

Ensemble Learning

8

Having studied how to massage the input and calibrate the output, we now turn to techniques for combining different models learned from the data. There are some surprises in store. For example, it is often advantageous to take the training data and derive several different training sets from it, learn a model from each, and combine them to produce an ensemble of learned models. Indeed, techniques for doing this can be very powerful. It is, for example, possible to transform a relatively weak learning scheme into an extremely strong one (in a precise sense that we will explain). Loss of interpretability is a drawback when applying ensemble learning, but there are ways to derive intelligible structured descriptions based on what these methods learn. Finally, if several learning schemes are available, it may be advantageous not to choose the best-performing one for your dataset (using cross-validation) but to use them all and combine the results.

Many of these results are quite counterintuitive, at least at first blush. How can it be a good idea to use many different models together? How can you possibly do better than choose the model that performs best? Surely, all this runs counter to Occam's razor, which advocates simplicity? How can you possibly obtain first-class performance by combining indifferent models, as one of these techniques appears to do? But consider committees of humans, which often come up with wiser decisions than individual experts. Recall Epicurus' view that, faced with alternative explanations, one should retain them all. Imagine a group of specialists each of whom excels in a limited domain even though none is competent across the board. In struggling to understand how these methods work, researchers have exposed all sorts of connections and links that have led to even greater improvements.

8.1 COMBINING MULTIPLE MODELS

When wise people make critical decisions, they usually take into account the opinions of several experts rather than relying on their own judgment or that of a solitary trusted advisor. For example, before choosing an important new policy direction, a

benign dictator consults widely: He or she would be ill advised to follow just one expert's opinion blindly. In a democratic setting, discussion of different viewpoints may produce a consensus; if not, a vote may be called for. In either case, different expert opinions are being combined.

In data mining, a model generated by machine learning can be regarded as an expert. *Expert* is probably too strong a word!—depending on the amount and quality of the training data, and whether the learning algorithm is appropriate to the problem at hand, the expert may in truth be regrettably ignorant—but we use the term nevertheless. An obvious approach to making decisions more reliable is to combine the output of several different models. Several machine learning techniques do this by learning an ensemble of models and using them in combination: Prominent among these are schemes called *bagging*, *boosting*, and *stacking*. They can all, more often than not, increase predictive performance over a single model. And they are general techniques that are able to be applied to classification tasks and numeric prediction problems.

Bagging, boosting, and stacking have been developed over the last couple of decades, and their performance is often astonishingly good. Machine learning researchers have struggled to understand why. And during that struggle, new methods have emerged that are sometimes even better. For example, whereas human committees rarely benefit from noisy distractions, shaking up bagging by adding random variants of classifiers can improve performance. Closer analysis reveals that boosting—perhaps the most powerful of the three methods—is closely related to the established statistical technique of additive models, and this realization has led to improved procedures.

These combined models share the disadvantage of being rather hard to analyze: They can comprise dozens or even hundreds of individual models, and although they perform well it is not easy to understand in intuitive terms what factors are contributing to the improved decisions. In the last few years methods have been developed that combine the performance benefits of committees with comprehensible models. Some produce standard decision tree models; others introduce new variants of trees that provide optional paths.

8.2 BAGGING

Combining the decisions of different models means amalgamating the various outputs into a single prediction. The simplest way to do this in the case of classification is to take a vote (perhaps a weighted vote); in the case of numeric prediction it is to calculate the average (perhaps a weighted average). Bagging and boosting both adopt this approach, but they derive the individual models in different ways. In bagging the models receive equal weight, whereas in boosting weighting is used to give more influence to the more successful ones—just as an executive might place

different values on the advice of different experts depending on how successful their predictions were in the past.

To introduce bagging, suppose that several training datasets of the same size are chosen at random from the problem domain. Imagine using a particular machine learning technique to build a decision tree for each dataset. You might expect these trees to be practically identical and to make the same prediction for each new test instance. But, surprisingly, this assumption is usually quite wrong, particularly if the training datasets are fairly small. This is a rather disturbing fact and seems to cast a shadow over the whole enterprise! The reason for it is that decision tree induction (at least the standard top-down method described in Chapter 4) is an unstable process: Slight changes to the training data may easily result in a different attribute being chosen at a particular node, with significant ramifications for the structure of the subtree beneath that node. This automatically implies that there are test instances for which some of the decision trees produce correct predictions and others do not.

Returning to the preceding experts analogy, consider the experts to be the individual decision trees. We can combine the trees by having them vote on each test instance. If one class receives more votes than any other, it is taken as the correct one. Generally, the more the merrier: Predictions made by voting become more reliable as more votes are taken into account. Decisions rarely deteriorate if new training sets are discovered, trees are built for them, and their predictions participate in the vote as well. In particular, the combined classifier will seldom be less accurate than a decision tree constructed from just one of the datasets. (Improvement is not guaranteed, however. It can be shown theoretically that pathological situations exist in which the combined decisions are worse.)

Bias–Variance Decomposition

The effect of combining multiple hypotheses can be viewed through a theoretical device known as the *bias–variance decomposition*. Suppose we could have an infinite number of independent training sets of the same size and use them to make an infinite number of classifiers. A test instance is processed by all classifiers, and a single answer is determined by majority vote. In this idealized situation, errors will still occur because no learning scheme is perfect: The error rate will depend on how well the machine learning method matches the problem at hand, and there is also the effect of noise in the data, which cannot possibly be learned.

Suppose the expected error rate were evaluated by averaging the error of the combined classifier over an infinite number of independently chosen test examples. The error rate for a particular learning algorithm is called its *bias* for the learning problem and measures how well the learning method matches the problem. (We include the “noise” component in the bias term because it is generally unknown in practice anyway.) This technical definition is a way of quantifying the vaguer notion

of bias that was introduced in Section 1.5 (page 31): It measures the “persistent” error of a learning algorithm that can’t be eliminated even by taking an infinite number of training sets into account. Of course, it cannot be calculated exactly in practical situations; it can only be approximated.

A second source of error in a learned model, in a practical situation, stems from the particular training set used, which is inevitably finite and therefore not fully representative of the actual population of instances. The expected value of this component of the error, over all possible training sets of the given size and all possible test sets, is called the *variance* of the learning method for that problem. The total expected error of a classifier is made up of the sum of bias and variance—this is the *bias–variance decomposition*.

