Learn^{++}.NC: Combining Ensemble of Classifiers With Dynamically Weighted Consult-and-Vote for Efficient Incremental Learning of New Classes

Michael D. Muhlbaier, Apostolos Topalis, and Robi Polikar, *Senior Member, IEEE*

***Abstract*—We have previously introduced an incremental learning algorithm Learn^{++}, which learns novel information from consecutive data sets by generating an ensemble of classifiers with each data set, and combining them by weighted majority voting. However, Learn^{++} suffers from an inherent "outvoting" problem when asked to learn a new class $\omega_{\rm new}$ introduced by a subsequent data set, as earlier classifiers not trained on this class are guaranteed to misclassify $\omega_{\rm new}$ instances. The collective votes of earlier classifiers, for an inevitably incorrect decision, then outweigh the votes of the new classifiers' correct decision on $\omega_{\rm new}$ instances—until there are enough new classifiers to counteract the unfair outvoting. This forces Learn^{++} to generate an unnecessarily large number of classifiers. This paper describes Learn^{++}.NC, specifically designed for efficient incremental learning of multiple *N*ew *C*lasses using significantly fewer classifiers. To do so, Learn^{++}.NC introduces *dynamically weighted consult and vote* (DW-CAV) , a novel voting mechanism for combining classifiers: individual classifiers consult with each other to determine which ones are most qualified to classify a given instance, and decide how much weight, if any, each classifier's decision should carry. Experiments on real-world problems indicate that the new algorithm performs remarkably well with substantially fewer classifiers, not only as compared to its predecessor Learn^{++}, but also as compared to several other algorithms recently proposed for similar problems.**

***Index Terms*—Consult-and-vote majority voting, incremental learning, multiple-classifier systems.**

I. Introduction

OBTAINING dense and representative data sets—essential for proper training of a supervised classifier—is often expensive, tedious, and time consuming for many real-world applications. Hence, it is not uncommon for real-world data to be acquired in smaller batches over time. Such scenarios require a classifier that can incrementally learn the novel information provided by the new data, without forgetting the previously acquired knowledge. Long term medical studies, climate or financial data analysis applications—where data are continuously obtained over several years—or any process that generates streaming data are examples of real-world applications that can benefit from such a classifier.

The ability of a classifier to learn under these conditions is known as *incremental learning*, whose several definitions have appeared in the literature. One of the earliest formal definitions is Gold's description of *learning in the limit* [1], which assumes that learning continues indefinitely, and the algorithm has access to all (potentially infinite) data generated thus far. Definitions with various restrictions, such as whether the learner has partial or no access to previous data [2]–[4], or whether new classes or new features are introduced with additional data [5], have also been proposed. In this work, we adopt a more general definition as suggested by several authors [4], [6]–[8]: a learning algorithm is incremental if, for a sequence of training data sets (or instances), it produces a sequence of hypotheses, where the current hypothesis describes all data that have been seen thus far, but depends only on previous hypotheses and the current training data. Hence, an incremental learning algorithm must learn the new information, and retain previously acquired knowledge, without having access to previously seen data.

This definition raises the so-called *stability–plasticity dilemma*: some information will inevitably be lost to learn new information [9]. A strictly stable classifier can preserve existing knowledge, but cannot learn new information, whereas a strictly plastic classifier can learn new information, but cannot retain prior knowledge. Whereas a gradual loss may be inevitable, sudden and complete loss of previously acquired knowledge should be avoided [10]. Hence, the goal in incremental learning is to learn the novel information while retaining as much of the previous knowledge as possible. Many popular classifiers, however, are not structurally suitable for incremental learning; either because they are stable [such as the multilayer perceptron (MLP), radial basis function (RBF) networks, or support vector machines (SVM)], or because they have high plasticity and cannot retain previously acquired knowledge, without having access to old data (such as k-nearest neighbor).

A further complication arises if the additional data introduce new concept classes. Not to be confused with *concept drift* [11] (which refers to class definitions continuously changing in time), introducing new classes represents a particularly hostile and abrupt change in the underlying data distribution. These problems are encountered often in practice, such as recognizing

Manuscript received May 11, 2007; revised November 16, 2007 and April 25, 2008; accepted June 12, 2008. First published December 22, 2008; current version published January 05, 2009. This work was supported by the U.S. National Science Foundation under Grant ECS 0239090.

M. D. Muhlbaier was with the Electrical and Computer Engineering, Rowan University, Glassboro, NJ 08028 USA. He is now with Spaghetti Engineering, Blackwood, NJ 08012 USA.

R. Polikar is with the Electrical and Computer Engineering, Rowan University, Glassboro, NJ 08028 USA (e-mail: polikar@rowan.edu).

A. Topalis was with the Electrical and Computer Engineering, Rowan University, Glassboro, NJ 08028 USA. He is now with Lockheed Martin, Moorsetown, NJ 08057 USA (e-mail: apostolos.topalis@lmco.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNN.2008.2008326

foreign characters for a handwritten text recognition system originally trained on English characters only [12]; identifying new types of defects in a gas-pipeline inspection system originally trained to recognize cracks only [13]; automated identification of additional arrhythmias with a system originally designed to detect other cardiovascular disorders [14], or recognizing new types of obstacles for an autonomous vehicle. Therefore, the ability to learn in these environments would be a desirable asset for an incremental learning algorithm.

We have previously introduced such an incremental learning algorithm, called Learn^{++}, which iteratively generates an ensemble of classifiers for each data set that becomes available, and combines them using weighted majority voting [15]. Despite its promising performance on a variety of applications, Learn^{++} suffers from a so-called "outvoting" problem, when instances of a previously unseen class are introduced with the new data. The problem is primarily due to earlier classifiers (incorrectly) voting for instances of new classes on which they were not trained. While Learn^{++} can learn new class boundaries, its performance on such applications is unstable, and requires generating a large number of classifiers for each new class to override the incorrect votes cast by earlier classifiers.

In this paper, we propose Learn^{++}.NC, which features a strategic voting-based ensemble combination rule that not only provides a significant improvement in performance and stability, but does so using substantially fewer classifiers than its predecessor Learn^{++}. Specifically, the new voting procedure treats individual classifiers as intelligent experts, where they consult with each other to collectively determine which classifiers have more confidence in identifying a given instance, and dynamically set their voting weights accordingly for that instance. For example, a classifier may determine that it has little or no knowledge of a particular class predicted by other classifiers, and withdraw its decision.

The rest of this paper is organized as follows. After a brief review of related work in Section II, we explain the outvoting problem in detail and formally introduce the Learn^{++}.NC algorithm in Section III. In Section IV, we present results on synthetic and real-world applications, and compare these results to those obtained by other recent algorithms proposed for similar applications. Concluding remarks, and the conditions under which Learn^{++}.NC is expected to perform well are discussed in Section V.

II. Background and Related Works

A. Incremental Learning

An often-used pragmatic approach for learning from new data is to discard the existing classifier, and retrain a new one using all data accumulated thus far. This approach results in loss of all previously acquired knowledge, commonly known as *catastrophic forgetting* [10]. In addition to violating the formal definition of incremental learning, this approach is also undesirable if retraining is computationally or financially costly, and may not even be feasible, if the original data are lost, corrupted, discarded, or otherwise unavailable. At the other end of the spectrum are theoretically tractable *online learning* algorithms, including Winston's seminal work [16], Littlestone's *Winnow* [17], and their recent variations [18]–[20]. However, online algorithms typically assume rather restricted forms of classifiers, such as those that learn Boolean or linear threshold functions, which limit their applicability [19], [20]. Lange *et al.* showed that the limited learning power of such algorithms can be improved if they are allowed to memorize a carefully selected subset of the previous data [4]. Maloof *et al.* follow such an approach using partial memory learning (PML), where the instances memorized are the "extreme examples" that lie along the decision boundaries [11]. A similar philosophy is also employed by so-called boundary methods, or maximum margin methods (such as SVMs) [21], [22]. Ferrari *et al.* use a constraint optimization approach for learning new knowledge, subject to retaining the prior knowledge expressed as constraints [23], whereas Ozawa *et al.* combine feature extraction and classification by using an incrementally updated principal-component-based approach that can learn online or in batches [24]. These approaches, however, have not been designed for, nor evaluated on incremental learning of new classes.

Rule-based systems have also been proposed, where incremental learning is achieved by adding new rules to a rule base, typically within a fuzzy inference system [8], [25], [26]. In other approaches, incremental learning involves controlled modification of classifier weights [27], [28], or architecture update by growing/pruning of decision trees, nodes or clusters [29]–[31], or both as in STAGGER [32] and hybrid decision trees [5]. These approaches evaluate local representation of current nodes, and add new ones or update existing ones, if present ones are not sufficient to represent the decision boundary being learned. Such methods are usually capable of incrementally learning new classes from new data.

Perhaps one of the most successful implementations of this approach is (fuzzy) ARTMAP [33]. ARTMAP generates a new cluster for each pattern that is deemed sufficiently different from the previous ones, and then maps each cluster to a target class. ARTMAP can learn incrementally simply by adding new clusters corresponding to new data. However, ARTMAP suffers from sensitivity to the selection of its vigilance parameter, to the noise levels in the training data, and to the order in which data are presented, as well as cluster proliferation. Various solutions, modifying ARTMAP [10], [34]–[36], or other approaches such as growing neural gas networks and cell structures [37] have also been proposed.

B. Ensemble of Classifiers

We propose an alternative approach to incremental learning: train an ensemble of classifiers for each data set that become available, and combine their outputs using an appropriate combination rule. The goal in ensemble systems is to generate a diverse set of classifiers, so that each realizes a slightly different decision boundary. Classifiers then make different errors on different instances, and a strategic combination of these classifiers can reduce the total error. Since its humble beginnings with such seminal works including [38]–[43], ensemble systems has recently become an important research area [44].

Ensemble systems typically use a classifier selection or a classifier fusion paradigm [44], [45]. In *classifier selection*, each classifier becomes an expert in some local area of the feature

space. The final decision is then based on the given instance: classifiers trained with data close to the vicinity of this instance are given higher credit [40], [45]–[47]. In *classifier fusion*, classifiers are trained over the entire feature space. The ensemble then provides a consensus opinion by combining the individual classifier decisions, such as in bagging- [48] or boosting-type algorithms [39], [49], [50], including Learn^{++}. Learn^{++}.NC, however, it is a hybrid algorithm that includes elements of both classifier selection and classifier fusion.

Once the classifiers are generated, one of many classifier combination strategies can be employed, such as voting or algebraic combinations of posterior probabilities [43], [51], Dempster–Shafer combination [52], decision templates [53], or meta decision trees [54]. Recent reviews of ensemble-based algorithms, various associated combination strategies, and their applications can be found in [55] and [56].

Thanks to this substantial body of work, many properties of ensemble systems, such as how they improve performance by reducing bias and variance, are now well understood [44]; however, their feasibility in addressing the incremental learning problem had been largely unexplored. This prompted us to develop Learn^{++} [15]. Since then, additional ensemble-based incremental learning algorithms have been proposed, such as online boosting [57], [58], combining single-class subnets [59], streaming ensemble algorithms (SEAs) [60], and dynamic weighted majority (DWM) [61]. These algorithms have not yet been tested on incremental learning of new classes, and their ability to perform under such settings is unknown as of this writing (except DWM, which we implemented and compared to Learn^{++}.NC). In fact, these algorithms are originally proposed either for online learning of stationary concepts, or for traditional concept-drift applications, where *existing* class definitions change in time. Consequently, most concept-drift algorithms include a forgetting mechanism, typically discarding classifiers that meet/fail certain age or performance criteria. Hence, previously acquired knowledge is gradually and *deliberately* lost. Unless the forgetting mechanism can be appropriately adjusted (as in DWM), these algorithms are typically not suitable for learning new class information, while retaining existing ones.

III. Ensemble-Based Incremental Learning

A. Learn^{++} and the Outvoting Problem in Learning New Concept Classes

Learn^{++} was originally inspired in part by AdaBoost: both algorithms sequentially generate an ensemble of classifiers, trained with bootstrapped samples of the training data drawn from an iteratively updated distribution. The primary difference between the two algorithms is in the distribution update rule. In Learn^{++}, the distribution is iteratively biased towards the novel instances that have not been properly learned by the *current ensemble*, as opposed to instances deemed difficult by the *previous classifier*, as in AdaBoost [49]. Classifiers are then combined through weighted majority voting, where voting weights are determined by relative performance of each classifier on training data. As new data become available, Learn^{++} generates additional *ensembles of classifiers*, until the novel information is learned. Learn^{++} has several desirable traits: it works well on a variety of real-world problems [13], it is classifier independent [62], it is surprisingly robust to minor changes in model parameters and to the order of data presentation [62], it can estimate its own confidence [63], and it can handle unbalanced data [64].

