than individually if they are all sufficiently good and sufficiently different from one another. These conditions create space for improvement by reinforcing strengths and compensating weaknesses (rather than compensating both, reinforcing both, or reinforcing weaknesses and compensating strengths).

As an elementary illustration, consider three classification models h_1, h_2, h_3, with the corresponding true misclassification error values $e_c(h_1)$, $e_c(h_2)$, $e_c(h_3)$ with respect to a target concept c. As explained in Section 7.2.1, these are the probabilities that the corresponding models would produce incorrect predictions for a randomly chosen instance from the domain. Now consider a simple combined model h_* that aggregates the predictions of base models h_1, h_2, h_3 by voting. Assuming a two-class classification task, the true error of this model is then the probability that majority of base models (i.e., two or three in our case) make mistakes, i.e.,

$$\begin{aligned} e_c(h_*) = {} & e_c(h_1)e_c(h_2)e_c(h_3) \\ & + (1 - e_c(h_1))e_c(h_2)e_c(h_3) + e_c(h_1)(1 - e_c(h_2))e_c(h_3) \\ & + e_c(h_1)e_c(h_2)(1 - ee_c(h_3)) \end{aligned} \tag{15.1}$$

Let us assume that all base models are sufficiently good and different from one another. The former may be represented by setting an upper bound ϵ for their error values and the latter –in the unrealistically idealized case – by considering their mistakes independent. Under these assumptions the above error may be bound as follows:

$$e_c(h_*) \leq \epsilon^3 + 3\epsilon^2(1 - \epsilon) \tag{15.2}$$

What one might be interested to see is how this compares to the base model error bound ϵ. The corresponding inequality

$$\epsilon^3 + 3\epsilon^2(1 - \epsilon) < \epsilon \tag{15.3}$$

may be easily solved, yielding $0 < \epsilon < 0.5$. This is a pretty weak requirement, which means (with the two-class assumption) that, for the ensemble to give an improvement, base models have to perform better than random (although not perfectly, as the latter clearly would leave no space for improvement). Thus, with just three base models, if they are just minimally reasonable, but fully independent, a simple voting-based combined model will perform better.

While the discussion above is an illustrative special case rather than a general argument, it at least demonstrates that the expectation of improved prediction performance by ensemble modeling is justified and which are the conditions necessary to actually make it happen. Its main limitation is not the small number of base models, since adding more models – which makes the error more complex to calculate – may only improve the prediction quality. It is the idealized assumption of model mistake independence that makes the derivation of error bounds practically inapplicable. It is still useful as a source of insights, though. In practice, creating multiple totally independent models for the same domain may be next to impossible, but it remains possible and worthwhile to approximate this ideal situation with models that are as diverse as possible, without sacrificing too much of their quality. The more such reasonable quality base models are available to combine and more diverse they are, more prediction quality improvement may be expected from the resulting model ensemble. While the actual results may also differ significantly depending on the particular model aggregation method, at least the available improvement potential depends on the base model portfolio.

15.3 Base models

As discussed above, the main challenge for successful ensemble modeling is creating sufficiently many sufficiently diverse and sufficiently good base models. Since any deterministic algorithm will yield the same model when applied to the same training set with the same parameter setup, the following approaches to ensuring diversity may be considered:

Different training sets. Use a different training set from the same domain to create each base model.

Different algorithms. Use a different algorithm to create each base model.

Different parameter setups. Use a different algorithm parameter setup to create each base model.

Algorithm randomization. Use independent runs of a nondeterministic algorithm to create each base model.

For reinforced effect, two or more of these approaches can also be applied in combination. Each of them must be used with care, though, as pressing too much on the diversity of base models may ruin their quality, which must remain at some reasonable level.

15.3.1 Different training sets

The most popular approach to creating multiple base models relies on the assumption that applying the same algorithm to different training sets for the same task from the same domain will yield models that are sufficiently diverse and sufficiently good at the same time. Ideally, we should be able to draw these training sets from the domain independently at random. In practice, no direct domain sampling is possible, though, and a number of different training sets may only be obtained by sampling or transforming the original training set supplied for the task at hand. This may include:

- sampling instances,
- replicating instances,
- varying instance weights (for weight-sensitive algorithms),
- sampling attributes,
- applying attribute transformations.

15.3.1.1 Instance sampling

Instance sampling is typically performed by drawing multiple bootstrap samples $T_1, T_2, \ldots, T_m$ of the original training set T, i.e., uniform random samples with replacement, usually of the same size as the former. As demonstrated in Section 7.3.6, when discussing bootstrapping as a model evaluation procedure, such a bootstrap sample may be expected to contain about 63.2% of instances from T. Each sample T_i is used to create a base model h_i using the same modeling algorithm.

For instance sampling to be successful in delivering diverse models, the latter should be created by an *unstable* algorithm, i.e., highly sensitive even to minor data variations. Decision and regression trees are the most obvious and natural candidates, since their split selection,

stop, and pruning criteria may yield different outcomes for slightly different datasets, resulting in different models being obtained. On the other hand, modeling algorithms that use the training data to estimate numerical parameters representing their models rather than to make discrete decisions, such as linear and other parametric models or the naïve Bayes classifier, do not react excessively to data perturbations and are considered stable.

Example 15.3.1 The following R code demonstrates the instance sampling approach to base model generation. The `base.ensemble.sample.x` function applies the specified algorithm to samples drawn from the provided dataset, using the standard `sample` function. Its default settings produce bootstrap samples of the same size as the original dataset. The function is applied to create 50 base decision tree and naïve Bayes models for the *HouseVotes84* data, as well as 50 base regression tree and linear regression models for the *Boston Housing* data. As an indirect and rough means of assessing the diversity of the base models obtained by instance sampling, their training and test set misclassification error or mean square error values are determined. Test set errors are of particular interest, since – while even substantially different models can be similarly good on the training set, they are more likely to differ with respect to their performance on previously unseen data.

One specific example of attribute transformation that is particularly relevant to ensemble modeling is multiclass encoding, normally used to handle more than two classes with algorithms capable of delivering two-class classification models only. As extensively discussed and demonstrated in Section 17.4, multiclass encoding techniques create and combine multiple two-class models, and therefore can also be viewed as ensemble modeling techniques.

15.3.2 Different algorithms

The approach of using different algorithms to create base models assumes that the very same training set is passed to a number of modeling algorithms that hopefully produce sufficiently good and sufficiently diverse base models. This rarely makes it possible to create more than a few or a dozen base models, as this is how many algorithms for the same modeling task are typically available in analytic toolboxes. This limits the utility of this approach, at least in its pure form. It may become more attractive, though, in combination with the two related techniques discussed below.

15.3.3 Different parameter setups

The same algorithm may sometimes deliver substantially different models based on the same data if used with different parameter setups. The corresponding approach to base model creation makes sense for modeling algorithms that have parameters altering sufficiently important aspects of their operation to yield diverse models. These could be, e.g., different split selection or pruning criteria for decision or regression trees or different kernel functions for support vector machines and support vector regression modeling algorithms that will be presented in Chapter 16. This technique alone does not usually make it possible to generate a large number of different base models and is of limited usefulness.