Note that we are glossing over the details here. The bias–variance decomposition was introduced in the context of numeric prediction based on squared error, where there is a widely accepted way of performing it. However, the situation is not so clear for classification, and several competing decompositions have been proposed. Regardless of the specific decomposition used to analyze the error, combining multiple classifiers in this manner generally decreases the expected error by reducing the variance component. The more classifiers that are included, the greater the reduction in variance. Of course, a difficulty arises when putting this voting scheme into practice: Usually there’s only one training set, and obtaining more data is either impossible or expensive.

Bagging attempts to neutralize the instability of learning methods by simulating the process described previously using a given training set. Instead of sampling a fresh, independent training dataset each time, the original training data is altered by deleting some instances and replicating others. Instances are randomly sampled, with replacement, from the original dataset to create a new one of the same size. This sampling procedure inevitably replicates some of the instances and deletes others. If this idea strikes a chord, it is because we encountered it in Section 5.4 (page 155) when explaining the bootstrap method for estimating the generalization error of a learning method; indeed, the term *bagging* stands for *bootstrap aggregating*. Bagging applies the learning scheme—for example, a decision tree inducer—to each one of these artificially derived datasets, and the classifiers generated from them vote for the class to be predicted. The algorithm is summarized in [Figure 8.1](#).

The difference between bagging and the idealized procedure described before is the way in which the training datasets are derived. Instead of obtaining independent datasets from the domain, bagging just resamples the original training data. The datasets generated by resampling are different from one another but are certainly not independent because they are all based on one dataset. However, it turns out that bagging produces a combined model that often performs significantly better than the single model built from the original training data, and is never substantially worse.

Bagging can also be applied to learning schemes for numeric prediction—for example, model trees. The only difference is that, instead of voting on the outcome, the individual predictions, being real numbers, are averaged. The bias–variance decomposition is applied to numeric prediction by decomposing the expected value

Model Generation

```
Let n be the number of instances in the training data.  
For each of t iterations:  
    Sample n instances with replacement from training data.  
    Apply the learning algorithm to the sample.  
    Store the resulting model.
```

Classification

```
For each of the t models:  
    Predict class of instance using model.  
Return class that has been predicted most often.
```

FIGURE 8.1

Algorithm for bagging.

of the mean-squared error of the predictions on fresh data. Bias is defined as the mean-squared error expected when averaging over models built from all possible training datasets of the same size, and variance is the component of the expected error of a single model that is due to the particular training data it was built from. It can be shown theoretically that averaging over infinitely many models built from independent training sets always reduces the expected value of the mean-squared error. (As we mentioned earlier, the analogous result is not true for classification.)

Bagging with Costs

Bagging helps most if the underlying learning scheme is unstable in that small changes in the input data can lead to quite different classifiers. Indeed, results can be improved by increasing the diversity in the ensemble of classifiers by making the learning scheme as unstable as possible. For example, when bagging decision trees, which are already unstable, better performance is often achieved by switching pruning off, which makes them even more unstable. Another improvement can be obtained by changing the way that predictions are combined for classification. As originally formulated, bagging uses voting. But when the models can output probability estimates and not just plain classifications, it makes intuitive sense to average these probabilities instead. Not only does this often improve classification slightly, but the bagged classifier also generates probability estimates—ones that are often more accurate than those produced by the individual models. Implementations of bagging commonly use this method of combining predictions.

In Section 5.7 (page 166), we showed how to make a classifier cost sensitive by minimizing the expected cost of predictions. Accurate probability estimates are necessary because they are used to obtain the expected cost of each prediction. Bagging is a

prime candidate for cost-sensitive classification because it produces very accurate probability estimates from decision trees and other powerful, yet unstable, classifiers. However, a disadvantage is that bagged classifiers are hard to analyze.

A method called *MetaCost* combines the predictive benefits of bagging with a comprehensible model for cost-sensitive prediction. It builds an ensemble classifier using bagging and deploys it to relabel the training data by giving every training instance the prediction that minimizes the expected cost, based on the probability estimates obtained from bagging. MetaCost then discards the original class labels and learns a single new classifier—for example, a single pruned decision tree—from the relabeled data. This new model automatically takes costs into account because they have been built into the class labels! The result is a single cost-sensitive classifier that can be analyzed to see how predictions are made.

In addition to the cost-sensitive *classification* technique just mentioned, Section 5.7 (page 167) also described a cost-sensitive *learning* method that learns a cost-sensitive classifier by changing the proportion of each class in the training data to reflect the cost matrix. MetaCost seems to produce more accurate results than this method, but it requires more computation. If there is no need for a comprehensible model, MetaCost's postprocessing step is superfluous: It is better to use the bagged classifier directly in conjunction with the minimum expected cost method.

8.3 RANDOMIZATION

Bagging generates a diverse ensemble of classifiers by introducing randomness into the learning algorithm's input, often with excellent results. But there are other ways of creating diversity by introducing randomization. Some learning algorithms already have a built-in random component. For example, when learning multilayer perceptrons using the backpropagation algorithm (as described in Section 6.4—page 235), the initial network weights are set to small randomly chosen values. The learned classifier depends on the random numbers because the algorithm may find a different local minimum of the error function. One way to make the outcome of classification more stable is to run the learner several times with different random number seeds and combine the classifiers' predictions by voting or averaging.

Almost every learning method is amenable to some kind of randomization. Consider an algorithm that greedily picks the best option at every step, such as a decision tree learner that picks the best attribute to split on at each node. It could be randomized by picking one of the N best options at random instead of a single winner, or by choosing a random subset of options and picking the best from that. Of course, there is a tradeoff: More randomness generates more variety in the learner but makes less use of the data, probably decreasing the accuracy of each individual model. The best dose of randomness can only be prescribed by experiment.

Although bagging and randomization yield similar results, it sometimes pays to combine them because they introduce randomness in different, perhaps complementary, ways. A popular algorithm for learning random forests builds a randomized

decision tree in each iteration of the bagging algorithm, and often produces excellent predictors.

