

Rowan University

Rowan Digital Works

Theses and Dissertations

12-31-2006

Boosted ensemble algorithm strategically trained for the incremental learning of unbalanced data

Michael David Muhlbaier
Rowan University

Follow this and additional works at: <https://rdw.rowan.edu/etd>



Part of the [Electrical and Computer Engineering Commons](#)

Let us know how access to this document benefits you -
share your thoughts on our feedback form.

Recommended Citation

Muhlbaier, Michael David, "Boosted ensemble algorithm strategically trained for the incremental learning of unbalanced data" (2006). *Theses and Dissertations*. 915.

<https://rdw.rowan.edu/etd/915>

This Thesis is brought to you for free and open access by Rowan Digital Works. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Rowan Digital Works. For more information, please contact LibraryTheses@rowan.edu.

BOOSTED ENSEMBLE ALGORITHM STRATEGICALLY TRAINED
FOR THE INCREMENTAL LEARNING OF UNBALANCED DATA

by

Michael David Muhlbaier

A Thesis Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE

Department: Electrical and Computer Engineering
Major: Engineering (Electrical Engineering)

Approved:

Members of the Committee

In Charge of Major Work

For the Major Department

For the College

Rowan University
Glassboro, New Jersey
2006
© Michael Muhlbaier

ABSTRACT

Michael D. Muhlbaier

BOOSTED ENSEMBLE ALGORITHM STRATEGICALLY TRAINED FOR INCREMENTAL LEARNING OF UNBALANCED DATA

2006

Dr. Robi Polikar

Master of Science in Electrical Engineering

Many pattern classification problems require a solution that needs to be incrementally updated over a period of time. Incremental learning problems are often complicated by the appearance of new concept classes and unbalanced cardinality in training data. The purpose of this research is to develop an algorithm capable of incrementally learning from severely unbalanced data. This work introduces three novel ensemble based algorithms derived from the incremental learning algorithm, Learn++. Learn++.NC is designed specifically for incrementally learning *New Classes* through dynamically adjusting the combination weights of the classifiers' decisions. Learn++.UD handles *Unbalanced Data* through class-conditional voting weights that are proportional to the cardinality differences among *training datasets*. Finally, we introduce the *Boosted Ensemble Algorithm Strategically Trained* (BEAST) for incremental learning of unbalanced data. BEAST combines Learn++.NC and Learn++.UD with additional strategies that compensate for unbalanced data arising from cardinality differences among *concept classes*. These three algorithms are investigated both analytically and empirically through a series of simulations. The simulation results are presented, compared and discussed. While Learn++.NC and Learn++.UD perform well on the specific problems they were designed for, BEAST provides a strong and more robust performance on a much broader spectrum of complex incremental learning and unbalanced data problems.

ACKNOWLEDGEMENTS

First I would like to thank Apostolos Topalis for his role in developing Learn++.MT (Muhlbaier-Topalis), Learn++.MT2, and the BEAST. I would also like to extend the deepest gratitude to my parents and twin sister who have shown me an overwhelming amount of love and support. This work is a direct result of their encouragement and dedication to my success.

I would like to thank the entire Electrical and Computer Engineering faculty for their investment in my undergraduate and graduate education. Particularly, I would like to thank my advisor, Dr. Robi Polikar, for his commitment to this field of research. He has graciously provided me with the intellectual support which has made this research possible. Additionally, I would like to thank my committee members, Dr. Shreekanth Mandayam and Dr. Adrian Rusu. I also want to thank my colleagues from Spaghetti Engineering and the ECE Graduate Student Office. I would also like to acknowledge the National Science Foundation who has supported this work under Grant No. ECS-0239090, "CAREER: An Ensemble of Classifiers Approach for Incremental Learning."

Finally, I would want to thank my creator, God almighty, all of my abilities and intelligence I owe to Him. I want to thank Him not only as my creator, but also as my savior, Jesus Christ, who has given His life so that I might live. I would like to dedicate this work to the glory of His name. "Praise and glory and wisdom and thanks and honor and power and strength be to our God for ever and ever Amen!" (Rev. 7:12)

TABLE OF CONTENTS

CHAPTER 1 - INTRODUCTION.....	1
1.1 Incremental Learning.....	1
1.2 Unbalanced Data.....	2
1.3 Scope of Thesis.....	3
1.4 Organization of Thesis.....	4
CHAPTER 2 - BACKGROUND.....	5
2.1 Incremental Learning.....	5
2.2 Ensemble of Classifiers.....	6
2.2.1 BAGGING.....	8
2.2.2 BOOSTING.....	9
2.2.3 AdaBoost.....	10
2.3 Learn++.....	13
CHAPTER 3 – ENSEMBLE OF CLASSIFIERS FOR INCREMENTAL LEARNING	17
3.1 The Out Voting Problem.....	17
3.2 Learn++.NC.....	19
3.2.1 Dynamically Weighted Majority Voting.....	25
3.3 Unbalanced Learning.....	27
3.4 Problems with Current Approaches.....	33
CHAPTER 4 – THE BEAST ALGORITHM.....	34
4.1 Different Elements of the BEAST Algorithm.....	34
4.1.1 Class Specific Weights.....	35
4.1.2 Preliminary Confidence and Normalization.....	36
4.1.3 Preliminary Confidence Transfer Function.....	36
4.1.4 Sub Ensembles and Decision Update.....	37
4.2 BEAST.....	39
4.3 BEAST-ED.....	41
CHAPTER 5 – IMPLEMENTATION RESULTS.....	46
5.1 Organization and Motivation of Simulations.....	47
5.1.1 Synthetic Incremental Learning Problems.....	47
5.1.2 Experimental Incremental Learning Problems.....	48
5.1.3 Presentation of Results.....	48
5.2 Simulation Results on Synthetic Databases.....	50
5.2.1 Synthetic Experiment 1 – Incremental Learning.....	51
5.2.2 Synthetic Experiment 2 – Unbalanced Data.....	59
5.2.3 Synthetic Experiment 3 - Spiral Database.....	67
5.3 Simulation Results on Experimental Databases.....	75
5.3.1 Volatile Organic Compounds Recognition Database.....	77
5.3.2 Optical Character Recognition Database.....	82

CHAPTER 6 - CONCLUSIONS	89
6.1 Summary of Experimental Findings	90
6.2 Contributions of this Work	91
6.3 Recommendations for Future Work.....	91
6.3.1 Preliminary Confidence Transfer Function	91
6.3.2 Sub-Ensemble Combination Techniques.....	92
6.3.3 Incorporation of Prior Knowledge.....	92
REFERENCES	94
APPENDIX A – EXPERIMENT WITH BEAST	100
APPENDIX B – RESULTS USING VALIDATION DATA.....	110
Synthetic Experiment 1 – Incremental Learning	110
Synthetic Experiment 2 – Unbalanced Data	112
Synthetic Experiment 3 - Spiral Database	114
Volatile Organic Compounds Recognition Database	115
Optical Character Recognition Database	116

LIST OF FIGURES

Figure 2.1 - <i>Graphical depiction of learning with multiple classifiers.</i>	7
Figure 2.2 - <i>BAGGING algorithm.</i>	8
Figure 2.3 - <i>BOOSTING algorithm.</i>	9
Figure 2.4 - <i>Algorithm Hedge(β).</i>	11
Figure 2.5 - <i>AdaBoost.M1 algorithm.</i>	13
Figure 2.6 - <i>Learn++ algorithm.</i>	15
Figure 3.1 - <i>Learn++.NC algorithm.</i>	21
Figure 3.2 - <i>Block diagram of the Learn++.NC algorithm.</i>	22
Figure 3.3 - <i>Dynamically weighted voting algorithm.</i>	27
Figure 3.4 - <i>Learn++.UD algorithm.</i>	32
Figure 4.1 - <i>Example of transfer functions generated by equation (4.6)</i>	37
Figure 4.2 - <i>The BEAST algorithm.</i>	40
Figure 4.3 - <i>BEAST-ED algorithm.</i>	44
Figure 5.1 - <i>Parzen window example showing the estimated PDF calculated from five observations.</i>	49
Figure 5.2 - <i>Four PDFs corresponding to the classes described in Table 5.1.</i>	52
Figure 5.3 - <i>Example training and testing data from Experiment 1.</i>	53
Figure 5.4 - <i>Posterior probability output of the Bayes classifier over the entire feature space.</i>	54
Figure 5.5 - <i>Generalization performance vs. number of classifiers.</i>	55
Figure 5.6 - <i>Theoretical performance probability density functions.</i>	56
Figure 5.7 - <i>Decision boundaries over the entire feature space.</i>	57
Figure 5.8 - <i>Class specific generalization performances from experiment 1.</i>	58
Figure 5.9 - <i>Example of the four PDF's corresponding to the classes described in Table 5.4.</i>	60
Figure 5.10 - <i>Example training and testing data distributed according to Table 5.5.</i>	61
Figure 5.11 - <i>Posterior probability output of the Bayes classifier over the entire feature space.</i>	62
Figure 5.12 - <i>Generalization performance vs. number of classifiers.</i>	63
Figure 5.13 - <i>Performance probability density functions from experiment 2.</i>	64
Figure 5.14 - <i>Decision boundaries over the entire feature space.</i>	65
Figure 5.15 - <i>Class specific generalization performances.</i>	67
Figure 5.16 - <i>Graph of the four PDF's corresponding to the spiral classes described Table 5.7. The PDF's are shows in polar coordinates (left) and cartesian coordinates (right).</i>	69
Figure 5.17 - <i>Sample training and testing data distributed according to Table 5.8.</i>	70
Figure 5.18 - <i>Posterior probability output of the Bayes classifier over the entire feature space.</i>	71
Figure 5.19 - <i>Generalization performance vs. number of classifiers.</i>	72
Figure 5.20 - <i>Performance probability density functions from the swirl database.</i>	73
Figure 5.21 - <i>Decision boundaries over the entire feature space.</i>	74
Figure 5.22 - <i>Class specific generalization performances.</i>	75

Figure 5.23 – k-fold cross validation diagram [54].	76
Figure 5.24 – <i>Example instances drawn from the VOC recognition database.</i>	77
Figure 5.25 - <i>Generalization performance vs. number of classifiers from VOC experiment.</i>	79
Figure 5.26 - <i>Performance probability density functions from VOC experiment.</i>	80
Figure 5.27 - <i>Class specific generalization performances on the VOC database.</i>	80
Figure 5.28 – <i>Class specific generalization performance of Learn++ with 10 additional classifiers.</i>	82
Figure 5.29 – <i>Example data samples from the OCR database.</i>	83
Figure 5.30 - <i>Generalization performance vs. number of classifiers from OCR experiment.</i>	84
Figure 5.31 - <i>Performance probability density functions from OCR experiment.</i>	86
Figure 5.32 - <i>Class specific generalization performances on the OCR database.</i>	86
Figure 5.33 - <i>Class specific generalization performance of Learn++ with 10 additional classifiers.</i>	88
Figure A.1 – <i>Example of the four PDF's corresponding to the classes described in Table A.1.</i>	100
Figure A.2 – <i>Preliminary Confidence plots after training on \mathfrak{D}_1.</i>	104
Figure A.3 – <i>Preliminary Confidence plots after training on \mathfrak{D}_2.</i>	105
Figure A.4 – <i>Preliminary Confidence plots after training on \mathfrak{D}_3.</i>	106
Figure A.5 – <i>Preliminary Confidence plots after training on \mathfrak{D}_4.</i>	107

LIST OF TABLES

Table 5.1 – Gaussian distribution information of the first experiment.....	51
Table 5.2 – Instance distribution of the Experiment 1.....	52
Table 5.3 – Generalization performance from Experiment 1.	55
Table 5.4 - Gaussian distribution information of the second experiment.....	59
Table 5.5 - Instance distribution of the Experiment 2.	61
Table 5.6 - Generalization performance from Experiment 2.....	63
Table 5.7 – Distribution information of the spiral database.	69
Table 5.8 - Instance distribution of the spiral database.	70
Table 5.9 - Generalization performance on the swirl database.....	72
Table 5.10 – Instance distribution for the VOC database experiment.....	78
Table 5.11 – Generalization performance on VOC database.	78
Table 5.12 – Instance distribution for the OCR database.	83
Table 5.13 - Generalization performance on OCR database.	84
Table A.1 - Gaussian distribution information for BEAST experiment.....	100
Table A.2 – Instance distribution for BEAST experiment.	101
Table B.1 – Instance distribution for experiment 1, with validation data.....	111
Table B.2 - Generalization performance from Experiment 1, with validation data	111
Table B.3 – Instance distribution for experiment 2, with validation data.....	113
Table B.4 - Generalization performance on Experiment 2, with validation data	113
Table B.5 – Instance distribution for swirl experiment, with validation data.....	114
Table B.6 - Generalization performance on Experiment 2, with validation data	114
Table B.7 – Instance distribution for the VOC database, with validation data	115
Table B.8 - Generalization performance on VOC database, with validation data.....	116
Table B.9 - Instance distribution for the OCR database, with validation data.	116
Table B.10 - Generalization performance on OCR database, with validation data.....	117

CHAPTER 1

INTRODUCTION

Supervised classification algorithms, such as neural networks and their many variations, have been extensively used in a wide range of pattern recognition and function approximation applications [1]. Traditionally, the learning process takes place in one step: data is provided to the algorithm along with the correct classification information and the algorithm learns the patterns in the data associated with each classification. All subsequently collected data is then used to test the classifier created by the algorithm. However, an increasing number of practical applications obtain data in installments. A simple solution is to eliminate the existing classifier, combine all obtained data, and repeat the learning process. Yet, some applications make this approach impractical or even impossible: impractical by increasing the training time, in some cases exponentially, as data is collected; impossible when the application does not allow access to previously used data. These applications require a technique that can be trained and incrementally updated, as new data become available, without forgetting the previous acquired knowledge [2;3].