15.3.4 Algorithm randomization

The same algorithm with the same parameter setup may yield different models for the same dataset if some of its processing steps are nondeterministic. While some algorithms may be nondeterministic by nature, it is much more common and useful to deliberately randomize deterministic (or nearly deterministic) algorithms.

Algorithm randomization consists in incorporating a nondeterministic modification to the standard algorithm operation that does not degrade model quality too severely, but makes different algorithm invocations likely to produce noticeably different models. The choice of algorithm steps to modify and the exact modifications is obviously algorithm specific. For algorithms that make internal decisions using certain criteria, based on evaluations of multiple possible candidate decisions, it usually makes sense to randomize such decision-making steps, e.g., by adding random noise to decision evaluations or randomly sampling the space of candidate decisions. In particular, for decision or regression tree growing the split selection operation is the natural candidate for randomization. It can be achieved either by randomly disturbing the split evaluation function, or limiting the set of candidate splits to consider at a given node to a randomly selected subset of all available splits or attributes. The latter resembles attribute sampling, but here an independent sample of attributes is drawn in each node instead of having a single fixed attribute sample for all nodes, as with the latter.

Notice that algorithm randomization can be easily applied in combination with any of the training set modification techniques for base model creation, such as instance sampling or

Decision tree growing randomization applied to the *HouseVotes84* data appears to give little effect when looking at training set error distribution, but this is to be expected given their default strict stop criteria leading to fitting the training set exactly. On the test set they exhibit randomized trees substantially more variability than those obtained using instance sampling, with somewhat reduced accuracy. Similar observations can be made for randomized regression trees on the *Boston Housing* data.

15.3.5 Base model diversity

While all the base model generation techniques presented in this section serve the same purpose of generating multiple diverse, but at the same time reasonably good, models for the same domain, they are not equally effective in achieving this goal. Instance sampling has the widest applicability and should be capable of delivering substantial diversity if used with an unstable modeling algorithm. Instance weighting makes most sense when it is desirable to somehow adjust subsequent base models to the performance exhibited by those created previously. Otherwise it offers no advantages over instance sampling. Attribute sampling is only applicable when there are sufficiently many attributes. Otherwise eliminating some of them may excessively degrade base model performance. Varying algorithm parameters may be effective only for some algorithms that are sufficiently sensitive to their parameter values. It has to be used with care to avoid destroying base model quality. Usually only a small number of different but sufficiently good base models can be created using this technique alone and it can be truly useful only in combination with one of the others, usually instance sampling. Algorithm randomization, whenever applicable, is easier to control and has much greater base model diversity potential. It can be applied as a standalone base model generation technique or as a diversity-stimulating companion to instance sampling.

Example 15.3.5 To illustrate the base model diversity potential of the techniques presented in this section, the following R code produces boxplots visualizing the training and test set performance of the base models created in the previous examples.

The obtained boxplots are presented in Figures 15.1 and 15.2. Clearly attribute sampling yields the most diverse base models (judging based on their test set performance), but this is at the cost of considerably worse error levels. Of the remaining techniques, instance sampling and algorithm randomization appear to offer acceptable levels of tradeoff between diversity and quality.

The barplots illustrating the performance of the created model ensembles are presented in Figure 15.3. Only one of the decision tree ensembles, the one using base models obtained by instance sampling, outperforms the single decision tree for the *HouseVotes84* data. The ensembles with base models obtained by attribute sampling are particularly poor, suggesting that this base model generation methods may be not very useful if used alone. No improvement can be observed for the naïve Bayes models. Somewhat better results are obtained for the *Boston Housing* dataset, with most regression tree ensembles (except that using base models obtained by attribute sampling) outperforming the single tree. The ensemble consisting of randomized regression trees turns out particularly successful. None of linear model ensembles brings any improvement, though. This is because the averaged predictions of multiple linear models remain linear, i.e., they could have been generated by a single linear model.

15.4.2 Probability averaging

For probabilistic classification models, generating class probabilities rather than or apart from class labels, an alternative probabilistic prediction combination scheme is possible. Let $P_{h_i}(d|x)$ denote the probability of class d for instance x delivered by base model h_i. Then the combined probability of class d for instance x is calculated by probability averaging as follows:

$$P_{h_*}(d|x) = \sum_{i=1}^{m} P(h_i) P_{h_i}(d|x) \tag{15.6}$$

where $P(h_i)$ is the probability of model h_i in the ensemble, assumed to be $\frac{1}{m}$ for all $i = 1, 2, \ldots, m$. This preserves the probability prediction capability of base models in the ensemble, with all the related advantages, including the possibility of misclassification cost minimization, as discussed in Section 6.3.3, or operating point tuning by ROC analysis, as discussed in Section 7.2.5. If these are not needed, class label predictions can be generated by simple probability maximization.

As a matter of fact, probabilistic predictions are possible even with model ensembles comprising nonprobabilistic base models, by taking

$$P_{h_i}(d|x) = \begin{cases} 1 & \text{if } h_i(x) = d \\ 0 & \text{otherwise} \end{cases} \tag{15.7}$$

The predicted probability of each class becomes then the number of votes for this class divided by the number of base models. Even though the quality of such probability predictions is likely to be inferior to that possible with proper probabilistic models, it may still be useful.

Example 15.4.2 The R code presented below defines the `predict.ensemble.prob` function that performs base classification model aggregation by probability averaging. The prediction function for base models passed via the `prob` argument is expected to produce class probability predictions. Such probabilities obtained for all base models are averaged and either returned directly, if the `prob` argument is set to `TRUE`, or used to assign maximum-probability class labels. The latter is the default behavior demonstrated by example calls which combine all the previously created decision tree and naïve Bayes base models for the *HouseVotes84* data, except those obtained using decision tree randomization, since its simple implementation lacks the probabilistic prediction functionality.

15.4.3 Weighted voting/averaging

It may be sometimes a good idea to weight base models depending on their training set performance or estimated true performance, with a weighting scheme that allows better models to have more prediction impact. Incorporating model weights W_i for each base model h_i leads to the following prediction combination schemes:

$$h_*(x) = \arg\max_{d \in C} \sum_{i=1}^{n} W_i \mathbb{I}_{h_i(x)=d} \tag{15.8}$$

for classification, and

$$h_*(x) = \frac{\sum_{i=1}^{m} W_i h_i(x)}{\sum_{i=1}^{m} W_i} \tag{15.9}$$

for regression, where the uppercase W_i is used to avoid confusion of model weights with instance weights w_x, also referred to in this chapter. Sometimes it may be more convenient and natural to use the weighted sum rather than the weighted average:

$$h_*(x) = \sum_{i=1}^{m} W_i h_i(x) \tag{15.10}$$

which is clearly the same if model weights are normalized to sum up to 1. Finally, the weighted version of class probability averaging is defined as follows:

$$P_{h_*}(d|x) = \frac{\sum_{i=1}^{m} W_i P(h_i) P_{h_i}(d|x)}{\sum_{i=1}^{m} W_i P(h_i)} \tag{15.11}$$

which can further be simplified, if all base models are assumed to have the same probability of $\frac{1}{m}$, to the following form:

$$P_{h_*}(d|x) = \frac{\sum_{i=1}^{m} W_i P_{h_i}(d|x)}{\sum_{i=1}^{m} W_i} \tag{15.12}$$

Model weighting schemes are usually specific to particular ensemble modeling techniques.