These studies have also shown that Learn^{++} is capable of learning new concept classes, albeit at a steep cost: learning new classes requires an increasingly large number of classifiers for each new class to be learned [15], [65]. We now realize that classifier proliferation of Learn^{++} is an artifact of the voting procedure, and in part, the ad hoc reinitialization of the algorithm every time a new data set arrives.

Specifically, if the problem requires incremental learning of new classes, the weighted majority voting becomes unfairly biased against the newly introduced class. To understand the underlying reason, consider an ensemble of T classifiers trained on data set $\boldsymbol{\mathcal{S}}_1$ with instances from C classes $\omega_1, \ldots, \omega_C$. Later, data set $\boldsymbol{\mathcal{S}}_2$ becomes available, introducing instances from a new class ω_{C+1}. We generate a new ensemble, starting with classifier $(T+1)$. During testing, any instance from ω_{C+1} will inevitably be misclassified into one of previously seen C classes by the first T classifiers, since these classifiers were not trained to recognize class ω_{C+1}. Therefore, any decision by the *new* classifiers correctly choosing class ω_{C+1} will be outvoted by the original T classifiers, until there are enough new classifiers to counteract the total vote of those original T 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. Learn^{++}.NC is designed to address—and avoid—this issue of classifier proliferation.

The primary novelty in Learn^{++}.NC is a new classifier combination strategy that allows individual classifiers to consult with each other to determine their voting weights for each test instance. Learn^{++}.NC also uses a new protocol for reinitializing the algorithm when new data arrive. The Learn^{++}.NC algorithm is described in detail in the next sections. Algorithmic and theoretical details of the original Learn^{++} can be found in [15].

B. Learn^{++}.NC

We start with a preview of the proposed combination rule. For any instance, Learn^{++}.NC asks individual classifiers to cross reference their predictions 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 classes others are predicting, and the classes on which it was trained. If not, the classifier reduces its vote, or possibly refrains from voting all together. Consider the cartoon illustration in Fig. 1. An ensemble of (four) classifiers $\boldsymbol{\mathcal{E}}_1$ are trained with instances from classes ω_1, ω_2, and ω_3; and a second ensemble of (say, two) classifiers $\boldsymbol{\mathcal{E}}_2$ are trained with instances from classes ω_1, ω_2, ω_3, and a new class ω_4 [Fig. 1(a)]. Assume that a test instance from ω_4 is then shown to all classifiers. Since $\boldsymbol{\mathcal{E}}_1$ classifiers do not recognize class ω_4, they incorrectly choose one of other classes, whereas $\boldsymbol{\mathcal{E}}_2$ classifiers correctly recognize ω_4. $\boldsymbol{\mathcal{E}}_1$ classifiers notice that a class they do not recognize is selected by all $\boldsymbol{\mathcal{E}}_2$ classifiers [Fig. 1(b)]. Realizing that they have not

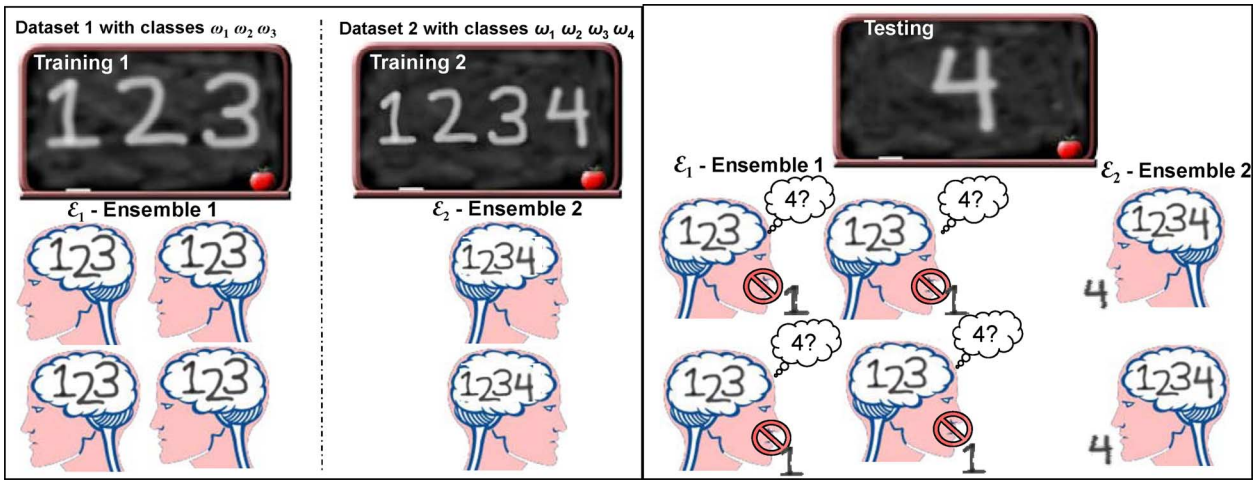

Fig. 1. (a). $\boldsymbol{\mathcal{E}}_1$ classifiers are trained to recognize classes $\omega_1 \sim \omega_3$, whereas $\boldsymbol{\mathcal{E}}_2$ classifiers are trained with new data that also includes class ω_4. (b) During testing, $\boldsymbol{\mathcal{E}}_1$ classifiers realize a class that they do not recognize; ω_4 is overwhelmingly chosen by $\boldsymbol{\mathcal{E}}_2$ classifiers. Hence, $\boldsymbol{\mathcal{E}}_1$ classifiers refrain from voting.

seen any data from ω_4, but that $\boldsymbol{\mathcal{E}}_2$ classifiers have, $\boldsymbol{\mathcal{E}}_1$ classifiers withhold judgment—to the extent $\boldsymbol{\mathcal{E}}_2$ classifiers are confident of their decision. Hence, $\boldsymbol{\mathcal{E}}_1$ classifiers reduce their voting weights proportional to the ratio of the classifiers in $\boldsymbol{\mathcal{E}}_2$ that pick class ω_4. We will refer to this ratio as the *class specific confidence* of $\boldsymbol{\mathcal{E}}_2$ in predicting class ω_4.

Learn^{++}.NC keeps track of which classifiers are trained on which classes. In this example, knowing that $\boldsymbol{\mathcal{E}}_2$ classifiers have seen class ω_4 instances, and that $\boldsymbol{\mathcal{E}}_1$ classifiers have not, it is reasonable to believe that $\boldsymbol{\mathcal{E}}_2$ classifiers are correct, particularly if they overwhelmingly choose class ω_4 for a given instance. To the extent $\boldsymbol{\mathcal{E}}_2$ classifiers are confident of their decision, the voting weights of $\boldsymbol{\mathcal{E}}_1$ classifiers can therefore be reduced. Then, $\boldsymbol{\mathcal{E}}_2$ no longer needs a large number of classifiers: in fact, if $\boldsymbol{\mathcal{E}}_2$ classifiers agree with each other on their correct decision, then very few classifiers will be adequate to remove any bias induced by $\boldsymbol{\mathcal{E}}_1$. We call this voting process *dynamically weighted consult and vote* (DW-CAV).

We now describe the Learn^{++}.NC algorithm in detail, whose pseudocode and block diagram appear in Figs. 2 and 3, respectively. For the kth database that becomes available to Learn^{++}.NC, the inputs to the algorithm are as follows: 1) $S_k = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \cdots, (\mathbf{x}_{m_k}, y_{m_k})\}$, a sequence of m_k training data instances $\mathbf{x}_i$ along with their labels y_i, $i = 1, \ldots, m_k$; 2) a supervised classification algorithm **BaseClassifier**; and 3) an integer T_k specifying the number of classifiers to be generated using currently available data. For each database, the algorithm generates an ensemble of classifiers, each trained on a different subset of the available training data $\boldsymbol{S_k}$. We have used 2/3 of $\boldsymbol{S_k}$ drawn without replacement (which adds diversity to the ensemble, compared to using the entire data set). The specific instances used to train each classifier are drawn from a distribution $\boldsymbol{D^k}$ obtained from a set of weights $\boldsymbol{w^k}$ maintained on the training data $\boldsymbol{S_k}$. If the algorithm is being trained on its first database $(k = 1)$, the distribution $\boldsymbol{D^k}$ is initialized to be uniform, giving equal probability to all instances to be selected into the first training subset $\boldsymbol{TR_1^k}$. If $k > 1$, then a distribution reinitialization sequence is applied as described later in this section.

For each database, the algorithm iteratively adds T_k classifiers to the ensemble. During the tth iteration (for the kth database), the training data weights $\boldsymbol{w_t^k}$ from the previous iteration are first normalized (step 1) to create a proper weight distribution $\boldsymbol{D_t^k}$

$$\boldsymbol{D_t^k} = \boldsymbol{w_t^k} \Big/ \sum_{i=1}^{m_k} w_t^k(i). \tag{1}$$

A (2/3) subset of $\boldsymbol{S_k}$ is drawn according to $\boldsymbol{D_t^k}$ to obtain $\boldsymbol{TR_t^k}$, the data set used to train the tth classifier (hypothesis) h_t^k, using the **BaseClassifier** (step 2). The error ε_t^k of classifier h_t^k is calculated on $\boldsymbol{S_k}$ as

$$\varepsilon_t^k = \sum_{i:h_t^k(\mathbf{x}_i)\neq y_i} D_t^k(i) = \sum_{i=1}^{m_k} D_t^k(i) \cdot \left[\left[h_t^k(\mathbf{x}_i) \neq y_i\right]\right] \tag{2}$$

where $[[\cdot]]$ evaluates to 1, if the predicate is true, and zero, otherwise.

The BaseClassifier can be any supervised classifier, whose parameters (e.g., the size or error goal of an MLP) can be adjusted to ensure adequate diversity, such that sufficiently different decision boundaries are generated each time the classifier is trained on a different training data set. However, a meaningful minimum performance is enforced: the probability of any classifier to produce the correct labels on a given training data set, weighted according to the current distribution, must be at least 1/2. If classifier outputs are class-conditionally independent, then the overall error monotonically decreases as new classifiers are added. Originally known as the Condorcet jury theorem (1786) [66], this condition is necessary and sufficient for a two-class problem $(C = 2)$; and it is sufficient, but not necessary, for $C > 2$.

If $\varepsilon_t^k > 1/2$, the algorithm deems the current classifier h_t^k to be too weak, discards it, and returns to step 2; otherwise, it calculates the normalized error β_t^k, $0 \leq \beta_t^k \leq 1$ (step 3) as

$$\beta_t^k = \varepsilon_t^k / \left(1 - \varepsilon_t^k\right). \tag{3}$$

Meanwhile, the class labels of the training instances used to generate h_t^k are stored as CL_t^k (step 4)

$$CL_t^k = Y_{k(t)} \subset \{\omega_1, \cdots \omega_C\} \tag{4}$$

Input: For each dataset $k=1,2,\ldots,K$

- Training data $S_k = \{(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_i, y_i), \cdots, (\mathbf{x}_{m_k}, y_{m_k})\}$, $\mathbf{x}_i \in X; y_i \in Y_k \subset \{\omega_1, \ldots, \omega_C\}$
- Weak learning algorithm **BaseClassifier**.
- Integer T_k, specifying the number of **BaseClassifier**s to create using $\mathcal{D}_k$.

Do for $k=1,2,\ldots,K$

Initialize the instance weights to be uniform when new data arrive $w_1^k(i) = 1/m_k, i = 1, \cdots, m_k$

If $k \neq 1$, **Set** t=0 and **Go to** step 5 of the inner **Do loop** to adjust initialization weights
Do for $t=1,2,\ldots, T_k$:

1. Set $\boldsymbol{D}_t^k = \boldsymbol{w}_t^k \Big/ \sum_{i=1}^{m_k} w_t^k(i)$ so that $\boldsymbol{D}_t^k$ is a distribution.
2. Call **BaseClassifier**, providing it with $TR_t^k \subset S_k$, drawn according to $\boldsymbol{D}_t^k$.
3. Obtain a hypothesis h_t^k:X $\rightarrow$Y$_k$, and calculate its error $\varepsilon_t^k = \sum_{i:h_t^k(\mathbf{x}_i) \neq y_i} D_t^k(i)$. If $\varepsilon_t^k > 1/2$, discard h_t^k and go to step 2. Otherwise, compute normalized error $\beta_t^k = \varepsilon_t^k / (1 - \varepsilon_t^k)$.
4. Let $CL_t^k = Y_{k(t)} \subset \{\omega_1, \cdots \omega_C\}$ be the class labels used in training h_t^k for dataset $\mathcal{D}_k$.
5. Call **DW-CAV** to obtain the composite hypothesis H_t^k.
6. Compute the error of the composite hypothesis $E_t^k = \sum_{i:H_t^k(\mathbf{x}_i) \neq y_i} D_t^k(i)$
7. Set $B_t^k = E_t^k/(1-E_t^k)$, and update the weights: $w_{t+1}^k(i) = w_t^k(i) \times \begin{cases} B_t^k, & \text{if } H_t^k(\mathbf{x}_i) = y_i \\ 1, & \text{otherwise} \end{cases}$

End
End
Call **DW-CAV** to obtain the final hypothesis, H_{final}

Fig. 2. Learn^{++}.NC algorithm.

where $Y_k(t)$ is the set of concept classes represented in the training data used to generate h_t^k.

