17 *Combining Multiple Learners*

We discussed many different learning algorithms in the previous chapters. Though these are generally successful, no one single algorithm is always the most accurate. Now, we are going to discuss models composed of multiple learners that complement each other so that by combining them, we attain higher accuracy.

17.1 Rationale

IN ANY APPLICATION, we can use one of several learning algorithms, and with certain algorithms, there are hyperparameters that affect the final learner. For example, in a classification setting, we can use a parametric classifier or a multilayer perceptron, and, for example, with a multilayer perceptron, we should also decide on the number of hidden units. The No Free Lunch Theorem states that there is no single learning algorithm that in any domain always induces the most accurate learner. The usual approach is to try many and choose the one that performs the best on a separate validation set.

Each learning algorithm dictates a certain model that comes with a set of assumptions. This inductive bias leads to error if the assumptions do not hold for the data. Learning is an ill-posed problem and with finite data, each algorithm converges to a different solution and fails under different circumstances. The performance of a learner may be fine-tuned to get the highest possible accuracy on a validation set, but this fine-tuning is a complex task and still there are instances on which even the best learner is not accurate enough. The idea is that there may be another learner that is accurate on these. By suitably combining multiple *base-learners* then, accuracy can be improved. Recently with computation and

BASE-LEARNER

memory getting cheaper, such systems composed of multiple learners have become popular (Kuncheva 2004).

There are basically two questions here:

1. How do we generate base-learners that complement each other?

2. How do we combine the outputs of base-learners for maximum accuracy?

Our discussion in this chapter will answer these two related questions. We will see that model combination is not a trick that always increases accuracy; model combination does always increase time and space complexity of training and testing, and unless base-learners are trained carefully and their decisions combined smartly, we will only pay for this extra complexity without any significant gain in accuracy.

17.2 Generating Diverse Learners

DIVERSITY

Since there is no point in combining learners that always make similar decisions, the aim is to be able to find a set of *diverse* learners who differ in their decisions so that they complement each other. At the same time, there cannot be a gain in overall success unless the learners are accurate, at least in their domain of expertise. We therefore have this double task of maximizing individual accuracies and the diversity between learners. Let us now discuss the different ways to achieve this.

Different Algorithms

We can use different learning algorithms to train different base-learners. Different algorithms make different assumptions about the data and lead to different classifiers. For example, one base-learner may be parametric and another may be nonparametric. When we decide on a single algorithm, we give emphasis to a single method and ignore all others. Combining multiple learners based on multiple algorithms, we free ourselves from taking a decision and we no longer put all our eggs in one basket.

Different Hyperparameters

We can use the same learning algorithm but use it with different hyperparameters. Examples are the number of hidden units in a multilayer perceptron, *k* in *k*-nearest neighbor, error threshold in decision trees, the kernel function in support vector machines, and so forth. With a Gaussian parametric classifier, whether the covariance matrices are shared or not is a hyperparameter. If the optimization algorithm uses an iterative procedure such as gradient descent whose final state depends on the initial state, such as in backpropagation with multilayer perceptrons, the initial state, for example, the initial weights, is another hyperparameter. When we train multiple base-learners with different hyperparameter values, we average over this factor and reduce variance, and therefore error.

Different Input Representations

Separate base-learners may be using different *representations* of the same input object or event, making it possible to integrate different types of sensors/measurements/modalities. Different representations make different characteristics explicit allowing better identification. In many applications, there are multiple sources of information, and it is desirable to use all of these data to extract more information and achieve higher accuracy in prediction.

For example, in speech recognition, to recognize the uttered words, in addition to the acoustic input, we can also use the video image of the speaker's lips and shape of the mouth as the words are spoken. This is similar to *sensor fusion* where the data from different sensors are integrated to extract more information for a specific application. Another example is information, for example, image retrieval where in addition to the image itself, we may also have text annotation in the form of keywords. In such a case, we want to be able to combine both of these sources to find the right set of images; this is also sometimes called *multi-view learning*.

SENSOR FUSION

MULTI-VIEW LEARNING

The simplest approach is to concatenate all data vectors and treat it as one large vector from a single source, but this does not seem theoretically appropriate since this corresponds to modeling data as sampled from one multivariate statistical distribution. Moreover, larger input dimensionalities make the systems more complex and require larger samples for the estimators to be accurate. The approach we take is to make separate predictions based on different sources using separate base-learners, then combine their predictions.

Even if there is a single input representation, by choosing random subsets from it, we can have classifiers using different input features; this is

RANDOM SUBSPACE

called the *random subspace method* (Ho 1998). This has the effect that different learners will look at the same problem from different points of view and will be robust; it will also help reduce the curse of dimensionality because inputs are fewer dimensional.

Different Training Sets

Another possibility is to train different base-learners by different subsets of the training set. This can be done randomly by drawing random training sets from the given sample; this is called *bagging*. Or, the learners can be trained serially so that instances on which the preceding base-learners are not accurate are given more emphasis in training later base-learners; examples are *boosting* and *cascading*, which actively try to generate complementary learners, instead of leaving this to chance.

The partitioning of the training sample can also be done based on locality in the input space so that each base-learner is trained on instances in a certain local part of the input space; this is what is done by the *mixture of experts* that we discussed in chapter 12 but that we revisit in this context of combining multiple learners. Similarly, it is possible to define the main task in terms of a number of subtasks to be implemented by the base-learners, as is done by *error-correcting output codes*.

Diversity vs. Accuracy

One important note is that when we generate multiple base-learners, we want them to be reasonably accurate but do not require them to be very accurate individually, so they are not, and need not be, optimized separately for best accuracy. The base-learners are not chosen for their accuracy, but for their simplicity. We do require, however, that the base-learners be diverse, that is, accurate on different instances, specializing in subdomains of the problem. What we care for is the final accuracy when the base-learners are combined, rather than the accuracies of the base-learners we started from. Let us say we have a classifier that is 80 percent accurate. When we decide on a second classifier, we do not care for the overall accuracy; we care only about how accurate it is on the 20 percent that the first classifier misclassifies, as long as we know when to use which one.

This implies that the required accuracy and diversity of the learners also depend on how their decisions are to be combined, as we will discuss next. If, as in a voting scheme, a learner is consulted for all inputs, it should be accurate everywhere and diversity should be enforced everywhere; if we have a partioning of the input space into regions of expertise for different learners, diversity is already guaranteed by this partitioning and learners need to be accurate only in their own local domains.

17.3 Model Combination Schemes

There are also different ways the multiple base-learners are combined to generate the final output:

MULTIEXPERT COMBINATION

- *Multiexpert combination* methods have base-learners that work in *parallel*. These methods can in turn be divided into two:
    - In the *global* approach, also called *learner fusion*, given an input, all base-learners generate an output and all these outputs are used. Examples are *voting* and *stacking*.
    - In the *local* approach, or *learner selection*, for example, in *mixture of experts*, there is a *gating* model, which looks at the input and chooses one (or very few) of the learners as responsible for generating the output.

MULTISTAGE COMBINATION

- *Multistage combination* methods use a *serial* approach where the next base-learner is trained with or tested on only the instances where the previous base-learners are not accurate enough. The idea is that the base-learners (or the different representations they use) are sorted in increasing complexity so that a complex base-learner is not used (or its complex representation is not extracted) unless the preceding simpler base-learners are not confident. An example is *cascading*.