Randomization versus Bagging

Randomization demands more work than bagging because the learning algorithm must be modified, but it can profitably be applied to a greater variety of learners. Bagging fails with stable learning algorithms whose output is insensitive to small changes in the input. For example, it is pointless to bag nearest-neighbor classifiers because their output changes very little if the training data is perturbed by sampling. But randomization can be applied even to stable learners: The trick is to randomize in a way that makes the classifiers diverse without sacrificing too much performance. A nearest-neighbor classifier's predictions depend on the distances between instances, which in turn depend heavily on which attributes are used to compute them, so nearest-neighbor classifiers can be randomized by using different, randomly chosen subsets of attributes. In fact, this approach is called the *random subspaces* method for constructing an ensemble of classifiers and was proposed as a method for learning a random forest. As with bagging, it does not require any modification to the learning algorithm. Of course, random subspaces can be used in conjunction with bagging in order to introduce randomness to the learning process in terms of both instances and attributes.

Returning to plain bagging, the idea is to exploit instability in the learning algorithm in order to create diversity among the ensemble members—but the degree of diversity achieved is less than that of other ensemble learning methods such as random forests because of randomization built into the learning algorithm, or boosting (discussed in [Section 8.4](#)). This is because bootstrap sampling creates training data sets with a distribution that resembles the original data. Consequently, the classifiers learned by bagging are individually quite accurate, but their low diversity can detract from the overall accuracy of the ensemble. Introducing randomness in the learning algorithm increases diversity but sacrifices accuracy of the individual classifiers. If it were possible for ensemble members to be both diverse and individually accurate, smaller ensembles could be used. Of course, this would have computational benefits.

Rotation Forests

An ensemble learning method called *rotation forests* has the specific goal of creating diverse yet accurate classifiers. It combines the random subspace and bagging approaches with principal components feature generation to construct an ensemble of decision trees. In each iteration, the input attributes are randomly divided into k disjoint subsets. Principal components analysis is applied to each subset in turn in order to create linear combinations of the attributes in the subset that are rotations of the original axes. The k sets of principal components are used to compute values for the derived attributes; these comprise the input to the tree learner at each

iteration. Because all the components obtained on each subset are retained, there are as many derived attributes as there are original ones. To discourage the generation of identical coefficients if the same feature subset is chosen in different iterations, principal component analysis is applied to training instances from a randomly chosen subset of the class values (however, the values of the derived attributes that are input to the tree learner are computed from all the instances in the training data). To further increase diversity, a bootstrap sample of the data can be created in each iteration before the principal components transformations are applied.

Experiments indicate that rotation forests can give similar performance to random forests, with far fewer trees. An analysis of diversity (measured by the Kappa statistic, introduced in Section 5.7 (page 166), which can be used to measure agreement between classifiers) versus error for pairs of ensemble members shows a minimal increase in diversity and reduction in error for rotation forests when compared to bagging. However, this appears to translate into significantly better performance for the ensemble as a whole.

8.4 BOOSTING

We have explained that bagging exploits the instability inherent in learning algorithms. Intuitively, combining multiple models only helps when these models are significantly different from one another and each one treats a reasonable percentage of the data correctly. Ideally, the models complement one another, each being a specialist in a part of the domain where the other models don't perform very well—just as human executives seek advisors whose skills and experience complement, rather than duplicate, one another.

The boosting method for combining multiple models exploits this insight by explicitly seeking models that complement one another. First, the similarities: Like bagging, boosting uses voting (for classification) or averaging (for numeric prediction) to combine the output of individual models. Again like bagging, it combines models of the same type—for example, decision trees. However, boosting is iterative. Whereas in bagging individual models are built separately, in boosting each new model is influenced by the performance of those built previously. Boosting encourages new models to become experts for instances handled incorrectly by earlier ones by assigning greater weight to those instances. A final difference is that boosting weights a model's contribution by its confidence rather than giving equal weight to all models.

AdaBoost

There are many variants on the idea of boosting. We describe a widely used method called *AdaBoost.M1* that is designed specifically for classification. Like bagging, it can be applied to any classification learning algorithm. To simplify matters we assume that the learning algorithm can handle weighted instances, where the weight of an instance is a positive number (we revisit this assumption later). The presence

of instance weights changes the way in which a classifier's error is calculated: It is the sum of the weights of the misclassified instances divided by the total weight of all instances, instead of the fraction of instances that are misclassified. By weighting instances, the learning algorithm can be forced to concentrate on a particular set of instances, namely those with high weight. Such instances become particularly important because there is a greater incentive to classify them correctly. The C4.5 algorithm, described in Section 6.1, is an example of a learning method that can accommodate weighted instances without modification because it already uses the notion of fractional instances to handle missing values.

The boosting algorithm, summarized in Figure 8.2, begins by assigning equal weight to all instances in the training data. It then calls the learning algorithm to form a classifier for this data and reweights each instance according to the classifier's output. The weight of correctly classified instances is decreased, and that of misclassified ones is increased. This produces a set of "easy" instances with low weight and a set of "hard" ones with high weight. In the next iteration—and all subsequent ones—a classifier is built for the reweighted data, which consequently focuses on classifying the hard instances correctly. Then the instances' weights are increased or decreased according to the output of this new classifier. As a result, some hard instances might become even harder and easier ones even easier; on the other hand, other hard instances might become easier, and easier ones harder—all possibilities can occur in practice. After each iteration, the weights reflect how often the instances have been misclassified by the classifiers produced so far. By maintaining a measure

Model Generation

```

Assign equal weight to each training instance.
For each of t iterations:
    Apply learning algorithm to weighted dataset and store resulting
    model.
    Compute error e of model on weighted dataset and store error.
    If e equal to zero, or e greater or equal to 0.5:
        Terminate model generation.
    For each instance in dataset:
        If instance classified correctly by model:
            Multiply weight of instance by e / (1 - e).
        Normalize weight of all instances.

```

Classification

```

Assign weight of zero to all classes.
For each of the t (or less) models:
    Add  $-\log(e / (1 - e))$  to weight of class predicted by model.
Return class with highest weight.

```

FIGURE 8.2

Algorithm for boosting.

of “hardness” with each instance, this procedure provides an elegant way of generating a series of experts that complement one another.

