Chapter 1
Ensemble Learning

Robi Polikar

1.1 Introduction

Over the last couple of decades, multiple classifier systems, also called ensemble systems have enjoyed growing attention within the computational intelligence and machine learning community. This attention has been well deserved, as ensemble systems have proven themselves to be very effective and extremely versatile in a broad spectrum of problem domains and real-world applications. Originally developed to reduce the variance—thereby improving the accuracy—of an automated decision-making system, ensemble systems have since been successfully used to address a variety of machine learning problems, such as feature selection, confidence estimation, missing feature, incremental learning, error correction, class-imbalanced data, learning concept drift from nonstationary distributions, among others. This chapter provides an overview of ensemble systems, their properties, and how they can be applied to such a wide spectrum of applications.

Truth be told, machine learning and computational intelligence researchers have been rather late in discovering the ensemble-based systems, and the benefits offered by such systems in decision making. While there is now a significant body of knowledge and literature on ensemble systems as a result of a couple of decades of intensive research, ensemble-based decision making has in fact been around and part of our daily lives perhaps as long as the civilized communities existed. You see, ensemble-based decision making is nothing new to us; as humans, we use such systems in our daily lives so often that it is perhaps second nature to us. Examples are many: the essence of democracy where a group of people vote to make a decision, whether to choose an elected official or to decide on a new law, is in fact based on ensemble-based decision making. The judicial system in many countries, whether based on a jury of peers or a panel of judges, is also based on

R. Polikar (✉)
Rowan University, Glassboro, NJ 08028, USA
e-mail: polikar@rowan.edu

C. Zhang and Y. Ma (eds.), *Ensemble Machine Learning: Methods and Applications*,
DOI 10.1007/978-1-4419-9326-7_1, © Springer Science+Business Media, LLC 2012

ensemble-based decision making. Perhaps more practically, whenever we are faced with making a decision that has some important consequence, we often seek the opinions of different "experts" to help us make that decision; consulting with several doctors before agreeing to a major medical operation, reading user reviews before purchasing an item, calling references before hiring a potential job applicant, even peer review of this article prior to publication, are all examples of ensemble-based decision making. In the context of this discussion, we will loosely use the terms expert, classifier, hypothesis, and decision interchangeably.

While the original goal for using ensemble systems is in fact similar to the reason we use such mechanisms in our daily lives—that is, to improve our confidence that we are making the right decision, by weighing various opinions, and combining them through some thought process to reach a final decision—there are many other machine-learning specific applications of ensemble systems. These include confidence estimation, feature selection, addressing missing features, incremental learning from sequential data, data fusion of heterogeneous data types, learning nonstationary environments, and addressing imbalanced data problems, among others.

In this chapter, we first provide a background on ensemble systems, including statistical and computational reasons for using them. Next, we discuss the three pillars of the ensemble systems: diversity, training ensemble members, and combining ensemble members. After an overview of commonly used ensemble-based algorithms, we then look at various aforementioned applications of ensemble systems as we try to answer the question "what else can ensemble systems do for you?"

1.1.1 Statistical and Computational Justifications for Ensemble Systems

The premise of using ensemble-based decision systems in our daily lives is fundamentally not different from their use in computational intelligence. We consult with others before making a decision often because of the variability in the past record and accuracy of any of the individual decision makers. If in fact there were such an expert, or perhaps an oracle, whose predictions were always true, we would never need any other decision maker, and there would never be a need for ensemble-based systems. Alas, no such oracle exists; every decision maker has an imperfect past record. In other words, the accuracy of each decision maker's decision has a nonzero variability. Now, note that any classification error is composed of two components that we can control: bias, the accuracy of the classifier; and variance, the precision of the classifier when trained on different training sets. Often, these two components have a trade-off relationship: classifiers with low bias tend to have high variance and vice versa. On the other hand, we also know that averaging has a smoothing (variance-reducing) effect. Hence, the goal of ensemble systems is to create several classifiers with relatively fixed (or similar) bias and then combining their outputs, say by averaging, to reduce the variance.

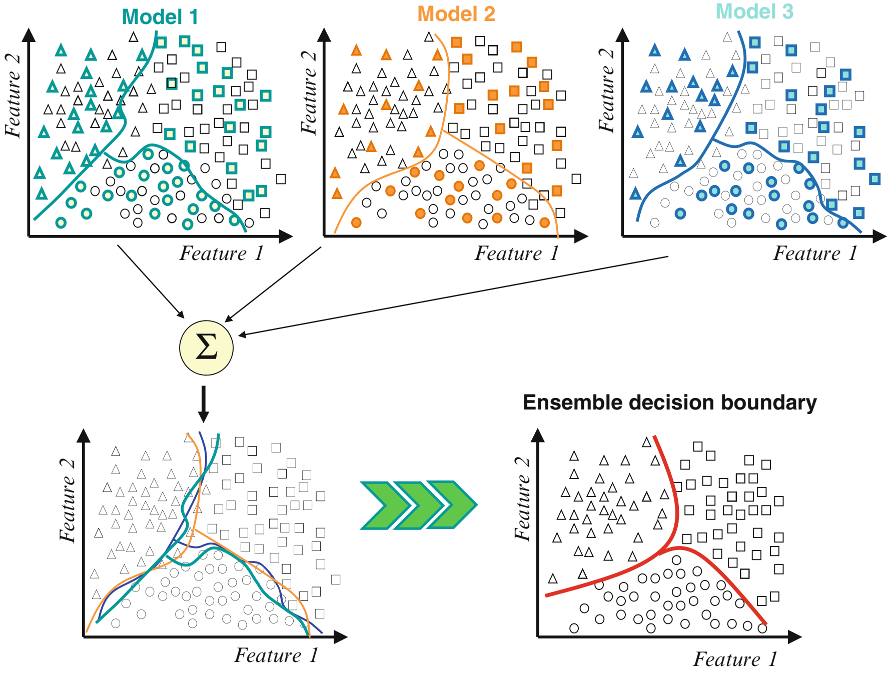


Fig. 1.1 Variability reduction using ensemble systems

The reduction of variability can be thought of as reducing high-frequency (high-variance) noise using a moving average filter, where each sample of the signal is averaged by a neighbor of samples around it. Assuming that noise in each sample is independent, the noise component is averaged out, whereas the information content that is common to all segments of the signal is unaffected by the averaging operation. Increasing classifier accuracy using an ensemble of classifiers works exactly the same way: assuming that classifiers make different errors on each sample, but generally agree on their correct classifications, averaging the classifier outputs reduces the error by averaging out the error components.

It is important to point out two issues here: first, in the context of ensemble systems, there are many ways of combining ensemble members, of which averaging the classifier outputs is only one method. We discuss different combination schemes later in this chapter. Second, combining the classifier outputs does not necessarily lead to a classification performance that is guaranteed to be better than the best classifier in the ensemble. Rather, it reduces our likelihood of choosing a classifier with a poor performance. After all, if we knew a priori which classifier would perform the best, we would only use that classifier and would not need to use an ensemble. A representative illustration of the variance reduction ability of the ensemble of classifiers is shown in Fig. 1.1.

1.1.2 Development of Ensemble Systems

Many reviews refer to Dasarathy and Sheela's 1979 work as one of the earliest example of ensemble systems [1], with their ideas on partitioning the feature space using multiple classifiers. About a decade later, Hansen and Salamon showed that an ensemble of similarly configured neural networks can be used to improve classification performance [2]. However, it was Schapire's work that demonstrated through a procedure he named *boosting* that a strong classifier with an arbitrarily low error on a binary classification problem, can be constructed from an ensemble of classifiers, the error of any of which is merely better than that of random guessing [3]. The theory of boosting provided the foundation for the subsequent suite of *AdaBoost* algorithms, arguably the most popular ensemble-based algorithms, extending the boosting concept to multiple class and regression problems [4]. We briefly describe the boosting algorithms below, but a more detailed coverage of these algorithms can be found in Chap. 2 of this book, and Kuncheva's text [5].

In part due to success of these seminal works, and in part based on independent efforts, research in ensemble systems have since exploded, with different flavors of ensemble-based algorithms appearing under different names: bagging [6], random forests (an ensemble of decision trees), composite classifier systems [1], mixture of experts (MoE) [7, 8], stacked generalization [9], consensus aggregation [10], combination of multiple classifiers [11–15], dynamic classifier selection [15], classifier fusion [16–18], committee of neural networks [19], classifier ensembles [19, 20], among many others. These algorithms, and in general all ensemble-based systems, typically differ from each other based on the selection of training data for individual classifiers, the specific procedure used for generating ensemble members, and/or the combination rule for obtaining the ensemble decision. As we will see, these are the three pillars of any ensemble system.

In most cases, ensemble members are used in one of two general settings: classifier selection and classifier fusion [5, 15, 21]. In *classifier selection*, each classifier is trained as a local expert in some local neighborhood of the entire feature space. Given a new instance, the classifier trained with data closest to the vicinity of this instance, in some distance metric sense, is then chosen to make the final decision, or given the highest weight in contributing to the final decision [7, 15, 22, 23]. In *classifier fusion* all classifiers are trained over the entire feature space, and then combined to obtain a composite classifier with lower variance (and hence lower error). Bagging [6], random forests [24], arc-x4 [25], and boosting/AdaBoost [3, 4] are examples of this approach. Combining the individual classifiers can be based on the labels only, or based on class-specific continuous valued outputs [18, 26, 27], for which classifier outputs are first normalized to the [0, 1] interval to be interpreted as the support given by the classifier to each class [18, 28]. Such interpretation leads to algebraic combination rules (simple or weighted majority voting, maximum/minimum/sum/product, or other combinations class-specific outputs) [12, 27, 29], the Dempster–Shafer-based classifier fusion [13, 30], or decision templates [18, 21, 26, 31]. Many of these combination rules are discussed below in more detail.

A sample of the immense literature on classifier combination can be found in Kuncheva's book [5] (and references therein), an excellent text devoted to theory and implementation of ensemble-based classifiers.

1.2 Building an Ensemble System

Three strategies need to be chosen for building an effective ensemble system. We have previously referred to these as the three pillars of ensemble systems: (1) data sampling/selection; (2) training member classifiers; and (3) combining classifiers.

1.2.1 Data Sampling and Selection: Diversity

Making different errors on any given sample is of paramount importance in ensemble-based systems. After all, if all ensemble members provide the same output, there is nothing to be gained from their combination. Therefore, we need diversity in the decisions of ensemble members, particularly when they are making an error. The importance of diversity for ensemble systems is well established [32, 33]. Ideally, classifier outputs should be independent or preferably negatively correlated [34, 35].

Diversity in ensembles can be achieved through several strategies, although using different subsets of the training data is the most common approach, also illustrated in Fig. 1.1. Different sampling strategies lead to different ensemble algorithms. For example, using bootstrapped replicas of the training data leads to bagging, whereas sampling from a distribution that favors previously misclassified samples is the core of boosting algorithms. On the other hand, one can also use different subsets of the available features to train each classifier, which leads to *random subspace methods* [36]. Other less common approaches also include using different parameters of the base classifier (such as training an ensemble of multilayer perceptrons, each with a different number of hidden layer nodes), or even using different base classifiers as the ensemble members. Definitions of different types of diversity measures can be found in [5, 37, 38]. We should also note that while the importance of diversity, and lack of diversity leading to inferior ensemble performance has been wellestablished, an explicit relationship between diversity and ensemble accuracy has not been identified [38, 39].

