# Chapter 9
# Support Vector Machines and Support Vector Regression


## 9.1 Introduction to Support Vector Machine

The Support Vector Machine (SVM) is one of the most popular and efficient supervised statistical machine learning algorithms, which was proposed to the computer science community in the 1990s by Vapnik (1995) and is used mostly for classification problems. Its versatility is due to the fact that it can learn nonlinear decision surfaces and perform well in the presence of a large number of predictors, even with a small number of cases. This makes the SVM very appealing for tackling a wide range of problems such as speech recognition, text categorization, image recognition, face detection, faulty card detection, junk mail classification, credit rating analysis, cancer and diabetes classification, among others (Attewell et al. 2015; Byun and Lee 2002). Most of the groundwork for the SVM was laid by Vladimir Vapnik while he was working on his Ph.D. thesis in the Soviet Union in the 1960s. Then Vapnik emigrated to U.S. in 1990 to work with AT&T. In 1992, Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik suggested applying the kernel trick to maximum-margin hyperplanes to capture nonlinearities in classification problems. Finally, Cortes and Vapnik (1995) introduced the SVM to the world in its more efficient mode, and since the mid-1990s, the SVM has been a very popular topic in statistical machine learning.

The SVM method works by representing the observations (data) as points in space by mapping the original observations of different classes (categories) in such a way that they are divided by an evident gap that is as extensive as possible. The predictions of new observations are done by mapping these observations into the same space, and they are allocated to one or another category depending on which side of the gap they fall.

SVM methods are very efficient for classifying nonlinear separable patterns in part by the use of the kernel trick, explained in the previous chapter, which consists of transforming the original input information into a high-dimensional feature space by enlarging the feature space using functions of the predictors; this makes it



O. A. Montesinos López et al., *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, https://doi.org/10.1007/978-3-030-89010-0_9

possible to accommodate a nonlinear boundary between the classes, without significantly increasing the computational cost.

As mentioned above, the SVM is a type of supervised learning method, which means that it cannot be implemented when the data do not have a dependent or output variable ($y$). Also, it is important to point out that the mathematics behind the SVM has been around for a long time and is quite complex, but the popularity of this method is very recent. The popularity of this method can be attributed to three main reasons: (a) the increase in computational power, (b) ample evidence of the high prediction performance of this method, and (c) the availability of user-friendly libraries in many languages that are able to implement the SVM method. For this reason, the SVM has been implemented in many domains that range from social science to natural sciences, since it is not only used for tasks relating to the prediction of categorical variables but also for the prediction of continuous outputs.

The mathematics of the SVM was originally developed for classifying binary outputs, and for this reason, this type of application is more popular and better understood. However, there is also evidence that the SVM is doing a good job predicting continuous outputs and novelty detection. The power of the SVM can be attributed to the following facts: (a) it is a kernel-based algorithm that has a sparse solution, since the prediction of new inputs is done by evaluating the kernel function in a subset of the training data points and (b) the estimation of the model parameters corresponds to a convex optimization problem, which means that they always provide a global optimum (Bishop 2006).

First, we will study the SVM for classification and then for the prediction of continuous outputs. To understand the SVM better, it is important to understand its ancestors, i.e., the maximum margin classifier and the support vector classifier. The *maximum margin classifier* is a simple and elegant method for classifying binary outputs that assume that the classes are separable by a linear boundary. However, it is not feasible to apply this method to many data sets since it requires the strong assumption that classes are separable by a linear boundary. The *support vector classifier* is an extension of the maximum margin classifier that allows to misclassify some of the training data and thus creates a separable linear boundary with a reasonable width (margin) that is more robust to overfitting. Finally, the *support vector machine* is a generalization of the support vector classifier that classifies the observations using nonlinear boundaries by expanding the feature space with the help of kernels (James et al. 2013).

## 9.2 Hyperplane

A *hyperplane* is a subspace whose dimension (cardinality) is one less than that of its original space. This means that the hyperplane of a $p$-dimensional space has a subspace of dimension $p - 1$. In Fig. 9.1 (left), we can see a two-dimensional space whose resulting hyperplane is a line, a flat one-dimensional subspace, while in Fig. 9.1 (right) there is a three-dimensional space whose hyperplane is a plane, a flat

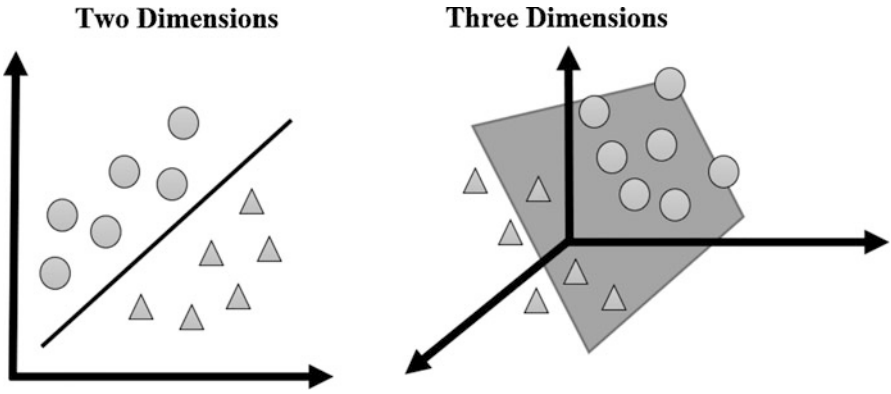



**Fig. 9.1** Hyperplanes in two (left) and three (right) dimensions

two-dimensional subspace. Although it is hard to visualize a hyperplane when the original space has a dimension of four or more, it still applies for the $(p - 1)$-dimensional flat subspace (James et al. 2013). In higher dimensions, it is useful to think of a hyperplane as a member of an affine family of $(p - 1)$-dimensional subspaces (affine spaces look and behave very similarly to linear spaces without the requirement to contain the origin), such that the whole space is partitioned into these family subspaces.

From a mathematical point of view, a hyperplane is defined as (James et al. 2013)

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 = 0 \tag{9.1}$$

for parameters $\beta_0$, $\beta_1$, $\beta_2$, and $\beta_3$. (9.1) "defines" a hyperplane, since any $X = (X_1, X_2, X_3)^{\mathrm{T}}$ for which (9.1) holds is a point in the hyperplane. Equation (9.1) is the equation of a plane, since in three dimensions, as mentioned before, a hyperplane is a plane, as can be observed in Fig. 9.1 (right).

For the $p$-dimensional space, the dimension of the hyperplane generated is $p - 1$, and it is simply an extension of (9.1) as

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p = 0 \tag{9.2}$$

In the same way, any point $X = (X_1, X_2, \ldots . X_p)^{\mathrm{T}}$ in the $p$-dimensional space that satisfies (9.2) defines a $(p - 1)$-dimensional hyperplane, which means that the hyperplane is formed by those points of $X$ that satisfy (9.2) (James et al. 2013). But those points of $X$ that do not satisfy (9.2) like, for example,

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p < 0 \tag{9.3}$$

There are points that satisfying (9.3) lie on one side of the hyperplane. Similarly, the $X$ points that correspond to

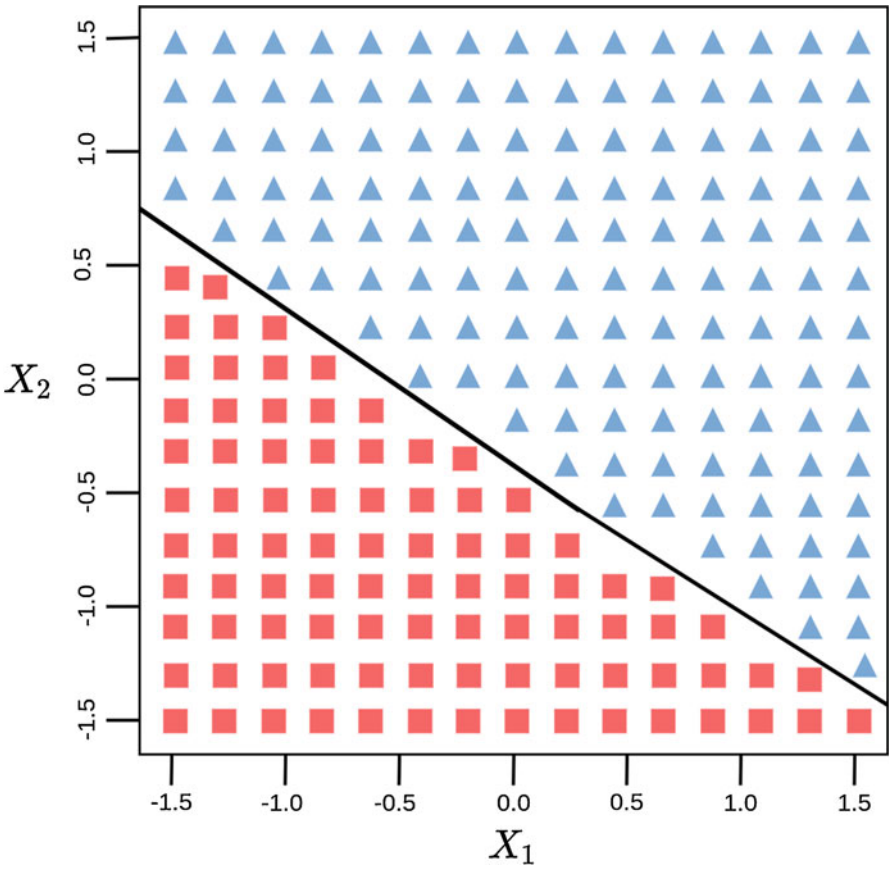



**Fig. 9.2** The hyperplane $1 + 2X_1 + 3X_2 = 0$ is shown. The blue region is the set of points for which $1 + 2X_1 + 3X_2 > 0$, and the red region is the set of points for which $1 + 2X_1 + 3X_2 < 0$ (James et al. 2013)

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p > 0 \tag{9.4}$$

will lie on the other side of the hyperplane. This means that we can think of the hyperplane as a mechanism that can divide the $p$-dimensional space into two halves. By simply calculating the sign of the left-hand side of (9.2), one can determine on which side of the hyperplane a point lies (James et al. 2013). Figure 9.2 shows a hyperplane in two-dimensional space.

## 9.3 Maximum Margin Classifier

We assume that we measure a training sample with pairs $(y_i, \boldsymbol{x}_i^{\mathrm{T}})$ for $i = 1, 2, \ldots, n$, where $y_i$ is the response variable (output) for sample $i$, and $\boldsymbol{x}_i^{\mathrm{T}} = (x_{i1}, \ldots, x_{ip})$ is a $p$-dimensional vector of predictors (inputs) measured in sample $i$. We also assume that

the response variable is binary (two classes) and coded as 1 for representing class 1 and −1 for representing class 2. A fitting function of the form

$$f(\boldsymbol{x}_i) = \beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta} \tag{9.5}$$

can be used for building a classifier based on the training data set, where $\beta_0$ is an intercept term, and $\boldsymbol{\beta}^{\mathrm{T}} = (\beta_1, \ldots, \beta_p)$ are the beta coefficients (weights) that need to be estimated to build the required classifier. Once the beta coefficients have been estimated, $\left(\widehat{\beta}_0, \widehat{\boldsymbol{\beta}}\right)$, they can be used to predict the output of a test observation that contains $\boldsymbol{x}_{i*}^{\mathrm{T}} = \left(x_{i1*}, \ldots, x_{ip*}\right)$ as a predictor. The prediction of this new test observation is labeled as 1 if $\widehat{f}(\boldsymbol{x}_{i*}) = \widehat{\beta}_0 + \boldsymbol{x}_{i*}^{\mathrm{T}}\widehat{\boldsymbol{\beta}}$ is positive, and labeled as −1 if $\widehat{f}(\boldsymbol{x}_{i*})$ is negative. $\widehat{f}(\boldsymbol{x}_{i*})$ is calculated with the estimates of the beta coefficients. Before estimating the required beta coefficients, we assume for the moment that the training data set is linearly separable in the predictor space, which means that there is at least one set of beta coefficient parameters, $(\beta_0, \boldsymbol{\beta})$, so that using the function given in (9.5), we can assume that $f(\boldsymbol{x}_i) < 0$, for observations having $y_i = -1$ and $f(\boldsymbol{x}_i) > 0$ for observations having $y_i = 1$, so that $y_i f(\boldsymbol{x}_i) > 0$ for all training observations (Bishop 2006).

