A Comparison of Adaboost and Learn++ Distribution Update Rule

G.T. PRASANNA KUMAR1, K.SEKAR2, R.POORNIMA3
1,2Assoc.Prof, Research Scholar, Dept of Computer Science and Engineering,  
3Asst.Prof, Dept of Computer Science and Engineering,  
S.V.Engineering College for Women, Tirupathi-517 501, AP, INDIA  
tabithaprasanna@gmail.com, sekhar_sangam@yahoo.com, rpoornima582@gmail.com

ABSTRACT

In the AdaBoost formulation from learning new data, if instances of previously unseen instances are introduced. Actually AdaBoost do nothing. AdaBoost should work for incremental learning, but it was made more efficient by modifying the distribution update rule to make the update based on the ensemble decision, not just the previous classifier. The ability of a classifier to take on new information and classes by evolving the classifier without it having to be fully retrained is known as incremental learning. Incremental learning has been successfully applied to many classification problems, where the data is changing and is not all available at once.

Keywords: AdaBoost, Incremental learning, distribution update rule

1. INTRODUCTION

Incremental learning has been applied in different ways. The simplest of the incremental learning approaches is one of storing all the data which allows for retraining with all the data. At the other extreme is the training of the data, instance by instances, in an online learning fashion. Methods using the online learning approach for incremental learning have been implemented but have not considered all the issues of learning, particularly the learning of new classes. For a classifier to be incremental it should satisfy the following criteria:

1. “It should be able to learn additional information from new data.
2. It should not require access to the original data used to train the existing classifier.
3. It should preserve previously acquired knowledge (that is, it should not suffer from catastrophic forgetting).
4. It should be able to accommodate new classes that may be introduced with new data”.

In certain applications, it is not uncommon for the entire dataset to gradually become available in small batches over a period of time. Furthermore, datasets acquired in subsequent batches may introduce instances of new classes that were not present in previous datasets. In such settings, it is necessary to learn the novel information content in the new data, without forgetting the previously acquired knowledge, and without requiring access to previously seen data. The ability of a classifier to learn under these circumstances is usually referred to as incremental learning.

A practical approach for learning from new data involves discarding the existing classifier, and retraining a new one using all data that have been accumulated thus far. This approach, however, results in loss of all previously acquired information, a phenomenon known as catastrophic forgetting (or interfering). Not conforming to the above stated definition of incremental learning aside, this approach is desirable, if retraining is computationally or financially costly; but more importantly, it is unfeasible, if the original dataset is lost, corrupted, discarded, or otherwise unavailable. Such scenarios are not uncommon in databases of restricted or confidential access, such as in medical and military applications.

Ensemble systems have been successfully used to address this problem. The underlying idea is to generate additional ensembles of classifiers with each subsequent database that becomes available, and combine their outputs using one of the combination methods. The algorithm Learn++ and its recent variations have been shown to achieve incremental learning on a broad spectrum of applications, even when new data introduce instances of previously unseen classes. Learn++ introduces the notion of composite hypothesis—the combined ensemble generated thus far at any given iteration—and updates its distribution based on the performance of the current ensemble through the use of this composite hypothesis. Learn++ then focuses on instances that have not been properly learned by the entire ensemble. During incremental learning, previously unseen instances, particularly those coming from a new class, are bound to be misclassified by the ensemble, forcing the algorithm to focus on learning such instances introduced by the new data. Furthermore, Learn++ creates an ensemble of ensemble classifiers, one ensemble for each database. The ensembles are then combined through a modified weighted majority voting algorithm. More recently, it was shown that the algorithm also works well, even when the data distribution is unbalanced, where the number of instances in each class or database vary significantly.

The most popular and successful of all ensemble generation algorithms, AdaBoost (Adaptive Boosting) is an extension of the original boosting algorithm, that extends boosting to the multi-class problems. AdaBoost
generates an ensemble of classifiers, the training data of each is drawn from a distribution that starts uniform and iteratively changes into one that provides more weight to those instances that are misclassified. Each classifier in AdaBoost focuses increasingly on the more difficult to classify instances. The classifiers are then combined through weighted majority voting as shown in Figure 1.

