Incremental Learning of New Classes from Unbalanced Data

Gregory Ditzler, Gail Rosen, and Robi Polikar

***Abstract*—Multiple classifier systems tend to suffer from out-voting when new concept classes need to be learned incrementally. Out-voting is primarily due to existing classifiers being unable to recognize the new class until there is a sufficient number of new classifiers that can influence the ensemble decision. This problem of learning new classes was explicitly addressed in Learn^{++}.NC, our previous work, where ensemble members dynamically adjust their own weights by consulting with each other based on their individual and collective confidence in classifying each concept class. Learn^{++}.NC works remarkably well for learning new concept classes while requiring few ensemble members to do so. Learn^{++}.NC cannot cope with the class imbalance problem, however, as it was not designed to do so. Yet, class imbalance is a common and important problem in machine learning, made even more challenging in an incremental learning setting. In this paper, we extend Learn^{++}.NC so that it can incrementally learn new concept classes even if their instances are drawn from severely imbalanced class distributions. We show that the proposed algorithm is quite robust compared to other state-of-the-art algorithms.**

***Keywords*-multiple classifier systems; unbalanced data; incremental learning;**

I. Introduction

Learning from large volumes of data over time, including streaming data, is typically handled by incremental or online learning algorithms. A learning algorithm is incremental if, for a sequence of training data sets (or instances), it produces a sequence of hypotheses, each describing all data 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 [1], which then raises the so-called stability–plasticity dilemma [2]. 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. Multiple classifier systems (MCS) provide a good balance between the two seemingly opposing goals, by simultaneously increasing the memory (to aid stability) and learning capacity (to aid plasticity) of the learning algorithm: adding new classifiers to the ensemble help learn the new data, while retaining previous classifiers maintain existing knowledge. However, such an approach has its own shortcomings: there will always be classifiers trained on only a subset of the concept classes, hence guaranteed to misclassify instances from other classes on which they were not trained. This is a problem, because classifiers not trained on a particular concept class can out-vote those that were trained on that class. Of course, classifiers not trained on a particular class should ideally not vote for instances of that class, but that requirement creates a paradox: the whole purpose of the classifier is to identify the class of a previously unknown instance. If we knew to which class an instance belongs – so that we prevent classifiers voting on instances whose classes they are not trained – we would not need a classifier in the first place. Learn^{++}.NC (*New Classes*) explicitly addresses out-voting problem in ensembles [3], by allowing a classifier not trained on a novel class to reduce or withdraw its vote on an instance, if other member classifiers trained on the novel class overwhelmingly choose that class. This voting procedure is referred to as Dynamically-Weighted Consult and Vote (DW-CAV).

As an example, consider an incremental learning algorithm that builds classifiers from animal image databases. Let the first database only contains images of cats and dogs. Therefore, the classifier trained on the first database can only accurately predict on images of cats and dogs. Later, a new database becomes available with cats, dogs, deer, and bears. Unfortunately, the classifier(s) trained on the previous database will be unable to correctly identify images of deer and bears (assuming the classifier is not updated). In fact, the classifier will label all images of deer and bears as cats and dogs, because those are the only concept classes known to the classifier. DW-CAV presents an elegant solution to combining classifiers that have been trained on a subset of the concept classes within the overall context of the learning problem.

Incremental learning of new classes becomes even more challenging if new classes are also unbalanced. Unbalanced (or, also referred to as "imbalanced") data occurs when one or more classes are heavily under-represented in the training data, a common occurrence in real world machine learning applications [4]. Unbalanced data are usually handled by over sampling the minority class or under sampling the majority class data, the most successful implementation of which is the Synthetic Minority Oversampling Technique (SMOTE) [5]. Ensemble-based algorithms, in particular variations of AdaBoost, have also been used in learning class imbalance through cost-sensitive measures or appropriate (over or under) sampling mechanisms for choosing training data. SMOTE-Boost [6]; AdaCost [7]; DataBoost [8] and RAMOBoost [9] are examples of such approaches. While all of the aforementioned algorithms are able to address unbalanced data (with varying levels of success), none of them are well suited

G. Ditzler and G. Rosen are with the Dept. of Electrical & Computer Engineering at Drexel University, Philadelphia, PA. They are supported by the National Science Foundation (NSF) CAREER award #0845827, NSF Award #1120622, and the Department of Energy Award #SC004335. Author email: gregory.ditzler@gmail.com, gailr@ece.drexel.edu

R. Polikar is with the Dept. of Electrical & Computer Engineering at Rowan University. He is supported by the NSF under Grant No: ECCS-0926159. Author email: polikar@rowan.edu

for incremental learning, let alone incremental learning of new classes. Given that there are successful algorithms for incremental learning of new classes, such as Learn^{++}.NC, and for learning from unbalanced data, such as SMOTE, a logical approach for incremental learning of new under-represented classes is a suitable integration of such approaches. Feasibility, implementation and performance of such a suitable integration constitute the focus of this work.

We previously explored a simpler version for incremental learning from imbalanced data [10]; however, that approach assumed that certain classes are only underrepresented in the current training batch, but themselves were not actually a minority class. That is, the class imbalance is local and not global. In this work, we address the more general, and the more practical problem of global class imbalance for the incremental learning of new classes.

The primary contribution of this work is a hybrid algorithm, Learn^{++}.NCS (*New Classes with Sampling*) that uses a SMOTE wrapper to facilitate learning of a minority classes even when new/old classes are introduced/removed intermittently during an experiment. The wrapper can be extended to learning algorithms other than Learn^{++}.NC as demonstrated in Section III.

II. Approach

Learn^{++}.NCS employs Learn^{++}.NC for the incremental learning of new classes along with a SMOTE-based sampling wrapper to better learn a minority class.