Example 15.4.3 Weighted voting/averaging model combination is implemented by the `predict.ensemble.weighted` function defined by the following R code. Weighted voting is performed using the `weighted.modal` function, and weighted averaging using the standard `weighted.mean` function. Optionally, summing may be requested instead of averaging. The `predict.ensemble.weighted` function is demonstrated by applying it to combine all the base models generated in

Ex. 2.4.20
dmr.stats

the previous examples. A straightforward inverse error model weighting scheme is employed. As before, the misclassification error or mean square error values on the test sets are calculated and plotted for the combined models.

The barplots illustrating the performance of the created model ensembles are presented in Figure 15.5. There is no significant impact of base model weighting on the prediction quality of most of the model ensembles, which remains the same as with basic voting/averaging. Only for the worst attribute sampling ensembles some improvement due to weighting may be observed. The applied weighting scheme may not sufficiently vary the contributions of better and worse base models, or the quality of base models may not sufficiently differ.

15.4.4 Using as attributes

A more refined approach than plain or weighted voting or averaging consists in using a modeling algorithm to create the aggregated model h_*, with base models $h_1, h_2, \ldots, h_m$ playing the role of (the only or additional) attributes. Technically, this means that their predictions for the training set are generated and used instead of or apart form the original attribute values. Such data is passed to the modeling algorithm used to create the aggregated model, which may, but does not have to, be the same as (possibly one of those) used for base model creation. It is more common to use rather simple algorithms for model combination, but more refined ones for base model generation.

It is also possible to consider multiple levels of such model aggregation, leading to a hierarchical model ensemble. In this approach, base models created using the original set of attributes, that may be referred to as level 0 models, are used as attributes to create multiple level 1 models, which then in turn are used to create level 2 models, etc. The same techniques for base model creation as discussed above may be used on each level to obtain multiple

aggregated models – what changes is only the set of attributes used, which constitutes of or is supplemented by the lower level models.

Despite its refinement and conceptual elegance, the approach of using base models as attributes for model aggregation is not necessarily superior to simple voting or averaging. Relationships between particular base models and the target attribute may not be predictively useful enough to outperform the latter. This is not to say that this aggregation technique is universally poor and useless, but rather warn that it is not necessarily superior to the much simpler alternatives discussed previously.

Unlike basic or weighted voting/averaging, using base models as attributes means that an actual representation of the combined model is created. The training set therefore needs to remain available in the model combination phase.

Example 15.4.4 Base model combination by using them as attributes is implemented and demonstrated by the R code. The implementation comprises two functions, `combine.ensemble.attributes` and `predict.ensemble.attributes`. The former creates the combined model using the training set, with base models used instead of or apart from the original attributes (depending on the value of the `append` argument). The latter applies such a combined model for prediction. The presented implementation assumes that the target attribute name is available in the `terms` component of the model object structure used to represent base models. This is true for some, but not all modeling algorithms available in R – in particular, for the `rpart` and `lm` models, but neither for the `naiveBayes` model nor for the randomized decision and regression trees created by the `grow.randdectree` and `grow.randregtree` functions. The demonstrations presented below are actually limited to combining `rpart` decision tree and regression tree models only, using the naïve Bayes and linear regression algorithms on the second level.

15.5 Specific ensemble modeling algorithms

Various combinations of all the possible approaches to base model creation and aggregation discussed above may be used, yielding a variety of ensemble modeling techniques. Some of them have proved particularly useful and become extremely popular. These most noteworthy specific instantiations of model ensembles are overviewed in this section.

15.5.1 Bagging

Bagging (standing for *bootstrap aggregating*) is definitely the simplest ensemble modeling algorithm that combines the very basic approaches to base model creation and aggregation:

- base models are created using bootstrap samples of the training set,
- combined by plain (unweighted) voting for the classification task or averaging for the regression task.

If using probabilistic base classification models, class label voting can be replaced by class probability averaging, leading to a probabilistic version of bagging.

This technique may not promise extreme prediction quality, but is likely to give an improvement compared to single models created using the same algorithm as base models, as long as the algorithm is unstable. For stable algorithms, with base models not sufficiently diverse, there may be no improvement or even minor degradation of prediction quality. There are no particular requirements for the modeling algorithm other than instability. Actually, it may be simplified compared to what would be normally used for single model creation, if this makes it more unstable. This may include, in particular, giving up overfitting precautions used in some algorithms, such as pruning decision or regression trees. Models overfitted to their particular bootstrap samples are more likely to differ. The overfitting of base models will not entail the overfitting of the ensemble, as their aggregation will effectively cancel it out. Similarly, there is usually no need to bother with attribute selection, as more attributes provide more opportunities to create many diverse models. This is a striking difference compared to what is typical when single models are created.

Bagging may be thought of as a means of stabilizing unstable algorithms. Single models obtained using such algorithms may be subject to considerable variation depending on a particular training set. There is always a possibility that for a slightly different training set a better or worse model would be created. Creating multiple models based on different data samples without combining them into an ensemble, and simply selecting one of them that appears the best does not provide a valuable solution. This is because model selection would have to based on model evaluation and the latter, as discussed in Section 7.3.1, only makes sense for a repeatable modeling procedure. In particular, producing low-variance performance estimates that could serve for model selection requires repeating training and evaluation cycles multiple times. This is completely impossible for models that only differ in their training samples.

Bagging, with sufficiently many base models, allows one to be pretty confident that the final model is at least as good as a single model in the optimistic case, and possibly even improve over that. This is enough to justify the use of bagging if computational resources permit creating dozens or more models, as typically used for this technique and if model

human readability is not required. The bagging ensemble performance tends to improve with increasing the number of base models up to a certain point, after which it stabilizes. This is where the limit of model diversity possible using bootstrap samples is achieved. Additional models are too much similar to the other ones to make any difference.

Example 15.5.1 The bagging ensemble modeling technique is implemented and demonstrated by the R code presented below. Since bagging is the most straightforward combination of instance sampling for base model creation and voting/averaging for ensemble prediction, the corresponding functions are simple wrappers around the functions defined in Examples 15.3.1 and 15.4.1. The demonstrations also follow the same pattern and include the application of bagging with decision trees and the naïve Bayes classifier to the *HouseVotes84* data, and with regression trees and linear regression to the *Boston Housing* data.

The bagging ensembles for the *HouseVotes84* bring no improvement over the corresponding single models. For the *Boston Housing* data regression tree bagging ensemble outperforms the single tree considerably, though. The lack of improvement for the ensemble of linear models is not at all surprising, as bagging cannot overcome their linearity limitation in any way.

15.5.2 Stacking

The combination of using different algorithms (possibly with instance sampling to enable a greater number of diverse models) for base model creation and using base models as attributes for their aggregation yields the technique known as *stacking*. This term suggests a multilevel hierarchy of models could be created, as discussed in Section 15.4.4. The number of levels (no more than a few), the number of models, and the choice of algorithms used on particular levels are design decisions that may have a significant impact on the final ensemble quality. This makes stacking much more difficult to properly use than bagging, where just one algorithm and the number of base models need to be selected. Even in the simplest one-level setting, stacking is actually more refined than just using base model outputs as attributes for creating an aggregated model. It employs an internal data splitting technique related to the k-fold evaluation procedure presented in Section 7.3.4 which makes sure that predictions serving as attribute values for any instance x are produced by base models created with x excluded from the training set.

