Chapter 4
Over-Fitting and Model Tuning

Many modern classification and regression models are highly adaptable; they are capable of modeling complex relationships. However, they can very easily overemphasize patterns that are not reproducible. Without a methodological approach to evaluating models, the modeler will not know about the problem until the next set of samples are predicted.

Over-fitting has been discussed in the fields of forecasting (Clark 2004), medical research (Simon et al. 2003; Steyerberg 2010), chemometrics (Gowen et al. 2010; Hawkins 2004; Defernez and Kemsley 1997), meteorology (Hsieh and Tang 1998), finance (Dwyer 2005), and marital research (Heyman and Slep 2001) to name a few. These references illustrate that over-fitting is a concern for any predictive model regardless of field of research. The aim of this chapter is to explain and illustrate key principles of laying a foundation onto which trustworthy models can be built and subsequently used for prediction. More specifically, we will describe strategies that enable us to have confidence that the model we build will predict new samples with a similar degree of accuracy on the set of data for which the model was evaluated. Without this confidence, the model's predictions are *useless*.

On a practical note, all model building efforts are constrained by the existing data. For many problems, the data may have a limited number of samples, may be of less-than-desirable quality, and/or may be unrepresentative of future samples. While there are ways to build predictive models on small data sets, which we will describe in this chapter, we will assume that data quality is sufficient and that it is representative of the entire sample population.

Working under these assumptions, we must use the data at hand to find the best predictive model. Almost all predictive modeling techniques have tuning parameters that enable the model to flex to find the structure in the data. Hence, we must use the existing data to identify settings for the model's parameters that yield the best and most realistic predictive performance (known as model tuning). Traditionally, this has been achieved by splitting the existing data into training and test sets. The training set is used to build and tune the model and the test set is used to estimate the model's

M. Kuhn and K. Johnson, *Applied Predictive Modeling*,
DOI 10.1007/978-1-4614-6849-3_4,

predictive performance. Modern approaches to model building split the data into multiple training and testing sets, which have been shown to often find more optimal tuning parameters and give a more accurate representation of the model's predictive performance.

To begin this chapter we will illustrate the concept of over-fitting through an easily visualized example. To avoid over-fitting, we propose a general model building approach that encompasses model tuning and model evaluation with the ultimate goal of finding the reproducible structure in the data. This approach entails splitting existing data into distinct sets for the purposes of tuning model parameters and evaluating model performance. The choice of data splitting method depends on characteristics of the existing data such as its size and structure. In Sect. 4.4, we define and explain the most versatile data splitting techniques and explore the advantages and disadvantages of each. Finally, we end the chapter with a computing section that provides code for implementing the general model building strategy.

4.1 The Problem of Over-Fitting

There now exist many techniques that can learn the structure of a set of data so well that when the model is applied to the data on which the model was built, it correctly predicts every sample. In addition to learning the general patterns in the data, the model has also learned the characteristics of each sample's unique noise. This type of model is said to be over-fit and will usually have poor accuracy when predicting a new sample. To illustrate over-fitting and other concepts in this chapter, consider the simple classification example in Fig. 4.1 that has two predictor variables (i.e., independent variables). These data contain 208 samples that are designated either as "Class 1" or "Class 2." The classes are fairly balanced; there are 111 samples in the first class and 97 in the second. Furthermore, there is a significant overlap between the classes which is often the case for most applied modeling problems.

One objective for a data set such as this would be to develop a model to classify new samples. In this two-dimensional example, the classification models or rules can be represented by boundary lines. Figure 4.2 shows example class boundaries from two distinct classification models. The lines envelop the area where each model predicts the data to be the second class (blue squares). The left-hand panel ("Model #1") shows a boundary that is complex and attempts to encircle every possible data point. The pattern in this panel is not likely to generalize to new data. The right-hand panel shows an alternative model fit where the boundary is fairly smooth and does not overextend itself to correctly classify every data point in the training set.

To gauge how well the model is classifying samples, one might use the training set. In doing so, the estimated error rate for the model in the left-hand panel would be overly optimistic. Estimating the utility of a model

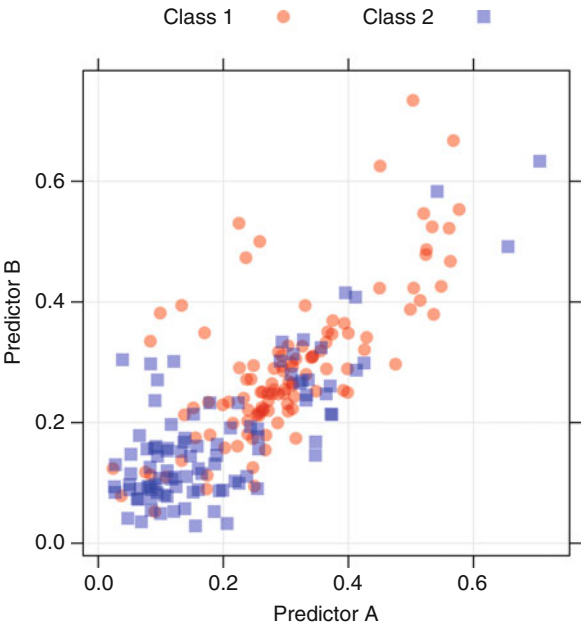

Fig. 4.1: An example of classification data that is used throughout the chapter

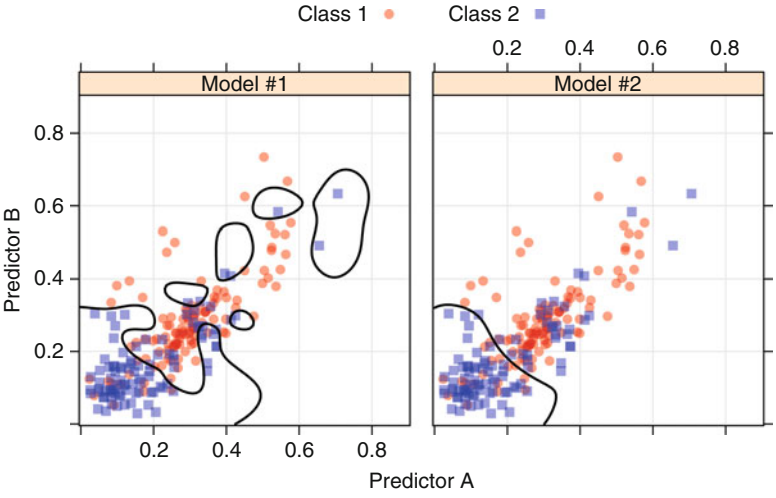

Fig. 4.2: An example of a training set with two classes and two predictors. The panels show two different classification models and their associated class boundaries

by re-predicting the training set is referred to *apparent performance* of the model (e.g., the apparent error rate). In two dimensions, it is not difficult to visualize that one model is over-fitting, but most modeling problems are in much higher dimensions. In these situations, it is very important to have a tool for characterizing how much a model is over-fitting the training data.

4.2 Model Tuning

Many models have important parameters which cannot be directly estimated from the data. For example, in the K-nearest neighbor classification model, a new sample is predicted based on the K-closest data points in the training set. An illustration of a 5-nearest neighbor model is shown in Fig. 4.3. Here, two new samples (denoted by the solid dot and filled triangle) are being predicted. One sample (•) is near a mixture of the two classes; three of the five neighbors indicate that the sample should be predicted as the first class. The other sample (▲) has all five points indicating the second class should be predicted. The question remains as to how many neighbors should be used. A choice of too few neighbors may over-fit the individual points of the training set while too many neighbors may not be sensitive enough to yield

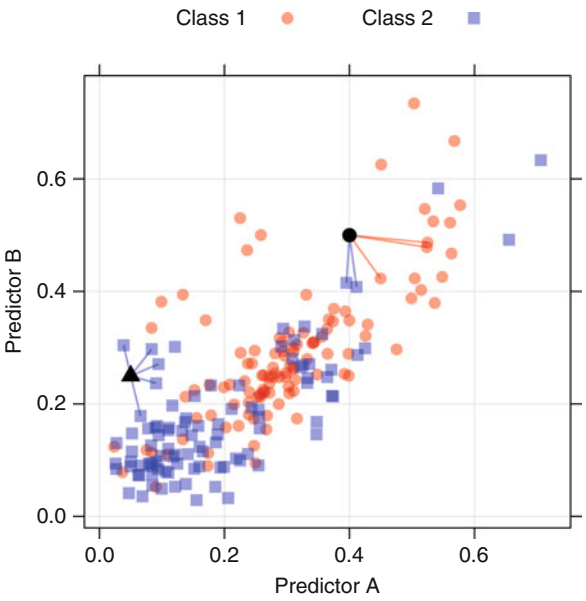

Fig. 4.3: The K-nearest neighbor classification model. Two new points, symbolized by *filled triangle* and *solid dot*, are predicted using the training set

reasonable performance. This type of model parameter is referred to as a *tuning parameter* because there is no analytical formula available to calculate an appropriate value.