A. Learn^{++}.NCS

Learn^{++}.NCS, whose pseudo code is shown in Fig. 1, is an incremental learning algorithm that avoids the classifier out-voting suffered by other ensemble based approaches [3]. Learn^{++}.NCS receives a series of k batches of data (presented at subsequent time stamps), where $\mathcal{D}_k$ contains training instances $(\mathbf{x}_i, y_i)$ in each batch. In an incremental learning setting, not all classes may be represented in each batch at time k. For example, new classes may be added in time, whereas some of the old ones may not be updated. Learn^{++}.NCS generates T_k classifiers for each database, using *BaseClassifier*, which can be any supervised learner. Learn^{++}.NCS begins by initializing the distribution of instance weights, D_t^k in Line 5 (of the inner loop), to be uniform, unless there is prior knowledge (i.e., $k > 1$ and an ensemble is already available). A subset of $\mathcal{D}_k$ is sampled with replacement according to the probability distribution given by D_t^k. Refer to this data sample as $\mathcal{D}_t^k$.

The SMOTE wrapper, discussed in more detail in the next section, is applied to $\mathcal{D}_t^k$ to select a suitable set of SMOTE parameters to generate strategically selected synthetic data (line 6) for the minority class. The union of the synthetic data and $\mathcal{D}_t^k$ is denoted as $\mathcal{H}_t^k$, which is used by BaseClassifier to generate a new classifier (hypothesis) h_t^k in line 7. The error of this hypothesis, ϵ_t^k, with respect to the probability distribution D_t^k is then calculated (line 8), where $[\![\cdot]\!]$ is the indicator function. If the error on D_t^k is greater than 1/2, a new classifier is generated otherwise the error is normalized in line 9. The threshold for $1/2$ assures that the classifier will be performing better than random chance, and in many situations with a large number of classes, the classifier would perform much better than random chance. Since h_t^k is not trained on every class, we store the classes on which h_t^k is trained as Ω_t^k.

Unlike AdaBoost and other algorithms based on AdaBoost, Learn^{++}.NCS cumulatively combines the ensembles, and not the individual classifiers. To do so, Learn^{++}.NCS calls Dynamically Weighted Consult–and–Vote (DW-CAV) where voting weights of the classifiers are dynamically adjusted for each instance based on the collective decisions of all current ensemble members, and the classes on which each is trained

Input: Batch data: $\mathcal{D}_k = \{(\mathbf{x}_i, y_i)\}\ \forall i \in [m_k]$
T_k: Number of classifiers generated on the th data set
BaseClassifier: weak classification algorithm
Initialize: $D_t^k(i) = 1/m^k$

1: **for** $k = 1, \ldots, K$ **do**
2: Go to Step 5 to adjust instance weights
3: If $k \neq 1$ then set $t = 0$ and go to line 11
4: **for** $t = 1, \ldots, T_k$ **do**
5: Normalize instance weights $D_t^k(i)$ using,

$$D_t^k(i) = \frac{w_t^k(i)}{\sum_{j=1}^{m_k} w_t^k(j)}$$

6: Obtain $\mathcal{H}_t^k$ by calling SMOTE wrapper with $\mathcal{D}_t^k$ and $\mathcal{D}_k$
7: Call *BaseClassifier* with $\mathcal{H}_t^k$
8: Obtain a hypothesis $h_t^k : \mathcal{X} \mapsto \Omega$ and a pseudo error of the hypothesis

$$\epsilon_t^k = \sum_{i=1}^{m_k} D_t^k(i)[\![h_t^k(x_i) \neq y_i]\!]$$

if $\epsilon_t^k > \frac{1}{2}$, generate a new hypothesis
9: Set $\beta_t^k = \epsilon_t^k/(1 - \epsilon_t^k)$
10: Let Ω_t^k be the classes on which h_t^k was trained
11: Call DW–CAV with $\mathcal{D}_k$ to obtain a composite hypothesis H_t^k
12: Compute pseudo error of the composite hypothesis

$$E_t^k = \sum_{i=1}^{m_k} D_t^k(i)[\![H_t^k(x_i) \neq y_i]\!]$$

13: Set $B_t^k = E_t^k/(1 - E_t^k)$
14: Update instance weights

$$w_t^k(i) = w_t^k(i) B^{1-[\![H_t^k(x_i) \neq y_i]\!]}$$

15: **end for**
16: **end for**

Fig. 1. Learn^{++}.NCS pseudo code

Input: Instance $\mathbf{x}$, Classifiers, h_t^k, along with training information in Ω_t^k, and normalized pseudo error β_t^k

Initialize: $W_t^k = \log(1/\beta_t^k)$

1) Compute a confidence factor Z_c

$$Z_c = \sum_k \sum_{t:\omega_c \in \Omega_t^k} W_t^k$$

2) Compute class specific confidence

$$P_c(i) = \frac{1}{Z_c}\left(\sum_k \sum_{t:h_t^k(\mathbf{x})\in\Omega_t^k} W_k^t\right)$$

3) If $P_j(i) = P_l(i) = 1$ $(j \neq l)$ such that $\mathcal{E}_j \cup \mathcal{E}_l = \emptyset$ then set $P_j(i) = P_l(i) = 0$ (where $\mathcal{E}_j$ is the set of classifiers whose training data included ω_c)

4) Update classifier voting weights for $\mathbf{x}$

$$\hat{W}_t^k = W_t^k \prod_{c:\omega_c \notin \Omega_t^k} (1 - P_c(i))$$

5) Compute final (or current) hypothesis for $\mathbf{x}$

$$H_f(\mathbf{x}) = \arg\max_{y\in\Omega} \sum_k \sum_{t:h_t^k(\mathbf{x})=y} \hat{W}_t^k$$

Fig. 2. DW-CAV pseudo code

[3]. DW-CAV, whose pseudo-code is shown in Fig. 2, first initializes classifier-voting weights (W_t^k), based on the log normalized classifier errors on their own training data. A normalization constant, Z_c, is computed for each class c, which is the sum of the initial weights of the classifiers that were trained on class c. Then, a preliminary confidence, $P_c(i)$, is computed for each instance $\mathbf{x}_i$, and each class ω_c, which is given by,

$$P_c(i) = \frac{1}{Z_c}\left(\sum_k \sum_{t:h_t^k(\mathbf{x})\in\Omega_t^k} W_k^t\right) \tag{1}$$

Thus, $P_c(i)$ is the ratio of the weight of the classifiers that choose class c for instance i to the total weight of the classifiers trained on class c.