1.2.2 Training Member Classifiers

At the core of any ensemble-based system is the strategy used to train individual ensemble members. Numerous competing algorithms have been developed for training ensemble classifiers; however, bagging (and related algorithms arc-x4 and random forests), boosting (and its many variations), stack generalization and hierarchical MoE remain as the most commonly employed approaches. These approaches are discussed in more detail below, in Sect. 1.3.

1.2.3 Combining Ensemble Members

The last step in any ensemble-based system is the mechanism used to combine the individual classifiers. The strategy used in this step depends, in part, on the type of classifiers used as ensemble members. For example, some classifiers, such as support vector machines, provide only discrete-valued label outputs. The most commonly used combination rules for such classifiers is (simple or weighted) majority voting followed at a distant second by the Borda count. Other classifiers, such as multilayer perceptron or (naïve) Bayes classifier, provide continuous valued class-specific outputs, which are interpreted as the support given by the classifier to each class. A wider array of options is available for such classifiers, such as arithmetic (sum, product, mean, etc.) combiners or more sophisticated decision templates, in addition to voting-based approaches. Many of these combiners can be used immediately after the training is complete, whereas more complex combination algorithms may require an additional training step (as used in stacked generalization or hierarchical MoE). We now briefly discuss some of these approaches.

1.2.3.1 Combining Class Labels

Let us first assume that only the class labels are available from the classifier outputs, and define the decision of the t^{th} classifier as $d_{t,c} \in \{0,1\}, t = 1, \ldots, T$ and $c = 1, \ldots, C$, where T is the number of classifiers and C is the number of classes. If t^{th} classifier (or hypothesis) h_t chooses class ω_c, then $d_{t,c} = 1$, and 0, otherwise. Note that the continuous valued outputs can easily be converted to label outputs (by assigning $d_{t,c} = 1$ for the class with the highest output), but not vice versa. Therefore, the combination rules described in this section can also be used by classifiers providing specific class supports.

Majority Voting

Majority voting has three flavors, depending on whether the ensemble decision is the class (1) on which all classifiers agree (*unanimous voting*); (2) predicted by at least one more than half the number of classifiers (*simple majority*); or (3) that receives the highest number of votes, whether or not the sum of those votes exceeds 50% (*plurality voting*). When not specified otherwise, majority voting usually refers to plurality voting, which can be mathematically defined as follows: choose class ω_{c^*}, if

$$\sum_{t=1}^{T} d_{t,c^*} = \max_c \sum_{t=1}^{T} d_{t,c} \tag{1.1}$$

If the classifier outputs are independent, then it can be shown that majority voting is the optimal combination rule. To see this, consider an odd number of T classifiers, with each classifier having a probability of correct classification p. Then, the ensemble makes the correct decision if at least $\lfloor T/2 \rfloor + 1$ of these classifiers choose the correct label. Here, the floor function $\lfloor \blacksquare \rfloor$ returns the largest integer less than or equal to its argument. The accuracy of the ensemble is governed by the binomial distribution; the probability of having $k \geq T/2 + 1$ out of T classifiers returning the correct class. Since each classifier has a success rate of p, the probability of ensemble success is then

$$p_{\text{ens}} = \sum_{k=\frac{T}{2}+1}^{T} \binom{T}{k} p^k (1-p)^{T-k} \tag{1.2}$$

Note that P_{ens} approaches 1 as $T \rightarrow \infty$, if $p > 0.5$; and it approaches 0 if $p < 0.5$. This result is also known as the Condorcet Jury theorem (1786), as it formalizes the probability of a plurality-based jury decision to be the correct one. Equation (1.2) makes a powerful statement: if the probability of a member classifier giving the correct answer is higher than $1/2$, which really is the least we can expect from a classifier on a binary class problem, then the probability of success approaches 1 very quickly. If we have a multiclass problem, the same concept holds as long as each classifier has a probability of success better than random guessing (i.e., $p > 1/4$ for a four class problem). An extensive and excellent analysis of the majority voting approach can be found in [5].

Weighted Majority Voting

If we have reason to believe that some of the classifiers are more likely to be correct than others, weighting the decisions of those classifiers more heavily can further improve the overall performance compared to that of plurality voting. Let us assume that we have a mechanism for predicting the (future) approximate generalization performance of each classifier. We can then assign a weight W_t to classifier h_t in proportion of its estimated generalization performance. The ensemble, combined according to weighted majority voting then chooses class c^*, if

$$\textstyle\sum_{t=1}^{T} w_t d_{t,c^*} = \max_c \sum_{t=1}^{T} w_t d_{t,c} \tag{1.3}$$

that is, if the total weighted vote received by class ω_{c^*} is higher than the total vote received by any other class. In general, voting weights are normalized such that they add up to 1.

So, how do we assign the weights? If we knew, a priori, which classifiers would work better, we would only use those classifiers. In the absence of such information, a plausible and commonly used strategy is to use the performance of a classifier on a separate validation (or even training) dataset, as an estimate of that classifier's generalization performance. As we will see in the later sections, AdaBoost follows such an approach. A detailed discussion on weighted majority voting can also be found in [40].

Borda Count

Voting approaches typically use a winner-take-all strategy, i.e., only the class that is chosen by each classifier receives a vote, ignoring any support that nonwinning classes may receive. Borda count uses a different approach, feasible if we can rank order the classifier outputs, that is, if we know the class with the most support (the winning class), as well as the class with the second most support, etc. Of course, if the classifiers provide continuous outputs, the classes can easily be rank ordered with respect to the support they receive from the classifier.

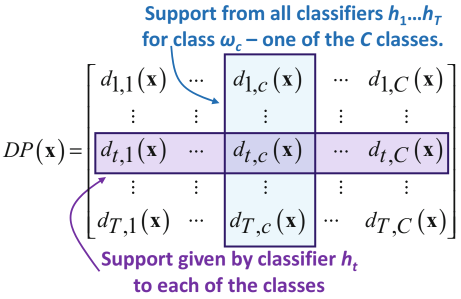
In Borda count, devised in 1770 by Jean Charles de Borda, each classifier (decision maker) rank orders the classes. If there are C candidates, the winning class receives C-1 votes, the class with the second highest support receives C-2 votes, and the class with the i^{th} highest support receives C-i votes. The class with the lowest support receives no votes. The votes are then added up, and the class with the most votes is chosen as the ensemble decision.

1.2.3.2 Combining Continuous Outputs

If a classifier provides continuous output for each class (such as multilayer perceptron or radial basis function networks, naïve Bayes, relevance vector machines, etc.), such outputs—upon proper normalization (such as softmax normalization in (1.4) [41])—can be interpreted as the degree of support given to that class, and under certain conditions can also be interpreted as an estimate of the posterior probability for that class. Representing the actual classifier output corresponding to class ω_c for instance $\boldsymbol{x}$ as $g_c(\boldsymbol{x})$, and the normalized values as $\tilde{g}_c(\boldsymbol{x})$, approximated posterior probabilities $P(\omega_c|\boldsymbol{x})$ can be obtained as

$$P(\omega_c|\boldsymbol{x}) \approx \tilde{g}_c(\boldsymbol{x}) = \frac{\mathrm{e}^{g_c(x)}}{\sum_{i=1}^{C} \mathrm{e}^{g_i(x)}} \Rightarrow \sum_{i=1}^{C} \tilde{g}_i(\boldsymbol{x}) = 1 \qquad (1.4)$$

Fig. 1.2 Decision profile for a given instance ***x***


Support from all classifiers $h_1 \ldots h_T$ for class ω_c – one of the C classes.
$DP(\mathbf{x}) = \begin{bmatrix} d_{1,1}(\mathbf{x}) & \cdots & d_{1,c}(\mathbf{x}) & \cdots & d_{1,C}(\mathbf{x}) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ d_{t,1}(\mathbf{x}) & \cdots & d_{t,c}(\mathbf{x}) & \cdots & d_{t,C}(\mathbf{x}) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ d_{T,1}(\mathbf{x}) & \cdots & d_{T,c}(\mathbf{x}) & \cdots & d_{T,C}(\mathbf{x}) \end{bmatrix}$
Support given by classifier h_t to each of the classes

In order to consolidate different combination rules, we use Kuncheva's *decision profile* matrix $DP(\boldsymbol{x})$ [18], whose elements $d_{t,c} \in [0, 1]$ represent the support given by the t^{th} classifier to class ω_c. Specifically, as illustrated in Fig. 1.2, the rows of $DP(\boldsymbol{x})$ represent the support given by individual classifiers to each of the classes, whereas the columns represent the support received by a particular class c from all classifiers.

Algebraic Combiners

In algebraic combiners, the total support for each class is obtained as a simple algebraic function of the supports received by individual classifiers. Following the notation used in [18], let us represent the total support received by class ω_c, the c^{th} column of the decision profile $DP(\boldsymbol{x})$, as

$$\mu_c(\boldsymbol{x}) = F\left[d_{1,c}(\boldsymbol{x}), \ldots, d_{T,C}(\boldsymbol{x})\right] \tag{1.5}$$

where $F[\blacksquare]$ is one of the following combination functions.

Mean Rule: The support for class ω_c is the average of all classifiers' c^{th} outputs,

$$\mu_c(\boldsymbol{x}) = \frac{1}{T}\sum\nolimits_{t=1}^{T} d_{t,c}(\boldsymbol{x}) \tag{1.6}$$

hence the function $F[\cdot]$ is the averaging function. Note that the mean rule results in the identical final classification as the sum rule, which only differs from the mean rule by the $1/T$ normalization factor. In either case, the final decision is the class ω_c for which the total support $\mu_c(\boldsymbol{x})$ is the highest.

Weighted Average: The weighted average rule combines the mean and the weighted majority voting rules, where the weights are applied not to class labels, but to the actual continuous outputs. The weights can be obtained during the ensemble generation as part of the regular training, as in AdaBoost, or a separate training can be used to obtain the weights, such as in a MoE. Usually, each classifier h_t receives a weight, although it is also possible to assign a weight to each class output of each classifier. In the former case, we have T weights, $w_1, \ldots, w_T$, usually obtained as *estimated* generalization performances based on training data, with the total support for class ω_c as

$$\mu_c(\boldsymbol{x}) = \frac{1}{T}\sum\nolimits_{t=1}^{T} w_t d_{t,c}(\boldsymbol{x}) \tag{1.7}$$

In the latter case, there are $T * C$ class and classifier-specific weights, which leads to a class-conscious combination of classifier outputs [18]. Total support for class ω_c is then

$$\mu_c(\boldsymbol{x}) = \frac{1}{T}\sum\nolimits_{t=1}^{T} w_{t,c} d_{t,c}(\boldsymbol{x}) \tag{1.8}$$

where $w_{t,c}$ is the weight of the t^{th} classifier for classifying class ω_c instances.

Trimmed mean: Sometimes classifiers may erroneously give unusually low or high support to a particular class such that the correct decisions of other classifiers are not enough to undo the damage done by this unusual vote. This problem can be avoided by discarding the decisions of those classifiers with the highest and lowest support before calculating the mean. This is called trimmed mean. For a $R\%$ trimmed mean, $R\%$ of the support from each end is removed, with the mean calculated on the remaining supports, avoiding the extreme values of support. Note that 50% trimmed mean is equivalent to the median rule discussed below.

