Chapter 7
Nonlinear Regression Models

The previous chapter discussed regression models that were intrinsically linear. Many of these models can be adapted to nonlinear trends in the data by manually adding model terms (e.g., squared terms). However, to do this, one must know the specific nature of the nonlinearity in the data.

There are numerous regression models that are inherently nonlinear in nature. When using these models, the exact form of the nonlinearity does not need to be known explicitly or specified prior to model training. This chapter looks at several models: neural networks, multivariate adaptive regression splines (MARS), support vector machines (SVMs), and K-nearest neighbors (KNNs). Tree-based models are also nonlinear. Due to their popularity and use in ensemble models, we have devoted the next chapter to those methods.

7.1 Neural Networks

Neural networks (Bishop 1995; Ripley 1996; Titterington 2010) are powerful nonlinear regression techniques inspired by theories about how the brain works. Like partial least squares, the outcome is modeled by an intermediary set of unobserved variables (called *hidden variables* or *hidden units* here). These hidden units are linear combinations of the original predictors, but, unlike PLS models, they are not estimated in a hierarchical fashion (Fig. 7.1).

As previously stated, each hidden unit is a linear combination of some or all of the predictor variables. However, this linear combination is typically transformed by a nonlinear function $g(\cdot)$, such as the logistic (i.e., sigmoidal) function:

$$h_k(\mathbf{x}) = g\left(\beta_{0k} + \sum_{i=1}^{P} x_j \beta_{jk}\right), \quad \text{where}$$
$$g(u) = \frac{1}{1+e^{-u}}.$$

DOI 10.1007/978-1-4614-6849-3_7,

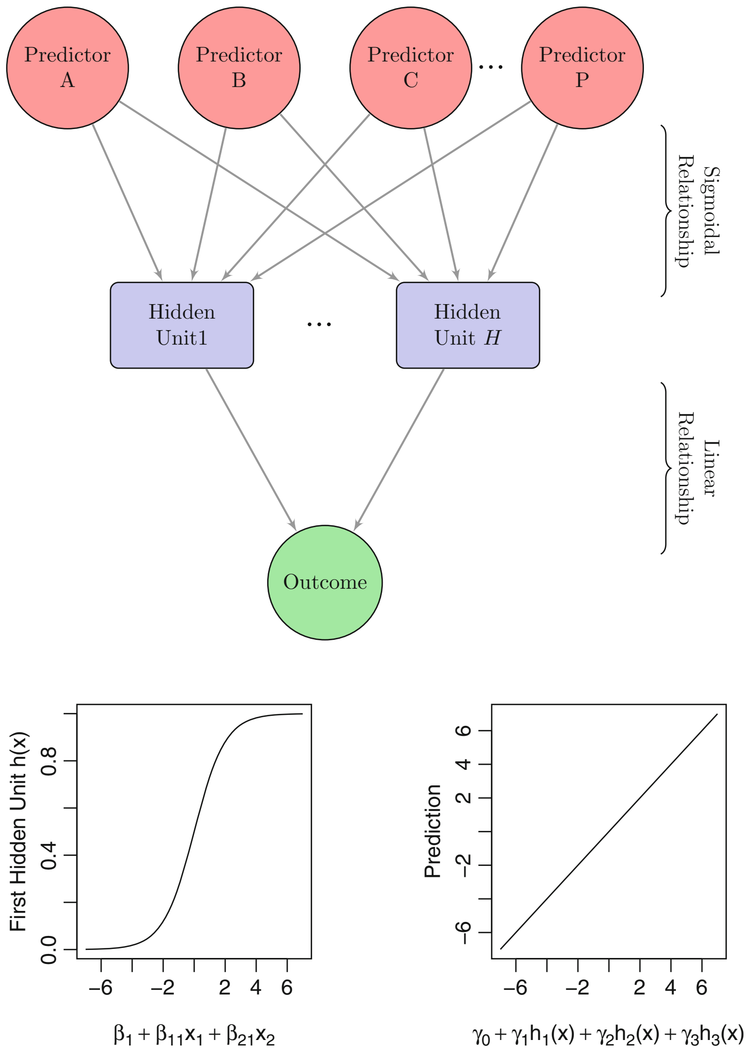

Fig. 7.1: A diagram of a neural network with a single hidden layer. The hidden units are linear combinations of the predictors that have been transformed by a sigmoidal function. The output is modeled by a linear combination of the hidden units

The β coefficients are similar to regression coefficients; coefficient β_{jk} is the effect of the jth predictor on the kth hidden unit. A neural network model usually involves multiple hidden units to model the outcome. Note that, unlike the linear combinations in PLS, there are no constraints that help define these linear combinations. Because of this, there is little likelihood that the coefficients in each unit represent some coherent piece of information.

Once the number of hidden units is defined, each unit must be related to the outcome. Another linear combination connects the hidden units to the outcome:

$$f(\mathbf{x}) = \gamma_0 + \sum_{k=1}^{H} \gamma_k h_k.$$

For this type of network model and P predictors, there are a total of $H(P+1)+H+1$ total parameters being estimated, which quickly becomes large as P increases. For the solubility data, recall that there are 228 predictors. A neural network model with three hidden units would estimate 691 parameters while a model with five hidden units would have 1,151 coefficients.

Treating this model as a nonlinear regression model, the parameters are usually optimized to minimize the sum of the squared residuals. This can be a challenging numerical optimization problem (recall that there are no constraints on the parameters of this complex nonlinear model). The parameters are usually initialized to random values and then specialized algorithms for solving the equations are used. The back-propagation algorithm (Rumelhart et al. 1986) is a highly efficient methodology that works with derivatives to find the optimal parameters. However, it is common that a solution to this equation is not a *global* solution, meaning that we cannot guarantee that the resulting set of parameters are uniformly better than any other set.

Also, neural networks have a tendency to over-fit the relationship between the predictors and the response due to the large number of regression coefficients. To combat this issue, several different approaches have been proposed. First, the iterative algorithms for solving for the regression equations can be prematurely halted (Wang and Venkatesh 1984). This approach is referred to as *early stopping* and would stop the optimization procedure when some estimate of the error rate starts to increase (instead of some numerical tolerance to indicate that the parameter estimates or error rate are stable). However, there are obvious issues with this procedure. First, how do we estimate the model error? The apparent error rate can be highly optimistic (as discussed in Sect. 4.1) and further splitting of the training set can be problematic. Also, since the measured error rate has some amount of uncertainty associated with it, how can we tell if it is truly increasing?

Another approach to moderating over-fitting is to use *weight decay*, a penalization method to *regularize* the model similar to ridge regression discussed in the last chapter. Here, we add a penalty for large regression coefficients so that any large value must have a significant effect on the model errors to be tolerated. Formally, the optimization produced would try to minimize a alternative version of the sum of the squared errors:

$$\sum_{i=1}^{n} (y_i - f_i(x))^2 + \lambda \sum_{k=1}^{H} \sum_{j=0}^{P} \beta_{jk}^2 + \lambda \sum_{k=0}^{H} \gamma_k^2$$

for a given value of λ. As the regularization value increases, the fitted model becomes more smooth and less likely to over-fit the training set. Of course, the value of this parameter must be specified and, along with the number of hidden units, is a tuning parameter for the model. Reasonable values of λ range between 0 and 0.1. Also note that since the regression coefficients are being summed, they should be on the same scale; hence the predictors should be centered and scaled prior to modeling.

The structure of the model described here is the simplest neural network architecture: a single-layer feed-forward network. There are many other kinds, such as models where there are more than one layer of hidden units (i.e., there is a layer of hidden units that models the other hidden units). Also, other model architectures have loops going both directions between layers. Practitioners of these models may also remove specific connections between objects to further optimize the model. There have also been several Bayesian approaches to neural networks (Neal 1996). The Bayesian framework outlined in Neal (1996) for these models automatically incorporates regularization and automatic feature selection. This approach to neural networks is very powerful, but the computational aspects of the model become even more formidable. A model very similar to neural networks is self-organizing maps (Kohonen 1995). This model can be used as an unsupervised, exploratory technique or in a supervised fashion for prediction (Melssen et al. 2006).

Given the challenge of estimating a large number of parameters, the fitted model finds parameter estimates that are locally optimal; that is, the algorithm converges, but the resulting parameter estimates are unlikely to be the globally optimal estimates. Very often, different locally optimal solutions can produce models that are very different but have nearly equivalent performance. This model instability can sometimes hinder this model. As an alternative, several models can be created using different starting values and averaging the results of these model to produce a more stable prediction (Perrone and Cooper 1993; Ripley 1995; Tumer and Ghosh 1996). Such *model averaging* often has a significantly positive effect on neural networks.

These models are often adversely affected by high correlation among the predictor variables (since they use gradients to optimize the model parameters). Two approaches for mitigating this issue is to pre-filter the predictors to remove the predictors that are associated with high correlations. Alternatively a feature extraction technique, such as principal component analysis, can be used prior to modeling to eliminate correlations. One positive side effect of both these approaches is that fewer model terms need to be optimized, thus improving computation time.