$P_c(i)$ can be interpreted as follows: a large value of $P_c(i)$ (e.g., close to 1) indicates an overwhelming majority of classifiers trained on class c identify $\mathbf{x}_i$ belonging to class c. If classifiers trained on class c overwhelmingly choose class c for $\mathbf{x}_i$, then classifiers not trained on c realize that they are trying to identify a class on which they were not trained, and hence reduce their voting weight in proportion to $P_c(i)$ using,

$$\hat{W}_t^k = W_t^k \prod_{c:\omega_c \notin \Omega_t^k} (1 - P_c(i)) \tag{2}$$

The current ensemble decision is obtained as the composite hypothesis $H(\mathbf{x}_i)$ using weights determined by DW-CAV using,

$$H_f(\mathbf{x}) = \arg\max_{y\in\Omega} \sum_k \sum_{t:h_t^k(\mathbf{x})=y} \hat{W}_t^k \tag{3}$$

Finally, the distribution weights are updated using,

$$w_t^k(i) = w_t^k(i) B^{1-[\![H_t^k(x_i)\neq y_i]\!]} \tag{4}$$

This process is repeated iteratively for $k \in [K]$ and $t \in [T_k]$.

Input: $\mathcal{D}_t^k$ and $\mathcal{D}_k$
$\hat{C}$: set of minority classes
α: change threshold (default=0.05)
Initialize: SMO_AMOUNT[c]=0 $\forall c \in \hat{C}$

1: **for** $c \in \hat{C}$ **do**
2: Convert $\mathcal{D}_k$ to a binary classification problem with c as the minority class. Call this data set $\mathcal{H}$.
3: Sort $\mathcal{D}_t^k$ into $\mathcal{H}_{\text{min}}$ (minority class c) and $\mathcal{H}_{\text{maj}}$ (all other classes except c)
4: Train a classifier with $\mathcal{H}_{\text{min}}$ and $\mathcal{H}_{\text{maj}}$
5: Test the classifier with $\mathcal{H}$ and compute the f-measure on the majority (f_{maj}) and minority class (f_{min}).
6: Set $\gamma = f_{\text{maj}} - \alpha f_{\text{maj}}$, $f_{\text{old}} = 0$, and SMO_AMOUNT[c]=100
7: **while** true **do**
8: Generate $\mathcal{H}'$ by calling SMOTE on $\mathcal{H}_{\text{min}}$ with SMO_AMOUNT[c]% of oversampling.
9: Generate a new classifier with $\{\mathcal{H}_{\text{min}}, \mathcal{H}_{\text{maj}}, \mathcal{H}'\}$ then test on $\mathcal{H}$ and compute f_{maj} and f_{min}.
10: **if** $(f_{\text{maj}} \geq \gamma)\&\&(f_{\text{min}} \geq f_{\text{old}})$ **then**
11: SMO_AMOUNT[c] += 100
12: **else**
13: **break**
14: **end if**
15: $f_{\text{old}} = f_{\text{min}}$
16: **end while**
17: **end for**

Fig. 3. SMOTE wrapper pseudo code

B. SMOTE Wrapper

A SMOTE wrapper is used by Learn^{++}.NCS to determine an appropriate level of synthetic minority instances (SMOTE percentage) to be used by the algorithm. A wrapper is useful for several reasons: it limits the amount of prior knowledge needed, it avoids the heuristic of using a rule of thumb to select SMOTE parameters; and it reduces the number of free parameters by performing a parameter search.

The wrapper, whose pseudo code is shown in Fig. 3, receives bootstrap data sample $\mathcal{D}_t^k$ from Learn^{++}.NCS in line 6 of Fig. 1, the training data set $\mathcal{D}_k$, the labels of the set of minority classes $\hat{C}$ and a change threshold α (default 0.05). The minority classes can be determined empirically by

Input: $\mathbf{x}_\ell$ minority data $\forall \ell \in [L]$ where L is the cardinality of the minority data
SMO: Amount of SMOTE (%)
Initialize: $\mathcal{S} = \emptyset$
for $\ell = 1, \ldots, L$ **do**
  Find k nearest neighbors of $\mathbf{x}_\ell$
  $Q = \lfloor \text{SMO}/100 \rfloor$
  while $Q \neq 0$ **do**
    Randomly select one of the k neighbors, call this $\bar{\mathbf{x}}$
    Select a random number $\alpha \in [0, 1]$
    $\hat{\mathbf{x}} = \mathbf{x}_\ell + \alpha(\bar{\mathbf{x}} - \mathbf{x}_\ell)$
    Append $\hat{\mathbf{x}}$ to $\mathcal{S}$
    $Q = Q - 1$
  end while
end for
Output: Synthetic data $\mathcal{S}$

Fig. 4. SMOTE pseudo code

examining prior probabilities or by using prior knowledge (e.g., in a credit card fraud scenario the minority class will be associated with fraudulent charges). The wrapper begins by converting $\mathcal{D}_k$ into a binary classification problem, where c is the current minority class and all other classes are considered part of the majority class. Similarly, $\mathcal{D}_t^k$ is split into a majority ($\mathcal{H}_{\text{maj}}$) and minority ($\mathcal{H}_{\text{min}}$) class subsets. A hypothesis is generated by calling *BaseClassifier* on $\mathcal{D}_t^k$ so that a baseline *f*-measure for the majority class is obtained and is used to set a threshold $\gamma = (1 - \alpha) f_{\text{maj}}$, where f_{maj} (f_{min}) is the f-measure when the target is the majority (minority) class. This threshold will serve as a stopping criterion, which determines when enough synthetic data have been injected into $\mathcal{D}_t^k$ before a severe degradation of the *f*-measure occurs. The SMOTE percentage (SMO) is initialized to 100%, amounting to doubling minority class data. Next, the wrapper enters a loop, which continues until the stopping criterion is met. Inside the loop, SMOTE (refer to Fig. 4) is called with SMO and $\mathcal{H}_{\text{min}}$, returning the synthetic minority data $\mathcal{H}'$. A new base classifier is then generated with $\mathcal{H}_{\text{maj}}$, $\mathcal{H}_{\text{min}}$, and $\mathcal{H}'$ (i.e., original data plus the synthetic data generated using SMOTE). f_{maj} and f_{min} are computed for the new classifier on $\mathcal{D}_k$. If f_{min} has increased and f_{maj} has not dropped below γ, the current value of SMOTE is saved, continuing the loop; otherwise, the loop is broken and the current value of SMO is kept for class c.