Several models discussed in this text have at least one tuning parameter. Since many of these parameters control the complexity of the model, poor choices for the values can result in over-fitting. Figure 4.2 illustrates this point. A support vector machine (Sect. 13.4) was used to generate the class boundaries in each panel. One of the tuning parameters for this model sets the price for misclassified samples in the training set and is generally referred to as the "cost" parameter. When the cost is large, the model will go to great lengths to correctly label every point (as in the left panel) while smaller values produce models that are not as aggressive. The class boundary in the left panel was created by manually setting the cost parameter to a very high number. In the right panel, the cost value was determined using cross-validation (Sect. 4.4).

There are different approaches to searching for the best parameters. A general approach that can be applied to almost any model is to define a set of candidate values, generate reliable estimates of model utility across the candidates values, then choose the optimal settings. A flowchart of this process is shown in Fig. 4.4.

Once a candidate set of parameter values has been selected, then we must obtain trustworthy estimates of model performance. The performance on the hold-out samples is then aggregated into a performance profile which is then used to determine the final tuning parameters. We then build a final model with all of the training data using the selected tuning parameters. Using the K-nearest neighbor example to illustrate the procedure of Fig. 4.4, the candidate set might include all odd values of K between 1 and 9 (odd values are used in the two-class situation to avoid ties). The training data would then be resampled and evaluated many times for each tuning parameter value. These results would then be aggregated to find the optimal value of K.

The procedure defined in Fig. 4.4 uses a set of candidate models that are defined by the tuning parameters. Other approaches such as genetic algorithms (Mitchell 1998) or simplex search methods (Olsson and Nelson 1975) can also find optimal tuning parameters. These procedures algorithmically determine appropriate values for tuning parameters and iterate until they arrive at parameter settings with optimal performance. These techniques tend to evaluate a large number of candidate models and can be superior to a defined set of tuning parameters when model performance can be efficiently calculated. Cohen et al. (2005) provides a comparison of search routines for tuning a support vector machine model.

A more difficult problem is obtaining trustworthy estimates of model performance for these candidate models. As previously discussed, the apparent error rate can produce extremely optimistic performance estimates. A better approach is to test the model on samples that were not used for training.

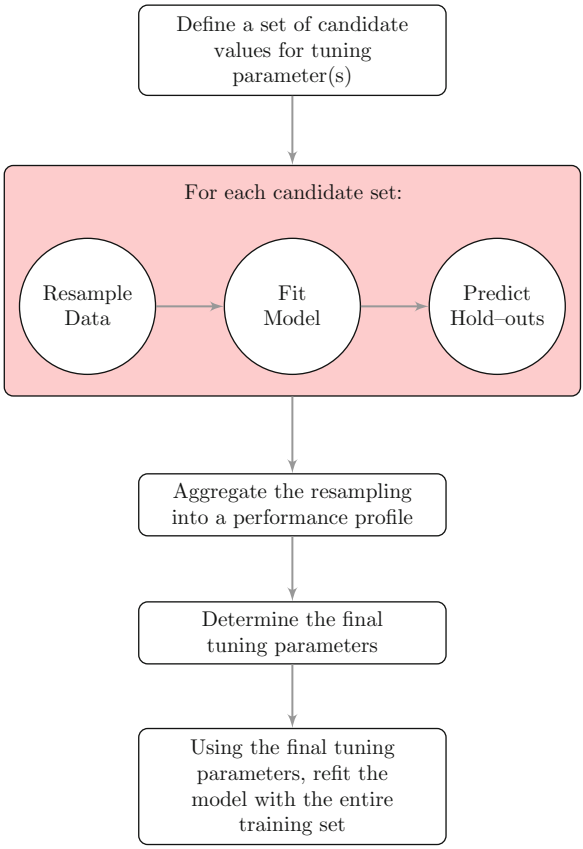

Fig. 4.4: A schematic of the parameter tuning process. An example of a candidate set of tuning parameter values for K-nearest neighbors might be odd numbers between 1 and 9. For each of these values, the data would be resampled multiple times to assess model performance for each value

Evaluating the model on a test set is the obvious choice, but, to get reasonable precision of the performance values, the size of the test set may need to be large.

An alternate approach to evaluating a model on a single test set is to *resample* the training set. This process uses several modified versions of the training set to build multiple models and then uses statistical methods to provide honest estimates of model performance (i.e., not overly optimistic). Section 4.4 illustrates several resampling techniques, and Sect. 4.6 discusses approaches to choose the final parameters using the resampling results.

4.3 Data Splitting

Now that we have outlined the general procedure for finding optimal tuning parameters, we turn to discussing the heart of the process: data splitting. A few of the common steps in model building are:

- Pre-processing the predictor data
- Estimating model parameters
- Selecting predictors for the model
- Evaluating model performance
- Fine tuning class prediction rules (via ROC curves, etc.)

Given a fixed amount of data, the modeler must decide how to "spend" their data points to accommodate these activities.

One of the first decisions to make when modeling is to decide which samples will be used to evaluate performance. Ideally, the model should be evaluated on samples that were not used to build or fine-tune the model, so that they provide an unbiased sense of model effectiveness. When a large amount of data is at hand, a set of samples can be set aside to evaluate the final model. The "training" data set is the general term for the samples used to create the model, while the "test" or "validation" data set is used to qualify performance.

However, when the number of samples is not large, a strong case can be made that a test set should be avoided because every sample may be needed for model building. Additionally, the size of the test set may not have sufficient power or precision to make reasonable judgements. Several researchers (Molinaro 2005; Martin and Hirschberg 1996; Hawkins et al. 2003) show that validation using a single test set can be a poor choice. Hawkins et al. (2003) concisely summarize this point: "holdout samples of tolerable size [...] do not match the cross-validation itself for reliability in assessing model fit and are hard to motivate." Resampling methods, such as cross-validation, can be used to produce appropriate estimates of model performance using the training set. These are discussed in length in Sect. 4.4. Although resampling techniques can be misapplied, such as the example shown in Ambroise and McLachlan (2002), they often produce performance estimates superior to a single test set because they evaluate many alternate versions of the data.

If a test set is deemed necessary, there are several methods for splitting the samples. Nonrandom approaches to splitting the data are sometimes appropriate. For example,

- If a model was being used to predict patient outcomes, the model may be created using certain patient sets (e.g., from the same clinical site or disease stage), and then tested on a different sample population to understand how well the model generalizes.
- In chemical modeling for drug discovery, new "chemical space" is constantly being explored. We are most interested in accurate predictions in the chemical space that is currently being investigated rather than the space that

was evaluated years prior. The same could be said for spam filtering; it is more important for the model to catch the new spamming techniques rather than prior spamming schemes.

However, in most cases, there is the desire to make the training and test sets as homogeneous as possible. Random sampling methods can be used to create similar data sets.

The simplest way to split the data into a training and test set is to take a simple random sample. This does not control for any of the data attributes, such as the percentage of data in the classes. When one class has a disproportionately small frequency compared to the others, there is a chance that the distribution of the outcomes may be substantially different between the training and test sets.

To account for the outcome when splitting the data, stratified random sampling applies random sampling within subgroups (such as the classes). In this way, there is a higher likelihood that the outcome distributions will match. When the outcome is a number, a similar strategy can be used; the numeric values are broken into similar groups (e.g., low, medium, and high) and the randomization is executed within these groups.

Alternatively, the data can be split on the basis of the predictor values. Willett (1999) and Clark (1997) propose data splitting based on *maximum dissimilarity sampling*. Dissimilarity between two samples can be measured in a number of ways. The simplest method is to use the distance between the predictor values for two samples. If the distance is small, the points are in close proximity. Larger distances between points are indicative of dissimilarity. To use dissimilarity as a tool for data splitting, suppose the test set is initialized with a single sample. The dissimilarity between this initial sample and the unallocated samples can be calculated. The unallocated sample that is most dissimilar would then be added to the test set. To allocate more samples to the test set, a method is needed to determine the dissimilarities between *groups* of points (i.e., the two in the test set and the unallocated points). One approach is to use the average or minimum of the dissimilarities. For example, to measure the dissimilarities between the two samples in the test set and a single unallocated point, we can determine the two dissimilarities and average them. The third point added to the test set would be chosen as having the maximum average dissimilarity to the existing set. This process would continue until the targeted test set size is achieved.