Figure1 : Combining Classifiers using Weighted Majority Voting

2. **ADABOOST**

Create a discrete distribution of the training data by assigning a weight to each instance. Initially, the distribution is uniform, hence all weights are the same. Algorithm was shown in Figure 2 and explained in Figure 3.

- Draw a subset from this distribution and train a weak classifier with this dataset.
- Compute the error, \( \varepsilon \), of this classifier on its own training dataset. Make sure that this error is less than \( \frac{1}{2} \).
- Test the entire training data on this classifier:
  - If an instance \( x \) is correctly classified, reduce its weight proportional to \( \varepsilon \)
  - If it is misclassified, increase its weight proportional to \( \varepsilon \)
  - Normalize the weights such that they constitute a distribution
- Repeat until \( T \) classifiers are generated
- Combine the classifiers using weighted majority voting

```plaintext
Algorithm AdaBoost-M1
Input: Sequence of \( m \) examples \( S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\} \) with labels \( y_i \in \{1, \ldots, C\} \) drawn from a distribution \( D_0 \).
Weak learning algorithm WeakLearn, Integer \( T \) specifying number of iterations.

Initialize \( D_t(i) = \frac{1}{m} \) \( \forall i \).

Do for \( t = 1, 2, \ldots, T \)
1. Call WeakLearn, providing it with the distribution \( D_t \).
2. Get back a hypothesis \( h_t : X \rightarrow Y \)
3. Calculate the error of \( h_t \):
   \[ \varepsilon_t = \sum_{i=1}^{m} D_t(i) \]
   If \( \varepsilon_t > \frac{1}{2} \), then set \( T = T - 1 \) and abort loop.
4. Set \( \beta_t = \varepsilon_t / (1 - \varepsilon_t) \).
5. Update distribution \( D_{t+1} \):
\[
D_{t+1}(i) = \frac{D_t(i) \beta_t}{Z_t}, \quad \text{if} \quad h_t(x_i) = y_i
\]
\[
D_{t+1}(i) = \frac{D_t(i)}{Z_t}, \quad \text{otherwise}
\]
where \( Z_t = \sum_{i} D_{t+1}(i) \) is a normalization constant chosen so that \( D_{t+1} \) becomes a distribution function

Output the final hypothesis:
\[
h_{\text{final}}(x) = \arg \max_{h \in \text{hyp}} \sum_{i=1}^{m} \log \frac{1}{\beta_t}
\]
```

Figure2 : Algorithm AdaBoost

**Problem:** How many hours should our children (5-10 yrs) spend on homework?

2. **LEARN++**

The original version of Learn++ followed the AdaBoost approach in determining voting weights, which were assigned during training depending on the classifiers’ performance on their own training data. While this approach makes perfect sense when the entire data come from the same database, it does have a handicap when used in an incremental learning setting: since each classifier is trained to recognize (slightly) different portions
of the feature space, classifiers performing well on a region represented by their training data may not do so well when classifying instances coming from different regions of the space. Therefore, assigning voting weights primarily on the training performance of each classifier is suboptimal. Estimating the potential performance of a classifier on a test instance using a statistical distance metric, and assigning voting weights based on these estimates may be more optimal. AdaBoost is capable of incremental learning, albeit with a lower performance, efficiency and stability than either version of Learn++.  

For each database $D_k$, $k = 1,\ldots,K$ that becomes available, the inputs to Learn++ are: (1) labeled training data $S_k = \{(x_i, y_i)\}, i = 1,\ldots,m_k$, where $x_i$ is the training instance and $y_i$ is the correct label; (2) a weak learning algorithm $Base$ Classifier: and (3) an integer $T_k$, the total number of weak classifiers to be generated. The $Base$ Classifier can be any supervised algorithm that can obtain a minimum of 50% classification performance on training data, ensuring the classifier is relatively weak, yet reasonably strong to have a meaningful performance. Using a weak classifier has the additional advantage of rapid learning, since the time-consuming fine-tuning step, which could potentially cause overfitting, is avoided. Unless there is compelling reason to choose otherwise, the distribution weights are initialized to be uniform, so that all instances have the same probability of being selected into the first training subset. If $k > 1$ (that is, new database has been introduced), a distribution initialization sequence reinitializes the data distribution (the If block in Fig 1) based on the performance of the current ensemble on the new data. At each iteration $t$, the distribution $D_t$ is obtained by normalizing the weights $w_i$ of the instances based on their individual classification by the current ensemble (step 1).