How much should the weights be altered after each iteration? The answer depends on the current classifier’s overall error. More specifically, if e denotes the classifier’s error on the weighted data (a fraction between 0 and 1), then weights are updated by

$$\text{weight} \leftarrow \text{weight} \times e/(1-e)$$

for correctly classified instances, and the weights remain unchanged for misclassified ones. Of course, this does not increase the weight of misclassified instances as claimed earlier. However, after all weights have been updated they are renormalized so that their sum remains the same as it was before. Each instance’s weight is divided by the sum of the new weights and multiplied by the sum of the old ones. This automatically increases the weight of each misclassified instance and reduces that of each correctly classified one.

Whenever the error on the weighted training data exceeds or equals 0.5, the boosting procedure deletes the current classifier and does not perform any more iterations. The same thing happens when the error is 0 because then all instance weights become 0.

We have explained how the boosting method generates a series of classifiers. To form a prediction, their output is combined using a weighted vote. To determine the weights, note that a classifier that performs well on the weighted training data from which it was built (e close to 0) should receive a high weight, and a classifier that performs badly (e close to 0.5) should receive a low one. The AdaBoost.M1 algorithm uses

$$\text{weight} = -\log \frac{e}{1-e}$$

which is a positive number between 0 and infinity. Incidentally, this formula explains why classifiers that perform perfectly on the training data must be deleted: When e is 0 the weight is undefined. To make a prediction, the weights of all classifiers that vote for a particular class are summed, and the class with the greatest total is chosen.

We began by assuming that the learning algorithm can cope with weighted instances. Any algorithm can be adapted to deal with weighted instances; we explained how at the end of Section 6.6, Locally Weighted Linear Regression (page 258). Instead of changing the learning algorithm, it is possible to generate an unweighted dataset from the weighted data by resampling—the same technique that bagging uses. Whereas for bagging, each instance is chosen with equal probability, for boosting, instances are chosen with probability proportional to their weight. As a result, instances with high weight are replicated frequently, and ones with low weight may never be selected. Once the new dataset becomes as large as the original one, it is fed into the learning scheme instead of the weighted data. It’s as simple as that.

A disadvantage of this procedure is that some instances with low weight don't make it into the resampled dataset, so information is lost before the learning scheme is applied. However, this can be turned into an advantage. If the learning scheme produces a classifier with an error that exceeds 0.5, boosting must terminate if the weighted data is used directly, whereas with resampling it might be possible to produce a classifier with an error below 0.5 by discarding the resampled dataset and generating a new one from a different random seed. Sometimes more boosting iterations can be performed by resampling than when using the original weighted version of the algorithm.

The Power of Boosting

The idea of boosting originated in a branch of machine learning research known as *computational learning theory*. Theoreticians are interested in boosting because it is possible to derive performance guarantees. For example, it can be shown that the error of the combined classifier on the training data approaches 0 very quickly as more iterations are performed (exponentially quickly in the number of iterations). Unfortunately, as explained in Section 5.1, guarantees for the training error are not very interesting because they do not necessarily indicate good performance on fresh data. However, it can be shown theoretically that boosting only fails on fresh data if the individual classifiers are too “complex” for the amount of training data present or if their training errors become too large too quickly (in a precise sense explained by Schapire et al., 1997). As usual, the problem lies in finding the right balance between the individual models' complexity and their fit to the data.

If boosting does succeed in reducing the error on fresh test data, it often does so in a spectacular way. One very surprising finding is that performing more boosting iterations can reduce the error on new data long after the error of the combined classifier on the training data has dropped to 0. Researchers were puzzled by this result because it seems to contradict Occam's razor, which declares that, of two hypotheses that explain the empirical evidence equally well, the simpler one is to be preferred. Performing more boosting iterations without reducing the training error does not explain the training data any better, and it certainly adds complexity to the combined classifier. The contradiction can be resolved by considering the classifier's confidence in its predictions. More specifically, we measure confidence by the difference between the estimated confidence for the true class and that of the most likely predicted class other than the true class—a quantity known as the *margin*. The larger the margin, the more confident the classifier is in predicting the true class. It turns out that boosting can increase the margin long after the training error has dropped to 0. The effect can be visualized by plotting the cumulative distribution of the margin values of all the training instances for different numbers of boosting iterations, giving a graph known as the *margin curve*. Thus, if the explanation of empirical evidence takes the margin into account, Occam's razor remains as sharp as ever.

The beautiful thing about boosting is that a powerful combined classifier can be built from very simple ones as long as they achieve less than 50% error on the reweighted data. Usually, this is easy—certainly for learning problems with two classes! Simple learning schemes are called *weak* learners, and boosting converts weak learners into strong ones. For example, good results for two-class problems can often be obtained by boosting extremely simple decision trees that have only one level, called *decision stumps*. Another possibility is to apply boosting to an algorithm that learns a single conjunctive rule—such as a single path in a decision tree—and classifies instances based on whether or not the rule covers them. Of course, multiclass datasets make it more difficult to achieve error rates below 0.5. Decision trees can still be boosted, but they usually need to be more complex than decision stumps. More sophisticated algorithms have been developed that allow very simple models to be boosted successfully in multiclass situations.

Boosting often produces classifiers that are significantly more accurate on fresh data than ones generated by bagging. However, unlike bagging, boosting sometimes fails in practical situations: It can generate a classifier that is significantly less accurate than a single classifier built from the same data. This indicates that the combined classifier overfits the data.

8.5 ADDITIVE REGRESSION

When boosting was first investigated it sparked intense interest among researchers because it could coax first-class performance from indifferent learners. Statisticians soon discovered that it could be recast as a greedy algorithm for fitting an additive model. Additive models have a long history in statistics. Broadly, the term refers to any way of generating predictions by summing up contributions obtained from other models. Most learning algorithms for additive models do not build the base models independently but ensure that they complement one another and try to form an ensemble of base models that optimizes predictive performance according to some specified criterion.

Boosting implements *forward stagewise additive modeling*. This class of algorithms starts with an empty ensemble and incorporates new members sequentially. At each stage the model that maximizes the predictive performance of the ensemble as a whole is added, without altering those already in the ensemble. Optimizing the ensemble's performance implies that the next model should focus on those training instances on which the ensemble performs poorly. This is exactly what boosting does by giving those instances larger weights.