Figure 4.5 illustrates this process for the example classification data. Dissimilarity sampling was conducted separately within each class. First, a sample within each class was chosen to start the process (designated as ■ and ● in the figure). The dissimilarity of the initial sample to the unallocated samples within the class was computed and the most dissimilar point was added to the test set. For the first class, the most dissimilar point was in the extreme Southwest of the initial sample. On the second round, the dissimilarities were aggregated using the minimum (as opposed to the average). Again,

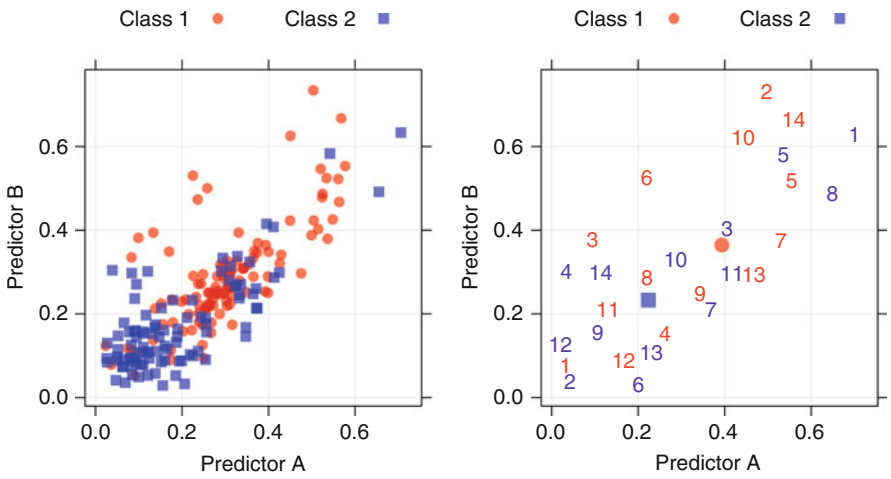

Fig. 4.5: An example of maximum dissimilarity sampling to create a test set. After choosing an initial sample within a class, 14 more samples were added

for the first class, the chosen point was far in the Northeast of the predictor space. As the sampling proceeds, samples were selected on the periphery of the data then work inward.

Martin et al. (2012) compares different methods of splitting data, including random sampling, dissimilarity sampling, and other methods.

4.4 Resampling Techniques

Generally, resampling techniques for estimating model performance operate similarly: a subset of samples are used to fit a model and the remaining samples are used to estimate the efficacy of the model. This process is repeated multiple times and the results are aggregated and summarized. The differences in techniques usually center around the method in which subsamples are chosen. We will consider the main flavors of resampling in the next few subsections.

k-Fold Cross-Validation

The samples are randomly partitioned into k sets of roughly equal size. A model is fit using the all samples except the first subset (called the first *fold*). The held-out samples are predicted by this model and used to estimate performance measures. The first subset is returned to the training set and

procedure repeats with the second subset held out, and so on. The k resampled estimates of performance are summarized (usually with the mean and standard error) and used to understand the relationship between the tuning parameter(s) and model utility. The cross-validation process with $k = 3$ is depicted in Fig. 4.6.

A slight variant of this method is to select the k partitions in a way that makes the folds balanced with respect to the outcome (Kohavi 1995). Stratified random sampling, previously discussed in Sect. 4.3, creates balance with respect to the outcome.

Another version, leave-one-out cross-validation (LOOCV), is the special case where k is the number of samples. In this case, since only one sample is held-out at a time, the final performance is calculated from the k individual held-out predictions. Additionally, repeated k-fold cross-validation replicates the procedure in Fig. 4.6 multiple times. For example, if 10-fold cross-validation was repeated five times, 50 different held-out sets would be used to estimate model efficacy.

The choice of k is usually 5 or 10, but there is no formal rule. As k gets larger, the difference in size between the training set and the resampling subsets gets smaller. As this difference decreases, the *bias* of the technique becomes smaller (i.e., the bias is smaller for $k = 10$ than $k = 5$). In this context, the bias is the difference between the estimated and true values of performance.

Another important aspect of a resampling technique is the uncertainty (i.e., variance or noise). An unbiased method may be estimating the correct value (e.g., the true theoretical performance) but may pay a high price in uncertainty. This means that repeating the resampling procedure may produce a very different value (but done enough times, it will estimate the true value). k-fold cross-validation generally has high variance compared to other methods and, for this reason, might not be attractive. It should be said that for large training sets, the potential issues with variance and bias become negligible.

From a practical viewpoint, larger values of k are more computationally burdensome. In the extreme, LOOCV is most computationally taxing because it requires as many model fits as data points and each model fit uses a subset that is nearly the same size of the training set. Molinaro (2005) found that leave-one-out and k =10-fold cross-validation yielded similar results, indicating that $k = 10$ is more attractive from the perspective of computational efficiency. Also, small values of k, say 2 or 3, have high bias but are very computationally efficient. However, the bias that comes with small values of k is about the same as the bias produced by the bootstrap (see below), but with much larger variance.

Research (Molinaro 2005; Kim 2009) indicates that repeating k-fold cross-validation can be used to effectively increase the precision of the estimates while still maintaining a small bias.

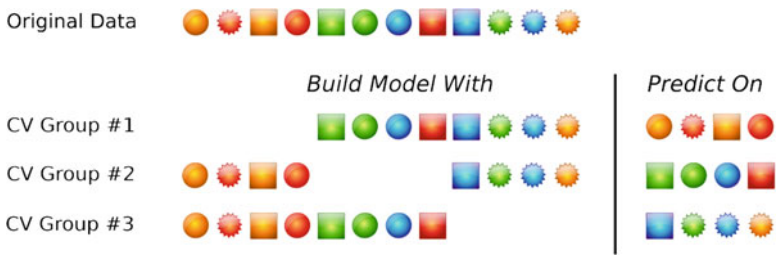

Fig. 4.6: A schematic of threefold cross-validation. Twelve training set samples are represented as symbols and are allocated to three groups. These groups are left out in turn as models are fit. Performance estimates, such as the error rate or R^2 are calculated from each set of held-out samples. The average of the three performance estimates would be the cross-validation estimate of model performance. In practice, the number of samples in the held-out subsets can vary but are roughly equal size

Generalized Cross-Validation

For linear regression models, there is a formula for approximating the leave-one-out error rate. The generalized cross-validation (GCV) statistic (Golub et al. 1979) does not require iterative refitting of the model to different data subsets. The formula for this statistic is the i_{th} training set outcome

$$\text{GCV} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{y}_i}{1 - df/n}\right)^2,$$

where y_i is the i^{th} in the training set set outcome, $\hat{y}_i$ is the model prediction of that outcome, and df is the degrees of freedom of the model. The degrees of freedom are an accounting of how many parameters are estimated by the model and, by extension, a measure of complexity for linear regression models. Based on this equation, two models with the same sums of square errors (the numerator) would have different GCV values if the complexities of the models were different.

Repeated Training/Test Splits

Repeated training/test splits is also known as "leave-group-out cross-validation" or "Monte Carlo cross-validation." This technique simply creates multiple splits of the data into modeling and prediction sets (see Fig. 4.7). The proportion of the data going into each subset is controlled by the practitioner as is the number of repetitions. As previously discussed, the bias

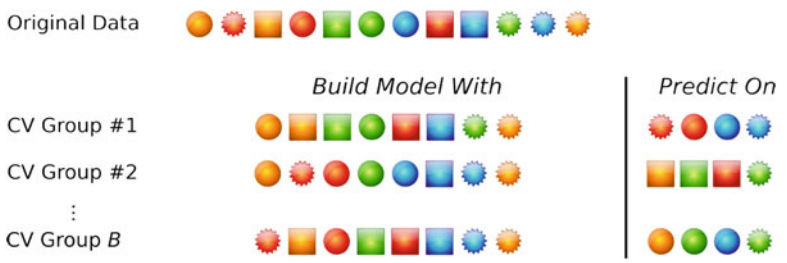

Fig. 4.7: A schematic of B repeated training and test set partitions. Twelve training set samples are represented as symbols and are allocated to B subsets that are 2/3 of the original training set. One difference between this procedure and k-fold cross-validation are that samples can be represented in multiple held-out subsets. Also, the number of repetitions is usually larger than in k-fold cross-validation

of the resampling technique decreases as the amount of data in the subset approaches the amount in the modeling set. A good rule of thumb is about 75–80 %. Higher proportions are a good idea if the number of repetitions is large.