The *DW-CAV* subroutine of Learn^{++}.NC, described below, is called to combine all hypotheses generated thus far to obtain the composite hypothesis H_t^k of the ensemble (step 5). Hence, H_t^k represents the current knowledge retained by the entire ensemble. The error E_t^k of the ensemble is then computed (on all examples of $\boldsymbol{S_k}$) and normalized to obtain $0 \leq B_t^k \leq 1$ (step 6)

$$E_t^k = \sum_{i:H_t^k(\mathbf{x}_i) \neq y_i} D_t^k(i) = \sum_{i=1}^{m_k} D_t^k(i) \left[\left[H_t^k(\mathbf{x}_i) \neq y_i \right] \right] \quad (5)$$

$$B_t^k = E_t^k / \left(1 - E_t^k\right). \quad (6)$$

The instance weights $\boldsymbol{w}_t^k$ are finally updated according to the performance of H_t^k (step 7) by

$$\begin{aligned} w_{t+1}^k(i) &= w_t^k(i) \times \left(B_t^k\right)^{1 - \left[\left[H_t^k(\mathbf{x}_i) \neq y_i\right]\right]} \\ &= w_t^k(i) \times \begin{cases} B_t^k, & \text{if } H_t^k(\mathbf{x}_i) = y_i \\ 1, & \text{otherwise.} \end{cases} \end{aligned} \quad (7)$$

Equation (7) indicates that the distribution weights of the instances correctly classified by the composite hypothesis H_t^k are reduced by a factor of $B_t^k (0 < B_t^k < 1)$, making misclassified instances more likely to be selected to the training subset of the next iteration. Defining a composite hypothesis and using its ensemble decision for distribution update constitutes a major departure point from most boosting-based approaches, such as AdaBoost. Those algorithms update their weight distribution based on the performance of a single hypothesis h_t generated during the previous iteration (an exception is arc-x4 [50], which we compare to Learn^{++}.NC in Sections IV-B–IV-D). Using the ensemble decision to choose the training subset of the next classifier, as in Learn^{++}.NC, greatly facilitates incremental learning because such a procedure forces the algorithm to focus on instances that have not been seen or properly learned by the current ensemble. Instances introduced by new data, in particular, those that belong to a new class, are precisely those instances not yet learned by the ensemble. It can be argued that AdaBoost also looks (albeit indirectly) at the ensemble decision, since, while based on a single hypothesis, the distribution update is cumulative. However, directly tying the distribution update to the ensemble decision has been found to be more effective for learning new information in our previous trials [67].

We should also note a subtle but important aspect of Learn^{++}.NC: when new data become available, Learn^{++}.NC first reinitializes distribution $\boldsymbol{D_1^{k+1}}$ to be uniform, and then further adjusts this distribution based on the performance of the current ensemble on the new data. Hence, the algorithm focuses on that portion of the new data space that has not already been learned by previous classifiers.

C. Dynamically Weighted Consult and Vote

Learn^{++}.NC uses a new voting procedure DW-CAV, where classifiers examine each other's decisions, cross reference those decisions with the list of class labels on which each was trained, and dynamically adjust their voting weight for each instance (Figs. 4 and 5). The inputs to DW-CAV are as follows: 1) the

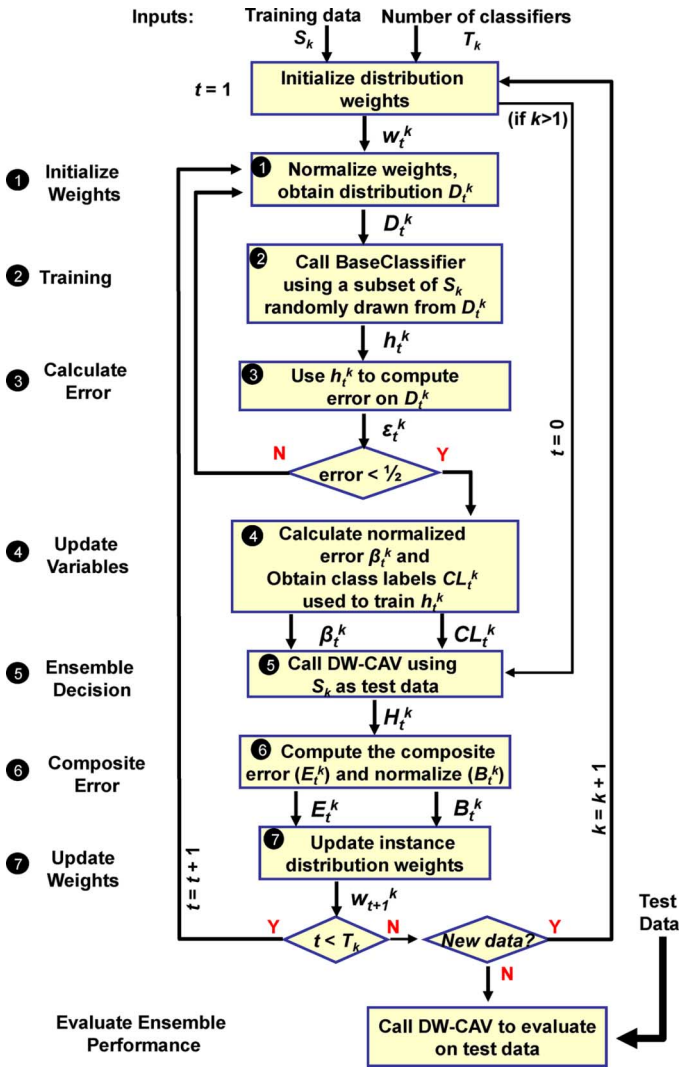

Fig. 3. Block diagram of the Learn^{++}.NC algorithm.

data points to be classified; 2) classifiers h_t^k and their normalized errors β_t^k; and 3) the vector CL_t^k containing the class labels on which h_t^k have been trained. Voting weights are initialized according to (8). Each classifier first receives a standard static weight that is inversely proportional to its normalized error β_t^k: classifiers that performed well on their training data are given higher voting weights. Note that these initial weights are independent of the instance $\mathbf{x}_i$

$$W_t^k = \log\left(1/\beta_t^k\right). \tag{8}$$

Let us represent (the ensemble of) those classifiers whose training data included class ω_C as $\boldsymbol{\mathcal{E}_C}$, those that did not as $\boldsymbol{\mathcal{E}_{\not C}}$, and let Z_c denote the sum of performance-based weights of all classifiers in $\boldsymbol{\mathcal{E}_C}$

$$Z_c = \sum_k \sum_{t:\omega_c \in CL_t^k} W_t^k. \tag{9}$$

Inputs:
- Instance $\mathbf{x}_i$ 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 $\omega_C \in \{\omega_1, \ldots, \omega_C\}$

- Normalization factor $Z_c = \sum_k \sum_{t:c\in CL_t^k} W_t^k$
- Class-specific confidence $P_c(i) = \dfrac{\sum_k \sum_{t:h_t^k(\mathbf{x}_i)=\omega_c} W_t^k}{Z_c}$

If $P_k(i) = P_l(i) = 1, k \neq l$ such that

$\mathcal{E}_k \cap \mathcal{E}_l = \varnothing \rightarrow$ Set $P_k(i) = P_l(i) = 0$

(where $\mathcal{E}_k$ is the set of classifiers that have seen class ω_k)

Update voting weights for instance $\mathbf{x}_i$

$$W_t^k(i) = W_t^k \cdot \prod_{c:\omega_c \notin CL_t^k} \left(1 - P_c(i)\right)$$

Compute final (or current composite) hypothesis

$$H_{final}(\mathbf{x}_i) = \arg\max_{\omega_c \in \Omega} \sum_k \sum_{t:h_t^k(\mathbf{x}_i)=\omega_c} W_t^k(i)$$

Fig. 4. DW-CAV algorithm for Learn^{++}.NC.

For each instance $\mathbf{x}_i$, we then define a preliminary class-specific confidence $Pc(i)$ for each class ω_C

$$P_c(i) = \frac{\sum_k \sum_{t:h_t^k(\mathbf{x}_i)=\omega_c} W_t^k}{Z_c}. \tag{10}$$

$P_c(i)$ is the ratio of total weight of all classifiers that choose class ω_c, to the total weight of all classifiers in $\boldsymbol{\mathcal{E}_C}$. The class-specific confidence $P_c(i)$ represents the collective confidence of $\boldsymbol{\mathcal{E}_C}$ classifiers in choosing class ω_c for instance $\mathbf{x}_i$. A high value of $P_c(i)$, close to 1, indicates that classifiers trained to recognize class ω_c have overwhelmingly picked class ω_c. Classifiers in $\boldsymbol{\mathcal{E}_{\not C}}$—not trained on class ω_c—then look at this decision of $\boldsymbol{\mathcal{E}_C}$ classifiers choosing class ω_c. $\boldsymbol{\mathcal{E}_{\not C}}$ classifiers then have a good reason to believe that they are probably incorrect on $\mathbf{x}_i$. Therefore, the voting weights of classifiers in $\boldsymbol{\mathcal{E}_{\not C}}$ are reduced in proportion to $P_c(i)$, the class-specific confidence of $\boldsymbol{\mathcal{E}_C}$ classifiers in choosing class ω_c

$$W_t^k(i) = W_t^k \cdot \prod_{c:\omega_c \notin CL_t^k} (1 - P_c(i)). \tag{11}$$

Note that unlike the original voting weights W_t^k, of standard weighted majority voting, the weights used by Learn^{++}.NC, are dynamically adjusted based on the classifiers' decision on current instance $\mathbf{x}_i$.

It is worth noting a pathological worst-case condition: consider an ensemble $\boldsymbol{\mathcal{E}}_A = \boldsymbol{\mathcal{E}}_1 \cup \boldsymbol{\mathcal{E}}_2$ trained on ω_1 and ω_2, and ensemble $\boldsymbol{\mathcal{E}}_B = \boldsymbol{\mathcal{E}}_3 \cup \boldsymbol{\mathcal{E}}_4$ trained on ω_3 and ω_4 instances only, with no class overlap in instances used to train two ensembles,

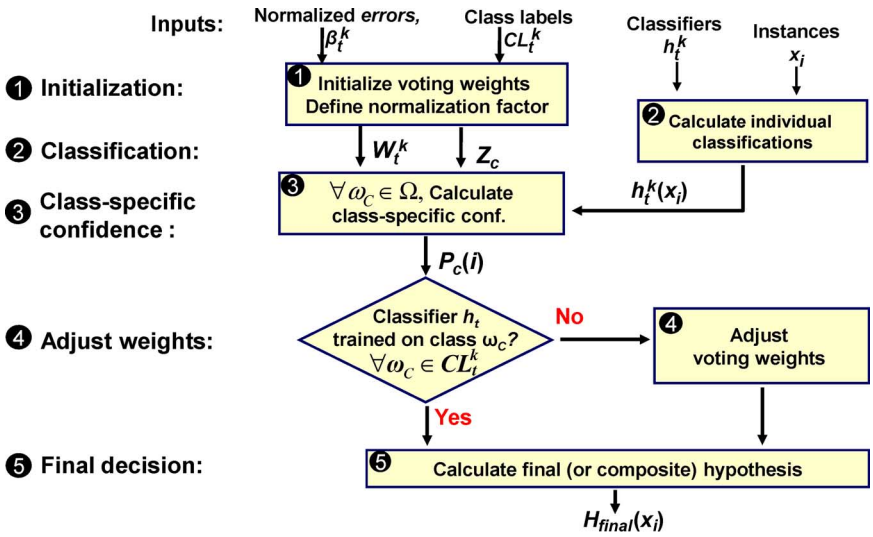

Fig. 5. Block diagram of the DW-CAV algorithm.

that is, $\boldsymbol{\mathcal{E}}_A \cap \boldsymbol{\mathcal{E}}_B = \emptyset$. Now, given a test instance $\mathbf{x}_i$ of say, class ω_1, further assume that *all* $\boldsymbol{\mathcal{E}}_A$ classifiers correctly identify $\mathbf{x}_i$ as ω_1, and *all* $\boldsymbol{\mathcal{E}}_B$ classifiers misclassify $\mathbf{x}_i$ as ω_4. In this case, both $P_1(i)$ and $P_4(i)$ will be 1, effectively setting all voting weights to zero. If for any instance this unlikely scenario takes place, the class confidences are set to zero and the algorithm returns to standard (training-performance-based) weighted majority voting.