\[
D_t = w_t \frac{\sum_{i=1}^{m} w_i (i)}{(1)
\]

The training dataset $S_k$ is divided into a training (TR$_t$) and a testing subset (TE$_t$) according to $D_t$ (step 2). Learn++ then calls $Base$ Classifier (Step 3) and trains it with TR$_t$ to generate a weak hypothesis $h_t$. The error of this hypothesis is calculated on the current training data $S_k$ by adding the distribution weights of the misclassified instances (step 4)  

\[
\varepsilon_t = \sum_{i:H_t(x_i) \neq y_i} D_t (i) = \sum_{i=1}^{m} D_t (i) [H_t(x_i) \neq y_i]  \]

Where $[\cdot]$ is 1 if the predicate is true, and 0 otherwise. If $\varepsilon > \frac{1}{2}$, current $h_t$ is deemed too weak, and is replaced with a new $h_t$ generated from a fresh set of TR$_t$ and TE$_t$. If $\varepsilon < \frac{1}{2}$, the current hypothesis is added to the previous hypotheses and all hypotheses generated during the previous $t$ iterations are then combined using the weighted majority voting to construct the composite hypothesis $H_t$ (step 5).
Figure 4: Algorithm Learn++

In original Learn++, the voting weights were calculated based on the error $\varepsilon_n$ so that hypotheses with lower error were given higher weights, resulting in classes predicted by these hypotheses to be weighted more heavily. Since the hypothesis is to be weighted more heavily, since the hypothesis weights are assigned prior to testing based on their individual training performance, this weight assignment is suboptimal. This is because, hypotheses are trained with different (and possibly overlapping) portions of the feature space, and it may not be reasonable to expect a classifier to perform well on test instances that may come from different portions of the feature space. This is not likely to be a major issue when only a single database is used (as in AdaBoost); however, it is a valid concern in an incremental learning setting. A more optimal rule would be to dynamically estimate which hypotheses are more likely to correctly classify any given instance and give them higher voting weights accordingly. Therefore, we modify the expression for composite hypotheses, representing ensemble decision, as

$$H_t = \arg \max_{y \in Y} \sum_i D W_i(x)$$

Where $D W_i(x)$ is the dynamic weight assigned to hypothesis $h_i$ for the instance $x$. As described below, dynamic weights are determined by using Mahalanobis-distance based estimated likelihood of $h_i$ to correctly classify the instance $x$. The composite error $E_t$ made be $H_t$, that is, the performance of the entire ensemble constructed thus far, is then determined by summing up the distribution weights of all instances misclassified by the ensemble (step 6).

$$E_t = \sum_{i=1}^{H_t(x_i) = y_i} D W_i(x)$$

Finally, the composite normalized error is determined as

$$B_t = E_t / (Q - E_t), \quad 0 < E_t < 1$$

and distribution weights are updated according to the ensemble performance (step 7).
This expression reduces the weights of those instances correctly classified by the composite hypothesis $H_t$ by a factor of $B_t$, while the weights of the misclassified instances are kept unchanged. At $t+1^{th}$ iteration, after normalization of the weights in step 1, the probability of choosing previously correctly classified instances for $TR_{t+1}$ is reduced, while that of misclassified instances is effectively increased. This would be a logical place to pause and point out to some of the main differences between AdaBoost and Learn++. The distribution update rule in AdaBoost is based on the performance of the previous hypothesis, which focuses the algorithm on difficult instances with respect to different sampling of a given single database, whereas that of Learn++ is based on the performance of the entire ensemble, which focuses this algorithm on instances that carry novel information with respect to consecutive databases.

This becomes particularly critical when new database introduces instances from a previously unseen class. Since none of the previous classifiers in the ensemble has seen the instances from the new class, $H_t$ will initially misclassify them, forcing the algorithm to focus on these instances that carry novel information. The procedure would not work nearly as efficiently, however, if the weight update rule were based on the performance of $h_t$ only (as AdaBoost does) instead of $H_t$. This is because the training performance of the first $h_t$ on instances from the new class is independent of that of the previously generated classifiers. Therefore, $h_t$ is likely to correctly classify new class instances that it has just seen, but only at the time they are first introduced. This would cause the algorithm to focus on other difficult to learn instances, such as outliers, rather than the instances with novel information content.