Numeric Prediction

Here's a well-known forward stagewise additive modeling method for numeric prediction. First, build a standard regression model—for example, a regression tree. The errors it exhibits on the training data—the differences between predicted and

observed values—are called *residuals*. Then correct for these errors by learning a second model—perhaps another regression tree—that tries to predict the observed residuals. To do this, simply replace the original class values by their residuals before learning the second model. Adding the predictions made by the second model to those of the first one automatically yields lower error on the training data. Usually some residuals still remain because the second model is not a perfect one, so we continue with a third model that learns to predict the residuals of the residuals, and so on. The procedure is reminiscent of the use of rules with exceptions for classification that we discussed in Section 3.4 (page 73).

If the individual models minimize the squared error of the predictions, as linear regression models do, this algorithm minimizes the squared error of the ensemble as a whole. In practice, it also works well when the base learner uses a heuristic approximation instead, such as the regression and model tree learners described in Section 6.6. In fact, there is no point in using standard linear regression as the base learner for additive regression because the sum of linear regression models is again a linear regression model and the regression algorithm itself minimizes the squared error. However, it is a different story if the base learner is a regression model based on a single attribute, the one that minimizes the squared error. Statisticians call this *simple* linear regression, in contrast to the standard multi-attribute method, properly called *multiple* linear regression. In fact, using additive regression in conjunction with simple linear regression and iterating until the squared error of the ensemble decreases no further yields an additive model identical to the least-squares multiple linear regression function.

Forward stagewise additive regression is prone to overfitting because each model added fits the training data closer and closer. To decide when to stop, use cross-validation. For example, perform a cross-validation for every number of iterations up to a user-specified maximum and choose the one that minimizes the cross-validated estimate of squared error. This is a good stopping criterion because cross-validation yields a fairly reliable estimate of the error on future data. Incidentally, using this method in conjunction with simple linear regression as the base learner effectively combines multiple linear regression with built-in attribute selection. The reason is that the next most important attribute's contribution is only included if it decreases the cross-validated error.

For implementation convenience, forward stagewise additive regression usually begins with a level-0 model that simply predicts the mean of the class on the training data so that every subsequent model fits residuals. This suggests another possibility for preventing overfitting: Instead of subtracting a model's entire prediction to generate target values for the next model, shrink the predictions by multiplying them by a user-specified constant factor between 0 and 1 before subtracting. This reduces the model's fit to the residuals and consequently reduces the chance of overfitting. Of course, it may increase the number of iterations needed to arrive at a good additive model. Reducing the multiplier effectively damps down the learning process, increasing the chance of stopping at just the right moment but also increasing runtime.

Additive Logistic Regression

Additive regression can also be applied to classification just as linear regression can. But we know from Section 4.6 that logistic regression is more suitable than linear regression for classification. It turns out that a similar adaptation can be made to additive models by modifying the forward stagewise modeling method to perform additive *logistic* regression. Use the logit transform to translate the probability estimation problem into a regression problem, as we did in Section 4.6 (page 126), and solve the regression task using an ensemble of models—for example, regression trees—just as for additive regression. At each stage, add the model that maximizes the probability of the data given the ensemble classifier.

Suppose f_j is the j th regression model in the ensemble and $f_j(\mathbf{a})$ is its prediction for instance \mathbf{a} . Assuming a two-class problem, use the additive model $\sum f_j(\mathbf{a})$ to obtain a probability estimate for the first class:

$$p(1 | \mathbf{a}) = \frac{1}{1 + e^{-\sum f_j(\mathbf{a})}}$$

This closely resembles the expression used in Section 4.6 (page 126), except that here it is abbreviated by using vector notation for the instance \mathbf{a} and the original weighted sum of the attributes' values is replaced by a sum of arbitrarily complex regression models f .

Figure 8.3 shows the two-class version of the so-called *LogitBoost* algorithm, which performs additive logistic regression and generates the individual models f_j . Here, y_i is 1 for an instance in the first class and 0 for an instance in the second. In each iteration this algorithm fits a regression model f_j to a weighted version of the original dataset based on dummy class values z_i and weights w_i . We assume that $p(1 | \mathbf{a})$ is computed using the f_j that were built in previous iterations.

The derivation of this algorithm is beyond the scope of this book, but it can be shown that the algorithm maximizes the probability of the data with respect to the ensemble if each model f_j is determined by minimizing the squared error on the corresponding regression problem. In fact, if multiple linear regression is used to form the f_j , the algorithm converges to the maximum-likelihood, linear-logistic regression model: It is an incarnation of the iteratively reweighted least-squares method mentioned in Section 4.6 (page 126).

Superficially, LogitBoost looks quite different from AdaBoost, but the predictors they produce differ mainly in that the former optimizes the likelihood directly whereas the latter optimizes an exponential loss function that can be regarded as an approximation to it. From a practical perspective, the difference is that LogitBoost uses a regression scheme as the base learner whereas AdaBoost works with classification algorithms.

We have only shown the two-class version of LogitBoost, but the algorithm can be generalized to multiclass problems. As with additive regression, the danger of overfitting can be reduced by shrinking the predictions of the individual f_j by a

Model generation

```

for j = 1 to t iterations:
  for each instance a[i]:
    set the target value for the regression to
       $z[i] = (y[i] - p(1|a[i])) / [p(1|a[i]) * (1 - p(1|a[i]))]$ 
    set the weight  $w[i]$  of instance a[i] to  $p(1|a[i]) * (1 - p(1|a[i]))$ 
  fit a regression model  $f[j]$  to the data with class values  $z[i]$  and
  weights  $w[i]$ 

```

Classification

```

predict class 1 if  $p(1 | a) > 0.5$ , otherwise predict class 0

```

FIGURE 8.3

Algorithm for additive logistic regression.