Our SMOTE wrapper differs from other wrappers, such as the one presented in [11]. Specifically, the proposed wrapper: 1) is performed at each boosting iteration rather then applying the wrapper to a single classifier; 2) focuses on maximizing the *f*-measure on the majority *and* minority classes; and 3) has lower computational complexity than the methods presented in [11] because the proposed approach is not searching for an under-sampling percentage for the majority instances.

TABLE I
CLASS INTRODUCTION AND REMOVAL FOR THE OCR DATA SET

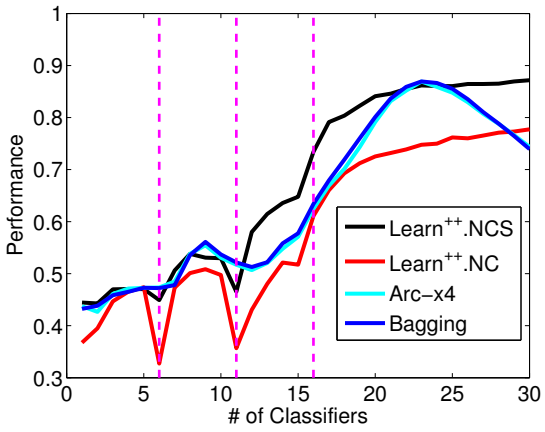
	ω_0	ω_1	ω_2	ω_3	ω_4	ω_5	ω_6	ω_7	ω_8	ω_9
$\mathcal{D}_1$	0	257	248	0	0	248	248	0	0	252
$\mathcal{D}_2$	0	0	247	257	0	0	247	252	0	0
$\mathcal{D}_3$	248	0	0	256	0	0	0	252	248	0
$\mathcal{D}_4$	247	256	0	0	257	242	0	0	247	252
$\mathcal{D}_H$	60	60	60	60	60	60	60	60	60	60

III. EXPERIMENTS

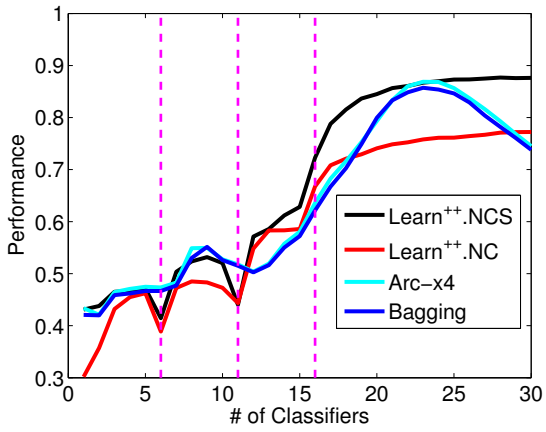
The proposed approach, denoted as Learn^{++}.NCS (which includes Learn^{++}.NC and the SMOTE wrapper), is compared against the original Learn^{++}.NC implementation [3], Bagging [12], and arc-x4 [13], all of which can learn incrementally. Note that SMOTE itself cannot learn incrementally, and therefore is not included in the comparisons. However, the SMOTE wrapper, as used in Learn^{++}.NCS is also independently applied to Bagging and arc-x4 (without using Learn^{++}.NC) in order to make a fair comparison that will allow us to determine how to properly attribute any observed difference in performance. Specifically, we want to know whether SMOTE wrapper can allow any incremental learning algorithm to accommodate unbalanced data, or Learn^{++}.NCS does really have any tangible advantage over the incremental learning algorithms. Each experiment is averaged over 15 independent trials. We report the individual class performances (i.e., recall of every class – majority and minority) on four experiments using two real-world multi-class data sets, and the overall rank of each algorithm after each new database introduction.

A. Data Set Description & Classifier Selection

The optical character recognition (OCR) database consists of ten classes with 64 attributes (reduced to 62 due to zero variance in two of the features), obtained from handwritten numeric characters (0-9) [14]. The class introduction/removal learning scenario for the OCR data set is shown in Table I. We created a particularly challenging learning scenario, where multiple new classes are introduced and removed at the same time in each data set. Each training data set is denoted as $\mathcal{D}_k$ for $k = 1, \ldots, 4$ and $\mathcal{D}_H$ is a hold-out data set for testing the classifiers. We sample $\mathcal{D}_H$ without replacement from the entire database to form the testing set, then sample the $\mathcal{D}_k$ with replacement from the remaining data. We set the number of minority class instances to 25 (about 5% of all training data during that time stamp) for each time stamp a minority class is presented for learning. For example, if ω_0 is the minority class for an experiment, then number of class ω_0 instances in the training set at each time stamp will be [0,0,25,25]. Due to space constraints, we present only a small representative portion of the different minority class combinations to demonstrate the general behavior of the algorithm. Similar results are obtained when other combinations of classes are in the minority.