Minimum/Maximum/Median Rule: These functions simply take the minimum, maximum, or the median among the classifiers' individual outputs.

$$\begin{aligned}\mu_c(\boldsymbol{x}) &= \min\nolimits_{t=1,\ldots,T}\{d_{t,c}(\boldsymbol{x})\}\\ \mu_c(\boldsymbol{x}) &= \max\nolimits_{t=1,\ldots,T}\{d_{t,c}(\boldsymbol{x})\}\\ \mu_c(\boldsymbol{x}) &= \text{median}_{t=1,\ldots,T}\{d_{t,c}(\boldsymbol{x})\}\end{aligned} \tag{1.9}$$

where the ensemble decision is chosen as the class for which total support is largest. Note that the *minimum rule* chooses the class for which the *minimum support* among the classifiers is highest.

Product Rule: The product rule chooses the class whose product of supports from each classifier is the highest. Due to the nulling nature of multiplying with zero, this rule decimates any class that receives at least one zero (or very small) support.

$$\mu_c(\boldsymbol{x}) = \frac{1}{T}\prod\nolimits_{t=1}^{T} d_{t,c}(\boldsymbol{x}) \tag{1.10}$$

Generalized Mean: All of the aforementioned rules are in fact special cases of the generalized mean,

$$\mu_c(\boldsymbol{x}) = \left(\frac{1}{T}\sum\nolimits_{t=1}^{T} (d_{t,c}(\boldsymbol{x}))^{\alpha}\right)^{1/\alpha} \tag{1.11}$$

where different choices of α lead to different combination rules. For example, $\alpha \rightarrow$ -∞, leads to minimum rule, and $\alpha \rightarrow 0$, leads to

$$\mu_c(x) = \left(\prod\nolimits_{t=1}^{T} (d_{t,c}(x))\right)^{1/T} \tag{1.12}$$

which is the geometric mean, a modified version of the product rule. For $\alpha \rightarrow 1$, we get the mean rule, and $\alpha \rightarrow \infty$ leads to the maximum rule.

Decision Template: Consider computing the average decision profile observed for each class throughout training. Kuncheva defines this average decision profile as the *decision template* of that class [18]. We can then compare the decision profile of a given instance to the decision templates (i.e., average decision profiles) of each class, choosing the class whose decision template is closest to the decision profile of the current instance, in some similarity measure. The decision template for class ω_c is then computed as

$$DT_c = 1/N_c \sum\nolimits_{X_c \in \omega_c} DP\ (X_c) \tag{1.13}$$

as the average decision profile obtained from $\boldsymbol{X}_c$, the set of training instances (of cardinality N_c) whose true class is ω_c. Given an unlabeled test instance $\boldsymbol{x}$, we first construct its decision profile $DP(\boldsymbol{x})$ from the ensemble outputs and calculate the similarity S between $DP(\boldsymbol{x})$ and the decision template DT_c for each class ω_c as the degree of support given to class ω_c.

$$\mu_c(\boldsymbol{x}) = S(DP(x), DT_c), c = 1, \ldots, C \tag{1.14}$$

where the similarity measure S is usually a squared Euclidean distance,

$$\mu_c(\boldsymbol{x}) = 1 - \frac{1}{T \times C} \sum\nolimits_{t=1}^{T} \sum\nolimits_{i=1}^{C} (DT_c(t,i) - d_{t,i}(\boldsymbol{x}))^2 \tag{1.15}$$

and where $DT_c(t,i)$ is the decision template support given by the t^{th} classifier to class ω_i, i.e., the support given by the t^{th} classifier to class ω_i, averaged over all class ω_c training instances. We expect this support to be high when $i = c$, and low otherwise. The second term $d_{t,i}(x)$ is the support given by the t^{th} classifier to class ω_i for the given instance $\boldsymbol{x}$. The class with the highest total support is then chosen as the ensemble decision.

1.3 Popular Ensemble-Based Algorithms

A rich collection of ensemble-based classifiers have been developed over the last several years. However, many of these are some variation of the select few well-established algorithms whose capabilities have also been extensively tested and widely reported. In this section, we present an overview of some of the most prominent ensemble algorithms.

Algorithm 1 Bagging

```
Inputs: Training data $S$; supervised learning algorithm, BaseClassifier, integer $T$
specifying ensemble size; percent $R$ to create bootstrapped training data.
Do $t = 1, \dots, T$
    1. Take a bootstrapped replica $S_t$ by randomly drawing $R\%$ of $S$.
    2. Call BaseClassifier with $S_t$ and receive the hypothesis (classifier) $h_t$.
    3. Add $h_t$ to the ensemble, $\mathcal{E} \leftarrow \mathcal{E} \cup \quad h_t$.
End
Ensemble Combination: Simple Majority Voting—Given unlabeled instance $\boldsymbol{x}$
    1. Evaluate the ensemble $\mathcal{E} = \{h_1, \dots, h_T\}$ on $\boldsymbol{x}$.
    2. Let $v_{t,c} = 1$ if $h_t$ chooses class $\omega_c$, and 0, otherwise.
    3. Obtain total vote received by each class
           $V_c = \sum_{t=1}^{T} v_{t,c}, \; c = 1, ..., C$        (1.16)
Output: Class with the highest $V_c$.
```

1.3.1 Bagging

Breiman's bagging (short for Bootstrap Aggregation) algorithm is one of the earliest and simplest, yet effective, ensemble-based algorithms. Given a training dataset S of cardinality N, bagging simply trains T independent classifiers, each trained by sampling, with replacement, N instances (or some percentage of N) from S. The diversity in the ensemble is ensured by the variations within the bootstrapped replicas on which each classifier is trained, as well as by using a relatively *weak classifier* whose decision boundaries measurably vary with respect to relatively small perturbations in the training data. Linear classifiers, such as decision stumps, linear SVM, and single layer perceptrons are good candidates for this purpose. The classifiers so trained are then combined via simple majority voting. The pseudocode for bagging is provided in Algorithm 1.

Bagging is best suited for problems with relatively small available training datasets. A variation of bagging, called *Pasting Small Votes* [42], designed for problems with large training datasets, follows a similar approach, but partitioning the large dataset into smaller segments. Individual classifiers are trained with these segments, called *bites*, before combining them via majority voting.

Another creative version of bagging is the *Random Forest* algorithm, essentially an ensemble of decision trees trained with a bagging mechanism [24]. In addition to choosing instances, however, a random forest can also incorporate random subset selection of features as described in Ho's random subspace models [36].

1.3.2 Boosting and AdaBoost

Boosting, introduced in Schapire's seminal work *strength of weak learning* [3], is an iterative approach for generating a strong classifier, one that is capable of achieving arbitrarily low training error, from an ensemble of weak classifiers, each of which can barely do better than random guessing. While boosting also combines an ensemble of weak classifiers using simple majority voting, it differs from bagging in one crucial way. In bagging, instances selected to train individual classifiers are bootstrapped replicas of the training data, which means that each instance has equal chance of being in each training dataset. In boosting, however, the training dataset for each subsequent classifier increasingly focuses on instances misclassified by previously generated classifiers.

Boosting, designed for binary class problems, creates sets of three weak classifiers at a time: the first classifier (or hypothesis) h_1 is trained on a random subset of the available training data, similar to bagging. The second classifier, h_2, is trained on a different subset of the original dataset, precisely half of which is correctly identified by h_1, and the other half is misclassified. Such a training subset is said to be the "most informative," given the decision of h_1. The third classifier h_3 is then trained with instances on which h_1 and h_2 disagree. These three classifiers are then combined through a three-way majority vote. Schapire proved that the training error of this three-classifier ensemble is bounded above by $g(\varepsilon) < 3\varepsilon^2 \quad - 2\varepsilon^3$, where ε is the error of any of the three classifiers, provided that each classifier has an error rate ε< 0.5, the least we can expect from a classifier on a binary classification problem.

AdaBoost (short for *Adaptive Boosting*) [4], and its several variations later extended the original boosting algorithm to multiple classes (AdaBoost.M1, AdaBost.M2), as well as to regression problems (AdaBoost.R). Here we describe the AdaBoost.M1, the most popular version of the AdaBoost algorithms.

AdaBoost has two fundamental differences from boosting: (1) instances are drawn into the subsequent datasets from an iteratively updated *sample distribution* of the training data; and (2) the classifiers are combined through weighted majority voting, where *voting weights* are based on classifiers' training errors, which themselves are weighted according to the sample distribution. The sample distribution ensures that harder samples, i.e., instances misclassified by the previous classifier are more likely to be included in the training data of the next classifier.

The pseudocode of the AdaBoost.M1 is provided in Algorithm 2. The sample distribution, $D_t(i)$ essentially assigns a weight to each training instance $\boldsymbol{x}_i$, $i = 1, \ldots, N$, from which training data subsets S_t are drawn for each consecutive classifier (hypothesis) h_t. The distribution is initialized to be uniform; hence, all instances have equal probability to be drawn into the first training dataset. The training error ε_t of classifier h_t is then computed as the sum of these distribution weights of the instances misclassified by h_t ((1.17), where $[\![\blacksquare]\!]$ is 1 if its argument is true and 0 otherwise). AdaBoost.M1 requires that this error be less than $1/2$, which is then normalized to obtain β_t, such that $0 < \beta_t \ < 1$ for $0 < \varepsilon_t \ < 1/2$.

Algorithm 2 AdaBoost.M1

```
Inputs: Training data = $\{x_i, y_i\}, i = 1, \ldots, N \; y_i \in \{\omega_1, \ldots, \omega_C\}$, supervised learner BaseClassifier; ensemble size $T$.
Initialize $D_1(i) = 1/N$.
Do for $t = 1, 2, \ldots, T$:
    1. Draw training subset $S_t$ from the distribution $D_t$.
    2. Train BaseClassifier on $S_t$, receive hypothesis $h_t\colon X \to Y$
    3. Calculate the error of $h_t$:
           $\varepsilon_t = \sum_i I[\![h_t(x_i \neq y_i)]\!] D_t(x_i) \qquad (1.17)$
       If $\varepsilon_t > 1/2$ abort.
    4. Set
           $\beta_t = \varepsilon_t/(1 - \varepsilon_t) \qquad (1.18)$
    5. Update sampling distribution
           $D_{t+1}(i) = \frac{D_t(i)}{Z_t} \cdot \begin{cases} \beta_t, & \text{if } h_t(x_i) = y_i \\ 1, & \text{otherwise} \end{cases} \qquad (1.19)$
       where $Z_t = \sum_i D_t(i)$ is a normalization constant to ensure that $D_{t+1}$ is a proper distribution function.
End
Weighted Majority Voting: Given unlabeled instance $z$,
    obtain total vote received by each class
           $V_c = \sum_{t: h_t(z) = \omega_c} \log\left(\frac{1}{\beta_t}\right), \quad c = 1, \ldots, C \qquad (1.20)$
Output: Class with the highest $V_c$.
```

The heart of AdaBoost.M1 is the distribution update rule shown in (1.19): the distribution weights of the instances correctly classified by the current hypothesis h_t are reduced by a factor of β_t, whereas the weights of the misclassified instances are left unchanged. When the updated weights are renormalized by Z_t to ensure that D_{t+1} is a proper distribution, the weights of the misclassified instances are effectively increased. Hence, with each new classifier added to the ensemble, AdaBoost focuses on increasingly difficult instances. At each iteration t, (1.19) raises the weights of misclassified instances such that they add up to $1/2$, and lowers those of correctly classified ones, such that they too add up to $1/2$. Since the base model learning algorithm **BaseClassifier** is required to have an error less than $1/2$, it is guaranteed to correctly classify at least one previously misclassified training example. When it is unable to do so, AdaBoost aborts; otherwise, it continues until T classifiers are generated, which are then combined using the weighted majority voting.