Let us say that we have L base-learners. We denote by $d_j(x)$ the prediction of base-learner $\mathcal{M}_j$ given the arbitrary dimensional input x. In the case of multiple representations, each $\mathcal{M}_j$ uses a different input representation x_j. The final prediction is calculated from the predictions of the base-learners:

$$y = f(d_1, d_2, \ldots, d_L|\Phi) \tag{17.1}$$

where $f(\cdot)$ is the combining function with Φ denoting its parameters.

When there are K outputs, for each learner there are $d_{ji}(x), i = 1, \ldots, K$, $j = 1, \ldots, L$, and, combining them, we also generate K values, $y_i, i =$

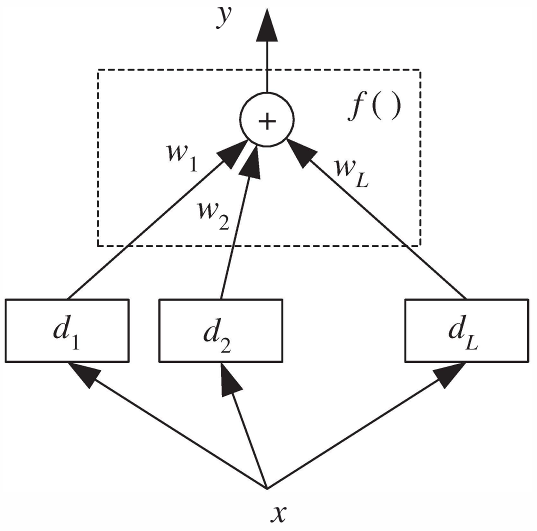

Figure 17.1 Base-learners are d_j and their outputs are combined using $f(\cdot)$. This is for a single output; in the case of classification, each base-learner has K outputs that are separately used to calculate y_i, and then we choose the maximum. Note that here all learners observe the same input; it may be the case that different learners observe different representations of the same input object or event.

$1, \ldots, K$ and then for example in classification, we choose the class with the maximum y_i value:

$$\text{Choose } C_i \text{ if } y_i = \max_{k=1}^{K} y_k$$

17.4 Voting

VOTING

The simplest way to combine multiple classifiers is by *voting*, which corresponds to taking a linear combination of the learners (see figure 17.1):

$$y_i = \sum_j w_j d_{ji} \text{ where } w_j \geq 0, \sum_j w_j = 1 \tag{17.2}$$

ENSEMBLES
LINEAR OPINION POOLS

This is also known as *ensembles* and *linear opinion pools*. In the simplest case, all learners are given equal weight and we have *simple voting*

Table 17.1 Classifier combination rules

Rule	Fusion function $f(\cdot)$
Sum	$y_i = \frac{1}{L}\sum_{j=1}^{L} d_{ji}$
Weighted sum	$y_i = \sum_j w_j d_{ji}, w_j \geq 0, \sum_j w_j = 1$
Median	$y_i = \text{median}_j d_{ji}$
Minimum	$y_i = \min_j d_{ji}$
Maximum	$y_i = \max_j d_{ji}$
Product	$y_i = \prod_j d_{ji}$

Table 17.2 Example of combination rules on three learners and three classes

	C_1	C_2	C_3
d_1	0.2	0.5	0.3
d_2	0.0	0.6	0.4
d_3	0.4	0.4	0.2
Sum	0.2	**0.5**	0.3
Median	0.2	**0.5**	0.4
Minimum	0.0	**0.4**	0.2
Maximum	0.4	**0.6**	0.4
Product	0.0	**0.12**	0.032

that corresponds to taking an average. Still, taking a (weighted) sum is only one of the possibilities and there are also other combination rules, as shown in table 17.1 (Kittler et al. 1998). If the outputs are not posterior probabilities, these rules require that outputs be normalized to the same scale (Jain, Nandakumar, and Ross 2005).

An example of the use of these rules is shown in table 17.2, which demonstrates the effects of different rules. Sum rule is the most intuitive and is the most widely used in practice. Median rule is more robust to outliers; minimum and maximum rules are pessimistic and optimistic, respectively. With the product rule, each learner has veto power; regardless of the other ones, if one learner has an output of 0, the overall output goes to 0. Note that after the combination rules, y_i do not necessarily sum up to 1.

In weighted sum, d_{ji} is the vote of learner j for class C_i and w_j is the weight of its vote. Simple voting is a special case where all voters have equal weight, namely, $w_j = 1/L$. In classification, this is called *plurality voting* where the class having the maximum number of votes is the winner. When there are two classes, this is *majority voting* where the winning class gets more than half of the votes (exercise 1). If the voters can also supply the additional information of how much they vote for each class (e.g., by the posterior probability), then after normalization, these can be used as weights in a *weighted voting* scheme. Equivalently, if d_{ji} are the class posterior probabilities, $P(C_i|x, \mathcal{M}_j)$, then we can just sum them up ($w_j = 1/L$) and choose the class with maximum y_i.

In the case of regression, simple or weighted averaging or median can be used to fuse the outputs of base-regressors. Median is more robust to noise than the average.

Another possible way to find w_j is to assess the accuracies of the learners (regressor or classifier) on a separate validation set and use that information to compute the weights, so that we give more weights to more accurate learners. These weights can also be learned from data, as we will discuss when we discuss stacked generalization in section 17.9.

Voting schemes can be seen as approximations under a Bayesian framework with weights approximating prior model probabilities, and model decisions approximating model-conditional likelihoods. This is *Bayesian model combination*—see section 16.6. For example, in classification we have $w_j \equiv P(\mathcal{M}_j)$, $d_{ji} = P(C_i|x, \mathcal{M}_j)$, and equation 17.2 corresponds to

BAYESIAN MODEL COMBINATION

$$P(C_i|x) = \sum_{\text{all models } \mathcal{M}_j} P(C_i|x, \mathcal{M}_j)P(\mathcal{M}_j) \tag{17.3}$$

Simple voting corresponds to a uniform prior. If we have a prior distribution preferring simpler models, this would give larger weights to them. We cannot integrate over all models; we only choose a subset for which we believe $P(\mathcal{M}_j)$ is high, or we can have another Bayesian step and calculate $P(\mathcal{M}_j|\mathcal{X})$, the probability of a model given the sample, and sample high probable models from this density.

Hansen and Salamon (1990) have shown that given independent two-class classifiers with success probability higher than 1/2, namely, better than random guessing, by taking a majority vote, the accuracy increases as the number of voting classifiers increases.

Let us assume that d_j are iid with expected value $E[d_j]$ and variance $\text{Var}(d_j)$, then when we take a simple average with $w_j = 1/L$, the expected value and variance of the output are

$$E[y] = E\left[\sum_j \frac{1}{L} d_j\right] = \frac{1}{L} L E[d_j] = E[d_j]$$

$$\text{Var}(y) = \text{Var}\left(\sum_j \frac{1}{L} d_j\right) = \frac{1}{L^2}\text{Var}\left(\sum_j d_j\right) = \frac{1}{L^2} L \text{Var}(d_j) = \frac{1}{L}\text{Var}(d_j) \tag{17.4}$$

We see that the expected value does not change, so the bias does not change. But variance, and therefore mean square error, decreases as the number of independent voters, L, increases. In the general case,

$$\text{Var}(y) = \frac{1}{L^2}\text{Var}\left(\sum_j d_j\right) = \frac{1}{L^2}\left[\sum_j \text{Var}(d_j) + 2\sum_j \sum_{i<j} \text{Cov}(d_j, d_i)\right] \tag{17.5}$$

which implies that if learners are positively correlated, variance (and error) increase. We can thus view using different algorithms and input features as efforts to decrease, if not completely eliminate, the positive correlation. In section 17.10, we discuss pruning methods to remove learners with high positive correlation fron an ensemble.

