Combining Classifiers

Component classifiers with discriminant functions

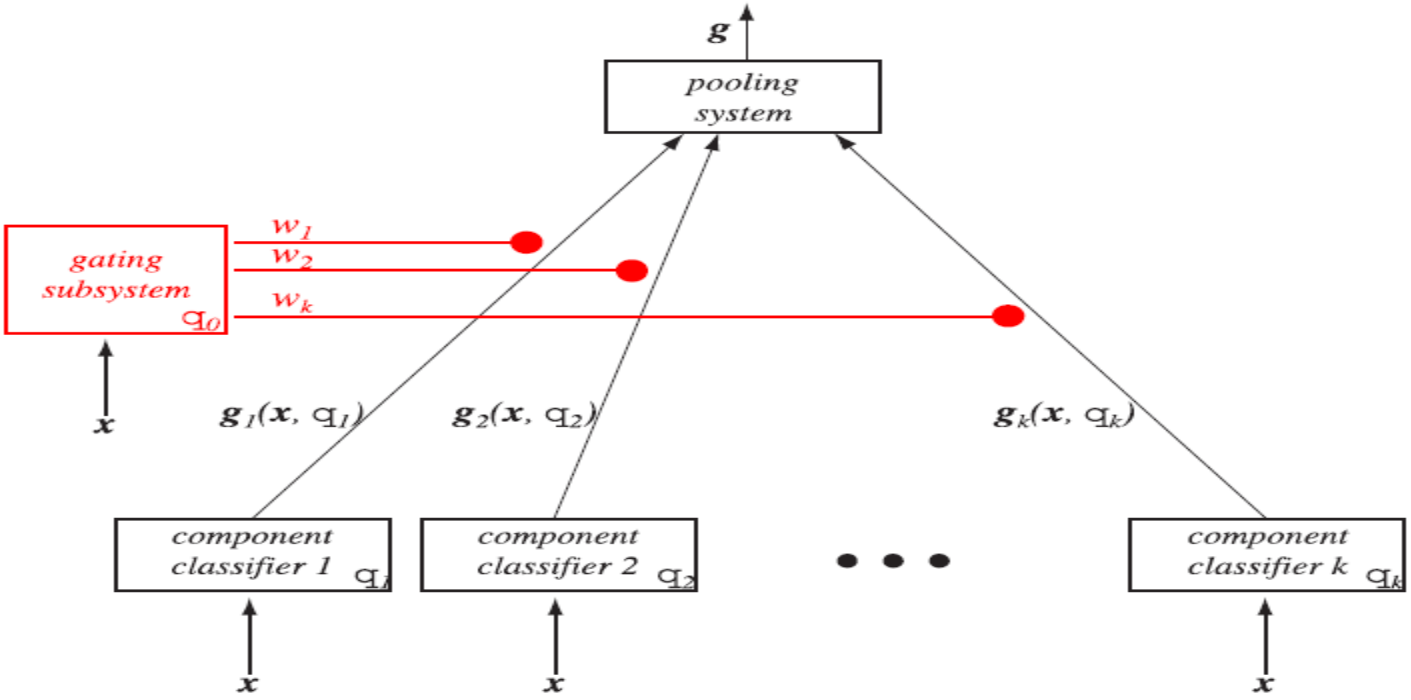
We assume that each pattern is produced by a *mixture model*, in which first some fundamental process or function indexed by r (where $1 \leq r \leq k$) is randomly chosen according to distribution $P(r|\mathbf{x}, \boldsymbol{\theta}_0^0)$ where $\boldsymbol{\theta}_0^0$ is a parameter vector. Next, the selected process r emits an output $\mathbf{y}$ (e.g., a category label) according to $P(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}_r^0)$, where the parameter vector $\boldsymbol{\theta}_r^0$ describes the state of the process. (The superscript 0 indicates the properties of the generating model. Below, terms without this superscript refer to the parameters in a classifier.) The overall probability of producing output $\mathbf{y}$ is then the sum over all the processes according to:

$$P(\mathbf{y}|\mathbf{x}, \boldsymbol{\Theta}^0) = \sum_{r=1}^{k} P(r|\mathbf{x}, \boldsymbol{\eta}^0) P(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}^0), \tag{54}$$

where $\boldsymbol{\Theta}^0 = [\boldsymbol{\theta}_0^0, \boldsymbol{\theta}_1^0, \ldots, \boldsymbol{\theta}_k^0]^t$ represents the vector of all relevant parameters. Equation 54 describes a *mixture density*, which could be discrete or continuous (Chap. **??**).

Figure 9.19 shows the basic architecture of an ensemble classifier whose task is to classify a test pattern $\mathbf{x}$ into one of c categories; this architecture matches the assumed mixture model. A test pattern $\mathbf{x}$ is presented to each of the k component classifiers, each of which emits c scalar discriminant values, one for each category. The c discriminant values from component classifier r are grouped and marked $\mathbf{g}(\mathbf{x}, \boldsymbol{\theta}_r)$ in the figure, with

$$\sum_{j=1}^{c} g_{rj} = 1 \text{ for all } r. \tag{55}$$



All discriminant values from component classifier r are multiplied by a scalar weight w_r, governed by the *gating subsystem*, which has a parameter vector $\boldsymbol{\theta}_0$. Below we shall use the conditional mean of the mixture density, which can be calculated from Eq. 54

$$\boldsymbol{\mu} = \mathcal{E}[\mathbf{y}|\mathbf{x}, \boldsymbol{\Theta}] = \sum_{r=1}^{k} w_r \boldsymbol{\mu}_r \tag{56}$$

where $\boldsymbol{\mu}_r$ is the conditional mean associated with $P(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}_r^0)$.