Note that the reciprocals of the normalized errors of individual classifiers are used as voting weights in *weighted majority voting* in AdaBoost.M1; hence, classifiers that have shown good performance during training (low β_t) are rewarded with higher voting weights. Since the performance of a classifier on its own training data can be very close to zero, β_t can be quite large, causing numerical instabilities. Such instabilities are avoided by the use of the logarithm in the voting weights (1.20).

Much of the popularity of AdaBoost.M1 is not only due to its intuitive and extremely effective structure but also due to Freund and Schapire's elegant proof that shows the training error of AdaBoost.M1 as bounded above

$$E_{\text{ensemble}} < 2^T \prod_{t=1}^{T} \sqrt{\varepsilon_t(1-\varepsilon_t)} \tag{1.21}$$

Since $\varepsilon_t < 1/2$, E_{ensemble}, the error of the ensemble, is guaranteed to decrease as the ensemble grows. It is interesting, however, to note that AdaBoost.M1 still requires the classifiers to have a (weighted) error that is less than ½ even on nonbinary class problems. Achieving this threshold becomes increasingly difficult as the number of classes increase. Freund and Schapire recognized that there is information even in the classifiers' nonselected class outputs. For example, in handwritten character recognition problem, the characters "1" and "7" look alike, and the classifier may give a high support to both of these classes, and low support to all others. AdaBoost.M2 takes advantage of the supports given to nonchosen classes and defines a pseudo-loss, and unlike the error in AdaBoost.M1, is no longer required to be less than ½. Yet AdaBoost.M2 has a very similar upper bound for training error as AdaBoost.M1. AdaBoost.R is another variation—designed for function approximation problems—that essentially replaces classification error with regression error [4].

1.3.3 Stacked Generalization

The algorithms described so far use nontrainable combiners, where the combination weights are established once the member classifiers are trained. Such a combination rule does not allow determining which member classifier has learned which partition of the feature space. Using trainable combiners, it is possible to determine which classifiers are likely to be successful in which part of the feature space and combine them accordingly. Specifically, the ensemble members can be combined using a separate classifier, trained on the outputs of the ensemble members, which leads to the stacked generalization model.

Wolpert's stacked generalization [9], illustrated in Fig. 1.3, first creates T Tier-1 classifiers, $C_1, \ldots, C_T$, based on a cross-validation partition of the training data. To do so, the entire training dataset is divided into B blocks, and each Tier-1 classifier is first trained on (a different set of) $B-1$ blocks of the training data. Each classifier is then evaluated on the B^{th} (pseudo-test) block, not seen during training. The outputs of these classifiers on their pseudo-training blocks constitute the training data for

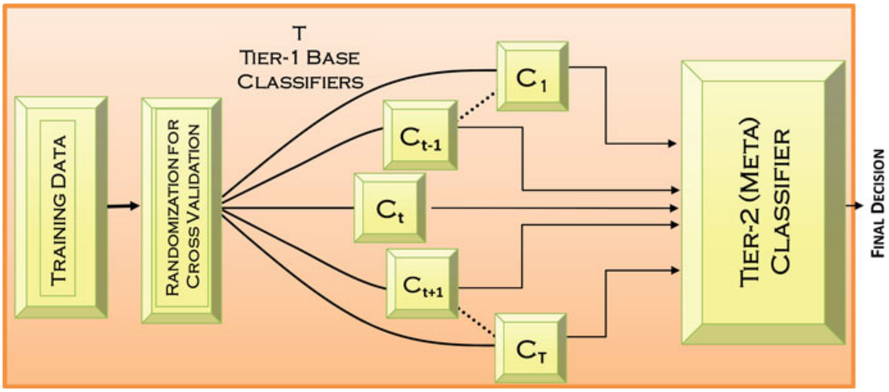


Fig. 1.3 Stacked generalization

the Tier-2 (meta) classifier, which effectively serves as the combination rule for the Tier-1 classifiers. Note that the meta-classifier is not trained on the original feature space, but rather on the decision space of Tier-1 classifiers.

Once the meta-classifier is trained, all Tier-1 classifiers (each of which has been trained B times on overlapping subsets of the original training data) are discarded, and each is retrained on the combined entire training data. The stacked generalization model is then ready to evaluate previously unseen field data.

1.3.4 Mixture of Experts

Mixture of experts is a similar algorithm, also using a trainable combiner. MoE, also trains an ensemble of (Tier-1) classifiers using a suitable sampling technique. Classifiers are then combined through a weighted combination rule, where the weights are determined through a gating network [7], which itself is typically trained using expectation-maximization (EM) algorithm [8,43] on the original training data. Hence, the weights determined by the gating network are dynamically assigned based on the given input, as the MoE effectively learns which portion of the feature space is learned by each ensemble member. Figure 1.4 illustrates the structure of the MoE algorithm.

Mixture-of-experts can also be seen as a classifier selection algorithm, where individual classifiers are trained to become experts in some portion of the feature space. In this setting, individual classifiers are indeed trained to become experts, and hence are usually not weak classifiers. The combination rule then selects the most appropriate classifier, or classifiers weighted with respect to their expertise, for each given instance. The pooling/combining system may then choose a single classifier with the highest weight, or calculate a weighted sum of the classifier outputs for each class, and pick the class that receives the highest weighted sum.

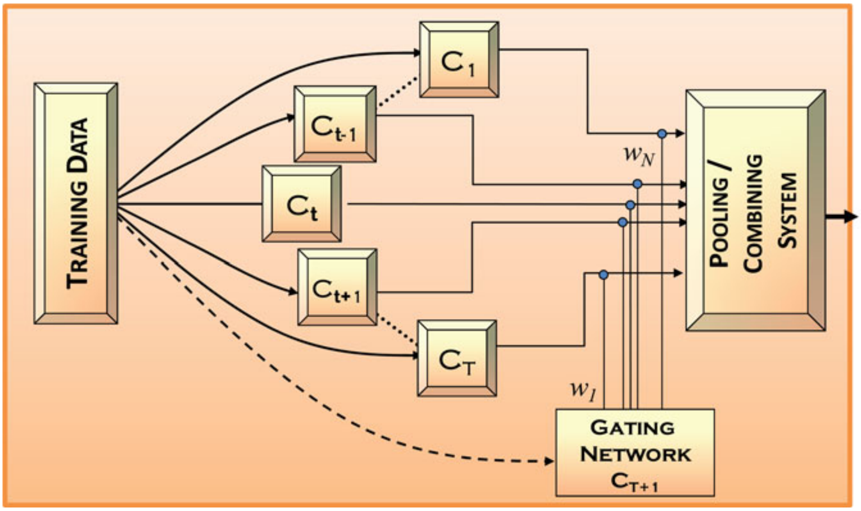


Fig. 1.4 Mixture of experts model

1.4 What Else Can Ensemble Systems Do for You?

While ensemble systems were originally developed to reduce the variability in classifier decision and thereby increase generalization performance, there are many additional problem domains where ensemble systems have proven to be extremely effective. In this section, we discuss some of these emerging applications of ensemble systems along with a family of algorithms, called Learn^{++}, which are designed for these applications.

1.4.1 Incremental Learning

In many real-world applications, particularly those that generate large volumes of data, such data often become available in batches over a period of time. These applications need a mechanism to incorporate the additional data into the knowledge base in an incremental manner, preferably without needing access to the previous data. Formally speaking, incremental learning refers to sequentially updating a hypothesis using current data and previous hypotheses—but not previous data—such that the current hypothesis describes all data that have been acquired thus far. Incremental learning is associated with the well-known stability–plasticity dilemma, where stability refers to the algorithm's ability to retain existing knowledge and plasticity refers to the algorithm's ability to acquire new data. Improving one usually comes at the expense of the other. For example, online data streaming algorithms usually have good plasticity but poor stability, whereas many of the well-established supervised algorithms, such as MLP, SVM, and kNN have good stability but poor plasticity properties.

Ensemble-based systems provide an intuitive approach for incremental learning that also provides a balanced solution to the stability–plasticity dilemma. Consider the AdaBoost algorithm which directs the subsequent classifiers toward increasingly difficult instances. In an incremental learning setting, some of the instances introduced by the new batch can also be interpreted as “difficult” if they carry novel information. Therefore, an AdaBoost-like approach can be used in an incremental learning setting with certain modifications, such as creating a new ensemble with each batch that become available; resetting the sampling distribution based on the performance of the existing ensemble on the new batch of training data, and relaxing the abort clause. Note, however, that distribution update rule in AdaBoost directs the sampling distribution toward those instances misclassified by the *previous classifier*. In an incremental learning setting, it is necessary to direct the algorithm to focus on those novel instances introduced by the new batch of data that are not yet learned by the *current ensemble*, not by the previous classifier. Learn^{++} algorithm, introduced in [44, 45], incorporate these ideas.

The incremental learning problem becomes particularly challenging if the new data also introduce new classes. This is because classifiers previously trained on earlier batches of data inevitably misclassify instances of the new class on which they were not trained. Only the new classifiers are able to recognize the new class(es). Therefore, any decision by the *new* classifiers correctly choosing the new class is outvoted by the earlier classifiers, until there are enough new classifiers to counteract the total vote of those original classifiers. Hence, a relatively large number of new classifiers that recognize the new class are needed, so that their total weight can overwrite the incorrect votes of the original classifiers.

The Learn^{++}.NC (for *N*ew *C*lasses), described in Algorithm 3, addresses these issues [46] by assigning dynamic weights to ensemble members, based on its prediction of which classifiers are likely to perform well on which classes. Learn^{++}.NC cross-references the predictions of each classifier—with those of others—with respect to classes on which they were trained. Looking at the decisions of other classifiers, each classifier decides whether its decision is in line with the predictions of others, and the classes on which it was trained. If not, the classifier reduces its vote, or possibly refrains from voting altogether. As an example, consider an ensemble of classifiers, E_1, trained with instances from two classes ω_1, and ω_2; and a second ensemble, E_2, trained on instances from classes ω_1, ω_2, and a new class, ω_3. An instance from the new class ω_3 is shown to all classifiers. Since E_1 classifiers do not recognize class ω_3, they incorrectly choose ω_1 or ω_2, whereas E_2 classifiers correctly recognize ω_3. Learn^{++}.NC keeps track of which classifiers are trained on which classes. In this example, knowing that E_2 classifiers have seen ω_3 instances, and that E_1 classifiers have not, it is reasonable to believe that E_2 classifiers are correct, particularly if they overwhelmingly choose ω_3 for that instance. To the extent E_2 classifiers are confident of their decision, the voting weights of E_1 classifiers can therefore be reduced. Then, E_2 no longer needs a large number of classifiers: in fact, if E_2 classifiers agree with each other on their correct decision, then very few classifiers are adequate to remove any bias induced by E_1. This voting process, described in Algorithm 4, is called *dynamically weighted consult-and-vote* (DW-CAV) [46].