The number of repetitions is important. Increasing the number of subsets has the effect of decreasing the uncertainty of the performance estimates. For example, to get a gross estimate of model performance, 25 repetitions will be adequate if the user is willing to accept some instability in the resulting values. However, to get stable estimates of performance, it is suggested to choose a larger number of repetitions (say 50–200). This is also a function of the proportion of samples being randomly allocated to the prediction set; the larger the percentage, the more repetitions are needed to reduce the uncertainty in the performance estimates.

The Bootstrap

A bootstrap sample is a random sample of the data taken *with replacement* (Efron and Tibshirani 1986). This means that, after a data point is selected for the subset, it is still available for further selection. The bootstrap sample is the same size as the original data set. As a result, some samples will be represented multiple times in the bootstrap sample while others will not be selected at all. The samples not selected are usually referred to as the "out-of-bag" samples. For a given iteration of bootstrap resampling, a model is built on the selected samples and is used to predict the out-of-bag samples (Fig. 4.8).

In general, bootstrap error rates tend to have less uncertainty than k-fold cross-validation (Efron 1983). However, on average, 63.2 % of the data points the bootstrap sample are represented at least once, so this technique has bias

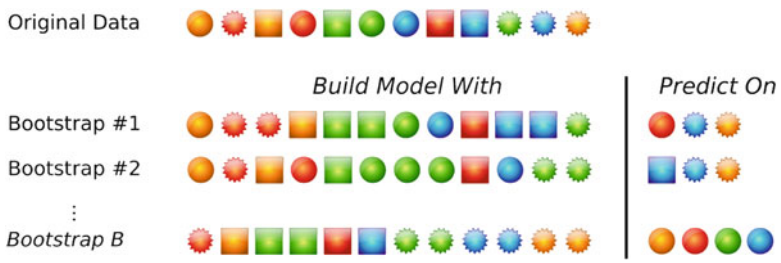

Fig. 4.8: A schematic of bootstrap resampling. Twelve training set samples are represented as symbols and are allocated to B subsets. Each subset is the same size as the original and can contain multiple instances of the same data point. Samples not selected by the bootstrap are predicted and used to estimate model performance

similar to k-fold cross-validation when $k \approx 2$. If the training set size is small, this bias may be problematic, but will decrease as the training set sample size becomes larger.

A few modifications of the simple bootstrap procedure have been devised to eliminate this bias. The "632 method" (Efron 1983) addresses this issue by creating a performance estimate that is a combination of the simple bootstrap estimate and the estimate from re-predicting the training set (e.g., the apparent error rate). For example, if a classification model was characterized by its error rate, the 632 method would use

$$(0.632 \times \text{simple bootstrap estimate}) + (0.368 \times \text{apparent error rate}).$$

The modified bootstrap estimate reduces the bias, but can be unstable with small samples sizes. This estimate can also result in unduly optimistic results when the model severely over-fits the data, since the apparent error rate will be close to zero. Efron and Tibshirani (1997) discuss another technique, called the "632+ method," for adjusting the bootstrap estimates.

4.5 Case Study: Credit Scoring

A straightforward application of predictive models is credit scoring. Existing data can be used to create a model to predict the probability that applicants have good credit. This information can be used to quantify the risk to the lender.

The German credit data set is a popular tool for benchmarking machine learning algorithms. It contains 1,000 samples that have been given labels of good and bad credit. In the data set, 70 % were rated as having good

credit. As discussed in Sect. 11.2, when evaluating the accuracy of a model, the baseline accuracy rate to beat would be 70 % (which we could achieve by simply predicting all samples to have good credit).

Along with these outcomes, data were collected related to credit history, employment, account status, and so on. Some predictors are numeric, such as the loan amount. However, most of the predictors are categorical in nature, such as the purpose of the loan, gender, or marital status. The categorical predictors were converted to "dummy variables" that related to a single category. For example, the applicant's residence information was categorized as either "rent," "own," or "free housing." This predictor would be converted to three yes/no bits of information for each category. For example, one predictor would have a value of one if the applicant rented and is zero otherwise. Creation of dummy variables is discussed at length in Sect. 3.6. In all, there were 41 predictors used to model the credit status of an individual.

We will use these data to demonstrate the process of tuning models using resampling, as defined in Fig. 4.4. For illustration, we took a stratified random sample of 800 customers to use for training models. The remaining samples will be used as a test set to verify performance when a final model is determined. Section 11.2 will discuss the results of the test set in more detail.

4.6 Choosing Final Tuning Parameters

Once model performance has been quantified across sets of tuning parameters, there are several philosophies on how to choose the final settings. The simplest approach is to pick the settings associated with the numerically best performance estimates.

For the credit scoring example, a nonlinear support vector machine model[1] was evaluated over cost values ranging from 2^{-2} to 2^{7}. Each model was evaluated using five repeats of 10-fold cross-validation. Figure 4.9 and Table 4.1 show the accuracy profile across the candidate values of the cost parameter. For each model, cross-validation generated 50 different estimates of the accuracy; the solid points in Fig. 4.9 are the average of these estimates. The bars reflect the average plus/minus two-standard errors of the mean. The profile shows an increase in accuracy until the cost value is one. Models with cost values between 1 and 16 are relatively constant; after which, the accuracy decreases (likely due to over-fitting). The numerically optimal value of the cost parameter is 8, with a corresponding accuracy rate of 75 %. Notice that the apparent accuracy rate, determined by re-predicting the training set samples, indicates that the model improves as the cost is increased, although more complex models over-fit the training set.

[1] This model uses a radial basis function kernel, defined in Sect. 13.4. Although not explored here, we used the analytical approach discussed later for determining the kernel parameter and fixed this value for all resampling techniques.

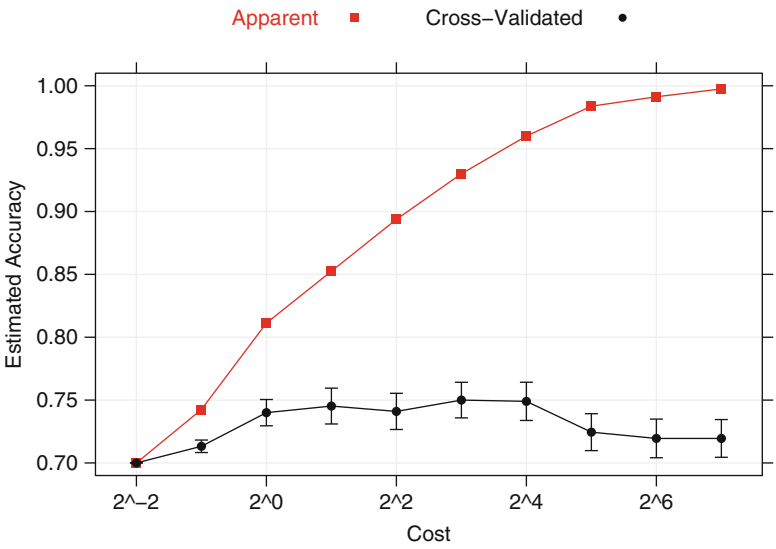

Fig. 4.9: The performance profile of a radial basis function support vector machine for the credit scoring example over different values of the cost parameter. The *vertical lines* indicate ± two-standard errors of the accuracy

In general, it may be a good idea to favor simpler models over more complex ones and choosing the tuning parameters based on the numerically optimal value may lead to models that are overly complicated. Other schemes for choosing less complex models should be investigated as they might lead to simpler models that provide acceptable performance (relative to the numerically optimal settings).

The "one-standard error" method for choosing simpler models finds the numerically optimal value and its corresponding standard error and then seeks the simplest model whose performance is within a single standard error of the numerically best value. This procedure originated with classification and regression trees (Breiman et al. (1984) and Sects. 8.1 and 14.1). In Fig. 4.10, the standard error of the accuracy values when the cost is 8 is about 0.7 %. This technique would find the simplest tuning parameter settings associated with accuracy no less than 74.3 % (75 %–0.7 %). This procedure would choose a value of 2 for the cost parameter.

Another approach is to choose a simpler model that is within a certain tolerance of the numerically best value. The percent decrease in performance could be quantified by $(X - O)/O$ where X is the performance value and O is the numerically optimal value. For example, in Fig. 4.9, the best accuracy value across the profile was 75 %. If a 4 % loss in accuracy was acceptable as a trade-off for a simpler model, accuracy values greater than 71.2 % would

Table 4.1: Repeated cross-validation accuracy results for the support vector machine model

	Resampled accuracy (%)		
Cost	Mean	Std. error	% Tolerance
0.25	70.0	0.0	−6.67
0.50	71.3	0.2	−4.90
1.00	74.0	0.5	−1.33
2.00	74.5	0.7	−0.63
4.00	74.1	0.7	−1.20
8.00	75.0	0.7	0.00
16.00	74.9	0.8	−0.13
32.00	72.5	0.7	−3.40
64.00	72.0	0.8	−4.07
128.00	72.0	0.8	−4.07

The one-standard error rule would select the simplest model with accuracy no less than 74.3 % (75 %–0.7 %). This corresponds to a cost value of 2. The "pick-the-best" solution is shown in bold

be acceptable. For the profile in Fig. 4.9, a cost value of 1 would be chosen using this approach.