Let us assume that 40 hybrids of maize were evaluated for the presence (1) or absence (−1) of a certain disease and that in addition to the output of interest, we also measured in each hybrid two predictors $(x_1, x_2)$, which could be markers linked to this disease. Figure 9.3 shows that the 40 observations can be separated by a line into

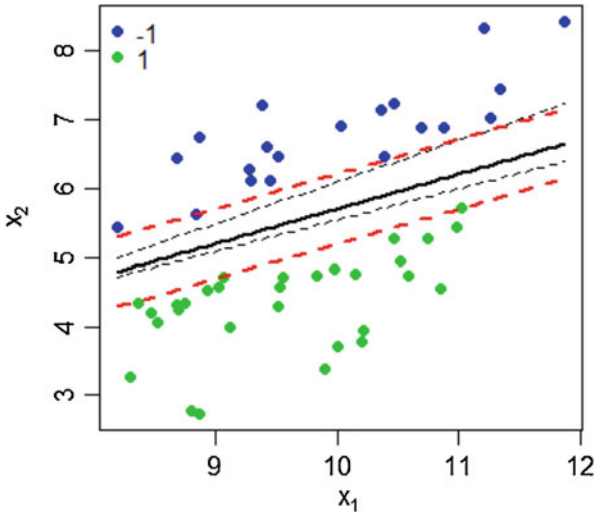



**Fig. 9.3** Synthetic linear separable data set with the hyperplane $1.3467 + 0.9927x_1 - 1.9779x_2 = 0$. The blue points correspond to the individuals with response −1 and the green points to individuals with response equal to 1. Note that the black dotted lines are two of many possible linear hyperplanes that correctly classify both classes. The black continuous line corresponds to the optimum hyperplane; the dotted red lines correspond to the maximum margin bounds and, relative to the rest, they play the biggest role in predicting new points

the two classes, that is, it is possible to construct a hyperplane that is able to perfectly classify both classes of the training data set. As can be seen in Fig. 9.3, the 1s (green points) and $-1$s (blue points) are each located in quite different areas of the two-dimensional space defined by the two predictors. For this reason, it is possible to perfectly separate the training data with a dividing line between the 1s (green points) and $-1$s (blue points). However, in Fig. 9.3, three possible dividing lines (two dotted lines and one continuous line) were used to separate the two classes perfectly, but of course the separation can be made with an infinite number of dividing lines. Therefore, the question of interest is: How to choose the dividing line in such a way that we can separate the training sample perfectly and, in addition, classify new samples with a low rate of misclassification? The answer to this question is not hard, but neither is it straightforward since there are many possible dividing lines when the pattern of the data is similar to the one shown in Fig. 9.3.

In terms of equations, this hyperplane has the property that

$$\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta} < 0 \text{ if } y_i = -1,$$

and

$$\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta} > 0 \text{ if } y_i = 1.$$

In its equivalent formulation, the hyperplane can be expressed as

$$y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) > 0$$

for all $i = 1, 2, \ldots, n$. When this separating hyperplane is found, a natural classifier is built and testing observations are classified depending on which side of the hyperplane they are located and, as mentioned above, test observation $\boldsymbol{x}_{i*}$ is used to calculate $\widehat{f}(\boldsymbol{x}_{i*}) = \widehat{\beta}_0 + \boldsymbol{x}_{i*}^{\mathrm{T}}\widehat{\boldsymbol{\beta}}$; if $\widehat{f}(\boldsymbol{x}_{i*})$ is negative, it is classified in class $-1$, but if $\widehat{f}(\boldsymbol{x}_{i*})$ is positive, it is classified in class 1. Large positive (or negative) values of $\widehat{f}(\boldsymbol{x}_{i*})$ indicate that we can be more confident about our class assignment for $\boldsymbol{x}_{i*}$, while values close to zero of $\widehat{f}(\boldsymbol{x}_{i*})$ indicate that we should be less certain of the class assignment of $\boldsymbol{x}_{i*}$. For this reason, classifiers based on a separating hyperplane require defining a linear decision boundary (James et al. 2013).

For data with patterns similar to the pattern in Fig. 9.3, the maximum margin classifier solves the problem of finding the "best" decision boundary by building two parallel lines on each side of the decision boundary and at the same distance from the decision boundary. The two lines should be as far apart as possible, taking care that any observation is within the space between them. The space between the two lines is called the margin, and it is a kind of "buffer zone" that is often also called width of the street. The preferred term is maximum margin or maximum width of the street, since intuitively it looks like it will improve the chances of correctly classifying even new observations not used for training. In other words, the strategy is to find the "street" which separates the data into two groups such that the street is as wide as

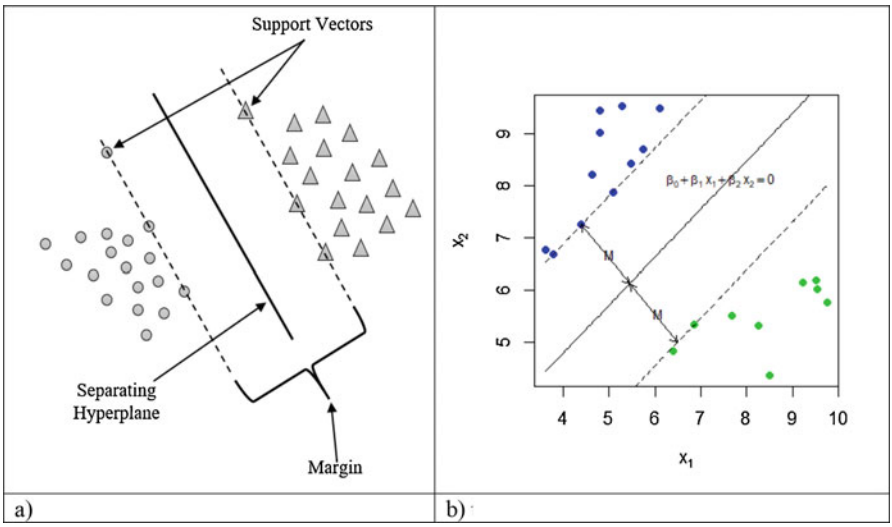



**Fig. 9.4** Maximum margin hyperplane when there are two separable classes. The maximum margin hyperplane is shown as a dashed line. The margin is the distance from the dashed line to any point on the solid line. The support vectors are the dots from each class that touch to the maximum margin hyperplane and each class must have a least one support vector. In (**a**) the two classes are circles and triangles and in (**b**) the two classes are dots in green and blue

possible, and the equation that would correspond to the "median" of this street. Our decision is made according to the position of a point relative to this median.

Figure 9.4a, b shows the margin ($M$), that is, the distance between any point and the hyperplane, while the whole width of the street is $2M$. The points touching this boundary are the support vectors (in Fig. 9.4a, the circles and triangles shown are the support vectors, while in Fig. 9.4b, they are green and blue dots) and each class must have at least one support vector. Here, the solid line maximizes the distance, so it is the best. It is possible to define the maximum margin hyperplane with only the support vectors, and for this reason, they provide a very compact way of storing a classification model, even if the number of predictors is very large.

The algorithm used to find the right support vectors relies on vector geometry and involves novel math that will be explained next.

### *9.3.1 Derivation of the Maximum Margin Classifier*

We assume that the training sample is linearly separable, that is, that there is a hyperplane that separates the training sample perfectly into two populations that can be labeled as 1s and $-1$s or as white and black or any other two labeling options. However, as pointed out before, there is an infinite number of such separating hyperplanes; for this reason, to select one reasonable hyperplane, we will choose

the hyperplane with the maximum margin ($M$). The continous line in Fig. 9.4 illustrates the best choice. Therefore, assuming that we have a training set with input information, $\boldsymbol{x}_i^{\mathrm{T}} = \left(x_1, \ldots, x_p\right)$ and with output information, $y_i \in (-1, 1)$ for $i = 1, \ldots, n$. Next, we derive the maximum margin hyperplane, which is the solution to the following optimization problem (James et al. 2013):

$$\underbrace{\text{maximize}}_{\beta_0, \beta_1, \beta_2, \ldots, \beta_p} M \tag{9.6}$$

$$\text{subject to } \sum_{j=1}^{p} \beta_j^2 = 1,$$

$$y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) \geq M, \quad i = 1, \ldots, n$$

The term $y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right)$ in the restrictions of (9.6) of this optimization problem is the distance between the $i$th observation and the decision boundary and is essential for correctly identifying classified observations on or beyond the margin boundary, given that $M$ is positive. $2M$ is the whole margin or width of the street (see Fig. 9.4b), since $M$ (half-width of the street) is the distance, centered on the decision boundary, to the margin boundary from the decision boundary. It is important to point out that the constraints given in (9.4) and (9.5) guarantee that each observation will fall on the correct side of the hyperplane and at a distance of at least $M$ from the hyperplane. The fact that the last restriction of (9.6) applies to all observations ($i = 1$, …, $n$) means that no observations are inside the street (whole margin) or fences. Hence, the goal of the maximum margin hyperplane is to find the values of the beta coefficients, $\beta_0$, $\beta_1$, $\beta_2$, …, $\beta_p$, that maximize the margin ($M$) avoiding that some observations are inside the fences (street).

To obtain the distance from a point to the hyperplane, consider point $\boldsymbol{x}$ in Fig. 9.5. Note that from any two points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ lying in hyperplane $H$, we have that $\beta_0 + \boldsymbol{x}_1^{\mathrm{T}}\boldsymbol{\beta} = 0$ and $\beta_0 + \boldsymbol{x}_2^{\mathrm{T}}\boldsymbol{\beta} = 0$, which implies that $(\boldsymbol{x}_1 - \boldsymbol{x}_2)^{\mathrm{T}}\boldsymbol{\beta} = 0$. But because $\boldsymbol{x}_1 - \boldsymbol{x}_2$ is a vector in $H$, then $\boldsymbol{\beta}$ is orthogonal to $H$, and consequently also to the normalized $\boldsymbol{\beta}$ vector, $\boldsymbol{\beta}^* = \frac{\boldsymbol{\beta}}{||\boldsymbol{\beta}||}$ (see Fig. 9.5). To solve the optimization problem (9.6), it is very important to determine the distance (margin, $M$) from point $\boldsymbol{x}$ to hyperplane $H$, which is given by the norm of the projection vector of $\boldsymbol{x} - \boldsymbol{x}_i$ on vector $\boldsymbol{\beta}^*$, where $\boldsymbol{x}_i$ is the vector formed by the intersection point of vector $\boldsymbol{\beta}^*$ and the hyperplane. Recall that the projection of $\boldsymbol{a}$ onto $\boldsymbol{b}$ is equal to $P_{\boldsymbol{b}}(\boldsymbol{a}) = \left(\frac{\boldsymbol{a}^{\mathrm{T}}\boldsymbol{b}}{\left\|\boldsymbol{b}\right\|}\right)\left(\frac{\boldsymbol{b}}{\left\|\boldsymbol{b}\right\|}\right)$. Therefore, $P_{\boldsymbol{\beta}^*}(\boldsymbol{x} - \boldsymbol{x}_i) = \left(\frac{(\boldsymbol{x} - \boldsymbol{x}_i)^{\mathrm{T}}\boldsymbol{\beta}^*}{\left\|\boldsymbol{\beta}^*\right\|^2}\right)\boldsymbol{\beta}^*$ but because $\|\boldsymbol{\beta}^*\| = 1$, then $P_{\boldsymbol{\beta}^*}(\boldsymbol{x} - \boldsymbol{x}_i) = \left((\boldsymbol{x} - \boldsymbol{x}_i)^{\mathrm{T}}\boldsymbol{\beta}^*\right)\boldsymbol{\beta}^* = \frac{\left(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{\beta} + \beta_0\right)\boldsymbol{\beta}^*}{\left\|\boldsymbol{\beta}\right\|}$. Therefore, the norm of $P_{\boldsymbol{\beta}^*}(\boldsymbol{x} - \boldsymbol{x}_i)$ is equal to the margin between the hyperplane and any of the support vectors ($\boldsymbol{M}$), which is equal to