The mixture-of-experts architecture is trained so that each component classifier models a corresponding process in the mixture model, and the gating subsystem models the mixing parameters $P(r|\mathbf{x}, \boldsymbol{\theta}_0^0)$ in Eq. 54. The goal is to find parameters that maximize the log-likelihood for n training patterns $\mathbf{x}^1, \ldots \mathbf{x}^n$ in set $\mathcal{D}$:

$$l(\mathcal{D}, \boldsymbol{\Theta}) = \sum_{i=1}^{n} ln \left(\sum_{r=1}^{k} P(r|\mathbf{x}^i, \boldsymbol{\theta}_0) P(\mathbf{y}^i|\mathbf{x}^i, \boldsymbol{\theta}_r) \right). \tag{57}$$

A straightforward approach is to use gradient descent on the parameters, where the derivatives are (Problem 43)

$$\frac{\partial l(\mathcal{D}, \boldsymbol{\Theta})}{\partial \boldsymbol{\mu}_r} = \sum_{i=1}^{n} P(r|\mathbf{y}^i, \mathbf{x}^i) \frac{\partial}{\partial \boldsymbol{\mu}_r} ln[P(\mathbf{y}^i|\mathbf{x}^i, \boldsymbol{\theta}_r)] \text{ for } r = 1, \ldots k \tag{58}$$

and

$$\frac{\partial l(\mathcal{D}, \boldsymbol{\Theta})}{\partial g_r} = \sum_{i=1}^{n} (P(r|\mathbf{y}^i, \mathbf{x}^i) - w_r^i). \tag{59}$$

Here ($P(r|\mathbf{y}^i, \mathbf{x}^i)$ is the posterior probability of process r conditional on the input and output being $\mathbf{x}^i$ and $\mathbf{y}^i$, respectively. Moreover, w_r^i is the prior probability $P(r|\mathbf{x}^i)$ that process r is chosen given the input is $\mathbf{x}^i$. Gradient descent according to Eq. 59 moves the prior probabilities to the posterior probabilities. The Expectation-Maximization (EM) algorithm can be used to train this architecture as well (Chap. **??**).

The final decision rule is simply to choose the category corresponding to the maximum discriminant value after the pooling system. An alternative, *winner-take-all* method is to use the decision of the single component classifier that is "most confident," i.e., has the largest single discriminant value g_{rj}. While the winner-take-all method is provably sub-optimal, it nevertheless is simple and can work well if the component classifiers are experts in separate regions of the input space.

Component classifiers without discriminant functions

Occasionally we seek to form an ensemble classifier from highly trained component classifiers, some of which might not themselves compute discriminant functions. For instance, we might have four component classifiers — a k-nearest-neighbor classifier, a decision tree, a neural network, and a rule-based system — all addressing the same problem. While a neural network would provide analog values for each of the c categories, the rule-based system would give only a single category label (i.e., a one-of-c representation) and the k-nearest neighbor classifier would give only rank order of the categories.

In order to integrate the information from the component classifiers we must convert the their outputs into discriminant values obeying the constraint of Eq. 55 so we can use the framework of Fig. 9.19. The simplest heuristics to this end are the following:

Analog If the outputs of a component classifier are analog values $\tilde{g}_i$, we can use the *softmax* transformation,

$$g_i = \frac{e^{\tilde{g}_i}}{\sum_{j=1}^{c} e^{\tilde{g}_i}}. \tag{60}$$

to convert them to values g_i.

Rank order If the output is a rank order list, we assume the discriminant function is linearly proportional to the rank order of the item on the list. Of course, the resulting g_i should then be properly normalized, and thus sum to 1.0.

One-of-c If the output is a one-of-c representation, in which a single category is identified, we let $g_j = 1$ for the j corresponding to the chosen category, and 0 otherwise.

Analog value		Rank order		One-of-c	
$\tilde{g}_i$	g_i	$\tilde{g}_i$	g_i	$\tilde{g}_i$	g_i
0.4	0.158	3rd	$4/21 = 0.194$	0	0
0.6	0.193	6th	$1/21 = 0.048$	1	1.0
0.9	0.260	5th	$2/21 = 0.095$	0	0
0.3	0.143	1st	$6/21 = 0.286$	0	0
0.2	0.129	2nd	$5/21 = 0.238$	0	0
0.1	0.111	4th	$3/21 = 0.143$	0	0

Once the outputs of the component classifiers have been converted to effective discriminant functions in this way, the component classifiers are themselves held fixed, but the gating network is trained as described in Eq. 59. This method is particularly useful when several highly trained component classifiers are pooled to form a single decision.

Bagging

Bagging — a name derived from "bootstrap aggregation" — uses multiple versions of a training set, each created by drawing $n' < n$ samples from $\mathcal{D}$ with replacement. Each of these bootstrap data sets is used to train a different *component classifier* and the final classification decision is based on the vote of each component classifier.* Traditionally the component classifiers are of the same general form — i.e., all hidden Markov models, or all neural networks, or all decision trees — merely the final parameter values differ among them due to their different sets of training patterns.

A classifier/learning algorithm combination is informally called *unstable* if "small" changes in the training data lead to significantly different classifiers and relatively "large" changes in accuracy.

In general, bagging improves recognition for unstable classifiers since it effectively averages over such discontinuities. There are no convincing theoretical derivations or simulation studies showing that bagging will help all stable classifiers, however.

Bagging is our first encounter with multiclassifier systems, where a final overall classifier is based on the outputs of a number of component classifiers. The global decision rule in bagging — a simple vote among the component classifiers — is the most elementary method of pooling or integrating the outputs of the component classifiers.