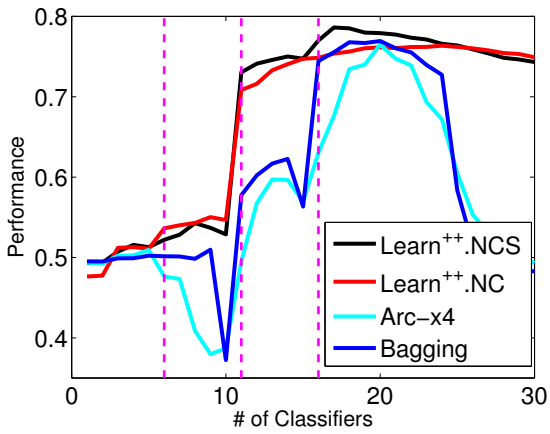


(a) OCR data with ω_0 and ω_2 as the under-represented classes

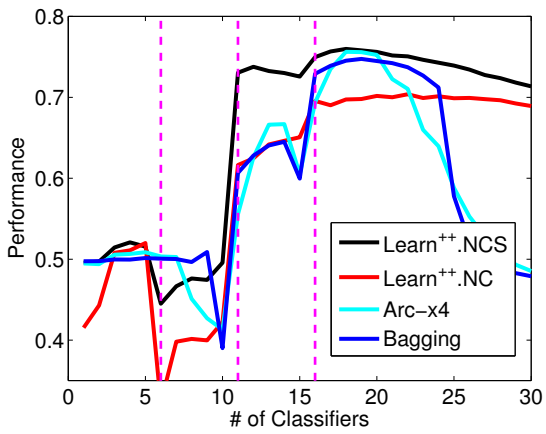


(b) OCR data with ω_1 and ω_6 as the under-represented classes

Fig. 5. Algorithm comparison on OCR database with different classes being a part of the minority class (number of classifiers vs. raw classification accuracy). The vertical lines represent the time stamps where there is a transition to a new data set.



(a) COVTYPE data with ω_1 and ω_7 as the under-represented classes



(b) COVTYPE data with ω_2 and ω_3 as the under-represented classes

Fig. 6. Algorithm comparison on COVTYPE database with different classes being a part of the minority class (number of classifiers vs. raw classification accuracy). The vertical lines represent the time stamps where there is a transition to a new data set.

TABLE II
CLASS INTRODUCTION AND REMOVAL FOR THE COVTYPE DATA SET

	ω_1	ω_2	ω_3	ω_4	ω_5	ω_6	ω_7
$\mathcal{D}_1$	5000	0	5000	5000	0	0	5000
$\mathcal{D}_2$	0	5000	5000	0	5000	5000	0
$\mathcal{D}_3$	5000	5000	0	0	5000	0	5000
$\mathcal{D}_4$	5000	5000	0	5000	0	5000	0
$\mathcal{D}_H$	100	100	100	100	100	100	100

The forest cover type (COVTYPE) contains 54 attributes and is used to determine type of forest cover (Aspen, Douglas-fir, etc) [14]. There are seven classes for this data set. The class introduction/removal learning scenario for the COVTYPE data set is presented in Table II. The same sampling techniques were applied to the COVTYPE data set as in the OCR data set. Because this is a much larger dataset, we set the number of minority class instances to 250 for any given database $\mathcal{D}_k$ in the COVTYPE data set.

A multi-layer perceptron neural network (MLPNN) was used as the base classifier for all experiments. The MLPNN is trained with a $64 \times 20 \times 10$ (OCR) or $50 \times 20 \times 7$ (COVTYPE) structure with an error goal of 0.1. Sigmoid activation functions are applied to the hidden and output layers. Learn^{++}.NC generates five classifiers on each database with the exception of $\mathcal{D}_4$ where it generates 15 classifiers. Additional classifiers are generated on the last data set to demonstrate the stability and the ability of Learn^{++}.NCS to continue learning while other approaches quickly forget previously learned knowledge with additional classifiers if all classes are not represented in the database.

For each data set tested, we experiment with two different combinations of minority classes. Note that the minority classes are chosen arbitrarily.

TABLE III

ACCURACIES FOR EACH INDIVIDUAL CLASS AND THE AVERAGE ACCURACY (LAST COLUMN) ON THE HOLD OUT TESTING DATA SET FOR THE OCR PROBLEM. ω_0 AND ω_2 ARE THE MINORITY CLASSES. THE RANK OF EACH ALGORITHM ON EACH INTRODUCTION OF A NEW DATA SET IS INDICATED IN THE LAST COLUMN.

	ω_0	ω_1	ω_2	ω_3	ω_4	ω_5	ω_6	ω_7	ω_8	ω_9	$\bar{\mu}$
						Bagging					
$\mathcal{D}_1$	0	94.7	93.4	0	0	94.3	97.9	0	0	92.6	47.3 (2.5)
$\mathcal{D}_2$	0	80.0	96.3	81.1	0	60.1	99.0	93.3	0	27.0	53.7 (1)
$\mathcal{D}_3$	61.7	51.1	93.9	97.2	0	35.2	99.0	99.4	37.8	2.22	57.8 (2)
$\mathcal{D}_4$	99.2	95.6	41.2	27.8	94.4	96.3	46.2	56.9	88.7	93.0	73.9 (4)
						Arc-x4					
$\mathcal{D}_1$	0	94.7	93.1	0	0	94.8	97.6	0	0	92.4	47.3 (2.5)
$\mathcal{D}_2$	0	79.3	95.9	85.6	0	56.1	99.3	92.6	0	21.8	53.1 (2)
$\mathcal{D}_3$	68.1	49.2	93.1	98.7	0	26.3	99.3	99.4	35.0	1.56	57.1 (3)
$\mathcal{D}_4$	99.3	96.4	35.3	25.7	95.4	97.1	48.4	57.4	93.1	95.3	74.4 (3)
						Learn^{++}.NC					
$\mathcal{D}_1$	0	96.1	88.8	0	0	95.4	98.9	0	0	94.8	47.4 (1)
$\mathcal{D}_2$	0	73.3	71.9	88.6	0	50.9	99.1	91.0	0	21.9	49.7 (4)
$\mathcal{D}_3$	40.4	34.1	43.2	89.0	0	42.3	59.0	94.7	84.4	30.1	51.7 (4)
$\mathcal{D}_4$	52.2	91.8	36.0	85.9	90.7	85.8	93.1	91.6	84.1	66.3	77.7 (2)
						Learn^{++}.NCS					
$\mathcal{D}_1$	0	93.7	90.6	0	0	96.7	98	0	0	91.4	47.0 (4)
$\mathcal{D}_2$	0	70.1	90.7	79.8	0	67.0	99.1	90.0	0	33.0	53.0 (3)
$\mathcal{D}_3$	87.1	43.3	75.3	92.7	0	56.2	81.2	94.3	85.4	32.1	64.8 (1)
$\mathcal{D}_4$	97.4	90.9	73.6	83.9	91.1	91.1	88.3	93.4	86.6	75.3	87.2 (1)