Algorithm 3 Learn^{++}.NC

```
Input: For each dataset $k = 1, \ldots, K$, training data $S_k = \{\boldsymbol{x}_i, y_i\}$, $i = 1, \ldots, N_k$
$y_i \in \Omega = \{\omega_1, \ldots, \omega_C\}$, supervised learner BaseClassifier; ensemble size $T_k$.
Do for $k = 1, \ldots, K$.
    1. Initialize instance weights $w_1^k(i) = 1/N_k$.
    2. If $k \neq 1$, Set $t = 0$ and Go to Step 5 to adjust initialization weights.
    Do for $t = 1, \ldots, T_k$.
    1. Set
          $D_t^k = \boldsymbol{w}_t^k \Big/ \sum_{i=1}^{N_k} w_t^k(i)$          (1.22)
       so that $D_t^k$ is a distribution.
    2. Train BaseClassifier on $TR_t^k \subset S_k$ drawn from $D_t^k$, receive $h_t^k$.
    3. Calculate error
          $\varepsilon_t^k = \sum_i I\,[\![h_t^k(\boldsymbol{x}_i \neq y_i)]\!]\, D_t^k(\boldsymbol{x}_i)$          (1.23)
    4. If $\varepsilon_t^k > \frac{1}{2}$ discard $h_t^k$ and go to Step 2. Otherwise, normalize $\varepsilon_t^k$:
       Normalize $\varepsilon_t^k$ :
          $\beta_t^k = \varepsilon_t^k / \left(1 - \varepsilon_t^k\right)$          (1.24)
    5. Let $CL_t^k$ be the set of class labels used in training $h_t^k$ for dataset $S_k$.
    6. Call DW-CAV to obtain the composite hypothesis $H_t^k$.
    7. Compute the error of the composite hypothesis
          $E_t^k = \sum_i I\,[\![H_t^k(\boldsymbol{x}_i \neq y_i)]\!]\, D_t^k(\boldsymbol{x}_i)$          (1.25)
    8. Normalize $E_t^k$: $B_t^k = E_t^k \big/ \left(1 - E_t^k\right)$, and update the weights:
          $W_{t+1}^k(i) = w_t^k(i) \cdot \begin{cases} B_t^k, & H_t^k(\boldsymbol{x}_i = y_i) \\ 1, & \text{otherwise} \end{cases}$          (1.26)
End
End
Call DW-CAV to obtain the final hypothesis, $H_{final}$.
```

Algorithm 4 DW-CAV (Dynamically Weighed—Consult and Vote).

Inputs: Instance $\boldsymbol{x}_i$ to be classified; all classifiers h_t^k generated thus far; normalized error values, β_t^k; class labels, CL_t^k used in training h_t^k.
Initialize classifier voting weights $W_t^k = \log\left(1/\beta_t^k\right)$.
Calculate for each $\omega_c \in \{\omega_1, \ldots, \omega_C\}$.

1. Normalization factor:

$$Z_c = \sum\nolimits_k \sum\nolimits_{t:c \in CL_t^k} W_t^k \tag{1.27}$$

2. Class-specific confidence:

$$P_c(i) = \frac{\sum_k \sum_{t:h_k^t(x_i) = \omega_c} W_t^k}{Z_c} \tag{1.28}$$

3. If $P_k(i) = P_l(i), k \neq l$ such that $\mathcal{E}_k \cap \mathcal{E}_l = \emptyset \, P_k(i) = P_l(i) = 0$

where $\mathcal{E}_k$ is the set of classifiers that have seen class ω_k.
Update voting weights for instance x_i

$$W_t^k(i) = W_t^k \cdot \prod\nolimits_{c:\omega_c \notin CL_t^k} (1 - P_c(i)) \tag{1.29}$$

Compute final (current composite) hypothesis

$$H_{\text{final}}(x_i) = \arg\max_{\omega \in \Omega} \sum\nolimits_k \sum\nolimits_{t:h_k^t(x_i) = \omega_c} W_t^k(i) \tag{1.30}$$

Specifically, Learn^{++}.NC updates its sampling distribution based on the composite hypothesis H ((1.25)), which is the ensemble decision of all classifiers generated thus far. The composite hypothesis H_t^k for the first t classifiers from the k^{th} batch is computed by the weighted majority voting of all classifiers using the weights W_t^k, which themselves are weighted based on each classifiers class-specific confidence P_c ((1.27) and (1.28)).

The class-specific confidence $P_c(i)$ for instance $\boldsymbol{x}_i$ is the ratio of total weight of all classifiers that choose class ω_c (for instance $\boldsymbol{x}_i$), to the total weight of all classifiers that have seen class ω_c. Hence, $P_c(i)$ represents the collective confidence of classifiers trained on class ω_c in choosing class ω_c for instance $\boldsymbol{x}_i$. A high value of $P_c(i)$, close to 1, indicates that classifiers trained to recognize class ω_c have in fact overwhelmingly picked class ω_c, and hence those that were not trained on ω_c should not vote (or reduce their voting weight) for that instance.

Extensive experiments with Learn^{++}.NC showed that the algorithm can very quickly learn new classes when they are present, and in fact is also able to remember a class, when it is no longer present in future data batches [46].

1.4.2 Data Fusion

A common problem in many large-scale data analysis and automated decision making applications is to combine information from different data sources that often provide heterogeneous data. Diagnosing a disease from several blood or behavioral tests, imaging results, and time series data (such as EEG or ECG) is such an application. Detecting the health of a system or predicting weather patterns based on data from a variety of sensors, or the health of a company based on several sources of financial indicators are other examples of data fusion. In most data fusion applications, the data are heterogeneous, that is, they are of different format, dimensionality, or structure: some are scalar variables (such as blood pressure, temperature, humidity, speed), some are time series data (such as electrocardiogram, stock prices over a period of time, etc.), some are images (such as MRI or PET images, 3D visualizations, etc.).

Ensemble systems provide a naturally suited solution for such problems: individual classifiers (or even an ensemble of classifiers) can be trained on each data source and then combined through a suitable combiner. The stacked generalization or MoEs structures are particularly well suited for data fusion applications. In both cases, each classifier (or even a model of ensemble of classifiers) can be trained on a separate data source. Then, a subsequent meta-classifier or a gating network can be trained to learn which models or experts have better prediction accuracy, or which ones have learned which feature space. Figure 1.5 illustrates this structure.

A comprehensive review of using ensemble-based systems for data fusion, as well as detailed description of Learn^{++} implementation for data fusion—shown to be quite successful on a variety of data fusions problems—can be found in [47]. Other ensemble-based fusion approaches include combining classifiers using Dempster–Shafer-based combination [48–50], ARTMAP [51], genetic algorithms [52], and other combinations of boosting/voting methods [53–55]. Using diversity metrics for ensemble-based data fusion is discussed in [56].

1.4.3 Feature Selection and Classifying with Missing Data

While most ensemble-based systems create individual classifiers by altering the training data instances—but keeping all features for a given instance—individual features can also be altered by using all of the training data available. In such a setting, individual classifiers are trained with different subsets of the entire feature set. Algorithms that use different feature subsets are commonly referred to as random subspace methods, a term coined by Ho [36]. While Ho used this approach for creating random forests, the approach can also be used for feature selection as well as diversity enhancement.

Another interesting application of RSM-related methods is to use the ensemble approach to classify data that have missing features. Most classification algorithms have matrix multiplications that require the entire feature vector to be available.

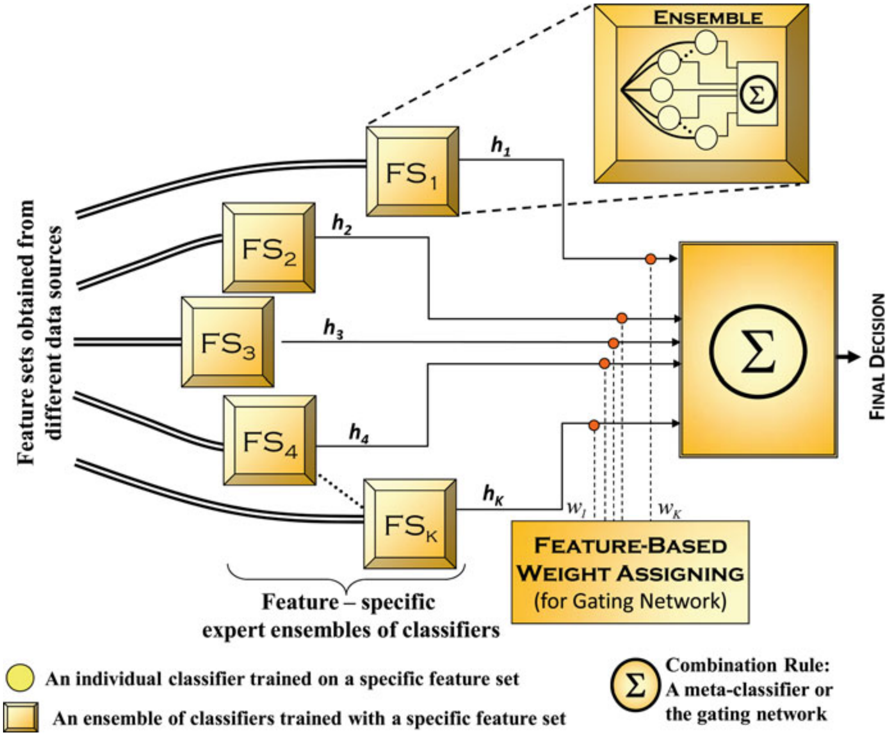
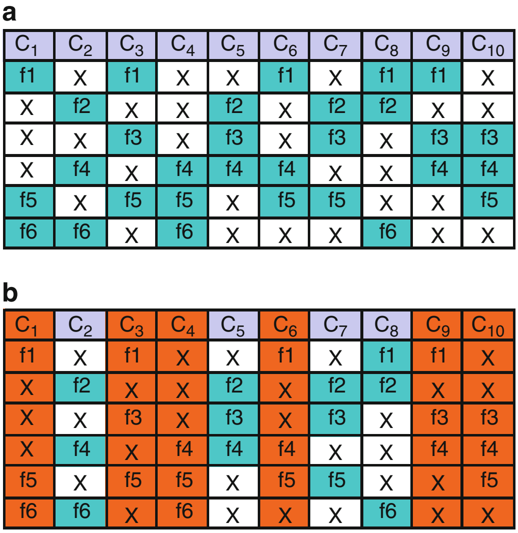

Ensemble
Feature sets obtained from different data sources
FS1
FS2
FS3
FS4
FSK
h1
h2
h3
h4
hK
Σ
Final Decision
w1
wK
Feature-Based Weight Assigning (for Gating Network)
Feature – specific expert ensembles of classifiers
An individual classifier trained on a specific feature set
An ensemble of classifiers trained with a specific feature set
Combination Rule: A meta-classifier or the gating network

Fig. 1.5 Ensemble systems for data fusion

However, missing data is quite common in real-world applications: bad sensors, failed pixels, unanswered questions in surveys, malfunctioning equipment, medical tests that cannot be administered under certain conditions, etc. are all common scenarios in practice that can result in missing attributes. Feature values that are beyond the expected dynamic range of the data due to extreme noise, signal saturation, data corruption, etc. can also be treated as missing data.

Typical solutions to missing features include imputation algorithms where the value of the missing variable is estimated based on other observed values of that variable. Imputation-based algorithms (such as expectation maximization, mean imputation, k-nearest neighbor imputation, etc.), are popular because they are theoretically justified and tractable; however, they are also prone to significant estimation errors particularly for large dimensional and/or noisy datasets.