Boosting

The goal of boosting is to improve the accuracy of any given learning algorithm. In boosting we first create a classifier with accuracy on the training set greater than average, and then add new component classifiers to form an ensemble whose joint decision rule has arbitrarily high accuracy on the training set. In such a case we say the classification performance has been "boosted." In overview, the technique trains successive component classifiers with a subset of the training data that is "most informative" given the current set of component classifiers. Classification of a test point $\mathbf{x}$ is based on the outputs of the component classifiers, as we shall see.

For definiteness, consider creating three component classifiers for a two-category problem through boosting. First we randomly select a set of $n_1 < n$ patterns from the full training set $\mathcal{D}$ (without replacement); call this set $\mathcal{D}_1$. Then we train the first classifier, C_1, with $\mathcal{D}_1$. Classifier C_1 need only be a *weak learner*, i.e., have accuracy only slightly better than chance. (Of course, this is the minimum requirement; a weak learner could have high accuracy on the training set. In that case the benefit

of boosting will be small.) Now we seek a second training set, $\mathcal{D}_2$, that is the "most informative" given component classifier C_1. Specifically, half of the patterns in $\mathcal{D}_2$ should be correctly classified by C_1, half incorrectly classified by C_1 (Problem 29). Such an informative set $\mathcal{D}_2$ is created as follows: we flip a fair coin. If the coin is heads, we select remaining samples from $\mathcal{D}$ and present them, one by one to C_1 until C_1 misclassifies a pattern. We add this misclassified pattern to $\mathcal{D}_2$. Next we flip the coin again. If heads, we continue through $\mathcal{D}$ to find another pattern misclassified by C_1 and add it to $\mathcal{D}_2$ as just described; if tails we find a pattern which C_1 classifies correctly. We continue until no more patterns can be added in this manner. Thus half of the patterns in $\mathcal{D}_2$ are correctly classified by C_1, half are not. As such $\mathcal{D}_2$ provides information complementary to that represented in C_1. Now we train a second component classifier C_2 with $\mathcal{D}_2$.

Next we seek a third data set, $\mathcal{D}_3$, which is not well classified by the combined system C_1 and C_2. We randomly select a training pattern from those remaining in $\mathcal{D}$, and classify that pattern with C_1 and with C_2. If C_1 and C_2 disagree, we add this pattern to the third training set $\mathcal{D}_3$; otherwise we ignore the pattern. We continue adding informative patterns to $\mathcal{D}_3$ in this way; thus $\mathcal{D}_3$ contains those not well represented by the combined decisions of C_1 and C_2. Finally, we train the last component classifier, C_3, with the patterns in $\mathcal{D}_3$.

AdaBoost

There are a number of variations on basic boosting. The most popular, AdaBoost — from "adaptive" boosting — allows the designer to continue adding weak learners until some desired low training error has been achieved. In AdaBoost each training pattern receives a weight which determines its probability of being selected for a training set for an individual component classifier. If a training pattern is accurately classified, then its chance of being used again in a subsequent component classifier is reduced; conversely, if the pattern is not accurately classified, then its chance of being used again is raised. In this way, AdaBoost "focuses in" on the informative or "difficult" patterns. Specifically, we initialize these weights across the training set to to be uniform. On each iteration k, we draw a training set at random according to these weights, and train component classifier C_k on the patterns selected. Next we increase weights of training patterns misclassified by C_k and decrease weights of the patterns correctly classified by C_k. Patterns chosen according to this new distribution are used to train the next classifier, C_{k+1}, and the process is iterated.

Algorithm 1 (AdaBoost)

1 **begin initialize** $\mathcal{D} = \{\mathbf{x}^1, y_1, \mathbf{x}^2, y_2, \ldots, \mathbf{x}^n, y_n\}, k_{max}, W_1(i) = 1/n, i = 1, \ldots, n$
2 $k \leftarrow 0$
3 **do** $k \leftarrow k + 1$
4 Train weak learner C_k using $\mathcal{D}$ sampled according to distribution $W_k(i)$
5 $E_k \leftarrow$ Training error of C_k measured on $\mathcal{D}$ using $W_k(i)$
6 $\alpha_k \leftarrow \frac{1}{2}\ln[(1 - E_k)/E_k]$

7 $$W_{k+1}(i) \leftarrow \frac{W_k(i)}{Z_k} \times \begin{cases} e^{-\alpha_k} & \text{if } h_k(\mathbf{x}^i) = y_i \text{ (correctly classified)} \\ e^{\alpha_k} & \text{if } h_k(\mathbf{x}^i) \neq y_i \text{ (incorrectly classified)} \end{cases}$$

8 **until** $k = k_{max}$
9 **return** C_k and α_k for $k = 1$ to k_{max} (ensemble of classifiers with weights)
10 **end**

Example ADA-Boost

Sample indices	x	y	Weights	$\hat{y}(x <= 3.0)$?	Correct?	Updated weights
1	1.0	1	0.1	1	Yes	0.072
2	2.0	1	0.1	1	Yes	0.072
3	3.0	1	0.1	1	Yes	0.072
4	4.0	-1	0.1	-1	Yes	0.072
5	5.0	-1	0.1	-1	Yes	0.072
6	6.0	-1	0.1	-1	Yes	0.072
7	7.0	1	0.1	-1	Yes	0.167
8	8.0	1	0.1	-1	Yes	0.167
9	9.0	1	0.1	-1	Yes	0.167
10	10.0	-1	0.1	-1	Yes	0.072

$$\varepsilon = 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 0$$

$$+ 0.1\times 0 = \frac{3}{10} = 0.3$$

Next we compute the coefficient α_j (shown in step 6), which is later used in step 7 to update the weights as well as for the weights in majority vote prediction (step 10):