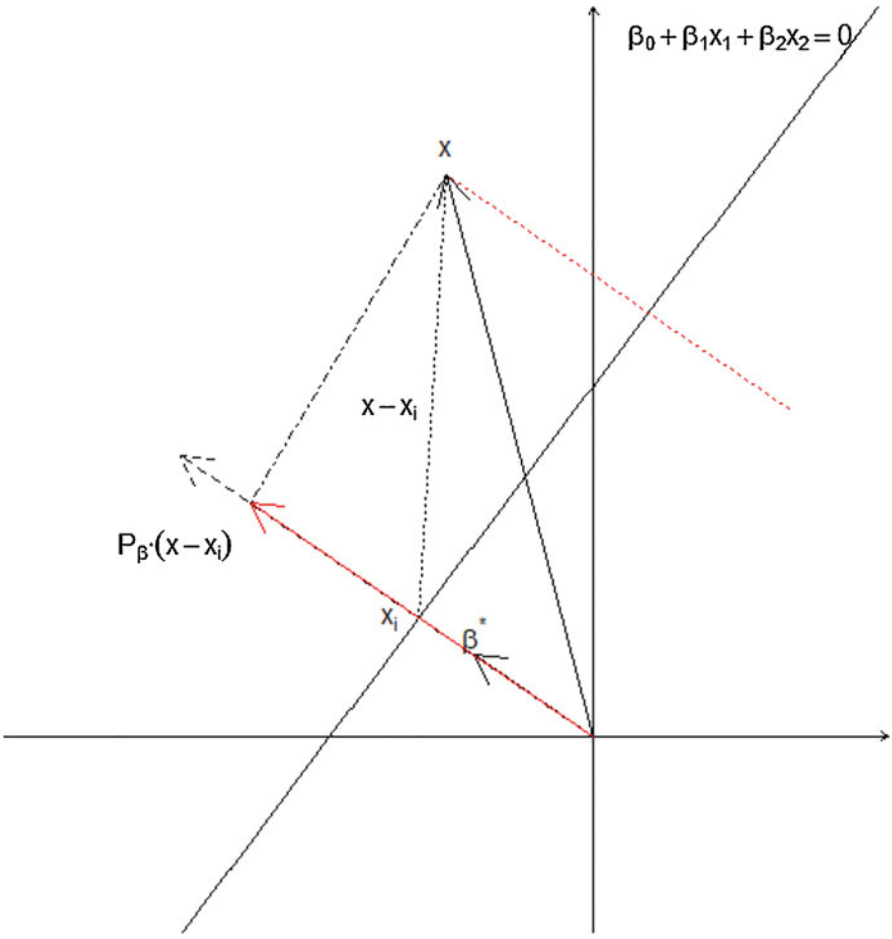



**Fig. 9.5** Distance from a point ($\boldsymbol{x}$) to a point ($\boldsymbol{x}_i$) in the hyperplane ($\beta_0 + \beta_1 x_1 + \beta_2 x_2 = 0$)

$$\boldsymbol{M} = \frac{\left|\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right| \left\|\boldsymbol{\beta}^*\right\|}{\left\|\boldsymbol{\beta}\right\|} = \frac{\left|\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right| 1}{\left\|\boldsymbol{\beta}\right\|} = 1/\|\boldsymbol{\beta}\|$$

This distance is equal to the distance (margin, $M$) from hyperplane ($h_0{=}\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta} = 0$) to hyperplane ($h_1 = \beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta} = 1$). This means that the total distance is equal to $2\boldsymbol{M} = 2/\|\boldsymbol{\beta}\|$. This implies that maximizing $\boldsymbol{M} = 1/\|\boldsymbol{\beta}\|$ subject to the constraints of (9.6) is equivalent to minimizing ($\|\boldsymbol{\beta}\|$), subject to the same constraints.

Due to the fact that $\|\boldsymbol{\beta}\|$ is naturally nonnegative and that $\frac{\|\boldsymbol{\beta}\|^2}{2}$ is monotone increasing for $\|\boldsymbol{\beta}\| \geq \mathbf{0}$, we can now reformulate the optimization problem given in (9.6) as

$$\underbrace{\text{minimize}}_{\beta_0,\beta_1,\beta_2,\ldots,\beta_p} \frac{1}{2}\|\boldsymbol{\beta}\|^2 \tag{9.7}$$

$$y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) \geq 1, \quad i = 1, \ldots, n \tag{9.8}$$

To be able to solve the optimization problem, it is important to understand the Wolfe dual result, which is explained below. Also, remember that $\frac{1}{2}\|\boldsymbol{\beta}\|^2 = \frac{1}{2}\boldsymbol{\beta}^{\mathbf{T}}\boldsymbol{\beta}$.

### *9.3.2 Wolfe Dual*

Assume we have the following general optimization problem:

$$\underbrace{\text{minimize}}_{x} f(x) \quad x \in \mathbf{R}^n \tag{9.9}$$

$$\text{subject to } h_i(x) = 0 \;\; i = 1, \ldots, n \tag{9.10}$$

$$g_i(x) \geq 0, \quad i = 1, \ldots, p \tag{9.11}$$

Assume that we are searching for the minimization value of $f(x)$ in an $n$-dimensional space with $m$ equality constraints and $p$ inequality constraints. The Wolfe dual of this optimization problem is

$$\underbrace{\text{maximize}}_{x,\lambda,\mu} f(x) - \sum\nolimits_{i=1}^{m} \lambda_i h_i(x) - \sum\nolimits_{i=1}^{p} \alpha_i g_i(x) \tag{9.12}$$

$$\text{subject to} \quad \nabla f(x) - \sum\nolimits_{i=1}^{m} \lambda_i \nabla h_i(x) - \sum\nolimits_{i=1}^{p} \alpha_i \nabla g_i(x) = 0 \tag{9.13}$$

$$\alpha_i \geq 0, \quad i = 1, \ldots, p \tag{9.14}$$

This changes the searching space to an $(n + m + p)$-dimensional space, $x$, $\lambda$, $\alpha$, with $p + 1$ constraints. The Wolfe dual is a type of Lagrange dual problem. It is important to point out that the sign of the equality constraint does not matter, and we may define it as addition or subtraction, as we wish. However, the sign of the inequality constraint is crucial and should be negative for minimization and positive for maximization.

**Illustrative Example 9.1**

$$\underbrace{\text{minimize}}_{x} x^2 \tag{9.15}$$

$$\text{subject to} \quad x \geq 1 \tag{9.16}$$

Its dual version according to Wolfe is equal to

$$\underbrace{\text{maximize}}_{x,\alpha} f(x,\alpha) = x^2 - 2\alpha(x-1) \tag{9.17}$$

$$\text{subject to} \quad \frac{\partial f(x,\alpha)}{\partial x} = 2x - 2\alpha = 0$$
$$\text{and } \alpha \geq 0 \tag{9.18}$$

Then the last version of the Wolfe dual can be simplified as

$$\underbrace{\text{maximize}}_{\alpha} L(\lambda) = -\alpha^2 + 2\alpha \tag{9.19}$$

$$\text{subject to } \alpha \geq 0 \tag{9.20}$$

With this last version of the Wolfe dual, we obtained the solution to the original optimization problem with the solution for $x = 1$ and $\alpha = 1$.

**Illustrative Example 9.2**

$$\underbrace{\text{minimize}}_{x,y} x^2 + y^2 \tag{9.21}$$

$$\textit{subject to} \qquad x + y \geq 2 \tag{9.22}$$

Its dual version according to Wolfe is equal to

$$\underbrace{\text{maximize}}_{x,y,\alpha} f(x,y,\alpha) = x^2 + y^2 - 2\alpha(x+y-2) \tag{9.23}$$

$$\text{subject to} \quad \frac{\partial f(x,y,\alpha)}{\partial x} = 2x - 2\alpha = 0$$
$$\frac{\partial f(x,y,\alpha)}{\partial y} = 2y - 2\alpha = 0 \tag{9.24}$$
$$\text{and } \alpha \geq 0$$

The last version of the Wolfe dual can be simplified by replacing $x = y = \alpha$ in the dual version, and we obtained:

$$\underbrace{\text{maximize}}_{\alpha} L(\alpha) = -2\alpha^2 + 4\alpha \tag{9.25}$$

$$\text{subject to } \alpha \geq 0 \tag{9.26}$$

With this last version of the Wolfe dual, we obtained the solution to the original optimization problem with the solution for $x = y = 1$ and $\alpha = 1$.

Now that we understand the Wolfe dual result and how to use it to obtain optimal values from optimization problems, we will solve the optimization problem given in (9.7) and (9.8). First, we present its Wolfe dual version (maximization problem), which is equal to

$$L(\boldsymbol{\beta}, \beta_0, \boldsymbol{\alpha}) = \frac{1}{2}\|\boldsymbol{\beta}\|^2 - \sum_{i=1}^{n} \alpha_i \left[ y_i \left( \beta_0 + \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{\beta} \right) - 1 \right], \tag{9.27}$$

where $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n)^{\mathrm{T}}$ and the auxiliary nonnegative variables $\alpha_i$ for $i = 1, 2, \ldots, n$ are called *Lagrange multipliers*. Setting the derivatives of $L(\boldsymbol{\beta}, \beta_0, \boldsymbol{\alpha})$ with regard to $\boldsymbol{\beta}$ and $\beta_0$ equal to zero, we obtain the following conditions:

$$\frac{\partial L(\boldsymbol{\beta}, \beta_0, \boldsymbol{\alpha})}{\partial \boldsymbol{\beta}} = \boldsymbol{\beta} - \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i = 0 \Rightarrow \boldsymbol{\beta} = \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i \tag{9.28}$$

$$\frac{\partial L(\boldsymbol{\beta}, \beta_0, \boldsymbol{\alpha})}{\partial \beta_0} = \sum_{i=1}^{n} \alpha_i y_i = 0 \Rightarrow \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{9.29}$$

$$\begin{aligned} \alpha_i \left[ y_i \left( \beta_0 + \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{\beta} \right) - 1 \right] = 0 \text{ for } i = 1, \ldots, n \Rightarrow \alpha_i \\ = 0 \text{ and } y_i \left( \beta_0 + \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{\beta} \right) = 1 \end{aligned} \tag{9.30}$$

The conditions that the solution must satisfy are called the Karush–Kuhn–Tucker conditions. They are required to ensure that the function is convex to guarantee a local optimum of nonlinear programming problems.

We can see from (9.30) that

(a) If $\alpha_i > 0$, then $y_i \left( \beta_0 + \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{\beta} \right) = 1$, or in other words, $\boldsymbol{x}_i$ is on the boundary of the slab.
(b) If $y_i \left( \beta_0 + \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{\beta} \right) > 1$, $\boldsymbol{x}_i$ is not on the boundary of the slab, and $\alpha_i = 0$.

From (9.28), we can see that the beta coefficients (with the exception of the intercept) of the maximum margin hyperplane problem are a linear combination of the training vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$. A vector $\boldsymbol{x}_i$ belongs to that expansion if, and only if, $\alpha_i \neq 0$ and these vectors are called support vectors. By condition (9.30), if $\alpha_i \neq 0$, then $y_i \left( \beta_0 + \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{\beta} \right) = 1$. Thus, support vectors lie on the marginal hyperplane $y_i \left( \beta_0 + \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{\beta} \right) = \pm 1$.

The maximum margin hyperplane is fully defined by support vectors. The definition of these hyperplanes is not affected by vectors that are not lying on the marginal hyperplanes, since in their absence, the solution for the maximum margin hyperplane remains unchanged.

By placing solutions (9.28) and (9.29) back into $L(\boldsymbol{\beta}, \beta_0, \boldsymbol{\alpha})$, we obtain the Wolfe dual simplified version (maximization) of the optimization problem:

$$L(\boldsymbol{\alpha}) = \underbrace{\frac{1}{2}\left\|\sum_{i=1}^{n}\alpha_i y_i \boldsymbol{x}_i\right\|^2 - \sum_{i=1}^{n}\alpha_i\alpha_j y_i y_j\left(\boldsymbol{x}_i.\boldsymbol{x}_j\right)}_{-0.5\times\sum_{i=1}^{n}\alpha_i\alpha_j y_i y_j\left(\boldsymbol{x}_i.\boldsymbol{x}_j\right)} - \sum_{i=1}^{n}\alpha_i y_i\beta_0 + \sum_{i=1}^{n}\alpha_i \tag{9.31}$$

Simplifying (9.31) leads to the dual optimization problem for the maximum margin classifier

$$\underbrace{\text{maximize}}_{\boldsymbol{\alpha}} L(\boldsymbol{\alpha}) = \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\sum_{i=1}^{n}\alpha_i\alpha_j y_i y_j\left(\boldsymbol{x}_i.\boldsymbol{x}_j\right) \tag{9.32}$$

$$\text{subject to : } \alpha_i \geq 0 \text{ and } \sum_{i=1}^{n}\alpha_i y_i = 0 \text{ for } i = 1, \ldots, n \tag{9.33}$$

The dual problem that needs to be maximized in (9.32) and (9.33) for the maximum margin classifier is cast entirely in terms of the training data and depends only on dot (inner) products of data vectors, $\boldsymbol{x}_i, \boldsymbol{x}_j$, and not on the vectors themselves. The operation $\boldsymbol{x}_i \cdot \boldsymbol{x}_j$ denotes the dot product of vectors $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$. This means that we do not exactly need the exact data points, but only their inner products to compute our decision boundary. What it implies is that if we want to transform our existing data into a higher dimensional data, which in many cases helps us classify better (see the image below for an example), we need not compute the exact transformation of our data, we just need the inner product of our data in that higher dimensional space.

It is important to point out that the constraints in (9.33) are affine and convex. Also, (9.32) is infinitely differentiable and its Hessian is positive semi-definite which implies that the maximization problem in (9.32) and (9.33) is equivalent to a convex optimization problem. For these reasons, the maximum margin hyperplane provides a unique solution to the separating hyperplane problem and in general does a good job of classifying the testing data due to the fact that the maximization of the margin between the two classes is optimal. Therefore, the dual optimization problem has the following two advantages: (a) there is no need to access the original data, only the dot products and (b) the number of free parameters is bound by the number of support vectors and not by the number of variables (beneficial for high-dimensional problems).