Once $T_k$ hypotheses are generated for each database $D_k$, the final hypothesis $H_{final}$ can be obtained by combining all hypotheses by dynamically weighted majority voting, choosing the class that receives the highest total vote among all hypotheses:

$$H_{final}(x) = \arg \max_y \sum_{k=1}^{K} \sum_{t \in H_k} D_{W_t}(x)$$

The intuition in using dynamically updated voting weights is as follows: if we knew which hypotheses would perform best ahead of time, we would give those hypotheses higher weights. We cannot have this information a priori, however, we can estimate which classifiers are more likely to correctly identify a given instance based on the location of that instance in the feature space with respect to the instances used to train individual classifiers. If an instance is spatially close – in a distance metric sense – to the training data used to train a classifier, then it is reasonable to expect that classifier will perform well on the given instance.

We use the class-specific Mahalanobis distance metric to compute the distance between the training data and the unknown instance for each classifier. Classifiers whose training data set are closer to the unknown instance are weighted higher. We note that previously seen data need not be stored in order to compute the desired distances, but only the means and covariance matrices of the training sets. We formalize the computation of these weights as follows:

Let us define $TR_c$ as the subset of $TR_t$, the training dataset is used during the $t^{th}$ iteration, to include only those instances that belongs to class $c$, that is:

$$TR_t = \{x_t | x_t \in TR_t \& y_t = c\} \Rightarrow TR_t = \bigcup_{c=1}^{C} TR_{tc}$$

where $C$ is the total number of classes. Class-specific Mahalanobis distance is then computed as:

$$M_{tc}(x) = (x - m_{tc})^T C_{tc}^{-1} (x - m_{tc}), c = 1, ..., C$$

where $m_{tc}$ is the mean and $C_{tc}$ is the covariance metric of $TR_{tc}$. For any instance $x$, the Mahalanobis distance based dynamic weight of the $t^{th}$ hypothesis is then computed as

$$D_{W_t}(x) = \frac{1}{\min\{M_{tc}(x)\}}, c = 1, ..., C, t = 1, ..., T$$

where $T$ is the total number of hypotheses generated.

The Mahalanobis distance implicitly assumes that the underlying data distribution is Gaussian, which is general is not the case. Yet it is more informative then other distance metrics as it takes the data covariance into consideration, and provides promising results demonstrating its effectiveness.
CONCLUSIONS
Learn++ also inherits performance improvement properties of AdaBoost. Learn++ is based on the following intuition: Each new classifier added to the ensemble is trained using a set of examples drawn according to a distribution, which ensures that examples that are misclassified by the current ensemble have a high probability of being sampled. In an incremental learning setting, the examples that have a high probability of error are precisely those that are unknown or that have not yet been used to train the classifier. Both AdaBoost and Learn++ generate weak hypotheses and combine them through weighted majority voting of the classes predicted by the individual hypotheses. The hypotheses are obtained by retraining a base classifier (weak learner) using strategically updated distributions of the training database. AdaBoost’s distribution update rule is optimized for improving classifier accuracy, whereas Learn++ distribution update rule is optimized for incremental learning of new data, in particular when the new data introduces new classes.

REFERENCES

AUTHOR’S BIOGRAPHY

Mrs G.T.Prassana Kumari is an Associate Professor in the Department of Computer Science and Engineering, S.V.Engineering College for Women, Tirupati. She is pursuing Ph.D., in Computer Science in the area of Ensemble based Classifiers. She is in teaching since 1999. She presented papers at National and International Conferences and published articles in International journals.

Mr.K.Sekar is an Associate Professor in the Department of Computer Science and Engineering, S.V.Engineering College for Women, Tirupati. He is pursuing Ph.D., in Computer Science in the area of Software Engineering. He is in teaching since 1998. He presented papers at National and International Conferences and published articles in National & International journals.

Mrs R.Poornima is an Assistant Professor in the Department of Computer Science and Engineering, S.V.Engineering College for Women, Tirupati. She is in teaching since 2012. Areas of Interest are Computer Networks, Data Mining and Web Designing.