1.1 *Incremental Learning*

The ability of a classifier to learn over a period of time is known as incremental (also called cumulative or lifelong) learning. This style of learning has been given different definitions throughout literature [4-7]. Some definitions assume the algorithm has access to all data, others partial access or no access to previously obtained data. For the purposes of this research it is assumed that incremental learning implies that previously

used data is inaccessible. According to this assumption, an incremental learning algorithm must be capable of learning from newly obtained data while retaining previously learned knowledge. This challenging task raises the stability – plasticity dilemma: a completely stable classifier can retain knowledge, however, cannot learn new information, whereas a completely plastic classifier can instantly learn new information, but cannot retain previous knowledge [8;9]. Many popular classifiers, such as the multilayer perceptron, radial basis function networks, and support vector machine are not structurally suitable for incremental learning, since they are completely “stable”. As previously mentioned, a trivial procedure for learning from new data using such classifiers involves discarding the existing classifier and combining the old and new data to train a new classifier [10;11]. This approach causes all previously learned information to be lost, a phenomenon known as catastrophic forgetting [12]. To further stress the importance of proper incremental learning, consider the following example: suppose your mind was completely “stable”, when trying to learn a face, after meeting a new person, you would have to compare their face against all the faces you have ever seen; conversely, if your mind was completely “plastic” you would only be able to recognize the face of the last person you saw. Fortunately your mind is neither completely “stable” nor completely “plastic”, but instead your mind perfectly balances stability and plasticity in order to learn new information with a minimal loss of previously acquired knowledge. This balance should be the goal of any incremental learning algorithm.

1.2 *Unbalanced Data*

Another common problem in pattern recognition is the lack of properly balanced data. Many real applications make it unpractical to collect equal amount of data from each

concept class. This may be due to different costs involved in collecting the data, or because more samples are difficult, or even impossible, to collect. The relative difference of individual class's cardinalities within a training dataset is commonly referred to as unbalanced data within datasets. In some circumstances these cardinalities are correlated to a quantity often described as class prior probabilities. However, in many practical applications these different cardinalities are due to data collection issues, and can not be assumed to be an indicator of the prior probabilities.

In the incremental learning setting, the issue of unbalanced data can be extended to the discrepancy in the cardinality of each dataset used for incremental learning. In the absence of other information, and under the generally valid assumptions that (i) no instance is repeated and (ii) the noise distribution remains relatively unchanged among datasets, it is reasonable to believe that the dataset that has more instances carries more information. It is not unusual to see major discrepancies in the cardinalities of datasets that subsequently become available. However, a large majority of incremental learning algorithms do not accommodate to differences in dataset cardinalities, resulting in poor performance under these conditions.

1.3 *Scope of Thesis*

The overall objective of this research is to develop an incremental learning algorithm offering superior performance on a broad range of incremental learning problems, specifically those involving unbalanced data, where unbalanced data can refer to any combination of the following: the difference in class cardinalities of one data set; the difference of dataset cardinalities; and the most extreme case, the complete lack or introduction of all instances from a specific class. Previous work introduced the Learn++

algorithm which performs well on a variety of incremental learning problems [13-17]. However, this algorithm performs poorly or performs inconsistently when used on a number of unbalanced data problems [18-20]. The clear identification of these problems led to the development of the following incremental learning algorithms:

- 1) Learn++.NC, which quickly and effectively learns from data that introduce or remove all information from one or more New Classes [18;20].
- 2) Learn++.UD, which handles Unbalanced Data between training sets, allowing information to be learned even from small datasets [19].
- 3) BEAST which performs exceptionally well on the broad range of unbalanced data problems described above.

In this work, these algorithms are analyzed through a rigorous set of experiments to determine their strengths and weaknesses on various types of incremental learning and unbalanced data problems. Their theoretical analysis constitutes future work.

1.4 *Organization of Thesis*

Chapter 2 provides background on ensemble of classifiers systems, including the Learn++ algorithm. Learn++.NC and Learn++.UD are described in detail in Chapter 3 along with a description of the problems they are designed to solve. In addition, the problems which these algorithms can not solve are clearly identified. Chapter 4 introduces the BEAST algorithm and discusses the novel methods used to solve unbalanced data problems. A set of experiments and their results on synthetic and real world databases are presented in Chapter 5. Finally, a summary of conclusions and suggestions for future work are presented in Chapter 6.

CHAPTER 2

BACKGROUND

The incremental learning problem is defined in Section 2.1, along with a brief overview of single classifier solutions to the problem. All work following this section is concerned with ensembles of classifiers, thus a considerable amount of this chapter is spent on the history of multiple classifier systems (Section 2.2). The algorithms developed during the course of this research originated from the Learn++ algorithm are described in Section 2.3.

2.1 *Incremental Learning*

Before trying to compare and contrast various incremental learning algorithms it is important to clearly define the desired characteristics of a solution to the incremental learning problem. An incremental learning algorithm should be able to:

- 1) learn additional information from new data
- 2) learn without access to previously used training data
- 3) retain previously acquired knowledge to avoid catastrophic forgetting
- 4) accommodate newly introduced classes.

Many incremental learning algorithms that exhibit some of these qualities have been developed over the last two decades [21-24]. Some algorithms are designed to retain using a subset of previously used data, in [3;10] the subset is randomly selected and used in a method called pseudorehearsal; whereas [25;26] encodes the previously used data in a way that stores the more informative instances. One of the more notable algorithms is fuzzy ARTMAP which adequately meets all of the above incremental learning

constraints [27-29]. ARTMAP generates a new decision cluster for each perceived new pattern; each cluster is then mapped to a target class. Since the previously created clusters are retained, ARTMAP is capable of incremental learning through the generation of new clusters without access to previously seen data. Furthermore, the new clusters can be mapped to previously seen targets or to novel targets, allowing ARTMAP to accommodate newly introduced classes. Although ARTMAP fits perfectly into the description of incremental learning, it has some drawbacks. In many cases it has been noted that ARTMAP is very sensitive to the selection of the vigilance parameter, which controls the threshold that determines whether a new cluster needs to be created, or if existing clusters can be added to existing ones. ARTMAP is also very sensitive to noise levels in the training data and to the order in which the training data are presented to the algorithm. For example, poor selection of the vigilance parameter can prevent the algorithm from learning complex decision boundaries or cause severe overtraining [30]. However, several methods have been proposed to select parameters and training methods that will increase the generalization performance [31;32].

2.2 *Ensemble of Classifiers*

Ensemble systems were first introduced to improve generalization performance by intelligently combining decisions from multiple classifiers. This is based on the proven assumption that by combining several diverse classifiers a greater performance can be obtained over a single classifier [33]. The general process of ensemble learning is, given training data $\mathfrak{D} = \{(\mathbf{x}_n, y_n), n = 1, \dots, N\}$, \mathfrak{D} is then divided into subsets, \mathfrak{D}_t , and used to independently train a classifier h_t . The decisions of all h_t are then combined in some way to create an ensemble classification H_t . Figure 2.1 graphically shows this process on a toy

dataset. In step (1) the dataset \mathfrak{D} is shown in the feature space. In steps (2)-(4) decision boundaries are generated on random subsets of \mathfrak{D} , these boundaries indicate the decision of the respective classifiers. In step (5) the three decision boundaries are combined to create a more robust ensemble decision, shown in step (6).

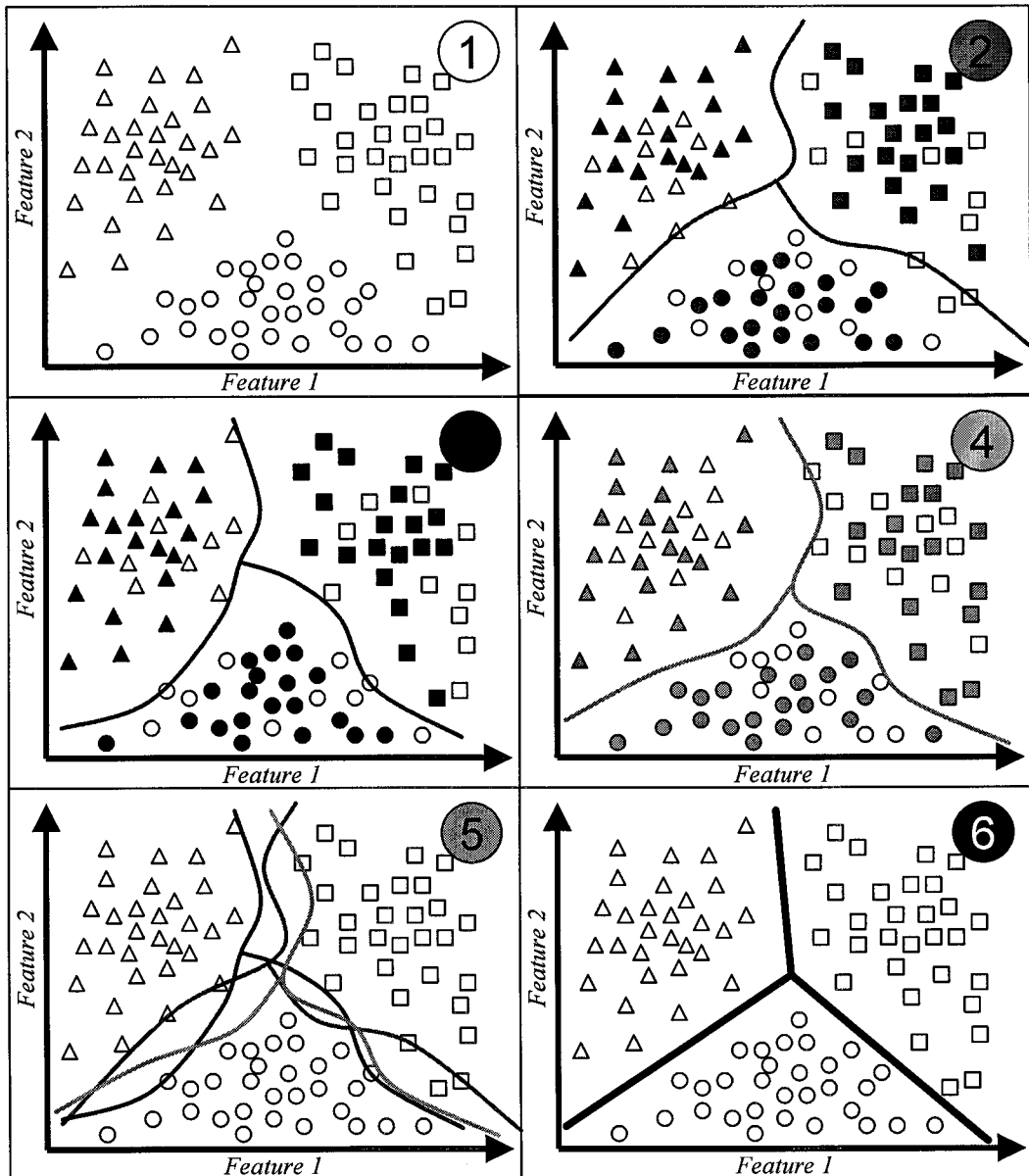


Figure 2.1 - Graphical depiction of learning with multiple classifiers.

The most interesting, and consequently most researched, aspects of ensemble learning are the selection of training data subsets, and the methods used to combine classifier decisions [34-38].

2.2.1 BAGGING

One of the most well known algorithms which combine classifiers to improve performance are bagging and boosting. Bagging, introduced by Breiman [39] as “bootstrap aggregating,” is undoubtedly the simplest implementation of the ensemble method shown in Figure 2.1. Bagging works by randomly sampling \mathcal{D} , with replacement, in order to create T subsets, \mathcal{D}_t , which are then used to train T different classifiers, h_t . The T classifiers are added to form an ensemble, and used to classify test instances. The ensemble classification, H , is then determined to be the class with the most votes.

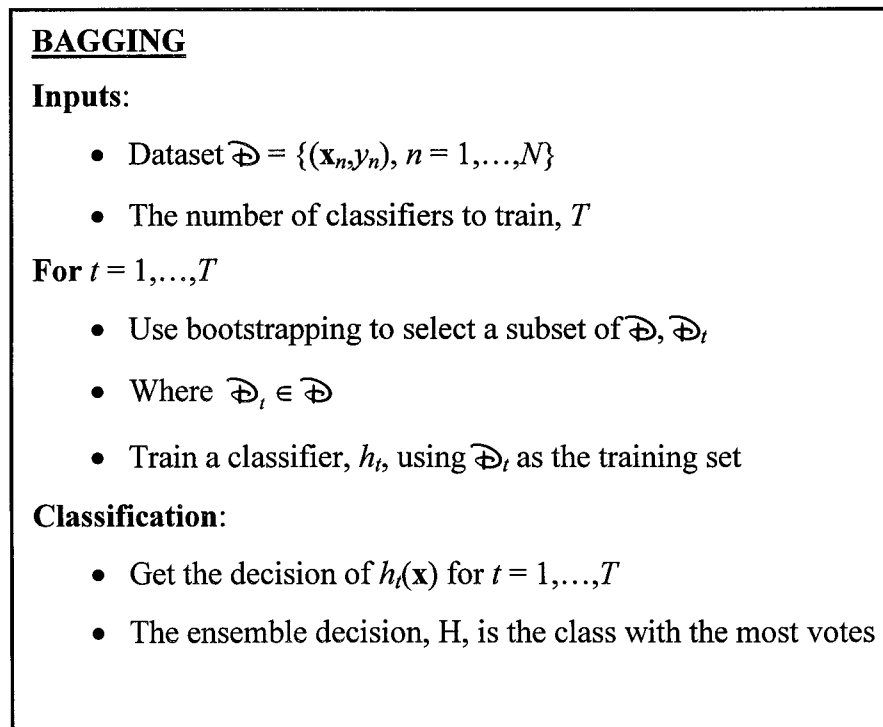


Figure 2.2 - BAGGING algorithm.

2.2.2 BOOSTING

Other basic ensemble algorithms are the boosting [40;41] and LEARN [33] algorithms which use a more sophisticated approach to generate an ensemble with only three classifiers.

BOOSTING

Inputs:

- Dataset $\mathcal{D} = \{(\mathbf{x}_n, y_n), n = 1, \dots, N\}$

Training:

- Select $N_1 < N$ training points from \mathcal{D} without replacement to create \mathcal{D}_1
- Train classifier h_1 on \mathcal{D}_1
- Select data for \mathcal{D}_2 such that half of the instances are classified correctly by h_1
- Loop until no data remains in \mathcal{D}

- Generate a random bit (0 or 1)

If 0

Select an instance from \mathcal{D} and present it to h_1 until the first instance is misclassified, add this instance to \mathcal{D}_2 .

If 1

Select an instance from \mathcal{D} and present it to h_1 until the first instance is correctly classified, add this instance to \mathcal{D}_2 .