$$\alpha_j = \frac{0.5\log(1-\varepsilon)}{\varepsilon} \approx 0.424$$

$$\boldsymbol{w} := \boldsymbol{w} \times \exp\left(-\alpha_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y}\right)$$

$$0.1\times\exp\left(-0.424\times 1\times 1\right) \approx 0.066$$

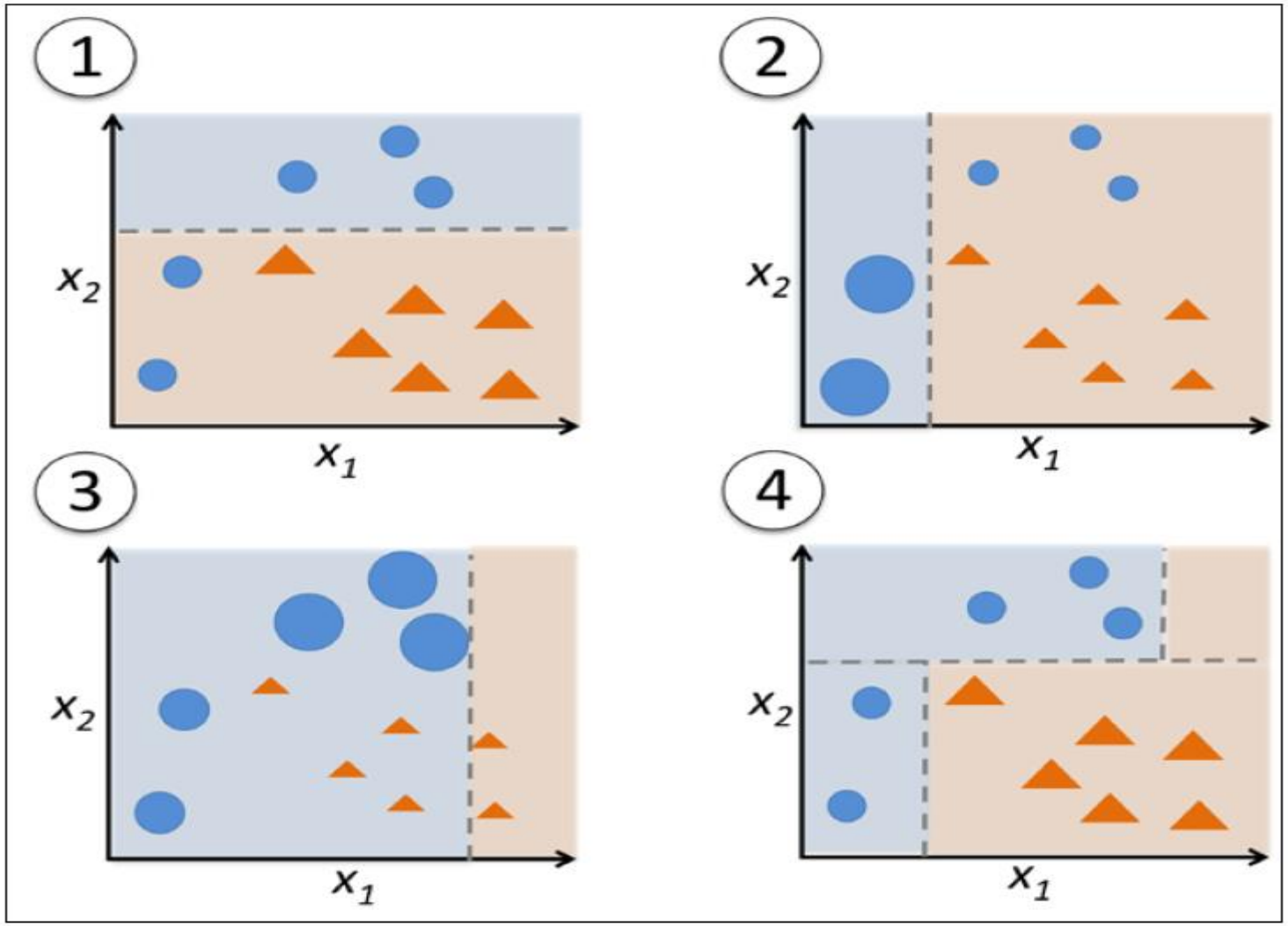
$$0.1\times\exp\left(-0.424\times 1\times(-1)\right) \approx 0.153$$

$$0.1\times\exp\left(-0.424\times(-1)\times(1)\right) \approx 0.153$$

$$\boldsymbol{w} := \frac{\boldsymbol{w}}{\sum_i w_i}$$

Here, $\sum_i w_i = 7\times 0.065 + 3\times 0.153 = 0.914$.