We also see here that further decrease in variance is possible if the voters are not independent but negatively correlated. The error then decreases if the accompanying increase in bias is not higher because these aims are contradictory; we cannot have a number of classifiers that are all accurate *and* negatively correlated. In mixture of experts for example, where learners are localized, the experts are negatively correlated but biased (Jacobs 1997).

If we view each base-learner as a random noise function added to the true discriminant/regression function and if these noise functions are uncorrelated with 0 mean, then the averaging of the individual estimates is like averaging over the noise. In this sense, voting has the effect of smoothing in the functional space and can be thought of as a regularizer with a smoothness assumption on the true function (Perrone 1993). We saw an example of this in figure 4.5d, where, averaging over models with large variance, we get a better fit than those of the individual models. This is the idea in voting: We vote over models with high variance and low bias so that after combination, the bias remains small and we reduce the variance by averaging. Even if the individual models are biased, the decrease in variance may offset this bias and still a decrease in error is possible.

17.5 Error-Correcting Output Codes

ERROR-CORRECTING OUTPUT CODES

In *error-correcting output codes* (ECOC) (Dietterich and Bakiri 1995), the main classification task is defined in terms of a number of subtasks that are implemented by the base-learners. The idea is that the original task of separating one class from all other classes may be a difficult problem. Instead, we want to define a set of simpler classification problems, each specializing in one aspect of the task, and combining these simpler classifiers, we get the final classifier.

Base-learners are binary classifiers having output $-1/+1$, and there is a *code matrix* $\mathbf{W}$ of $K \times L$ whose K rows are the binary codes of classes in terms of the L base-learners d_j. For example, if the second row of $\mathbf{W}$ is $[-1, +1, +1, -1]$, this means that for us to say an instance belongs to C_2, the instance should be on the negative side of d_1 and d_4, and on the positive side of d_2 and d_3. Similarly, the columns of the code matrix defines the task of the base-learners. For example, if the third column is $[-1, +1, +1]^T$, we understand that the task of the third base-learner, d_3, is to separate the instances of C_1 from the instances of C_2 and C_3 combined. This is how we form the training set of the base-learners. For example in this case, all instances labeled with C_2 and C_3 form $\mathcal{X}_3^+$ and instances labeled with C_1 form $\mathcal{X}_3^-$, and d_3 is trained so that $x^t \in \mathcal{X}_3^+$ give output $+1$ and $x^t \in \mathcal{X}_3^-$ give output -1.

The code matrix thus allows us to define a polychotomy ($K > 2$ classification problem) in terms of dichotomies ($K = 2$ classification problem), and it is a method that is applicable using any learning algorithm to implement the dichotomizer base-learners—for example, linear or multilayer perceptrons (with a single output), decision trees, or SVMs whose original definition is for two-class problems.

The typical one discriminant per class setting corresponds to the diagonal code matrix where $L = K$. For example, for $K = 4$, we have

$$\mathbf{W} = \begin{bmatrix} +1 & -1 & -1 & -1 \\ -1 & +1 & -1 & -1 \\ -1 & -1 & +1 & -1 \\ -1 & -1 & -1 & +1 \end{bmatrix}$$

The problem here is that if there is an error with one of the base-learners, there may be a misclassification because the class code words are so similar. So the approach in error-correcting codes is to have $L > K$ and increase the Hamming distance between the code words. One possibility is *pairwise separation* of classes where there is a separate base-learner to separate C_i from C_j, for $i < j$ (section 10.4). In this case, $L = K(K-1)/2$ and with $K = 4$, the code matrix is

$$\mathbf{W} = \begin{bmatrix} +1 & +1 & +1 & 0 & 0 & 0 \\ -1 & 0 & 0 & +1 & +1 & 0 \\ 0 & -1 & 0 & -1 & 0 & +1 \\ 0 & 0 & -1 & 0 & -1 & -1 \end{bmatrix}$$

where a 0 entry denotes "don't care." That is, d_1 is trained to separate C_1 from C_2 and does not use the training instances belonging to the other classes. Similarly, we say that an instance belongs to C_2 if $d_1 = -1$ and $d_4 = d_5 = +1$, and we do not consider the values of d_2, d_3, and d_6. The problem here is that L is $\mathcal{O}(K^2)$, and for large K pairwise separation may not be feasible.

If we can have L high, we can just randomly generate the code matrix with $-1/+1$ and this will work fine, but if we want to keep L low, we need to optimize $\mathbf{W}$. The approach is to set L beforehand and then find $\mathbf{W}$ such that the distances between rows, and at the same time the distances between columns, are as large as possible, in terms of Hamming distance. With K classes, there are $2^{(K-1)} - 1$ possible columns, namely, two-class problems. This is because K bits can be written in 2^K different ways and complements (e.g., "0101" and "1010," from our point of view, define the same discriminant) dividing the possible combinations by 2 and then subtracting 1 because a column of all 0s (or 1s) is useless. For example, when $K = 4$, we have

$$\mathbf{W} = \begin{bmatrix} -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & +1 & +1 & +1 & +1 \\ -1 & +1 & +1 & -1 & -1 & +1 & +1 \\ +1 & -1 & +1 & -1 & +1 & -1 & +1 \end{bmatrix}$$

When K is large, for a given value of L, we look for L columns out of the $2^{(K-1)}-1$. We would like these columns of $\mathbf{W}$ to be as different as possible so that the tasks to be learned by the base-learners are as different from each other as possible. At the same time, we would like the rows of $\mathbf{W}$ to be as different as possible so that we can have maximum error correction in case one or more base-learners fail.

ECOC can be written as a voting scheme where the entries of $\mathbf{W}$, w_{ij}, are considered as vote weights:

$$y_i = \sum_{j=1}^{L} w_{ij} d_j \tag{17.6}$$

and then we choose the class with the highest y_i. Taking a weighted sum and then choosing the maximum instead of checking for an exact match allows d_j to no longer need to be binary but to take a value between -1 and $+1$, carrying soft certainties instead of hard decisions. Note that a value p_j between 0 and 1, for example, a posterior probability, can be converted to a value d_j between -1 and $+1$ simply as

$$d_j = 2p_j - 1$$

The difference between equation 17.6 and the generic voting model of equation 17.2 is that the weights of votes can be different for different classes, namely, we no longer have w_j but w_{ij}, and also that $w_j \geq 0$ whereas w_{ij} are -1, 0, or $+1$.

One problem with ECOC is that because the code matrix **W** is set a priori, there is no guarantee that the subtasks as defined by the columns of **W** will be simple. Dietterich and Bakiri (1995) report that the dichotomizer trees may be larger than the polychotomizer trees and when multilayer perceptrons are used, there may be slower convergence by backpropagation.