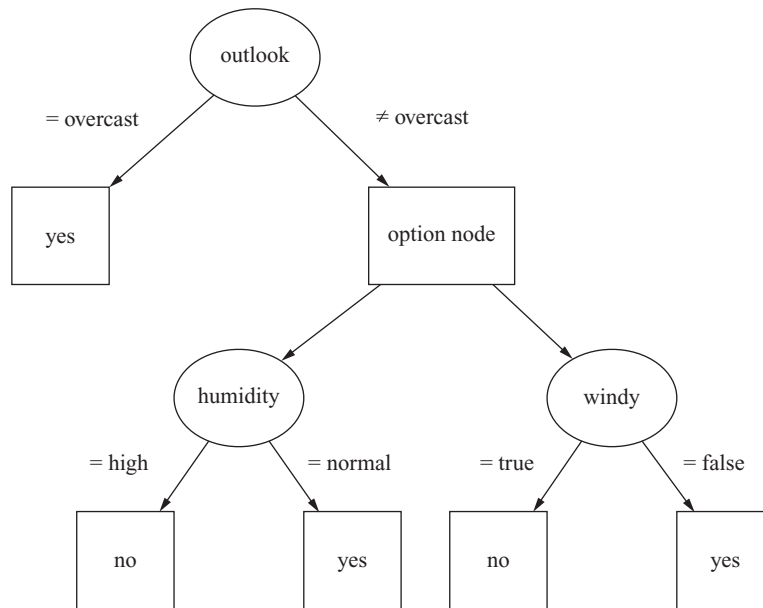
predetermined multiplier and using cross-validation to determine an appropriate number of iterations.

8.6 INTERPRETABLE ENSEMBLES

Bagging, boosting, and randomization all produce ensembles of classifiers. This makes it very difficult to analyze what kind of information has been extracted from the data. It would be nice to have a single model with the same predictive performance. One possibility is to generate an artificial dataset, by randomly sampling points from the instance space and assigning them the class labels predicted by the ensemble classifier, and then learn a decision tree or rule set from this new dataset. To obtain similar predictive performance from the tree as from the ensemble, a huge dataset may be required, but in the limit this strategy should be able to replicate the performance of the ensemble classifier—and it certainly will if the ensemble itself consists of decision trees.

Option Trees

Another approach is to derive a single structure that can represent an ensemble of classifiers compactly. This can be done if the ensemble consists of decision trees; the result is called an *option tree*. Option trees differ from decision trees in that they contain two types of node: decision nodes and option nodes. Figure 8.4 shows a simple example for the weather data, with only one option node. To classify an instance, filter it down through the tree. At a decision node take just one of the

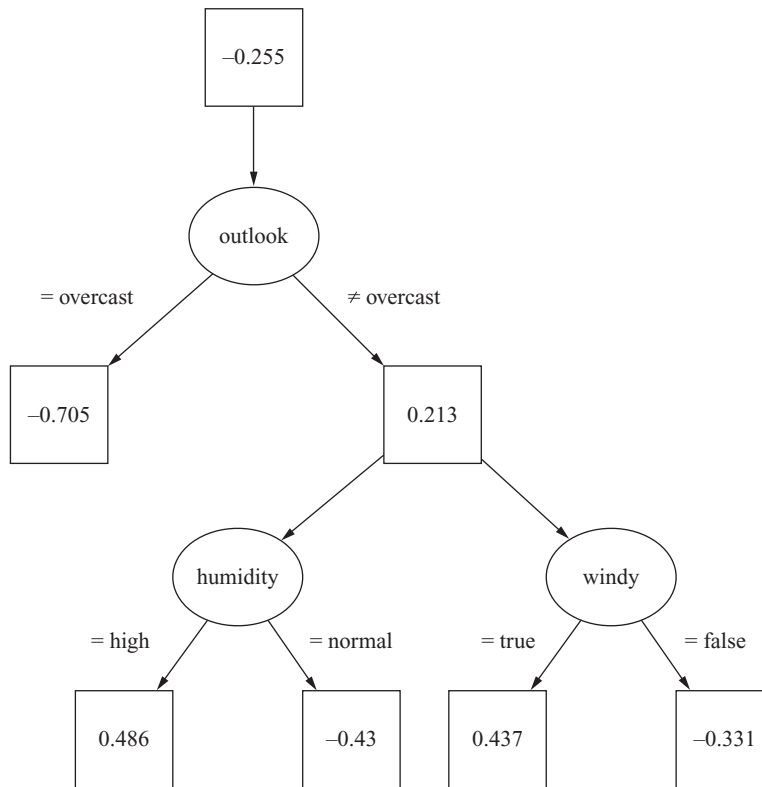
**FIGURE 8.4**

Simple option tree for the weather data.

branches, as usual, but at an option node take *all* of the branches. This means that the instance ends up in more than one leaf, and the classifications obtained from those leaves must somehow be combined into an overall classification. This can be done simply by voting, taking the majority vote at an option node to be the prediction of the node. In that case it makes little sense to have option nodes with only two options (as in Figure 8.4) because there will only be a majority if both branches agree. Another possibility is to average the probability estimates obtained from the different paths, using either an unweighted average or a more sophisticated Bayesian approach.

Option trees can be generated by modifying an existing decision tree learner to create an option node if there are several splits that look similarly useful according to their information gain. All choices within a certain user-specified tolerance of the best one can be made into options. During pruning, the error of an option node is the average error of its options.

Another possibility is to grow an option tree by incrementally adding nodes to it. This is commonly done using a boosting algorithm, and the resulting trees are usually called *alternating decision trees* instead of option trees. In this context, the decision nodes are called *splitter nodes* and the option nodes are called *prediction nodes*. Prediction nodes are leaves if no splitter nodes have been added to them yet. The standard alternating decision tree applies to two-class problems, and with each prediction node a positive or negative numeric value is associated. To obtain a

**FIGURE 8.5**

Alternating decision tree for the weather data.

prediction for an instance, filter it down all applicable branches and sum up the values from any prediction nodes that are encountered; predict one class or the other depending on whether the sum is positive or negative.

A simple example tree for the weather data is shown in [Figure 8.5](#), where a positive value corresponds to class *play = no* and a negative one to *play = yes*. To classify an instance with *outlook = sunny*, *temperature = hot*, *humidity = normal*, and *windy = false*, filter it down to the corresponding leaves, obtaining values -0.255 , 0.213 , -0.430 , and -0.331 . The sum of these values is negative; thus, predict *play = yes*. Alternating decision trees always have a prediction node at the root, as in this example.