- Train classifier h_2 on \mathcal{D}_2
- Create \mathcal{D}_3 by selecting all patterns from \mathcal{D} where h_1 and h_2 disagree
- Train classifier h_3 on \mathcal{D}_3

Classification :

Classify the test data instance, \mathbf{x} , with all classifiers, $h_1(\mathbf{x})$, $h_2(\mathbf{x})$, and $h_3(\mathbf{x})$

If $h_1(\mathbf{x}) = h_2(\mathbf{x})$

$$H(\mathbf{x}) = h_1(\mathbf{x}) = h_2(\mathbf{x})$$

If $h_1(\mathbf{x}) \neq h_2(\mathbf{x})$

$$H(\mathbf{x}) = h_3(\mathbf{x})$$

Figure 2.3 - BOOSTING algorithm.

The approach in boosting is to ensure that the three classifiers in the ensemble are as diverse and complementary as possible. The first classifier, h_1 , is trained with a randomly drawn subset of the training data. The second classifier, h_2 , is trained with the most informative dataset given what h_1 already learned. This is done by selecting a set of instances where half of the instances are classified correctly by h_1 and the other half misclassified. The third classifier, h_3 , is then trained on all instances where the decisions of h_1 and h_2 disagree. Figure 2.3 shows the pseudocode for the boosting algorithm.

The strong learning algorithm, LEARN, is an extension of the boosting algorithm in Figure 2.3. The LEARN algorithm adds several complex controls after the training of h_1 and h_2 to ensure that their associated errors fall within certain tolerances [33].

2.2.3 AdaBoost

In [42] Freund and Schapire seek to improve on previously introduced BOOSTING algorithms using a more statistical approach for both the training and combining “experts”. Assuming that a set of “experts” are available, which perform slightly better than random guessing, their algorithm is theoretically shown to reduce error as more “experts” are added to the ensemble. AdaBoost is an extension of Freund and Schapire’s solution to the online allocation problem, known as **Hedge**(β).

2.2.3.1 *Hedge*(β)

Given N strategies (or “experts”) the allocation algorithm generates a weight vector \mathbf{p}^t which attempts minimize the loss of the system over a number of time steps $t = 1, \dots, T$. There is assumed to be a loss vector, ℓ_t , received from the environment. The overall loss of the system is then:

$$L = \sum_{t=1}^T \mathbf{p}^t \cdot \ell^t \quad (2.1)$$

The goal is to set \mathbf{p}^t such that L is minimized. Thus, it is intuitive to allow ℓ_t to modify \mathbf{p}^t such that the strategies that incur more loss are weighted less.

$$w_i^{t+1} = w_i^t \cdot \beta^{\ell_i^t} \quad (2.2)$$

$$\mathbf{p}^t = \frac{\mathbf{w}^t}{\sum_{i=1}^N w_i^t} \quad (2.3)$$

where $\beta \in [0, 1]$ and $\ell_i^t \in [0, 1]$, thus also bounding \mathbf{w}^t between 0 and 1. Furthermore, equation (2.2) can be expanded to update w_i^t using any function bounded by 0 and 1. Figure 2.4, shows the **Hedge**(β) algorithm in its entirety as presented in [42].

Algorithm Hedge(β)

Parameters: $\beta \in [0, 1]$

initial weight vector $\mathbf{w}^1 \in [0, 1]^N$ with $\sum_{i=1}^N w_i^1 = 1$

number of trials T

Do for $t = 1, 2, \dots, T$

1. Choose allocation

$$\mathbf{p}^t = \frac{\mathbf{w}^t}{\sum_{i=1}^N w_i^t}$$
2. Receive loss vector $\ell^t \in [0, 1]^N$ from environment.
3. Suffer loss $\mathbf{p}^t \cdot \ell^t$.
4. Set the new weights vector to be

$$w_i^{t+1} = w_i^t \beta^{\ell_i^t}$$

Figure 2.4 - Algorithm Hedge(β).

2.2.3.2 AdaBoost

AdaBoost, short for **Adaptive Boosting**, combines the online allocation method of **Hedge(β)** with the majority voting [43] version of the BAGGING method shown in Figure 2.2. The algorithm **WeakLearn** is used to generate a weak hypothesis or classifier given a set of training data. Unlike the initial boosting algorithms proposed by Schapire [33] and Freund [43;44], AdaBoost does not require prior information about the accuracies of the weak hypothesis. Instead AdaBoost adapts to these accuracies through its weighted majority voting mechanism.

Figure 2.5 shows the pseudocode for the most common version of AdaBoost, which is designed for multiclass problems. AdaBoost uses **WeakLearn** to generate T classifiers using data selected from \mathfrak{D} according to D_t . D_t is updated using a method very similar to that of **Hedge(β)**. Instead of reducing the weights of strategies that incur heavy losses, the algorithm increases the weights of training examples which are difficult to classify. This ultimately reduces the loss of the system by increasing the likelihood of creating a classifier that is complementary to the ensemble. Furthermore, **Hedge(β)** requires the β parameter to be constant, where as AdaBoost adaptively optimizes the β parameter for each training example. Once AdaBoost is finished training, classifiers generate the final hypothesis as the weighted sum of the individual hypotheses, where each hypothesis is weighted proportional to its performance on the entire training data \mathfrak{D} .

AdaBoost.M1

Inputs:

- Dataset $\mathfrak{D} = \{(\mathbf{x}_n, y_n), n = 1, \dots, N\}$
- Distribution D over all N instances
- Weak learning algorithm **WeakLearn**
- Integer T specifying the number of classifiers to generate

Initialize the weight vector: $w_1(i) = D(i)$ for $i = 1, \dots, N$

Do for $t = 1, 2, \dots, T$

1. Set $D_t = w_t / \sum_{i=1}^m w_t(i)$ so that D_t is a distribution.

2. Call **WeakLearn** providing it with training data selected according to D_t ; get back a hypothesis $h_t: X \rightarrow Y$.

3. Calculate the error of h_t : $\varepsilon_t = \sum_{i: h_t(x_i) \neq y_i} D_t(i)$.

If $\varepsilon_t > 1/2$, then set $T = t - 1$ and abort loop.

4. Set $\beta = \varepsilon_t / (1 - \varepsilon_t)$.

5. Set the new weight vector:

$$w_{t+1}(i) = w_t \times \begin{cases} \beta, & \text{if } h_t(\mathbf{x}_i) = y_i \\ 1, & \text{otherwise} \end{cases}$$

Output the hypothesis

$$H_f(x_i) = \arg \max_c \sum_{t: h_t(x_i) = c} \left(\log \frac{1}{\beta_t} \right)$$

Figure 2.5 - AdaBoost.M1 algorithm.

2.3 Learn++

Learn++ was proposed as an incremental learning algorithm that exhibits a fine balance across the stability – plasticity spectrum: it is capable of learning from new data, while substantially retaining previous knowledge without requiring access to the previously

used data, even when new data includes instances from previously unseen classes [15]. The algorithm, inspired by AdaBoost, takes advantage of the synergistic learning ability of an ensemble of classifiers. While both algorithms sequentially generate an ensemble of diverse classifiers that are combined through a weighted majority voting procedure, Learn++ is primarily geared towards incremental learning [16;17], while AdaBoost is intended for improving generalization performance of a weak classifier [42]. Learn++ aims to incrementally learn the newly available information. More specifically, each classifier generated by Learn++ is trained on a subset of the current training dataset. The instances of each subset are drawn according to an iteratively updated distribution that is strategically biased towards those instances that carry novel information. The relative performance of each classifier on its training data then determines its voting weight to be used in weighted majority voting [45], where the ensemble chooses the class that receives the highest total vote from individual classifiers. As new data become available, Learn++ generates additional classifiers, until the ensemble learns the novel information. Since no classifier is discarded, previously acquired knowledge is not lost. Previous studies have shown that Learn++ is capable of using any supervised neural network as its base classifier, thus making the algorithm classifier independent [16].

Algorithm Learn++

Input: For each dataset \mathfrak{D}^k $k=1, 2, \dots, K$

- Training data $\mathfrak{D}^k = \{(x_1, y_1), (x_2, y_2), \dots, (x_{m_k}, y_{m_k})\}$ $y_i \in Y_k \subset \{\omega_1, \dots, \omega_c\}$
- Weak learning algorithm **BaseClassifier**.
- Integer T_k , specifying the number of **BaseClassifiers** to create using \mathfrak{D}^k .

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

If $k \neq 1$, **Set** $t = 0$ and **Go to** step 5 of the following Do loop to adjust initialization weights

Do for $t = eT_k + 1, eT_k + 2, \dots, eT_k + T_k$:

1. Set $D_t(i) = w_t(i) / \sum_{i=1}^m w_t(i)$ so that D_t is a distribution.
2. Call **BaseClassifier**, providing it with $\mathfrak{D}_t^k \in \mathfrak{D}^k$, drawn according to D_t .
3. Obtain a hypothesis $h_t : X \rightarrow Y$, and calculate its the error $\varepsilon_t = \sum_{i: h_t(x_i) \neq y_i} D_t(i)$. If $\varepsilon_t > 1/2$, discard h_t and go to step 2. Otherwise, compute normalized error $\beta_t = \varepsilon_t / (1 - \varepsilon_t)$.
4. Call weighted majority voting to obtain the composite hypothesis

$$H_t(x_i) = \arg \max_{\omega_c} \sum_{t: h_t(x_i) = \omega_c} \left(\log \frac{1}{\beta_t} \right)$$

5. Compute the error of the composite hypothesis $E_t = \sum_{i: H_t(x_i) \neq y_i} D_t(i)$

6. Set $B_t = E_t / (1 - E_t)$, and update the instance weights:

$$w_{t+1}(i) = w_t \times \begin{cases} B_t, & \text{if } H_t(x_i) = y_i \\ 1, & \text{otherwise} \end{cases}$$

Call weighted majority voting to obtain the final hypothesis

$$H_{final}(x_i) = \arg \max_{\omega_c} \sum_{t: h_t(x_i) = \omega_c} \left(\log \frac{1}{\beta_t} \right)$$

Figure 2.6 – Learn++ algorithm.

As reported in [15-18;46;47], Learn++ works rather well on a variety of real world problems, even when new classes are introduced with the new data. However, learning new classes come at the expense of requiring a large number of classifiers, an outcome originally thought to be due to well established property of ensemble systems performing best with large number of classifiers. We now realize, however, that classifier proliferation is really an artifact of the voting procedure, and the number of classifiers required to learn new classes can in fact be drastically reduced using a more strategic classifier combination process. In the following section the fundamentals of this problem and potential solutions are explained in detail.

CHAPTER 3

ENSEMBLE OF CLASSIFIERS FOR INCREMENTAL LEARNING

The primary focus of this chapter is to propose novel solutions to specific problems that are not addressed by existing. Two main problems are discussed, the outvoting problem (Section 3.1), and the unbalanced data problem (Section 3.3). A novel solution to the outvoting problem is introduced in Section 3.2 along with a partial solution to the unbalanced data problem in Section 3.3. The chapter concludes with Section 3.4, which highlights the limitations of the proposed algorithms along with the identification of problems which they leave unsolved.

3.1 *The Out Voting Problem*

Despite its promising performance on a variety of applications, Learn++ has its own shortcomings. One particularly important issue is the “out-voting” problem which occurs when a new class is to be learned by an existing ensemble. While Learn++ is capable of learning new class boundaries, it does so at a cost of generating a large number of classifiers. To understand the root cause of the classifier proliferation, consider an ensemble of classifiers that are combined through majority voting. In such a voting mechanism, individual classifiers vote on the class they predict, and the final classification is determined as the class that receives the highest total vote from all classifiers. Learn++ originally used the weighted majority voting, inherited from AdaBoost, where each classifier receives a voting weight based on its training performance. This works rather well in practice even in an incremental learning setting, where new ensemble members are generated with each new dataset to be learned.

However, if the incremental learning problem involves introduction of new classes, as well as some or a subset of the previously seen ones, then the voting scheme proves to be unfair towards the newly introduced class. This is because, instances of a new class that are introduced by the additional data, will be naturally, yet incorrectly, classified into one of previously seen classes by the existing classifiers. Since none of the previously generated classifiers can pick the new class, the decision of the newly generated classifiers on the new class will be outvoted by the previously trained classifiers, until there are enough new classifiers to out vote the previously trained classifiers. In other words, a relatively large number of new classifiers need to be generated that recognize the new class, so that their total weight can out-vote the previous batch of classifiers on instances coming from this new class, consequently populating the ensemble with an unnecessarily large number of classifiers.

We now understand that the out-voting problem is a natural outcome of the voting process, and it can be rectified by a more strategic classifier combination process. The goal is therefore to address this out-voting problem by changing the classifier combination process in such a way that classifiers dynamically adjust their voting weights in proportion to the predicted confidence they have on their decision, which are themselves weighed by the confidence of other classifiers in the ensemble.

The clear identification of this problem led to the development of Learn++.NC (New Class) which is the new name for the previously published Learn++.MT. Learn++.NC is specifically designed to address this issue of classifier proliferation. The primary novelty in Learn++.NC is the way in which the voting weights are determined. Learn++.NC, just like its predecessor Learn++, also obtains a set of voting weights based

on the individual performances of the classifier, however, these weights are then dynamically adjusted based on the classification of the specific instance at the time of testing. The following section explains in detail the Learn++.NC algorithm and why it is able to greatly reduce classifier proliferation.

3.2 *Learn++.NC*

For any given test instance, Learn++.NC compares the class predictions of each classifier and cross-references them with the classes on which they were trained. Essentially, if a subsequent ensemble of classifiers trained on a previously unseen class overwhelmingly predicts the new class on a given instance, then the voting weights of those classifiers that have not seen the new class are proportionally reduced. As an example, assume that an ensemble of classifiers is trained with instances from classes ω_1 and ω_2 , and a second ensemble of classifiers is trained with instances from classes ω_1 , ω_2 and ω_3 . For any given test instance, if the second ensemble picks ω_3 , the classifiers in the first ensemble realize that they have not seen any data from a third class and reduce their voting weights proportional to the ratio of the classifiers in the second ensemble that pick class ω_3 . We will refer to this ratio as the preliminary confidence of the second ensemble in predicting class ω_3 . Learn++.NC can keep track of which ensembles have been trained on any given class. In the above example, knowing that the second ensemble of classifiers have seen class ω_3 instances, and that the first ensemble classifiers have not, it is reasonable to believe that the second ensemble of classifiers are correct in their decision, particularly if these classifiers overwhelmingly choose class ω_3 for a given instance. To the extent that the second ensemble of classifiers are confident of their decision, the voting weights of the first ensemble of classifiers are then reduced. The

pseudocode and block diagram of the Learn++.NC algorithm are given in Figure 3.1 and Figure 3.2, respectively, and the algorithm is formally explained in detail below.