The final output of the DW-CAV is the class for which the sum of adjusted voting weights is the largest. At any intermediate iteration t, this represents the decision of the composite hypothesis of the classifiers generated thus far; if $t = T_k$, it is the decision of the entire ensemble, hence the final hypothesis

$$H_{final}(\mathbf{x}_i) = \arg\max_{\omega_c \in \Omega} \sum_k \sum_{t:h_t^k(\mathbf{x}_i)=\omega_c} W_t^k(i). \qquad (12)$$

IV. Experiments With Learn^{++}.NC

The properties and performance of Learn^{++}.NC were observed through several experiments involving both synthetic and real-world data. Its performance was also compared to that of Learn^{++}, bagging, arc-x4, and DWM. Due to detail involved in describing each experiment, we present results on eight experiments performed on four data sets of different characteristics and difficulty.

A. Triple Spiral Synthetic Database

This data set was used to provide a visual representation of how quickly Learn^{++}.NC learns a new decision boundary. For each class $c = 1, 2, 3$, the spiral data were generated using the polar coordinates

$$\theta_c = r + 2\pi(c-1)/3, \qquad c = 1, 2, 3, \qquad r \in [0, 2\pi] \quad (13)$$

which were then converted to Cartesian coordinates to obtain 2-D feature vectors

$$X_c = r\cos\theta_c \quad Y_c = r\sin\theta_c, \qquad c = 1, 2, 3. \qquad (14)$$

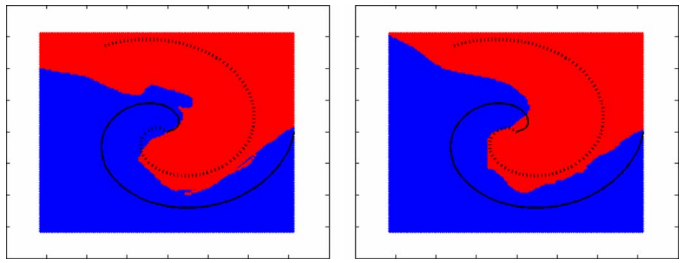

Fig. 6. Decision boundaries generated by Learn^{++} (left) and Learn^{++}.NC (right) with ten classifiers.

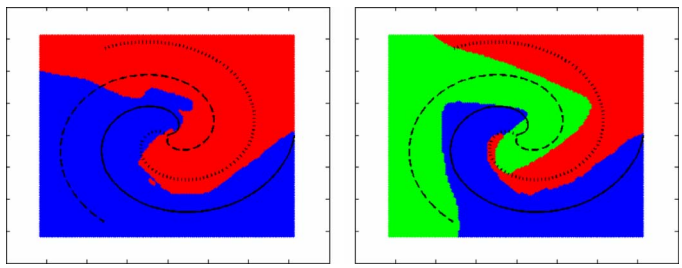

Fig. 7. Decision boundaries generated by Learn^{++} (left) and Learn^{++}.NC (right) with 11 classifiers.

Two spirals, 200 training points each, were first generated by randomly drawing r values from a uniform distribution in the $[0\ 2\pi]$ interval. An ensemble was trained on the two-spiral data, and evaluated on the entire 2-D feature space (grid of $126 \times 126 = 15\,876$) to highlight the decision boundaries. Fig. 6 illustrates the boundaries generated by Learn^{++} and Learn^{++}.NC, with ten classifiers ($2 \times 20 \times 2$ architecture MLPs, 0.05 error goal) in each ensemble. The training data are indicated as black spirals, and painted regions represent classification over the entire feature space. Both algorithms were able to learn the appropriate decision boundaries. In order to simulate incremental learning of a new class, a third spiral of 200 training points, and a fresh set of 200 points from each of the first two spirals were used to train additional classifiers. Fig. 7 shows the classification regions generated by

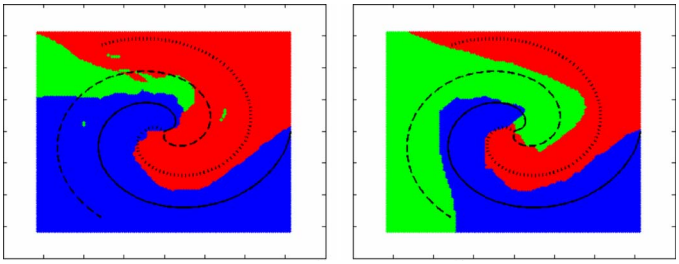

Fig. 8. Decision boundary generated by Learn^{++} (left) and Learn^{++}.NC (right) with 24 classifiers.

TABLE I
VOC DATABASE DISTRIBUTION

Class→	ET	OC	TL	TCE	XL
S_1	20	20	40	0	0
S_2	17	17	17	34	0
S_3	17	17	19	20	72
Test	6	6	11	6	8

each algorithm after the addition of just one classifier trained with instances from all three spirals. Learn^{++}.NC immediately learns the newly introduced class with only one additional classifier, whereas Learn^{++} shows no indication of a third class. In Learn^{++}, all decisions in favor of the new class cast by the 11th classifier (the only classifier thus far trained on the third spiral) are outvoted by the previous ten that had to choose one of the first two classes. In Learn^{++}.NC, however, the original ten classifiers realize that the new classifier is voting for a class they were not trained on, and withdraw their votes on all third-spiral instances. This in turn allows quick learning of the new class. Also note that Learn^{++}.NC preserves its knowledge on the remaining feature space, through the original ten classifiers.

With the addition of more classifiers, Learn^{++} eventually demonstrates its ability to learn the newly introduced spiral. Fig. 8 illustrates the decision regions generated by both algorithms with 24 classifiers: ten trained on two spirals, and 14 trained on three spirals. Note that Learn^{++} begins learning the new class at the boundary of the previously learned classes. This makes intuitive sense: individual classifiers are more likely to disagree on instances that are near the decision boundary, allowing the new classifiers avoid being outvoted by the existing classifiers in a region where they disagree with each other. We also note that Learn^{++}.NC appears to resist overfitting, despite the addition of several classifiers. A movie showing how decision boundaries change with each added classifier can be found online [68].

B. Volatile Organic Compound Recognition Database

The volatile organic compound (VOC) database is generated from a challenging real-world problem of identifying one of five VOCs based on the responses of six chemical sensors [69]. The individual VOCs were ethanol (ET), octane (OC), toluene (TL), tricholorethylene (TCE), and xylene (XL). The data set was divided into four disjoint subsets: three training subsets $\boldsymbol{S_1} \sim \boldsymbol{S_3}$ and a test set. The performance on this data was then compared to that of Learn^{++}, bagging, arc-x4, and DWM on two separate experiments, which used two separate data distributions, shown

TABLE II
VOC DATABASE DISTRIBUTION WITH VALIDATION SET

Class→	ET	OC	TL	TCE	XL
S_1	20	20	40	0	0
S_2	11	11	25	32	0
S_3	11	11	10	10	56
Valid.	12	12	24	12	16
Test	6	6	11	6	8

TABLE III
VOC DATA CLASS-SPECIFIC AND OVERALL GENERALIZATION PERFORMANCES (FIXED ENSEMBLE SIZE)

	Class→	ET	OC	TL	TCE	XL	Gen. Perf.
Learn^{++}	TS_1	88%	91%	89%	0%	0%	55.6%±0.8%
	TS_2	87%	92%	88%	34%	0%	60.9%±1.1%
	TS_3	89%	94%	84%	67%	0%	**65.5%±1.1%**
Bagging	TS_1	84%	96%	87%	0%	0%	54.8%±0.8%
	TS_2	86%	94%	84%	66%	0%	64.8%±1.1%
	TS_3	87%	96%	81%	85%	0%	**67.5%±1.0%**
ARC - X4	TS_1	88%	96%	87%	0%	0%	55.7%±0.8%
	TS_2	88%	95%	83%	76%	0%	66.7%±1.0%
	TS_3	89%	93%	81%	92%	0%	**68.4%±0.9%**
DWM - NB[1]	TS_1	86%	89%	83%	0%	0%	52.9%±0.8%
	TS_2	86%	83%	68%	76%	0%	60.0%±1.4%
	TS_3	89%	87%	56%	77%	88%	**76.6%±1.6%**
Learn^{++}.NC	TS_1	89%	88%	90%	0%	0%	55.3%±0.9%
	TS_2	85%	90%	79%	95%	0%	67.2%±1.0%
	TS_3	89%	93%	58%	90%	97%	**82.4%±1.0%**

[1] All algorithms use 30 classifiers, except DWM-NB, which uses 69.

in Tables I and II. In both cases, the distribution of the data was deliberately made challenging by biasing towards the new class instances.

In our first experiment, all algorithms were allowed to create a fixed number of (ten) classifiers for each data set presented, for a total of 30 MLP-type classifiers ($6 \times 20 \times 5$ architecture, 0.05 error goal). The only exception was DWM, which automatically prunes itself to determine the ensemble size, (typically proportional to the training data size). Used with its default parameter values recommended in [61], DWM generated 69 classifiers on this database. We note that DWM needs an online base classifier (naive Bayes or decision trees is recommended in [61]), and cannot be run with batch learning algorithms.

Table III lists class-specific and overall generalization performances after each training session TS_k, where only the current $\boldsymbol{S_k}$ was used for training (with no access to previous data sets). All performances and 95% confidence intervals are obtained as averages of ten independent trials of tenfold cross validation. We make two observations from Table III. First, the generalization performances start around 55%, as they are calculated on the test data set that includes instances from all five classes, whereas the ensembles have only seen instances from three classes by the end of TS_1 (see Table I). The performances improve progressively as new data sets (and new classes) are learned by the ensembles. Second, despite the overall improvement in performance, Learn^{++} is unable to adequately learn the new classes with just ten classifiers. Its performance on the newly introduced classes is poor by the end of the training session in which these

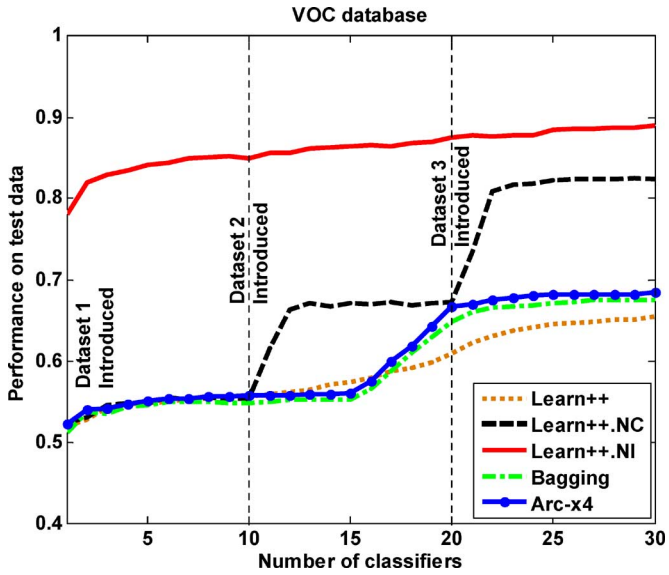

Fig. 9. Performance comparison on VOC data (Table I dist.) as a function of ensemble size.

TABLE IV
VOC DATA CLASS-SPECIFIC AND OVERALL GENERALIZATION PERFORMANCES (OPTIMIZED ENSEMBLE SIZE)

	Class→	ET	OC	TL	TCE	XL	Gen. Perf.
Learn++	TS_1 (4.1)	84%	92%	89%	0%	0%	55.0%±0.8%
	TS_2 (8.2)	83%	90%	81%	84%	0%	65.8%±1.1%
	TS_3 (18.7)	86%	94%	68%	84%	79%	**80.2%** ±1.5%
Bagging	TS_1 (3.0)	84%	93%	87%	0%	0%	54.4%±0.8%
	TS_2 (4.6)	84%	92%	77%	88%	0%	65.6%±1.0%
	TS_3 (8.3)	87%	94%	62%	74%	85%	**78.3%** ±1.2%
ARC - X4	TS_1 (3.2)	86%	94%	85%	0%	0%	54.5%±0.8%
	TS_2 (4.9)	84%	91%	77%	93%	0%	66.3%±0.9%
	TS_3 (8.0)	86%	91%	60%	81%	88%	**78.4%** ±1.2%
DWM - NB	TS_1 (16)	86%	89%	83%	0%	0%	52.9%±0.8%
	TS_2 (27)	86%	83%	68%	76%	0%	60.0%±1.4%
	TS_3 (26)	89%	87%	56%	77%	88%	**76.6%±1.6%**
Learn++ .NC	TS_1 (4.5)	84%	92%	88%	0%	0%	54.7%±0.8%
	TS_2 (2.8)	82%	89%	75%	95%	0%	65.3%±1.0%
	TS_3 (4.0)	84%	92%	52%	78%	94%	**77.0%** ±1.3%

classes are first introduced, e.g., 34% on class TCE by the end of TS_2. The performance improves, marginally, to 67% on this class by the end of TS_3, as more instances have been introduced from TCE during TS_3. However, the algorithm fails to learn the new class (XL) introduced in TS_3, as there are too many previously generated classifiers that cast incorrect votes for XL. Clearly, ten classifiers per data set were not enough for Learn^{++} ensembles to overwrite the votes on the incorrect decision of the earlier classifiers. Similar arguments can also be made for bagging and arc-x4. DWM seems to learn new classes relatively well, with a final performance of 76.4%, however, it had to use a total of 69 classifiers to do so. Learn^{++}.NC had the best final performance of 82.4% with 30 classifiers.