For the solubility data, model averaged neural networks were used. Three different weight decay values were evaluated ($\lambda = 0.00, 0.01, 0.10$) along with a single hidden layer with sizes ranging between 1 and 13 hidden units. The final predictions are the averages of five different neural networks created using different initial parameter values. The cross-validated RMSE profiles of these models are displayed in Fig. 7.2. Increasing the amount of weight decay

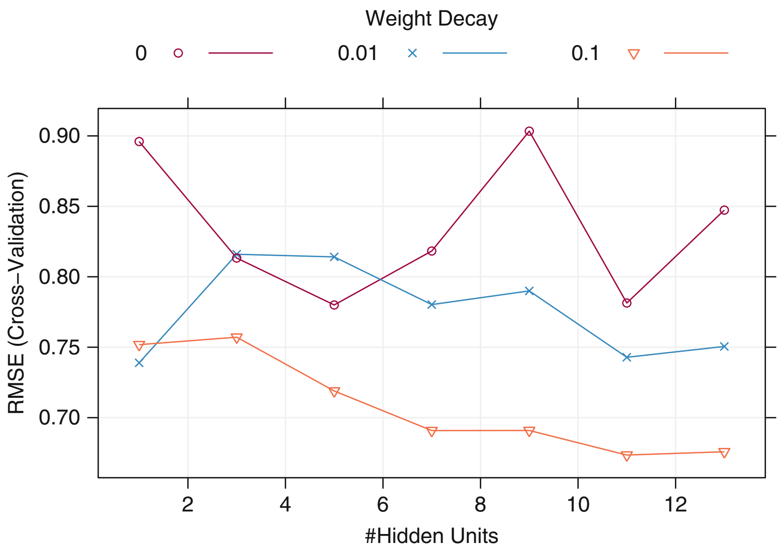

Fig. 7.2: RMSE profiles for the neural network model. The optimal model used $\lambda = 0.1$ and 11 hidden units

clearly improved model performance, while more hidden units also reduce the model error. The optimal model used 11 hidden units with a total of 2,531 coefficients. The performance of the model is fairly stable for a high degree of regularization (i.e., $\lambda = 0.1$), so smaller models could also be effective for these data.

7.2 Multivariate Adaptive Regression Splines

Like neural networks and partial least squares, MARS (Friedman 1991) uses surrogate features instead of the original predictors. However, whereas PLS and neural networks are based on linear combinations of the predictors, MARS creates two contrasted versions of a predictor to enter the model. Also, the surrogate features in MARS are usually a function of only one or two predictors at a time. The nature of the MARS features breaks the predictor into two groups and models linear relationships between the predictor and the outcome in each group. Specifically, given a cut point for a predictor, two new features are "hinge" or "hockey stick" functions of the original (see Fig. 7.3). The "left-hand" feature has values of zero greater than the cut point, while the second feature is zero less than the cut point. The new features are added to a basic linear regression model to estimate the slopes and intercepts. In effect, this scheme creates a *piecewise linear model* where each new feature models an isolated portion of the original data.

How was the cut point determined? Each data point for each predictor is evaluated as a candidate cut point by creating a linear regression model with the candidate features, and the corresponding model error is calculated. The predictor/cut point combination that achieves the smallest error is then used for the model. The nature of the predictor transformation makes such a large number of linear regressions computationally feasible. In some MARS implementations, including the one used here, the utility of simple linear terms for each predictor (i.e., no hinge function) is also evaluated.

After the initial model is created with the first two features, the model conducts another exhaustive search to find the next set of features that, given the initial set, yield the best model fit. This process continues until a stopping point is reached (which can be set by the user).

In the initial search for features in the solubility data, a cut point of 5.9 for molecular weight had the smallest error rate. The resulting artificial predictors are shown in the top two panels of Fig. 7.3. One predictor has all values less than the cut point set to zero and values greater than the cut point are left unchanged. The second feature is the mirror image of the first. Instead of the original data, these two new predictors are used to predict the outcome in a linear regression model. The bottom panel of Fig. 7.3 shows the result of the linear regression with the two new features and the piecewise nature of the relationship. The "left-hand" feature is associated with a negative slope when the molecular weight is less than 5.9 while the "right-hand" feature estimates a positive slope for larger values of the predictor.

Mathematically, the hinge function for new features can be written as

$$h(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{7.1}$$

A pair of hinge functions is usually written as $h(x - a)$ and $h(a - x)$. The first is nonzero when $x > a$, while the second is nonzero when $x < a$. Note that when this is true the value of the function is actually $-x$. For the MARS model shown in Fig. 7.3, the actual model equation would be

$$-5 + 2.1 \times h(MolWeight - 5.94516) + 3 \times h(5.94516 - MolWeight).$$

The first term in this equation (-5) is the intercept. The second term is associated with the right-hand feature shown in Fig. 7.3, while the third term is associated with the left-hand feature.

Table 7.1 shows the first few steps of the feature generation phase (prior to pruning). The features were entered into the linear regression model from top to bottom. Here the binary fingerprint descriptor enters the model as a plain linear term (splitting a binary variable would be nonsensical). The generalized cross-validation (GCV) column shows the estimated RMSE for

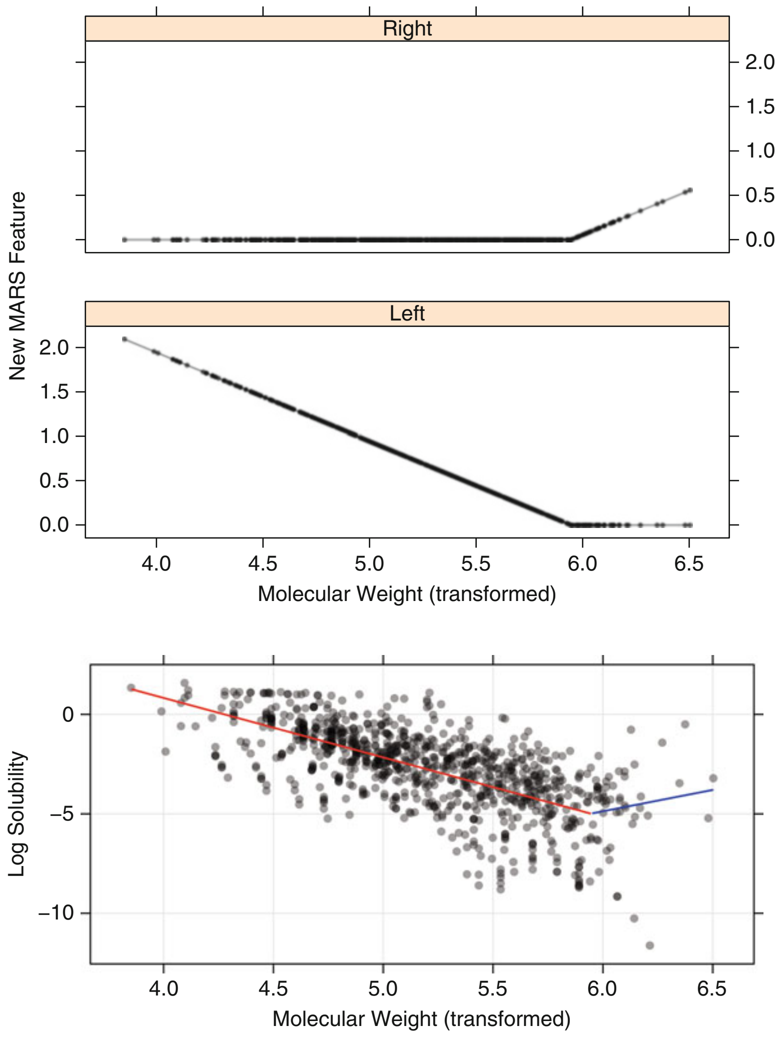

Fig. 7.3: An example of the features used by MARS for the solubility data. After finding a cut point of 5.9 for molecular weight, two new features are created and used in a linear regression model. The *top* two panels show the relationship between the original predictor and the two resulting features. The *bottom* panel shows the predicted relationship when using these two features in a linear regression model. The *red line* indicates the contribution of the "left-hand" hinge function while the *blue line* is associated with the other feature

Table 7.1: The results of several iterations of the MARS algorithm prior to pruning

Predictor	Type	Cut	RMSE	Coefficient
Intercept			4.193	−9.33
MolWeight	Right	5.95	2.351	−3.23
MolWeight	Left	5.95	1.148	0.66
SurfaceArea1	Right	1.96	0.935	0.19
SurfaceArea1	Left	1.96	0.861	−0.66
NumNonHAtoms	Right	3.00	0.803	−7.51
NumNonHAtoms	Left	3.00	0.761	8.53
FP137	Linear		0.727	1.24
NumOxygen	Right	1.39	0.701	2.22
NumOxygen	Left	1.39	0.683	−0.43
NumNonHBonds	Right	2.58	0.670	2.21
NumNonHBonds	Left	2.58	0.662	−3.29

The root mean squared error was estimated using the GCV statistic

the model containing terms on the current row and all rows above. Prior to pruning, each pair of hinge functions is kept in the model despite the slight reduction in the estimated RMSE.