As an illustration, additional resampling methods were applied to the same data: repeated 10-fold cross-validation, LOOCV, the bootstrap (with and without the 632 adjustment), and repeated training/test splits (with 20 % held-out). The latter two methods used 50 resamples to estimate performance.

The results are shown in Fig. 4.10. A common pattern within the cross-validation methods is seen where accuracy peaks at cost values between 4 and 16 and stays roughly constant within this window.

In each case, performance rapidly increases with the cost value and then, after the peak, decreases at a slower rate as over-fitting begins to occur. The cross-validation techniques estimate the accuracy to be between 74.5 % and 76.6 %. Compared to the other methods, the simple bootstrap is slightly pessimistic, estimating the accuracy to be 74.2 % while the 632 rule appears to overcompensate for the bias and estimates the accuracy to be 82.3 %. Note that the standard error bands of the simple 10-fold cross-validation technique are larger than the other methods, mostly because the standard error is a function of the number of resamples used (10 versus the 50 used by the bootstrap or repeated splitting).

The computational times varied considerably. The fastest was 10-fold cross-validation, which clocked in at 0.82 min. Repeated cross-validation, the bootstrap, and repeated training-test splits fit the same number of models and, on average, took about 5-fold more time to finish. LOOCV, which fits as many models as there are samples in the training set, took 86-fold longer and should only be considered when the number of samples is very small.

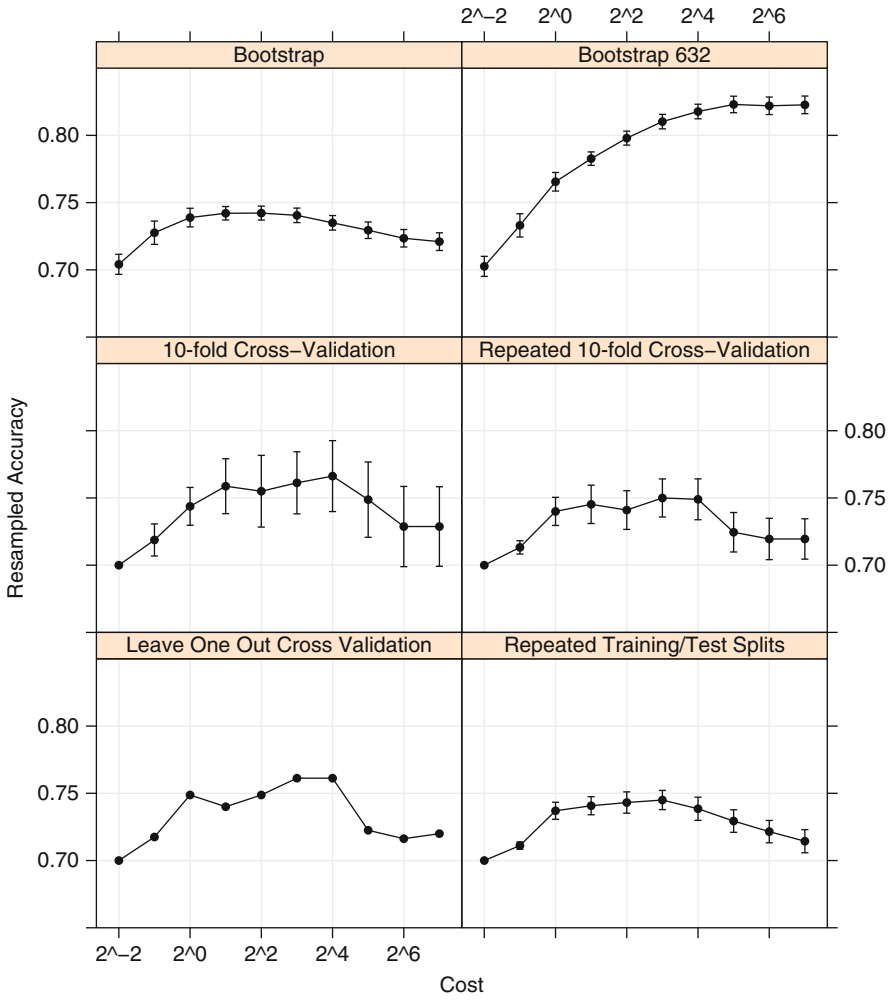

Fig. 4.10: The performance profile of nonlinear support vector machine over different values of the cost parameter for the credit scoring example using several different resampling procedures. The *vertical lines* indicate ± two-standard errors of the accuracy

4.7 Data Splitting Recommendations

As previously discussed, there is a strong technical case to be made against a single, independent test set:

- A test set is a single evaluation of the model and has limited ability to characterize the uncertainty in the results.

- Proportionally large test sets divide the data in a way that increases bias in the performance estimates.
- With small sample sizes:
  - The model may need every possible data point to adequately determine model values.
  - The uncertainty of the test set can be considerably large to the point where different test sets may produce very different results.
- Resampling methods can produce reasonable predictions of how well the model will perform on future samples.

No resampling method is uniformly better than another; the choice should be made while considering several factors. If the samples size is small, we recommend repeated 10-fold cross-validation for several reasons: the bias and variance properties are good and, given the sample size, the computational costs are not large. If the goal is to choose between models, as opposed to getting the best indicator of performance, a strong case can be made for using one of the bootstrap procedures since these have very low variance. For large sample sizes, the differences between resampling methods become less pronounced, and computational efficiency increases in importance. Here, simple 10-fold cross-validation should provide acceptable variance, low bias, and is relatively quick to compute.

Varma and Simon (2006) and Boulesteix and Strobl (2009) note that there is a potential bias that can occur when estimating model performance during parameter tuning. Suppose that the final model is chosen to correspond to the tuning parameter value associated with the smallest error rate. This error rate has the potential to be optimistic since it is a random quantity that is chosen from a potentially large set of tuning parameters. Their research is focused on scenarios with a small number of samples and a large number of predictors, which exacerbates the problem. However, for moderately large training sets, our experience is that this bias is small. In later sections, comparisons are made between resampled estimates of performance and those derived from a test set. For these particular data sets, the *optimization bias* is insubstantial.

4.8 Choosing Between Models

Once the settings for the tuning parameters have been determined for each model, the question remains: how do we choose between multiple models? Again, this largely depends on the characteristics of the data and the type of questions being answered. However, predicting which model is most fit for purpose can be difficult. Given this, we suggest the following scheme for finalizing the type of model:

1. Start with several models that are the least interpretable and most flexible, such as boosted trees or support vector machines. Across many problem domains, these models have a high likelihood of producing the empirically optimum results (i.e., most accurate).
2. Investigate simpler models that are less opaque (e.g., not complete black boxes), such as multivariate adaptive regression splines (MARS), partial least squares, generalized additive models, or naïve Bayes models.
3. Consider using the simplest model that reasonably approximates the performance of the more complex methods.

Using this methodology, the modeler can discover the "performance ceiling" for the data set before settling on a model. In many cases, a range of models will be equivalent in terms of performance so the practitioner can weight the benefits of different methodologies (e.g., computational complexity, easy of prediction, interpretability). For example, a nonlinear support vector machine or random forest model might have superior accuracy, but the complexity and scope of the prediction equation may prohibit exporting the prediction equation to a production system. However, if a more interpretable model, such as a MARS model, yielded similar accuracy, the implementation of the prediction equation would be trivial and would also have superior execution time.

Consider the credit scoring support vector machine classification model that was characterized using resampling in Sect. 4.6. Using repeated 10-fold cross-validation, the accuracy for this model was estimated to be 75 % with most of the resampling results between 66 % and 82 %.

Logistic regression (Sect. 12.2) is a more simplistic technique than the nonlinear support vector machine model for estimating a classification boundary. It has no tuning parameters and its prediction equation is simple and easy to implement using most software. Using the same cross-validation scheme, the estimated accuracy for this model was 74.9 % with most of the resampling results between 66 % and 82 %.

The same 50 resamples were used to evaluate each model. Figure 4.11 uses box plots to illustrate the distribution of the resampled accuracy estimates. Clearly, there is no performance loss by using a more straightforward model for these data.