Using a modeling algorithm instead of simple voting or averaging to combine base models might appear a much more powerful and promising approach, capable of delivering at least as good, and likely better prediction quality. This is not necessarily the case, though, since base models may not be sufficiently good attributes for typical modeling algorithms. This is because the latter are usually designed to search for relationship patterns between the target attribute and other attributes. Such patterns may not exist, or may be not predictively be strong enough to outperform simple voting or averaging. In other words, using detailed information how particular base models predicted may not permit any improvement over simply using the very basic summary statistics: mode or mean. While there is definitely evidence of the usefulness of stacking in some cases, this ensemble modeling technique has not become nearly as popular as the other techniques reviewed in this section.

15.5.3 Boosting

Boosting can be best explained as an enhancement of bagging that attempts to include base model diversity by shifting the focus during base model creation toward instances that turn out the most "predictively difficult." This effectively makes consecutive base models specialized in different domain regions.

15.5.3.1 Base models

The shift of focus that underlies boosting is most naturally achieved by instance weighting. A single modeling algorithm is applied to the same original training set T using a sequence of varying weight vectors $w^{(1)}, w^{(2)}, \ldots, w^{(n)}$. It does not necessarily rule out the application of boosting with modeling algorithms that are not weight sensitive, though, since weighting can be approximated by random sampling with replacement, using weights – normalized to sum up to 1 – as instance selection probabilities. This sampling-based form of boosting most directly corresponds to bagging and makes the view of boosting as a bagging enhancement the most natural, but – as an approximation to the ordinary weighting-based boosting – is usually not used unless necessary.

Starting from uniform initial weights $w^{(1)}$, the weight vector is modified after each base model has been created and applied to the training set T. Instances for which the model yields poor predictions have their weights increased and/or those for which it yields good predictions

have their weights decreased. For the classification task, this means simply raising the weights of misclassified training instances and/or reducing the weights of correctly classified training instances. For the regression task weight modifications would depend on model residuals.

For the regression task, it is also possible to use the previous models' residuals as target function values for subsequent base model creation instead of instance weighting. This will make the regression algorithm attempt to compensate the previous models' deficiencies rather than optimize its own training performance. The first model h_1 is created in the usual way. For $i > 1$, after models $h_1, \ldots, h_{i-1}$ have been created, their combined residuals are used instead of the target function values to create model h_i. This is another way of achieving the shift of focus effect that is at the heart of boosting.

15.5.3.2 Model aggregation

Base models are combined using weighted voting, with model weights $W_1, W_2, \ldots, W_m$ based on their training performance (and of course better models assigned higher voting weights). This is necessary, since (unlike for bagging) – due to the shift of focus during their creation – base models may exhibit considerably different training performance levels. In particular, if sufficiently many of them are created, the most recent ones may be entirely focused on the "most difficult" instances and yield poor predictions. It is important to underline that weighted model performance measures need to be adopted, with the same instance weights vector $w^{(i)}$ previously used to create model h_i also used to evaluate it and assign its voting weight W_i. This can be, in particular, the weighted misclassification error defined in Section 7.2.2 or any of weighted residual-based regression performance measures defined in Section 10.2.8.

15.5.3.3 Properties

Notice that the shift of focus during base model creation in boosting not only stimulates their diversity, but also drives the overall prediction quality, since instances that turned out to be "predictively difficult" keep receiving increasingly more attention. This is expected to boost the ensemble performance, as reflected by the term "boosting." Indeed, boosting model ensembles often belong to the most accurate models that can be achieved, at least for the classification task, on which boosting research and applications are mostly focused. Interestingly, the performance of even very simple and imperfect base models may be boosted substantially. It is particularly common to apply this technique with simple decision or regression trees limited to just a few levels, or even just a single split. No overfitting prevention, parameter tuning, or attribute selection is then necessary or desirable. It is actually sufficient that base models are just better than random guessing. Algorithms with parameters set up to yield such just-above-random models are referred to as *weak learners*. All base models are then nearly useless individually, yet they still form a powerful ensemble collectively. Each of them is much more likely to be "underfitted" than overfitted, and it is the boosting process, with its instance and model weight adjustments, that is responsible for most of the actual "fitting" to the training set. This is in contrast to bagging, where the overfitting of individual base models is normal and even desired for greater diversity, but canceled out by aggregation.

One possible disadvantage of boosting in comparison to bagging is that base models are not independent and have to be created sequentially. For bagging, all base models can be created in parallel, which enables efficient parallel implementations. This may be important for such computation-intensive algorithms.

15.5.3.4 Instantiations

Different schemes for instance weight modifications (or other focus shift techniques) and model weighting may be used to instantiate boosting, which makes it actually a family of ensemble modeling algorithms. The most noteworthy of these boosting instantiations for classification and regression are briefly reviewed below.

AdaBoost The AdaBoost (standing for *adaptive boosting*) algorithm is the best-known instantiation of boosting, applicable to two-class classification tasks. As usual in this book, we will assume that the set of classes is $C = \{0, 1\}$, although it is more common to present the algorithm for the $\{-1, 1\}$ set of classes, which makes some steps easier to write by implicitly exploiting the numerical nature of class labels. The essential specific features that AdaBoost brings to the generic boosting techniques are its instance and model weighting schemes. The weight of the model h_i depends on its training set weighted misclassification error $e_{c,T,w^{(i)}}(h_i)$, calculated according to the definition presented in Section 7.2.2, in the following way:

$$W_i = \frac{1}{2} \ln \frac{1 - e_{c,T,w^{(i)}}(h_i)}{e_{c,T,w^{(i)}}(h_i)} \tag{15.13}$$

This weighting scheme is a decreasing function of error values, which gives more weight to more accurate models. The weight of model h_i is not only used for voting during prediction, but also to control the degree of instance weight modifications. The latter is performed as follows:

$$w_x^{(i+1)} = w_x^{(i)} e^{W_i(2\mathbb{I}_{h_i(x)\neq c(x)}-1)} \tag{15.14}$$

where the indicator function $\mathbb{I}_{h_i(x)\neq c(x)}$ returns 0 is the model predicts correctly for instance x and 1 otherwise. The $2\mathbb{I}_{h_i(x)\neq c(x)} - 1$ expression is therefore equal to -1 if x is classified correctly and 1 if x is misclassified. This increases the weights of misclassified instances and decreases the weights of correctly classified instances to a degree that depends on the weight of model h_i. More accurate (higher weighted) models result in more extensive instance weight updates.