TABLE IV

ACCURACIES FOR EACH INDIVIDUAL CLASS AND THE AVERAGE ACCURACY (LAST COLUMN) ON THE HOLD OUT TESTING DATA SET FOR THE OCR PROBLEM. ω_1 AND ω_6 ARE THE MINORITY CLASSES. THE RANK OF EACH ALGORITHM ON EACH INTRODUCTION OF A NEW DATA SET IS INDICATED IN THE LAST COLUMN.

	ω_0	ω_1	ω_2	ω_3	ω_4	ω_5	ω_6	ω_7	ω_8	ω_9	$\bar{\mu}$
						Bagging					
$\mathcal{D}_1$	0	87.9	97.1	0	0	92.7	96	0	0	93.0	46.7 (3)
$\mathcal{D}_2$	0	73.3	98.8	78.9	0	51.3	98.2	92.2	0	33.2	52.6 (2)
$\mathcal{D}_3$	67.2	45.3	97.6	96.7	0	28.4	98.2	98.7	37.1	3.0	57.2 (4)
$\mathcal{D}_4$	99.7	88.6	39.1	30.0	93.9	95.0	44.4	63.2	91.0	93.2	73.8 (4)
						Arc-x4					
$\mathcal{D}_1$	0	94.3	97.3	0	0	96.7	95.9	0	0	90.7	47.5 (1)
$\mathcal{D}_2$	0	73.9	99.6	86.4	0	57.2	98.7	93.8	0	17.9	52.7 (1)
$\mathcal{D}_3$	71.2	46.9	96.0	97.2	0	30.7	98.7	99.0	39.1	1.78	58.1 (3)
$\mathcal{D}_4$	99.7	87.1	37.9	29.3	94.7	97.4	47.8	62.0	93.9	95.4	74.5 (3)
						Learn^{++}.NC					
$\mathcal{D}_1$	0	77.8	98.4	0	0	97.0	93.4	0	0	95.7	46.2 (4)
$\mathcal{D}_2$	0	38.1	97.9	84.6	0	60.2	78.2	87.7	0	26.1	47.3 (4)
$\mathcal{D}_3$	91.6	13.4	88.7	87.2	0	63.1	39.2	91.2	76.6	34.8	58.6 (2)
$\mathcal{D}_4$	98.7	41.1	89.2	82.3	92.8	89.8	15.7	93.2	90.8	78.7	77.2 (2)
						Learn^{++}.NCS					
$\mathcal{D}_1$	0	89.6	97.8	0	0	93.9	95.9	0	0	94.2	47.1 (2)
$\mathcal{D}_2$	0	61.4	98.1	88.4	0	53.8	97.4	88.7	0	32.3	52.0 (3)
$\mathcal{D}_3$	89.2	35.6	83.9	90.0	0	40.1	80.9	91.8	79.7	37.2	62.8 (1)
$\mathcal{D}_4$	98.0	82.4	82.7	84.4	87.9	91.7	89.7	92.6	86.2	80.7	87.6 (1)

B. Results on the OCR Data Set

Our first OCR experiment under-samples ω_0 and ω_2 using the approach described in the previous section. Each minority class represents only $\approx$1.3% of the training data after the minority classes have been formed as described in the previous section. Table III presents the class specific performances on the OCR data set for Learn^{++}.NC, arc-x4, bagging and Learn^{++}.NCS. The $\bar{\mu}$ column is the average accuracy across all classes. The final accuracies after each batch of data are ranked (low rank $\leftrightarrow$ high accuracy). Generally, Learn^{++}.NCS, arc-x4, and bagging outperform Learn^{++}.NC on the minority class. This, of course, should come as no surprise as Learn^{++}.NC has no mechanism to learn a minority class, and the other classifiers do have such a mechanism. We observe from Fig. 5 that Learn^{++}.NCS learns the minority class faster than bagging or arc-x4 when ω_0 is introduced in $\mathcal{D}_3$ and Learn^{++}.NCS is leading by a large margin on the specific accuracy of ω_0. Also, Learn^{++}.NCS maintains a sustained average accuracy ($\bar{\mu}$) after instances from a new class are introduced, both with $\mathcal{D}_3$ and with $\mathcal{D}_4$. Also note that Learn^{++}.NC and Learn^{++}.NCS are able to maintain their classification performance as additional classifiers are added

to the ensemble as shown in Fig. 5(a), even when the new classifiers are trained on datasets that include only a subset of the classes. On the other hand, bagging and arc-x4's overall accuracy deteriorate as classifiers are added with the last data set. This result is important, as indicates that the sensitivity of Learn^{++}.NCS (and Learn^{++}.NC) to the variations in its primary free parameter (i.e., the number of classifiers generated with each batch of data) is well controlled.

The second OCR experiment uses ω_1 and ω_6 as the minority classes (note that ω_0 and ω_2 are no longer not part of the minority class for this experiment). The same class introduction/removal scheme is applied as in Table I, only now the imbalanced classes are ω_1 and ω_6 with only 25 instances in the training data sets. The class specific accuracies on ω_6 for bagging and arc-x4 incur a sudden drop when $\mathcal{D}_4$ is introduced. This suggests, again, that bagging and arc-x4 are quick to forget the minority class as it has not been present in the last two data sets.

More generally, bagging and arc-x4 will "forget" any class that has not been present in current or recent data sets as an effect of outvoting (similar results are found in [3]). A similar effect was observed with class ω_2 in the previous OCR experiment. This effect can also be observed in Fig. 5(b), which shows that after 22 classifiers, the accuracy of arc-x4 and bagging drop off as classifiers trained on $\mathcal{D}_4$ are added to the ensemble; however, Learn^{++}.NCS maintains a steady accuracy.