Fig. 9 illustrates the generalization performance of each algorithm, as new classifiers are added to the ensemble, over 30 classifiers. DWM is not included in this plot since it required 69 classifiers. Included for comparison, however, is the performance of Learn^{++} on nonincremental learning (indicated as Learn^{++}.NI in legend), obtained when all training data were provided at once. Fig. 9 indicates that the performances of all algorithms are identical when there is only one data set. However, as additional classes are introduced with new data sets, Learn^{++}.NC rapidly learns new classes, indicated by the sharp increase in performance after adding just one classifier with each new data set. All other algorithms trail Learn^{++}.NC in performance and speed of recognizing new classes. Furthermore, the final performance of Learn^{++}.NC closely approaches that of the nonincremental ensemble performance, an unofficial upper bound for performance that can be achieved with this database.

One can argue that arbitrarily fixing the number of classifiers (to ten, for example, as we did above) may unfairly restrict an algorithm, and prevent it from reaching its optimal performance. We therefore allowed the algorithms to generate as many classifiers as they need to optimize their individual performances, and then compare the number of classifiers each needs to reach its optimum performance. The optimal number of classifiers was determined by generating an excessive number of classifiers and then retaining only those required to reach the peak performance on a separate validation data set. The data distribution used for such an experiment is shown in Table II. The corresponding results are presented in Table IV, where the average number of classifiers (over $10 \times 10 = 100$ runs) determined to be optimum on validation data is shown in parentheses for each training session. Recall that DWM automatically prunes itself to its optimal size. Hence, the results given in Tables III and IV are identical for DWM.

Table IV indicates that the overall performances of all algorithms are now very close to each other at around 77%–80%, with little or no statistically significant differences among them. However, the ensembles created by Learn^{++} required 31 classifiers, bagging required 22, and arc-x4 required 16, whereas Learn^{++}.NC used only 11 classifiers to reach its final performance.

We make two additional interesting observations. First, the optimal number of classifiers for Learn^{++} $(4+8+19)$, bagging and arc-x4 (both $3+5+8$) are actually fewer than (or similar to, for Learn^{++}) the 30 used in the first experiment. It is fair to ask, then, why those ensembles were not able to learn the new classes with larger number of classifiers (i.e., 30) in the first experiment. Our second observation answers this question, and in fact, justifies the approach used by Learn^{++}.NC: the number of classifiers required to learn additional classes increase as new classes are added in all algorithms—except for Learn^{++}.NC—where it remains constant. These two phenomena are interrelated and are both due to outvoting. Specifically, in the fixed ensemble experiment, using ten classifiers in TS_1 meant that Learn^{++}, bagging and arc-x4 actually needed more than ten classifiers in the next round to overwrite the incorrect votes of the first ten just so that they can learn additional classes. Not having that many classifiers caused these classifiers to perform poorly. On the other hand, if fewer classifiers were used during TS_1, then similarly

TABLE V
IMAGE-SEGMENTATION DATA DISTRIBUTION

Cl→	B	S	F	C	W	P	G
S_1	100	100	100	0	0	0	0
S_2	100	100	100	200	200	0	0
S_3	97	97	97	97	97	297	297
Test	33	33	33	33	33	33	33

TABLE VI
IMAGE-SEGMENTATION DATA DISTRIBUTION WITH VALIDATION SET

Cl→	B	S	F	C	W	P	G
S_1	100	100	100	0	0	0	0
S_2	80	80	80	180	180	0	0
S_3	84	84	84	84	84	264	264
Valid	33	33	33	33	33	33	33
Test	33	33	33	33	33	33	33

fewer classifiers would have been required during TS_2 to outvote the classifiers of TS_1.

Furthermore, the outvoting problem can vary heavily from database to database. For example, consider a three-class database appearing in two training sessions: TS_1 with classes ω_1 and ω_2, and TS_2 with all three classes. In the best-case scenario, TS_1 will create T classifiers with relatively equal weights, and classify ω_3 instances randomly as ω_1 or ω_2. Therefore, for any ω_3 instance, (approximately) $T/2$ classifiers will choose ω_1 and $T/2$ will choose ω_2. Then, only $T/2+1$ classifiers (with weights roughly equal to the weights of previously generated classifiers) would be adequate to outvote the existing classifiers. Now, consider the (near) worst-case scenario: T classifiers are trained during TS_1, and they all classify ω_3 instances *unanimously* as ω_1 (or ω_2). If TS_2 classifiers have approximately equal weights (to those obtained in TS_1), then it would require $T+1$ classifiers to outvote the existing classifiers (as opposed to $T/2+1$ in the previous scenario). However, it could get even worse: if the two-class problem in TS_1 is perfectly separable (hence those T classifiers received very high weights), and the new three-class problem in TS_2 is inseparable (hence TS_2 classifiers receive lower weights), substantially higher number of classifiers (many more than $T+1$) will be required during TS_2 to outvote the existing classifiers. Therefore, if for any application, the first data set requires a large number of classifiers to be generated, and/or the additional data sets introduce a more difficult problem, then all consecutive training sessions will necessarily require even larger number of classifiers—hence classifier proliferation. Learn^{++}.NC completely avoids this problem.

C. Image Segmentation Database

The image segmentation database comes from the University of California at Irvine (UCI) repository [70]. The data consist of 2310 instances, with 19 attributes for segmenting the images into one of seven categories: brickface (B), sky (S), foliage (F), cement (C), window (W), path (P), and grass (G). The data set was divided into three training subsets $\boldsymbol{S_1} \sim \boldsymbol{S_3}$ and a test set. The data distribution is shown in Table V for fixed ensemble size experiment and in Table VI for optimized-ensemble-size experiment. In the first experiment, all algorithms (except DWM) generated a fixed number of ten MLPs ($6\times20\times5$ architecture, 0.05 error goal), whose test performances are shown in Fig. 10 and

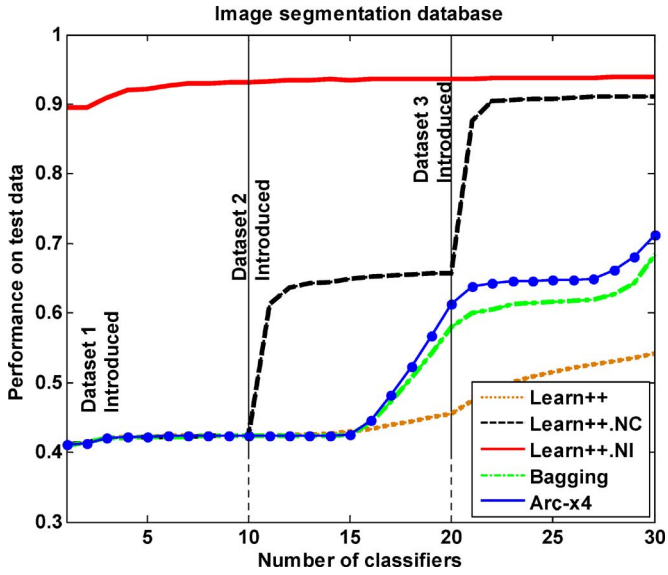

Fig. 10. Performance comparison on image data (Table V dist.) as a function of ensemble size.

Table VII. DWM, used with its default recommended parameters and naive—Bayes base classifier [61], generated a total of 352 classifiers on this database.

The performance trends are very similar to those obtained with the VOC data set. Specifically, Learn^{++}, bagging, and arc-x4 cannot handle new class information with ten classifiers per data set, and could only reach a performance of mid 50% $\sim$ 70% by the end of three training sessions. DWM was able to reach 73.9%, but only with 352 classifiers (recall that the number of classifiers is automatically determined in DWM). Learn^{++}.NC, however, it was able to learn the new classes very quickly, and reached a final generalization performance of 91% using ten classifiers per data set. When allowed to train as many classifiers to reach their optimal performances, the final performances (shown in Table VIII) were similar for all four algorithms. However, bagging and arc-x4 needed 20 classifiers, Learn^{++} needed 39, whereas Learn^{++}.NC needed only 13 to reach those optimal performances. Note that the same outvoting phenomenon discussed in detail above for the VOC data set is observed on this data set as well.

D. Optical Character Recognition Database

The optical character recognition (OCR) database consists of ten classes with 64 attributes, obtained from handwritten numeric characters 0 $\sim$ 9, digitized on an 8 $\times$ 8 grid [70]. The database was split into five disjoint sets to create four training subsets $\boldsymbol{S_1} \sim \boldsymbol{S_4}$, and one test subset. The data distribution, shown in Table IX, was designed to test the algorithms' ability to learn *two* new classes with each data set, while retaining previous knowledge. Similar to earlier experiments, all algorithms (except DWM) were first asked to create a fixed number of (five) classifiers for each data set presented, for a total of 20 MLP-type classifiers ($64\times20\times10$ architecture, 0.05 error goal). Used with its default recommended parameter values mentioned in [61], DWM generated 669 (average over 100 repetitions) classifiers on this database.

TABLE VII
IMAGE-SEGMENTATION DATA CLASS-SPECIFIC AND OVERALL PERFORMANCES (FIXED ENSEMBLE SIZE)

	Class→	B	S	F	C	W	P	G	Gen. Perf.
Learn^{++}	TS_1	98%	100%	98%	0%	0%	0%	0%	42.3%±0.9%
	TS_2	99%	100%	98%	11%	11%	0%	0%	45.6%±0.8%
	TS_3	99%	100%	98%	40%	38%	2%	3%	**54.2%** ±1.5%
Bagging	TS_1	98%	100%	98%	0%	0%	0%	0%	42.3%±0.9%
	TS_2	98%	100%	97%	60%	50%	0%	0%	57.9%±0.7%
	TS_3	98%	100%	93%	75%	67%	21%	25%	**68.4%** ±0.9%
ARC - X4	TS_1	98%	100%	99%	0%	0%	0%	0%	42.4%±0.9%
	TS_2	98%	100%	96%	69%	66%	0%	0%	61.3%±0.5%
	TS_3	97%	99%	89%	85%	82%	7%	40%	**71.1%** ±0.7%
DWM - NB[2]	TS_1	99%	100%	90%	0%	0%	0%	0%	41.3%±0.2%
	TS_2	84%	89%	24%	79%	68%	0%	0%	49.1%±2.2%
	TS_3	90%	94%	22%	77%	63%	86%	87%	**73.9%±3.4%**
Learn^{++} .NC	TS_1	99%	100%	98%	0%	0%	0%	0%	42.4%±0.9%
	TS_2	98%	100%	90%	89%	84%	0%	0%	65.7%±0.3%
	TS_3	98%	100%	93%	72%	75%	100%	99%	**91.0%** ±0.4%

[2]All algorithms use 30 classifiers, except DWM-NB, which uses 352.

TABLE VIII
IMAGE-SEGMENTATION DATA CLASS-SPECIFIC AND OVERALL PERFORMANCES (OPTIMIZED ENSEMBLE SIZE)

	Class→	B	S	F	C	W	P	G	Gen. Perf.
Learn^{++}	TS_1(4.0)	97%	100%	98%	0%	0%	0%	0%	42.2%±1.0%
	TS_2(10.9)	98%	100%	96%	71%	65%	0%	0%	61.4%±1.0%
	TS_3(23.6)	99%	100%	95%	78%	69%	92%	75%	**86.8%** ±1.7%
Bagging	TS_1(4.0)	98%	100%	97%	0%	0%	0%	0%	42.1%±1.3%
	TS_2(6.7)	97%	100%	89%	83%	76%	0%	0%	63.5%±0.4%
	TS_3(9.4)	98%	99%	89%	73%	70%	99%	96%	**89.2%** ±0.4%
ARC - X4	TS_1(4.0)	98%	100%	97%	0%	0%	0%	0%	42.1%±1.1%
	TS_2(6.7)	97%	99%	85%	89%	85%	0%	0%	65.1%±0.3%
	TS_3(8.9)	97%	98%	89%	75%	80%	99%	99%	**91.0%** ±0.5%
DWM - NB	TS_1(18)	99%	100%	90%	0%	0%	0%	0%	41.3%±0.2%
	TS_2(170)	84%	89%	24%	79%	68%	0%	0%	49.1%±2.2%
	TS_3(164)	90%	94%	22%	77%	63%	86%	87%	**73.9%** ±3.4%
Learn^{++} .NC	TS_1(4.0)	99%	100%	98%	0%	0%	0%	0%	42.1%±1.2%
	TS_2(4.5)	98%	100%	90%	89%	84%	0%	0%	64.8%±0.3%
	TS_3(4.5)	98%	100%	93%	72%	75%	100%	99%	**89.4%** ±0.4%

TABLE IX
OCR DATABASE DISTRIBUTION—TWO NEW CLASSES ARE INTRODUCED WITH EACH DATA SET

Class→	0	1	2	3	4	5	6	7	8	9
S_1	124	129	124	129	0	0	0	0	0	0
S_2	124	128	124	128	252	247	0	0	0	0
S_3	124	128	124	128	126	124	371	378	0	0
S_4	123	128	123	128	126	124	124	126	495	504
Test	55	57	55	57	56	55	55	56	55	56

Results are shown in Table X and Fig. 11. All algorithms start around 40% (four out of ten classes are seen in TS_1) and progressively improve their performances as new data sets (and new classes) are learned by the ensembles. However, Learn^{++}, bagging, and arc-x4 could not learn the new classes with fixed number of classifiers: the performances on new classes are increasingly poorer for these algorithms in subsequent training sessions, as expected due to the outvoting problem. DWM seems to learn new classes relatively well, albeit using 669 classifiers to do so. Learn^{++}.NC had the best final performance of 92.3%, and as the class-specific performances indicate, it was able to learn all classes equally well with just five classifiers per data set. Fig. 11 provides a comparative visual summary of the test performances with respect to ensemble size, as each data set is introduced.