Example



Estimating and comparing classifiers

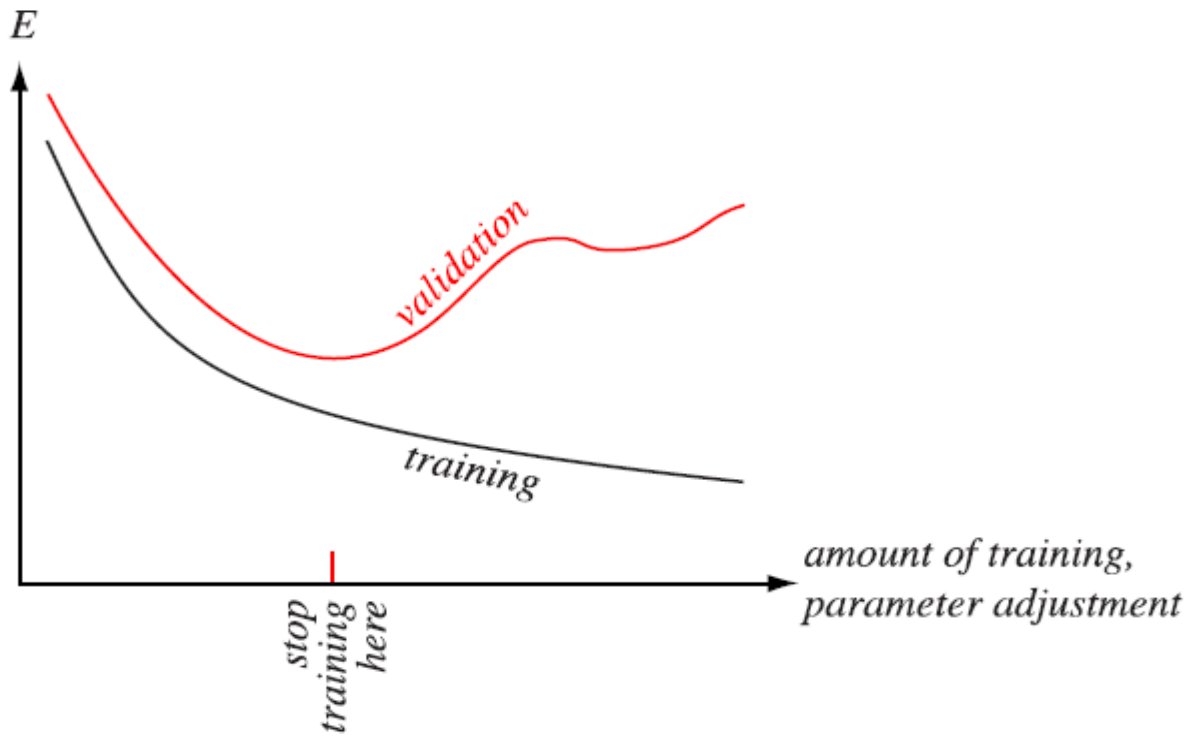
There are at least two reasons for wanting to know the generalization rate of a classifier on a given problem. One is to see if the classifier performs well enough to be useful; another is to compare its performance with that of a competing design. Estimating the final generalization performance invariably requires making assumptions about the classifier or the problem or both, and can fail if the assumptions are not valid. We should stress, then, that all the following methods are heuristic. Indeed, if there were a foolproof method for choosing which of two classifiers would generalize better on an arbitrary new problem, we could incorporate such a method into the learning and violate the No Free Lunch Theorem. Occasionally our assumptions are explicit (as in parametric models), but more often than not they are implicit and difficult to identify or relate to the final estimation (as in empirical methods).

Parametric models

One approach to estimating the generalization rate is to compute it from the assumed parametric model. For example, in the two-class multivariate normal case, we might estimate the probability of error using the Bhattacharyya or Chernoff bounds (Chap **??**), substituting estimates of the means and the covariance matrix for the unknown parameters. However, there are three problems with this approach. First, such an error estimate is often overly optimistic; characteristics that make the training samples peculiar or unrepresentative will not be revealed. Second, we should always suspect the validity of an assumed parametric model; a performance evaluation based on the same model cannot be believed unless the evaluation is unfavorable. Finally, in more general situations where the distributions are not simple it is very difficult to compute the error rate exactly, even if the probabilistic structure is known completely.

Cross validation

In cross validation we randomly split the set of labeled training samples $\mathcal{D}$ into two parts: one is used as the traditional training set for adjusting model parameters in the classifier. The other set — the *validation set* — is used to estimate the generalization error. Since our ultimate goal is low generalization error, we train the classifier until we reach a minimum of this validation error, as sketched in Fig. 9.9. It is essential that the validation (or the test) set not include points used for training the parameters in the classifier — a methodological error known as "testing on the training set."*



Cross validation can be applied to virtually every classification method, where the specific form of learning or parameter adjustment depends upon the general training method. For example, in neural networks of a fixed topology (Chap. **??**), the amount of training is the number of epochs or presentations of the training set. Alternatively, the number of hidden units can be set via cross validation. Likewise, the width of the Gaussian window in Parzen windows (Chap. **??**), and an optimal value of k in the k-nearest neighbor classifier (Chap. **??**) can be set by cross validation.

Cross validation is heuristic and need not (indeed cannot) give improved classifiers in every case. Nevertheless, it is extremely simple and for many real-world problems is found to improve generalization accuracy. There are several heuristics for choosing the portion γ of $\mathcal{D}$ to be used as a validation set ($0 < \gamma < 1$). Nearly always, a smaller portion of the data should be used as validation set ($\gamma < 0.5$) because the validation set is used merely to set a *single* global property of the classifier (i.e., when to stop adjusting parameters) rather than the large number of classifier parameters learned using the training set. If a classifier has a large number of free parameters or degrees of freedom, then a larger portion of $\mathcal{D}$ should be used as a training set, i.e., γ should be reduced. A traditional default is to split the data with $\gamma = 0.1$, which has proven effective in many applications. Finally, when the number of degrees of freedom in the classifier is small compared to the number of training points, the predicted generalization error is relatively insensitive to the choice of γ.

A simple generalization of the above method is *m-fold cross validation.* Here the training set is randomly divided into m disjoint sets of equal size n/m, where n is again the total number of patterns in $\mathcal{D}$. The classifier is trained m times, each time with a different set held out as a validation set. The estimated performance is the mean of these m errors. In the limit where $m = n$, the method is in effect the leave-one-out approach to be discussed in Sect. 9.6.3.