17.6 Bagging

BAGGING

Bagging is a voting method whereby base-learners are made different by training them over slightly different training sets. Generating L slightly different samples from a given sample is done by bootstrap, where given a training set $\mathcal{X}$ of size N, we draw N instances randomly from $\mathcal{X}$ *with replacement*. Because sampling is done with replacement, it is possible that some instances are drawn more than once and that certain instances are not drawn at all. When this is done to generate L samples $\mathcal{X}_j, j = 1, \ldots, L$, these samples are similar because they are all drawn from the same original sample, but they are also slightly different due to chance. The base-learners d_j are trained with these L samples $\mathcal{X}_j$.

UNSTABLE ALGORITHM

A learning algorithm is an *unstable algorithm* if small changes in the training set causes a large difference in the generated learner, namely, the learning algorithm has high variance. Bagging, short for bootstrap aggregating, uses bootstrap to generate L training sets, trains L base-learners using an unstable learning procedure, and then, during testing, takes an average (Breiman 1996). Bagging can be used both for classification and regression. In the case of regression, to be more robust, one can take the median instead of the average when combining predictions.

We saw before that averaging reduces variance only if the positive correlation is small; an algorithm is stable if different runs of the same algorithm on resampled versions of the same dataset lead to learners with high positive correlation. Algorithms such as decision trees and multilayer perceptrons are unstable. Nearest neighbor is stable, but condensed nearest neighbor is unstable (Alpaydın 1997). If the original training set is large, then we may want to generate smaller sets of size $N' < N$ from them using bootstrap, since otherwise the bootstrap replicates $\mathcal{X}_j$ will be too similar, and d_j will be highly correlated.

17.7 Boosting

In bagging, generating complementary base-learners is left to chance and to the unstability of the learning method. In boosting, we actively try to generate complementary base-learners by training the next learner on the mistakes of the previous learners. The original *boosting* algorithm (Schapire 1990) combines three weak learners to generate a strong learner. A *weak learner* has error probability less than 1/2, which makes it better than random guessing on a two-class problem, and a *strong learner* has arbitrarily small error probability.

BOOSTING

WEAK LEARNER

STRONG LEARNER

Given a large training set, we randomly divide it into three. We use $\mathcal{X}_1$ and train d_1. We then take $\mathcal{X}_2$ and feed it to d_1. We take all instances misclassified by d_1 and also as many instances on which d_1 is correct from $\mathcal{X}_2$, and these together form the training set of d_2. We then take $\mathcal{X}_3$ and feed it to d_1 and d_2. The instances on which d_1 and d_2 disagree form the training set of d_3. During testing, given an instance, we give it to d_1 and d_2; if they agree, that is the response, otherwise the response of d_3 is taken as the output. Schapire (1990) has shown that this overall system has reduced error rate, and the error rate can arbitrarily be reduced by using such systems recursively, that is, a boosting system of three models used as d_j in a higher system.

Though it is quite successful, the disadvantage of the original boost-

```
Training:
    For all $\{x^t, r^t\}_{t=1}^N \in \mathcal{X}$, initialize $p_1^t = 1/N$
    For all base-learners $j = 1, \ldots, L$
        Randomly draw $\mathcal{X}_j$ from $\mathcal{X}$ with probabilities $p_j^t$
        Train $d_j$ using $\mathcal{X}_j$
        For each $(x^t, r^t)$, calculate $y_j^t \leftarrow d_j(x^t)$
        Calculate error rate: $\epsilon_j \leftarrow \sum_t p_j^t \cdot 1(y_j^t \neq r^t)$
        If $\epsilon_j > 1/2$, then $L \leftarrow j - 1$; stop
        $\beta_j \leftarrow \epsilon_j / (1 - \epsilon_j)$
        For each $(x^t, r^t)$, decrease probabilities if correct:
            If $y_j^t = r^t$, then $p_{j+1}^t \leftarrow \beta_j p_j^t$ Else $p_{j+1}^t \leftarrow p_j^t$
        Normalize probabilities:
            $Z_j \leftarrow \sum_t p_{j+1}^t$;  $p_{j+1}^t \leftarrow p_{j+1}^t / Z_j$
Testing:
    Given $x$, calculate $d_j(x), j = 1, \ldots, L$
    Calculate class outputs, $i = 1, \ldots, K$:
        $y_i = \sum_{j=1}^L \left(\log \frac{1}{\beta_j}\right) d_{ji}(x)$
```

Figure 17.2 AdaBoost algorithm.

ing method is that it requires a very large training sample. The sample should be divided into three and furthermore, the second and third classifiers are only trained on a subset on which the previous ones err. So unless one has a quite large training set, d_2 and d_3 will not have training sets of reasonable size. Drucker et al. (1994) use a set of 118,000 instances in boosting multilayer perceptrons for optical handwritten digit recognition.

ADABOOST

Freund and Schapire (1996) proposed a variant, named *AdaBoost*, short for adaptive boosting, that uses the same training set over and over and thus need not be large, but the classifiers should be simple so that they do not overfit. AdaBoost can also combine an arbitrary number of base-learners, not three.

Many variants of AdaBoost have been proposed; here, we discuss the original algorithm AdaBoost.M1 (see figure 17.2). The idea is to modify the probabilities of drawing the instances as a function of the error. Let us say p_j^t denotes the probability that the instance pair (x^t, r^t) is drawn to train the jth base-learner. Initially, all $p_1^t = 1/N$. Then we add new base-learners as follows, starting from $j = 1$: ϵ_j denotes the error rate of d_j. AdaBoost requires that learners are weak, that is, $\epsilon_j < 1/2, \forall j$; if not, we stop adding new base-learners. Note that this error rate is not on the original problem but on the dataset used at step j. We define $\beta_j = \epsilon_j/(1 - \epsilon_j) < 1$, and we set $p^t_{j+1} = \beta_j p^t_j$ if d_j correctly classifies x^t; otherwise, $p^t_{j+1} = p^t_j$. Because p^t_{j+1} should be probabilities, there is a normalization where we divide p^t_{j+1} by $\sum_t p^t_{j+1}$, so that they sum up to 1. This has the effect that the probability of a correctly classified instance is decreased, and the probability of a misclassified instance increases. Then a new sample of the same size is drawn from the original sample according to these modified probabilities, p^t_{j+1}, with replacement, and is used to train d_{j+1}.

This has the effect that d_{j+1} focuses more on instances misclassified by d_j; that is why the base-learners are chosen to be simple and not accurate, since otherwise the next training sample would contain only a few outlier and noisy instances repeated many times over. For example, with decision trees, *decision stumps*, which are trees grown only one or two levels, are used. So it is clear that these would have bias but the decrease in variance is larger and the overall error decreases. An algorithm like the linear discriminant has low variance, and we cannot gain by AdaBoosting linear discriminants.

Once training is done, AdaBoost is a voting method. Given an instance, all d_j decide and a weighted vote is taken where weights are proportional to the base-learners' accuracies (on the training set): $w_j = \log(1/\beta_j)$. Freund and Schapire (1996) showed improved accuracy in twenty-two benchmark problems, equal accuracy in one problem, and worse accuracy in four problems.

Schapire et al. (1998) explain that the success of AdaBoost is due to its

MARGIN

property of increasing the *margin*. If the margin increases, the training instances are better separated and an error is less likely. This makes AdaBoost's aim similar to that of support vector machines (chapter 13).