TABLE X
OCR DATA CLASS-SPECIFIC AND OVERALL GENERALIZATION PERFORMANCES (FIXED ENSEMBLE SIZE)

	Class→	0	1	2	3	4	5	6	7	8	9	Gen. Perf.
Learn^{++}	TS_1	99%	96%	95%	96%	0%	0%	0%	0%	0%	0%	38.7%±0.1%
	TS_2	99%	96%	96%	96%	62%	73%	0%	0%	0%	0%	52.4%±0.7%
	TS_3	99%	96%	97%	97%	89%	92%	43%	47%	0%	0%	66.0%±0.8%
	TS_4	99%	96%	97%	96%	93%	95%	89%	89%	4%	2%	**76.1%** ±0.4%
Bagging	TS_1	99%	95%	94%	95%	0%	0%	0%	0%	0%	0%	38.5%±0.1%
	TS_2	99%	96%	95%	96%	76%	84%	0%	0%	0%	0%	54.7%±0.2%
	TS_3	99%	95%	96%	96%	91%	92%	55%	61%	0%	0%	68.7%±0.3%
	TS_4	99%	95%	97%	96%	93%	95%	96%	94%	9%	4%	**77.8%** ±0.2%
ARC-X4	TS_1	99%	96%	94%	95%	0%	0%	0%	0%	0%	0%	38.5%±0.1%
	TS_2	98%	95%	94%	93%	77%	88%	0%	0%	0%	0%	54.6%±0.3%
	TS_3	97%	93%	93%	92%	93%	95%	56%	65%	0%	0%	68.5%±0.4%
	TS_4	97%	93%	94%	92%	93%	95%	97%	97%	25%	11%	**79.3%±** 0.3%
DWM-NB[3]	TS_1	98%	92%	95%	95%	0%	0%	0%	0%	0%	0%	38.2%±0.1%
	TS_2	91%	83%	94%	92%	96%	92%	0%	0%	0%	0%	54.9%±0.6%
	TS_3	90%	79%	95%	93%	95%	92%	96%	92%	0%	0%	73.3%±0.3%
	TS_4	86%	63%	90%	87%	90%	86%	91%	85%	78%	73%	**82.3%±** 0.3%
Learn^{++}.NC	TS_1	99%	95%	95%	96%	0%	0%	0%	0%	0%	0%	38.6%±0.1%
	TS_2	98%	94%	96%	95%	97%	97%	0%	0%	0%	0%	57.7%±0.1%
	TS_3	98%	92%	95%	93%	90%	93%	99%	99%	0%	0%	75.9%±0.2%
	TS_4	98%	84%	93%	90%	90%	90%	97%	94%	96%	96%	**92.3%** ±0.2%

[3]All algorithms use 20 classifiers, except DWM-NB, which uses 669.

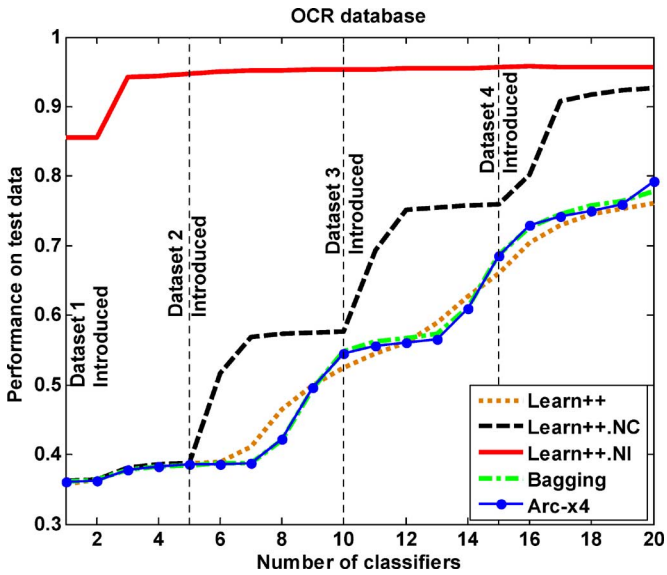

Fig. 11. Performance comparison on OCR data (Table IX distribution) as a function of ensemble size.

As before, we wanted to compare the number of classifiers each algorithm needs to reach its optimal performance. Table XI shows the data distribution for this experiment, which now includes a validation data set. The corresponding results are presented in Table XII, where the number of classifiers determined to be optimum on validation data is shown in parentheses for each training session. Table XII indicates that the overall generalization performances of all algorithms are now very close to that of each other around 90%–92%, with no statistically significant difference among them (except DWM whose performance was 82%). However, the ensembles created by Learn^{++}

TABLE XI
OCR DATABASE DISTRIBUTION, INCLUDING A VALIDATION SET, TO DETERMINE T_k FOR EACH ALGORITHM

Class→	0	1	2	3	4	5	6	7	8	9
S_1	110	114	110	114	0	0	0	0	0	0
S_2	110	114	110	114	224	220	0	0	0	0
S_3	110	114	110	114	112	110	330	336	0	0
S_4	110	114	110	114	112	110	110	112	440	448
Valid.	55	57	55	57	56	55	55	56	55	56
Test	55	57	55	57	56	55	55	56	55	56

required (on average) 56 classifiers, bagging required 23, arc-x4 required 21, DWM needed 669 (same as before due to automatic pruning), and Learn^{++}.NC used only 12 classifiers to reach its final performance.

An additional experiment was conducted for this data set (made possible by its larger number of classes), designed to determine what happens when instances of a previously seen class are not present in future data sets, and how well the algorithms retain their previous information when such information is not reinforced. Table XIII shows the data distribution used to simulate a rather extreme case of such a scenario. Classes ω_1, ω_5, and ω_9 are no longer present in $\boldsymbol{S}_2$ and $\boldsymbol{S}_3$; ω_3 and ω_7 do not appear in $\boldsymbol{S}_4$; and ω_2 and ω_6 instances do not appear in $\boldsymbol{S}_3$ and $\boldsymbol{S}_4$. As in the first experiment, a fixed number of classifiers (five) were added to the ensemble in each training session. Results are shown in Table XIV and Fig. 12. All algorithms now tend to misclassify instances from classes that are no longer included in the most recent training session. However, Learn^{++}.NC is the most robust and able to retain the most information, while learning new information fastest (hence, the best stability–plasticity performance). Learn^{++}.NC achieved a final performance of 88% followed by arc-x4 and bagging with 83%. An interesting observation was with DWM:

TABLE XII
OCR DATA CLASS-SPECIFIC AND OVERALL TEST PERFORMANCES (OPTIMIZED ENSEMBLE SIZE)

	Class→	0	1	2	3	4	5	6	7	8	9	Gen. Perf.
Learn^{++}	TS_1 (2.9)	98%	93%	92%	94%	0%	0%	0%	0%	0%	0%	37.9%±0.1%
	TS_2 (4.6)	99%	95%	95%	95%	93%	92%	0%	0%	0%	0%	56.9%±0.1%
	TS_3 (9.4)	99%	96%	96%	96%	93%	95%	92%	91%	0%	0%	75.9%±0.2%
	TS_4 (39.1)	99%	95%	97%	95%	94%	95%	98%	97%	86%	77%	**93.4%** ±0.3%
Bagging	TS_1 (2.5)	99%	93%	90%	91%	0%	0%	0%	0%	0%	0%	37.4%±0.2%
	TS_2 (4.1)	98%	93%	94%	95%	94%	94%	0%	0%	0%	0%	56.9%±0.1%
	TS_3 (5.8)	99%	93%	95%	95%	92%	94%	96%	95%	0%	0%	76.0%±0.2%
	TS_4 (10.9)	99%	88%	95%	94%	92%	93%	98%	97%	91%	88%	**93.3%** ±0.3%
ARC-X4	TS_1 (2.7)	99%	94%	90%	93%	0%	0%	0%	0%	0%	0%	37.7%±0.1%
	TS_2 (4.2)	98%	94%	93%	93%	96%	96%	0%	0%	0%	0%	57.1%±0.1%
	TS_3 (5.9)	97%	91%	93%	92%	91%	94%	98%	98%	0%	0%	75.4%±0.82%
	TS_4 (8.0)	97%	85%	92%	90%	90%	91%	98%	96%	92%	89%	**91.9%** ±0.4%
DWM-NB	TS_1 (41)	98%	92%	95%	95%	0%	0%	0%	0%	0%	0%	38.2%±0.1%
	TS_2 (82)	91%	83%	94%	92%	96%	92%	0%	0%	0%	0%	54.9%±0.6%
	TS_3 (134)	90%	79%	95%	93%	95%	92%	96%	92%	0%	0%	73.3%±0.3%
	TS_4 (412)	86%	63%	90%	87%	90%	86%	91%	85%	78%	73%	**82.3%±** 0.3%
Learn^{++}.NC	TS_1 (2.9)	98%	94%	92%	95%	0%	0%	0%	0%	0%	0%	38.1%±0.2%
	TS_2 (2.9)	97%	91%	93%	93%	97%	96%	0%	0%	0%	0%	56.8%±0.2%
	TS_3 (3.1)	96%	90%	93%	92%	88%	92%	99%	99%	0%	0%	74.9%±0.2%
	TS_4 (3.7)	97%	82%	92%	89%	90%	89%	97%	94%	96%	96%	**92.0%** ±0.3%

TABLE XIII
DATABASE DISTRIBUTION WITH REMOVAL OF CLASSES

Class→	0	1	2	3	4	5	6	7	8	9
S_1	0	257	248	0	0	248	248	0	0	252
S_2	0	0	247	257	0	0	247	252	0	0
S_3	248	0	0	256	0	0	0	252	248	0
S_4	247	256	0	0	252	247	0	0	247	252
Test	55	57	55	57	56	55	55	56	55	56

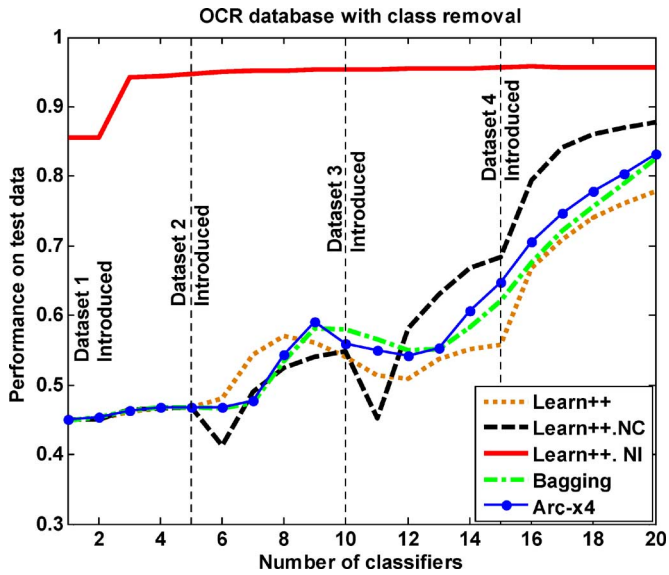

Fig. 12. Performance comparison on OCR data (Table XIII distribution, with class removal).

despite large number of classifiers, DWM was unable to retain previous knowledge once that information was no longer being reinforced.