Example 15.5.2 The following R code produces plots that illustrate the AdaBoost weighting schemes.

```
curve(0.5*log((1-x)/x), from=0, to=0.5,
      xlab="model error", ylab="model weight")
curve(exp(0.5*log((1-x)/x)), from=0, to=0.5,
      xlab="model error", ylab="instance weight multiplier", ylim=c(0, 10), lty=2)
curve(exp(-0.5*log((1-x)/x)), from=0, to=0.5, lty=3, add=TRUE)
legend("topright", legend=c("misclassified", "correctly classified"), lty=2:3)
```

The obtained plots are presented in Figure 15.7. The first plot represents the dependence of model weight on model error and the other the dependence of the multiplier applied to modify instance weights on model error. The latter contains two curves, a gashed one for misclassified instances, and a dotted one for correctly classified instances. In all the cases, the range of model error values is limited to the $[0, 0.5]$ interval, assuming base models have above-random training performance.

As we can see, model weights may considerably exceed 1 for good models (with error of about 0.1 and below) and approach 0 for poor models with near-random training performance. Model weight changes are more rapid for small error values than for large ones. The instance weight multiplier applied for incorrectly classified instances grows rapidly with model error dropping below about 0.1, correspondingly. Then it near-linearly drops from about 3 to about 2 for model error increasing from 0.1 to 0.2, and also near-linearly goes down from about 2 to about 1 for model error raising from 0.2 to 0.5. The multiplier applied to the weights of correctly classified instances changes from 0 for perfectly accurate models to 1 for near-random models in a mostly linear manner, except for small-error models, when it drops toward 0 faster.

The complete AdaBoost algorithm is presented below. It assumes that the modeling algorithm used, referred to as $\mathcal{M}$, is weight sensitive and does not require instance weights to sum up to 1 (if the latter is not true, the normalization of the weight vector is required). Similarly the weighted misclassification error is assumed to be calculated correctly without requiring instance weights to sum up to 1. The algorithm performs at most m iterations, with m designating the specified maximum number of base models, but may terminate earlier after obtaining a base model that is not sufficiently better than random. To check this condition, the model's weighted misclassification error on the training set is compared against the expected random guess error 0.5, using a specified margin $\epsilon > 0$. Receiving such a poor model before reaching the maximum number of base models indicates that no further improvement is probably possible, and putting more weight on misclassified instances would result in further degradation rather than improvement. The algorithm returns the set of created models and their weights, to be used for weighted voting prediction.

It can be shown that AdaBoost solves the optimization problem consisting in minimizing the *exponential loss* of the created ensemble model h_* on the training set

$$\sum_{x \in T} e^{2\mathbb{I}_{h_*(x) \neq c(x)} - 1} \tag{15.15}$$

which clearly leads to minimizing the training misclassification error as well. Despite perfectly fitting to the training set, it is not prone to overfitting – although not completely overfitting resistant. This resistance is reinforced if base models are indeed severely underfitted, just above random. Hence the popularity of *decision stumps*, i.e., one-split decision trees, in this role. On the other hand, the risk of overfitting is increased for noisy data. These intuitively "obvious" statements are not necessarily fully supported by empirical evidence, which sometimes provide surprising counter arguments, but in general – boosting does manage to avoid overfitting in most practical classification tasks much better than most nonensemble classification algorithms.

Example 15.5.3 The R code presented below implements the AdaBoost algorithm, using the `base.ensemble.weight.x` function from Example 15.3.2 for base model generation and the `predict.ensemble.weighted` function from Example 15.4.3 for prediction combination. The former requires the instance reweighting function to be provided, which does most of the work. Notice that the function takes care, in particular, of calculating and retaining base model weights. The model weighting function applied includes an additional term that depends on the number of classes and is equal to 0 for the two-class setting assumed by AdaBoost. This actually implements one of its possible multiclass extensions, as discussed in the next subsection. The algorithm is demonstrated in the same way as before, though, using the two-class *HouseVotes84* dataset. Decision trees of fixed maximum depth equal to 1, 3, and 5 are used as base models.

Notice that depth-1 decision trees (i.e., decision stumps) achieve the least misclassification error, improving over that obtained for bagging in Example 15.5.1. Larger trees give worse results.

Multiclass AdaBoost The AdaBoost algorithm strongly relies on the assumption that the error of all base models does not exceed 0.5. This is perfectly reasonable if there are two classes, for which this is the random guess performance level, but cannot be expected otherwise. With errors above 0.5 the AdaBoost model weighting scheme is no longer useful, as it may deliver negative weights.

The restriction to two-class classification tasks limits the practical utility of the AdaBoost algorithm in an important way. There have been several attempts to overcome this restriction. They have different levels of complexity, theoretical justifications, and practical advantages.

One self-suggesting approach is to apply one of the binary multiclass encoding techniques described in Section 17.4. The simplest of them would be to decompose a multiclass classification task into multiple two-class tasks using the "1 vs. rest" approach. Conceptually, it consists in replacing the original target concept $c : X \to C$ with $|C|$ concepts c_d for each $d \in C$, where

$$c_d(x) = \begin{cases} 1 & \text{if } c(x) = d \\ 0 & \text{otherwise} \end{cases} \tag{15.16}$$

For each of these a separate AdaBoost binary model ensemble can be created in the usual way. This applies the 1-of-k encoding presented in Section 17.4.2.

A more refined incarnation of this "1 vs. rest" idea is also possible. Basically, instead of multiple applications of the AdaBoost algorithm, one may apply the algorithm once, but with each training instance x replaced by its $|C|$ copies $\langle x, d\rangle$, each with one of the original class labels $d \in C$ appended. The weights vector used for base model creation is correspondingly extended, to assign a numerical weight $w_{x,d}$ to instance $\langle x, d\rangle$. Base models created for the resulting extended training set and weight vector are assumed, correspondingly, to make binary predictions for instance-class pairs: $h_i : X \times C \to \{0, 1\}$. Weights for extended instances (i.e., instance-class pairs) are modified using the same formula as in the original algorithm. The ensemble's prediction for instance x would be then obtained by weighted voting:

$$h_*(x, d) = \arg \max_{b \in \{0,1\}} \sum_{i=1}^{m} W^i \mathbb{I}_{h_i(x,d)=b} \tag{15.17}$$

$$h_*(x) = \arg \max_{d \in C} h_*(x, d) \tag{15.18}$$

This technique is known as the AdaBoost.MH algorithm.

Another approach, known as the SAMME algorithm (*stagewise additive modeling using an exponential loss function*) proceeds in a completely different way, by directly creating an ensemble multiclass base models. It uses a modified model weighting scheme, incorporating an apparently minor, but important change:

$$W_i = \frac{1}{2} \ln \frac{1 - e_{c,T,w_i}(h)}{e_{c,T,w_i}(h)} + \frac{1}{2} \ln(|C| - 1) \tag{15.19}$$

It is equivalent to AdaBoost for $|C| = 2$ and for $|C| > 2$ it incorporates an adjustment term that preserves the exponential loss minimization property.

Gradient boosting Gradient boosting applies the idea of boosting to the regression task. A sequence of regression models is created, with each model trying to contribute an improvement to the training set performance achieved by its predecessors. Unlike for AdaBoost or other instantiations of classification boosting, this is achieved not by instance weighting, but rather by using the residuals of the ensemble of previously created base models instead of the original target function values when creating a subsequent base model. Base models are then combined using weighted averaging (or, actually, summation). This technique is presented below in its most generic form, although a randomized version thereof that additionally applies instance sampling for greater base model diversity, referred to as *stochastic gradient boosting*, may often perform better.