We emphasize that cross validation is a heuristic and need not work on every problem. Indeed, there are problems for which *anti-cross validation* is effective — halting the adjustment of parameters when the validation error is the first local *maximum.* As such, in any particular problem designers must be prepared to explore different values of γ, and possibly abandon the use of cross validation altogether if performance cannot be improved (Computer exercise 5).

Cross validation is, at base, an empirical approach that tests the classifier experimentally. Once we train a classifier using cross validation, the validation error gives an estimate of the accuracy of the final classifier on the unknown test set. If the true but unknown error rate of the classifier is p, and if k of the n' independent, randomly drawn test samples are misclassified, then k has the binomial distribution

$$P(k) = \binom{n'}{k} p^k (1-p)^{n'-k}. \tag{38}$$

Thus, the fraction of test samples misclassified is exactly the maximum likelihood estimate for p (Problem 39):

$$\hat{p} = \frac{k}{n'}. \tag{39}$$

Jackknife and bootstrap estimation of classification accuracy

A method for comparing classifiers closely related to cross validation is to use the jackknife or bootstrap estimation procedures (Sects. 9.4.1 & 9.4.2). The application of the jackknife approach to classification is straightforward. We estimate the accuracy of a given algorithm by training the classifier n separate times, each time using the training set $\mathcal{D}$ from which a different single training point has been deleted. This is merely the $m = n$ limit of m-fold cross validation. Each resulting classifier is tested on the single deleted point and the jackknife estimate of the accuracy is then simply the mean of these leave-one-out accuracies. Here the computational complexity may be very high, especially for large n (Problem 28).

The jackknife, in particular, generally gives good estimates, since each of the the n classifiers is quite similar to the classifier being tested (differing solely due to a single training point). Likewise, the jackknife estimate of the variance of this estimate is given by a simple generalization of Eq. 32. A particular benefit of the jackknife approach is that it can provide measures of confidence or statistical significance in the comparison between two classifier designs. Suppose trained classifier C_1 has an accuracy of 80% while C_2 has accuracy of 85%, as estimated by the jackknife procedure. Is C_2 really better than C_1? To answer this, we calculate the jackknife estimate of the variance of the classification accuracies and use traditional hypothesis testing to see if C_1's apparent superiority is statistically significant (Fig. 9.11).

There are several ways to generalize the bootstrap method to the problem of estimating the accuracy of a classifier. One of the simplest approaches is to train B classifiers, each with a different bootstrap data set, and test on other bootstrap data sets. The bootstrap estimate of the classifier accuracy is simply the mean of these bootstrap accuracies. In practice, the high computational complexity of bootstrap estimation of classifier accuracy is rarely worth possible improvements in that estimate. In Sect. 9.5.1 we shall discuss bagging, a useful modification of bootstrap estimation.