The alternating tree is grown using a boosting algorithm—for example, a boosting algorithm that employs a base learner for numeric prediction, such as the Logit-Boost method described previously. Assume that the base learner produces a single conjunctive rule in each boosting iteration. Then an alternating decision tree can be

generated by simply adding each rule into the tree. The numeric scores associated with the prediction nodes are obtained from the rules. However, the resulting tree would grow large very quickly because the rules from different boosting iterations are likely to be different. Thus, learning algorithms for alternating decision trees consider only those rules that extend one of the *existing* paths in the tree by adding a splitter node and two corresponding prediction nodes (assuming binary splits). In the standard version of the algorithm, every possible location in the tree is considered for addition and a node is added according to a performance measure that depends on the particular boosting algorithm employed. However, heuristics can be used instead of an exhaustive search to speed up the learning process.

Logistic Model Trees

Option trees and alternating trees yield very good classification performance based on a single structure, but they may still be difficult to interpret when there are many option nodes because it becomes difficult to see how a particular prediction is derived. However, it turns out that boosting can also be used to build very effective decision trees that do not include any options at all. For example, the LogitBoost algorithm has been used to induce trees with linear-logistic regression models at the leaves. These are called *logistic model trees* and are interpreted in the same way as the model trees for regression described in Section 6.6.

LogitBoost performs additive logistic regression. Suppose that each iteration of the boosting algorithm fits a simple regression function by going through all the attributes, finding the simple regression function with the smallest error, and adding it into the additive model. If the LogitBoost algorithm is run until convergence, the result is a maximum-likelihood, multiple-logistic regression model. However, for optimum performance on future data it is usually unnecessary to wait for convergence—and to do so is often detrimental. An appropriate number of boosting iterations can be determined by estimating the expected performance for a given number of iterations using cross-validation and stopping the process when performance ceases to increase.

A simple extension of this algorithm leads to logistic model trees. The boosting process terminates when there is no further structure in the data that can be modeled using a linear-logistic regression function. However, there may still be structure that linear models can fit if attention is restricted to subsets of the data, obtained, for example, by splitting the data using a standard decision tree criterion such as information gain. Then, once no further improvement can be obtained by adding more simple linear models, the data is split and boosting is resumed separately in each subset. This process takes the logistic model generated so far and refines it separately for the data in each subset. Again, cross-validation is run in each subset to determine an appropriate number of iterations to perform in that subset.

The process is applied recursively until the subsets become too small. The resulting tree will surely overfit the training data, and one of the standard methods of

decision tree learning can be used to prune it. Experiments indicate that the pruning operation is very important. Using the cost-complexity pruning method discussed in Section 6.1 (page 202), which chooses the right tree size using cross-validation, the algorithm produces small but very accurate trees with linear logistic models at the leaves.

8.7 STACKING

Stacked generalization, or *stacking* for short, is a different way of combining multiple models. Although developed some years ago, it is less widely used than bagging and boosting partly because it is difficult to analyze theoretically and partly because there is no generally accepted best way of doing it—the basic idea can be applied in many different variations.

Unlike bagging and boosting, stacking is not normally used to combine models of the same type—for example, a set of decision trees. Instead, it is applied to models built by different learning algorithms. Suppose you have a decision tree inducer, a Naïve Bayes learner, and an instance-based learning scheme and you want to form a classifier for a given dataset. The usual procedure would be to estimate the expected error of each algorithm by cross-validation and to choose the best one to form a model for prediction on future data. But isn't there a better way? With three learning algorithms available, can't we use all three for prediction and combine the outputs?

One way to combine outputs is by voting—the same mechanism used in bagging. However, (unweighted) voting only makes sense if the learning schemes perform comparably well. If two of the three classifiers make predictions that are grossly incorrect, we will be in trouble! Instead, stacking introduces the concept of a *meta-learner*, which replaces the voting procedure. The problem with voting is that it's not clear which classifier to trust. Stacking tries to *learn* which classifiers are the reliable ones, using another learning algorithm—the *metalearner*—to discover how best to combine the output of the base learners.

The input to the metamodel—also called the *level-1 model*—are the predictions of the base models, or *level-0 models*. A level-1 instance has as many attributes as there are level-0 learners, and the attribute values give the predictions of these learners on the corresponding level-0 instance. When the stacked learner is used for classification, an instance is first fed into the level-0 models, and each one guesses a class value. These guesses are fed into the level-1 model, which combines them into the final prediction.

There remains the problem of training the level-1 learner. To do this, we need to find a way of transforming the level-0 training data (used for training the level-0 learners) into level-1 training data (used for training the level-1 learner). This seems straightforward: Let each level-0 model classify a training instance, and attach to their predictions the instance's actual class value to yield a level-1 training instance. Unfortunately, this doesn't work well. It allows simplistic rules to be learned, such as

always believe the output of classifier A, and ignore B and C. This rule may well be appropriate for particular base classifiers A, B, and C; and if so it will probably be learned. But just because it seems appropriate on the training data doesn't necessarily mean that it will work well on the test data—because it will inevitably learn to prefer classifiers that overfit the training data over ones that make decisions more realistically.

Consequently, stacking does not simply transform the level-0 training data into level-1 data in this manner. Recall from Chapter 5 that there are better methods of estimating a classifier's performance than using the error on the training set. One is to hold out some instances and use them for an independent evaluation. Applying this to stacking, we reserve some instances to form the training data for the level-1 learner and build level-0 classifiers from the remaining data. Once the level-0 classifiers have been built they are used to classify the instances in the holdout set, forming the level-1 training data. Because the level-0 classifiers haven't been trained on these instances, their predictions are unbiased; therefore, the level-1 training data accurately reflects the true performance of the level-0 learning algorithms. Once the level-1 data has been generated by this holdout procedure, the level-0 learners can be reapplied to generate classifiers from the full training set, making slightly better use of the data and leading to better predictions.

The holdout method inevitably deprives the level-1 model of some of the training data. In Chapter 5, cross-validation was introduced as a means of circumventing this problem for error estimation. This can be applied in conjunction with stacking by performing a cross-validation for every level-0 learner. Each instance in the training data occurs in exactly one of the test folds of the cross-validation, and the predictions of the level-0 inducers built from the corresponding training fold are used to build a level-1 training instance from it. This generates a level-1 training instance for each level-0 training instance. Of course, it is slow because a level-0 classifier has to be trained for each fold of the cross-validation, but it does allow the level-1 classifier to make full use of the training data.