For each database (\mathfrak{D}^k) that becomes available to Learn++.NC, the inputs to the algorithm are (i) $\mathfrak{D}^k = \{(x_1, y_1), (x_2, y_2), \dots, (x_{m_k}, y_{m_k})\}$, the k^{th} dataset consisting of a sequence of m_k training data instances x_i along with their correct labels y_i , $i = 1, \dots, m_k$; (ii) a classification algorithm **BaseClassifier**, and (iii) an integer T_k specifying the maximum number of classifiers to be generated using database \mathfrak{D}^k . For each such database, the algorithm generates an ensemble of classifiers, each trained on a different subset, \mathfrak{D}_t^k , of the available training data. The instances to be used for training each classifier is drawn from a distribution D_t obtained from a set of weights w_t , maintained on the training data. If the algorithm is being trained on its first database ($k = 1$), the data distribution, D , is initialized to be uniform, making the probability of any instance being selected equal. If $k > 1$ then a distribution re-initialization sequence initializes the data distribution. For each database \mathfrak{D}^k the algorithm then adds T_k classifiers to the ensemble starting at $t=eT_k+1$ where eT_k denotes the number of classifiers that currently exist in the ensemble.

For each iteration t , the instance weights, w_t , from the previous iteration are first normalized (step 1) to create the weight distribution D_t .

$$D_t = \mathbf{w}_t / \sum_{i=1}^m w_t(i) \quad (3.1)$$

Algorithm Learn++.NC

Input: For each dataset $\mathfrak{D}^k, k = 1, 2, \dots, K$

- Training data $\mathfrak{D}^k = \{(x_1, y_1), (x_2, y_2), \dots, (x_{m_k}, y_{m_k})\}$ $y_i \in Y_k \subset \{\omega_1, \dots, \omega_c\}$
- Weak learning algorithm **BaseClassifier**.
- Integer T_k , specifying the number of **BaseClassifiers** to create using \mathfrak{D}^k .

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

If $k \neq 1$, **Set** $t = 0$ and **Go to** step 5 of the following **Do** loop to adjust initialization weights

Do for $t = eT_k + 1, eT_k + 2, \dots, eT_k + T_k$:

1. Set $D_t(i) = w_t(i) / \sum_{i=1}^m w_t(i)$ so that D_t is a distribution.
2. Call **BaseClassifier**, providing it with $\mathfrak{D}_t^k \in \mathfrak{D}^k$, drawn according to D_t .
3. Obtain a hypothesis $h_t : X \rightarrow Y$, and calculate its the error $\varepsilon_t = \sum_{i: h_t(x_i) \neq y_i} D_t(i)$. If $\varepsilon_t > 1/2$, discard h_t and go to step 2. Otherwise, compute normalized error $\beta_t = \varepsilon_t / (1 - \varepsilon_t)$.
4. Let $CTr_t = Y_{k(t)} \subset \{\omega_1, \dots, \omega_c\}$, be the class labels used in training h_t for dataset D_k .
5. Call **DWV** to obtain the composite hypothesis H_t .
6. Compute the error of the composite hypothesis $E_t = \sum_{i: H_t(x_i) \neq y_i} D_t(i)$
7. Set $B_t = E_t / (1 - E_t)$, and update the instance weights:

$$w_{t+1}(i) = w_t \times \begin{cases} B_t, & \text{if } H_t(x_i) = y_i \\ 1, & \text{otherwise} \end{cases}$$

Call **DWV** to obtain the final hypothesis, H_{final}

Figure 3.1 – *Learn++.NC* algorithm.

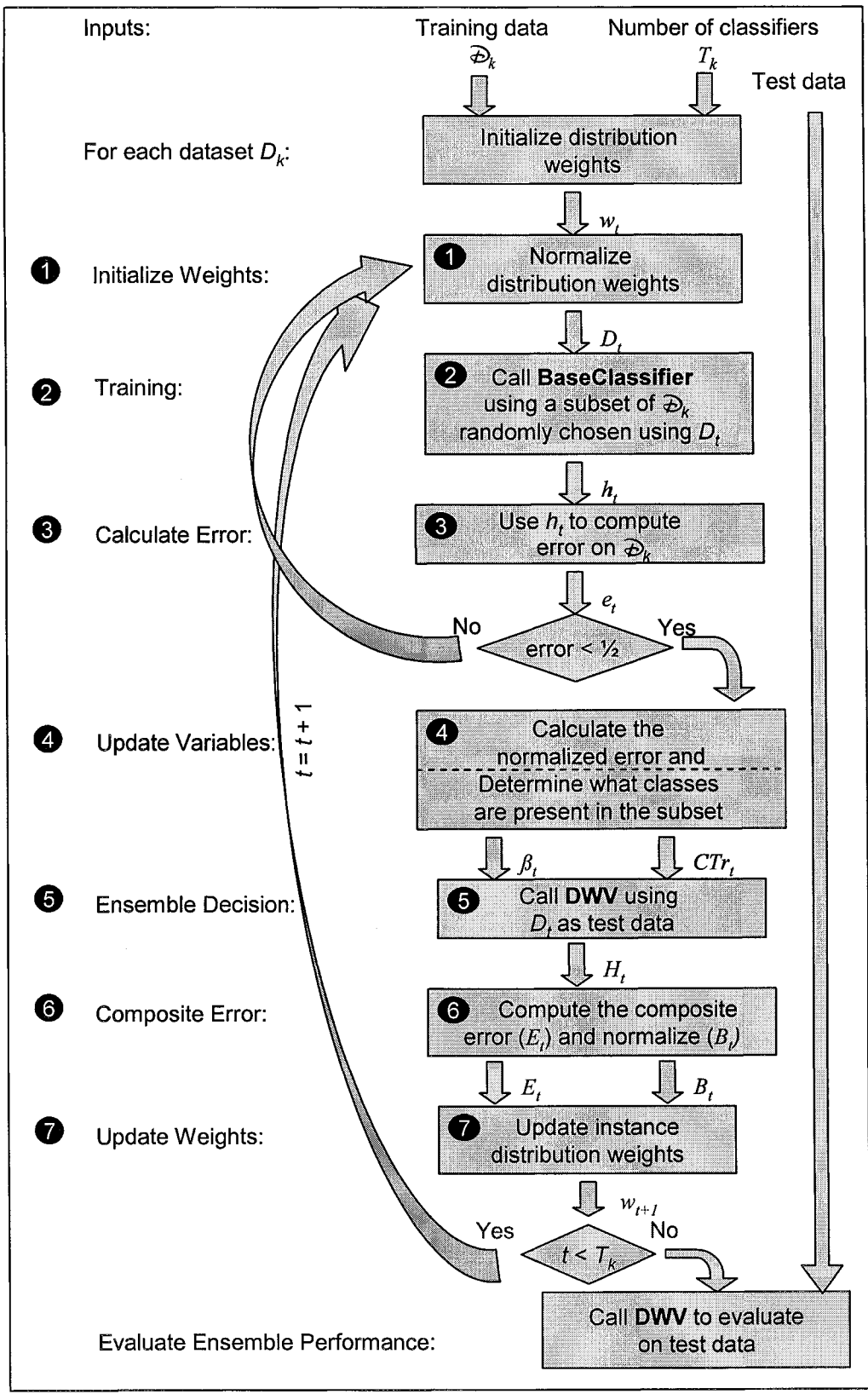


Figure 3.2 - Block diagram of the Learn++.NC algorithm.

A subset of \mathfrak{D}^k is drawn according to D_t , to obtain \mathfrak{D}_t^k , the training data subset to train the t^{th} classifier (hypothesis) h_t , using the **BaseClassifier** (step 2). The error, ε_t , of h_t is calculated on all instances contained in \mathfrak{D}_t^k .

$$\varepsilon_t = \sum_{i:h_t(\mathbf{x}_i) \neq y_i} D_t(i) = \sum_{i=1}^{m_k} D_t(i) \cdot \llbracket h_t(\mathbf{x}_i) \neq y_i \rrbracket \quad (3.2)$$

where $\llbracket \cdot \rrbracket$ evaluates to 1, if the predicate is true, and zero, otherwise. The **BaseClassifier** can be any supervised classifier, whose weakness can be adjusted to ensure adequate diversity, whereby sufficiently different decision boundaries are generated each time the classifier is trained on a different training dataset. This weakness can be controlled by adjusting training parameters (such as the size or error goal of a neural network) with respect to the complexity of the problem. However, a meaningful minimum performance is enforced: the probability of any classifier to produce the correct labels on a given training dataset, weighted proportionally to individual instances' probability of appearance, must be at least $\frac{1}{2}$. 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) [48-50], this condition is necessary and sufficient for a two-class problem ($C=2$); and it is sufficient, but not necessary, for $C>2$.

Therefore, if $\varepsilon_t > \frac{1}{2}$, the algorithm deems the current classifier, h_t , to be too weak, discards it, and returns to step 2, otherwise, calculates the normalized error ε_t , $0 \leq \varepsilon_t \leq 1$ (step 3).

$$\beta_t = \varepsilon_t / (1 - \varepsilon_t) \quad (3.3)$$

The class labels of the training instances used to generate h_t are then stored as CTr_t (step 4).

$$CTr_t = Y_{k(t)} \subset \{\omega_1, \dots, \omega_c\} \quad (3.4)$$

where $Y_{k(t)}$ is the set of concept classes represented by the training data used to generate h_t . The dynamically weighted majority voting (DWMV) subroutine of Learn++.NC, described below, is then called to combine all hypotheses generated thus far, that is, to obtain the composite hypothesis, H_t , of the ensemble (step 5). H_t represents the ensemble decision of the first t hypotheses generated thus far. The error of the composite hypothesis, E_t is then computed and normalized to obtain $0 \leq B_t \leq 1$. (step 6).

$$E_t = \sum_{i: H_t(\mathbf{x}_i) \neq y_i} D_t(i) = \sum_{i=1}^{m_k} D_t(i) [H_t(\mathbf{x}_i) \neq y_i] \quad (3.5)$$

$$B_t = E_t / (1 - E_t) \quad (3.6)$$

The instance weights w_t are finally updated according to the performance of H_t (step 7) such that the weights of instances correctly classified by H_t are reduced and those that are misclassified are effectively increased.

$$w_{t+1}(i) = w_t(i) \times B_t^{1 - [H_t(\mathbf{x}_i) \neq y_i]} = w_t(i) \times \begin{cases} B_t, & \text{if } H_t(\mathbf{x}_i) = y_i \\ 1, & \text{otherwise} \end{cases} \quad (3.7)$$

Equation (3.7) indicates that the distribution weights of the instances correctly classified by the composite hypothesis H_t are reduced by a factor of B_t ($0 < B_t < 1$), which effectively increases the weights of the misclassified instances making them more likely to be selected to the training subset of the next iteration. We note that this weight update rule, based on the performance of the current ensemble, facilitates incremental learning. This is because, unlike AdaBoost and its variations whose weight distribution update is based on the performance of the previously generated hypothesis h_t , Learn++.NC updates its distribution based on the performance of the composite hypothesis (that is, the entire ensemble). This composite hypothesis based weight update procedure forces the

algorithm to focus more and more on instances that have not been seen or properly learned by the current ensemble. This allows efficient incremental learning, because the instances introduced by the new dataset – in particular if they come from a new class – are precisely such instances that are not yet learned (or seen) by the ensemble. It can be argued that AdaBoost too looks (albeit indirectly) at the ensemble decision since, while based on a single hypothesis, the distribution update is cumulative. However, the update in Learn++ is directly tied to the ensemble decision, and hence been found to be more efficient in learning new information in our previous trials [46].

3.2.1 Dynamically Weighted Majority Voting

The dynamically weighted voting (DWV) is illustrated in Figure 3, and described below in detail. The inputs to DWV are (i) the data points to be classified, (ii) classifiers h_t ; (iii) ϵ_t , normalized error for each h_t , and (iv) a vector CTr_t containing the classes on which h_t has been trained. Classifier weights are first initialized according to Equation (1). Each classifier then receives a standard weight that is inversely proportional to its normalized error ϵ_t so that classifiers that performed well on their training data are given higher voting weights.

$$W_t = \log(1/\beta_t) \quad (3.8)$$

A normalization factor Z_c is then created as the sum of the weights of all classifiers trained with class ω_c .

$$Z_c = \sum_{t:c \in CTr_t} W_t \quad (3.9)$$

For each instance, a preliminary confidence factor P_c is generated for each class.

$$P_c(\mathbf{x}_i) = \frac{\sum_{t:h_t(\mathbf{x}_i)=c} W_t}{Z_c} \quad (3.10)$$

P_c is the sum of weights of all classifiers that choose class ω_c , divided by the sum of the weights of all classifiers trained with class ω_c (Z_c). The class-specific confidence $P_c(\mathbf{x}_i)$ represents the collective confidence of classifiers trained on class ω_c in classifying instance \mathbf{x}_i as class ω_c . A high value of $P_c(\mathbf{x}_i)$ indicates that classifiers trained to recognize class ω_c have overwhelmingly picked class ω_c . Classifiers not trained on class ω_c then look at the overwhelming preliminary confidence on ω_c and conclude that they are probably incorrect in their classification on \mathbf{x}_i . Therefore, the voting weights of the classifier not trained with ω_c are reduced in proportion to $P_c(\mathbf{x}_i)$, that is, the weights are lowered proportional to the ensemble's preliminary confidence on ω_c .

$$W_{t:c \notin CTr_t} = W_{t:c \in CTr_t} (1 - P_c) \quad (3.11)$$

In other words, the weight of a classifier that has not been trained on a given class, $W_{t:c \notin CTr_t}$, will be reduced in proportion to the ensemble confidence on that class. Furthermore, if a classifier has not been trained on more than one class then the expression in Equation (3.11) will be applied for each CTr_t .

$$H_{final}(\mathbf{x}_i) = \arg \max_{\omega_c} \sum_{t:h_t(\mathbf{x}_i)=\omega_c} W_t \quad (3.12)$$

The final composite hypothesis is then calculated as the maximum sum of the weights that chose a particular class.