Hothorn et al. (2005) and Eugster et al. (2008) describe statistical methods for comparing methodologies based on resampling results. Since the accuracies were measured using identically resampled data sets, statistical methods for *paired comparisons* can be used to determine if the differences between models are statistically significant. A paired t-test can be used to evaluate the hypothesis that the models have equivalent accuracies (on average) or, analogously, that the mean difference in accuracy for the resampled data sets is zero. For these two models, the average difference in model accuracy was 0.1 %, with the logistic regression supplying the better results. The 95 % confidence interval for this difference was (−1.2 %, 1 %), indicating that there

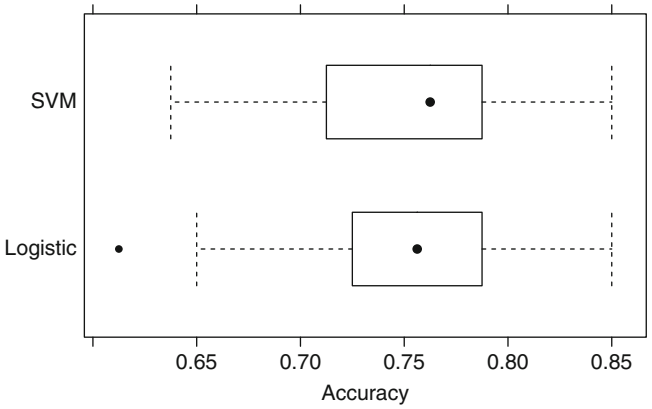

Fig. 4.11: A comparison of the cross-validated accuracy estimates from a support vector machine model and a logistic regression model for the credit scoring data described in Sect. 4.5

is no evidence to support the idea that the accuracy for either model is significantly better. This makes intuitive sense; the resampled accuracies in Fig. 4.11 range from 61.3 % to 85 %; given this amount of variation in the results, a 0.1 % improvement of accuracy is not meaningful.

When a model is characterized in multiple ways, there is a possibility that comparisons between models can lead to different conclusions. For example, if a model is created to predict two classes, sensitivity and specificity may be used to characterize the efficacy of models (see Chap. 11). If the data set includes more events than nonevents, the sensitivity can be estimated with greater precision than the specificity. With increased precision, there is a higher likelihood that models can be differentiated in terms of sensitivity than for specificity.

4.9 Computing

The R language is used to demonstrate modeling techniques. A concise review of R and its basic usage are found in Appendix B. Those new to R should review these materials prior to proceeding. The following sections will reference functions from the AppliedPredictiveModeling, caret, Design, e1071, ipred and MASS packages. Syntax will be demonstrated using the simple two-class example shown in Figs. 4.2 and 4.3 and the data from the credit scoring case study.

Data Splitting

The two-class data shown in Fig. 4.1 are contained in the AppliedPredictiveModeling package and can be obtained using

```
> library(AppliedPredictiveModeling)
> data(twoClassData)
```

The predictors for the example data are stored in a data frame called `predictors`. There are two columns for the predictors and 208 samples in rows. The outcome classes are contained in a factor vector called `classes`.

```
> str(predictors)
  'data.frame':        208 obs. of  2 variables:
   $ PredictorA: num  0.158 0.655 0.706 0.199 0.395 ...
   $ PredictorB: num  0.1609 0.4918 0.6333 0.0881 0.4152 ...
> str(classes)
   Factor w/ 2 levels "Class1","Class2": 2 2 2 2 2 2 2 2 2 2 ...
```

The base R function `sample` can create simple random splits of the data. To create stratified random splits of the data (based on the classes), the `createDataPartition` function in the caret package can be used. The percent of data that will be allocated to the training set should be specified.

```
> # Set the random number seed so we can reproduce the results
> set.seed(1)
> # By default, the numbers are returned as a list. Using
> # list = FALSE, a matrix of row numbers is generated.
> # These samples are allocated to the training set.
> trainingRows <- createDataPartition(classes,
+                                     p = .80,
+                                     list= FALSE)
> head(trainingRows)
      Resample1
  [1,]        99
  [2,]       100
  [3,]       101
  [4,]       102
  [5,]       103
  [6,]       104


> # Subset the data into objects for training using
> # integer sub-setting.
> trainPredictors <- predictors[trainingRows, ]
> trainClasses <- classes[trainingRows]
> # Do the same for the test set using negative integers.
> testPredictors <- predictors[-trainingRows, ]
> testClasses <- classes[-trainingRows]


> str(trainPredictors)
```

```
  'data.frame':        167 obs. of  2 variables:
   $ PredictorA: num  0.226 0.262 0.52 0.577 0.426 ...
   $ PredictorB: num  0.291 0.225 0.547 0.553 0.321 ...
> str(testPredictors)
  'data.frame':        41 obs. of  2 variables:
   $ PredictorA: num  0.0658 0.1056 0.2909 0.4129 0.0472 ...
   $ PredictorB: num  0.1786 0.0801 0.3021 0.2869 0.0414 ...
```

To generate a test set using maximum dissimilarity sampling, the caret function `maxdissim` can be used to sequentially sample the data.

Resampling

The caret package has various functions for data splitting. For example, to use repeated training/test splits, the function `createDataPartition` could be used again with an additional argument named `times` to generate multiple splits.

```
> set.seed(1)
> # For illustration, generate the information needed for three
> # resampled versions of the training set.
> repeatedSplits <- createDataPartition(trainClasses, p = .80,
+                                       times = 3)
> str(repeatedSplits)
  List of 3
   $ Resample1: int [1:135] 1 2 3 4 5 6 7 9 11 12 ...
   $ Resample2: int [1:135] 4 6 7 8 9 10 11 12 13 14 ...
   $ Resample3: int [1:135] 2 3 4 6 7 8 9 10 11 12 ...
```

Similarly, the caret package has functions `createResamples` (for bootstrapping), `createFolds` (for k-old cross-validation) and `createMultiFolds` (for repeated cross-validation). To create indicators for 10-fold cross-validation,

```
> set.seed(1)
> cvSplits <- createFolds(trainClasses, k = 10,
+                         returnTrain = TRUE)
> str(cvSplits)
  List of 10
   $ Fold01: int [1:151] 1 2 3 4 5 6 7 8 9 11 ...
   $ Fold02: int [1:150] 1 2 3 4 5 6 8 9 10 12 ...
   $ Fold03: int [1:150] 1 2 3 4 6 7 8 10 11 13 ...
   $ Fold04: int [1:151] 1 2 3 4 5 6 7 8 9 10 ...
   $ Fold05: int [1:150] 1 2 3 4 5 7 8 9 10 11 ...
   $ Fold06: int [1:150] 2 4 5 6 7 8 9 10 11 12 ...
   $ Fold07: int [1:150] 1 2 3 4 5 6 7 8 9 10 ...
   $ Fold08: int [1:151] 1 2 3 4 5 6 7 8 9 10 ...
   $ Fold09: int [1:150] 1 3 4 5 6 7 9 10 11 12 ...
   $ Fold10: int [1:150] 1 2 3 5 6 7 8 9 10 11 ...
> # Get the first set of row numbers from the list.
> fold1 <- cvSplits[[1]]
```

To get the first 90 % of the data (the first fold):

```
> cvPredictors1 <- trainPredictors[fold1,]
> cvClasses1 <- trainClasses[fold1]
> nrow(trainPredictors)
  [1] 167
> nrow(cvPredictors1)
  [1] 151
```

In practice, functions discussed in the next section can be used to automatically create the resampled data sets, fit the models, and evaluate performance.

Basic Model Building in R

Now that we have training and test sets, we could fit a 5-nearest neighbor classification model (Fig. 4.3) to the training data and use it to predict the test set. There are multiple R functions for building this model: the `knn` function in the MASS package, the `ipredknn` function in the ipred package, and the `knn3` function in caret. The `knn3` function can produce class predictions as well as the proportion of neighbors for each class.

There are two main conventions for specifying models in R: the formula interface and the non-formula (or "matrix") interface. For the former, the predictors are explicitly listed. A basic R formula has two sides: the left-hand side denotes the outcome and the right-hand side describes how the predictors are used. These are separated with a tilde (∼). For example, the formula

```
> modelFunction(price ~ numBedrooms + numBaths + acres,
+               data = housingData)
```

would predict the closing price of a house using three quantitative characteristics. The formula `y ~ .` can be used to indicate that all of the columns in the data set (except `y`) should be used as a predictor. The formula interface has many conveniences. For example, transformations such as `log(acres)` can be specified in-line. Unfortunately, R does not efficiently store the information about the formula. Using this interface with data sets that contain a large number of predictors may unnecessarily slow the computations.