The first model h_1 is created in the usual way. For $i = 2, \ldots, m$ model h_i is created to predict the so-called pseudoresiduals:

$$r_x^{(i)} = -\frac{\partial \mathcal{L}(f(x), h_{1:i-1}(x))}{\partial h_{1:i-1}(x)} \tag{15.20}$$

of the partial ensemble $h_{1:i-1}$, consisting of the previously created models $h_1, \ldots, h_{i-1}$, for each $x \in T$. In this equation, $\mathcal{L}$ is the adopted loss function to be minimized, as discussed in Section 10.2.9. Predicting its negated derivative with respect to the previous iteration's predictions is expected to decrease the total loss. For the most popular quadratic loss (i.e., mean square error minimization), we would take $r_x^{(i)} = f(x) - h_{1:i-1}(x)$. These pseudoresiduals are passed to the regression algorithm instead of the original target function values. Base models are combined by weighted summation to achieve both the partial and final ensemble, i.e.,

$$h_{1:i-1}(x) = \sum_{j=1}^{i-1} W_j h_j(x) \tag{15.21}$$

$$h_*(x) = h_{1:m}(x) = \sum_{j=1}^{m} W_j h_j(x) \tag{15.22}$$

The weight W_i of model h_i is selected to minimize the total loss, under the adopted loss function, for the ensemble extended to include the model, which may be written as follows:

$$W_i = \beta \arg\min_W \sum_{x \in T} \mathcal{L}(f(x), h_{1:i-1}(x) + W h_i(x)) \tag{15.23}$$

where $0 < \beta \leq 1$ is a step-size parameter. While the basic version of gradient boosting assumes $\beta = 1$, using a smaller β value may help to reduce the risk of overfitting and improve the generalization capabilities of the resulting model ensemble. This form of overfitting prevention is referred to as *shrinkage*. The complete gradient boosting algorithm is presented below.

1: $h_1 := \mathcal{M}(T, f)$; $W_1 := 1$;
2: **for** $i = 2, 3, ..., m$ **do**
3: **for all** $x \in T$ **do**
4: $r_x^{(i)} := -\frac{\partial \mathcal{L}(f(x), h_{1:i-1}(x))}{\partial h_{1:i-1}(x)}$;
5: **end for**
6: $h_i := \mathcal{M}(T, r^{(i)})$;
7: $W_i := \beta \arg\min_W \sum_{x \in T} \mathcal{L}(f(x), h_{1:i-1}(x) + W h_i(x))$;
8: **end for**
9: **return** $\langle h_1, W_1 \rangle, ..., \langle h_m, W_m \rangle$;

The regression algorithm $\mathcal{M}$ used for base model creation is assumed to receive the training set as well as the corresponding target values, with the original target function values used for h_1 and the current residuals used afterward. It is not uncommon, though, to create a particularly simple first model that predicts a constant value, chosen to minimize the adopted loss function. In particular, for the quadratic loss, this constant model would simply predict the mean target function value for the training set:

$$h_1(x) = \frac{1}{|T|} \sum_{x \in T} f(x) \tag{15.24}$$

as this clearly minimizes the mean square error over all possible constant models.

For this most common special case of the quadratic loss (mean square error minimization) we have

$$\sum_{x\in T} \mathcal{L}(f(x), h_{1:i-1}(x) + Wh_i(x)) = \sum_{x\in T} \big(f(x) - (h_{1:i-1}(x) + Wh_i(x))\big)^2 \tag{15.25}$$

Minimizing this quantity with respect to W yields, by equating the corresponding derivative to 0:

$$\sum_{x\in T} (f(x) - h_{1:i-1}(x) - Wh_i(x))h_i(x) = 0 \tag{15.26}$$

from which one can obtain

$$W = \frac{\sum_{x\in T}(f(x) - h_{1:i-1}(x))h_i(x)}{\sum_{x\in T} h_i^2(x)} = \frac{\sum_{x\in T} r_x^{(i)} h_i(x)}{\sum_{x\in T} h_i^2(x)} \tag{15.27}$$

This will clearly yield 1 if model h_i does indeed perfectly predict the previous ensemble's residuals, but can be verified to be also equal to 1 even for completely imperfect regression trees with target value means assigned to leaves. To see why, consider a leaf $\mathbf{l}$ assigned a target function value equal to the mean target value for the corresponding subset of training instances:

$$v_{\mathbf{l}} = \frac{1}{|T_{\mathbf{l}}|} \sum_{x\in T_{\mathbf{l}}} f(x) \tag{15.28}$$

Then the sum of target function value and prediction products for training instances assigned to leaf $\mathbf{l}$ can be written as

$$\sum_{x\in T_{\mathbf{l}}} f(x)h(x) = \sum_{x_1\in T_{\mathbf{l}}} \left(f(x_1) \frac{1}{|T_{\mathbf{l}}|} \sum_{x_2\in T_{\mathbf{l}}} f(x_2) \right) = \frac{1}{|T_{\mathbf{l}}|} \sum_{x_1\in T_{\mathbf{l}}} \sum_{x_2\in T_{\mathbf{l}}} f(x_1)f(x_2) \tag{15.29}$$

On the other hand, the sum of squared predictions for the same subset of training instances can be transformed in the following way:

$$\begin{aligned} \sum_{x\in T_{\mathbf{l}}} h^2(x) &= \sum_{x\in T_{\mathbf{l}}} \left(\frac{1}{|T_{\mathbf{l}}|} \sum_{x_1\in T_{\mathbf{l}}} f(x_1) \right) \left(\frac{1}{|T_{\mathbf{l}}|} \sum_{x_2\in T_{\mathbf{l}}} f(x_2) \right) \\ &= |T_{\mathbf{l}}| \frac{1}{|T_{\mathbf{l}}|^2} \sum_{x_1\in T_{\mathbf{l}}} \sum_{x_2\in T_{\mathbf{l}}} f(x_1)f(x_2) = \frac{1}{|T_{\mathbf{l}}|} \sum_{x_1\in T_{\mathbf{l}}} \sum_{x_2\in T_{\mathbf{l}}} f(x_1)f(x_2) \end{aligned} \tag{15.30}$$

which yields

$$\sum_{x\in T_{\mathbf{l}}} f(x)h(x) = \sum_{x\in T_{\mathbf{l}}} h^2(x) \tag{15.31}$$

This immediately implies

$$\sum_{x\in T} f(x)h(x) = \sum_{x\in T} h^2(x) \tag{15.32}$$

since summation over all training instances can be decomposed into summation over leaves and then training instances assigned to these leaves. This property holds for an arbitrary target function, including, in particular, the previous ensemble's residuals in gradient boosting:

$$\sum_{x\in T} r_x^{(i)} h_i(x) = \sum_{x\in T} h_i^2(x) \tag{15.33}$$

from which $W = 1$.

Notice that model weights for regression trees are all equal to 1, as expected. Unfortunately, the mean square error levels for gradient boosting with regression trees are worse than obtained for regression tree bagging in Example 15.5.1, although – with a maximum depth of 3 – better than for the single regression tree model.