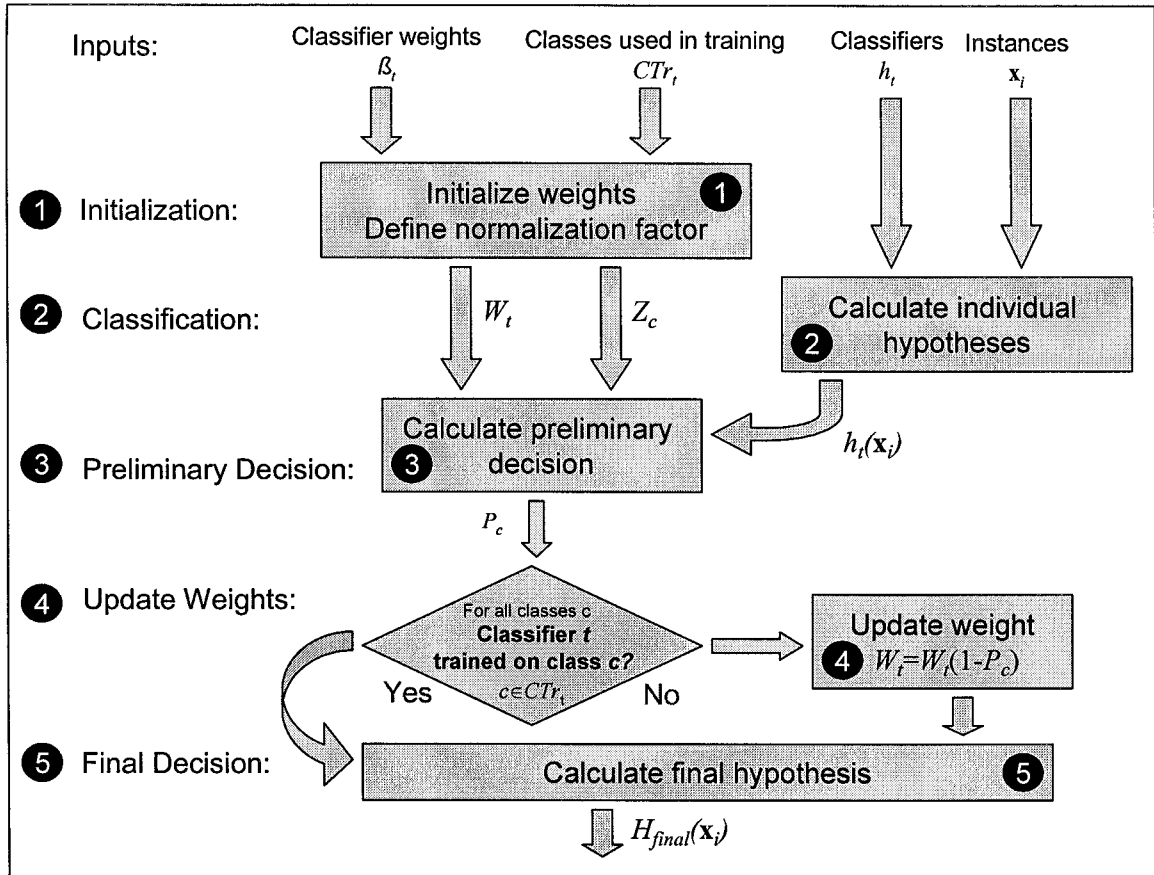


Figure 3.3 - Dynamically weighted voting algorithm.

3.3 Unbalanced Learning

An interesting problem in the incremental learning setting is the issue of unbalanced data, which we define as the discrepancy in the *cardinality* of each dataset used in incremental learning. If one dataset has substantially more data than the other, this can unfairly bias the ensemble decision towards the data with the lower cardinality. This is because, the voting weights of each classifier is determined solely by its performance on its respective training data. Even though the cardinality of a given dataset may be small, the classifier may perform well on its own limited training dataset, and therefore receive a high voting weight. This classifier is most likely to perform poorly – relative to other classifiers

generated with larger cardinality data – on the unseen instances, since it was trained on limited data.

In the absence of any other information, and under the generally valid assumptions that (i) no instance is repeated and (ii) the noise distribution remains relatively unchanged among datasets, it is reasonable to believe that the dataset that has more instances carries more information. Classifiers generated with such data should therefore be weighted more heavily.

It is not unusual to see major discrepancies in the cardinalities of datasets that subsequently become available. Consequently, in any ensemble based learning algorithm that employs a classifier combination scheme, the cardinality of each dataset should be taken into consideration.

An unbalanced data need not be caused simply due to discrepancy among dataset cardinalities, but may also be due to relative cardinalities of individual classes within the training data, a quantity often described as class prior probabilities. While class priors appear conspicuously within the Bayesian setting, they are not as heavily utilized in many other algorithms. Commonly used ensemble combination schemes, such as voting, sum or product based combination, often do not take class priors into consideration [51;52].

Learn++.UD (**Unbalanced Data**), previously published as Learn++.MT2, proposes a set of modifications to address both aspects of unbalanced data described above. While, the approach is described specifically for Learn++, it is nevertheless quite general and can be easily adapted to any ensemble based algorithm such as AdaBoost, or Learn++.NC.

The primary novelty in Learn++.UD is the way by which the voting weights are determined. Learn++.UD attempts to address the unbalanced data problem by keeping track of the number of instances from each class with which each classifier is trained. Similar to Learn++, each classifier is first given a weight based on its own training data performance; however, this weight is later adjusted according to its class conditional weight factor. For each classifier the class conditional weight factor is the ratio of instances from a particular class used for training that classifier, to the number of instances from that class used for training all classifiers thus far within the ensemble. The pseudocode of the entire algorithm is given in Figure 3.4.

For each dataset (\mathfrak{D}^k) that becomes available, the inputs to the algorithm are (i) a sequence of m_k training data instances \mathbf{x}_i along with their correct labels y_i , (ii) a classification algorithm **BaseClassifier**, (iii) an integer T_k specifying the maximum number of classifiers to be generated during the k^{th} training session, and (iv) a variable eN_c is created to hold the current value of N_c which is then updated as the sum of all class- c instances contained in \mathfrak{D}^1 through \mathfrak{D}^k . For the first database ($k = 1$), a data distribution (D_t) – from which training instances will be drawn – is initialized to be uniform, making the probability of any instance being selected equal. The number of instances from each class $c \in \{1, \dots, C\}$ in \mathfrak{D}^1 is stored in N_c . If $k > 1$ then a distribution initialization sequence re-initializes the data distribution (the IF block in Figure 3.4) and updates the existing class-conditional weights according to Equation (3.13).

$$w_{t,c} = w_{t,c} \frac{\sum_{i=1}^{k-1} N_{i,c}}{\sum_{i=1}^k N_{i,c}} \quad (3.13)$$

The algorithm then adds T_k classifiers to the ensemble starting at $t=eT_k+1$ where eT_k is the number of classifiers that currently exist in the ensemble. For each iteration t , the instance distribution, D_t , from the previous iteration is first normalized (step 1).

$$D_t = D_t / \sum_{i=1}^m D_t(i) \quad (3.14)$$

A hypothesis (classifier), h_t , is generated by training on a subset of the dataset, \mathfrak{D}_t^k that is drawn according to D_t (step 2). The error, ε_t , of h_t is then calculated; if $\varepsilon_t > 1/2$, the algorithm deems the current classifier h_t to be too weak, discards it, and returns to step 2, otherwise, calculates the normalized performance p_t (step 3).

$$\varepsilon_t = \sum_{i:h_t(x_i) \neq y_i} D_t(i) \quad (3.15)$$

$$p_t = 1 - 2\varepsilon_t, \quad 0 \leq p_t \leq 1 \quad (3.16)$$

A class conditional weight factor ($w_{t,c}$) is created for each classifier, which is proportional to its classification performance on the entire training data \mathfrak{D}^k (including the portion unused during its training) and the number of class ω_c instances on which the classifier was trained (step 4).

$$w_{t,c} = p_t \frac{n_c}{\sum_{i=1}^k N_{t,c}} \quad (3.17)$$

where n_c is the number of instances from class ω_c used in training. The weighted majority voting algorithm is called to obtain the composite hypothesis, H_t , of the ensemble (step 5).

$$H_t(x_i) = \arg \max_{c \in Y_k} \sum_{t:h_t(x_i)=c} w_{t,c} \quad (3.18)$$

where H_t represents the ensemble decision of the first t hypotheses generated thus far.

The error of the composite hypothesis, E_t is then computed and normalized (step 6).

$$E_t = \sum_{i: H_t(\mathbf{x}_i) \neq y_i} D_t(i) \quad (3.19)$$

The instance distribution D_t is finally updated according to the performance of H_t (step 7) such that the weights of instances correctly classified by H_t are reduced and those that are misclassified are effectively increased.

$$D_{t+1}(i) = D_t \times \begin{cases} B_t, & \text{if } H_t(\mathbf{x}_i) = y_i \\ 1, & \text{otherwise} \end{cases} \quad (3.20)$$

This distribution update rule, based on the performance of the ensemble, ensures that the algorithm chooses instances to train on are difficult to classify, not yet learned, or not previously seen by the ensemble. The algorithm achieves incremental learning, because novel instances introduced by a new dataset are precisely those that are difficult, not yet learned or not yet seen instances. The final hypothesis of the ensemble can be obtained by calling the weighted majority voting algorithm, shown in Equation (3.21).

$$H_{final}(\mathbf{x}_i) = \arg \max_{c \in Y_k} \sum_{t: h_t(\mathbf{x}_i) = c} w_{t,c} \quad (3.21)$$

Algorithm Learn++.UD

Input: For each dataset \mathfrak{D}^k , for $k = 1, 2, \dots, K$

- Training data $\mathfrak{D}^k = \{(x_1, y_1), (x_2, y_2), \dots, (x_{m_k}, y_{m_k})\}$ $y_i \in Y_k \subset \{\omega_1, \dots, \omega_C\}$
- $N_{k,c}$, the number of class- c instances in \mathfrak{D}^k
- Weak learning algorithm **BaseClassifier**.
- Integer T_k , specifying the number of iterations.

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

Initialize $D_1(i) = 1/m_k$, $eT_1 = 0$, $i = 1, \dots, m_k$

IF $k > 1$, Go to Step 5, evaluate current ensemble on the new dataset \mathfrak{D}^k ,

- Update D_t and current number of classifiers $eT_k = \sum_{j=1}^{k-1} T_j$.

- Update $w_{t,c} = w_{t,c} \frac{\sum_{i=1}^{k-1} N_{i,c}}{\sum_{i=1}^k N_{i,c}}$, $t = 1, \dots, eT_k$, $c = 1, \dots, C$

Do for $t = eT_k + 1, eT_k + 2, \dots, eT_k + T_k$

1. Set $D_t = D_t / \sum_{i=1}^m D_t(i)$ so that D_t is a distribution.

2. Call **BaseClassifier** providing it with a subset (\mathfrak{D}_t^k) of \mathfrak{D}^k randomly chosen according to D_t .

3. Obtain a hypothesis $h_t : X \rightarrow Y$, and compute the error $\varepsilon_t = \sum_{i: h_t(x_i) \neq y_i} D_t(i)$

If $\varepsilon_t > 1/2$, discard h_t and go to step 2. Otherwise, compute the normalized performance $p_t = 1 - 2\varepsilon_t$, $0 \leq p_t \leq 1$.

4. Compute the class specific weight as

$$w_{t,c} = p_t \frac{n_c}{\sum_{i=1}^k N_{i,c}}, c = 1, \dots, C$$

where n_c is the number of class- c instances in \mathfrak{D}_t^k

5. Call weighted majority voting to obtain the composite hypothesis

$$H_t(\mathbf{x}_i) = \arg \max_{c \in Y_k} \sum_{t: h_t(\mathbf{x}_i) = c} w_{t,c}$$

6. Compute the error of the composite hypothesis

$$E_t = \sum_{i: H_t(\mathbf{x}_i) \neq y_i} D_t(i)$$

7. Set $B_t = E_t / (1 - E_t)$, $0 < B_t < 1$, and update the instance weights:

$$D_{t+1}(i) = D_t(i) \times \begin{cases} B_t, & \text{if } H_t(\mathbf{x}_i) = y_i \\ 1, & \text{otherwise} \end{cases}$$

Call weighted majority voting to obtain the final hypothesis.

$$H_{final}(\mathbf{x}_i) = \arg \max_{c \in Y_k} \sum_{t: h_t(\mathbf{x}_i) = c} w_{t,c}$$

Figure 3.4 - Learn++.UD algorithm.

3.4 *Problems with Current Approaches*

Although Learn++, Learn++.NC, and Learn++.UD perform well on a variety of incremental learning problems, they all have certain issues inherent in their design, which prevent them from learning properly under specific circumstances. Learn++ suffers severely from the outvoting problem when required to learn new classes. Learn++.NC uses a dynamic weight update rule that allows the algorithm to overcome the outvoting problem. However, there are two known incremental learning scenarios under which Learn++.NC performs poorly, 1) problems where a dataset will simultaneously introduce novel class information and lack information from previously learned classes; and 2) problems which feature severely unbalanced datasets. Learn++.UD is designed to perform well when data between datasets is unbalanced. However, the algorithm is not designed to handle unbalanced data within datasets, nor the introduction of new classes. In Chapter 4 we introduce the BEAST algorithm, which combines the concepts from Learn++.NC and Learn++.UD with some other novel voting techniques. The goal is to create one algorithm that will perform well on a broad spectrum of incremental learning problems.

CHAPTER 4

THE **BEAST** ALGORITHM

Learn++.NC was developed as a logical solution to the out-voting problem inherent in Learn++. Learn++.NC introduced the concept of dynamically adjusting classifier voting weights based on the hypotheses of other classifiers. In other words, each classifier's weight is dependent on the decisions of other classifiers. This novel approach can be extended to optimally combine classifiers by incorporating information inferred from previously trained classifiers. Several methods were developed to help reduce the adverse effects commonly experienced in incremental learning problems. Each method was designed to reduce the effects commonly occurring in unbalanced data problems. Unfortunately, many of these methods exhibited adverse effects when used on a wider range of unbalanced data problems, resulting in instabilities of the algorithm. The BEAST algorithm provides a strategic combination of these methods that performs consistently well over a broad range of incremental learning applications. This chapter is broken down to first describe the individual components that are found in the BEAST algorithm, followed by overall look and explanation of the algorithm.

4.1 *Different Elements of the BEAST Algorithm*

As mentioned, BEAST is composed of several novel methods which we will discuss in detail. The motivation behind each method is explained as to how and why they work.