Once the full set of features has been created, the algorithm sequentially removes individual features that do not contribute significantly to the model equation. This "pruning" procedure assesses each predictor variable and estimates how much the error rate was decreased by including it in the model. This process does not proceed backwards along the path that the features were added; some features deemed important at the beginning of the process may be removed while features added towards the end might be retained. To determine the contribution of each feature to the model, the *GCV* statistic is used. This value is a computational shortcut for linear regression models that produces an error value that approximates leave-one-out cross-validation (Golub et al. 1979). GCV produces better estimates than the apparent error rate for determining the importance of each feature in the model. The number of terms to remove can be manually set or treated as a tuning parameter and determined using some other form of resampling.

The process above is a description of an additive MARS model where each surrogate feature involves a single predictor. However, MARS can build models where the features involve multiple predictors at once. With a *second-degree* MARS model, the algorithm would conduct the same search of a single term that improves the model and, after creating the initial pair of features, would instigate another search to create new cuts to couple with each of the original features. Suppose the pair of hinge functions are denoted as A and B.

The search procedure attempts to find hinge functions C and D that, when multiplied by A, result in an improvement in the model; in other words, the model would have terms for A, $A \times B$ and $A \times C$. The same procedure would occur for feature B. Note that the algorithm will not add additional terms if the model is not improved by their addition. Also, the pruning procedure may eliminate the additional terms. For MARS models that can include two or more terms at a time, we have observed occasional instabilities in the model predictions where a few sample predictions are wildly inaccurate (perhaps an order of magnitude off of the true value). This problem has not been observed with additive MARS models.

To summarize, there are two tuning parameters associated with the MARS model: the degree of the features that are added to the model and the number of retained terms. The latter parameter can be automatically determined using the default pruning procedure (using GCV), set by the user or determined using an external resampling technique. For our analysis of the solubility data, we used 10-fold cross-validation to characterize model performance over first- and second-order models and 37 values for the number of model terms, ranging from 2 to 38. The resulting performance profile is shown in Fig. 7.4. There appears to be very little difference in the first- and second-degree models in terms of RMSE.

The cross-validation procedure picked a second-degree model with 38 terms. However, because the profiles of the first- and second-order model are almost identical, the more parsimonious first-order model was chosen as the final model. This model used 38 terms but was a function of only 30 predictors (out of a possible 228).

Cross-validation estimated the RMSE to be 0.7 log units and the R^2 to be 0.887. Recall that the MARS procedure internally uses GCV to estimate model performance. Using GCV, the RMSE was estimated to be 0.4 log units and an R^2 of 0.908. Using the test set of 316 samples, the RMSE was determined to be 0.7 with a corresponding R^2 of 0.879. Clearly, the GCV estimates are more encouraging than those obtained by the cross-validation procedure or the test set. However, note that the internal GCV estimate that MARS employs evaluates an individual model while the external cross-validation procedure is exposed to the variation in the entire model building process, including feature selection. Since the GCV estimate does not reflect the uncertainty from feature selection, it suffers from *selection bias* (Ambroise and McLachlan 2002). This phenomenon will be discussed more in Chap. 19.

There are several advantages to using MARS. First, the model automatically conducts feature selection; the model equation is independent of predictor variables that are not involved with any of the final model features. This point cannot be underrated. Given a large number of predictors seen in many problem domains, MARS potentially thins the predictor set using the same algorithm that builds the model. In this way, the feature selection routine has a direct connection to functional performance. The second advantage is interpretability. Each hinge feature is responsible for modeling a specific

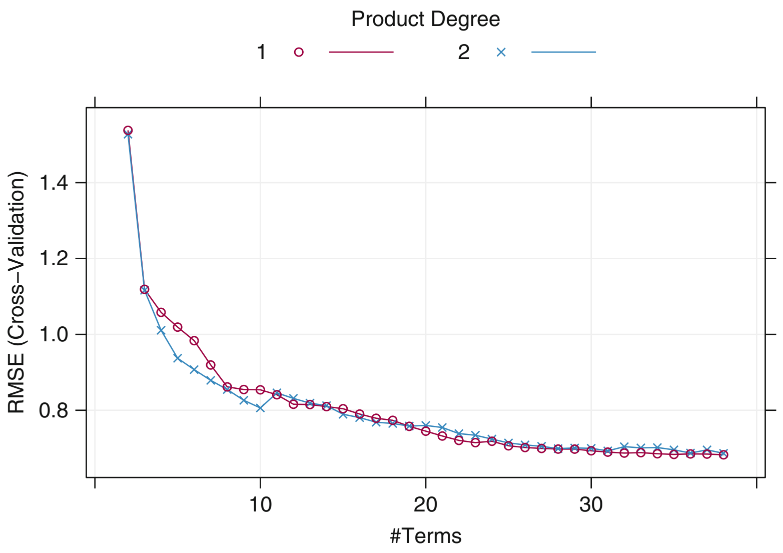

Fig. 7.4: RMSE profiles for the MARS model. The cross-validation procedure picked a second-degree model with 38 terms, although there is little difference between the first- and second-degree models. Given this equivalence, the more simplistic first-order model was chosen as the final model

region in the predictor space using a (piecewise) linear model. When the MARS model is additive, the contribution of each predictor can be isolated without the need to consider the others. This can be used to provide clear interpretations of how each predictor relates to the outcome. For nonadditive models, the interpretive power of the model is not reduced. Consider a second-degree feature involving two predictors. Since each hinge function is split into two regions, three of the four possible regions will be zero and offer no contribution to the model. Because of this, the effect of the two factors can be further isolated, making the interpretation as simple as the additive model. Finally, the MARS model requires very little pre-processing of the data; data transformations and the filtering of predictors are not needed. For example, a zero variance predictor will never be chosen for a split since it offers no possible predictive information. Correlated predictors do not drastically affect model performance, but they can complicate model interpretation. Suppose the training set contained two predictors that were nearly perfectly correlated. Since MARS can select a predictor more than once during the iterations, the choice of which predictor is used in the feature is essentially random. In this case, the model interpretation is hampered by two redundant pieces of information that show up in different parts of the model under different names.

Another method to help understand the nature of how the predictors affect the model is to quantify their *importance* to the model. For MARS, one technique for doing this is to track the reduction in the root mean squared error (as measured using the GCV statistic) that occurs when adding a particular feature to the model. This reduction is attributed to the original predictor(s) associated with the feature. These improvements in the model can be aggregated for each predictor as a relative measure of the impact on the model. As seen in Table 7.1, there is a drop in the RMSE from 4.19 to 1.15 (a reduction of 3.04) after the two molecular weight features were added to the model. After this, adding terms for the first surface area predictor decreases the error by 0.29. Given these numbers, it would appear that the molecular weight predictor is more important to the model than the first surface area predictor. This process is repeated for every predictor used in the model. Predictors that were not used in any feature have an importance of zero. For the solubility model, the predictors `MolWeight`, `NumNonHAtoms`, and `SurfaceArea2` appear to be have the greatest influence on the MARS model (see the Computing section at the end of the chapter for more details).

Figure 7.5 illustrates the interpretability of the additive MARS model with the continuous predictors. For each panel, the line represents the prediction profile for that variable when all the others are held constant at their mean level. The additive nature of the model allows each predictor to be viewed in isolation; changing the values of the other predictor variables will not alter the shape of the profile, only the location on the y-axis where the profile starts.

7.3 Support Vector Machines

SVMs are a class of powerful, highly flexible modeling techniques. The theory behind SVMs was originally developed in the context of classification models. Later, in Chap. 13, the motivation for this technique is discussed in its more natural form. For regression, we follow Smola (1996) and Drucker et al. (1997) and motivate this technique in the framework of *robust regression* where we seek to minimize the effect of outliers on the regression equations. Also, there are several flavors of support vector regression and we focus on one particular technique called *ϵ-insensitive regression.*

Recall that linear regression seeks to find parameter estimates that minimize SSE (Sect. 6.2). One drawback of minimizing SSE is that the parameter estimates can be influenced by just one observation that falls far from the overall trend in the data. When data may contain influential observations, an alternative minimization metric that is less sensitive, such as the Huber function, can be used to find the best parameter estimates. This function uses the squared residuals when they are "small" and uses the absolute residuals when the residuals are large. See Fig. 6.6 on p. 110 for an illustration.