In AdaBoost, although different base-learners have slightly different training sets, this difference is not left to chance as in bagging, but is a function of the error of the previous base-learner. The actual performance of boosting on a particular problem is clearly dependent on the data and the base-learner. There should be enough training data and the base-learner should be weak but not too weak, and boosting is especially susceptible to noise and outliers.

AdaBoost has also been generalized to regression: One straightforward way, proposed by Avnimelech and Intrator (1997), checks for whether the prediction error is larger than a certain threshold, and if so marks it as error, then uses AdaBoost proper. In another version (Drucker 1997), probabilities are modified based on the magnitude of error, such that instances where the previous base-learner commits a large error, have a higher probability of being drawn to train the next base-learner. Weighted average, or median, is used to combine the predictions of the base-learners.

17.8 The Mixture of Experts Revisited

MIXTURE OF EXPERTS

In voting, the weights w_j are constant over the input space. In the *mixture of experts* architecture, which we previously discussed in section 12.8) as a local method, as an extension of radial basis functions, there is a gating network whose outputs are weights of the experts. This architecture can then be viewed as a voting method where the votes depend on the input, and may be different for different inputs. The competitive learning algorithm used by the mixture of experts localizes the base-learners such that each of them becomes an expert in a different part of the input space and have its weight, $w_j(x)$, close to 1 in its region of expertise. The final output is a weighted average as in voting

$$y = \sum_{j=1}^{L} w_j(x)d_j \tag{17.7}$$

except in this case, both the base-learners and the weights are a function of the input (see figure 17.3).

Jacobs (1997) has shown that in the mixture of experts architecture, experts are biased but are negatively correlated. As training proceeds, bias decreases and expert variances increase but at the same time as experts localize in different parts of the input space, their covariances get more and more negative, which, due to equation 17.5, decreases the total variance, and thus the error. In section 12.8, we considered the case where both experts and gating are linear functions but a nonlinear method, for example, a multilayer perceptron with hidden units, can also be used for both. This may decrease the expert biases but risks increasing expert variances and overfitting.

DYNAMIC CLASSIFIER SELECTION

In *dynamic classifier selection*, similar to the gating network of mixture of experts, there is first a system which takes a test input and estimates

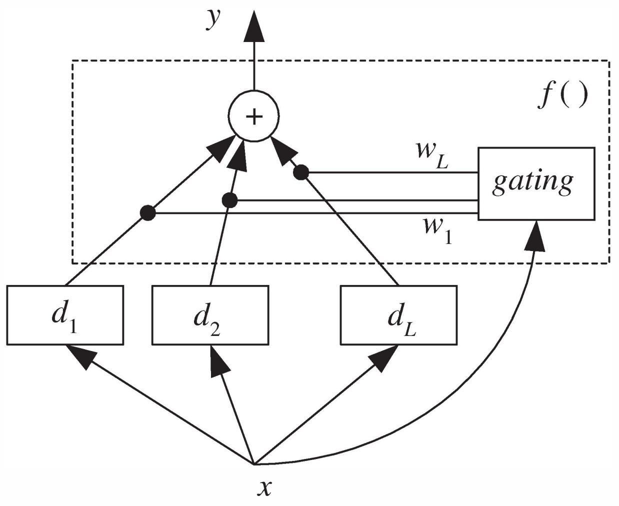

Figure 17.3 Mixture of experts is a voting method where the votes, as given by the gating system, are a function of the input. The combiner system f also includes this gating system.

the competence of base-classifiers in the vicinity of the input. It then picks the most competent to generate output and that output is given as the overall output. Woods, Kegelmeyer, and Bowyer (1997) find the k nearest training points of the test input, look at the accuracies of the base classifiers on those, and choose the one that performs the best on them. Only the selected base-classifier need be evaluated for that test input. To decrease variance, at the expense of more computation, one can take a vote over a few competent base-classifiers instead of using just a single one.

Note that in such a scheme, one should make sure that for any region of the input space, there is a competent base-classifier; this implies that there should be some partitioning of the learning of the input space among the base-classifiers. This is the nice property of mixture of experts, namely, the gating model that does the selection and the expert base-learners that it selects from are trained in a coupled manner. It would be straightforward to have a regression version of this dynamic learner selection algorithm (exercise 5).

17.9 Stacked Generalization

STACKED GENERALIZATION

Stacked generalization is a technique proposed by Wolpert (1992) that extends voting in that the way the output of the base-learners is combined need not be linear but is learned through a combiner system, $f(\cdot|\Phi)$, which is another learner, whose parameters Φ are also trained (see figure 17.4):

$$y = f(d_1, d_2, \ldots, d_L|\Phi) \tag{17.8}$$

The combiner learns what the correct output is when the base-learners give a certain output combination. We cannot train the combiner function on the training data because the base-learners may be memorizing the training set; the combiner system should actually learn how the base-learners make errors. Stacking is a means of estimating and correcting for the biases of the base-learners. Therefore, the combiner should be trained on data unused in training the base-learners.

If $f(\cdot|w_1, \ldots, w_L)$ is a linear model with constraints, $w_i \geq 0, \sum_j w_j = 1$, the optimal weights can be found by constrained regression, but of course we do not need to enforce this; in stacking, there is no restriction on the combiner function and unlike voting, $f(\cdot)$ can be nonlinear. For example, it may be implemented as a multilayer perceptron with Φ its connection weights.

The outputs of the base-learners d_j define a new L-dimensional space in which the output discriminant/regression function is learned by the combiner function.

In stacked generalization, we would like the base-learners to be as different as possible so that they will complement each other, and, for this, it is best if they are based on different learning algorithms. If we are combining classifiers that can generate continuous outputs, for example, posterior probabilities, it is better that they be the combined rather than hard decisions.

When we compare a trained combiner as we have in stacking, with a fixed rule such as in voting, we see that both have their advantages: A trained rule is more flexible and may have less bias, but adds extra parameters, risks introducing variance, and needs extra time and data for training. Note also that there is no need to normalize classifier outputs before stacking.

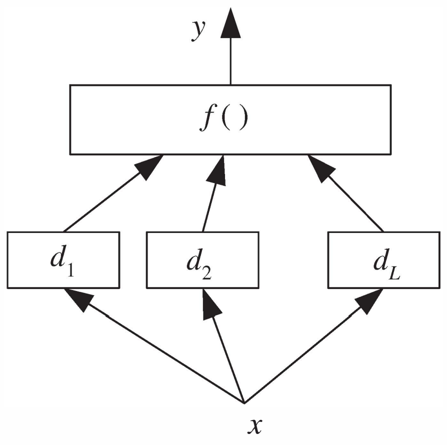

Figure 17.4 In stacked generalization, the combiner is another learner and is not restricted to being a linear combination as in voting.

17.10 Fine-Tuning an Ensemble

Model combination is not a magical formula that is always guaranteed to decrease error; base-learners should be diverse and accurate—that is, they should provide useful information. If a base-learner does not add to accuracy, it can be discarded; also, of the two base-learners that are highly correlated, one is not needed. Note that an inaccurate learner can worsen accuracy, for example, majority voting assumes more than half of the classifiers to be accurate for an input. Therefore, given a set of candidate base-learners, it may not be a good idea to use them as they are, and instead, we may want to do some preprocessing.