Since $L(\boldsymbol{\alpha})$ is a quadratic function of $\boldsymbol{\alpha}$, this dual optimization problem is a quadratic problem, and standard quadratic programming solvers can be used to obtain the optimal solution for the maximum margin classifier. Once the optimization problem is solved and the values of $\boldsymbol{\alpha}$ are found, we proceed to obtain $\widehat{\boldsymbol{\beta}} = \sum_{i=1}^{n}\alpha_i y_i \boldsymbol{x}_i$. Then we obtain the value of the intercept $\beta_0$ by the fact that any support vector $\boldsymbol{x}_i$ satisfies $y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) = 1$, that is,

$$y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) = y_i\left(\beta_0 + \sum_{j\in S}\alpha_j y_j\left(\boldsymbol{x}_i.\boldsymbol{x}_j\right)\right) = 1,$$

where the set of indices of the support vectors is denoted as $S$. Although we can solve this equation for the intercept $\beta_0$ using an arbitrarily chosen support sector $\boldsymbol{x}_i$, a numerically more stable solution is obtained by first multiplying by $y_i$ making use of $y_i^2 = 1$, and then averaging this equation over all support vectors and solving for $\beta_0$, which gives

$$\beta_0 = \frac{1}{N_S} \sum_{i \in S} \left( y_i - \sum_{j \in S} \alpha_j y_j (\boldsymbol{x}_i . \boldsymbol{x}_j) \right),$$

where $N_S$ is the total number of support vectors. The maximum margin classifier produces a function $\widehat{f}(\boldsymbol{x}_i) = \widehat{\beta}_0 + \boldsymbol{x}_i^{\mathrm{T}} \widehat{\boldsymbol{\beta}}$ that can be used to classify training and testing observations as

$$\widehat{y}_i = sign\left[\widehat{f}(\boldsymbol{x}_i)\right]$$

Due to the construction of this method, none of the training observations falls in the margin, but for testing observations this is not guaranteed. It is expected that the larger the margin in the training data, the better the classification for testing observations. This method is quite robust to misclassification of testing observations because its construction focuses only on the fraction of points that count (support points), and those that have $\alpha_i > 0$ for $i = 1, \ldots, n$, but of course, finding those support points requires using all the training data.

**Example 9.1 A Hand Computation of the Maximum Margin Classifier**
Let the data points and labels be as follows:

$$\boldsymbol{X} = \begin{bmatrix} 0.5 & 1 \\ -0.5 & 1 \\ -0.5 & -1 \end{bmatrix}, \ \boldsymbol{y} = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}, \ \boldsymbol{Q} = \begin{bmatrix} -0.5 & -1 \\ 0.5 & -1 \\ -0.5 & -1 \end{bmatrix}$$

The matrix $\boldsymbol{Q}$ on the right incorporates the class labels, i.e., the rows are $x_i y_i$. Then

$$\boldsymbol{Q}\boldsymbol{Q}^{\mathrm{T}} = \begin{bmatrix} 1.25 & 0.75 & 1.25 \\ 0.75 & 1.25 & 0.75 \\ 1.25 & 0.75 & 1.25 \end{bmatrix}$$

The dual optimization problem is thus

$$\underbrace{\text{maximize}}_{\boldsymbol{\alpha}} L(\boldsymbol{\alpha}) = \alpha_1 + \alpha_2 + \alpha_3 - \frac{1}{2}\begin{array}{l}\big(1.25\alpha_1^2 + 0.75\alpha_1\alpha_2 + 1.25\alpha_1\alpha_3 + 0.75\alpha_2\alpha_1 \\ +1.25\alpha_2^2 + 0.75\alpha_2\alpha_3 + 1.25\alpha_3\alpha_1 + 0.75\alpha_3\alpha_2 + 1.25\alpha_3^2\big)\end{array}$$

$$\underbrace{\text{maximize}}_{\boldsymbol{\alpha}} L(\boldsymbol{\alpha}) = \alpha_1 + \alpha_2 + \alpha_3 - \frac{1}{2} \times \big(1.25\alpha_1^2 + 1.5\alpha_1\alpha_2 + 2.5\alpha_1\alpha_3 + 1.25\alpha_2^2 + 1.5\alpha_2\alpha_3 + 1.25\alpha_3^2\big)$$

Subject to $a_1 \geq 0$, $a_2 \geq 0$, $a_3 \geq 0$ and since $\sum_{i=1}^{3} \alpha_i y_i = 0$, then $-\alpha_1 - \alpha_2 + \alpha_3 = 0$, which is equivalent to $\alpha_3 = \alpha_1 + \alpha_2$. While in practice such problems are solved by delicate quadratic optimization solvers, here we will show how to solve this toy problem by hand.

Using the equality constraint, we can eliminate one of the variables, say $\alpha_3$, and simplify the objective function to

$$\underbrace{\text{maximize}}_{\boldsymbol{\alpha}} L(\boldsymbol{\alpha}) = -\frac{1}{2}\Big(1.25\alpha_1^2 + 1.5\alpha_1\alpha_2 + 2.5\alpha_1(\alpha_1+\alpha_2) + 1.25\alpha_2^2 + 1.5\alpha_2(\alpha_1+\alpha_2) + 1.25(\alpha_1+\alpha_2)^2\Big) + 2\alpha_1 + 2\alpha_2$$

$$\underbrace{\text{maximize}}_{\boldsymbol{\alpha}} L(\boldsymbol{\alpha}) = -\frac{1}{2}\big(5\alpha_1^2 + 8\alpha_1\alpha_2 + 4\alpha_2^2\big) + 2\alpha_1 + 2\alpha_2$$

By setting partial derivatives to 0, we obtain $-5\alpha_1 - 4\alpha_2 + 2 = 0$ and $-4\alpha_1 - 4\alpha_2 + 2 = 0$ (notice that, because the objective function is quadratic, these equations are guaranteed to be linear). We therefore obtain the solution $\alpha_1 = 0$ and $\alpha_2 = \alpha_3 = 0.5$. Recall that $\widehat{\boldsymbol{\beta}} = \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i = \boldsymbol{X}^{\mathbf{T}}\boldsymbol{Z}$, where $\boldsymbol{Z}^{\mathbf{T}} = (\boldsymbol{\alpha} \circ \boldsymbol{y})^{\mathbf{T}} = [-1, -1, 1][0, 0.5, 0.5] = [0, -0.5, 0.5]$, and $\circ$ represents the cell-by-cell product between matrices or vectors.

Therefore,

$$\widehat{\boldsymbol{\beta}} = \boldsymbol{X}^{\mathrm{T}}\boldsymbol{Z} = \begin{bmatrix} 0.5 & -0.5 & -0.5 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ -0.5 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

Next, we will calculate the intercept, $\beta_0 = \frac{1}{N_S} \sum_{i \in S} \left( y_i - \sum_{j \in S} \alpha_j y_j \big(\boldsymbol{x}_i \boldsymbol{x}_j\big) \right)$, but first we calculate

$$\boldsymbol{y}^* - \boldsymbol{X}^*\boldsymbol{X}^{*\mathrm{T}}\boldsymbol{Z}^* = \begin{bmatrix} -1 \\ 1 \end{bmatrix} - \begin{bmatrix} -0.5 & 1 \\ -0.5 & -1 \end{bmatrix} \begin{bmatrix} -0.5 & -0.5 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$
$$- \begin{bmatrix} 1.25 & -0.75 \\ -0.75 & 1.25 \end{bmatrix} \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix} - \begin{bmatrix} -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$\boldsymbol{y}^*$ is equal to $\boldsymbol{y}$ but without the rows for those *Lagrange multipliers* ($\alpha_i$) that are equal to zero. $\boldsymbol{X}^*$ is equal to $\boldsymbol{X}$ but without those $\alpha_i$ that are equal to zero, and $\boldsymbol{Z}^*$ is equal to $\boldsymbol{Z}$ but without those $\alpha_i$ that are equal to zero. $N_S = 2$ since only one $\alpha_i$ is equal to zero, and this was observation 1. Therefore, $\beta_0$ is

$$\beta_0 = \frac{1}{2}(0 + 0) = 0$$

Next we calculate the $\widehat{f}(\boldsymbol{x}_i)$ values

$$\begin{bmatrix} \widehat{f}(\boldsymbol{x}_1) \\ \widehat{f}(\boldsymbol{x}_2) \\ \widehat{f}(\boldsymbol{x}_3) \end{bmatrix} = \begin{bmatrix} 0.5 & 1 \\ -0.5 & 1 \\ -0.5 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ -1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}$$

Finally, we proceed to calculate the predicted values using $\widehat{y}_i = \text{sign}[\widehat{f(\boldsymbol{x}_i)}]$, then

$$\begin{bmatrix} \widehat{y}_1 \\ \widehat{y}_2 \\ \widehat{y}_3 \end{bmatrix} = \begin{bmatrix} \text{sign}\left[\widehat{f}(\boldsymbol{x}_1)\right] \\ \text{sign}\left[\widehat{f}(\boldsymbol{x}_2)\right] \\ \text{sign}\left[\widehat{f}(\boldsymbol{x}_3)\right] \end{bmatrix} = \begin{bmatrix} \text{sign}(-1) \\ \text{sign}(-1) \\ \text{sign}(1) \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}.$$

```
#####Calculations of the hard margin classifier with library e1071####
rm(list=ls())
library(BMTME)
library(e1071)
library(caret)

#########Input data
X1=data.frame(matrix(c(0.5,-0.5,-0.5,1,1,-1), ncol=2))
y1=c(1,1,-1)
dat=data.frame(y=as.factor(y1),x1=X1[,1],x2=X1[,2])

####### Fitting the SVM model with library e1071 ######################
fm1=svm(y=as.factor(y1), x=X1, kernel="linear", scale =F)
ypred=predict(fm1,X1)
Predicted=ypred
```

```
####Useful information that we can extract###########
head(fm1$fitted) ######predicted values of the training data
head(fm1$index) #######index of support vectors
head(fm1$SV,5) ###design matrix of X of support vectors
head(c(fm1$coefs)) ####Coefs=Support vectors*y_i
fm1$rho ##### Extracting the negative value of b (intercept)
Beta=t(fm1$coefs)%*%fm1$SV ###Option 1 for computation of beta
coefficients (weights)
Beta
#####Option 2 for computing beta coefficients
Beta_Coef=t(fm1$coefs)%*%as.matrix(X1)[fm1$index,]
head(Beta_Coef)
Alphas=c(fm1$coefs)*y1[fm1$index] ### Lagrange multiplier's
coefficients
Alphas

> ####Output of implementing the svm in library
e1071#############################
> head(fm1$fitted) ######predicted values of the training data
1  1 -1
> head(fm1$index)#######index of support vectors
[1] 2 3
> head(c(fm1$coefs)) ####Coefs=Support vectors*y_i
[1]  0.5 -0.5
> fm1$rho #####Extracting the negative value of b (intercept)
[1] 0
> #Find value of Beta coefficients=weights
> Beta=t(fm1$coefs)%*%fm1$SV ## Option 1 for computation of beta
coefficients (weights)
> Beta
   X1 X2
[1,]  0  1
> #####Option 2 of beta coefficients calculation
> Beta_Coef=t(fm1$coefs)%*%as.matrix(X1)[fm1$index,]
> head(Beta_Coef)
   X1 X2
[1,]  0  1
> ####### Lagrange multiplier alpha coefficients ############
> Alphas=c(fm1$coefs)*y1[fm1$index]
> Alphas
[1] 0.5 0.5
```

From the output of the svm() function in the e1071 library, we can see that fm1$fitted produced exactly the same predictions as those we obtained with hand computation. Also, the hand computation and the index of the output of the svm() function, as fm1$index, agree that the indices of the support vector are observations 2 and 3 since observation 1 is equal to zero. The coefficients that result from the product of the Lagrange multipliers with the response variable, $\alpha_j y_j$, also agree. Also, we obtained the same intercept using hand calculation as that extracted from the fitted model with the svm() function of the e1071 library. We found agreement between the at hand computation and the results of the e1071 library for the Lagrange multipliers.

## 9.4 Derivation of the Support Vector Classifier

The method just studied does a good job when the data are linearly separable, but what can we do when the data are not linearly separable? The solution is to create a *soft margin classifier* that allows some points to fall on the incorrect side of the margin by using slack variables ($\zeta_i$). Adding slack variables to the optimization problem allows some points to be on the wrong side of the margin and, consequently, to be misclassified (James et al. 2013). In Fig. 9.6 there are two points that fall on the wrong side of the boundary line with the corresponding slack term denoted as $\zeta_i$ (James et al. 2013).