SVMs for regression use a function similar to the Huber function, with an important difference. Given a threshold set by the user (denoted as ϵ),

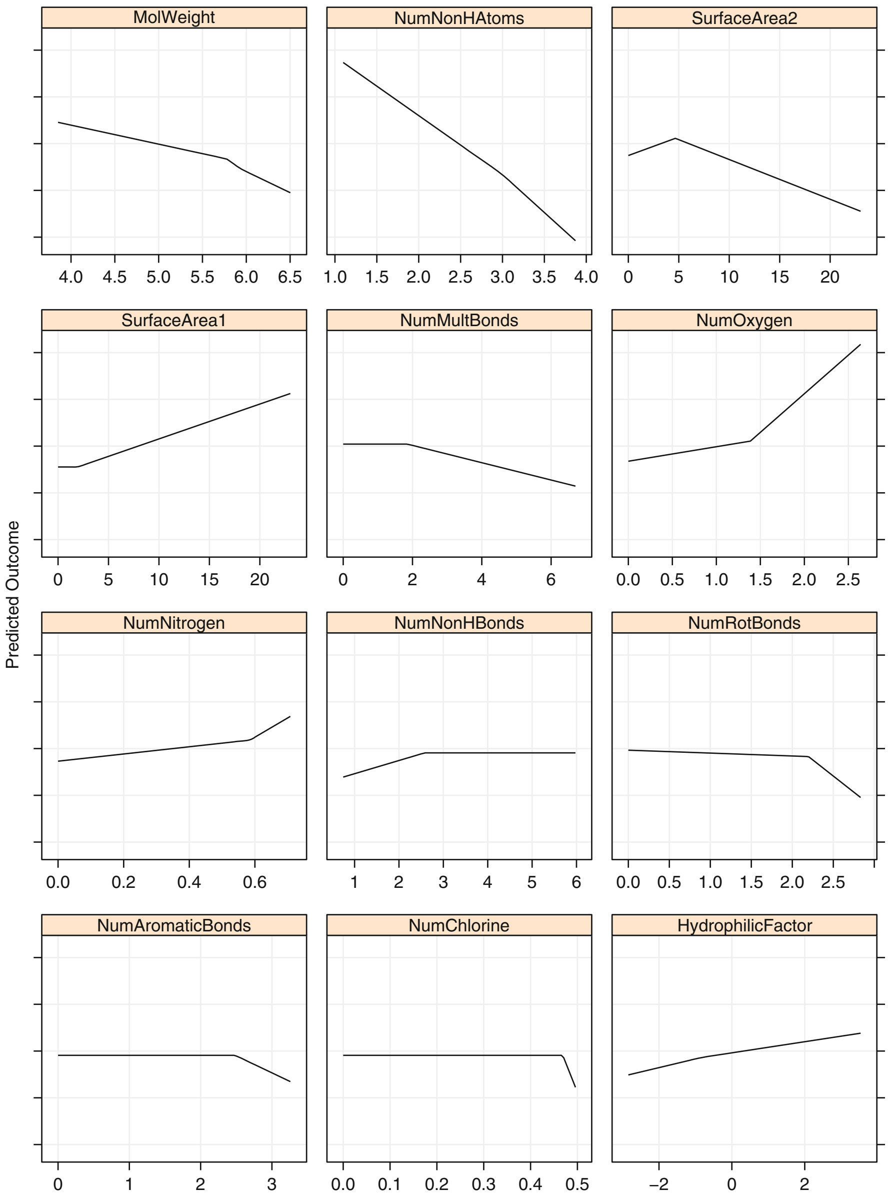

Fig. 7.5: The predicted relationship between the outcome and the continuous predictors using the MARS model (holding all other predictors at their mean value). The additive nature of the model allows each predictor to be viewed in isolation. Note that the final predicted values are the summation of each individual profile. The panels are ordered from top to bottom by their importance to the model

data points with residuals within the threshold do not contribute to the regression fit while data points with an absolute difference greater than the threshold contribute a linear-scale amount. There are several consequences to this approach. First, since the squared residuals are not used, large outliers have a limited effect on the regression equation. Second, samples that the model fits well (i.e., the residuals are small) have *no* effect on the regression equation. In fact, if the threshold is set to a relatively large value, then the outliers are the only points that define the regression line! This is somewhat counterintuitive: the poorly predicted points define the line. However, this approach has been shown to be very effective in defining the model.

To estimate the model parameters, SVM uses the ϵ loss function shown in Fig. 7.6 but also adds a penalty. The SVM regression coefficients minimize

$$Cost \sum_{i=1}^{n} L_\epsilon(y_i - \hat{y}_i) + \sum_{j=1}^{P} \beta_j^2,$$

where $L_\epsilon(\cdot)$ is the ϵ-insensitive function. The $Cost$ parameter is the *cost* penalty that is set by the user, which penalizes large residuals.[1]

Recall that the simple linear regression model predicted new samples using linear combinations of the data and parameters. For a new sample, u, the prediction equation is

$$\begin{aligned}\hat{y} &= \beta_0 + \beta_1 u_1 + \ldots + \beta_P u_P \\ &= \beta_0 + \sum_{j=1}^{P} \beta_j u_j\end{aligned}$$

The linear support vector machine prediction function is very similar. The parameter estimates can be written as functions of a set of unknown parameters (α_i) and the training set data points so that

$$\begin{aligned}\hat{y} &= \beta_0 + \beta_1 u_1 + \ldots + \beta_P u_P \\ &= \beta_0 + \sum_{j=1}^{P} \beta_j u_j \\ &= \beta_0 + \sum_{j=1}^{P}\sum_{i=1}^{n} \alpha_i x_{ij} u_j \\ &= \beta_0 + \sum_{i=1}^{n} \alpha_i \left(\sum_{j=1}^{P} x_{ij} u_j\right). \end{aligned} \tag{7.2}$$

[1] The penalty here is written as the reverse of ridge regression or weight decay in neural networks since it is attached to residuals and not the parameters.

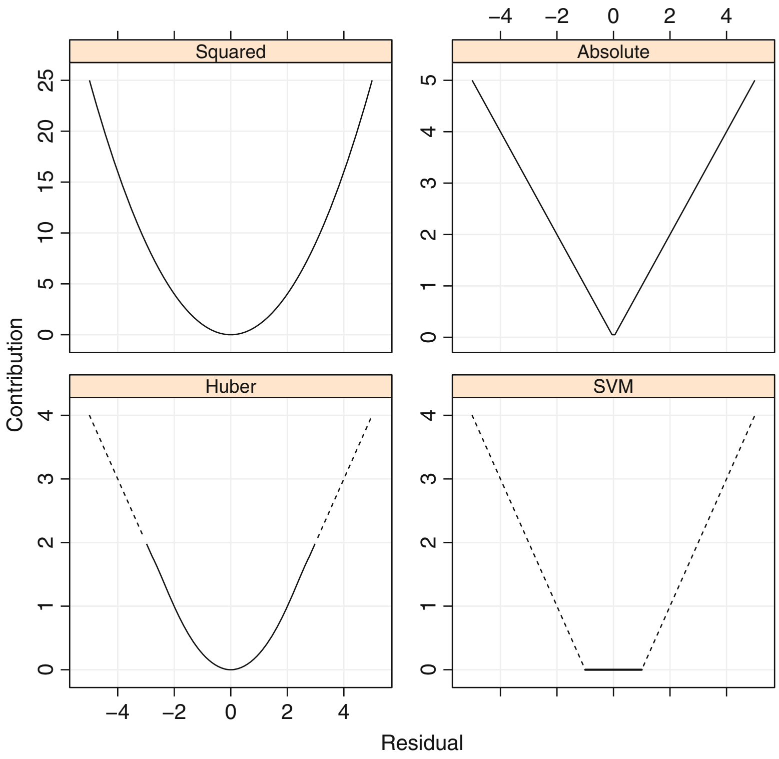

Fig. 7.6: The relationship between a model residual and its contribution to the regression line for several techniques. For the Huber approach, a threshold of 2 was used while for the support vector machine, a value of $\epsilon = 1$ was used. Note that the y-axis scales are different to make the figures easier to read

There are several aspects of this equation worth pointing out. First, there are as many α parameters as there are data points. From the standpoint of classical regression modeling, this model would be considered *over-parameterized*; typically, it is better to estimate fewer parameters than data points. However, the use of the cost value effectively regularizes the model to help alleviate this problem.

Second, the individual training set data points (i.e., the x_{ij}) are required for new predictions. When the training set is large, this makes the prediction equations less compact than other techniques. However, for some percentage of the training set samples, the α_i parameters will be exactly zero, indicating that they have no impact on the prediction equation. The data points associated with an α_i parameter of zero are the training set samples that are within $\pm\epsilon$ of the regression line (i.e., are within the "funnel" or "tube" around the regression line). As a consequence, only a subset of training set data points, where $\alpha \neq 0$, are needed for prediction. Since the regression line is determined using these samples, they are called the *support vectors* as they support the regression line.