We can actually think of the outputs of our base-learners as forming a feature vector for the later stage of combination, and we remember from chapter 6 that we have the same problem with features. Some may be just useless, and some may be highly correlated. Hence, we can use the same ideas of feature selection and extraction here too. Our first approach is to select a subset from the set of base-learners, keeping some and discarding the rest, and the second approach is to define few, new, uncorrelated metalearners from the original base-learners.

17.10.1 Choosing a Subset of the Ensemble

ENSEMBLE SELECTION

Choosing a subset from an ensemble of base-learners is similar to input feature selection, and the possible approaches for *ensemble selection* are the same. We can have a forward/incremental/growing approach where at each iteration, from a set of candidate base-learners, we add to the ensemble the one that most improves accuracy, we can have a backward/decremental/pruning approach where at each iteration, we remove the base-learner whose absence leads to highest improvement, or we can have a floating approach where both additions and removals are allowed.

The combination scheme can be a fixed rule, such as voting, or it can be a trained stacker. Such a selection scheme would not include inaccurate learners, ones that are not diverse enough or are correlated (Caruana et al. 2004; Ruta and Gabrys 2005). So discarding the useless also decreases the overall complexity. Different learners may be using different representations, and such an approach also allows choosing the best complementary representations (Demir and Alpaydın 2005). Note that if we use a decision tree as the combiner, it acts both as a selector and a combiner (Ulaş et al. 2009).

17.10.2 Constructing Metalearners

No matter how we vary the learning algorithms, hyperparameters, resampled folds, or input features, we get positively correlated classifiers (Ulaş, Yıldız, and Alpaydın 2012), and postprocessing is needed to remove this correlation that may be harmful. One possibility is to discard some of the correlated ones, as we discussed earlier; another is to apply a feature extraction method where from the space of the outputs of base-learners, we go to a new, lower-dimensional space where we define uncorrelated metalearners that will also be fewer in number.

Merz (1999) proposes the SCANN algorithm that uses correspondence analysis—a variant of principal components analysis (section 6.3)—on the crisp outputs of base classifiers and combines them using the nearest mean classifier. Actually, any linear or nonlinear feature extraction method we discussed in chapter 6 can be used and its (preferrably continuous) output can be fed to any learner, as we do in stacking.

Let us say we have L learners each having K outputs. Then, for example, using principal component analysis, we can map from the $K \cdot L$-dimensional space to a new space of lower-dimensional, uncorrelated space of “eigenlearners” (Ulaş, Yıldız, and Alpaydın 2012). We can then train the combiner in this new space (using a separate dataset unused to train the base-learners and the dimensionality reducer). Actually, by looking at the coefficients of the eigenvectors, we can also understand the contribution of the base-learners and assess their utility.

It has been shown by Jacobs (1995) that L dependent learners are worth the same as L' independent learners where $L' \leq L$, and this is exactly the idea here. Another point to note is that rather than drastically discarding or keeping a subset of the ensemble, this approach uses all the base-learners, and hence all the information, but at the expense of more computation.

17.11 Cascading

The idea in cascaded classifiers is to have a *sequence* of base-classifiers d_j sorted in terms of their space or time complexity, or the cost of the representation they use, so that d_{j+1} is costlier than d_j (Kaynak and Alpaydın 2000). *Cascading* is a multistage method, and we use d_j only if all preceding learners, $d_k, k < j$ are not confident (see figure 17.5). For this, associated with each learner is a *confidence* w_j such that we say d_j is confident of its output and can be used if $w_j > \theta_j$ where $1/K < \theta_j \leq \theta_{j+1} < 1$ is the confidence threshold. In classification, the confidence function is set to the highest posterior: $w_j \equiv \max_i d_{ji}$; this is the strategy used for rejections (section 3.3).

CASCADING

We use learner d_j if all the preceding learners are not confident:

$$y_i = d_{ji} \text{ if } w_j > \theta_j \text{ and } \forall k < j, w_k < \theta_k \tag{17.9}$$

Starting with $j = 1$, given a training set, we train d_j. Then we find all instances from a separate validation set on which d_j is not confident, and these constitute the training set of d_{j+1}. Note that unlike in AdaBoost, we choose not only the misclassified instances but the ones for which the previous base-learner is not confident. This covers the misclassifications as well as the instances for which the posterior is not high enough; these are instances on the right side of the boundary but for which the distance to the discriminant, namely, the margin, is not large enough.

The idea is that an early simple classifier handles the majority of instances, and a more complex classifier is used only for a small percentage, thereby not significantly increasing the overall complexity. This is

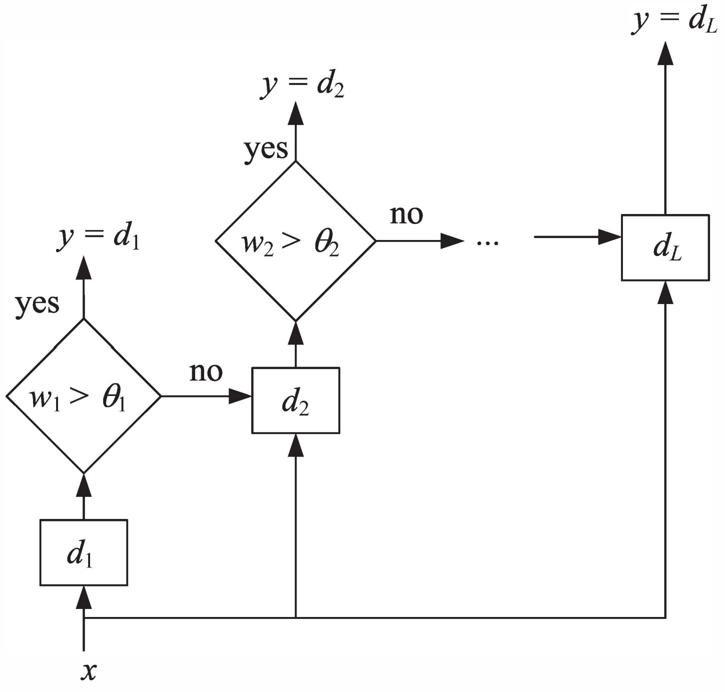

Figure 17.5 Cascading is a multistage method where there is a sequence of classifiers, and the next one is used only when the preceding ones are not confident.

contrary to the multiexpert methods like voting where all base-learners generate their output for any instance. If the problem space is complex, a few base-classifiers may be cascaded increasing the complexity at each stage. In order not to increase the number of base-classifiers, the few instances not covered by any are stored as they are and are treated by a nonparametric classifier, such as *k*-NN.

The inductive bias of cascading is that the classes can be explained by a small number of “rules” in increasing complexity, with an additional small set of “exceptions” not covered by the rules. The rules are implemented by simple base-classifiers, for example, perceptrons of increasing complexity, which learn general rules valid over the whole input space. Exceptions are localized instances and are best handled by a nonparametric model.

Cascading thus stands between the two extremes of parametric and nonparametric classification. The former—for example, a linear model—finds a single rule that should cover all the instances. A nonparametric classifier—for example, *k*-NN—stores the whole set of instances without generating any simple rule explaining them. Cascading generates a rule (or rules) to explain a large part of the instances as cheaply as possible and stores the rest as exceptions. This makes sense in a lot of learning applications. For example, most of the time the past tense of a verb in English is found by adding a "–d" or "–ed" to the verb; there are also irregular verbs—for example, "go"/"went"—that do not obey this rule.