The $\zeta_i$ called slack variables are used in optimization problems to define relaxed versions of some constraints. The slack variable, $\zeta_i$, measures the distance by which vector $\boldsymbol{x}_i$ violates the established inequality, $y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) \geq 1$. For a hyperplane $y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) = 0$, an $\boldsymbol{x}_i$ vector with $\zeta_i > 0$ can be viewed as an outlier. Each $\boldsymbol{x}_i$ must be positioned on the correct side of the appropriate marginal hyperplane so as not to be considered an outlier. This implies that those vectors that fall between $0 < y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) < 1$ are correctly classified by the hyperplane $y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) = 0$ and are no longer considered outliers. By omitting the outlier observations, the training data can be classified correctly by hyperplane $y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) = 0$ with margin $M = \|\boldsymbol{\beta}\|^{-1}$, which is called the *soft margin*, as opposed to the separable case that we call the *hard margin* classifier. For this reason, the soft margin classifier is more robust to individual observations and does a better job classifying the training and testing observations. However, this method does not guarantee that every observation is on the correct side of the margin and hyperplane since it allows some observations to be on the incorrect side of the margin or hyperplane. It is from this property that this method takes its name since the margin is *soft* in the sense that it can be violated by some of the training observations. As mentioned above, an observation can be not only on the wrong side of the hyperplane but also on the incorrect side of the margin.

Then the question of interest is: how to select the hyperplane in the non-separable case? One option is to choose the hyperplane with minimum empirical error. However, this option does not guarantee that a large margin can be found, and for choosing the right hyperplane, we need to find: (a) a balance between the limit of the total amount of slack due to outliers, measured as $\sum_{i=1}^{n}\zeta_i$ and (b) a hyperplane with a large margin, but if the margin is larger, more outliers are possible, which implies a larger amount of slack. The optimization problem now consists of finding a hyperplane that is able to classify most of the training observations in the two classes; this can be accomplished by obtaining the solution to the following optimization problem:

$$\underbrace{\text{maximize}}_{\beta_0,\beta_1,\beta_2,\ldots,\ \beta_p,\zeta_1,\ldots,\zeta_n} \quad M \tag{9.34}$$

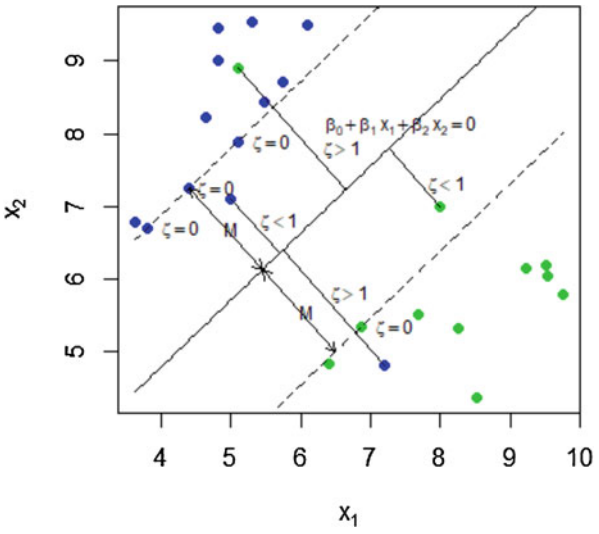



**Fig. 9.6** Soft margin support vector machine in non-separable data training. Dots with $0 \leq \zeta_i \leq 1$ or not labeled are correctly classified, while those with $\zeta_i > 1$ are on the wrong side of the decision boundary and incorrectly classified

$$\text{subject to } \sum_{j=1}^{p} \beta_j^2 = 1, \tag{9.35}$$

$$y_i \left( \beta_0 + \sum_{j=1}^{p} \beta_{j1} x_{ij} \right) \geq M(1 - \zeta_i), \tag{9.36}$$

$$\zeta_i \geq 0, \sum_{i=1}^{n} \zeta_i \leq T, \tag{9.37}$$

where $\beta_0, \beta_1, \beta_2, \ldots, \beta_p$ are the coefficients of the maximum margin hyperplane. $T$ is a nonnegative tuning parameter that determines the number and severity of the violations to the margin (and to the hyperplane) that we will tolerate, and it is seen as the total amount of errors allowed since it is the bound of the sum of $\zeta_i^{'}s$. $T$ is like a budget for the amount that the margin can be violated by the $n$ observations. For $T$ close to zero, the soft-margin SVM allows very little error and is similar to the hard-margin classifier (James et al. 2013). The larger $T$ is, the more error is allowed, which in turn allows for wider margins. These parameters play a key role in controlling the bias-variance trade-off of this statistical learning method. In practice, $T$ is a hyperparameter that needs to be tuned, for example, by using cross-validation. $M$ is the width of the margin and we seek to make this quantity as large as possible. In (9.37), $\zeta_1, \ldots, \zeta_n$ are slack (error) variables that allow individual observations to be on the wrong side of the margin or the hyperplane. The slack variable $\zeta_i$ tells us where the $i$th observation is located, relative to the hyperplane and relative to the margin. If $\zeta_i = 0$, then the $i$th observation is on the correct side of the margin. If

$\zeta_i > 0$, then the $i$th observation is on the wrong side of the margin, and this means that the $i$th observation has broken the margin. If $\zeta_i > 1$, then it is on the wrong side of the hyperplane. If $T = 0$, this implies that no budget is available for violations to the margin, and it must be that $\zeta_1 = \ldots = \zeta_n = 0$, in which case the optimization problem is equal to that of the maximum margin hyperplane. The larger the budget $T$, the wider the margin and the larger the number of support vectors, which means that we are more tolerant of violations to the margin. In contrast, the lower the $T$, the narrower the margin and fewer support vectors are selected, which means less tolerance of violations to the margin. Similar to the maximum margin classifier, only the support vectors (observations that lie on the margin) and observations that violate the margin affect the hyperplane and the resulting classifier. However, all observations that lie strictly on the correct side of the margin do not affect the support vector classifier.

Since this method is based only on a small fraction of the training observations (support vectors), it is quite robust to the classification of new observations that are far away from the hyperplane. Once we solve (9.34)–(9.37), we classify a test observation $x^*$ by simply determining on which side of the hyperplane it lies. That is, we classify the test observation in the training/testing sets based on the sign of $f(x^*) = \widehat{\beta}_0 + \widehat{\beta}_1 x_1^* + \widehat{\beta}_2 x_2^* + \ldots + \widehat{\beta}_p x_p^*$; if $f(x^*) < 0$, then the observation is assigned to the class corresponding to $-1$, but if $f(x^*) > 0$, then the observation is assigned to the class corresponding to 1 (James et al. 2013), which is exactly as in the hard margin classification method described before.

Next, we present the Wolfe primal version (for minimization) of the support vector classifier, which is equal to

$$L(\boldsymbol{\beta}, \beta_0, \boldsymbol{\epsilon}, \boldsymbol{\alpha}, \boldsymbol{\delta}) = \frac{1}{2}\|\boldsymbol{\beta}\|^2 - T\sum_{i=1}^{n}\zeta_i - \sum_{i=1}^{n}\alpha_i\left[y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) - 1 + \zeta_i\right] + \sum_{i=1}^{n}\delta_i\zeta_i, \tag{9.38}$$

where $\boldsymbol{\zeta} = (\zeta_1, \ldots, \zeta_n)^{\mathrm{T}}$, $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n)^{\mathrm{T}}$, $\delta_i > 0$ for $i = 1, \ldots, n$ associated with the nonnegativity constraints of the slack variables, $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_n)^{\mathrm{T}}$. By setting the derivatives of $L = L(\boldsymbol{\beta}, \beta_0, \boldsymbol{\zeta}, \boldsymbol{\alpha}, \boldsymbol{\delta})$ with regard to $\boldsymbol{\beta}$, $\beta_0$, and $\boldsymbol{\zeta}$ equal to zero, we obtain the following conditions:

$$\frac{\partial L}{\partial \boldsymbol{\beta}} = \boldsymbol{\beta} - \sum_{i=1}^{n}\alpha_i y_i \boldsymbol{x}_i = 0 \;\Rightarrow\; \boldsymbol{\beta} = \sum_{i=1}^{n}\alpha_i y_i \boldsymbol{x}_i \tag{9.39}$$

$$\frac{\partial L}{\partial \beta_0} = -\sum_{i=1}^{n}\alpha_i y_i = 0 \Rightarrow \quad \sum_{i=1}^{n}\alpha_i y_i = 0 \tag{9.40}$$

$$\frac{\partial L}{\partial \epsilon_i} = T - \alpha_i - \delta_i = 0 \Rightarrow \quad \alpha_i + \delta_i = T \tag{9.41}$$

$$\alpha_i\left[y_i\left(\beta_0+\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right)-1+\zeta_i\right]=0 \text{ for } i=1,\ldots,n \Rightarrow \alpha_i = 0 \text{ and } y_i\left(\beta_0+\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right)=1-\zeta_i \tag{9.42}$$

$$\delta_i\zeta_i=0 \text{ for } i=1,\ldots,n \Rightarrow \delta_i=0 \text{ and } \zeta_i=0 \tag{9.43}$$

By placing solutions (9.39)–(9.43) back into $L=L(\boldsymbol{\beta},\beta_0,\boldsymbol{\zeta},\boldsymbol{\alpha},\boldsymbol{\delta})$, we obtain the Wolfe dual version (maximization problem) of the optimization problem

$$\underbrace{\text{maximize}}_{\boldsymbol{\alpha}} L(\boldsymbol{\alpha})=\sum_{i=1}^{n}\alpha_i-\frac{1}{2}\sum_{i=1}^{n}\alpha_i\alpha_j y_i y_j\left(\boldsymbol{x}_i.\boldsymbol{x}_j\right) \tag{9.44}$$

$$\text{subject to}: \quad 0\le\alpha_i\le T \text{ and } \sum_{i=1}^{n}\alpha_i y_i=0 \text{ for } i=1,\ldots,n \tag{9.45}$$

This problem is very similar to the one in the previous section and, again, it is a convex quadratic programming problem that can be solved using conventional quadratic programming software since the objective function is concave and infinitely differentiable.

Again, the solution to $\boldsymbol{\alpha}$ in (9.44) can be used to make the predictions as follows:

$$\widehat{y}_i=\text{sign}\left(\sum_{i=1}^{N_S}\widehat{\alpha}_j y_j\left(\boldsymbol{x}_j.\boldsymbol{x}\right)+\widehat{\beta}_0\right),$$

where $N_S$ is the total number of support vectors lying on a marginal hyperplane, that is, those vectors $\boldsymbol{x}_i$ with $0\le\alpha_i\le T$ and $\widehat{\beta}_0=\frac{1}{N_S}\sum_{i\in S}\left(y_i-\sum_{j\in S}\widehat{\alpha}_j y_j\left(\boldsymbol{x}_i.\boldsymbol{x}_j\right)\right)$.

The predicted values depend only on the inner products between vectors and not directly on the vectors themselves; this fact is the key for expanding this method to define nonlinear decision boundaries.

## 9.5 Support Vector Machine

When the data are linearly inseparable in a low-dimensional space, it is possible to separate them in a higher dimensional space. For example, when all data points within a circle in a two-dimensional space belong to one class, those outside the circle belong to another class. In this case, it is not possible to use a straight line to separate the two classes, but by adding two more features, $x_1^2$, $x_2^2$, it is possible to do so, as can be seen in Fig. 9.7.

Figure 9.8 provides another example of a nonlinear problem that can be mapped to a linear problem by using a nonlinear transformation, $\varphi$, of the input data.