4.1.1 Class Specific Weights

Learn++.UD introduced a method of adjusting classifier weight values based on the relative amount of data on which they have been trained. The weight for class c and classifier t is then calculated as follows.

$$w_{t,c} = p_t \frac{n_c}{\sum_{i=1}^k N_{i,c}} \quad (4.1)$$

where

$$p_t = 1 - 2\varepsilon_t, \quad 0 \leq p_t \leq 1 \quad (4.2)$$

In Equations (4.1) and (4.2) n_c represents the number of class- c instances in the training data \mathfrak{D}_t^k , ε_t is the error on classifier t calculated in Equation (3.2) and p_t is the performance of classifier t . AdaBoost, Learn++, and Learn++.NC calculate the performance of classifier t according to equation (4.3), which, instead of ranging from 0 to 1 ranges from 0 to infinity, this property in and of itself greatly reduces the outvoting problem by preventing classifiers that perform exceptionally well on the training data to be assigned voting weights that are difficult and sometimes impossible to overcome.

$$p_t = \log \frac{1}{\beta_t}, \quad \text{where } \beta_t = \varepsilon_t / (1 - \varepsilon_t) \quad (4.3)$$

Analyzing the impact of Equation (4.1), we observe that the weights of classifiers trained with more instances of a specific class are higher, allowing classifiers with more “experience” on particular classes to be weighted higher when choosing that class. This process helps when the number of instances from each class in any given dataset are correlated to the true prior probabilities of that environment. Thus, if datasets are unbalanced in a different manner, from one to the next, this weighting method will fail. Therefore, this cardinality based weighting method should only be used by itself for non-

incremental boosting, or incremental learning where each dataset’s distribution is correlated to the true distribution of the environment.

4.1.2 Preliminary Confidence and Normalization

Learn++.NC introduces the concept of creating a preliminary confidence on each class to be used to modify the overall decision. This preliminary confidence on class c is defined as the sum of the weights from classifiers that choose class ω_c , divided by the sum of the weights of all classifiers trained on class c . Recall:

$$P_c = \frac{\sum_{t:h_t(x_i)=c} w_t}{Z_c}, \quad \text{where } Z_c = \sum_{t:c \in CTr_t} w_t \quad (4.4)$$

The preliminary confidence in equation (4.4) can be represented as the ensemble’s confidence on selecting each class. This information can then be used in a number of ways to modify how the algorithm proceeds with its final decision.

4.1.3 Preliminary Confidence Transfer Function

Arguably the key method in the BEAST algorithm for handling unbalanced data is hidden inside the preliminary confidence transfer function. This function could technically be any monotonically increasing function, whose output is bounded between 0 and 1 when the input is bounded between 0 and 1.

$$\bar{P} = f(P) \quad (4.5)$$

In this work, the transfer function has been designed to compensate for unbalanced data. The idea is to make it more difficult to classify data that is abundantly available in the training set, hence the design of the following transfer function.

$$f(P) = P^{N_c / \min(N)} \quad (4.6)$$

Where N_c is the number of instances from class c and $\min(N)$ is the non-zero number of instances from the least occurring class. Figure 4.1 shows what these transfer functions would look like for the following ratios of $N_c/\min(N)$, 1, 2, 4, and 8.

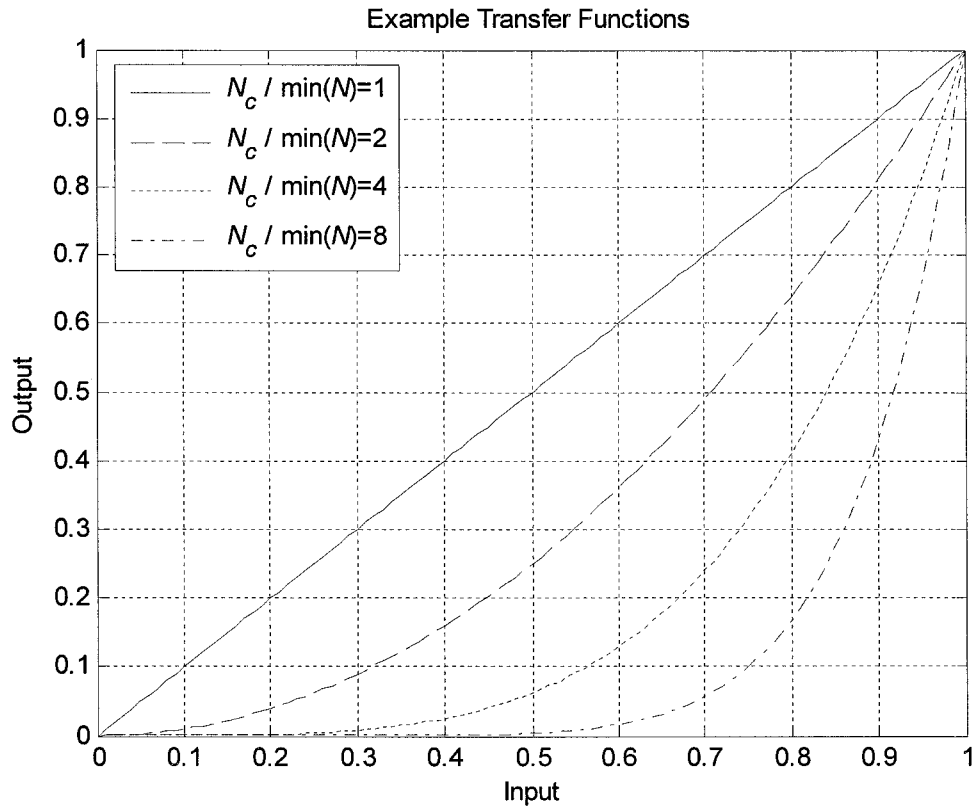


Figure 4.1 – Example of transfer functions generated by equation (4.6)

This transfer function is discussed in more detail in the later in this chapter.

4.1.4 Sub Ensembles and Decision Update

Until now, all known ensemble methods have combined classifiers as one ensemble. We now propose a process by where classifiers generated on each dataset are combined separately as sub-ensembles and then the sub-ensembles are combined subsequently to create the entire ensemble. Effectively, creating sub-ensembles adds a second layer to the classifier combination process. Since classifiers from each dataset become a sub-

ensemble, Equations (4.4) and (4.6) need to be modified so that they can handle each sub-ensemble trained on the k^{th} dataset.

$$P_{k,c} = \frac{\sum_{t: h_t(x_i)=c} w_{t,c}}{Z_{k,c}}, \quad \text{for } t = eT_k + 1, \dots, eT_k + T_k \quad (4.7)$$

$$Z_{k,c} = \sum_{t=eT_k+1}^{eT_k+T_k} w_{t,c} \quad (4.8)$$

$$\bar{P}_{k,c} = f(P_{k,c}) = P_{k,c}^{N_{k,c}/\min(N_k)} \quad (4.9)$$

In Equations (4.7) and (4.8), eT_k represents the number of classifiers that exist in the ensemble upon the introduction of \mathfrak{D}_k . Thus each sub-ensemble will have a different preliminary confidence, normalization factor, as well as a different set of class specific transfer functions. Furthermore, the preliminary confidence of each sub-ensemble is normalized again according to the relative amount of data they have been trained with, shown in Equation (4.10).

$$\bar{\bar{P}}_{k,c} = \bar{P}_{k,c} \frac{N_{k,c}}{\sum_{j=1}^K N_{j,c}} \quad (4.10)$$

The functionality of equation (4.10) is very similar to equation (4.1), used for class specific weights, and also fails under the same circumstances as discussed in Section 4.1.1.

These four aforementioned methods by themselves are biased towards certain types of problems; however, when properly combined they balance each other out to create a robust algorithm that is suited for a wide variety of incremental learned problems.

4.2 BEAST

For the purposes of this analysis, the BEAST algorithm is split into two parts. The first part of the BEAST algorithm is very similar to the Learn++.UD algorithm. The second part of the algorithm contains novel classifier combination methods for computing the **Ensemble Decision**, which will be referred to as BEAST-ED and described in section 4.3.

The following is description of how the first part of the BEAST algorithm operates; it is also described in pseudocode in Figure 4.2. For each dataset \mathfrak{D}^k that becomes available, the inputs to the algorithm are (i) a sequence of m_k training data instances \mathbf{x}_i along with their correct class labels y_i , (ii) a classification algorithm **BaseClassifier**, (iii) $N_{k,c}$, the number of instances, from each class, contained in \mathfrak{D}^k , (iv) and an integer T_k specifying the number of classifiers to be generated using \mathfrak{D}^k . For the first database, the data distribution, D_t , is initialized to be uniform, making the probability of any instance being selected equal. If $k > 1$ the distribution D_t is re-initialized in the same manner in which it is updated (step 5-7).

The algorithm then adds T_k classifiers to the ensemble starting at $t=eT_k+1$ where eT_k is the number of classifiers that exist in the ensemble when \mathfrak{D}^k is made available. For each iteration t , the instance distribution, D_t , from the previous iteration is normalized (step 1). A hypothesis (classifier), h_t , is generated by training on a subset of the dataset, \mathfrak{D}_t^k , drawn according to D_t (step 2). The error, ϵ_t , of h_t is calculated; if $\epsilon_t > 1/2$, the algorithm rejects classifier h_t and returns to step 2, otherwise, calculates the normalized performance p_t (step 3).

The BEAST Algorithm

Input: For each dataset $\mathfrak{D}^k, k = 1, 2, \dots, K$

- Training data $\mathfrak{D}^k = \{(x_1, y_1), (x_2, y_2), \dots, (x_{m_k}, y_{m_k})\}$ $y_i \in Y_k \subset \{\omega_1, \dots, \omega_C\}$
- Learning algorithm **BaseClassifier**.
- $N_{k,c}$, the number of class-c instances in \mathfrak{D}^k
- Integer T_k , specifying the number of **BaseClassifiers** to create using \mathfrak{D}^k .

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

Initialize $D_1(i) = 1/m_k, eT_k = \sum_{j=1}^{k-1} T_j, i = 1, \dots, m_k$

IF $k > 1$, Go to Step 5, evaluate current ensemble on the new dataset \mathfrak{D}^k ,

- Update D_t and current number of classifiers $eT_k = \sum_{j=1}^{k-1} T_j$.

- Update $w_{t,c} = w_{t,c} \frac{\sum_{i=1}^{k-1} N_{i,c}}{\sum_{i=1}^k N_{i,c}}, t = 1, \dots, eT_k, c = 1, \dots, C$

Do for $t = eT_k + 1, eT_k + 2, \dots, eT_k + T_k$:

1. Set $D_t(i) = D_t(i) / \sum_{i=1}^m D_t(i)$ so that D_t is a distribution.
2. Call **BaseClassifier**, providing it with $\mathfrak{D}_t^k \in \mathfrak{D}^k$, drawn according to D_t .
3. Obtain a hypothesis $h_t: X \rightarrow Y$, and calculate its the error

$$\varepsilon_t = \sum_{i: h_t(x_i) \neq y_i} D_t(i).$$

If $\varepsilon_t > 1/2$, discard h_t and go to step 2. Otherwise, compute the normalized performance $p_t = 1 - 2\varepsilon_t, 0 \leq p_t \leq 1$.

4. Compute the class specific weight as

$$w_{t,c} = p_t \frac{n_c}{\sum_{i=1}^k N_{i,c}}, c = 1, \dots, C$$

where n_c is the number of class-c instances in \mathfrak{D}_t^k

5. Call **BEAST-ED** to obtain the sub-ensemble composite hypothesis H_t .
6. Compute the error of the composite hypothesis $E_t = \sum_{i: H_t(x_i) \neq y_i} D_t(i)$
7. Set $B_t = E_t / (1 - E_t)$, and update the instance weights:

$$D_{t+1}(i) = D_t(i) \times \begin{cases} B_t(i), & \text{if } H_t(x_i) = y_i \\ 1, & \text{otherwise} \end{cases}$$

Call **BEAST-ED** to obtain the final hypothesis, H_{final}

Figure 4.2 – The BEAST algorithm.

A class conditional weight factor, $w_{t,c}$, is created for classifier t , which is proportional to its classification performance on the entire training data \mathfrak{D}^k and the number of class ω_c instances on which the classifier was trained (step 4). The BEAST-ED algorithm is called to obtain the sub-ensemble composite hypothesis, H_t (step 5). The error of the composite hypothesis, E_t is computed and normalized (step 6). The instance distribution D_t is finally updated according to the performance of H_t (step 7) such that the weights of instances correctly classified by H_t are reduced effectively increasing the weights of the misclassified instances.

One of the major differences from previous work, in step (5)-(7), the instance distribution weights are updated by the performance of the sub-ensemble, as opposed to the entire ensemble. Each sub-ensemble then focuses on maximizing its performance on its own training data. In past algorithms, such as Learn++, instance weights are updated such that misclassified instances from previous training are given higher weights, thus making the weighting of instances from one dataset subject to the performance of classifiers generated on another dataset. Although this may be desirable when there are no methods to handle unbalanced data, its effects are undesirable when such methods are in place.

4.3 *BEAST-ED*

The heart of the BEAST algorithm is contained in the methods used to combine individual classifier decisions into one ensemble decision. The pseudocode for BEAST-ED can be found in Figure 4.3.

The inputs to BEAST-ED are (i) a sequence of n instances $[\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n]$ to be classified, (ii) h_t the individual classifier hypothesis, (iii) $w_{t,c}$ the hypothesis weight

matrix, (iv) $N_{k,c}$ the number of instances from each class contained in \mathfrak{D}_k , and (v) T_k the number of classifiers created using \mathfrak{D}_k . Note: the set of classifiers trained on \mathfrak{D}_k will be commonly referred to as sub-ensemble k .

Initialization) Calculate a normalization factor, $Z_{k,c}$, for each class ω_c and each sub-ensemble k as the as the sum of the class ω_c weights of classifiers in sub-ensemble k .

$$Z_{k,c} = \sum_{t=eT_k}^{eT_k+T_k} w_{t,c} \quad (4.11)$$

where eT_k can be calculate using T_k in Equation (4.12)

$$eT_k = \sum_{j=1}^{k-1} T_j \quad (4.12)$$

for each instance \mathbf{x}_i , for $i = 1, 2, \dots, n$

Step 1) Calculate the preliminary confidences of each sub-ensemble k for each class ω_c as the sum of the classifiers from sub-ensemble k who chose class ω_c for \mathbf{x}_i , divided by the corresponding normalization factor found in Equation (4.11).

$$P_{k,c} = \frac{\sum_{t:h_t(\mathbf{x}_i)=c} w_{t,c}}{Z_{k,c}} \quad (4.13)$$

Step 2) Apply the transfer function to each preliminary confidence calculated in Step 1). As mentioned in Section 4.1.3, the transfer function can be any function $f(P_{k,c})$; however, the following transfer function is used throughout the remainder of this work.

$$\bar{P}_{k,c} = f(P_{k,c}) = (P_{k,c})^{N_{k,c}/\min(N_k)} \quad (4.14)$$

Equation (4.14) reduces the confidence on classes where data is abundantly available in training. This procedure assumes that classifiers will be biased towards classes which appeared more often in training.