C. Results on Forest Covertype Data Set

We conducted two similar experiments with the COVTYPE database, introducing two new minority classes in each experiment. First, ω_1 and ω_7 were under-sampled (Table II) to create an imbalanced learning scenario, whose results are shown in Fig. 6(a) and Table V. Minority class aside, we see that arc-x4 and bagging clearly "forget" classes once a new data set is presented that does not include data from a previously seen class. While there is some inevitable drop in classification performance with the sudden introduction of new classes observed with all algorithms, Learn^{++}.NCS handles this sudden change best and does not appear to suffer from forgetting classes since it is designed to avoid such problems. For example, consider the introduction/removal of ω_6 (majority) and ω_7 (minority). The class specific accuracy of arc-x4 and bagging plummet whenever ω_6 or ω_7 are not present in the database, even though these classes were previously introduced and should have been therefore learned. In terms of the prediction on the minority class, Learn^{++}.NCS starts with reasonably good accuracy on ω_1 although class accuracy is typically lower than that of bagging; however, Learn^{++}.NCS predicts much better on ω_7 compared to all other approaches as observed in Table V. Similar results are encountered when ω_2 and ω_3 are a part of the minority population as presented in Fig. 6(b) and Table VI. From this table,we observe that Learn^{++}.NCS predicts better than other algorithms on a minority class that has not been recently presented. Similar to our observations in Fig. 5 with respect to the OCR dataset, and Fig. 6(a) shows similar and sudden deterioration in the raw classification accuracies of arc-x4 and Bagging's as classifiers are added to the ensemble. This result can be attributed to arc-x4 and bagging generating classifiers that can only accurately predict on the classes that are currently present in the most recent batch of data. From Fig. 6(a) and 6(b), we also observe that Learn^{++}.NCS learns the new data faster than arc-x4 or bagging, which in turn can be attributed to arc-x4 and bagging suffering from out-voting.

D. Significance Summary

The non-parametric rank-based Friedman's hypothesis test was applied to check whether any of the algorithms perform significantly better/worse than others (i.e., null hypothesis is that the ranks are randomly distributed). While it is important to know that there may be one algorithm that provides significantly better results, the Friedman hypothesis test does not indicate which algorithm(s) are better. Therefore, the ranks from the Friedman test were used with the Bonferroni-Dunn correction procedure, as described in [15], to determine significance between algorithms. All statistical analysis were run with a 90% confidence level. The ranks indicated in the experimental result tables are used for the statistical hypothesis testing. We found that the Friedman test rejects the null hypothesis, when tested on the second to last and last data set, and Learn^{++}.NCS provides statistically significant improvement over Bagging and arc-x4. Significance was also found after the introduction of the 3rd data set as well. We can also state, with statistical significance, that Learn^{++}.NCS improves the minority class accuracy over Learn^{++}.NC. While, Learn^{++}.NCS consistently performed better in minority class accuracy than all other algorithms, the differences were not significant.

IV. Conclusion

In this work we presented a wrapper approach for Learn^{++}.NC, which allows for the learning of new classes that are under-represented in the training data. We have confirmed previous findings in [3] and demonstrated that the wrapper method in conjunction with Learn^{++}.NC is highly robust in predicting new classes, particularly if they are under-represented in the training data. Learn^{++}.NCS has the desired property to quickly and correctly predict a minority class when a new minority class is introduced. The results show that Learn^{++}.NCS is very robust and resilient to outvoting commonly observed with classifier ensembles. Learn^{++}.NCS typically predicted on minority classes better than other ensemble based approaches that can learn incrementally such as arc-x4 and bagging, even when the minority class was not present in the most recent database. The proposed approach learns the new classes (minority or majority) not by generating a large number of classifiers to avoid outvoting, but rather a classifier consulting procedure that dynamically assigns instance specific classifier weights. Furthermore, unlike arc-x4 and bagging, Learn^{++}.NCS does not appear to overfit, and retains its performance even when additional classifiers are generated with no new data. While there is some overhead

TABLE V

ACCURACIES FOR EACH INDIVIDUAL CLASS AND THE AVERAGE ACCURACY (LAST COLUMN) ON THE HOLD OUT TESTING DATA SET FOR THE COVTYPE PROBLEM. ω_1 AND ω_7 ARE THE MINORITY CLASSES. THE RANK OF EACH ALGORITHM ON EACH INTRODUCTION OF A NEW DATA SET IS INDICATED IN THE LAST COLUMN.

	ω_1	ω_2	ω_3	ω_4	ω_5	ω_6	ω_7	$\bar{\mu}$
	Bagging							
$\mathcal{D}_1$	86.1	0	83.4	93.9	0	0	88.2	50.2 (4)
$\mathcal{D}_2$	70.9	11.7	94.7	34.9	34.7	13.5	0.20	37.2 (4)
$\mathcal{D}_3$	54.0	70.1	81.0	7.53	89.9	3.80	88.0	56.3 (4)
$\mathcal{D}_4$	67.7	80.1	2.67	95.2	2.67	89.5	0	48.3 (4)
	Arc-x4							
$\mathcal{D}_1$	87.6	0	89.3	90.1	0	0	89.6	51.0 (3)
$\mathcal{D}_2$	37.9	40.1	97.7	8.67	58.1	28.1	0.20	38.7 (3)
$\mathcal{D}_3$	52.3	73.7	85.6	0.93	89.9	7.33	88.9	57.0 (3)
$\mathcal{D}_4$	55.7	89.9	6.07	96.8	4.47	92.4	0	49.3 (3)
	Learn^{++}.NC							
$\mathcal{D}_1$	86.5	0	93.9	89.5	0	0	87.7	51.1 (2)
$\mathcal{D}_2$	35.7	44.7	88.9	73.3	65.6	63.1	11.3	54.7 (1)
$\mathcal{D}_3$	39.9	79.1	85.9	68.8	91.2	69.5	88.7	74.7 (2)
$\mathcal{D}_4$	50.7	84.7	61.4	93.7	78.2	87.4	68.3	74.9 (1)
	Learn^{++}.NCS							
$\mathcal{D}_1$	87.8	0	92.1	88.7	0	0	90.4	51.3 (1)
$\mathcal{D}_2$	40.5	37.3	87.1	65.3	62.5	66.0	11.4	52.9 (2)
$\mathcal{D}_3$	43.7	77.8	84.8	60.9	90.9	71.3	93.7	74.7 (1)
$\mathcal{D}_4$	40.0	85.1	58.5	94.3	77.1	86.3	78.9	74.3 (2)