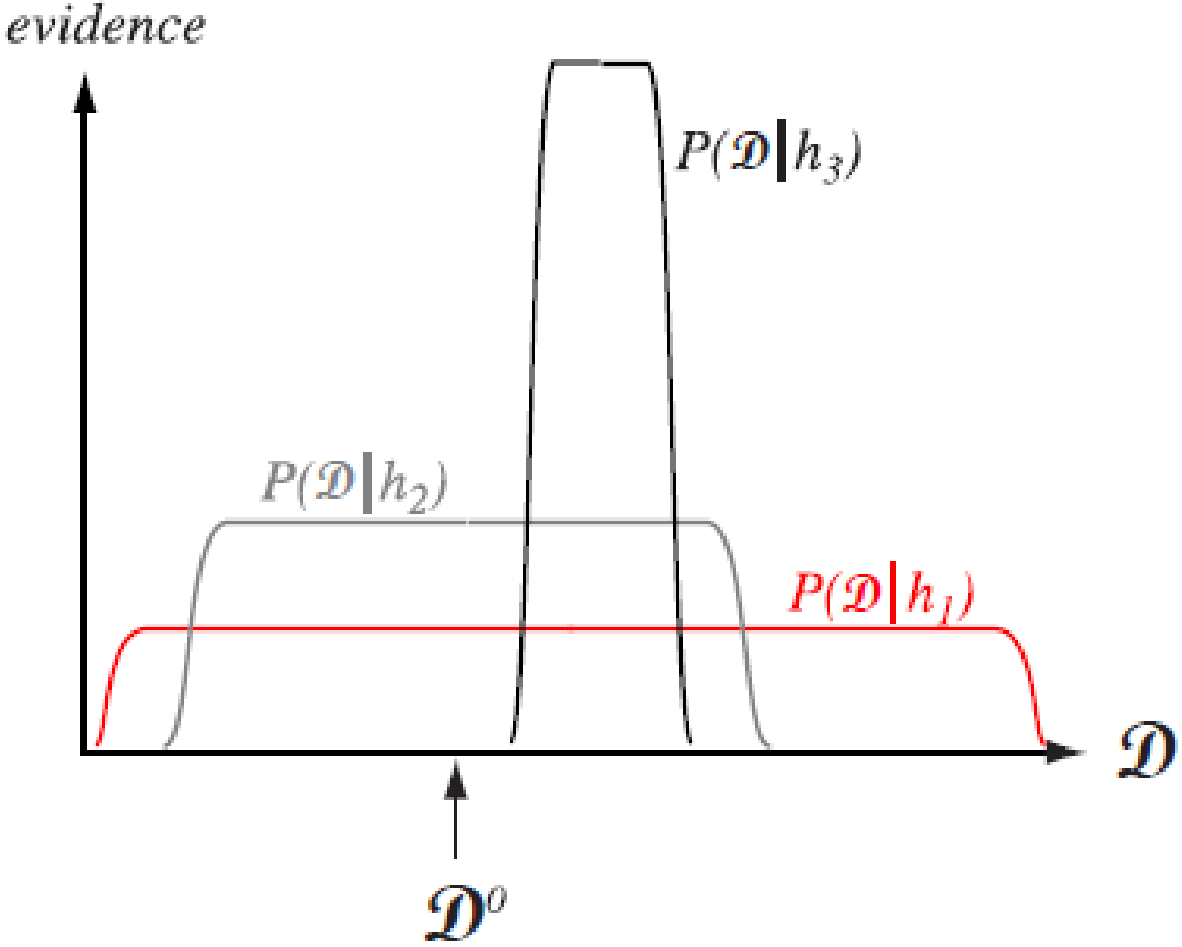
Maximum-likelihood model comparison

Recall first the maximum-likelihood parameter estimation methods discussed in Chap. **??**. Given a model with unknown parameter vector $\boldsymbol{\theta}$, we find the value $\hat{\boldsymbol{\theta}}$ which maximizes the probability of the training data, i.e., $p(\mathcal{D}|\hat{\boldsymbol{\theta}})$. Maximum-likelihood *model comparison* or maximum-likelihood *model selection* — sometimes called ML-II — is a direct generalization of those techniques. The goal here is to choose the *model* that best explains the training data, in a way that will become clear below.

We again let $h_i \in \mathcal{H}$ represent a candidate hypothesis or model (assumed discrete for simplicity), and $\mathcal{D}$ the training data. The posterior probability of any given model is given by Bayes' rule:

$$P(h_i|\mathcal{D}) = \frac{P(\mathcal{D}|h_i)P(h_i)}{p(\mathcal{D})} \propto P(\mathcal{D}|h_i)P(h_i), \tag{40}$$

where we will rarely need the normalizing factor $p(\mathcal{D})$. The data-dependent term, $P(\mathcal{D}|h_i)$, is the *evidence* for h_i; the second term, $P(h_i)$, is our subjective prior over the space of hypotheses — it rates our confidence in different models even before the data arrive. In practice, the data-dependent term dominates in Eq. 40, and hence the priors $P(h_i)$ are often neglected in the computation. In maximum-likelihood model comparison, we find the maximum likelihood parameters for each of the candidate models, calculate the resulting likelihoods, and select the model with the largest such likelihood in Eq. 40 (Fig. 9.12).


evidence
$P(\mathcal{D}|h_3)$
$P(\mathcal{D}|h_2)$
$P(\mathcal{D}|h_1)$
$\mathcal{D}$
$\mathcal{D}^0$

The evidence for h_i, i.e., $P(\mathcal{D}|h_i)$, was ignored in a maximum-likelihood setting of parameters $\hat{\boldsymbol{\theta}}$; nevertheless it is the central term in our comparison of models. As mentioned, in practice the evidence term in Eq. 40 dominates the prior term, and it is traditional to ignore such priors, which are often highly subjective or problematic anyway (Problem 38, Computer exercise 7). This procedure represents an inherent bias towards simple models (small $\Delta\boldsymbol{\theta}$); models that are overly complex (large $\Delta\boldsymbol{\theta}$) are automatically self-penalizing where "overly complex" is a data-dependent concept.

In the general case, the full integral of Eq. 41 is too difficult to calculate analytically or even numerically. Nevertheless, if $\boldsymbol{\theta}$ is k-dimensional and the posterior can be assumed to be a Gaussian, then the Occam factor can be calculated directly (Problem 37), yielding:

$$P(\mathcal{D}|h_i) \simeq \underbrace{P(\mathcal{D}|\hat{\boldsymbol{\theta}}, h_i)}_{\substack{\text{best fit} \\ \text{likelihood}}} \underbrace{p(\hat{\boldsymbol{\theta}}|h_i)(2\pi)^{k/2}|\mathbf{H}|^{-1/2}}_{\text{Occam factor}}. \tag{44}$$

where

$$\mathbf{H} = \frac{\partial^2 \ln p(\boldsymbol{\theta}|\mathcal{D}, h_i)}{\partial \boldsymbol{\theta}^2} \tag{45}$$

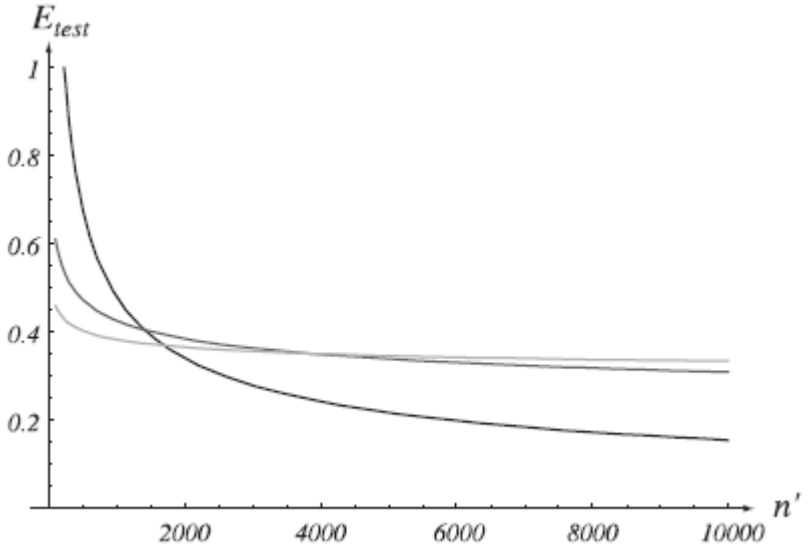
Predicting final performance from learning curves

Training on very large data sets can be computationally intensive, requiring days, weeks or even months on powerful machines. If we are exploring and comparing several different classification techniques, the total training time needed may be unacceptably long. What we seek, then, is a method to compare classifiers without the need of training all of them fully on the complete data set. If we can determine the most promising model quickly and efficiently, we need then only train this model fully.