An ensemble-based solution to this problem was offered in Learn^{++}.MF [57] (MF for Missing Features), which generates a large number of classifiers, each of which is trained using only random subsets of the available features. The instance sampling distribution in other versions of Learn^{++} algorithms is replaced with a



a

C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}
f1	X	f1	X	X	f1	X	f1	f1	X
X	f2	X	X	f2	X	f2	f2	X	X
X	X	f3	X	f3	X	f3	X	f3	f3
X	f4	X	f4	f4	f4	X	X	f4	f4
f5	X	f5	f5	X	f5	f5	X	X	f5
f6	f6	X	f6	X	X	X	f6	X	X

b

C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}
f1	X	f1	X	X	f1	X	f1	f1	X
X	f2	X	X	f2	X	f2	f2	X	X
X	X	f3	X	f3	X	f3	X	f3	f3
X	f4	X	f4	f4	f4	X	X	f4	f4
f5	X	f5	f5	X	f5	f5	X	X	f5
f6	f6	X	f6	X	X	X	f6	X	X

Fig. 1.6 (**a**) Training classifiers with random subsets of the features; (**b**) classifying an instance missing feature f_2. Only shaded classifiers can be used

feature sampling distribution, which favors those features that have not been well represented in the previous classifiers' feature sets. Then, a data instance with missing features is classified using the majority voting of only those classifiers whose feature sets did not include the missing attributes. This is conceptually illustrated in Fig. 1.6a, which shows 10 classifiers, each trained on three of the six features available in the dataset. Features that are not used during training are indicated with an "X." Then, at the time of testing, let us assume that feature number 2, f_2, is missing. This means that those classifiers whose training feature sets included f_2, that is, classifiers C_2, C_5, C_7, and C_8, cannot be used in classifying this instance. However, the remaining classifiers, shaded in Fig. 1.6b, did not use f_2 during their training, therefore those classifiers can still be used.

Learn^{++}.MF is listed in Algorithm 5 below. Perhaps the most important parameter of the algorithm is *nof*, the number of features, out of a total of f, to be used to train each classifier. Choosing a smaller *nof* allows a larger number of missing features to be accommodated by the algorithm. However, choosing a larger *nof* usually improves individual classifier performances. The primary assumption made by Learn^{++}.MF is that the dataset includes a redundant set of features, and the problem is at least partially solvable using a subset of the features, whose identities are unknown to us. Of course, if we knew the identities of those features, we would only use those features in the first place.

A theoretical analysis of this algorithm, including probability of finding at least one useable classifier in the absence of m missing features, when each classifier is trained using *nof* of a total of f features, as well as the number of classifiers needed to guarantee at least one useable classifier are provided in [57].

```
Algorithm 5 Learn$^{++}$.MF

Inputs: Sentinel value $sen$, BaseClassifier; the number of classifiers, $T$.
Training dataset $S = \{x_i, y_i\}, i = 1, \dots, N$, with $N$ instances of $f$ features from
$c$ classes, number of features used to train each classifier, $nof$;
Initialize feature distribution $D_1(j) = 1/f, \forall j, j = 1, \dots, f$;
Do for $t = 1, \dots, T$.
    1. Normalize $D_t$ to make it a proper distribution.
    2. Draw $nof$ features from $D_t$ to form selected features: $F_{selection}(t)$.
    3. Call BaseClassifier to train classifier $C_t$ using only those features in
       $F_{selection}(t)$.
    4. Add $C_t$ to the ensemble $\mathcal{E}$
    5. Obtain $Perf(t)$ the classification performance on $S$. If $Perf(t) < 1/c$, discard
       $C_t$ and go to Step 2.
    6. Update feature distribution

       $D_{t+1}\left(F_{\text{selection}}(t)\right) = (nof/f) \cdot D_t\left(F_{\text{selection}}(t)\right)$    (1.31)
End

Using trained ensemble
Given test/field data $z$,
    1. Determine missing features $M(z) = \arg(z(j) == sen), \forall j$
    2. Obtain ensemble decision as the class with the most votes among the outputs
       of classifiers $C_t^*$ trained on the nonmissing features:

       $\varepsilon(z) = \arg\max_y \sum_{t:C_t^*(z)=y} \llbracket M(z) \cap F_{\text{selection}}(t) \neq \emptyset \rrbracket$    (1.32)
```

1.4.4 Learning from Nonstationary Environments: Concept Drift

Much of computational intelligence literature is devoted to algorithms that can learn from data that are assumed to be drawn from a fixed but unknown distribution. For a great many applications, however, this assumption is simply not true. For example, predicting future weather patterns from current and past climate data, predicting future stock returns from current and past financial data, identifying e-mail spam from current and past e-mail content, determining which online adds a user will respond based on the user's past web surfing record, predicting future energy demand and prices based on current and past data are all examples of applications where the nature and characteristics of the data—and the underlying phenomena that generate such data—may change over time. Therefore, a learning model trained at a fixed point in time—and a decision boundary generated by such a model—may not reflect the current state of nature due to a change in the underlying environment. Such an environment is referred to as a nonstationary environment, and the problem of learning in such an environment is often referred to as learning *concept drift*. More specifically, given the Bayes posterior probability of class ω that a given instance x belongs, $P(\omega|x) = P(x|\omega)P(\omega)/P(x)$, concept drift can be formally defined as any scenario where the posterior probability changes over time, i.e., $P^{t+1}(\omega|x) \neq P^{t}(\omega|x)$.

To be sure, this is a very challenging problem in machine learning because the underlying change may be gradual or rapid, cyclical or noncyclical, systematic or random, with fixed or variable rate of drift, and with local or global activity in the feature space that spans the data. Furthermore, concept drift can also be perceived, rather than real, as a result of insufficient, unknown, or unobservable features in a dataset, a phenomenon known as *hidden context* [58]. In such a case, an underlying phenomenon provides a true and static description of the environment over time, which, unfortunately, is hidden from the learner's view. Having the benefit of knowing this hidden context would make the problem to have a fixed (and hence stationary) distribution.

Concept drift problems are usually associated with incremental learning or learning from a stream of data, where new data become available over time. Combining several authors' suggestions for desired properties of a concept drift algorithms, Elwell and Polikar provided the following guidelines for addressing concept drift problems: (1) any given instance of data—whether provided online or in batches—can only be used once for training (one-pass incremental learning); (2) knowledge should be labeled with respect to its relevance to the current environment, and be dynamically updated as new data continuously arrive; (3) the learner should have a mechanism to reconcile when existing and newly acquired knowledge conflict with each other; (4) the learner should be able—not only to temporarily forget information that is no longer relevant to the current environment but also to recall prior knowledge if the drift/change in the environment follow a cyclical nature; and (5) knowledge should be incrementally and periodically stored so that it can be recalled to produce the best hypothesis for an unknown (unlabeled) data instance at any time during the learning process [59].

Earliest examples of concept drift algorithms use a single classifier to learn from the latest batch of data available, using some form of windowing to control the batch size. Successful examples of this *instance selection* approach include STAGGER [60] and FLORA [58] algorithms, which use a sliding window to choose a block of (new) instances to train a new classifier. The window size can be dynamically updated using a "window adjustment heuristic," based on how fast the environment is changing. Instances that fall outside of the window are then assumed irrelevant and hence the information carried by them are irrecoverably forgotten. Other examples of this window-based approach include [61–63], which use different drift detection mechanisms or base classifiers. Such approaches are often either not truly incremental as they may access prior data, or cannot handle cyclic environments. Some approaches include a novelty (anomaly) detection to determine the precise moment when changes occur, typically by using statistical measures, such as control charts based CUSUM [64, 65], confidence interval on error [66, 67], or other statistical approaches [68]. A new classifier trained on new data since the last detection of change then replaces the earlier classifier(s).

The ensemble-based algorithms provide an alternate approach to concept drift problems. These algorithms generally belong to one of three categories [69]: (1) update the combination rules or voting weights of a fixed ensemble, such as [70, 71]; an approach loosely based on Littlestone's Winnow [72] and Freund and Schapire's Hedge (a precursor of AdaBoost) [4]; (2) update the parameters of existing ensemble members using an online learner [66, 73]; and/or (3) add new members to build an ensemble with each incoming dataset. Most algorithms fall into this last category, where the oldest (e.g., Streaming Ensemble Algorithm (SEA) [74] or Recursive Ensemble Approach (REA) [75]) or the least contributing ensemble members are replaced with new ones (as in Dynamic Integration [76], or Dynamic Weighted Majority (DWM) [77]). While many ensemble approaches use some form of voting, there is some disagreement on whether the voting should be weighted, e.g., giving higher weight to a classifier if its training data were in the same region as the testing example [76], or unweighted, as in [78, 79], where the authors argue that weights based on previous data, whose distribution may have changed, are uninformative for future datasets. Other efforts that combine ensemble systems with drift detection include Bifet's adaptive sliding window (ADWIN) [80, 81], also available within the WEKA-like software suite, Massive Online Analysis (MOA) at [82].

More recently, a new addition to Learn^{++} suite of algorithms, Learn^{++}.NSE, has been introduced as a general framework to learning concept drift that does not make any restriction on the nature of the drift. Learn^{++}.NSE (for NonStationary Environments) inherits the dynamic distribution-guided ensemble structure and incremental learning abilities of all Learn^{++} algorithms (hence strictly follows the one-pass rule). Learn^{++}.NSE trains a new classifier for each batch of data it receives, and combines the classifiers using a dynamically weighted majority voting. The novelty of the approach is in determining the voting weights, based on each classifier's time-adjusted accuracy on current and past environments, allowing the algorithm to recognize, and act accordingly, to changes in underlying data distributions, including possible reoccurrence of an earlier distribution [59].

The Learn^{++}.NSE algorithm is listed in Algorithm 6, which receives the training dataset $\mathrm{D}^t = \left\{x_i^t \in X; y_i^t \in Y\right\}, \quad i = 1, \ldots, m^t$, at time t. Hence x_i^t is the i^{th} instance of the dataset, drawn from an unknown distribution $P^t(\boldsymbol{x}, y)$, which is the currently available representation of a possibly drifting distribution at time t. At time $t + 1$, a new batch of data is drawn from $P^{t+1}(\boldsymbol{x}, y)$. Between any two consecutive batches, the environment may experience a change whose rate is not known, nor assumed to be constant. Previously seen data are not available to the algorithm, allowing Learn^{++}.NSE to operate in a truly incremental fashion.

Algorithm 6 Learn^{++}.NSE

```
Input: For each dataset $D^t$   $t = 1,2, \ldots$.
Training data $\{x^t(i) \in X;\ y^t(i) \in Y = \{1, \ldots, c\}\}$, $i = 1, \ldots, m^t$; Supervised learning algorithm BaseClassifier; Sigmoid parameters $a$ (slope) and $b$ (inflection point).
Do for $t = 1, 2, \ldots$.
    If $t = 1$, Initialize $D^1(i) = w^t(i) = 1/m^1$, $\forall i$, Go to step 3. Endif
    1. Compute error of the existing ensemble on new data
           $E_t = \sum_{i=1}^{m^t} 1/m^t \cdot [\![H^{t-1}(x^t(i)) \neq y^t(i)]\!]$   (1.33)
    2. Update and normalize instance weights
           $w_i^t = \frac{1}{m^t} \cdot \begin{cases} E^t, & H^{t-1}(x^t(i)) = y^t(i) \\ 1, & \text{otherwise} \end{cases}$   (1.34)
       Set
           $D^t = \mathbf{w}^t \Big/ \sum_{i=1}^{m^t} w^t(i) \Rightarrow D^t$   (1.35)
       is a distribution.
    3. Call BaseClassifier with $D^t$, obtain $h^t: X \to Y$.
    4. Evaluate all existing classifiers on new data $D^t$
           $\varepsilon_k^t = \sum_{i=1}^{m^t} D^t(i)\, [\![h_k(x^t(i)) \neq y^t(i)]\!]$ for $k = 1, \ldots, t$   (1.36)
       If $\varepsilon_{k=t}^t > 1/2$, generate a new $h_t$. If $\varepsilon_{k<t}^t > 1/2$, set $\varepsilon_k^t = 1/2$,
           $\beta_k^t = \varepsilon_k^t/(1 - \varepsilon_k^t)$, for $k = 1, \ldots, t \to 0 \leq \beta_k^t \leq 1$   (1.37)
    5. Sigmoid-based time averaging of normalized errors of $h_k$: For $a, b \in \mathbf{R}$
           $\omega_k^t = 1 \Big/ \left(1 + e^{-a(t-k-b)}\right)$, $\omega_k^t = \omega_k^t \Big/ \sum_{j=0}^{t-k} \omega_k^{t-j}$   (1.38)
           $\bar{\beta}_k^t = \sum_{j=0}^{t-k} \omega_k^{t-j}\ \beta_k^{t-j}$, for $k = 1, \ldots, t$   (1.39)
    6. Calculate classifier voting weights
           $W_k^t = \log\left(1/\bar{\beta}_k^t\right)$, for $k = 1, \ldots, t$   (1.40)
    7. Compute the composite hypothesis (the ensemble decision) as
```

$$H^t(x^t(i)) = \arg\max_c \sum\nolimits_k W_k^t \cdot [\![h_k\left(x^t(i)\right) = c]\!] \tag{1.41}$$

```
End Do.
Return the final hypothesis as the current composite hypothesis.
```