The non-formula interface specifies the predictors for the model using a matrix or data frame (all the predictors in the object are used in the model). The outcome data are usually passed into the model as a vector object. For example,

```
> modelFunction(x = housePredictors, y = price)
```

Note that not all R functions have both interfaces.

For knn3, we can estimate the 5-nearest neighbor model with

```
> trainPredictors <- as.matrix(trainPredictors)
> knnFit <- knn3(x = trainPredictors, y = trainClasses, k = 5)
> knnFit
  5-nearest neighbor classification model

  Call:
  knn3.matrix(x = trainPredictors, y = trainClasses, k = 5)

  Training set class distribution:

  Class1 Class2
      89     78
```

At this point, the knn3 object is ready to predict new samples. To assign new samples to classes, the predict method is used with the model object. The standard convention is

```
> testPredictions <- predict(knnFit, newdata = testPredictors,
+                            type = "class")
> head(testPredictions)
  [1] Class2 Class2 Class1 Class1 Class2 Class2
  Levels: Class1 Class2
> str(testPredictions)
   Factor w/ 2 levels "Class1","Class2": 2 2 1 1 2 2 2 2 2 2 ...
```

The value of the type argument varies across different modeling functions.

Determination of Tuning Parameters

To choose tuning parameters using resampling, sets of candidate values are evaluated using different resamples of the data. A profile can be created to understand the relationship between performance and the parameter values. R has several functions and packages for this task. The e1071 package contains the tune function, which can evaluate four types of models across a range of parameters. Similarly, the errorest function in the ipred package can resample single models. The train function in the caret package has built-in modules for 144 models and includes capabilities for different resampling methods, performances measures, and algorithms for choosing the best model from the profile. This function also has capabilities for parallel processing so that the resampled model fits can be executed across multiple computers or processors. Our focus will be on the train function.

Section 4.6 illustrated parameter tuning for a support vector machine using the credit scoring data. Using resampling, a value of the cost parameter was estimated. As discussed in later chapters, the SVM model is characterized

by what type of *kernel function* the model uses. For example, the linear kernel function specifies a linear relationship between the predictors and the outcome. For the credit scoring data, a radial basis function (RBF) kernel function was used. This kernel function has an additional tuning parameter associated with it denoted as σ, which impacts the smoothness of the decision boundary. Normally, several combinations of both tuning parameters would be evaluated using resampling. However, Caputo et al. (2002) describe an analytical formula that can be used to get reasonable estimates of σ. The caret function `train` uses this approach to estimate the kernel parameter, leaving only the cost parameter for tuning.

To tune an SVM model using the credit scoring training set samples, the `train` function can be used. Both the training set predictors and outcome are contained in an R data frame called `GermanCreditTrain`.

```
> library(caret)
> data(GermanCredit)
```

The `chapters` directory of the AppliedPredictiveModeling package contains the code for creating the training and test sets. These data sets are contained in the data frames `GermanCreditTrain` and `GermanCreditTest`, respectively.

We will use all the predictors to model the outcome. To do this, we use the formula interface with the formula `Class` $\sim$ `.` the classes are stored in the data frame column called `class`. The most basic function call would be

```
> set.seed(1056)
> svmFit <- train(Class ~ .,
>                 data = GermanCreditTrain,
>                 # The "method" argument indicates the model type.
>                 # See ?train for a list of available models.
>                 method = "svmRadial")
```

However, we would like to tailor the computations by overriding several of the default values. First, we would like to pre-process the predictor data by centering and scaling their values. To do this, the `preProc` argument can be used:

```
> set.seed(1056)
> svmFit <- train(Class ~ .,
>                 data = GermanCreditTrain,
>                 method = "svmRadial",
>                 preProc = c("center", "scale"))
```

Also, for this function, the user can specify the exact cost values to investigate. In addition, the function has algorithms to determine reasonable values for many models. Using the option `tuneLength = 10`, the cost values 2^{-2}, 2^{-2} $\ldots 2^{7}$ are evaluated.

```
> set.seed(1056)
> svmFit <- train(Class ~ .,
>                 data = GermanCreditTrain,
>                 method = "svmRadial",
>                 preProc = c("center", "scale"),
>                 tuneLength = 10)
```

By default, the basic bootstrap will be used to calculate performance measures. Repeated 10-fold cross-validation can be specified with the `trainControl` function. The final syntax is then

```
> set.seed(1056)
> svmFit <- train(Class ~ .,
>                 data = GermanCreditTrain,
>                 method = "svmRadial",
>                 preProc = c("center", "scale"),
>                 tuneLength = 10,
>                 trControl = trainControl(method = "repeatedcv",
>                                          repeats = 5,
>                                          classProbs = TRUE))
```

```
> svmFit
800 samples
 41 predictors
  2 classes: 'Bad', 'Good'

Pre-processing: centered, scaled
Resampling: Cross-Validation (10-fold, repeated 5 times)

Summary of sample sizes: 720, 720, 720, 720, 720, 720, ...

Resampling results across tuning parameters:

  C     Accuracy  Kappa  Accuracy SD  Kappa SD
  0.25  0.7       0      0            0
  0.5   0.724     0.141  0.0218       0.0752
  1     0.75      0.326  0.0385       0.106
  2     0.75      0.363  0.0404       0.0984
  4     0.754     0.39   0.0359       0.0857
  8     0.738     0.361  0.0404       0.0887
  16    0.738     0.361  0.0458       0.1
  32    0.732     0.35   0.043        0.0928
  64    0.732     0.352  0.0453       0.0961
  128   0.731     0.349  0.0451       0.0936

Tuning parameter 'sigma' was held constant at a value of 0.0202
Accuracy was used to select the optimal model using  the largest value.
The final values used for the model were C = 4 and sigma = 0.0202.
```

A different random number seed and set of cost values were used in the original analysis, so the results are not exactly the same as those shown in Sect. 4.6. Using a "pick the best" approach, a final model was fit to all 800 training set samples with a σ value of 0.0202 and a cost value of 4. The `plot` method can be used to visualize the performance profile. Figure 4.12 shows an example visualization created from the syntax

```
> # A line plot of the average performance
> plot(svmFit, scales = list(x = list(log = 2)))
```

To predict new samples with this model, the `predict` method is called

```
> predictedClasses <- predict(svmFit, GermanCreditTest)
> str(predictedClasses)
```

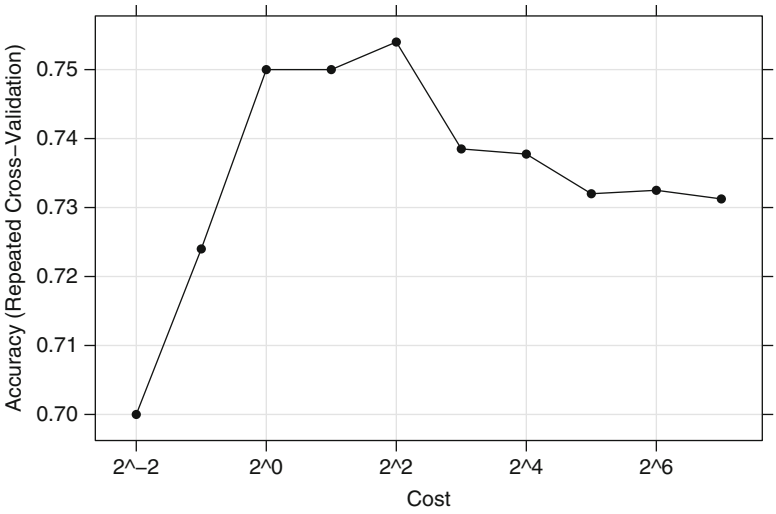

Fig. 4.12: A visualization of the average performance profile of an SVM classification model produced from the `plot` method for the `train` class

```
   Factor w/ 2 levels "Bad","Good": 1 1 2 2 1 2 2 2 1 1 ...

> # Use the "type" option to get class probabilities
> predictedProbs <- predict(svmFit, newdata = GermanCreditTest,
+                           type = "prob")
> head(predictedProbs)
          Bad      Good
  1 0.5351870 0.4648130
  2 0.5084049 0.4915951
  3 0.3377344 0.6622656
  4 0.1092243 0.8907757
  5 0.6024404 0.3975596
  6 0.1339467 0.8660533
```

There are other R packages that can estimate performance via resampling. The `validate` function in the Design package and the `errorest` function in the ipred package can be used to estimate performance for a model with a single candidate set of tuning parameters. The `tune` function of the e1071 package can also determine parameter settings using resampling.