Also note, in Fig. 12, the transient drop in performance of Learn^{++}.NC with the introduction of $\boldsymbol{S}_2$ and $\boldsymbol{S}_3$, which illustrates how the algorithm tries to balance learning new information and retaining existing information in this harsh scenario. Consider, for example, the transition from TS_2 to TS_3, where $\boldsymbol{S}_3$ introduces two new classes (ω_0, ω_8) and leaves out five of the seven classes seen thus far (about 70% of prior knowledge). The classifiers trained on $\boldsymbol{S}_3$ are bound to mislabel instances from the omitted classes. Now, if these instances are predominantly (or unanimously) mislabeled as one of the new classes (say, ω_8), the votes of older classifiers that have not seen ω_8 (denoted $\boldsymbol{\mathcal{E}}_{\not 8}$) are reduced (to prevent "correct" ω_8 decisions being outvoted). However, when there is only one new classifier during the transition, an incorrect decision of choosing ω_8 is de facto unanimous, resulting in class-specific confidence $P_8 = 1$. Then, the weights of $\boldsymbol{\mathcal{E}}_{\not 8}$ classifiers are annulled by (11)—hence the drop in ensemble performance on omitted classes. However, with additional *diverse* classifiers in TS_3, incorrect decisions are no longer unanimous $(P_8 < 1)$, allowing earlier classifiers to vote when they choose one of the omitted classes. In fact, the algorithm quickly recovers with the second classifier; and the ensemble performance improves significantly with addition of subsequent classifiers. Note that this transient condition that borders the worst-case scenario described earlier, only occurs when multiple classes are added *and* removed at the same time (resulting in little or no overlap among classes seen by different ensembles), *and* only for the first (or first few) classifiers trained after such a drastic change.

One final observation: Fig. 12 indicates that the final performances of arc-x4 and bagging are on an upward trend (after five classifiers during TS_4), whereas Learn^{++}.NC performance appears to level off. Knowing from previous results that other algorithms need additional classifiers to overcome the outvoting,

TABLE XIV
OCR DATA CLASS-SPECIFIC AND OVERALL TEST PERFORMANCES FOR TABLE XIII DISTRIBUTION

	Class→	0	1	2	3	4	5	6	7	8	9	Gen. Perf.
Learn^{++}	TS_1	0%	90%	95%	0%	0%	91%	99%	0%	0%	94%	46.8%±0.2%
	TS_2	0%	64%	97%	87%	0%	60%	100%	95%	0%	36%	54.1%±0.6%
	TS_3	65%	19%	94%	96%	0%	22%	99%	99%	60%	5%	55.8%±0.6%
	TS_4	98%	78%	87%	93%	9%	79%	98%	98%	89%	52%	**77.8%** ±0.5 %
Bagging	TS_1	0%	90%	95%	0%	0%	91%	99%	0%	0%	93%	46.7%±0.1%
	TS_2	0%	82%	97%	82%	0%	75%	100%	94%	0%	49%	57.9%±0.4%
	TS_3	67%	66%	96%	95%	0%	45%	99%	99%	45%	9%	62.0%±0.4%
	TS_4	99%	86%	90%	89%	24%	86%	98%	98%	85%	72%	**82.8%** ±0.4 %
ARC-X4	TS_1	0%	90%	96%	0%	0%	91%	99%	0%	0%	93%	46.7%±0.1%
	TS_2	0%	82%	97%	92%	0%	63%	99%	96%	0%	27%	55.9%±0.4%
	TS_3	85%	68%	93%	96%	0%	40%	99%	99%	60%	8%	64.8%±0.5%
	TS_4	99%	88%	76%	83%	37%	88%	95%	94%	90%	82%	**83.2%** ±0.4 %
DWM-NB[4]	TS_1	0%	84%	95%	0%	0%	92%	97%	0%	0%	95%	46.1%±0.1%
	TS_2	0%	0%	97%	96%	0%	0%	99%	98%	0%	0%	39.1%±0.8%
	TS_3	97%	0%	3%	95%	0%	0%	3%	98%	96%	0%	39.2%±0.6%
	TS_4	91%	79%	0%	5%	91%	88%	0%	5%	82%	87%	**52.9%±1.7** %
Learn^{++}.NC	TS_1	0%	90%	96%	0%	0%	91%	99%	0%	0%	94%	46.8%±0.1%
	TS_2	0%	78%	95%	81%	0%	70%	99%	81%	0%	43%	54.8%±0.4%
	TS_3	97%	46%	79%	82%	0%	72%	81%	82%	79%	68%	68.4%±0.6%
	TS_4	98%	86%	84%	86%	87%	87%	95%	92%	89%	73%	**87.7%** ±0.3 %

[4]All algorithms use 20 classifiers, except DWM-NB, which uses 412.

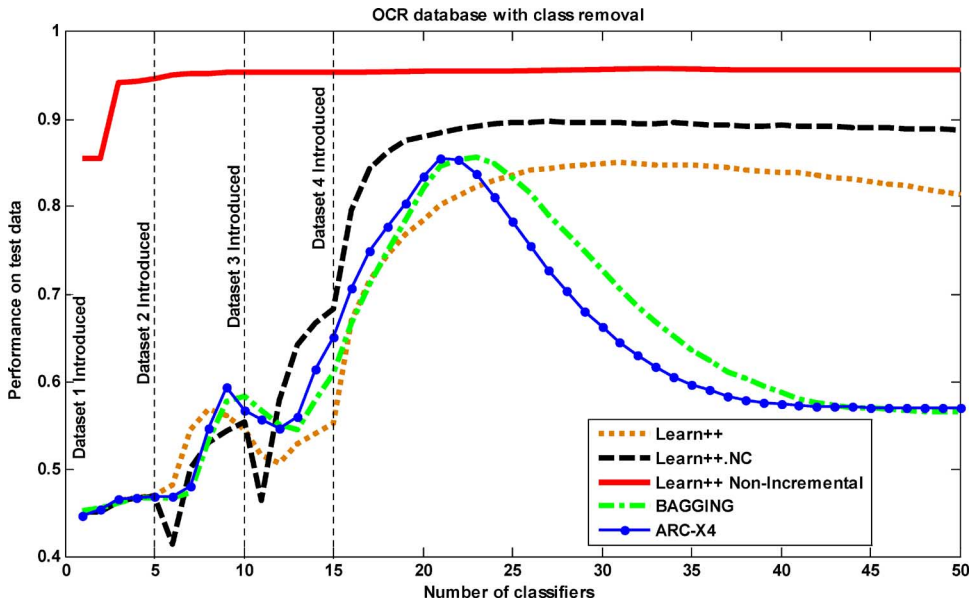

Fig. 13. Performance comparison on OCR data (Table XIII dist.) with additional classifiers in TS_4.

we allowed additional classifiers to be generated during TS_4. Fig. 13 illustrates the performances as a function of the ensemble size, which shows a sharp decline in bagging and arc-x4 performances. Ironically, trying to avoid outvoting on new class in stances by adding many new classifiers causes another form of outvoting: forgetting previous classes, primarily for bagging and arc-x4. Recall from Table XIII that TS_4 does not have any instances from four previously seen classes (ω_2, ω_3, ω_6, and ω_7). When too many classifiers are trained during TS_4, these classifiers improve their performance on the currently available classes, but start (incorrectly) outvoting previous classifiers trained on those four classes to which the new classifiers do not have access. In other words, earlier classifiers can no longer maintain enough votes to have the overall ensemble continue to perform well on those four classes absent in TS_4. Learn^{++}.NC does not have this problem, because the classifiers can determine the classes on which they are not trained, by looking at the decisions of other classifiers trained on those classes. Since Learn^{++}.NC forces classifiers to reduce their votes on instances of classes they do not recognize, the algorithm not only learns new classes very quickly, but also it never suffers from having too many classifiers.

V. Conclusion and Discussion

We introduced Learn^{++}.NC as an incremental learning algorithm for learning new classes that may later be introduced with additional data. Learn^{++}.NC creates a new ensemble of classifiers with each database that becomes available. Classifiers are then combined by using a novel combination rule, which allows classifiers to determine whether they are attempting to classify instances of a previously unseen class, and adjust their voting weights accordingly. This approach allows Learn^{++}.NC to avoid the outvoting problem (inherent in the original version of Learn^{++}, as well as other voting-based ensemble techniques used for incremental learning), and prevents proliferation of unnecessary classifiers.

While Learn^{++}.NC uses significantly fewer classifiers, there is an additional overhead in each classification due to computation of instance specific weights. However, the actual cost of this overhead is substantially less than that of training additional classifiers. It is also compensated during testing since fewer classifiers need to be evaluated and combined. Also worth noting is that the algorithm is independent of the base classifier, and can provide incremental learning ability to any supervised classifier that lacks this ability. This property is important because it allows the user to choose the particular model or classification algorithm that best matches the characteristics of the problem under consideration.

Finally, it is pertinent to discuss the type of problems for which Learn^{++}.NC is expected to perform well. In recognition of the no-free-lunch theorem, we do not claim Learn^{++}.NC to be a superior incremental learning algorithm on all applications. We simply propose it as an effective algorithm for certain types of incremental learning problems. Specifically, Learn^{++}.NC is designed for applications where new data introduce instances of previously unseen classes, and where formerly acquired knowledge regarding previously seen classes are still pertinent to the problem. Such applications are not uncommon in real world. Several applications of this type were mentioned in the Introduction, and several others were featured in the experiments. If new data do not introduce new classes, Learn^{++}.NC can still be used; though it will have no advantage over its predecessor Learn^{++} on such applications.

Learn^{++}.NC is also not intended for concept drift problems, where the data distribution of existing classes change in time (such as STAGGER [32] or SEA concepts [60]), rendering old information no longer relevant. Such applications require a built in forgetting mechanism. Another variation of Learn^{++}, called Learn^{++}.NSE (for nonstationary environments), is currently under development, which includes such a mechanism to detect and ignore knowledge that is no longer relevant [71], [72].

Theoretical analysis of Learn^{++}.NC for potential performance guarantees, its bias/variance analysis, and its evaluation on other scenarios of nonstationary learning constitute our current and future work.

Acknowledgment

The authors would like to thank the anonymous reviewers for the most thorough evaluation of this paper, and the immensely valuable suggestions, such as comparison of Learn^{++}.NC to other algorithms. They would also like to thank J. Kolter for his assistance in DWM implementation.

References

[1] E. M. Gold, "Language identification in the limit," *Inf. Control*, vol. 10, no. 5, pp. 447–474, May 1967.
[2] K. Jantke, "Types of incremental learning," in *Proc. AAAI Symp. Training Issues Incremental Learn.*, 1993, vol. SS-93-06, pp. 26–32.
[3] A. Sharma, "A note on batch and incremental learnability," *J. Comput. Syst. Sci.*, vol. 56, no. 3, pp. 272–276, Jun. 1998.
[4] S. Lange and G. Grieser, "On the power of incremental learning," *Theor. Comput. Sci.*, vol. 288, no. 2, pp. 277–307, Oct. 2002.
[5] Z. H. Zhou and Z. Q. Chen, "Hybrid decision tree," *Knowl.-Based Syst.*, vol. 15, no. 8, pp. 515–528, Nov. 2002.
[6] C. Giraud-Carrier, "A note on the utility of incremental learning," *Artif. Intell. Commun.*, vol. 13, no. 4, pp. 215–223, 2000.
[7] S. Lange and S. Zilles, "Formal models of incremental learning and their analysis," in *Proc. Int. Joint Conf. Neural Netw.*, 2003, vol. 4, pp. 2691–2696.
[8] L. M. Fu, "Incremental knowledge acquisition in supervised learning networks," *IEEE Trans. Syst. Man Cybern. A, Syst. Humans*, vol. 26, no. 6, pp. 801–809, Nov. 1996.
[9] S. Grossberg, "Nonlinear neural networks: Principles, mechanisms, and architectures," *Neural Netw.*, vol. 1, no. 1, pp. 17–61, 1988.
[10] F. H. Hamker, "Life-long learning cell structures—Continuously learning without catastrophic interference," *Neural Netw.*, vol. 14, no. 4–5, pp. 551–573, May 2001.
[11] M. A. Maloof and R. S. Michalski, "Incremental learning with partial instance memory," *Artif. Intell.*, vol. 154, no. 1–2, pp. 95–126, Apr. 2004.
[12] T. Artieres, S. Marukatat, and P. Gallinari, "Online handwritten shape recognition using segmental hidden Markov models," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 2, pp. 205–217, Feb. 2007.
[13] R. Polikar, L. Udpa, S. Udpa, and V. Honavar, "An incremental learning algorithm with confidence estimation for automated identification of NDE signals," *IEEE Trans. Ultrason. Ferroelectr. Freq. Control*, vol. 51, no. 8, pp. 990–1001, Aug. 2004.
[14] J. Macek, "Incremental learning of ensemble classifiers on ECG data," in *Proc. IEEE Symp. Comput. Based Med. Syst.*, 2005, pp. 315–320.
[15] R. Polikar, L. Udpa, S. S. Udpa, and V. Honavar, "Learn++: An incremental learning algorithm for supervised neural networks," *IEEE Trans. Syst. Man Cybern. C, Appl. Rev.*, vol. 31, no. 4, pp. 497–508, Jul. 2001.
[16] P. Winston, "Learning structural descriptions from examples," in *The Psychology of Computer Vision*, P. Winston, Ed. New York: Mc Graw Hill, 1975, pp. 157–209.
[17] N. Littlestone, "Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm," *Mach. Learn.*, vol. 2, no. 4, pp. 285–318, Apr. 1988.
[18] R. Servedio, "PAC analogues of perceptron and Winnow via boosting the margin," *Mach. Learn.*, vol. 47, no. 2–3, pp. 133–151, May 2002.
[19] S. Nieto Sanchez, E. Triantaphyllou, J. Chen, and T. W. Liao, "An incremental learning algorithm for constructing Boolean functions from positive and negative examples," *Comput. Operat. Res.*, vol. 29, no. 12, pp. 1677–1700, Oct. 2002.
[20] M. K. Warmuth, D. P. Helmbold, and S. Panizza, "Direct and indirect algorithms for online learning of disjunctions," *Theor. Comput. Sci.*, vol. 284, no. 1, pp. 109–142, 2002.
[21] J. L. Sancho, W. E. Pierson, B. Ulug, A. R. Figueiras-Vidal, and S. C. Ahalt, "Class separability estimation and incremental learning using boundary methods," *Neurocomputing*, vol. 35, no. 1–4, pp. 3–26, Nov. 2000.
[22] A. Shilton, M. Palaniswami, D. Ralph, and C. T. Ah, "Incremental training of support vector machines," *IEEE Trans. Neural Netw.*, vol. 16, no. 1, pp. 114–131, Jan. 2005.
[23] S. Ferrari and M. Jensenius, "A constrained optimization approach to preserving prior knowledge during incremental training," *IEEE Trans. Neural Netw.*, vol. 19, no. 6, pp. 996–1009, Jun. 2008.
[24] S. Ozawa, S. Pang, and N. Kasabov, "Incremental learning of chunk data for online pattern classification Systems," *IEEE Trans. Neural Netw.*, vol. 19, no. 6, pp. 1061–1074, Jun. 2008.
[25] K. Kasabov, "On-line learning, reasoning, rule extraction and aggregation in locally optimized evolving fuzzy neural networks," *Neurocomputing*, vol. 41, no. 1–4, pp. 25–45, Oct. 2001.
[26] G. G. Yen and P. Meesad, "Constructing a fuzzy rule-based systems using the ILFN network and genetic algorithm," *Int. J. Neural Syst.*, vol. 11, no. 5, pp. 427–443, 2001.