The algorithm is initialized with a single classifier on the first batch of data. With the arrival of each subsequent batch of data, the current ensemble, H^{t-1}—the composite hypothesis of all individual hypotheses previously generated, is first evaluated on the new data (Step 1 in Algorithm 6). In Step 2, the algorithm identifies those examples of the new environment that are not recognized by the existing ensemble, H^{t-1}, and updates the *penalty distribution* D^t. This distribution is used not for instance selection, but rather to assign penalties to classifiers on their ability to identify previously seen or unseen instances. A new classifier h^t, is then trained on the current training data in Step 3. In Step 4, each classifier generated thus far is evaluated on the training data weighted with respect to the penalty distribution. Note that since classifiers are generated at different times, each classifier receives a different number of evaluations: at time t, h^t receives its first evaluation, whereas h^1 is evaluated for t^{th} time. We use $\varepsilon_k^t, k = 1, ..., t$ to denote the error of h_k—the classifier generated at time step k—on dataset D^t. Higher weight is given to classifiers that correctly identify previously unknown instances, while classifiers that misclassify previously known data are penalized. Note that if the newest classifier has a weighted error greater than ½, i.e., if $\varepsilon_{k=t}^t \geq$ ½, this classifier is discarded and replaced with a new classifier. Older classifiers, with error $\varepsilon_{k<t}^t \geq$ ½, however, are retained but have their error saturated at ½ (which later corresponds to zero vote on that environment). The errors are then normalized, creating β_k^t that fall in the [0, 1] range.

In Step 5, classifier error is further weighted (using a sigmoid function) with respect to time so that recent competence (error rate) is considered more heavily. Such a sigmoid-based weighted averaging also serves to smooth out potential large swings in classifiers errors that may be due to noisy data rather than actual drift. Final voting weights are determined in Step 6 as log-normalized reciprocals of the weighted errors: if a classifier performs poorly on the current environment, it receives little or no weight, and is effectively—but only temporarily—removed from the ensemble. The classifier is not discarded; however, it is recalled through assignment of higher voting weights if it performs well on future environments. Learn^{++}.NSE forgets only temporarily, which is particularly useful in cyclical environments. The final decision is obtained in Step 7 as the weighted majority voting of the current ensemble members.

Learn^{++}.NSE has been evaluated and benchmarked against other algorithms, on a broad spectrum of real-world as well as carefully designed synthetic datasets—including gradual and rapid drift, variable rate of drift, cyclical environments, as well as environments that introduce or remove concepts. These experiments and their results are reported in [59], which shows that the algorithm can serve as a general framework for learning concept drift regardless of the environment that characterizes the drift.

1.4.5 Confidence Estimation

In addition to the various machine learning problems described above, ensemble systems can also be used to address other challenges that are difficult or impossible using a single classifier-based systems.

One such application is to determine the confidence of the (ensemble-based) classifier in its own decision. The idea is extremely intuitive as it directly follows the use of ensemble systems in our daily lives. Consider reading user reviews of a particular product, or consulting the opinions of several physicians on the risks of a particular medical procedure. If all—or at least most—users agree in their opinion that the product reviewed is very good, we would have higher confidence in our decision to purchase that item. Similarly, if all physicians agree on the effectiveness of a particular medical operation, then we would feel more comfortable with that procedure. On the other hand, if some of the reviews are highly complementary, whereas others are highly critical that casts doubt in our decision to purchase that item. Of course, in order for our confidence in the "ensemble of reviewers" to be valid, we must believe that the reviewers are independent of each other, and indeed independently review the items. If certain reviewers were writing reviews based on other reviewers' reviews they read, the confidence based on the ensemble becomes meaningless.

This idea can be naturally extended to classifiers. If considerable majority of the classifiers in an ensemble agree on their decisions, than we can interpret that outcome as ensemble having higher confidence in its decision, as opposed to only a mere majority of classifiers choosing a particular class. In fact, under certain conditions, the consistency of the classifier outputs can also be used to estimate the true posterior probability of each class [28]. Of course, similar to the examples given above, the classifier decisions must be independent for this confidence—and the posterior probabilities—to be meaningful.

1.5 Summary

Ensemble-based systems provide intuitive, simple, elegant, and powerful solutions to a variety of machine learning problems. Originally developed to improve classification accuracy by reducing the variance in classifier outputs, ensemble-based systems have since proven to be very effective in a number of problem domains that are difficult to address using a single model-based system.

A typical ensemble-based system consists of three components: a mechanism to choose instances (or features), which adds to the diversity of the ensemble; a mechanism for training component classifiers of the ensemble; and a mechanism to combine the classifiers. The selection of instances can either be done completely at random, as in bagging, or by following a strategy implemented through a dynamically updated distribution, as in boosting family of algorithms. In general, most ensemble-based systems are independent of the type of base classifier used to create the ensemble, a significant advantage that allows using a specific type of classifier that may be known to be best suited for a given application. In that sense, ensemble-based systems are also known as *algorithm-free-algorithms*.

Finally, a number of different strategies can be used to combine the classifiers, though sum rule, simple majority voting and weighted majority voting are the most commonly used ones due to certain theoretical guarantees they provide.

We also discussed a number of problem domains on which ensemble systems can be used effectively. These include incremental learning from additional data, feature selection, addressing missing features, data fusion, and learning from nonstationary data distributions. Each of these areas has several algorithms developed to address the relevant specific issue, which are summarized in this chapter. We also described a suite of algorithms, collectively known as Learn^{++} family of algorithms that is capable of addressing all of these problems with proper modifications to the base approach: all Learn^{++} algorithms are incremental algorithms that use an ensemble of classifiers trained on the current data only, then combined through majority voting. The individual members of Learn^{++} differ from each other according to the particular distribution update rule along with a creative weight assignment that is specific to the problem.

References

1. B. V. Dasarathy and B. V. Sheela, “Composite classifier system design: concepts and methodology,” *Proceedings of the IEEE*, vol. 67, no. 5, pp. 708–713, 1979
2. L. K. Hansen and P. Salamon, “Neural network ensembles,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 10, pp. 993–1001, 1990
3. R. E. Schapire, “The strength of weak learnability,” *Machine Learning*, vol. 5, no. 2, pp. 197–227, June 1990
4. Y. Freund and R. E. Schapire, “Decision-theoretic generalization of on-line learning and an application to boosting,” *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, 1997
5. L. I. Kuncheva, *Combining pattern classifiers, methods and algorithms*. New York, NY: Wiley Interscience, 2005
6. L. Breiman, “Bagging predictors,” *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996
7. R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, “Adaptive mixtures of local experts,” *Neural Computation*, vol. 3, no. 1, pp. 79–87, 1991
8. M. J. Jordan and R. A. Jacobs, “Hierarchical mixtures of experts and the EM algorithm,” *Neural Computation*, vol. 6, no. 2, pp. 181–214, 1994
9. D. H. Wolpert, “Stacked generalization,” *Neural Networks*, vol. 5, no. 2, pp. 241–259, 1992

10. J. A. Benediktsson and P. H. Swain, “Consensus theoretic classification methods,” *IEEE Transactions on Systems, Man and Cybernetics*, vol. 22, no. 4, pp. 688–704, 1992
11. L. Xu, A. Krzyzak, and C. Y. Suen, “Methods of combining multiple classifiers and their applications to handwriting recognition,” *IEEE Transactions on Systems, Man and Cybernetics*, vol. 22, no. 3, pp. 418–435, 1992
12. T. K. Ho, J. J. Hull, and S. N. Srihari, “Decision combination in multiple classifier systems,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 16, no. 1, pp. 66–75, 1994
13. G. Rogova, “Combining the results of several neural network classifiers,” *Neural Networks*, vol. 7, no. 5, pp. 777–781, 1994
14. L. Lam and C. Y. Suen, “Optimal combinations of pattern classifiers,” *Pattern Recognition Letters*, vol. 16, no. 9, pp. 945–954, 1995
15. K. Woods, W. P. J. Kegelmeyer, and K. Bowyer, “Combination of multiple classifiers using local accuracy estimates,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, no. 4, pp. 405–410, 1997
16. I. Bloch, “Information combination operators for data fusion: A comparative review with classification,” *IEEE Transactions on Systems, Man, and Cybernetics Part A:Systems and Humans*, vol. 26, no. 1, pp. 52–67, 1996
17. S. B. Cho and J. H. Kim, “Combining multiple neural networks by fuzzy integral for robust classification,” *IEEE Transactions on Systems, Man and Cybernetics*, vol. 25, no. 2, pp. 380–384, 1995
18. L. I. Kuncheva, J. C. Bezdek, and R. P. W. Duin, “Decision templates for multiple classifier fusion: an experimental comparison,” *Pattern Recognition*, vol. 34, no. 2, pp. 299–314, 2001
19. H. Drucker, C. Cortes, L. D. Jackel, Y. LeCun, and V. Vapnik, “Boosting and other ensemble methods,” *Neural Computation*, vol. 6, no. 6, pp. 1289–1301, 1994
20. L. I. Kuncheva, “Classifier ensembles for changing environments,” *5th International Workshop on Multiple Classifier Systems* in Lecture Notes in Computer Science, eds. F. Roli, J. Kittler, and T. Windeatt, vol. 3077, pp. 1–15, Cagliari, Italy, 2004
21. L. I. Kuncheva, “Switching between selection and fusion in combining classifiers: An experiment,” *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 32, no. 2, pp. 146–156, 2002
22. E. Alpaydin and M. I. Jordan, “Local linear perceptrons for classification,” *IEEE Transactions on Neural Networks*, vol. 7, no. 3, pp. 788–792, 1996
23. G. Giacinto and F. Roli, “Approach to the automatic design of multiple classifier systems,” *Pattern Recognition Letters*, vol. 22, no. 1, pp. 25–33, 2001
24. L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001
25. L. Breiman, “Arcing classifiers,” *Annals of Statistics*, vol. 26, no. 3, pp. 801–849, 1998
26. F. M. Alkoot and J. Kittler, “Experimental evaluation of expert fusion strategies,” *Pattern Recognition Letters*, vol. 20, no. 11–13, pp. 1361–1369, Nov. 1999
27. J. Kittler, M. Hatef, R. P. W. Duin, and J. Mates, “On combining classifiers,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 3, pp. 226–239, 1998
28. M. Muhlbaier, A. Topalis, and R. Polikar, “Ensemble confidence estimates posterior probability,” *6th Int. Workshop on Multiple Classifier Systems*, Lecture Notes on Computer Science, eds. N. C. Oza, R. Polikar, J. Kittler, and F. Roli, Eds., vol. 3541, pp. 326–335, Monterey, CA, 2005
29. L. I. Kuncheva, “A theoretical study on six classifier fusion strategies,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 2, pp. 281–286, 2002
30. Y. Lu, “Knowledge integration in a multiple classifier system,” *Applied Intelligence*, vol. 6, no. 2, pp. 75–86, 1996
31. D. M. J. Tax, M. van Breukelen, R. P. W. Duin, and J. Kittler, “Combining multiple classifiers by averaging or by multiplying?” *Pattern Recognition*, vol. 33, no. 9, pp. 1475–1485, 2000
32. G. Brown, “Diversity in neural network ensembles.” PhD, University of Birmingham, UK, 2004