Step 3) Update sub-ensemble confidence $\bar{P}_{k,c}$ according to how many instances from class ω_c sub-ensemble k was trained on relative to the number of instances from class ω_c that all sub ensembles have been trained on.

$$\bar{\bar{P}}_{k,c} = \bar{P}_{k,c} \frac{N_{k,c}}{\sum_{j=1}^K N_{j,c}} \quad (4.15)$$

Equation (4.1.5) increases the weights of sub-ensembles trained with more instances of a specific class, allowing sub-ensembles with more “experience” on class c to be weighted higher when choosing that class.

Step 4) Calculate the ensemble confidence on class ω_c as the sum of sub-ensemble decisions on class ω_c . The ensemble decision, or final hypothesis, on x_i then becomes the class corresponding to the highest ensemble confidence value.

$$H_{final}(x_i) = \arg \max_c \sum_{k=1}^K \bar{\bar{P}}_{k,c} \quad (4.16)$$

Steps (2) and (3) could potentially be combined into one step, as they both act as modifiers to the sub-ensemble confidences. However, they are kept separate to easily allow future modification to the transfer function and to separate the functions of each step.

Algorithm BEAST-ED**Input:**

- Sequence of n instances $[\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n]$.
- Classifiers h_t .
- Hypothesis weight matrix, $w_{t,c}$.
- $N_{k,c}$, the number of class- c instances in from \mathfrak{D}^k
- Integer T_k , specifying the number of **BaseClassifiers** created using \mathfrak{D}^k .

$$\text{Calculate } eT_k = \sum_{j=1}^{k-1} T_j$$

Create normalization factor, Z , for each class for each sub-ensemble

$$Z_{k,c} = \sum_{t=eT_k}^{eT_k+T_k} w_{t,c}, \quad \text{for } c = 1, \dots, C \text{ and } k = 1, \dots, K$$

Do for each $i=1, 2, \dots, n$

Obtain preliminary confidence

$$1. P_{k,c} = \frac{\sum_{t: h_t(x_i)=c} w_{t,c}}{Z_{k,c}}, \quad \text{for } t = eT_{k-1}, \dots, eT_k \quad c = 1, \dots, C \text{ and } k = 1, \dots, K$$

Apply transfer function to decision

$$2. \bar{P}_{k,c} = f(P_{k,c}) = (P_{k,c})^{N_{k,c}/\min(N_k)}, \quad \text{for } c = 1, \dots, C \text{ and } k = 1, \dots, K$$

where $\min(N_k) \neq 0$

Update the sub-ensemble decisions

$$3. \bar{\bar{P}}_{k,c} = \bar{P}_{k,c} \frac{N_{k,c}}{\sum_{j=1}^K N_{j,c}}, \quad \text{for } c = 1, \dots, C \text{ and } k = 1, \dots, K$$

Compute the ensemble decision

$$4. H_{final}(x_i) = \arg \max_c \sum_{k=1}^K \bar{\bar{P}}_{k,c}$$

Figure 4.3 – BEAST-ED algorithm.

As discussed throughout section 4.1, the properties of every step in the ensemble decision process, when separated from each other, work only on a limited number of cases. This is due to individual methods overcompensating for the problems they are meant to solve. However, when combined correctly these inconsistent methods work together to provide consistent results on a wealth of incremental learning problems. These problems can range from the introduction and removal of classes to unbalanced

data, within datasets and/or between datasets. Please refer to Appendix A, which provides an experimental analysis of BEAST with a particular focus on how the preliminary confidence is created and adjusted using BEAST-ED.

CHAPTER 5

IMPLEMENTATION RESULTS

In Chapter 4 the four algorithms capable of incremental learning, Learn++, Learn++.NC, Learn++.UD, and BEAST were described in detail. In addition to their respective functionality, the circumstances under which each algorithm is designed to perform well was described. To summarize: Learn++ is designed for incremental learning, in particular the introduction of novel classes. However, Learn++ suffers from the outvoting problem which can experience performance drops under certain circumstances. Learn++.NC, based off of Learn++, is designed to solve the outvoting problem, and is specifically intended for incremental learning problems that introduce new classes. However, Learn++.NC may also experience performance drops if the new training data simultaneously introduce new classes and remove previously learned classes. Learn++.UD is designed to handle the unbalanced data problems that may occur during incremental learning. However, the algorithm only handles unbalanced data between datasets, not unbalanced data between classes in one dataset. BEAST combines the original concept of Learn++, the novel concepts of Learn++.NC and Learn++.UD with additional concepts to handle the shortcomings mentioned above. The BEAST algorithm is designed to address the cases which both introduce new classes and remove previously seen ones. It also uses the weight adjustment transfer function to handle unbalanced data problems between classes in each dataset. The simulation results for all four aforementioned algorithms are presented in this chapter – testing each algorithm on a variety of synthetic and real world incremental learning problems. The ensemble approach to the incremental learning problem requires an integer value, T_k , specifying the

number of classifiers to generate on \mathcal{D}_k . Results in this chapter are based on a predetermined number of classifiers for each application. In some cases, the number of classifiers needed to incrementally learn an additional dataset varies between trials. One solution in determining T_k is to generate an excessive number of classifiers and use a validation dataset to find the optimal number of classifiers to retain for each trial. Results using this method can be found in Appendix B.

5.1 *Organization and Motivation of Simulations*

5.1.1 Synthetic Incremental Learning Problems

It is often helpful to be able to visualize both the incremental learning problem and the way in which a particular algorithm solves such a problem. This is often impossible on real world problems whose feature spaces span larger than 3-dimensional spaces. Thus it is necessary to create synthetic problems in a 2-D feature space which allow us to visually assess the ability of each algorithm. Three such synthetic databases were generated for this purpose. The first synthetic database presented in Section (5.2.1) contains four classes, each with a Gaussian distribution. This database introduces two of the four classes during incremental training in order to test the algorithm's ability to incrementally learn novel classes. The second database, in Section (5.2.2), again contains four classes, each with a Gaussian distribution. All four classes are available in every training dataset; however, each dataset is significantly unbalanced. The third database, in Section (5.2.3), contains four spiral shaped classes whose variance increases with the radius. The algorithms are incrementally trained with no more than three classes in any dataset; this emulates the addition of novel classes while simultaneously removing all instances from previously seen classes.

One key benefit in using synthetic data is the knowledge of the actual data distributions. With known distributions, the posterior probability can be calculated, allowing us to compare the algorithms to the Bayes classifier. This will be discussed further in section 5.2.

5.1.2 Experimental Incremental Learning Problems

As helpful as synthetic problems are in visualizing how an algorithm performs, it is essential to also benchmark the algorithms on real world data. Section 6.3 shows the results of experiments on two real world databases, one of which is from the UCI machine learning repository, which is widely accepted to be a benchmark database [53].

5.1.3 Presentation of Results

The most important aspect of conducting these simulations is the way the experiments and their associated results are presented. Consequently a great effort was made to provide the most informative presentation of the experimental setup and results. For each experiment the nature of the associated database is discussed along with how it is broken down into training and testing datasets. The data distributions of the training datasets are of particular importance as they are used to characterize an array of unbalanced data problems.

The results generated from each experiment are presented in tabular form. The results are collected for each algorithm at the end of each training session, and include the overall generalization performance and the performance for each class. Second, visual comparisons between the algorithms are shown by plotting the generalization performance of each algorithm as classifiers are subsequently added. Another way of analyzing the performance data is through examining the probability density function

(PDF) of the final performance figures for each algorithm. Such PDFs are estimated using Parzen windows, a nonparametric density estimation technique.

Assume we have n observations of a parameter x ; Parzen windows generates n kernel functions around each x and averages them together to calculate the PDF of parameter x . All results presented in this work use a Gaussian for the kernel function. An example of this technique can be found in Figure 5.1, where the estimated PDF is a solid line, the kernel functions are dotted lines, and the five input observations are dots.

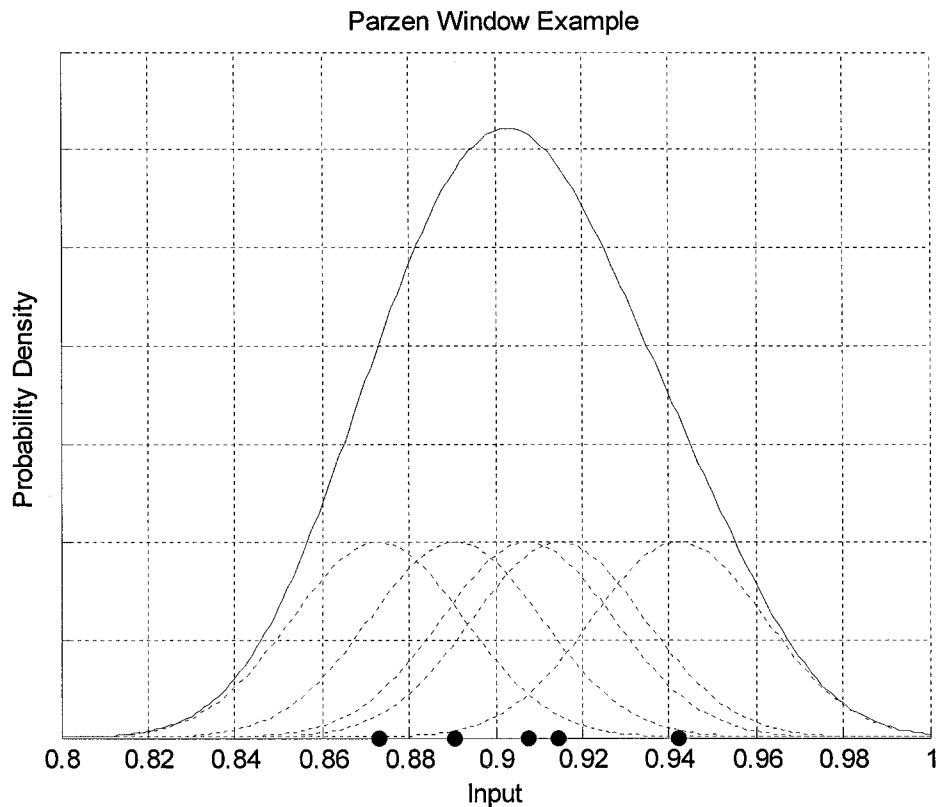


Figure 5.1 – Parzen window example showing the estimated PDF calculated from five observations.

Finally, a set of plots are generated for each algorithm showing the individual class performances as classifiers are added to the ensemble. This visual aid is typically the most informative representation of each algorithm’s behavior. Different

visualizations of the results are beneficial in properly analyzing each algorithm and comparing them against each other. Many of the novel concepts coming from this research are a direct result of properly understanding the behavior of these ensemble methods through various visualization techniques.

5.2 *Simulation Results on Synthetic Databases*

The presentations of results described in Section 5.1.3 are complimented with additional visual presentations. All experiments conducted in this section are in two dimensional environments with known data distributions. Thus, the following graphics aids are added to each experiment.

1) A 3-D plot of the data distribution of the environment which can be shown by plotting the class-conditional likelihood functions, $P(\mathbf{x}|\omega_c)$. Equation (5.1) can be used to calculate the probability density function for a Gaussian class distribution centered at $x_1 = \mu_1$ and $x_2 = \mu_2$ with a uniform variance of σ_2 .

$$P(\mathbf{x}|\omega_c) = \frac{1}{\sigma_c \sqrt{2\pi}} e^{-0.5((x_1 - \mu_{1c})^2 + (x_2 - \mu_{2c})^2) / \sigma_c^2} \quad (5.1)$$

2) 2-D sample plots of the experimental data for each training dataset and the test dataset.
 3) A 3-D plot of the posterior probability, $P(\omega_c|\mathbf{x})$, calculated using the Bayes classifier, equation (5.2).

$$P(\omega_c|\mathbf{x}) = \frac{P(\mathbf{x}|\omega_c) \cdot P(\omega_c)}{\sum_{k=1}^c P(\mathbf{x}|\omega_k) \cdot P(\omega_k)} \quad (5.2)$$

where the prior probability, $P(\omega_c)$ is either known or can be calculated as

$$P(\omega_c) = \frac{N_c}{\sum_{k=1}^C N_k} \quad (5.3)$$

where N_c is the number of instances from class ω_c contained in the training data. In this work all synthetic databases are known to have equal prior probabilities.

4) 2-D plots showing the classification of each algorithm on the entire feature space. This highlights the decision boundaries generated by each algorithm; the classification of the Bayes classifier will also be plotted for comparison.

Note: all results presented in this section are averaged over 40 independent trials.

5.2.1 Synthetic Experiment 1 – Incremental Learning

The first synthetic database is designed to test the algorithm's ability to learn new class information. Table 5.1 shows the information concerning the four Gaussian class distributions, all of which have a uniform variance. Figure 5.2 shows the probability density functions (PDFs) of these four classes.

Table 5.1 – Gaussian distribution information of the first experiment.

	Class 1	Class 2	Class 3	Class 4
μ_1	1	1	-1	0
μ_2	-1	1	0	0
Variance	0.30	0.30	0.30	0.15

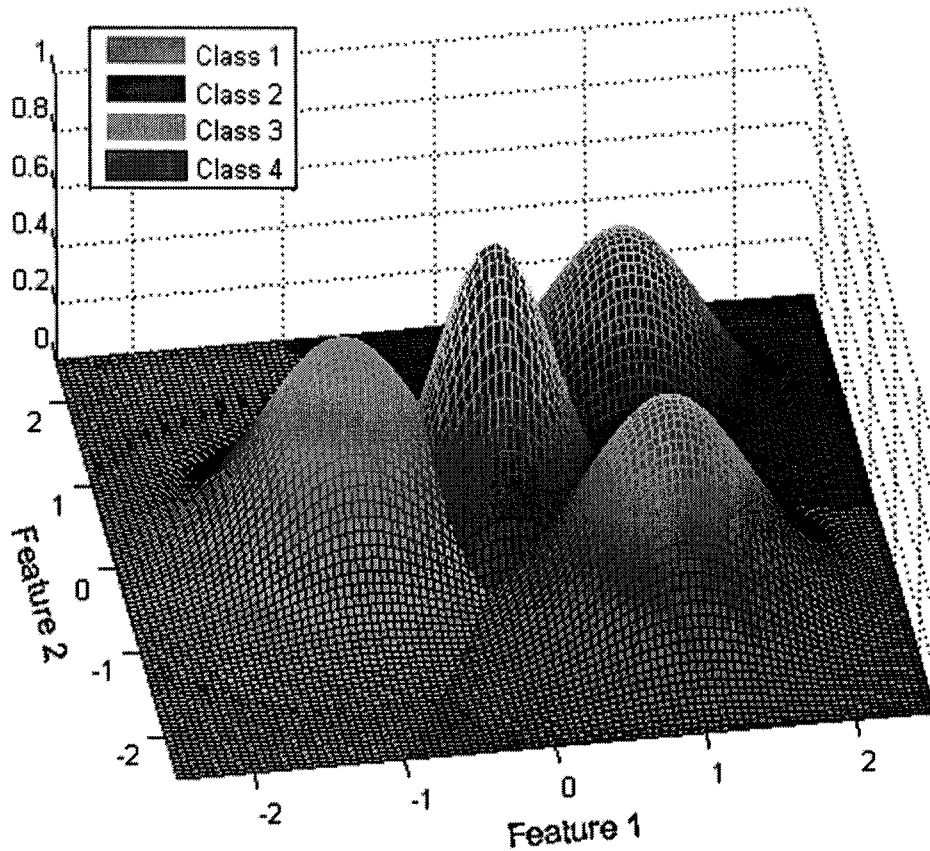


Figure 5.2 – Four PDFs corresponding to the classes described in Table 5.1.