Figure 7.7 illustrates the robustness of this model. A simple linear model was simulated with a slope of 4 and an intercept of 1; one extreme outlier was added to the data. The top panel shows the model fit for a linear regression model (black solid line) and a support vector machine regression model (blue dashed line) with $\epsilon = 0.01$. The linear regression line is pulled towards this point, resulting in estimates of the slope and intercept of 3.5 and 1.2, respectively. The support vector regression fit is shown in blue and is much closer to the true regression line with a slope of 3.9 and an intercept of 0.9. The middle panel again shows the SVM model, but the support vectors are solid black circles and the other points are shown in red. The horizontal grey reference lines indicate zero $\pm$ ϵ. Out of 100 data points, 70 of these were support vectors.

Finally, note that in the last form of Eq. 7.2, the new samples enter into the prediction function as sum of cross products with the new sample values. In matrix algebra terms, this corresponds to a *dot product* (i.e., $\mathbf{x}'\mathbf{u}$). This is important because this regression equation can be rewritten more generally as

$$f(\mathbf{u}) = \beta_0 + \sum_{i=1}^{n} \alpha_i K(\mathbf{x}_i, \mathbf{u}),$$

where $K(\cdot)$ is called the *kernel function.* When predictors enter the model linearly, the kernel function reduces to a simple sum of cross products shown above:

$$K(\mathbf{x}_i, \mathbf{u}) = \sum_{j=1}^{P} x_{ij} u_j = \mathbf{x}_i'\mathbf{u}.$$

However, there are other types of kernel functions that can be used to generalize the regression model and encompass *nonlinear* functions of the predictors:

$$\begin{aligned} \text{polynomial} &= \left(\phi\left(\mathbf{x}'\mathbf{u}\right) + 1\right)^{degree} \\ \text{radial basis function} &= \exp(-\sigma\|\mathbf{x} - \mathbf{u}\|^2) \\ \text{hyperbolic tangent} &= \tanh\left(\phi\left(\mathbf{x}'\mathbf{u}\right) + 1\right), \end{aligned}$$

where ϕ and σ are scaling parameters. Since these functions of the predictors lead to nonlinear models, this generalization is often called the "kernel trick."

To illustrate the ability of this model to adapt to nonlinear relationships, we simulated data that follow a sin wave in the bottom of Fig. 7.7. Outliers were also added to these data. A linear regression model with an intercept and a term for $sin(x)$ was fit to the model (solid black line). Again, the regression line is pulled towards the outlying points. An SVM model with a radial basis kernel function is represented by the blue dashed line (without specifying the sin functional form). This line better describes the overall structure of the data.

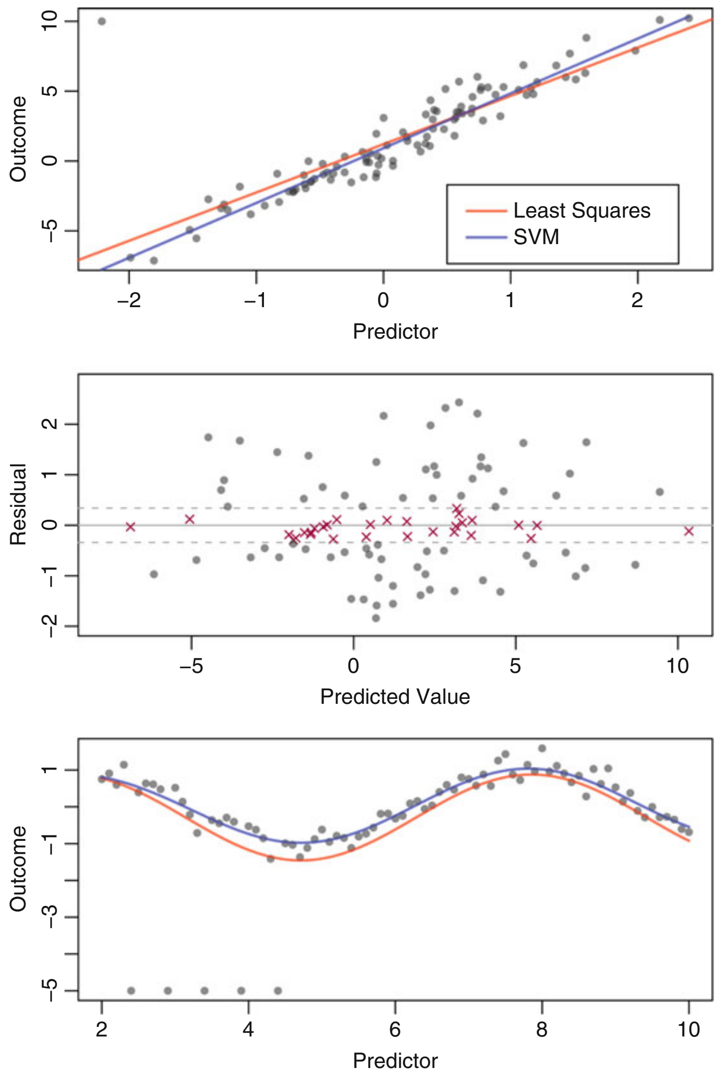

Fig. 7.7: The robustness qualities of SVM models. *Top*: a small simulated data set with a single large outlier is used to show the difference between an ordinary regression line (*red*) and the linear SVM model (*blue*). *Middle*: the SVM residuals versus the predicted values (the upper end of the y-axis scale was reduced to make the plot more readable). The plot symbols indicate the support vectors (shown as *grey colored circles*) and the other samples (*red crosses*). The *horizontal lines* are $\pm\epsilon = 0.01$. *Bottom*: A simulated sin wave with several outliers. The *red line* is an ordinary regression line (intercept and a term for $sin(x)$) and the *blue line* is a radial basis function SVM model

Which kernel function should be used? This depends on the problem. The radial basis function has been shown to be very effective. However, when the regression line is truly linear, the linear kernel function will be a better choice.

Note that some of the kernel functions have extra parameters. For example, the polynomial degree in the polynomial kernel must be specified. Similarly, the radial basis function has a parameter (σ) that controls the scale. These parameters, along with the cost value, constitute the tuning parameters for the model. In the case of the radial basis function, there is a possible computational shortcut to estimating the kernel parameter. Caputo et al. (2002) suggested that the parameter can be estimated using combinations of the training set points to calculate the distribution of $||x - x'||^2$, then use the 10th and 90th percentiles as a range for σ. Instead of tuning this parameter over a grid of candidate values, we can use the midpoint of these two percentiles.

The cost parameter is the main tool for adjusting the complexity of the model. When the cost is large, the model becomes very flexible since the effect of errors is amplified. When the cost is small, the model will "stiffen" and become less likely to over-fit (but more likely to underfit) because the contribution of the squared parameters is proportionally large in the modified error function. One could also tune the model over the size of the funnel (e.g., over ϵ). However, there is a relationship between ϵ and the cost parameter. In our experience, we have found that the cost parameter provides more flexibility for tuning the model. So we suggest fixing a value for ϵ and tuning over the other kernel parameters.

Since the predictors enter into the model as the sum of cross products, differences in the predictor scales can affect the model. Therefore, we recommend centering and scaling the predictors prior to building an SVM model.

SVMs were applied to the solubility data. First, a radial basis function kernel was used. The kernel parameter was estimated analytically to be $\sigma = 0.0039$ and the model was tuned over 14 cost values between 0.25 and 2048 on the $\log_2$ scale (Fig. 7.8). When the cost values are small, the model *under*-fits the data, but, as the error starts to increase when the cost approaches 2^{10}, over-fitting begins. The cost value associated with the smallest RMSE was 128. A polynomial model was also evaluated. Here, we tuned over the cost, the polynomial degree, and a scale factor. In general, quadratic models have smaller error rates than the linear models. Also, models associated with larger-scale factors have better performance. The optimal model was quadratic with a scale factor of 0.01 and a cost value of 2 (Fig. 7.9).

As a comparison, both the optimal radial basis and the polynomial SVM models use a similar number of support vectors, 623 and 627, respectively (out of 951 training samples). Also it is important to point out that tuning the radial basis function kernel parameter was easier than tuning the polynomial model (which has three tuning parameters).