17.12 Notes

The idea in combining learners is to divide a complex task into simpler tasks that are handled by separately trained base-learners. Each base-learner has its own task. If we had a large learner containing all the base-learners, then it would risk overfitting. For example, consider taking a vote over three multilayer perceptrons, each with a single hidden layer. If we combine them all together with the linear model combining their outputs, this is a large multilayer perceptron with two hidden layers. If we train this large model with the whole sample, it very probably overfits. When we train the three multilayer perceptrons separately, for example, using ECOC, bagging, and so forth, it is as if we define a required output for the second-layer hidden nodes of the large multilayer perceptron. This puts a constraint on what the overall learner should learn and simplifies learning.

One disadvantage of combining is that the combined system is not interpretable. For example, even though decision trees are interpretable, bagged or boosted trees are not interpretable. Error-correcting codes with their weights as $-1/0/+1$ allow some form of interpretability. Mayoraz and Moreira (1997) discuss incremental methods for learning the error-correcting output codes where base-learners are added when needed. Allwein, Schapire, and Singer (2000) discuss various methods for coding multiclass problems as two-class problems. Alpaydın and Mayoraz (1999) consider the application of ECOC where linear base-learners are combined to get nonlinear discriminants, and they also propose methods to learn the ECOC matrix from data.

The earliest and most intuitive approach is voting. Kittler et al. (1998) give a review of fixed rules and also discuss an application where multiple representations are combined. The task is person identification using three representations: frontal face image, face profile image, and voice. The error rate of the voting model is lower than the error rates when a single representation is used. Another application is given in Alimoğlu and Alpaydın 1997 where for improved handwritten digit recognition, two sources of information are combined: One is the temporal pen movement data as the digit is written on a touch-sensitive pad, and the other is the static two-dimensional bitmap image once the digit is written. In that application, the two classifiers using either of the two representations have around 5 percent error, but combining the two reduces the error rate to 3 percent. It is also seen that the critical stage is the design of the complementary learners and/or representations, the way they are combined is not as critical.

BIOMETRICS

Combining different modalities is used in *biometrics*, where the aim is authentication using different input sources, fingerprint, signature, face, and so on. In such a case, different classifiers use these modalities separately and their decisions are combined. This both improves accuracy and makes *spoofing* more difficult.

Noble (2004) makes a distinction between three type of combination strategies when we have information coming from multiple sources in different representations or modalities:

- In *early integration*, all these inputs are concatenated to form a single vector that is then fed to a single classifier. Previously we discussed why this is not a very good idea.

- In *late integration*, which we advocated in this chapter, different inputs are fed to separate classifiers whose outputs are then combined, by voting, stacking, or any other method we discussed.

MULTIPLE KERNEL LEARNING

- Kernel algorithms, which we discussed in chapter 13, allow a different method of integration that Noble (2004) calls *intermediate integration*, as being between early and late integration. This is the *multiple kernel learning* approach (see section 13.8) where there is a single kernel machine classifier that uses multiple kernels for different inputs and the combination is not in the input space as in early integration, or in the space of decisions as in late integration, but in the space of the basis functions that define the kernels. For different sources, there are different notions of similarity calculated by their kernels, and the classifier accumulates and uses them.

Some ensemble methods such as voting are similar to Bayesian averaging (chapter 16). For example when we do bagging and train the same model on different resampled training sets, we may consider them as being samples from the posterior distribution, but other combination methods such as mixture of experts and stacking go much beyond averaging over parameters or models.

When we are combining multiple views/representations, concatenating them is not really a good idea but one interesting possibility is to do some combined dimensionality reduction. We can consider a generative model (section 14.3) where we assume that there is a set of latent factors that generate these multiple views in parallel, and from the observed views, we can go back to that latent space and do classification there (Chen et al. 2012).

Combining multiple learners has been a popular topic in machine learning since the early 1990s, and research has been going on ever since. Kuncheva (2004) discusses different aspects of classifier combination; the book also includes a section on combination of multiple clustering results.

AdaBoosted decision trees are considered to be one of the best machine learning algorithms. There are also versions of AdaBoost where the next base-learner is trained on the residual of the previous base-learner (Hastie, Tibshirani, and Friedman 2001). Recently, it has been noticed that ensembles do not always improve accuracy and research has started to focus on the criteria that a good ensemble should satisfy or how to form a good one. A survey of the role of diversity in ensembles is given in Kuncheva 2005.

17.13 Exercises

1. If each base-learner is iid and correct with probability $p > 1/2$, what is the probability that a majority vote over L classifiers gives the correct answer?

   SOLUTION: It is given by a binomial distribution (see figure 17.6).

$$P(X \geq \lfloor L/2 \rfloor + 1) = \sum_{i=\lfloor L/2 \rfloor + 1}^{L} \binom{L}{i} p^i (1-p)^{L-i}$$

2. In bagging, to generate the L training sets, what would be the effect of using L-fold cross-validation instead of bootstrap?
3. Propose an incremental algorithm for learning error-correcting output codes

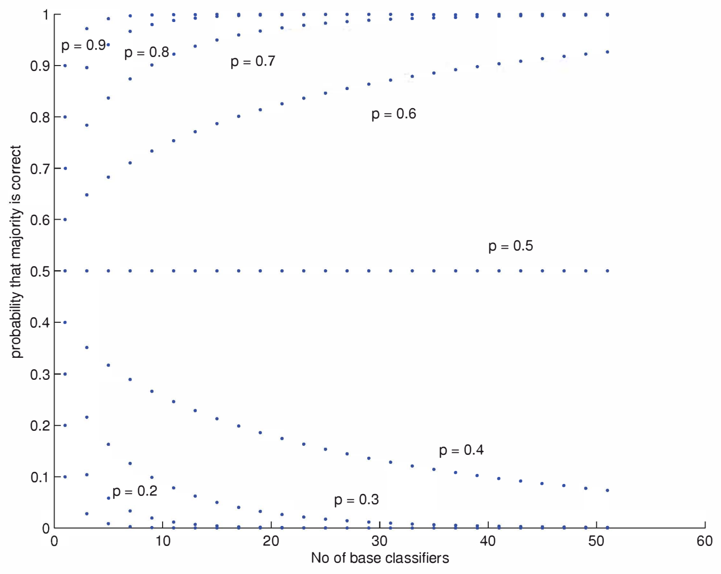

Figure 17.6 Probability that a majority vote is correct as a function of the number of base-learners for different p. The probabity increases only for $p > 0.5$.

where new two-class problems are added as they are needed to better solve the multiclass problem.

4. In the mixture of experts architecture, we can have different experts use different input representations. How can we design the gating network in such a case?

5. Propose a dynamic regressor selection algorithm.

6. What is the difference between voting and stacking using a linear perceptron as the combiner function?

   SOLUTION: If the voting system is also trained, the only difference would be that with stacking, the weights need not be positive or sum up to 1, and there is also a bias term. Of course, the main advantage of stacking is when the combiner is nonlinear.