[27] L. M. Fu, H. Hui-Huang, and J. C. Principe, "Incremental backpropagation learning networks," *IEEE Trans. Neural Netw.*, vol. 7, no. 3, pp. 757–761, May 1996.
[28] K. Yamauchi, N. Yamaguchi, and N. Ishii, "Incremental learning methods with retrieving of interfered patterns," *IEEE Trans. Neural Netw.*, vol. 10, no. 6, pp. 1351–1365, Nov. 1999.
[29] D. Obradovic, "On-line training of recurrent neural networks with continuous topology adaptation," *IEEE Trans. Neural Netw.*, vol. 7, no. 1, pp. 222–228, Jan. 1996.
[30] T. Hoya, "On the capability of accommodating new classes within probabilistic neural networks," *IEEE Trans. Neural Netw.*, vol. 14, no. 2, pp. 450–453, Mar. 2003.
[31] P. Utgoff, N. C. Berkman, and J. A. Clause, "Decision tree induction based on efficient tree restructuring," *Mach. Learn.*, vol. 29, no. 1, pp. 5–44, 1997.
[32] J. C. Schlimmer and R. H. Granger, "Incremental learning from noisy data," *Mach. Learn.*, vol. 1, no. 3, pp. 317–354, Sep. 1986.
[33] G. A. Carpenter, S. Grossberg, N. Markuzon, J. H. Reynolds, and D. B. Rosen, "Fuzzy ARTMAP: A neural network architecture for incremental supervised learning of analog multidimensional maps," *IEEE Trans. Neural Netw.*, vol. 3, no. 5, pp. 698–713, Sep. 1992.
[34] J. R. Williamson, "Gaussian ARTMAP: A neural network for fast incremental learning of noisy multidimensional maps," *Neural Netw.*, vol. 9, no. 5, pp. 881–897, Jul. 1996.
[35] C. P. Lim and R. F. Harrison, "An incremental adaptive network for on-line supervised learning and probability estimation," *Neural Netw.*, vol. 10, no. 5, pp. 925–939, Jul. 1997.
[36] M. C. Su, J. Lee, and K. L. Hsieh, "A new ARTMAP-based neural network for incremental learning," *Neurocomputing*, vol. 69, no. 16–18, pp. 2284–2300, 2006.
[37] D. Heinke and F. H. Hamker, "Comparing neural networks: A benchmark on growing neural gas, growing cell structures, and fuzzy ARTMAP," *IEEE Trans. Neural Netw.*, vol. 9, no. 6, pp. 1279–1291, Nov. 1998.
[38] L. K. Hansen and P. Salamon, "Neural network ensembles," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 12, no. 10, pp. 993–1001, Oct. 1990.
[39] R. E. Schapire, "The strength of weak learnability," *Mach. Learn.*, vol. 5, no. 2, pp. 197–227, Jun. 1990.
[40] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, "Adaptive mixtures of local experts," *Neural Comput.*, vol. 3, no. 1, pp. 79–87, 1991.
[41] D. H. Wolpert, "Stacked generalization," *Neural Netw.*, vol. 5, no. 2, pp. 241–259, 1992.
[42] T. K. Ho, J. J. Hull, and S. N. Srihari, "Decision combination in multiple classifier systems," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 16, no. 1, pp. 66–75, Jan. 1994.
[43] J. Kittler, M. Hatef, R. P. W. Duin, and J. Mates, "On combining classifiers," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 20, no. 3, pp. 226–239, Mar. 1998.
[44] L. I. Kuncheva, *Combining Pattern Classifiers, Methods and Algorithms*. New York: Wiley Interscience, 2005.
[45] K. Woods, W. P. J. Kegelmeyer, and K. Bowyer, "Combination of multiple classifiers using local accuracy estimates," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 19, no. 4, pp. 405–410, Apr. 1997.
[46] E. Alpaydin and M. I. Jordan, "Local linear perceptrons for classification," *IEEE Trans. Neural Netw.*, vol. 7, no. 3, pp. 788–792, May 1996.
[47] Z. H. Zhou, J. Wu, and W. Tang, "Ensembling neural networks: Many could be better than all," *Artif. Intell.*, vol. 137, no. 1–2, pp. 239–263, May 2002.
[48] L. Breiman, "Bagging predictors," *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, 1996.
[49] Y. Freund and R. E. Schapire, "Decision-theoretic generalization of on-line learning and an application to boosting," *J. Comput. Syst. Sci.*, vol. 55, no. 1, pp. 119–139, 1997.
[50] L. Breiman, "Arcing classifiers," *Ann. Statist.*, vol. 26, no. 3, pp. 801–849, 1998.
[51] G. Fumera and F. Roli, "A theoretical and experimental analysis of linear combiners for multiple classifier systems," *IEEE Tran. Pattern Anal. Mach. Intell.*, vol. 27, no. 6, pp. 942–956, Jun. 2005.
[52] G. Rogova, "Combining the results of several neural network classifiers," *Neural Netw.*, vol. 7, no. 5, pp. 777–781, 1994.
[53] L. I. Kuncheva, J. C. Bezdek, and R. P. W. Duin, "Decision templates for multiple classifier fusion: An experimental comparison," *Pattern Recognit.*, vol. 34, no. 2, pp. 299–314, Feb. 2001.
[54] L. Todorovski and L. Dzeroski, "Combining classifiers with meta decision trees," *Mach. Learn.*, vol. 50, no. 223, p. 249, 2003.
[55] R. Polikar, "Ensemble based systems in decision making," *IEEE Circuits Syst. Mag.*, vol. 6, no. 3, pp. 21–45, 2006.
[56] R. Polikar, "Bootstrap—Inspired techniques in computation intelligence," *IEEE Signal Process. Mag.*, vol. 24, no. 4, pp. 59–72, Jul. 2007.
[57] A. Fern and R. Givan, "Online ensemble learning: An empirical study," *Mach. Learn.*, vol. 53, no. 1–2, pp. 71–109, 2003.
[58] N. C. Oza, "Online bagging and boosting," in *Proc. IEEE Int. Conf. Syst. Man Cybern.*, Waikoloa, HI, 2005, vol. 3, pp. 2340–2345.
[59] D. Chakraborty and N. R. Pal, "A novel training scheme for multilayered perceptrons to realize proper generalization and incremental learning," *IEEE Trans. Neural Netw.*, vol. 14, no. 1, pp. 1–14, Jan. 2003.
[60] W. N. Street and Y. Kim, "A streaming ensemble algorithm (SEA) for large-scale classification," in *Proc. 7th ACM SIGKDD Int. Conf. Knowl. Disc. Data Mining*, 2001, pp. 377–382.
[61] J. Z. Kolter and M. A. Maloof, "Dynamic weighted majority: An ensemble method for drifting concepts," *J. Mach. Learn. Res.*, vol. 8, pp. 2755–2790, 2007.
[62] R. Polikar, J. Byorick, S. Krause, A. Marino, and M. Moreton, "Learn++: A classifier independent incremental learning algorithm for supervised neural networks," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, 2002, vol. 2, pp. 1742–1747.
[63] M. Muhlbaier, A. Topalis, and R. Polikar, "Ensemble confidence estimates posterior probability," in *Proc. 6th Int. Workshop Multiple Classifier Syst.*, N. C. Oza, R. Polikar, J. Kittler, and F. Roli, Eds., 2005, vol. 3541, pp. 326–335.
[64] M. Muhlbaier, A. Topalis, and R. Polikar, "Incremental learning from unbalanced data," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, 2004, vol. 2, pp. 1057–1062.
[65] M. Muhlbaier, A. Topalis, and R. Polikar, "Learn++.MT: A new approach to incremental learning," in *Lecture Notes in Computer Science*, F. Roli, J. Kittler, and T. Windeatt, Eds. Berlin, Germany: Springer-Verlag, 2004, vol. 3077, pp. 52–61.
[66] P. J. Boland, "Majority system and the condorcet jury theorem," *Statistician*, vol. 38, no. 3, pp. 181–189, 1989.
[67] A. Gangardiwala and R. Polikar, "Dynamically weighted majority voting for incremental learning and comparison of three boosting based approaches," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, 2005, vol. 2, pp. 1131–1136.
[68] R. Polikar and M. Muhlbaier, "Additional simulation results on Learn^{++}.NC," Jul. 18, 2008 [Online]. Available: http://www.engineering.rowan.edu/~polikar/Learn++
[69] R. Polikar, R. Shinar, L. Udpa, and M. D. Porter, "Artificial intelligence methods for selection of an optimized sensor array for identification of volatile organic compounds," *Sens. Actuators B, Chem.*, vol. 80, no. 3, pp. 243–254, 2001.
[70] A. Asuncion and D. J. Newman, UCI Repository of Machine Learning Database, Irvine, CA, Jul. 18, 2008 [Online]. Available: http://www.ics.uci.edu/~mlearn/MLReporitory.html
[71] M. Muhlbaier and R. Polikar, "An ensemble approach for incremental learning in nonstationery environments," in *Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 2007, vol. 4472, pp. 490–500.
[72] M. Karnick, M. Ahiskali, M. Muhlbaier, and R. Polikar, "Learning concept drift in nonstationary environments using an ensemble of classifiers based approach," in *Proc. World Congr. Comput. Intell./Int. Joint Conf. Neural Netw.*, Hong Kong, 2008, pp. 3455–3462.

Michael D. Muhlbaier received the B.Sc. and M.Sc. degrees in electrical and computer engineering from Rowan University, Glassboro, NJ, in 2004 and 2006, respectively.

While at Rowan University, he served as the IEEE student branch chair, organizing chapter and region wide conferences and events. He is currently the President of Spaghetti Engineering, an automotive engineering company, and a consultant researcher in machine learning. His research interests include multiple classifier systems, concept drift, and incremental learning, on which he continues to collaborate with his thesis advisor, Dr. R. Polikar at Rowan University.

Apostolos Topalis received the B.Sc. degree in electrical and computer engineering and the M.Sc. degree in electrical engineering from Rowan University, Glassboro, NJ, in 2004 and 2006, respectively.

He is currently a Systems Engineer at Lockheed Martin, Moorestown, NJ. His research interests include pattern recognition, ensemble systems, and digital signal processing for various radar applications.

Robi Polikar (S'93–M'00–SM'08) received the B.Sc. degree in electronics and communications engineering from Istanbul Technical University, Istanbul, Turkey, in 1993 and the M.Sc. and Ph.D. degrees, both comajors in electrical engineering and biomedical engineering, from Iowa State University, Ames, in 1995 and 2000, respectively.

Currently, he is an Associate Professor of Electrical and Computer Engineering at Rowan University, Glassboro, NJ. His current research interests within computational intelligence include ensemble systems, incremental and nonstationary learning, and various applications of pattern recognition in biomedical engineering. His current work is funded primarily through NSF's CAREER program and NIH's Collaborative Research in Computational Neuroscience program.

Dr. Polikar is a member of the American Society for Engineering Education (ASEE), Tau Beta Pi, and Eta Kappa Nu.