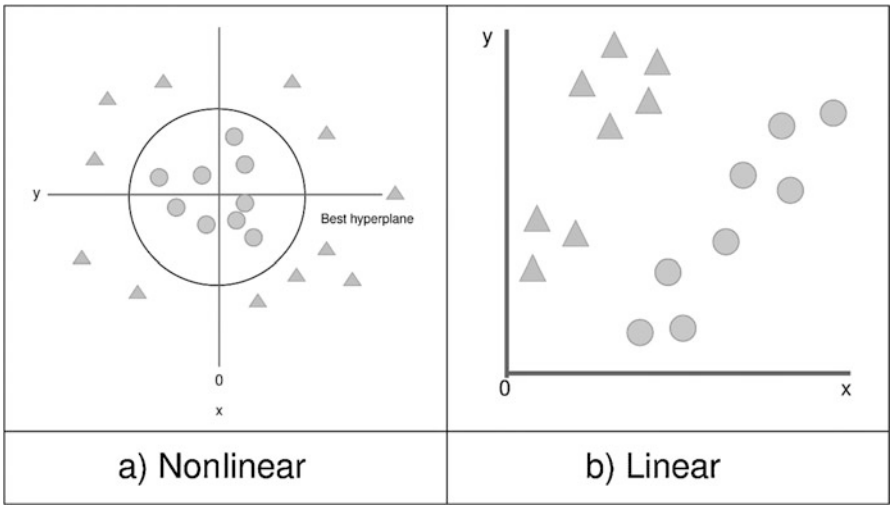



**Fig. 9.7** Transforming a nonlinear problem into a linear one

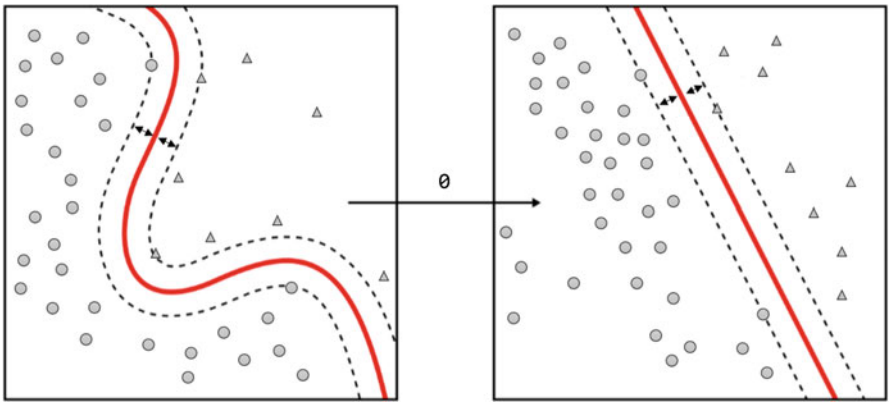
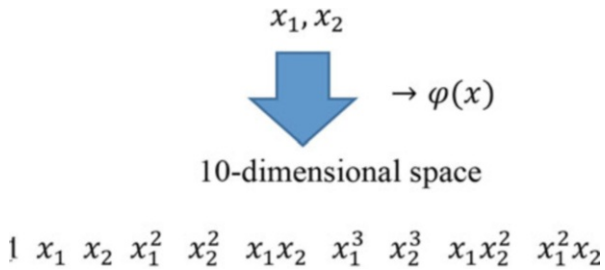



**Fig. 9.8** Transforming a complex nonlinear problem into a linear one

To see better how the transformation is implemented to expand the original input feature, we assume that we are dealing with two-dimensional data (i.e., in $\mathbb{R}^2$) and we will expand the input data using a polynomial kernel $(\boldsymbol{x}_i . \boldsymbol{x}_j + 1)^3$. The following illustration shows how this kernel maps the data.




This means that if such a linear decision surface does exist, the data are mapped into a much higher dimensional space ("feature space") where the separating decision surface is found, and the feature space is constructed via a very smart statistical projection ("kernel trick") studied in detail in the previous chapter.

This means that the construction of a higher dimensional space is done in general terms as

$$\boldsymbol{x} \rightarrow \varphi(\boldsymbol{x}).$$

That is, training input samples are transformed into a feature space using a nonlinear function $\varphi(.)$.

**Kernel functions** We define a kernel function as being a real-valued function of two arguments, $K(\boldsymbol{x}, \boldsymbol{x}^{\mathrm{T}}) \epsilon \mathbb{R}$, for $\boldsymbol{x}, \boldsymbol{x}^{\mathrm{T}} \epsilon \mathbb{R}$. The function is typically symmetric (i.e., $K(\boldsymbol{x}, \boldsymbol{x}^{\mathrm{T}}) = K(\boldsymbol{x}^{\mathrm{T}}, \boldsymbol{x})$) and nonnegative (i.e., $K(\boldsymbol{x}, \boldsymbol{x}^{\mathrm{T}}) \geq 0$), so it can be interpreted as a measure of similarity, but this is not required.

By making the following substitution, we can build an optimization problem in the new space:

$$\boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{x}_j \rightarrow \varphi(\boldsymbol{x}_i)^{\mathrm{T}} \varphi\left(\boldsymbol{x}_j\right).$$

This implies that the nonlinear *support vector machine* (SVM) is trained with the inner product $\varphi(\boldsymbol{x}_j)^{\mathrm{T}} \varphi(\boldsymbol{x}_j)$ as long as this inner product is known, which means that it does not matter if $\varphi(\boldsymbol{x}_j)$ is known. By using a kernel function, the kernel trick directly specifies the inner product:

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

Thanks to the kernel trick, the computational cost of training the SVM is independent of the dimensionality of the feature space. Some of the most popular kernels were described in the previous chapter and are: linear, polynomial, sigmoid,

Gaussian (radial), exponential, and arc-cosine (AK) with different numbers of hidden layers.

As pointed out above, the SVM kernel only needs the information of the kernel value $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$, assuming that this has been defined as was exemplified in the previous chapter. For this reason, nonvectorial patterns $\boldsymbol{x}$ such as sequences, trees, and graphs can be handled. 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.

The SVM is an extension of the *support vector classifier* when enlarging the feature space using kernels (James et al. 2013). This is possible thanks to the fact that the solution of the dual optimization problem for the support vector classifier does not directly depend on the input vectors but only on the inner products. Since positive definite symmetric (PDS) kernels implicitly define an inner product, we can extend the support vector classifier and combine it with an arbitrary PDS kernel $K$, by replacing each instance of an inner product $\boldsymbol{x}_i.\,\boldsymbol{x}_j$, with $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$. This leads to a general form of the support vector classifier that is called SVM, which is the solution to the following optimization problem:

$$\underbrace{\text{maximize}}_{\boldsymbol{\alpha}} L(\boldsymbol{\alpha}) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \alpha_i \alpha_j y_i y_j K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) \tag{9.46}$$

$$\text{subject to}: \quad 0 \leq \alpha_i \leq T \text{ and } \sum_{i=1}^{n} \alpha_i y_i = 0 \text{ for } i = 1, \ldots, n \tag{9.47}$$

Again, we classify the test observation in the training/testing sets based on the sign of $f(\boldsymbol{x}) = \sum_{i=1}^{N_S} \widehat{\alpha}_i y_i K(\boldsymbol{x}_i, \boldsymbol{x}) + \widehat{\beta}_0 = (\widehat{\boldsymbol{\alpha}}^{\circ}\boldsymbol{y})^{\mathrm{T}} K(\boldsymbol{x}_i, \boldsymbol{x}) + \widehat{\beta}_0$; if $f(\boldsymbol{x}) < 0$, then the observation is assigned to the class corresponding to $-1$, but if $f(\boldsymbol{x}) > 0$, then the observation is assigned to the class corresponding to 1 (James et al. 2013). Also, $\widehat{\beta}_0 = \frac{1}{N_S} \sum_{i \in S} \left( y_i - \sum_{j \in S} \widehat{\alpha}_j y_j K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) \right)$, where $N_S$ is the total number of support vectors lying on a marginal hyperplane and $\sum_{j \in S} \widehat{\alpha}_j y_j K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = (\widehat{\boldsymbol{\alpha}}^{\circ}\boldsymbol{y})^{\mathrm{T}} \boldsymbol{K} \boldsymbol{e}_i$, where $\boldsymbol{e}_i$ is the $i$th unit vector, therefore $\varphi(\boldsymbol{x}_i) = \boldsymbol{K}\boldsymbol{e}_i$, that is $\varphi(\boldsymbol{x}_i)$ is the $i$th column of $\boldsymbol{K}$, for $i = 1, 2, \ldots, n$. We chose $f(\boldsymbol{x})$ as a nonlinear function of $\boldsymbol{x}$ and the possible kernels are those explained above: linear, polynomial, Gaussian, or sigmoid. With the exception of the linear kernel, all these kernels are nonlinear functions of $\boldsymbol{x}$, but with fewer parameters than quadratic, cubic, or a higher order expansion of $x$.

The SVM can be implemented with the R package e1071 in the R statistical software (R Core Team 2018) with linear, polynomial, Gaussian, and sigmoid kernels. This software also allows implementing the SVM method with ordinal data under the following two approaches:

### 9.5.1 One-Versus-One Classification

When we have categorical (multi-class) data with more than two classes under the *one*-versus-*one* classification approach, we construct $K(K-1)/2$ binary SVMs, each of which compares a pair of classes. Each SVM compares the $k$th class, coded as +1, to the $k$'th class, coded as $-1$. At prediction time, a voting scheme is applied: all $K(K-1)/2$ binary SVMs are applied to an unseen sample and the class that gets the highest number of "+1" predictions gets predicted by the combined classifier (James et al. 2013).

### 9.5.2 One-Versus-All Classification

The one-versus-all approach is an alternative when there are more than two categories ($K > 2$) and consists of fitting $K$ SVMs, each time comparing one of the $K$ classes to the remaining $K-1$ classes, that is, in learning the $k$th classifier we treat all points not in class $k$ as a single not-$k$ class by lumping them all together. To learn each of the two class classifiers, we temporarily assign labels to n training points: observations in class $k$ and not-$k$ are assigned temporary labels +1 and $-1$, respectively. Having done this for all $K$ classes, we then predict the value of $y_i$ for input $\boldsymbol{x}$ by taking

$$\widehat{y}_i = \text{argmax}\left[ f(\boldsymbol{x})_k \right] \text{ for } k = 1, 2, \ldots, K,$$

where $f(\boldsymbol{x})_k = \sum_{j=1}^{N_S} \widehat{\alpha}_{jk} y_j K_k\left(\boldsymbol{x}_j, \boldsymbol{x}\right) + \widehat{\beta}_{0k}$. That is, for an $\boldsymbol{x}$ input, we classified the $i$th observation in the class for which $f(\boldsymbol{x})_k$ $k = 1, 2, \ldots, K$ is largest even if this evaluation is negative, since this indicates that we have the highest level of confidence that the test observation belongs to the $k$th class rather than to any of the other classes. In general, SVMs are very competitive when you have a large number of features (independent variables), for example, in genomic selection and in text classification. SVMs with nonlinear kernels perform quite well in most cases and are usually head to head with random forests, that is, sometimes random forests work slightly better and sometimes SVMs win. It is effective when the number of independent variables is greater than the number of observations. However, there are no free lunches and SVMs have their difficulties. They can be computationally expensive at times. SVMs do not perform well with noisy data sets. That being said, one should be careful about when to choose and when not to choose SVM as the classifier to solve the problem at hand.

**Example 9.1 for binary data** For this example, we used the EYT Toy data set 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).

First we load the data using load("Data_Toy_EYT.RData") using the following code:

```
rm(list=ls())
library(BMTME)
library(e1071)
library(caret)

load("Data_Toy_EYT.RData")
ls()

Gg=data.matrix(G_Toy_EYT)
G=Gg
```

This part of the code gives as output

```
> load("Data_Toy_EYT.RData")
> ls()
[1] "G_Toy_EYT"   "Pheno_Toy_EYT"
```

Here we can see two files: the first is the GRM and the second is the phenotypic information. Then, using data.matrix(G_Toy_EYT), we accommodate the GRM in the Gg object as a matrix.

```
> Gg=data.matrix(G_Toy_EYT)
```

With the next R code, we give a name to the phenotypic information; this is ordered as

```
Data.Final=Pheno_Toy_EYT
Data.Final=Data.Final[order(Data.Final$Env,Data.Final$GID),]
head(Data.Final)
```

The first six observations of this phenotypic information are

```
> head(Data.Final)
     GID   Env DTHD DTMT    GY Height
1 GID6569128 Bed5IR   1   1 6.119272   0
2 GID6688880 Bed5IR   2   2 5.855879   0
3 GID6688916 Bed5IR   2   2 6.434748   0
4 GID6688933 Bed5IR   2   2 6.350670   0
5 GID6688934 Bed5IR   1   2 6.523289   0
6 GID6688949 Bed5IR   1   2 5.984599   0
```

With the following code, we create the design matrices

```
########Creating the design matrix of lines ##################
Z1G=model.matrix(~0+as.factor(Data.Final$GID))
L=t(chol(Gg))
Z1G=Z1G%*%L
ZT=model.matrix(~0+as.factor(Data.Final$Env))
Z2TG=model.matrix(~0+Z1G:as.factor(Data.Final$Env))
```

Then with the next part of the code, we prepare the information to create the folds for implementing a five-fold CV strategy.

```
##########Preparation for building the five-fold CV######
Data.Final_1=Data.Final[,c(1:3)]
colnames(Data.Final_1)=c("Line","Env","Response")
Env=unique(Data.Final_1$Env)
nI=length(unique(Data.Final$Env))
nCV=5
```

Using the latter information, we created the five-folds using the CV.KFold function of the BMTME package

```
#############Training-testing partitions####################
CrossV<-CV.KFold(Data.Final_1, K =nCV, set_seed=123)
```

Then with the next code, we selected Height as the response variable, and we also got the number of rows in the data set

```
##########Selecting the Height, binary output######
y1=Data.Final$Height+1
y2=y1
n=dim(Data.Final)[1]
```

Next, we built the input data to implement the SVM method. To do this, we applied the following code, using only the information of environments and genotypic information of lines; PCCC_Part = c() is for saving the output of each testing fold.

```
##########Concatenating the information for input information####
X1=as.data.frame(cbind(ZT,Z1G))
dim(X1)
PCCC_Part=c()
```