One method is to use a classifier's performance on a relatively *small* training set to predict its performance on the ultimate large training set. Such performance is revealed in a type of learning curve in which the test error is plotted versus the size of the training set. Figure 9.15 shows the error rate on an independent test set after the classifier has been fully trained on $n' \leq n$ points in the training set. (Note that in this form of learning curve the training error decreases monotonically and does not show "overtraining" evident in curves such as Fig. 9.9.)

For many real-world problems, such learning curves decay monotonically and can be adequately described by a power-law function of the form

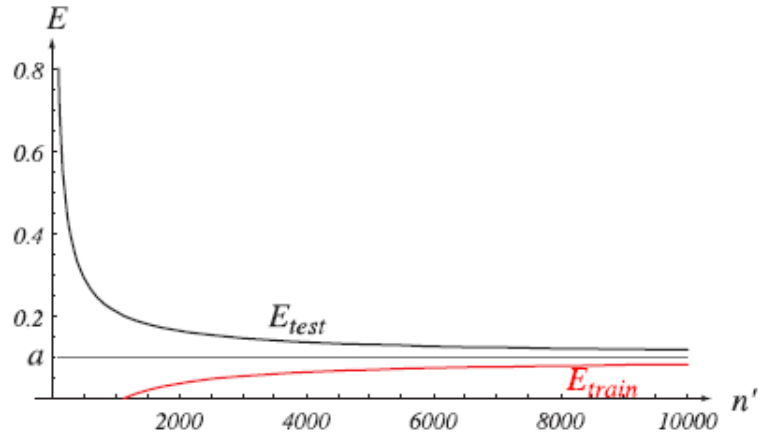
$$E_{test} = a + b/n'^{\alpha} \tag{49}$$



where a, b and $\alpha \geq 1$ depend upon the task and the classifier. In the limit of very large n', the training error equals the test error, since both the training and test sets represent the full problem space. Thus we also model the training error as a power-law function, having the same asymptotic error,

$$E_{train} = a - c/n'^{\beta}. \tag{50}$$

If the classifier is sufficiently powerful, this asymptotic error, a, is equal to the Bayes error. Furthermore, such a powerful classifier can learn perfectly the small training sets and thus the training error (measured on the n' points) will vanish at small n', as shown in Fig. 9.16.



Now we seek to estimate the asymptotic error, a, from the training and test errors on small and intermediate size training sets. From Eqs. 49 & 50 we find:

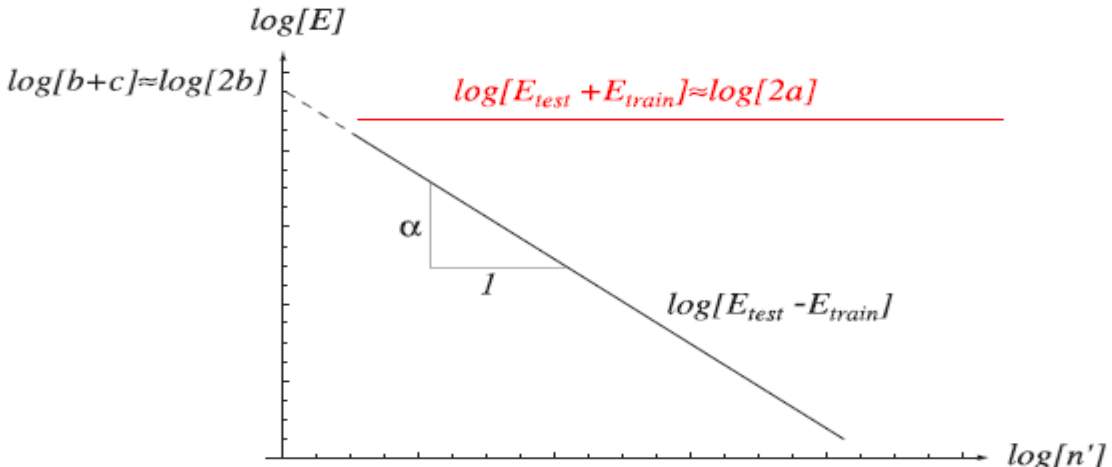
$$E_{test} + E_{train} = 2a + \frac{b}{n'^{\alpha}} - \frac{c}{n'^{\beta}} \tag{51}$$

$$E_{test} - E_{train} = \frac{b}{n^{\alpha}} + \frac{c}{n^{\beta}}.$$

If we make the assumption of $\alpha = \beta$ and $b = c$, then Eq. 51 reduces to

$$\begin{aligned} E_{test} + E_{train} &= 2a \\ E_{test} - E_{train} &= \frac{2b}{n'^{\alpha}}. \end{aligned} \tag{52}$$

Given this assumption, it is a simple matter to measure the training and test errors for small and intermediate values of n', plot them on a log-log scale and estimate a, as shown in Fig. 9.17. Even if the approximations $\alpha = \beta$ and $b = c$ do not hold in practice, the difference $E_{test} - E_{train}$ nevertheless still forms a straight line on a log-log plot and the sum, $s = b + c$, can be found from the height of the $log[E_{test} + E_{train}]$ curve. The weighted sum $cE_{test} + bE_{train}$ will be a straight line for some empirically set values of b and c, constrained to obey $b + c = s$, enabling a to be estimated (Problem 41). Once a has been estimated for each in the set of candidate classifiers, the one with the lowest a is chosen and must be trained on the full training set $\mathcal{D}$.



Thanks