The literature on SVM models and other kernel methods has been vibrant and many alternate methodologies have been proposed. One method,

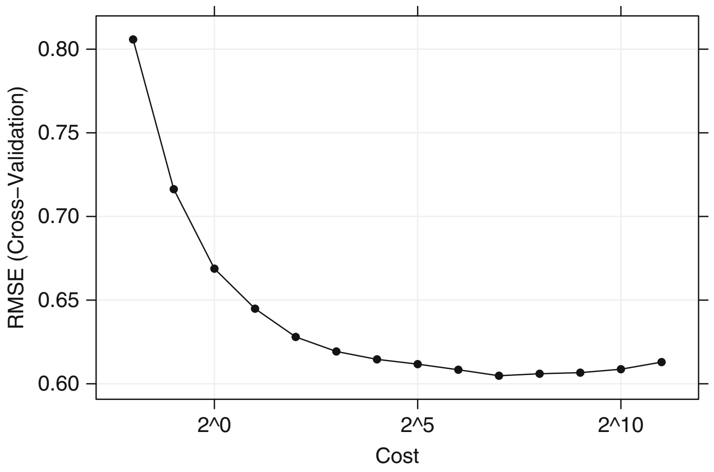

Fig. 7.8: The cross-validation profiles for a radial basis function SVM model applied to the solubility data. The kernel parameter was estimated analytically to be $\sigma = 0.0039$

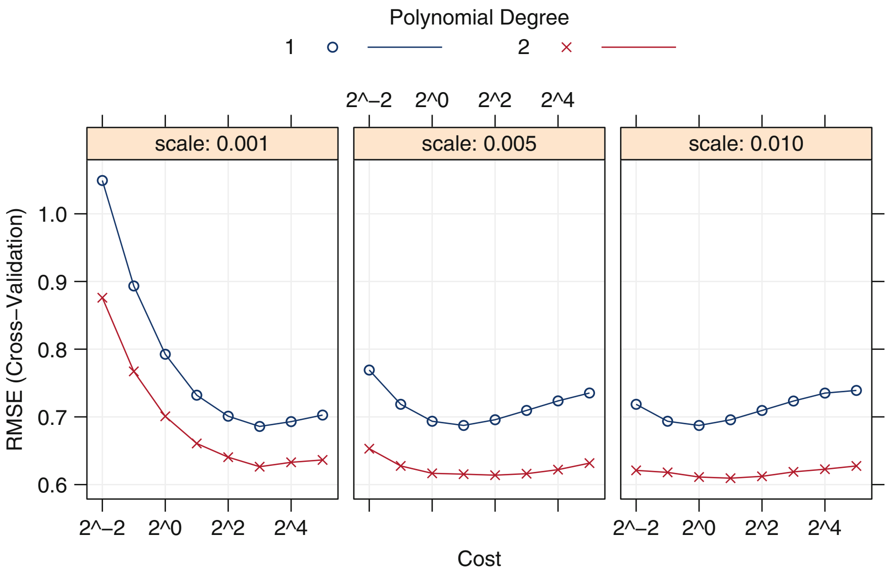

Fig. 7.9: Cross-validation results for the polynomial SVM model for the solubility data. The final model was fit using a quadratic model with a scale factor of 0.01 and a cost value of 2

the *relevance vector machine* (Tipping 2001), is a Bayesian analog to the SVM model. In this case, the α parameters described above have associated prior distributions and the selection of *relevance vectors* is determined using their posterior distribution. If the posterior distribution is highly concentrated around zero, the sample is not used in the prediction equation. There are usually less relevance vectors in this model than support vectors in an SVM model.

7.4 *K*-Nearest Neighbors

The KNN approach simply predicts a new sample using the K-closest samples from the training set (similar to Fig. 4.3). Unlike other methods in this chapter, KNN cannot be cleanly summarized by a model like the one presented in Eq. 7.2. Instead, its construction is solely based on the individual samples from the training data. To predict a new sample for regression, KNN identifies that sample's KNNs in the predictor space. The predicted response for the new sample is then the mean of the K neighbors' responses. Other summary statistics, such as the median, can also be used in place of the mean to predict the new sample.

The basic KNN method as described above depends on how the user defines distance between samples. Euclidean distance (i.e., the straight-line distance between two samples) is the most commonly used metric and is defined as follows:

$$\left(\sum_{j=1}^{P}(x_{aj}-x_{bj})^2\right)^{\frac{1}{2}},$$

where $\mathbf{x_a}$ and $\mathbf{x_b}$ are two individual samples. Minkowski distance is a generalization of Euclidean distance and is defined as

$$\left(\sum_{j=1}^{P}|x_{aj}-x_{bj}|^q\right)^{\frac{1}{q}},$$

where $q > 0$ (Liu 2007). It is easy to see that when $q = 2$, then Minkowski distance is the same as Euclidean distance. When $q = 1$, then Minkowski distance is equivalent to Manhattan (or city-block) distance, which is a common metric used for samples with binary predictors. Many other distance metrics exist, such as Tanimoto, Hamming, and cosine, and are more appropriate for specific types of predictors and in specific scientific contexts. Tanimoto distance, for example, is regularly used in computational chemistry problems when molecules are described using binary fingerprints (McCarren et al. 2011).

Because the KNN method fundamentally depends on distance between samples, the scale of the predictors can have a dramatic influence on the distances among samples. Data with predictors that are on vastly different scales will generate distances that are weighted towards predictors that have the largest scales. That is, predictors with the largest scales will contribute most to the distance between samples. To avoid this potential bias and to enable each predictor to contribute equally to the distance calculation, we recommend that all predictors be centered and scaled prior to performing KNN.

In addition to the issue of scaling, using distances between samples can be problematic if one or more of the predictor values for a sample is missing, since it is then not possible to compute the distance between samples. If this is the case, then the analyst has a couple of options. First, either the samples or the predictors can be excluded from the analysis. This is the least desirable option; however, it may be the only practical choice if the sample(s) or predictor(s) are sparse. If a predictor contains a sufficient amount of information across the samples, then an alternative approach is to impute the missing data using a naïve estimator such as the mean of the predictor, or a nearest neighbor approach that uses only the predictors with complete information (see Sect. 3.4).

Upon pre-processing the data and selecting the distance metric, the next step is to find the optimal number of neighbors. Like tuning parameters from other models, K can be determined by resampling. For the solubility data, 20 values of K ranging between 1 and 20 were evaluated. As illustrated in Fig. 7.10, the RMSE profile rapidly decreases across the first four values of K, then levels off through $K = 8$, followed by a steady increase in RMSE as K increases. This performance profile is typical for KNN, since small values of K usually over-fit and large values of K underfit the data. RMSE ranged from 1.041 to 1.23 across the candidate values, with the minimum occurring at $K = 4$; cross-validated R^2 at the optimum K is 0.747.

The elementary version of KNN is intuitive and straightforward and can produce decent predictions, especially when the response is dependent on the local predictor structure. However, this version does have some notable problems, of which researchers have sought solutions. Two commonly noted problems are computational time and the disconnect between local structure and the predictive ability of KNN.

First, to predict a sample, distances between the sample and all other samples must be computed. Computation time therefore increases with n because the training data must be loaded into memory and because distances between the new sample and all of the training samples must be computed. To mitigate this problem, one can replace the original data with a less memory-intensive representation of the data that describes the locations of the original data. One specific example of this representation is a k-dimensional tree (or k-d tree) (Bentley 1975). A k-d tree orthogonally partitions the predictor space using a tree approach but with different rules than the kinds of trees described

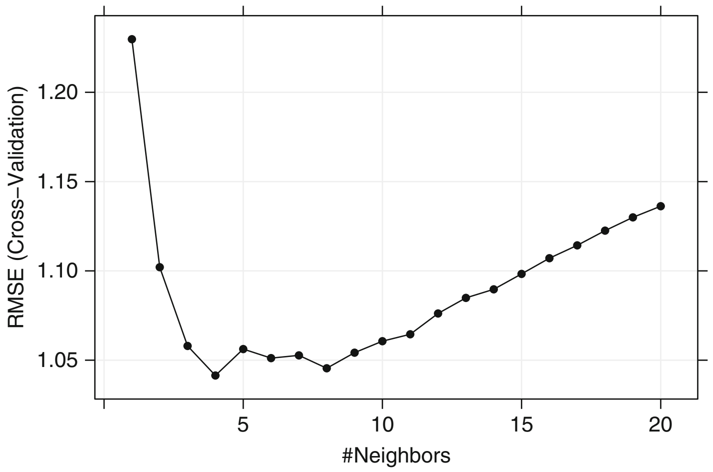

Fig. 7.10: The RMSE cross-validation profile for a KNN model applied to the solubility data. The optimal number of neighbors is 4

in Chap. 8. After the tree has been grown, a new sample is placed through the structure. Distances are only computed for those training observations in the tree that are close to the new sample. This approach provides significant computational improvements, especially when the number of training samples is much larger than the number of predictors.

The KNN method can have poor predictive performance when local predictor structure is not relevant to the response. Irrelevant or noisy predictors are one culprit, since these can cause similar samples to be driven away from each other in the predictor space. Hence, removing irrelevant, noise-laden predictors is a key pre-processing step for KNN. Another approach to enhancing KNN predictivity is to weight the neighbors' contribution to the prediction of a new sample based on their distance to the new sample. In this variation, training samples that are closer to the new sample contribute more to the predicted response, while those that are farther away contribute less to the predicted response.

7.5 Computing

This section will reference functions from the caret, earth, kernlab, and nnet packages.

R has a number of packages and functions for creating neural networks. Relevant packages include nnet, neural, and RSNNS. The nnet package is the focus here since it supports the basic neural network models outlined in this chapter (i.e., a single layer of hidden units) and weight decay and has simple syntax. RSNNS supports a wide array of neural networks. Bergmeir and Benitez (2012) outline the various neural network packages in R and contain a tutorial on RSNNS.

