jds.2007-0980

Vapnik V (1998) Statistical learning theory. Wiley, Hoboken, NJ

Wahba G (1990) Spline models for observational data. Society for Industrial and Applied Mathematics, Philadelphia

Williams CKI, Seeger M (2001) Using the Nyström method to speed up kernel machines. In: Leen TK, Diettrich TG, Tresp V (eds) Advances in neural information processing systems 13. MIT Press, Cambridge, MA, pp 682–688

Zhang Z, Dai G, Jordan MI (2011) Bayesian generalized kernel mixed models. J Mach Learn Res 12:111–139

Zuk O, Hechter E, Sunyaev SR, Lander ES (2012) The mystery of missing heritability: genetic interactions create phantom heritability. Proc Natl Acad Sci U S A 109:1193–1198