The next code implements the five-fold:

```
for(r in 1:nCV) {
###### a) input, output, and testing set##################
X2=X1
```

```
y1=as.factor(y2)
positionTST=c(CrossV$CrossValidation_list[[r]])

###### b) Training and testing sets#######################
X_tr=droplevels(X2[-positionTST,])
X_ts=droplevels(X2[positionTST,])
y_tr=y1[-positionTST] ###Training
y_ts=y1[positionTST] ###Testing

####### c) Deleting columns with no variance##############
var_x=apply(X_tr,2,var)
length(var_x)
pos_var0=which(var_x>0)
length(pos_var0)
X_tr_New=X_tr[,pos_var0]
X_ts_New=X_ts[,pos_var0]

####### d) Fitting the model with SVM#####################
fm1=svm(y=y_tr,x=X_tr_New)
ypred=predict(fm1,X_ts_New)
Predicted=ypred
Observed=y_ts
xtab <- table(Observed, Predicted)
Conf_Matrix=confusionMatrix(xtab)

####### e) Calculating the accuracy in terms of PCCC#########
PCCC=Conf_Matrix$overall[1]
PCCC_Part=c(PCCC_Part,PCCC)
}
PCCC_Part
mean(PCCC_Part)
```

In part a) of this code, we updated in each fold the input information (X matrix), the output information (y1), and the testing set of each fold. It is important to point out that the output variable (y1), when this is binary or ordinal, it is required to define it as a factor for SVM methods. The outputs and inputs of each fold are obtained in part b) of the code. In part c) of the code, those columns of the input information with zero variance are deleted, since if they are not deleted, the SVM fails to converge. In part d) of the code, the SVM is fitted, where we only provide the training set of the input and output information. By default, the SVM implements the radial basis function, or Gaussian kernel, and also by default, the input is scaled internally with the SVM function. Further, in part d) of the code, the corresponding predictions for the testing set of each fold are obtained. In part e) of the code, the metric PCCC is calculated with the help of the caret package for each fold and saved in PCCC_Part.

Finally, the output in terms of PCCC for each fold is obtained with PCCC_Part, and the average of the five-fold in terms of PCCC is obtained with mean (PCCC_Part). The output of the implementation is given below.

```
> PCCC_Part
Accuracy Accuracy Accuracy Accuracy Accuracy
 0.68750 0.78125 0.84375 0.71875 0.68750
> mean(PCCC_Part)
[1] 0.74375
```

We can see that the highest prediction was obtained in fold 3 with PCCC = 0.84375, while the lowest was observed in folds 1 and 5 with PCCC = 0.68750. Finally, the average of the five-fold was equal to PCCC = 0.74375, which means that 74.375% of the cases were correctly classified in the testing sets. It is important to point out that this result was obtained without taking into account the genotype × environment interaction. The same code can be used taking into account the $G \times E$ by only replacing

$$\text{X1} = \text{as.data.frame}(\text{cbind}(\text{ZT}, \text{Z1G}))$$

with $G \times E$, using

$$\text{X1} = \text{as.data.frame}(\text{cbind}(\text{ZT}, \text{Z1G}, \text{Z2TG}))$$

With $G \times E$, the average PCCC = 0.5375, which is 20.625% lower than when the $G \times E$ term is ignored. It is important to point out that to fit a model with the svm() function without the $G \times E$ term, we can implement not only the Gaussian kernel (radial) but also the linear, polynomial, and sigmoid kernels, by only specifying in svm(y = y_tr,x = X_tr_New, kernel = "linear"), the required kernel as linear, polynomial, or sigmoid. The outputs using the four available kernels are given next:

```
> results
     Type  PCCC
1   radial 0.74375
2   linear 0.74375
3 polynomial 0.71250
4  sigmoid 0.74375
```

Here we can see that the PCCC for radial, linear, and sigmoid was 0.74375, and the lowest PCCC was with the polynomial kernel with a value of 0.71250, while with the $G \times E$ interaction term the PCCC were

```
> results
     Type  PCCC
1   radial 0.53750
2   linear 0.55625
3 polynomial 0.51250
4  sigmoid 0.56875
```

Next, we provide the R code for tuning the hyperparameters under the SVM method without taking into account the $G \times E$ interaction term. This code should be used after part ###c) by deleting columns with no variance ###, of the code given above inside the loop, with five-folds, but using only the information of fold $= 2$. The tuning function requires the method (in this case, an SVM method) to be tuned, and then the training and testing sets. Then we need to specify the type of kernel, which in this case was a linear kernel, and in ranges of the tune function are specified the grid of values to the cost. It is important that, for the linear kernel, only the cost parameter needs to be tuned, but for the radial kernel, the gamma parameter should also be tuned.

```
############Tuning process#####################
obj <- tune(svm, train.y=y_tr,train.x=X_tr_New,kernel="linear",
ranges = list(cost =seq(0.001,0.5,0.005)))
summary(obj)
plot(obj)

Par_cost=as.numeric(obj$best.parameters[1])
Par_cost
bestmod=obj$best.model
bestmod

#########Predictions for the testing set#####################
ypred=predict(bestmod,X_ts_New)
Predicted=ypred
Observed=y_ts
xtab <- table(Observed, Predicted)
Conf_Matrix=confusionMatrix(xtab)

PCCC=Conf_Matrix$overall[1]
PCCC
```

Part of the output of this code is given below.

```
> summary(obj)

Parameter tuning of 'svm':

- sampling method: 10-fold cross validation

- best parameters:
 cost
0.061

- best performance: 0.2423077

- Detailed performance results:
  cost    error dispersion
```

```
1  0.001 0.5698718 0.1057401
2   0.031 0.2653846 0.1101288
3   0.061 0.2423077 0.1124914
4   0.091 0.2576923 0.1033534
5   0.121 0.2980769 0.1116542
6   0.151 0.3057692 0.1029218
7   0.181 0.2903846 0.1245948
8   0.211 0.3057692 0.1259071
9   0.241 0.3057692 0.1259071
10 0.271 0.3134615 0.1333871
11 0.301 0.3134615 0.1333871
12 0.331 0.3134615 0.1333871
13 0.361 0.3134615 0.1333871
14 0.391 0.3134615 0.1333871
15 0.421 0.3134615 0.1333871
16 0.451 0.3134615 0.1333871
17 0.481 0.3217949 0.1373119
```

Here we can see that the lower error is obtained when the cost = 0.061. It is important to point out that these errors are calculated using a ten-fold cross-validation set with the training set of outer fold 2. The plot(obj) resulting from the tuning process is given below.

In Fig. 9.9, again we can see that the minimum average validation error corresponds to a cost value of 0.061, which was extracted with the R code as.numeric (obj$best.parameters[1]). Then we used bestmod = obj$best.model to extract the best model that corresponds to the model with the lower error in the validation set with a cost value of 0.061. Finally, the predictions for the outer testing set were obtained with ypred = predict(bestmod,X_ts_New), where the predictions are performed using the best model of the grid, which in this case has a cost value of 0.061. The resulting prediction in terms of PCCC was 0.71875.

It is important to point out that in the case of nonlinear hyperplanes, a gamma parameter also needs to be tuned. It is expected that the higher the gamma value, the

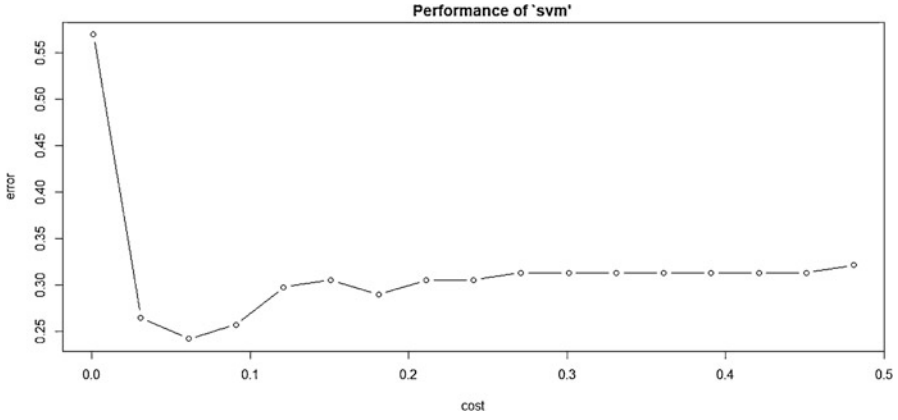



**Fig. 9.9** Average validation error of ten-fold cross-validation for the grid of cost values

better the fit to the training data set; for this reason, many times increasing the gamma parameter leads to overfitting. Now we implemented the SVM by tuning the gamma values and the cost of nonlinear kernels and only the cost for the linear hyperplane (the code we used is in Appendix 1), also ignoring the $G \times E$ interaction term. The results obtained for each type of kernel are given below.

```
> results
   Type       MSE
1  linear     0.74375
2  radial     0.74375
3  polynomial 0.74375
4  sigmoid    0.71875
```

We can see that tuning the parameters did not improve the prediction performance more than when using the default values for these hyperparameters. This means that many times the default values do a good job and that choosing the right values for the tuning process is challenging.

**Example 9.2 for ordinal data** Once again, we used the EYT Toy data set composed of 40 lines, four environments (Bed5IR, EHT, Flat5IR, and LHT), and four response variables: DTHD, DTMT, GY, and Height. But now we worked with the ordinary response variable (DTHD) that has three response options.

Since the data set was the same, the code used for its implementation was the same, but now we worked with the DTHD categorical variable. This means that the key modification was that now we used the DTHD as the response variable, which was chosen using the following code:

```
##########Selecting the DTHD, ordinal output######
y1=Data.Final$DTHD
y2=y1
n=dim(Data.Final)[1]
```

Also, by using five-folds without tuning and ignoring the $G \times E$ interaction term for the four types of kernels, we got the following results:

```
> results
   Type       PCCC
1  radial     0.76875
2  linear     0.65625
3  polynomial 0.76250
4  sigmoid    0.73125
```

Here the best predictions were obtained under the Gaussian or radial kernel with PCCC = 0.76875 and the worst under the linear kernel with PCCC = 0.65625. The complete code for reproducing these results is given in Appendix 2. Next, we

provide the performance also with five-fold cross-validation but tuning the gamma parameters and cost for nonlinear kernels, and only the cost for those with linear kernels.

```
> results
   Type        PCCC
1  linear      0.65625
2  radial      0.76875
3  polynomial 0.76250
4  sigmoid    0.74375
```

Once again, the best predictions were observed under the Gaussian kernel and the worst under the linear kernel; however, there was no improvement when using the default values for gamma and cost. In this example, we saw that implementation of the SVM method for binary or ordinal data with the e1071 library is the same, but taking care that the response variable is defined as a factor. However, this library works for binary data by default, since the machinery for SVM (derivations explained above) was designed for binary data. In the case of ordinal data (multi-class classification), by default, library e1071 implements the "*one*-vs-*one*" approach explained above, where $k(k-1)/2$ binary classifiers are trained and the appropriate class is found by a voting scheme.

## 9.6 Support Vector Regression

The support vector regression (SVR) is inspired by the support vector machine algorithm for binary response variables. The main idea of the algorithm consists of only using residuals smaller in absolute value than some constant (called $\varepsilon$-sensitivity), that is, fitting a tube of $\varepsilon$ width to the data, as illustrated in Fig. 9.10.

Two sets of points are defined as in binary classification: those falling inside the tube, which are $\varepsilon$-close to the predicted function and thus not penalized, and those falling outside, which are penalized based on their distance from the predicted function, in a way that is similar to the penalization used by SVMs in classification. Due to the fact that the idea behind support vector regression (SVR) is very similar to SVM, which consists of finding a well-fitting hyperplane in a kernel-induced feature space that will have good generalization performance using the original features. For this reason, detailed SVR theory is not covered in this book, but interested readers can find this information in the following references: Burges (1998); Awad and Khanna (2015). Also, there is no agreement that the performance of SVR is better compared to any type of regression machines for predicting continuous outcomes. For this reason, next we will illustrate the implementation of SVR in the e1071 library.

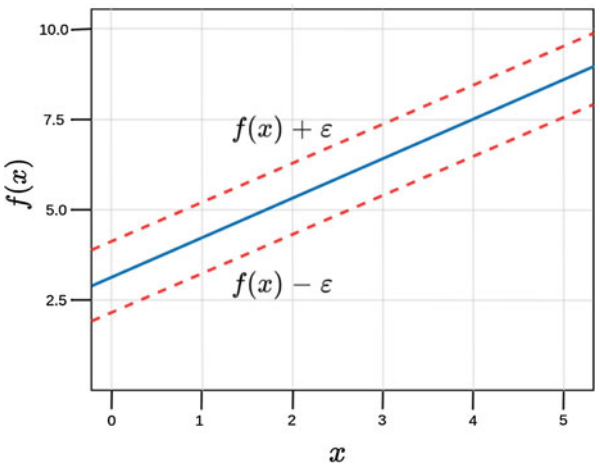



**Fig. 9.10** ϵ-insensitive regression band. The solid blue line represents the estimated regression curve $f(x)$

**Example 9.3 for continuous data** Once again, we used the EYT Toy data set composed of 40 lines, four environments (Bed5IR, EHT, Flat5IR, and LHT), and four response variables, DTHD, DTMT, GY, and Height, but now we work with the GY variable, which is continuous.