Neural Networks

To fit a regression model, the `nnet` function takes both the formula and non-formula interfaces. For regression, the linear relationship between the hidden units and the prediction can be used with the option `linout = TRUE`. A basic neural network function call would be

```
> nnetFit <- nnet(predictors, outcome,
+                  size = 5,
+                  decay = 0.01,
+                  linout = TRUE,
+                  ## Reduce the amount of printed output
+                  trace = FALSE,
+                  ## Expand the number of iterations to find
+                  ## parameter estimates..
+                  maxit = 500,
+                  ## and the number of parameters used by the model
+                  MaxNWts = 5 * (ncol(predictors) + 1) + 5 + 1)
```

This would create a single model with 5 hidden units. Note, this assumes that the data in `predictors` have been standardized to be on the same scale.

To use model averaging, the `avNNet` function in the caret package has nearly identical syntax:

```
> nnetAvg <- avNNet(predictors, outcome,
+                    size = 5,
+                    decay = 0.01,
+                    ## Specify how many models to average
+                    repeats = 5,
+                    linout = TRUE,
+                    ## Reduce the amount of printed output
+                    trace = FALSE,
+                    ## Expand the number of iterations to find
+                    ## parameter estimates..
+                    maxit = 500,
+                    ## and the number of parameters used by the model
+                    MaxNWts = 5 * (ncol(predictors) + 1) + 5 + 1)
```

Again, new samples are processed using

```
> predict(nnetFit, newData)
> ## or
> predict(nnetAvg, newData)
```

To mimic the earlier approach of choosing the number of hidden units and the amount of weight decay via resampling, the `train` function can be applied using either `method = "nnet"` or `method = "avNNet"`. First, we remove predictors to ensure that the maximum absolute pairwise correlation between the predictors is less than 0.75.

```
> ## The findCorrelation takes a correlation matrix and determines the
> ## column numbers that should be removed to keep all pair-wise
> ## correlations below a threshold
> tooHigh <- findCorrelation(cor(solTrainXtrans), cutoff = .75)
> trainXnnet <- solTrainXtrans[, -tooHigh]
> testXnnet <- solTestXtrans[, -tooHigh]
> ## Create a specific candidate set of models to evaluate:
> nnetGrid <- expand.grid(.decay = c(0, 0.01, .1),
+                         .size = c(1:10),
+                         ## The next option is to use bagging (see the
+                         ## next chapter) instead of different random
+                         ## seeds.
+                         .bag = FALSE)
> set.seed(100)
> nnetTune <- train(solTrainXtrans, solTrainY,
+                   method = "avNNet",
+                   tuneGrid = nnetGrid,
+                   trControl = ctrl,
+                   ## Automatically standardize data prior to modeling
+                   ## and prediction
+                   preProc = c("center", "scale"),
+                   linout = TRUE,
+                   trace = FALSE,
+                   MaxNWts = 10 * (ncol(trainXnnet) + 1) + 10 + 1,
+                   maxit = 500)
```

Multivariate Adaptive Regression Splines

MARS models are in several packages, but the most extensive implementation is in the earth package. The MARS model using the nominal forward pass and pruning step can be called simply

```
> marsFit <- earth(solTrainXtrans, solTrainY)
```

```
> marsFit
```

```
  Selected 38 of 47 terms, and 30 of 228 predictors
  Importance: NumNonHAtoms, MolWeight, SurfaceArea2, SurfaceArea1, FP142, ...
  Number of terms at each degree of interaction: 1 37 (additive model)
  GCV 0.3877448    RSS 312.877    GRSq 0.907529    RSq 0.9213739
```

Note that since this model used the internal GCV technique for model selection, the details of this model are different than the one used previously in the chapter. The `summary` method generates more extensive output:

```
> summary(marsFit)
```

```
Call: earth(x=solTrainXtrans, y=solTrainY)

                                coefficients
(Intercept)                        -3.223749
FP002                               0.517848
FP003                              -0.228759
FP059                              -0.582140
FP065                              -0.273844
FP075                               0.285520
FP083                              -0.629746
FP085                              -0.235622
FP099                               0.325018
FP111                              -0.403920
FP135                               0.394901
FP142                               0.407264
FP154                              -0.620757
FP172                              -0.514016
FP176                               0.308482
FP188                               0.425123
FP202                               0.302688
FP204                              -0.311739
FP207                               0.457080
h(MolWeight-5.77508)               -1.801853
h(5.94516-MolWeight)                0.813322
h(NumNonHAtoms-2.99573)            -3.247622
h(2.99573-NumNonHAtoms)             2.520305
h(2.57858-NumNonHBonds)            -0.564690
h(NumMultBonds-1.85275)            -0.370480
h(NumRotBonds-2.19722)             -2.753687
h(2.19722-NumRotBonds)              0.123978
h(NumAromaticBonds-2.48491)        -1.453716
h(NumNitrogen-0.584815)             8.239716
h(0.584815-NumNitrogen)            -1.542868
h(NumOxygen-1.38629)                3.304643
h(1.38629-NumOxygen)               -0.620413
h(NumChlorine-0.46875)            -50.431489
h(HydrophilicFactor- -0.816625)     0.237565
h(-0.816625-HydrophilicFactor)     -0.370998
h(SurfaceArea1-1.9554)              0.149166
h(SurfaceArea2-4.66178)            -0.169960
h(4.66178-SurfaceArea2)            -0.157970

Selected 38 of 47 terms, and 30 of 228 predictors
Importance: NumNonHAtoms, MolWeight, SurfaceArea2, SurfaceArea1, FP142, ...
Number of terms at each degree of interaction: 1 37 (additive model)
GCV 0.3877448    RSS 312.877    GRSq 0.907529    RSq 0.9213739
```

In this output, $h(\cdot)$ is the hinge function. In the output above, the term `h(MolWeight-5.77508)` is zero when the molecular weight is less than 5.77508 (i.e., similar to the top panel of Fig. 7.3). The reflected hinge function would be shown as `h(5.77508 - MolWeight)`.

The plotmo function in the earth package can be used to produce plots similar to Fig. 7.5. To tune the model using external resampling, the train function can be used. The following code reproduces the results in Fig. 7.4:

```
> # Define the candidate models to test
> marsGrid <- expand.grid(.degree = 1:2, .nprune = 2:38)
> # Fix the seed so that the results can be reproduced
> set.seed(100)
> marsTuned <- train(solTrainXtrans, solTrainY,
+                    method = "earth",
+                    # Explicitly declare the candidate models to test
+                    tuneGrid = marsGrid,
+                    trControl = trainControl(method = "cv"))


> marsTuned


  951 samples
  228 predictors

  No pre-processing
  Resampling: Cross-Validation (10-fold)

  Summary of sample sizes: 856, 857, 855, 856, 856, 855, ...

  Resampling results across tuning parameters:

    degree  nprune  RMSE   Rsquared  RMSE SD  Rsquared SD
    1       2       1.54   0.438     0.128    0.0802
    1       3       1.12   0.7       0.0968   0.0647
    1       4       1.06   0.73      0.0849   0.0594
    1       5       1.02   0.75      0.102    0.0551
    1       6       0.984  0.768     0.0733   0.042
    1       7       0.919  0.796     0.0657   0.0432
    1       8       0.862  0.821     0.0418   0.0237
    :       :         :      :         :        :
    2       33      0.701  0.883     0.068    0.0307
    2       34      0.702  0.883     0.0699   0.0307
    2       35      0.696  0.885     0.0746   0.0315
    2       36      0.687  0.887     0.0604   0.0281
    2       37      0.696  0.885     0.0689   0.0291
    2       38      0.686  0.887     0.0626   0.029

  RMSE was used to select the optimal model using  the smallest value.
  The final values used for the model were degree = 1 and nprune = 38.


> head(predict(marsTuned, solTestXtrans))
  [1]  0.3677522 -0.1503220 -0.5051844  0.5398116 -0.4792718  0.7377222
```

There are two functions that estimate the importance of each predictor in the MARS model: evimp in the earth package and varImp in the caret package (although the latter calls the former):

```
> varImp(marsTuned)
  earth variable importance

    only 20 most important variables shown (out of 228)

                 Overall
  MolWeight      100.00
  NumNonHAtoms    89.96
  SurfaceArea2    89.51
  SurfaceArea1    57.34
  FP142           44.31
  FP002           39.23
  NumMultBonds    39.23
  FP204           37.10
  FP172           34.96
  NumOxygen       30.70
  NumNitrogen     29.12
  FP083           28.21
  NumNonHBonds    26.58
  FP059           24.76
  FP135           23.51
  FP154           21.20
  FP207           19.05
  FP202           17.92
  NumRotBonds     16.94
  FP085           16.02
```

These results are scaled to be between 0 and 100 and are different than those shown in Table 7.1 (since the model in Table 7.1 did not undergo the full model growing and pruning process). Note that after the first few variables, the remainder have much smaller importance to the model.