Between-Model Comparisons

In Sect. 4.6, the SVM model was contrasted with a logistic regression model. While basic logistic regression has no tuning parameters, resampling can still be used to characterize the performance of the model. The `train` function is

once again used, with a different `method` argument of `"glm"` (for generalized linear models). The same resampling specification is used and, since the random number seed is set prior to modeling, the resamples are exactly the same as those in the SVM model.

```
> set.seed(1056)
> logisticReg <- train(Class ~ .,
+                      data = GermanCreditTrain,
+                      method = "glm",
+                      trControl = trainControl(method = "repeatedcv",
+                                               repeats = 5))
```

```
> logisticReg
```

```
800 samples
 41 predictors
  2 classes: 'Bad', 'Good'

No pre-processing
Resampling: Cross-Validation (10-fold, repeated 5 times)

Summary of sample sizes: 720, 720, 720, 720, 720, 720, ...

Resampling results

  Accuracy  Kappa  Accuracy SD  Kappa SD
  0.749     0.365  0.0516       0.122
```

To compare these two models based on their cross-validation statistics, the `resamples` function can be used with models that share a common set of resampled data sets. Since the random number seed was initialized prior to running the SVM and logistic models, paired accuracy measurements exist for each data set. First, we create a `resamples` object from the models:

```
> resamp <- resamples(list(SVM = svmFit, Logistic = logisticReg))
> summary(resamp)
```

```
Call:
summary.resamples(object = resamp)

Models: SVM, Logistic
Number of resamples: 50

Accuracy
           Min. 1st Qu. Median  Mean 3rd Qu. Max. NA's
SVM      0.6500  0.7375 0.7500 0.754  0.7625 0.85    0
Logistic 0.6125  0.7250 0.7562 0.749  0.7844 0.85    0

Kappa
             Min. 1st Qu. Median   Mean 3rd Qu.   Max. NA's
SVM      0.18920  0.3519 0.3902 0.3897  0.4252 0.5946    0
Logistic 0.07534  0.2831 0.3750 0.3648  0.4504 0.6250    0
```

The summary indicates that the performance distributions are very similar. The `NA` column corresponds to cases where the resampled models failed (usually due to numerical issues). The `resamples` class has several methods for visualizing the paired values (see `?xyplot.resamples` for a list of plot types). To assess possible differences between the models, the `diff` method is used:

```
> modelDifferences <- diff(resamp)
> summary(modelDifferences)

  Call:
  summary.diff.resamples(object = modelDifferences)

  p-value adjustment: bonferroni
  Upper diagonal: estimates of the difference
  Lower diagonal: p-value for H0: difference = 0

  Accuracy
            SVM    Logistic
  SVM              0.005
  Logistic 0.5921

  Kappa
            SVM    Logistic
  SVM              0.02498
  Logistic 0.2687
```

The p-values for the model comparisons are large (0.592 for accuracy and 0.269 for Kappa), which indicates that the models fail to show any difference in performance.

Exercises

4.1. Consider the music genre data set described in Sect. 1.4. The objective for these data is to use the predictors to classify music samples into the appropriate music genre.

(a) What data splitting method(s) would you use for these data? Explain.
(b) Using tools described in this chapter, provide code for implementing your approach(es).

4.2. Consider the permeability data set described in Sect. 1.4. The objective for these data is to use the predictors to model compounds' permeability.

(a) What data splitting method(s) would you use for these data? Explain.
(b) Using tools described in this chapter, provide code for implementing your approach(es).

4.3. Partial least squares (Sect. 6.3) was used to model the yield of a chemical manufacturing process (Sect. 1.4). The data can be found in the AppliedPredictiveModeling package and can be loaded using

	Resampled R^2	
Components	Mean	Std. Error
1	0.444	0.0272
2	0.500	0.0298
3	0.533	0.0302
4	0.545	0.0308
5	0.542	0.0322
6	0.537	0.0327
7	0.534	0.0333
8	0.534	0.0330
9	0.520	0.0326
10	0.507	0.0324

```
> library(AppliedPredictiveModeling)
> data(ChemicalManufacturingProcess)
```

The objective of this analysis is to find the number of PLS components that yields the optimal R^2 value (Sect. 5.1). PLS models with 1 through 10 components were each evaluated using five repeats of 10-fold cross-validation and the results are presented in the following table:

(a) Using the "one-standard error" method, what number of PLS components provides the most parsimonious model?
(b) Compute the tolerance values for this example. If a 10 % loss in R^2 is acceptable, then what is the optimal number of PLS components?
(c) Several other models (discussed in Part II) with varying degrees of complexity were trained and tuned and the results are presented in Fig. 4.13. If the goal is to select the model that optimizes R^2, then which model(s) would you choose, and why?
(d) Prediction time, as well as model complexity (Sect. 4.8) are other factors to consider when selecting the optimal model(s). Given each model's prediction time, model complexity, and R^2 estimates, which model(s) would you choose, and why?

4.4. Brodnjak-Vonina et al. (2005) develop a methodology for food laboratories to determine the type of oil from a sample. In their procedure, they used a gas chromatograph (an instrument that separate chemicals in a sample) to measure seven different fatty acids in an oil. These measurements would then be used to predict the type of oil in a food samples. To create their model, they used 96 samples[2] of seven types of oils.

These data can be found in the caret package using `data(oil)`. The oil types are contained in a factor variable called `oilType`. The types are pumpkin

[2] The authors state that there are 95 samples of known oils. However, we count 96 in their Table 1 (pp. 33–35 of the article).

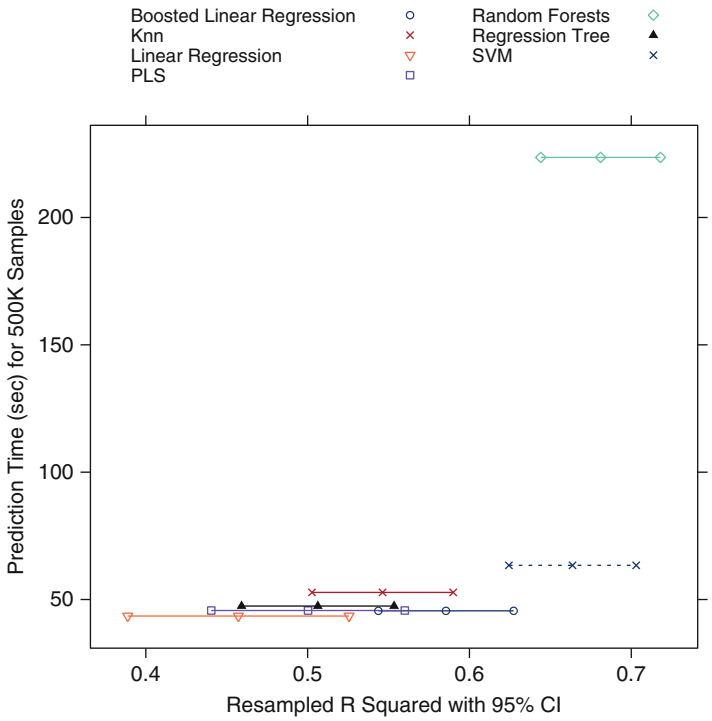

Fig. 4.13: A plot of the estimated model performance against the time to predict 500,000 new samples using the chemical manufacturing data

(coded as `A`), sunflower (`B`), peanut (`C`), olive (`D`), soybean (`E`), rapeseed (`F`) and corn (`G`). In R,

```
> data(oil)
> str(oilType)
   Factor w/ 7 levels "A","B","C","D",..: 1 1 1 1 1 1 1 1 1 1 ...
> table(oilType)

oilType
 A  B  C  D  E  F  G
37 26  3  7 11 10  2
```

(a) Use the `sample` function in base R to create a completely random sample of 60 oils. How closely do the frequencies of the random sample match the original samples? Repeat this procedure several times of understand the variation in the sampling process.

(b) Use the caret package function `createDataPartition` to create a stratified random sample. How does this compare to the completely random samples?

(c) With such a small samples size, what are the options for determining performance of the model? Should a test set be used?

(d) One method for understanding the uncertainty of a test set is to use a confidence interval. To obtain a confidence interval for the overall accuracy, the based R function `binom.test` can be used. It requires the user to input the number of samples and the number correctly classified to calculate the interval. For example, suppose a test set sample of 20 oil samples was set aside and 76 were used for model training. For this test set size and a model that is about 80 % accurate (16 out of 20 correct), the confidence interval would be computed using

```
> binom.test(16, 20)

        Exact binomial test

data:  16 and 20
number of successes = 16, number of trials = 20, p-value = 0.01182
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.563386 0.942666
sample estimates:
probability of success
                   0.8
```

In this case, the width of the 95 % confidence interval is 37.9 %. Try different samples sizes and accuracy rates to understand the trade-off between the uncertainty in the results, the model performance, and the test set size.