Given a test instance, most learning schemes are able to output probabilities for every class label instead of making a single categorical prediction. This can be exploited to improve the performance of stacking by using the probabilities to form the level-1 data. The only difference from the standard procedure is that each nominal level-1 attribute—representing the class predicted by a level-0 learner—is replaced by several numeric attributes, each representing a class probability output by the level-0 learner. In other words, the number of attributes in the level-1 data is multiplied by the number of classes. This procedure has the advantage that the level-1 learner is privy to the confidence that each level-0 learner associates with its predictions, thereby amplifying communication between the two levels of learning.

An outstanding question remains: What algorithms are suitable for the level-1 learner? In principle, any learning scheme can be applied. However, because most of the work is already done by the level-0 learners, the level-1 classifier is basically just an arbiter and it makes sense to choose a rather simple algorithm for this

purpose. In the words of David Wolpert, the inventor of stacking, it is reasonable that “relatively global, smooth” level-1 generalizers should perform well. Simple linear models or trees with linear models at the leaves usually work well.

Stacking can also be applied to numeric prediction. In that case, the level-0 models and the level-1 model all predict numeric values. The basic mechanism remains the same; the only difference lies in the nature of the level-1 data. In the numeric case, each level-1 attribute represents the numeric prediction made by one of the level-0 models, and instead of a class value the numeric target value is attached to level-1 training instances.

8.8 FURTHER READING

Ensemble learning is a popular research topic in machine learning research, with many related publications. The term *bagging* (for “bootstrap aggregating”) was coined by Breiman (1996b), who investigated the properties of bagging theoretically and empirically for both classification and numeric prediction.

The bias–variance decomposition for classification presented in [Section 8.2](#) is due to Dietterich and Kong (1995). We chose this version because it is both accessible and elegant. However, the variance can turn out to be negative because, as we mentioned, aggregating models from independent training sets by voting may in pathological situations actually *increase* the overall error compared to a model from a single training set. This is a serious disadvantage because variances are normally squared quantities—the square of the standard deviation—and therefore cannot become negative. In his technical report, Breiman (1996c) proposed a different bias–variance decomposition for classification. This has caused some confusion in the literature because three different versions of this report can be located on the Web. The official version, entitled “Arcing classifiers,” describes a more complex decomposition that cannot, by construction, produce negative variance. However, the original version, entitled “Bias, variance, and arcing classifiers,” follows Dietterich and Kong’s formulation (except that Breiman splits the bias term into bias plus noise). There is also an intermediate version with the original title but the new decomposition; it includes an appendix in which Breiman explains that he abandoned the old definition because it can produce negative variance. (Authors sometimes mistakenly refer to the earlier drafts, which have been superseded, or use an earlier title for the latest, official, report.) However, in the new version (and in decompositions proposed by other authors) the bias of the aggregated classifier can exceed the bias of a classifier built from a single training set, which also seems counterintuitive.

The MetaCost algorithm was introduced by Domingos (1999).

The random subspace method was suggested as an approach for learning ensemble classifiers by Ho (1998) and applied as a method for learning ensembles of nearest-neighbor classifiers by Bay (1999). Randomization was evaluated by Dietterich (2000) and compared with bagging and boosting. Random forests were introduced by Breiman (2001). Rotation forests are a relatively new ensemble learning method introduced by Rodriguez et al. (2006). Subsequent studies by Kuncheva and

Rodriguez (2007) show that the main factors responsible for its performance are the use of principal components transformations (as opposed to other feature-extraction methods such as random projections) and the application of principal components analysis to random subspaces of the original input attributes.

Freund and Schapire (1996) developed the AdaBoost.M1 boosting algorithm and derived theoretical bounds for its performance. Later they improved these bounds using the concept of margins (Freund and Schapire, 1999). Drucker (1997) adapted AdaBoost.M1 for numeric prediction. The LogitBoost algorithm was developed by Friedman et al. (2000). Friedman (2001) describes how to make boosting more resilient in the presence of noisy data.

Domingos (1997) describes how to derive a single interpretable model from an ensemble using artificial training examples. Bayesian option trees were introduced by Buntine (1992), and majority voting was incorporated into option trees by Kohavi and Kunz (1997). Freund and Mason (1999) introduced alternating decision trees; experiments with multiclass alternating decision trees were reported by Holmes et al. (2002). Landwehr et al. (2005) developed logistic model trees using the LogitBoost algorithm.

Stacked generalization originated with Wolpert (1992), who presented the idea in the neural-network literature; it was applied to numeric prediction by Breiman (1996a). Ting and Witten (1997a) compared different level-1 models empirically and found that a simple linear model performs best; they also demonstrated the advantage of using probabilities as level-1 data. A combination of stacking and bagging has also been investigated (Ting and Witten, 1997b).

8.9 WEKA IMPLEMENTATIONS

In Weka, ensemble learning is done using the mechanism of “metalearners,” described near the end of Section 11.2. They are covered in Section 11.5 and listed in Table 11.6, excluding the *RandomForest* classifier, which is located in Weka’s *trees* package (Section 11.4 and Table 11.5), and the interpretable ensembles, which are also located there:

- Bagging:
 - *Bagging* (bag a classifier; works for regression too)
 - *MetaCost* (make a classifier cost-sensitive)
- Randomization:
 - *RandomCommittee* (ensembles using different random number seeds)
 - *RandomSubSpace* (ensembles using random subsets of attributes)
 - *RandomForest* (bag ensembles of random trees)
 - *RotationForest* (ensembles using rotated random subspaces)
- Boosting: *AdaBoostM1*
- Additive regression:
 - *AdditiveRegression*
 - *LogitBoost* (additive logistic regression)

- Interpretable ensembles:
 - *ADTree* (alternating decision trees)
 - *LADTree* (learns alternating decision trees using LogitBoost)
 - *LMT* (logistic model trees)
- Selecting or combining algorithms:
 - *MultiScheme* (selection using cross-validation)
 - *Vote* (simple combination of predictions)
 - *Stacking* (learns how to combine predictions)