33. G. Brown, J. Wyatt, R. Harris, and X. Yao, "Diversity creation methods: a survey and categorisation," *Information Fusion*, vol. 6, no. 1, pp. 5–20, 2005
34. A. Chandra and X. Yao, "Evolving hybrid ensembles of learning machines for better generalisation," *Neurocomputing*, vol. 69, no. 7–9, pp. 686–700, Mar. 2006
35. Y. Liu and X. Yao, "Ensemble learning via negative correlation," *Neural Networks*, vol. 12, no. 10, pp. 1399–1404, 1999
36. T. K. Ho, "Random subspace method for constructing decision forests," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 8, pp. 832–844, 1998
37. R. E. Banfield, L. O. Hall, K. W. Bowyer, and W. P. Kegelmeyer, "Ensemble diversity measures and their application to thinning," *Information Fusion*, vol. 6, no. 1, pp. 49–62, 2005
38. L. I. Kuncheva and C. J. Whitaker, "Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy," *Machine Learning*, vol. 51, no. 2, pp. 181–207, 2003
39. L. I. Kuncheva, That elusive diversity in classifier ensembles," *Pattern Recognition and Image Analysis, Lecture Notes in Computer Science*, vol. 2652, 2003, pp. 1126–1138
40. N. Littlestone and M. Warmuth, "Weighted majority algorithm," *Information and Computation*, vol. 108, pp. 212–261, 1994
41. R. O. Duda, P. E. Hart, and D. Stork, "Algorithm independent techniques," in *Pattern classification,* 2 edn New York: Wiley, 2001, pp. 453–516
42. L. Breiman, "Pasting small votes for classification in large databases and on-line," *Machine Learning*, vol. 36, no. 1–2, pp. 85–103, 1999
43. M. I. Jordan and L. Xu, "Convergence results for the EM approach to mixtures of experts architectures," *Neural Networks*, vol. 8, no. 9, pp. 1409–1431, 1995
44. R. Polikar, L. Udpa, S. S. Udpa, and V. Honavar, "Learn++: An incremental learning algorithm for supervised neural networks," *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews*, vol. 31, no. 4, pp. 497–508, 2001
45. H. S. Mohammed, J. Leander, M. Marbach, Polikar, and R. Polikar, "Can AdaBoost.M1 learn incrementally? A comparison to Learn++ under different combination rules," *International Conference on Artificial Neural Networks (ICANN2006)* in Lecture Notes in Computer Science, vol. 4131, pp. 254–263, Springer, 2006
46. M. D. Muhlbaier, A. Topalis, and R. Polikar, "Learn++.NC: combining ensemble of classifiers with dynamically weighted consult-and-vote for efficient incremental learning of new classes," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 152–168, 2009
47. D. Parikh and R. Polikar, "An ensemble-based incremental learning approach to data fusion," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 37, no. 2, pp. 437–450, 2007
48. H. Altincay and M. Demirekler, "Speaker identification by combining multiple classifiers using Dempster-Shafer theory of evidence," *Speech Communication*, vol. 41, no. 4, pp. 531–547, 2003
49. Y. Bi, D. Bell, H. Wang, G. Guo, and K. Greer, "Combining multiple classifiers using dempster's rule of combination for text categorization," *First International Conference, MDAI 2004, Aug 2–4 2004* in Lecture Notes in Artificial Intelligence, vol. 3131, Barcelona, Spain, pp. 127–138, 2004
50. T. Denoeux, "Neural network classifier based on Dempster-Shafer theory," *IEEE Transactions on Systems, Man, and Cybernetics Part A:Systems and Humans*, vol. 30, no. 2, pp. 131–150, 2000
51. G. A. Carpenter, S. Martens, and O. J. Ogas, "Self-organizing information fusion and hierarchical knowledge discovery: a new framework using ARTMAP neural networks," *Neural Networks*, vol. 18, no. 3, pp. 287–295, 2005
52. B. F. Buxton, W. B. Langdon, and S. J. Barrett, "Data fusion by intelligent classifier combination," *Measurement and Control*, vol. 34, no. 8, pp. 229–234, 2001
53. G. J. Briem, J. A. Benediktsson, and J. R. Sveinsson, "Use of multiple classifiers in classification of data from multiple data sources," *2001 International Geoscience and Remote Sensing Symposium (IGARSS 2001)*, vol. 2, Sydney, NSW: Institute of Electrical and Electronics Engineers Inc., pp. 882–884, 2001

54. W. Fan, M. Gordon, and P. Pathak, "On linear mixture of expert approaches to information retrieval," *Decision Support Systems*, vol. 42, no. 2, pp. 975–987, 2005
55. S. Jianbo, W. Jun, and X. Yugeng, "Incremental learning with balanced update on receptive fields for multi-sensor data fusion," *IEEE Transactions on Systems, Man and Cybernetics (B)*, vol. 34, no. 1, pp. 659–665, 2004
56. D. Leonard, D. Lillis, L. Zhang, F. Toolan, R. Collier, and J. Dunnion, "Applying machine learning diversity metrics to data fusion in information retrieval," in *Advances in Information Retrieval,* Lecture Notes in Computer Science, vol. 6611, P. Clough, C. Foley, C. Gurrin, G. Jones, W. Kraaij, H. Lee, and V. Mudoch, eds. Springer, Berlin/Heidelberg, 2011, pp. 695–698
57. R. Polikar, J. DePasquale, H. Syed Mohammed, G. Brown, and L. I. Kuncheva, "Learn++.MF: A random subspace approach for the missing feature problem," *Pattern Recognition*, vol. 43, no. 11, pp. 3817–3832, 2010
58. G. Widmer and M. Kubat, "Learning in the presence of concept drift and hidden contexts," *Machine Learning*, vol. 23, no. 1, pp. 69–101, 1996
59. R. Elwell and R. Polikar, "Incremental learning of concept drift in nonstationary environments," *IEEE Transactions on Neural Networks*, doi: 10.1109/TNN.2011.2160459, vol. 22, no. 10, pp. 1517–1531, October 2011
60. J. C. Schlimmer and R. H. Granger, "Incremental learning from noisy data," *Machine Learning*, vol. 1, no. 3, pp. 317–354, Sept. 1986
61. R. Klinkenberg, "Learning drifting concepts: example selection vs. example weighting," *Intelligent Data Analysis, Special Issue on Incremental Learning Systems Capable of Dealing with Concept Drift*, vol. 8, no. 3, pp. 281–300, 2004
62. M. Nunez, R. Fidalgo, and R. Morales, "Learning in environments with unknown dynamics: towards more robust concept learners," *Journal of Machine Learning Research*, vol. 8, pp. 2595–2628, 2007
63. P. Wang, H. Wang, X. Wu, W. Wang, and B. Shi, "A low-granularity classifier for data streams with concept drifts and biased class distribution," *IEEE Transactions on Knowledge and Data Engineering,* vol. 19, no. 9, pp. 1202–1213, 2007
64. C. Alippi and M. Roveri, "Just-in-time adaptive classifiers; part I: detecting nonstationary changes," *IEEE Transactions on Neural Networks*, vol. 19, no. 7, pp. 1145–1153, 2008
65. C. Alippi and M. Roveri, "Just-in-time adaptive classifiers; part II: designing the classifier," *IEEE Transactions on Neural Networks*, vol. 19, no. 12, pp. 2053–2064, 2008
66. J. Gama, P. Medas, G. Castillo, and P. Rodrigues, "Learning with drift detection," *Advances in Artificial Intelligence—SBIA 2004* in Lecture Notes in Computer Science, vol. 3171, pp. 286–295, 2004
67. L. Cohen, G. Avrahami-Bakish, M. Last, A. Kandel, and O. Kipersztok, "Real-time data mining of non-stationary data streams from sensor networks," *Information Fusion*, vol. 9, no. 3, pp. 344–353, 2008
68. M. Markou and S. Singh, "Novelty detection: a review—part 2: neural network based approaches," *Signal Processing*, vol. 83, no. 12, pp. 2499–2521, 2003
69. L. I. Kuncheva, "Classifier ensembles for changing environments," *Multiple Classifier Systems (MCS 2004) in Lecture Notes in Computer Science*, vol. 3077, pp. 1–15, 2004
70. A. Blum, "Empirical support for winnow and weighted-majority algorithms: results on a calendar scheduling domain," *Machine Learning*, vol. 26, no. 1, pp. 5–23, 1997
71. Z. Xingquan, W. Xindong, and Y. Ying, "Dynamic classifier selection for effective mining from noisy data streams," *Fourth IEEE International Conference on Data Mining (ICDM '04)*, pp. 305–312, 2004
72. N. Littlestone, "Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm," *Machine Learning*, vol. 2, no. 4, pp. 285–318, Apr. 1988
73. N. Oza, "Online ensemble learning." Ph.D. Dissertation, University of California, Berkeley, 2001
74. W. N. Street and Y. Kim, "A streaming ensemble algorithm (SEA) for large-scale classification," *Seventh ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD-01)*, pp. 377–382, 2001

75. S. Chen and H. He, "Towards incremental learning of nonstationary imbalanced data stream: a multiple selectively recursive approach," *Evolving Systems*, vol. in press 2011
76. A. Tsymbal, M. Pechenizkiy, P. Cunningham, and S. Puuronen, "Dynamic integration of classifiers for handling concept drift," *Information Fusion*, vol. 9, no. 1, pp. 56–68, Jan. 2008
77. J. Z. Kolter and M. A. Maloof, "Dynamic weighted majority: an ensemble method for drifting concepts," *Journal of Machine Learning Research*, vol. 8, pp. 2755–2790, 2007
78. J. Gao, W. Fan, and J. Han, "On appropriate assumptions to mine data streams: analysis and practice," *International Conference on Data Mining*, pp. 143–152, 2007
79. J. Gao, B. Ding, F. Wei, H. Jiawei, and P. S. Yu, "Classifying data streams with skewed class distributions and concept drifts," *IEEE Internet Computing*, vol. 12, no. 6, pp. 37–49, 2008
80. A. Bifet, "Adaptive learning and mining for data streams and frequent patterns." Ph.D. Dissertation, Universitat Politècnica de Catalunya, 2009
81. A. Bifet, E. Frank, G. Holmes, and B. Pfahringer, "Accurate ensembles for data streams: Combining restricted Hoeffding trees using stacking," *2nd Asian Conference on Machine Learning* in Journal of Machine Learning Research, vol. 13, Tokyo, 2010
82. A. Bifet, MOA: Massive Online Analysis, Available at: http://moa.cs.waikato.ac.nz/. Lastaccessed:7/22/2011