Support Vector Machines

There are a number of R packages with implementations of support vector machine models. The `svm` function in the e1071 package has an interface to the LIBSVM library (Chang and Lin 2011) for regression. A more comprehensive implementation of SVM models for regression is the kernlab package (Karatzoglou et al. 2004). In that package, the `ksvm` function is available for regression models and a large number of kernel functions. The radial basis function is the default kernel function. If appropriate values of the cost and kernel parameters are known, this model can be fit as

```
> svmFit <- ksvm(x = solTrainXtrans, y = solTrainY,
+                kernel ="rbfdot", kpar = "automatic",
+                C = 1, epsilon = 0.1)
```

The function automatically uses the analytical approach to estimate σ. Since `y` is a numeric vector, the function knows to fit a regression model (instead of a classification model). Other kernel functions can be used, including the polynomial (using `kernel = "polydot"`) and linear (`kernel = "vanilladot"`).

If the values are unknown, they can be estimated through resampling. In `train`, the `method` values of `"svmRadial"`, `"svmLinear"`, or `"svmPoly"` fit different kernels:

```
> svmRTuned <- train(solTrainXtrans, solTrainY,
+                    method = "svmRadial",
+                    preProc = c("center", "scale"),
+                    tuneLength = 14,
+                    trControl = trainControl(method = "cv"))
```

The `tuneLength` argument will use the default grid search of 14 cost values between $2^{-2}, 2^{-1}, \ldots, 2^{11}$. Again, σ is estimated analytically by default.

```
> svmRTuned
  951 samples
  228 predictors

  Pre-processing: centered, scaled
  Resampling: Cross-Validation (10-fold)

  Summary of sample sizes: 855, 858, 856, 855, 855, 856, ...

  Resampling results across tuning parameters:

    C     RMSE   Rsquared  RMSE SD  Rsquared SD
    0.25  0.793  0.87      0.105    0.0396
    0.5   0.708  0.889     0.0936   0.0345
    1     0.664  0.898     0.0834   0.0306
    2     0.642  0.903     0.0725   0.0277
    4     0.629  0.906     0.067    0.0253
    8     0.621  0.908     0.0634   0.0238
    16    0.617  0.909     0.0602   0.0232
    32    0.613  0.91      0.06     0.0234
    64    0.611  0.911     0.0586   0.0231
    128   0.609  0.911     0.0561   0.0223
    256   0.609  0.911     0.056    0.0224
    512   0.61   0.911     0.0563   0.0226
    1020  0.613  0.91      0.0563   0.023
    2050  0.618  0.909     0.0541   0.023

  Tuning parameter 'sigma' was held constant at a value of 0.00387
  RMSE was used to select the optimal model using  the smallest value.
  The final values used for the model were C = 256 and sigma = 0.00387.
```

The subobject named `finalModel` contains the model created by the `ksvm` function:

```
> svmRTuned$finalModel
  Support Vector Machine object of class "ksvm"

  SV type: eps-svr  (regression)
   parameter : epsilon = 0.1  cost C = 256
```

```
Gaussian Radial Basis kernel function.
 Hyperparameter : sigma =  0.00387037424967707

Number of Support Vectors : 625

Objective Function Value : -1020.558
Training error : 0.009163
```

Here, we see that the model used 625 training set data points as support vectors (66 % of the training set).

kernlab has an implementation of the RVM model for regression in the function `rvm`. The syntax is very similar to the example shown for `ksvm`.

K-Nearest Neighbors

The `knnreg` function in the caret package fits the KNN regression model; `train` tunes the model over K:

```
> # Remove a few sparse and unbalanced fingerprints first
> knnDescr <- solTrainXtrans[, -nearZeroVar(solTrainXtrans)]
> set.seed(100)
> knnTune <- train(knnDescr,
+                  solTrainY,
+                  method = "knn",
+                  # Center and scaling will occur for new predictions too
+                  preProc = c("center", "scale"),
+                  tuneGrid = data.frame(.k = 1:20),
+                  trControl = trainControl(method = "cv"))
```

When predicting new samples using this object, the new samples are automatically centered and scaled using the values determined by the training set.

Exercises

7.1. Simulate a single predictor and a nonlinear relationship, such as a *sin* wave shown in Fig. 7.7, and investigate the relationship between the cost, ϵ, and kernel parameters for a support vector machine model:

```
> set.seed(100)
> x <- runif(100, min = 2, max = 10)
> y <-  sin(x) + rnorm(length(x)) * .25
> sinData <- data.frame(x = x, y = y)
> plot(x, y)
> ## Create a grid of x values to use for prediction
> dataGrid <- data.frame(x = seq(2, 10, length = 100))
```

(a) Fit different models using a radial basis function and different values of the cost (the `C` parameter) and ϵ. Plot the fitted curve. For example:

```
> library(kernlab)
> rbfSVM <- ksvm(x = x, y = y, data = sinData,
+                kernel ="rbfdot", kpar = "automatic",
+                C = 1, epsilon = 0.1)
> modelPrediction <- predict(rbfSVM, newdata = dataGrid)
> ## This is a matrix with one column. We can plot the
> ## model predictions by adding points to the previous plot
> points(x = dataGrid$x, y = modelPrediction[,1],
+        type = "l", col = "blue")
> ## Try other parameters
```

(b) The σ parameter can be adjusted using the `kpar` argument, such as `kpar = list(sigma = 1)`. Try different values of σ to understand how this parameter changes the model fit. How do the cost, ϵ, and σ values affect the model?

7.2. Friedman (1991) introduced several benchmark data sets create by simulation. One of these simulations used the following nonlinear equation to create data:

$$y = 10\sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5 + N(0, \sigma^2)$$

where the x values are random variables uniformly distributed between [0, 1] (there are also 5 other non-informative variables also created in the simulation). The package mlbench contains a function called `mlbench.friedman1` that simulates these data:

```
> library(mlbench)
> set.seed(200)
> trainingData <- mlbench.friedman1(200, sd = 1)
> ## We convert the 'x' data from a matrix to a data frame
> ## One reason is that this will give the columns names.
> trainingData$x <- data.frame(trainingData$x)
> ## Look at the data using
> featurePlot(trainingData$x, trainingData$y)
> ## or other methods.
>
> ## This creates a list with a vector 'y' and a matrix
> ## of predictors 'x'. Also simulate a large test set to
> ## estimate the true error rate with good precision:
> testData <- mlbench.friedman1(5000, sd = 1)
> testData$x <- data.frame(testData$x)
>
```

Tune several models on these data. For example:

```
> library(caret)
> knnModel <- train(x = trainingData$x,
+                   y = trainingData$y,
+                   method = "knn",
+                    preProc = c("center", "scale"),
+                    tuneLength = 10)
> knnModel
  200 samples
   10 predictors

  Pre-processing: centered, scaled
  Resampling: Bootstrap (25 reps)

  Summary of sample sizes: 200, 200, 200, 200, 200, 200, ...

  Resampling results across tuning parameters:

    k   RMSE  Rsquared  RMSE SD  Rsquared SD
    5   3.51  0.496     0.238    0.0641
    7   3.36  0.536     0.24     0.0617
    9   3.3   0.559     0.251    0.0546
    11  3.24  0.586     0.252    0.0501
    13  3.2   0.61      0.234    0.0465
    15  3.19  0.623     0.264    0.0496
    17  3.19  0.63      0.286    0.0528
    19  3.18  0.643     0.274    0.048
    21  3.2   0.646     0.269    0.0464
    23  3.2   0.652     0.267    0.0465

  RMSE was used to select the optimal model using  the smallest value.
  The final value used for the model was k = 19.
> knnPred <- predict(knnModel, newdata = testData$x)
> ## The function 'postResample' can be used to get the test set
> ## perforamnce values
> postResample(pred = knnPred, obs = testData$y)
       RMSE  Rsquared
  3.2286834 0.6871735
```

Which models appear to give the best performance? Does MARS select the informative predictors (those named `X1`–`X5`)?

7.3. For the Tecator data described in the last chapter, build SVM, neural network, MARS, and KNN models. Since neural networks are especially sensitive to highly correlated predictors, does pre-processing using PCA help the model?

7.4. Return to the permeability problem outlined in Exercise 6.2. Train several nonlinear regression models and evaluate the resampling and test set performance.

(a) Which nonlinear regression model gives the optimal resampling and test set performance?
(b) Do any of the nonlinear models outperform the optimal linear model you previously developed in Exercise 6.2? If so, what might this tell you about the underlying relationship between the predictors and the response?

(c) Would you recommend any of the models you have developed to replace the permeability laboratory experiment?

7.5. Exercise 6.3 describes data for a chemical manufacturing process. Use the same data imputation, data splitting, and pre-processing steps as before and train several nonlinear regression models.

(a) Which nonlinear regression model gives the optimal resampling and test set performance?
(b) Which predictors are most important in the optimal nonlinear regression model? Do either the biological or process variables dominate the list? How do the top ten important predictors compare to the top ten predictors from the optimal linear model?
(c) Explore the relationships between the top predictors and the response for the predictors that are unique to the optimal nonlinear regression model. Do these plots reveal intuition about the biological or process predictors and their relationship with yield?