7. In cascading, why do we require $\theta_{j+1} \geq \theta_j$?

   SOLUTION: Instances on which the confidence is less than θ_j have already been filtered out by d_j; we require the threshold to increase so that we can have higher confidences.

8. To be able to use cascading for regression, during testing, a regressor should

be able to say whether it is confident of its output. How can we implement this?

9. How can we combine the results of multiple clustering solutions?

   SOLUTION: The easiest is the following: Let us take any two training instances. Each clustering solution either places them in the same cluster or not; denote it as 1 and 0. The average of these counts over all clustering solutions is the overall probability that those two are in the same cluster (Kuncheva 2004).

10. In section 17.10, we discuss that if we use a decision tree as a combiner in stacking, it works both as a selector and a combiner. What are the other advantages and disadvantages?

    SOLUTION: A tree uses only a subset of the classifiers and not the whole. Using the tree is fast, and we need to evaluate only the nodes on our path, which may be short. See Ulaş et al. 2009 for more detail. The disadvantage is that the combiner cannot look at combinations of classifier decisions (assuming that the tree is univariate). Using a subset may also be harmful; we do not get the redundancy we need if some classifiers are faulty.

17.14 References

Alimoğlu, F., and E. Alpaydın. 1997. “Combining Multiple Representations and Classifiers for Pen-Based Handwritten Digit Recognition.” In *Fourth International Conference on Document Analysis and Recognition*, 637–640. Los Alamitos, CA: IEEE Computer Society.

Allwein, E. L., R. E. Schapire, and Y. Singer. 2000. “Reducing Multiclass to Binary: A Unifying Approach for Margin Classifiers.” *Journal of Machine Learning Research* 1:113–141.

Alpaydın, E. 1997. “Voting over Multiple Condensed Nearest Neighbors.” *Artificial Intelligence Review* 11:115–132.

Alpaydın, E., and E. Mayoraz. 1999. “Learning Error-Correcting Output Codes from Data.” In *Ninth International Conference on Artificial Neural Networks*, 743–748. London: IEE Press.

Avnimelech, R., and N. Intrator. 1997. “Boosting Regression Estimators.” *Neural Computation* 11:499–520.

Breiman, L. 1996. “Bagging Predictors.” *Machine Learning* 26:123–140.

Caruana, R., A. Niculescu-Mizil, G. Crew, and A. Ksikes. 2004. “Ensemble Selection from Libraries of Models.” In *Twenty-First International Conference on Machine Learning*, ed. C. E. Brodley, 137–144. New York: ACM.

Chen, N., J. Zhu, F. Sun, and E. P. Xing. 2012. “Large-Margin Predictive Latent Subspace Learning for Multiview Data Analysis.” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34:2365–2378.

Demir, C., and E. Alpaydın. 2005. “Cost-Conscious Classifier Ensembles.” *Pattern Recognition Letters* 26:2206–2214.

Dietterich, T. G., and G. Bakiri. 1995. “Solving Multiclass Learning Problems via Error-Correcting Output Codes.” *Journal of Artificial Intelligence Research* 2:263–286.

Drucker, H. 1997. “Improving Regressors using Boosting Techniques.” In *Fourteenth International Conference on Machine Learning*, ed. D. H. Fisher, 107–115. San Mateo, CA: Morgan Kaufmann.

Drucker, H., C. Cortes, L. D. Jackel, Y. Le Cun, and V. Vapnik. 1994. “Boosting and Other Ensemble Methods.” *Neural Computation* 6:1289–1301.

Freund, Y., and R. E. Schapire. 1996. “Experiments with a New Boosting Algorithm.” In *Thirteenth International Conference on Machine Learning*, ed. L. Saitta, 148–156. San Mateo, CA: Morgan Kaufmann.

Hansen, L. K., and P. Salamon. 1990. “Neural Network Ensembles.” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 12:993–1001.

Hastie, T., R. Tibshirani, and J. Friedman. 2001. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* New York: Springer.

Ho, T. K. 1998. “The Random Subspace Method for Constructing Decision Forests.” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20:832–844.

Jacobs, R. A. 1995. “Methods for Combining Experts’ Probability Assessments.” *Neural Computation* 7:867–888.

Jacobs, R. A. 1997. “Bias/Variance Analyses for Mixtures-of-Experts Architectures.” *Neural Computation* 9:369–383.

Jain, A., K. Nandakumar, and A. Ross. 2005. “Score Normalization in Multimodal Biometric Systems.” *Pattern Recognition* 38:2270–2285.

Kaynak, C., and E. Alpaydın. 2000. “MultiStage Cascading of Multiple Classifiers: One Man’s Noise is Another Man’s Data.” In *Seventeenth International Conference on Machine Learning*, ed. P. Langley, 455–462. San Francisco: Morgan Kaufmann.

Kittler, J., M. Hatef, R. P. W. Duin, and J. Matas. 1998. “On Combining Classifiers.” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20:226–239.

Kuncheva, L. I. 2004. *Combining Pattern Classifiers: Methods and Algorithms.* Hoboken, NJ: Wiley.

Kuncheva, L. I. 2005. Special issue on Diversity in Multiple Classifier Systems. *Information Fusion* 6:1–115.

Mayoraz, E., and M. Moreira. 1997. “On the Decomposition of Polychotomies into Dichotomies.” In *Fourteenth International Conference on Machine Learning*, ed. D. H. Fisher, 219–226. San Mateo, CA: Morgan Kaufmann.

Merz, C. J. 1999. “Using Correspondence Analysis to Combine Classifiers.” *Machine Learning* 36:33–58.

Noble, W. S. 2004. “Support Vector Machine Applications in Computational Biology.” In *Kernel Methods in Computational Biology*, ed. B. Schölkopf, K. Tsuda, and J.-P. Vert, 71–92. Cambridge, MA: MIT Press.

Özen, A., M. Gönen, E. Alpaydın, and T. Haliloğlu. 2009. “Machine Learning Integration for Predicting the Effect of Single Amino Acid Substitutions on Protein Stability.” *BMC Structural Biology* 9 (66): 1–17.

Perrone, M. P. 1993. “Improving Regression Estimation: Averaging Methods for Variance Reduction with Extensions to General Convex Measure.” Ph.D. thesis, Brown University.

Ruta, D., and B. Gabrys. 2005. “Classifier Selection for Majority Voting.” *Information Fusion* 6:63–81.

Schapire, R. E. 1990. “The Strength of Weak Learnability.” *Machine Learning* 5:197–227.

Schapire, R. E., Y. Freund, P. Bartlett, and W. S. Lee. 1998. “Boosting the Margin: A New Explanation for the Effectiveness of Voting Methods.” *Annals of Statistics* 26:1651–1686.

Ulaş, A., M. Semerci, O. T. Yıldız, and E. Alpaydın. 2009. “Incremental Construction of Classifier and Discriminant Ensembles.” *Information Sciences* 179:1298–1318.

Ulaş, A., O. T. Yıldız, and E. Alpaydın. 2012. “Eigenclassifiers for Combining Correlated Classifiers.” *Information Sciences* 187:109–120.

Wolpert, D. H. 1992. “Stacked Generalization.” *Neural Networks* 5:241–259.

Woods, K., W. P. Kegelmeyer Jr., and K. Bowyer. 1997. “Combination of Multiple Classifiers Using Local Accuracy Estimates.” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 19:405–410.