15.5.4 Random forest

The random forest technique of ensemble modeling can be viewed as another enhancement of bagging. This view is even more justified than that of boosting, since random forests actually use bootstrap data samples as training sets for base model creation, just like bagging. The enhancement consists in stimulating greater base model diversity by randomizing the modeling algorithm applied to these samples, which is – as the name of the technique suggests – a decision tree or regression tree algorithm. Being tied to a particular modeling algorithm (or a family of algorithms) is not such a distinctive feature of random forests as it might appear, though, given the prevailing practice of using (the standard unrandomized versions of) the very same algorithms with other ensemble modeling techniques.

15.5.4.1 Base models

Random forests combine two approaches to base model creation: instance sampling (using bootstrap samples) and algorithm nondeterminism. The latter is achieved by randomizing the split selection operation used for decision tree or regression tree growing. The randomization consists in drawing a random subset of available attributes in each node and restricting the subsequent split selection process to splits using attributes from that subset. The usual split evaluation criteria for decision trees or for regression trees are then applied. Otherwise the growing process remains unchanged. Stop criteria for decision or regression tree growing are set up to yield relatively large, accurately fitted (more than likely overfitted) trees and no pruning is applied. This setup, resulting in many splits being selected (and not pruned off), permits a very high level of base model diversity, at least unless the number of available splits (directly implied by the number of attributes) is overly small. A standard heuristic is to use the square root of the number of all available attributes as the size of the randomly drawn subset of attributes. Typically at least several hundred base models are created. Their individual overfitting is canceled out by the aggregation process, which makes the random forest ensemble highly resistant to overfitting.

15.5.4.2 Model aggregation

Randomized decision trees or regression trees used as base models for random forests are aggregated via plain (unweighted) voting or averaging/summation.

Example 15.5.5 The R code presented below implements a simple version of random forest ensemble modeling, using the `grow.randdectree`, `grow.randregtree`, and `base.ensemble.simple` functions from Example 15.3.4 and the `predict.ensemble`

`.basic` function from Example 15.4.1. The `predict.dectree` and `predict.regtree` functions, defined in Examples 3.5.1 and 9.5.1, are used to generate base model predictions. The random forest algorithm is demonstrated for the *HouseVotes84* and *Boston Housing* datasets, using three different maximum tree depth settings: 3, 5, and 8.

For the *HouseVotes84* data, the evaluated random forest ensembles improve over a single decision tree, unless using the greatest maximum tree depth. This may appear surprising, since increased tree depth permits more base model diversity and should therefore offer better improvement potential. Attribute sampling may be too aggressive or the small number of trees may be insufficient to compensate for their accuracy reduction due to randomized split selection. The inefficiency of the presented illustrative implementation prevents creating larger random forests in reasonable time. For the *Boston Housing* data, the observations

better match the expectations, with the random forest model using the least maximum depth performing worse and the other two models – better than single regression trees.

15.5.4.3 Side effects

Apart from delivering ensemble models, the random forest technique also has some quite useful "side effects," obtained by appropriately using individual decision or regression trees from the grown forest as well as the corresponding training sets or out-of-bag (OOB) instances. The most useful of those are briefly discussed below.

Performance estimates Since each tree in the forest is grown using a bootstrap sample drawn from the original training set, there is also the corresponding subset of instances not used for growing. These are the OOB instances that were not drawn to the training sample. For the particular tree these instances are therefore perfectly usable for the purpose of model evaluation, i.e., can be used to calculate true performance estimates as if the standard hold-out procedure were employed. It is the performance of the complete forest rather than that of individual trees that is to be estimated, though. This is possible by combining the OOB predictions of base models.

Let $T'_1, T'_2, \ldots, T'_m$ denote the sets of OOB instances for base models (trees) $h_1, h_2, \ldots, h_m$, grown using training sets $T_1, T_2, \ldots, T_m$, respectively. For any instance $x \in T$ let

$$I_x = \left\{ i \in \{1, 2, \ldots, m\} \mid x \in T'_i \right\} \tag{15.34}$$

designate the set of base model numbers for which x is an OOB instance. The OOB prediction for instance x, denoted by $h_{\text{OOB}}(x)$, is then obtained by combining the predictions of all models h_i for $i \in I_x$ via plain voting (for classification) or averaging (for regression). These predictions for all $x \in T$ may be then compared against true class labels $c(x)$ or target function values $f(x)$ to calculate any performance measure of interest. The misclassification error and the mean square error are of course the typical choices.

Notice that such OOB-based performance estimation technique is by no means the same as or a variation of the bootstrapping evaluation procedure discussed in Section 7.3.6, to which it is only superficially similar. This is because the latter estimates the performance of single models whereas the former estimates the performance of a complete ensemble. The resulting estimate is quite reliable and comparable to standard cross-validation with respect to its bias and variance. Given the computational cost of random forest growing (with hundreds or more trees), performing a standard cross-validation loop might easily become computationally prohibitive.

Instance proximity Random forests make it straightforward to measure instance proximity based on instance co-occurring statistics in individual trees. Basically, for each instance $x \in T$ and each tree h_i for $i = 1, 2, \ldots, n$, the corresponding tree leaf $\mathbf{l}_{i,x}$ may be determined, by passing down the training set through the tree. Then the proximity of instances $x_1, x_2 \in T$ is calculated as the number of trees where they both end up in the same leaf:

$$\varepsilon(x_1, x_2) = \sum_{i=1}^{m} \mathbb{I}_{\mathbf{l}_{i,x_1} = \mathbf{l}_{i,x_2}} \tag{15.35}$$

While definitely related to dissimilarity and similarity measures discussed in Chapter 11, random forest-based instance proximity is calculated in an entirely different way and may serve different purposes. While the former are based on attribute-value differences or correlations, the latter represent rather the property of (usually) falling into the same domain regions, with boundaries determined by the values of selected, predictively useful attributes. One step toward relating these two quantities would be therefore to consider proximity as similarity restricted only to predictively important attributes. Moreover, proximity is not necessarily sensitive to attribute value differences that usually have no high impact on model predictions, regardless of their scale. It may therefore not be appropriate for typical instantiations of the clustering task, but may become useful for other purposes, such as domain decomposition for modeling tasks or data preprocessing. In particular, one natural application of such a proximity measure is missing value imputation where missing attribute values for an instance may be imputed based on the known values observed for other instances with the highest proximity to that instance.

Attribute utility Out-of-bag instances are useful not only for estimating model quality, but also (for estimating) attribute utility, which may be viewed as particular attributes' impact on the former. With a set of spared nontraining instances for each tree one can observe how crucial particular attributes are for the obtained predictive performance level. One way to do this is to simulate "corrupting" each attribute (separately) by randomly permuting its values in each tree's OOB set and measure the effect of this "corruption."

Assuming the same notation as introduced earlier in this section, one would compare regular OOB predictions, yielding $h_{\text{OOB}}(x)$ for each $x \in T$, with the corresponding predictions $h_{\text{OOB},a}(x)$ obtained with the values of attribute a randomly permuted in each of the OOB sets $T'_1, T'_2, \ldots, T'_m$. The permutation, performed independently for each tree, will have a considerable impact on the predictions of those trees which use attribute a for splitting, particularly on high levels, and little or no impact otherwise. With some base models yielding different predictions, the combined predictions will change to some extent. For any selected performance measure – with the misclassification error typically used for classification and mean square error typically used for regression – the degradation observed for $h_{\text{OOB},a}$ in comparison to h_{OOB} may be considered a measure of the predictive utility of attribute a.