Since the data set is the same, all the codes used for its implementation are the same, but now we work with the continuous GY variable. This continuous response variable was chosen using the following code:

```
##########Selecting the GY, continuous response variable######
y1=Data.Final$GY
y2=y1
n=dim(Data.Final)[1]
```

It is important to point out that the code for implementing SVR is exactly the same as that used to implement SVM, but with the difference that here it is not necessary to put the response variable (outcome) as a factor, since now the response variable is continuous. The code used now without $G \times E$ interaction is given in Appendix 3, but the only difference between this code and the code given in Appendix 2 is the following:

```
X2=X1
actual_CV=r
y1=y2
positionTST=c(CrossV$CrossValidation_list[[r]])
```

By using five-folds without tuning and ignoring the $G \times E$ interaction term for the five types of kernels, we got the following results:

```
> results
    Type       MSE
1   linear     0.3403227
2   radial     1.0868555
3   polynomial 2.8076426
4   sigmoid    0.4211818
```

Here we see that the best predictions were obtained with the linear kernel, the second best with the sigmoid kernel, and the worst with the polynomial kernel. When the $G \times E$ interaction term is taken into account, the prediction performance in terms of MSE is equal to

```
> results
    Type        MSE
1   linear      1.366566e+01
2   radial      2.052131e+00
3   polynomial  6.675209e+06
4   sigmoid     2.296421e+00
```

In general, taking into account the $G \times E$ interaction term produced a worse prediction performance. But now the best predictions were under the radial kernel and the worst under the polynomial kernel. These results were also obtained using Appendix 3, but with X1 = as.data.frame(cbind(ZT,Z1G,Z2TG)).

Finally, this chapter provides the fundamentals of support vector machines which were studied in considerable detail and in such a way that the user understands the basis of this powerful method. We provided many examples applied for genomic predictions that illustrated how to fit SVM methods for binary, ordinal, and continuous outcomes with and without genotype × environment interaction. We also provided the components needed to build some kernels manually, which is the key for capturing nonlinearities of the input data.

## Appendix 1

Tuning process for different types of kernels ignoring the $G \times E$ interaction term with a binary response variable denoted as Height.

```
rm(list=ls())
library(BMTME)
library(plyr)
library(tidyr)
library(dplyr)
library(e1071)
library(caret)
##################Loading the data#################################
load("Data_Toy_EYT.RData")
ls()
```

```
Gg=data.matrix(G_Toy_EYT)
Data.Final=Pheno_Toy_EYT

##########Ordering the data#####################################
Data.Final=Data.Final[order(Data.Final$Env,Data.Final$GID),]

########Creating the design matrix of lines ##################
Z1G=model.matrix(~0+as.factor(Data.Final$GID))
L=t(chol(Gg))
Z1G=Z1G%*%L
ZT=model.matrix(~0+as.factor(Data.Final$Env))
Z2TG=model.matrix(~0+Z1G:as.factor(Data.Final$Env))

############Preparation for training-testing sets############
Data.Final_1=Data.Final[,c(1:3)]
colnames(Data.Final_1)=c("Line","Env","Response")
Env=unique(Data.Final_1$Env)
nI=length(unique(Data.Final$Env))

#############Training-testing partitions#####################
nCV=5
CrossV<-CV.KFold(Data.Final_1, K =nCV, set_seed=123)

Y=Data.Final[,3:ncol(Data.Final)]
y1=Y$Height+1
y2=y1
n=dim(Y)[1]

##########Concatenating the information for input information####
X1=as.data.frame(cbind(ZT,Z1G))
dim(X1)

Pred_all_traits<-data.frame()

results<-data.frame() #save cross-validation results
Type=list("linear","radial","polynomial","sigmoid")
for (i in 1:4){
PCCC_Part=c()
for(r in 1:nCV) {
 ##### a) input, output, and testing set###################
X2=X1
actual_CV=r
y1=as.factor(y2)
positionTST=c(CrossV$CrossValidation_list[[r]])

###### b) Training and testing sets#######################
X_tr=droplevels(X2[-positionTST,])
X_ts=droplevels(X2[positionTST,])
y_tr=y1[-positionTST] ###Training
y_ts=y1[positionTST]
```

```
####### c) Deleting columns with no variance###############
var_x=apply(X_tr,2,var)
length(var_x)
pos_var0=which(var_x>0)
length(pos_var0)
X_tr_New=X_tr[,pos_var0]
X_ts_New=X_ts[,pos_var0]

####### d) Grid and tuning process#############################
Nobs=nrow(X_tr_New)
Ncols=ncol(X_tr_New)
Ncols_2=Ncols-10

gamma_values=seq(1/(Ncols-3),1/(Ncols-20),1/(10*Ncols))

obj<-tune(svm,train.y=y_tr,train.x=X_tr_New, kernel=Type[[i]],
ranges=list(gamma=gamma_values, cost =seq(1.3, 2, 0.05)),tunecontrol
= tune.control(sampling = "fix"))

Par_gamma=as.numeric(obj$best.parameters[1])

Par_cost=as.numeric(obj$best.parameters[2])

Best.model=obj$best.model

##########e) predictions with the best model#####################
ypred=predict(Best.model,X_ts_New)
Predicted=ypred
Observed=y_ts
xtab <- table(Observed, Predicted)
Conf_Matrix=confusionMatrix(xtab)

#######Calculating the accuracy in terms of PCCC#########
PCCC=Conf_Matrix$overall[1]
PCCC_Part=c(PCCC_Part,PCCC)
}
PCCC_Part

results=rbind(results,data.frame(Type=Type[[i]], PCCC=mean
(PCCC_Part)))
}
results
```

## Appendix 2

Training SVM models for different types of kernels ignoring the $G \times E$ interaction term, without tuning, with the ordinal response variable DTHD with three classes.

```
rm(list=ls())
library(BMTME)
library(plyr)
library(tidyr)
library(dplyr)
library(e1071)
library(caret)

########Loading the data#########################################
load("Data_Toy_EYT.RData")
ls()
Gg=data.matrix(G_Toy_EYT)
Data.Final=Pheno_Toy_EYT

########Ordering the data#######################################
Data.Final=Data.Final[order(Data.Final$Env,Data.Final$GID),]

########Creating the design matrix of Lines ###################
Z1G=model.matrix(~0+as.factor(Data.Final$GID))
L=t(chol(Gg))
Z1G=Z1G%*%L
ZT=model.matrix(~0+as.factor(Data.Final$Env))
Z2TG=model.matrix(~0+Z1G:as.factor(Data.Final$Env))

######Preparation for building training-testing sets############
Data.Final_1=Data.Final[,c(1:3)]
colnames(Data.Final_1)=c("Line","Env","Response")
Env=unique(Data.Final_1$Env)
nI=length(unique(Data.Final$Env))

#############Training-testing partitions#####################
nCV=5
CrossV<-CV.KFold(Data.Final_1, K =nCV, set_seed=123)

##########Selecting the DTHD, ordinal output######
Y=Data.Final[,3:ncol(Data.Final)]
y1=Y$DTHD
y1
y2=y1
n=dim(Y)[1]

##########Concatenating the information for input information####
X1=as.data.frame(cbind(ZT,Z1G))
dim(X1)

Pred_all_traits<-data.frame()

results<-data.frame() #save cross-validation results
Type=list("radial","linear","polynomial","sigmoid")
```

```
for (i in 1:4){
 PCCC_Part=c()
for(r in 1:nCV) {
###### a) input, output, and testing set###################
X2=X1
actual_CV=r
y1=as.factor(y2)
positionTST=c(CrossV$CrossValidation_list[[r]])

###### b) Training and testing sets#######################
X_tr=droplevels(X2[-positionTST,])
X_ts=droplevels(X2[positionTST,])
y_tr=y1[-positionTST] ###Training
y_ts=y1[positionTST]

####### c) Deleting columns with no variance##############
var_x=apply(X_tr,2,var)
length(var_x)
pos_var0=which(var_x>0)
length(pos_var0)
X_tr_New=X_tr[,pos_var0]
X_ts_New=X_ts[,pos_var0]

####### d) Fitting the model with SVM######################
fm1=svm(y=y_tr,x=X_tr_New,kernel=Type[[i]])
ypred=predict(fm1,X_ts_New)
Predicted=ypred
Observed=y_ts
xtab <- table(Observed, Predicted)
Conf_Matrix=confusionMatrix(xtab)

####### e) Calculating the accuracy in terms of PCCC#########
PCCC=Conf_Matrix$overall[1]
PCCC_Part=c(PCCC_Part,PCCC)
}
PCCC_Part

results=rbind(results,data.frame(Type=Type[[i]], PCCC=mean
(PCCC_Part)))
}
results
```

## Appendix 3

Training SVR models for different types of kernels ignoring the $G \times E$ interaction term, without tuning, with the continuous response variable GY.

```
rm(list=ls())
library(BMTME)
```

```
library(plyr)
library(tidyr)
library(dplyr)
library(e1071)

load("Data_Toy_EYT.RData")
ls()
Gg=data.matrix(G_Toy_EYT)
G=Gg

Data.Final=Pheno_Toy_EYT
Data.Final=Data.Final[order(Data.Final$Env,Data.Final$GID),]

########Creating the design matrix of lines ##################
Z1G=model.matrix(~0+as.factor(Data.Final$GID))
L=t(chol(Gg))
Z1G=Z1G%*%L
ZT=model.matrix(~0+as.factor(Data.Final$Env))
Z2TG=model.matrix(~0+Z1G:as.factor(Data.Final$Env))
nCV=5

Data.Final_1=Data.Final[,c(1:3)]
colnames(Data.Final_1)=c("Line","Env","Response")
Env=unique(Data.Final_1$Env)
nI=length(unique(Data.Final$Env))

##############Training-testing partitions#####################
CrossV<-CV.KFold(Data.Final_1, K =nCV, set_seed=123)

Y=Data.Final[,3:ncol(Data.Final)]
head(Y)
y1=Y$GY
y2=y1
n=dim(Y)[1]
##########Joining the information for input information####
X1=as.data.frame(cbind(ZT,Z1G))
dim(X1)

Pred_all_traits<-data.frame()

results<-data.frame() #save cross-validation results
Type=list("linear","radial","polynomial","sigmoid")
for (i in 1:4){
MSE_Part=c()
for(r in 1:nCV) {
#r=1
X2=X1
actual_CV=r
y1=y2
#y1=as.factor(y2)
positionTST=c(CrossV$CrossValidation_list[[r]])
```

```
###############Training and testing sets#################
X_tr=droplevels(X2[-positionTST,])
X_ts=droplevels(X2[positionTST,])
y_tr=y1[-positionTST] ###Training
y_ts=y1[positionTST]

##########Deleting columns with no variance###############
var_x=apply(X_tr,2,var)
length(var_x)
pos_var0=which(var_x>0)
length(pos_var0)
X_tr_New=X_tr[,pos_var0]
X_ts_New=X_ts[,pos_var0]

##########Fitting the model with SVM#######################
fm1=svm(y=y_tr,x=X_tr_New,kernel=Type[[i]])
ypred=predict(fm1,X_ts_New)

Predicted=as.numeric(ypred)
Observed=as.numeric(y_ts)

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

results=rbind(results,data.frame(Type=Type[[i]], MSE=mean
(MSE_Part)))
}
results
```

## References

Attewell P, Monaghan DB, Kwong D (2015) Data mining for the social sciences: an introduction. University of California Press, Oakland

Awad M, Khanna R (2015) Efficient learning machines—theories, concepts, and applications for engineers and system designers. Apress Open

Bishop CM (2006) Pattern recognition and machine learning. Springer Science + Business Media, LCC, New York

Burges CJ (1998) A tutorial on support vector machines for pattern recognition. Data Min Knowl Disc 2(2):121–167

Byun H, Lee SW (2002) Applications of support vector machines for pattern recognition: a survey. In: SVM '02 proceedings of the first international workshop on pattern recognition with support vector machines. Springer, London, pp 213–236

Cortes C, Vapnik V (1995) Support-vector network. Mach Learn 20:125
James G, Witten D, Hastie T, Tibshirani R (2013) An introduction to statistical learning: with applications in R. Springer, New York
R Core Team (2018) R: a language and environment for statistical computing. R Foundation for Statistical Computing, Vienna. ISBN 3-900051-07-0. http://www.R-project.org/
Vapnik V (1995) The nature of statistical learning theory. Springer, New York