TABLE VI

ACCURACIES FOR EACH INDIVIDUAL CLASS AND THE AVERAGE ACCURACY (LAST COLUMN) ON THE HOLD OUT TESTING DATA SET FOR THE COVTYPE PROBLEM. ω_1 AND ω_2 ARE THE MINORITY CLASSES. THE RANK OF EACH ALGORITHM ON EACH INTRODUCTION OF A NEW DATA SET IS INDICATED IN THE LAST COLUMN.

	ω_1	ω_2	ω_3	ω_4	ω_5	ω_6	ω_7	$\bar{\mu}$
	Bagging							
$\mathcal{D}_1$	87.9	0	83.3	90.5	0	0	89.5	50.2 (4)
$\mathcal{D}_2$	80.1	4.00	85.9	51.9	24.8	23.7	2.60	39.0 (4)
$\mathcal{D}_3$	77.9	41.0	73.9	30.9	91.3	13.1	91.6	60.0 (4)
$\mathcal{D}_4$	84.5	59.5	1.40	94.5	4.53	90.8	0	47.9 (4)
	Arc-x4							
$\mathcal{D}_1$	89.7	0	78.7	95.9	0	0	91.8	50.9 (3)
$\mathcal{D}_2$	62.5	10.9	81.6	44.2	35.9	52.9	1.20	41.3 (3)
$\mathcal{D}_3$	80.1	41.5	67.1	23.2	93.3	24.7	93.5	60.5 (3)
$\mathcal{D}_4$	91.4	47.1	1.53	96.7	9.13	93.9	0	48.5 (3)
	Learn^{++}.NC							
$\mathcal{D}_1$	93.4	0	82.5	95.7	0	0	92.4	52.0 (1)
$\mathcal{D}_2$	50.9	18.6	26.9	45.0	46.9	86	21.5	42.2 (2)
$\mathcal{D}_3$	83.5	28.5	24.7	43.3	92.4	87.8	95.3	65.1 (2)
$\mathcal{D}_4$	86.1	34.6	9.67	96.1	85.0	91.3	79.8	68.9 (2)
	Learn^{++}.NCS							
$\mathcal{D}_1$	93.7	0	79.4	95.6	0	0	92.2	51.6 (2)
$\mathcal{D}_2$	60.1	19.6	65.7	73.3	42.8	72.7	12.9	49.6 (1)
$\mathcal{D}_3$	86.1	27.9	61.1	70.2	90.8	77.7	94.1	72.6 (1)
$\mathcal{D}_4$	88.7	36.3	36.5	94.0	80.2	90.9	73.0	71.4 (1)

computation in determining classifier weights for DW-CAV, we argue that the overhead associated with DW-CAV is far less expensive than the overhead in generating new classifiers.

REFERENCES

[1] R. Polikar, L. Udpa, S. S. Udpa, and V. Honavar, "Learn++: an incremental learning algorithm for supervised neural networks," *IEEE Trans. on Syst., Man and Cybern.*, vol. 31, no. 4, pp. 497–508, 2001.

[2] S. Grossberg, "Nonlinear neural networks: Principles, mechanisms, and architectures," *Neural Networks*, vol. 1, no. 1, pp. 17–61, 1988.

[3] M. 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.

[4] H. He and E. Garcia, "Learning from imbalanced data," *IEEE Trans. on Data and Knowl. Disc.*, vol. 12, no. 9, pp. 1263–1284, 2009.

[5] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[6] N. V. Chawla, A. Lazarevic, L. O. Hall, and K. W. Bowyer, "SMOTEBoost: Improving prediction of the minority class in boosting," in *European Conf. on Princ. and Pract. of Knowl. Disc. in Data.*, 2003, pp. 1–10.

[7] Y. Sun, M. S. Kamel, A. K. Wong, and Y. Wang, "Cost-sensitive boosting for classification of imbalanced data," *Pattern Recognition*, vol. 12, no. 40, pp. 3358–3378, 2007.

[8] H. Guo and H. L. Viktor, "Learning from imbalanced data sets with boosting and data generation: The Databoost-IM approach," *Sigkdd Explorations*, vol. 6, no. 1, pp. 30–39, 2004.

[9] S. Chen, H. He, and E. Garcia, "RAMOBoost: Ranked minority oversampling in boosting," *IEEE Trans. on Neur. Netw.*, vol. 21, no. 10, pp. 1624–1642, 2010.

[10] G. Ditzler, M. Muhlbaier, and R. Polikar, "Incremental learning of new classes in unbalanced datasets: Learn++.UDNC," in *Multiple Classifier Systems (MCS 2010)*, ser. Lecture Notes in Computer Science, N. El Gayar et al., eds., vol. 5997, 2010, pp. 33–42.

[11] N. V. Chawla, L. O. Hall, and A. Joshi, "Wrapper-based computation and evaluation of sampling methods for imbalanced datasets," in *ACM SIGKDD Workshop on Utility-Based Data Mining*, 2005, pp. 1–10.

[12] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[13] ——, "Arcing classifiers," *The Annals of Statistics*, vol. 26, no. 3, pp. 801–824, 1998.

[14] A. Frank and A. Asuncion, "UCI machine learning repository," University of California, Irvine, School of Information and Computer Sciences, 2010. [Online]. Available: http://archive.ics.uci.edu/ml

[15] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *Journal of Machine Learning Research*, vol. 7, pp. 1–30, 2006.