Attribute utility estimation is arguably the most useful of random forest side effects that sometimes becomes the main or only reason of creating a random forest. The estimated attribute utilities may be then used for attribute selection, as discussed in Section 19.4.5, and the final model may be created using another modeling algorithm based on the selected subset. Such a usage scenario basically treats a random forest as an attribute selection filter, and – setting the computational cost apart – it turns out to belong to the best filtering attribute selection algorithms.

15.5.5 Random Naïve Bayes

The success of the random forest ensemble has become motivation for exploring a similar combination of base model creation techniques (i.e., instance sampling and attribute sampling, which is closely related to decision tree randomization in random forests) with other modeling algorithms. One particularly interesting candidate is the naïve Bayes classifier. The resulting ensemble modeling algorithm is referred to as the random naïve Bayes classifier.

15.5.5.1 Base models

Basically, the random naïve Bayes ensemble consists of multiple naïve Bayes models $h_1, h_2, \ldots, h_m$, each created from an independent bootstrap training set sample T_i and permitted to use only a randomly selected subset $\mathbf{A}_i$ of attributes. Being a stable algorithm, the naïve Bayes classifier does not deliver sufficient base model diversity when created using bootstrap samples, as discussed above. Incorporating random attribute sampling apart from bootstrap instance sampling overcomes this deficiency, though.

15.5.5.2 Model combination

Unlike in most versions of bagging or random forests, the base models of the random naïve Bayes ensemble are not combined via simple voting. The inherent probabilistic prediction capability of the naïve Bayes classifier makes it much more reasonable to apply class probability averaging to aggregate base model predictions. The resulting combined class probabilities may be used for class label prediction in the usual way.

15.5.5.3 Properties

On one hand, random naïve Bayes is just one out of many possible random forest-like bagging extensions, combining instance sampling and attribute sampling for base model generation. There are some reasons, however, to consider it particularly interesting. This is because the simplicity of the naïve Bayes classifier makes it possible to create multiple base models with a relatively low computational expense, in particular much below that of decision trees. It may be therefore more practical to apply to large datasets. This is also because using independent attribute samples not only stimulates base model diversity, but additionally makes each of them less prone to harmful effects of the unsatisfied independence assumption on which naïve Bayesian classification is based. In smaller attribute subsets attribute dependences are less likely to occur. Therefore, each base model, while possibly inaccurate due to using incomplete information, is less likely to be fooled by attribute dependences. Random naïve Bayes may be therefore successful whenever the "naïvety" of the naïve Bayes classifier becomes a problem.

Example 15.5.6 The random naïve Bayes algorithm is implemented and demonstrated by the following R code, using the naïve Bayes classifier provided by the `e1071` package. The `randnaiveBayes` function is essentially a combination of the `base.ensemble.samle.x` function from Example 15.3.1 (instance sampling) and the `base.ensemble.sample.a` function from Example 15.3.3 (attribute sampling). The `predict.randnaiveBayes` function is basically a wrapper around the implementation of probability averaging from Example 15.4.2.

Unfortunately the random naïve Bayes algorithm performs slightly worse than the original deterministic algorithm. This may be due to a relatively small number of attributes, for which sampling deteriorates model quality too much.

15.6 Quality of ensemble predictions

Model ensembles may be often expected to outperform single models, even created using refined and carefully tuned algorithms. Sometimes, particularly for boosting and random forests, the improvement may be quite substantial. It is, however the combination of data, base model creation algorithms, and ensemble modeling techniques that is ultimately responsible for the final prediction quality.

Example 15.6.1 The misclassification error or mean square error values for the model ensembles created in the series of previous examples – using bagging and boosting with decision tree, naïve Bayes, regression tree, and linear base models, as well as the random forest and random naïve Bayes algorithms – are collected and compared to one another, as well as to those achieved by single models, by the R code presented below. For each of the two datasets used a barplot of error values is produced.

The obtained barplots are presented in Figure 15.8. On the *HouseVotes84* data the AdaBoost and random forest ensembles gives an improvement over single decision tree models, unless using excessive tree depth. Bagging produces worse predictions than single models in the case of decision trees and gave no effect for the naïve Bayes classifier. Introducing random attribute sampling to the latter turns out to be harmful rather than beneficial. On the *Boston Housing* dataset bagging applied to regression trees is the most successful, with the random forest ensemble approaching a similar performance level with sufficiently deep trees. Linear model ensembles all perform on the very same level as a single model, which is to be expected, since the averaged predictions of a multiple linear models remain linear.

15.7 Conclusion

There is a lot to be excited about in the idea of ensemble modeling. It is a conceptually appealing and extremely successful practically approach to improving the predictive power of inductive models. It not only makes it possible to get better predictive performance, but it also makes the modeling process easier for the human analyst. Ensemble modeling usually means no or little risk of overfitting, no or little parameter tuning, and no or little need for attribute selection (although it may provide useful tools for the latter, as in the case of random forests). This is because, when aggregating dozens or hundreds of base models, one may be much less concerned about their individual quality. Actually, base models that would be quite poor individually – in particular, overfitted due to lack of any overfitting prevention or underfitted due to using simplified modeling algorithms – are likely to be useful ensemble components.

These unquestionable benefits are not received without a price. What has to be paid is the vastly increased computational expense of creating many base models and using them

for prediction (although it could be partially ameliorated for some ensemble modeling techniques by appropriate parallel implementations) and the loss of human readability, even if individual base models are perfectly human readable. Despite some efforts toward developing human-readable representations of model ensembles, the latter may remain the primary limitation of their applicability in some domains.

15.8 Further readings

Ensemble modeling has been one of the hottest topics of machine learning research over the last two decades, also becoming increasingly popular in practical applications in which the predictive performance is of top priority. It has also found its way into recent comprehensive data mining and machine learning books (e.g., Bishop 2007; Han *et al*. 2011; Hastie *et al*. 2011; Tan *et al*. 2013; Witten *et al*. 2011). There are also several survey articles on model ensembles (e.g., Dietterich 2000a; Rokach 2010).

The idea of combining multiple models for improved predictive performance can be traced back to early financial forecasting research (e.g., Bates and Granger 1969; Clemen 1989; Reid 1968), but it became a hot topic in the area of machine learning in the 1990s. Bagging was introduced by Breiman (1996a) as an approach to stabilizing unstable algorithms and improving their model quality using the technique of bootstrapping (Efron 1979; Efron and Tibshirani 1994). Schapire (1990) developed theoretical foundations of boosting and an early boosting algorithm, following earlier theoretical work on weak and strong learnability (Kearns and Valiant 1989). Freund and Schapire (1995) subsequently introduced the more refined AdaBoost algorithm that remains the most widely used boosting algorithm. In the same article a regression version of AdaBoost was also presented, which has not reached similar popularity. Quinlan (1996) combined both bagging and boosting with his C4.5 decision tree induction algorithm. Dietterich (2000b) compared bagging and boosting with an ensemble of decision trees obtained by split selection randomization. Friedman *et al*. (2000) presented a statistical