This database was split into three training sets and one test dataset. The first training set, \mathcal{D}_1 , only contains information from classes ω_1 and ω_2 . The second dataset, \mathcal{D}_2 , introduces class ω_3 which is easily separable from the first two classes. The third dataset, \mathcal{D}_3 , introduces a fourth class which overlaps with the first three classes.

Table 5.2 – Instance distribution of the Experiment 1.

	Class 1	Class 2	Class 3	Class 4
\mathcal{D}_1	100	100	0	0
\mathcal{D}_2	50	50	150	0
\mathcal{D}_3	50	50	50	200
Test Data	200	200	200	200

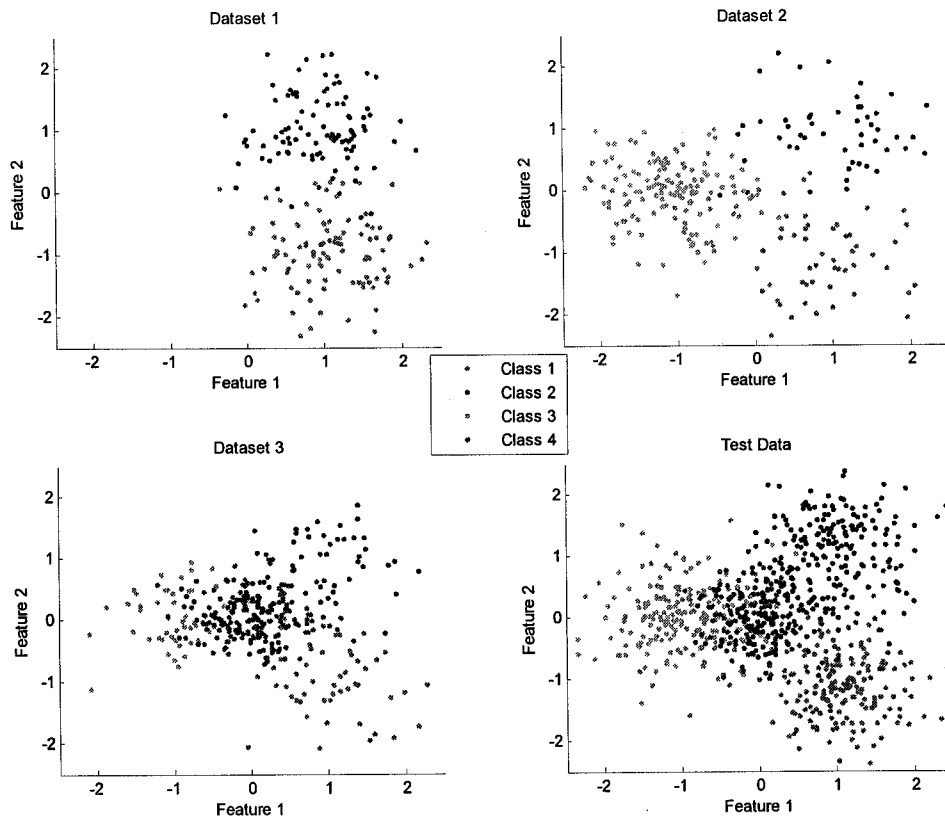


Figure 5.3 – *Example training and testing data from Experiment 1.*

The test is designed to observe the algorithm’s behavior in learning an easily distinguishable new class, and learning a new class that coincides with previously obtained knowledge. The instance based data distribution for this experiment can be found in Table 5.2. Furthermore, Figure 5.3 shows an example of the three training datasets and the test dataset.

Since we know the distribution information, we can also calculate the posterior probability for each class. Figure 5.4 shows the posterior probability plot for the Bayes classifier, the highest posterior probability, given that the prior probabilities of all classes are equal.

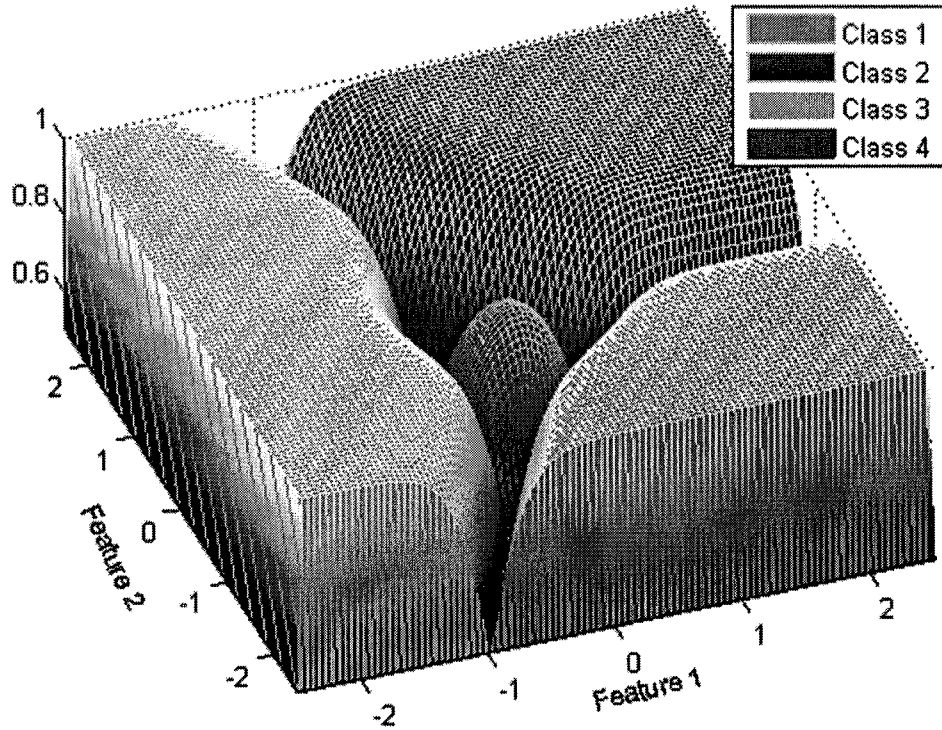


Figure 5.4 – *Posterior probability output of the Bayes classifier over the entire feature space.*

All four algorithms are restricted to generating MLP classifiers with identical network architectures, 0.025 error goal and 30 hidden layer nodes. On each training dataset the algorithms are allowed to create 5 classifiers, 15 classifiers total. Table 5.3 shows the average performance results for this test along with the 95% confidence interval on the generalization performance.

Table 5.3 – Generalization performance from Experiment 1.

		Class 1	Class 2	Class 3	Class 4	Gen. \pm CI
Learn++	Φ_1	96.1%	95.4%	-	-	48.1% \pm 0.3%
	Φ_2	95.3%	94.8%	77.0%	-	66.8% \pm 1.9%
	Φ_3	95.3%	94.6%	88.3%	16.8%	73.7% \pm 1.2%
Learn++.NC	Φ_1	95.5%	96.2%	-	-	48.0% \pm 1.1%
	Φ_2	92.6%	92.9%	97.7%	-	70.8% \pm 0.3%
	Φ_3	85.6%	85.5%	67.7%	89.1	82.0% \pm 0.5%
Learn++.UD	Φ_1	95.8%	95.6%	-	-	47.8% \pm 0.2%
	Φ_2	92.0%	92.4%	97.6%	-	70.5% \pm 0.4%
	Φ_3	85.4%	84.7%	71.1%	89.3%	82.6% \pm 0.5%
BEAST	Φ_1	96.0%	96.2%	-	-	48.0% \pm 0.1%
	Φ_2	94.5%	94.9%	95.5%	-	71.2% \pm 0.2%
	Φ_3	90.5%	90.7%	82.6%	70.3%	83.5% \pm 0.6%
Bayes Classifier						86.1%

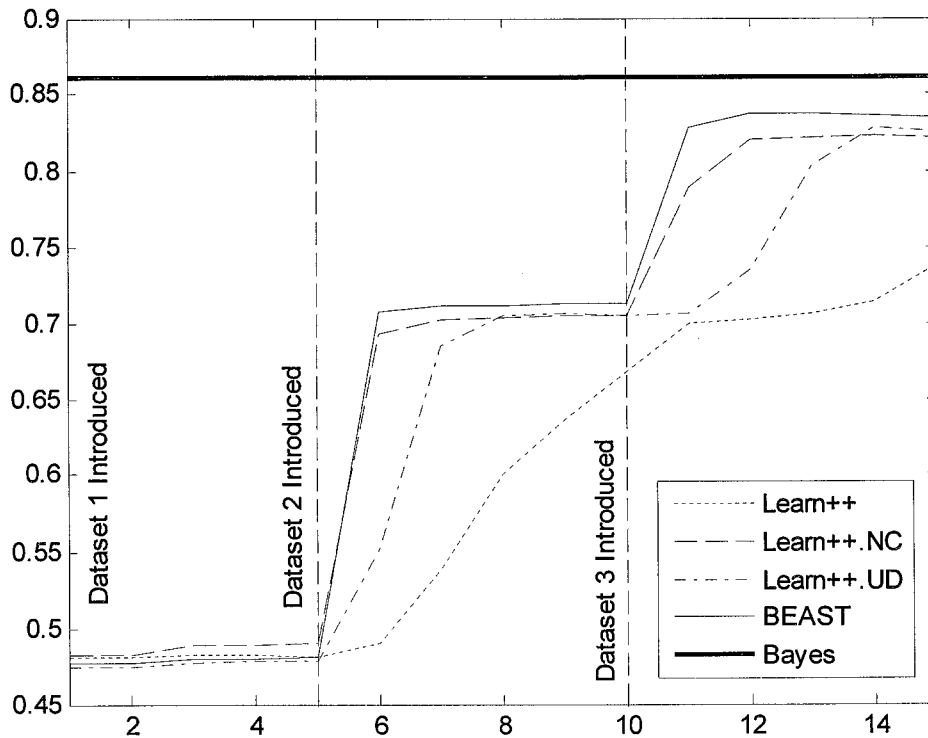


Figure 5.5 – Generalization performance vs. number of classifiers.

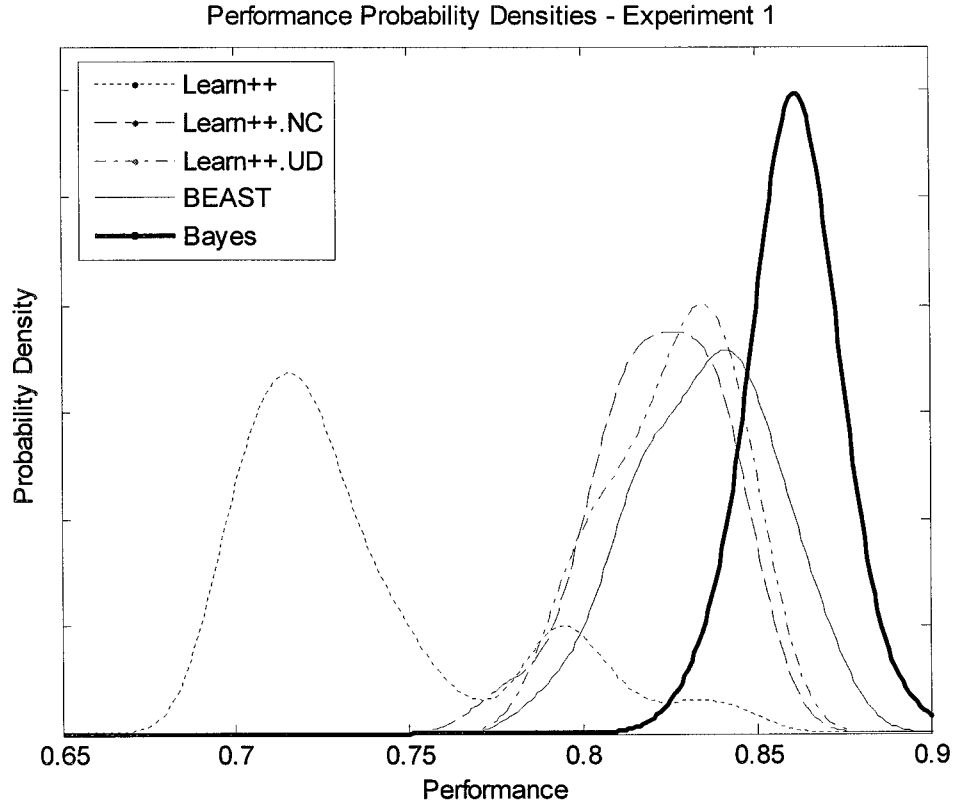


Figure 5.6 – *Theoretical performance probability density functions.*

Table 5.3 and Figure 5.5 indicate that the final performances of the four algorithms are very similar, with the exception of Learn++. Learn++ is unable to learn the last class introduced with the number of classifiers provided. However, Learn++ also retains more information on previously seen classes than the other three algorithms, implying that the algorithm is more robust in nature than the others. Learn++.NC and Learn++.UD are least stable in behavior, as seen by the drop in performance on previously seen data. BEAST is designed to fall on the optimal point of the stability-plasticity spectrum, where it seeks to be stable with previously learned information and plastic on newly introduced information. This is only possible to the extent that the newly introduced information does not conflict with previously learned information,

under which circumstances the algorithm maintains a balance between stability and plasticity.

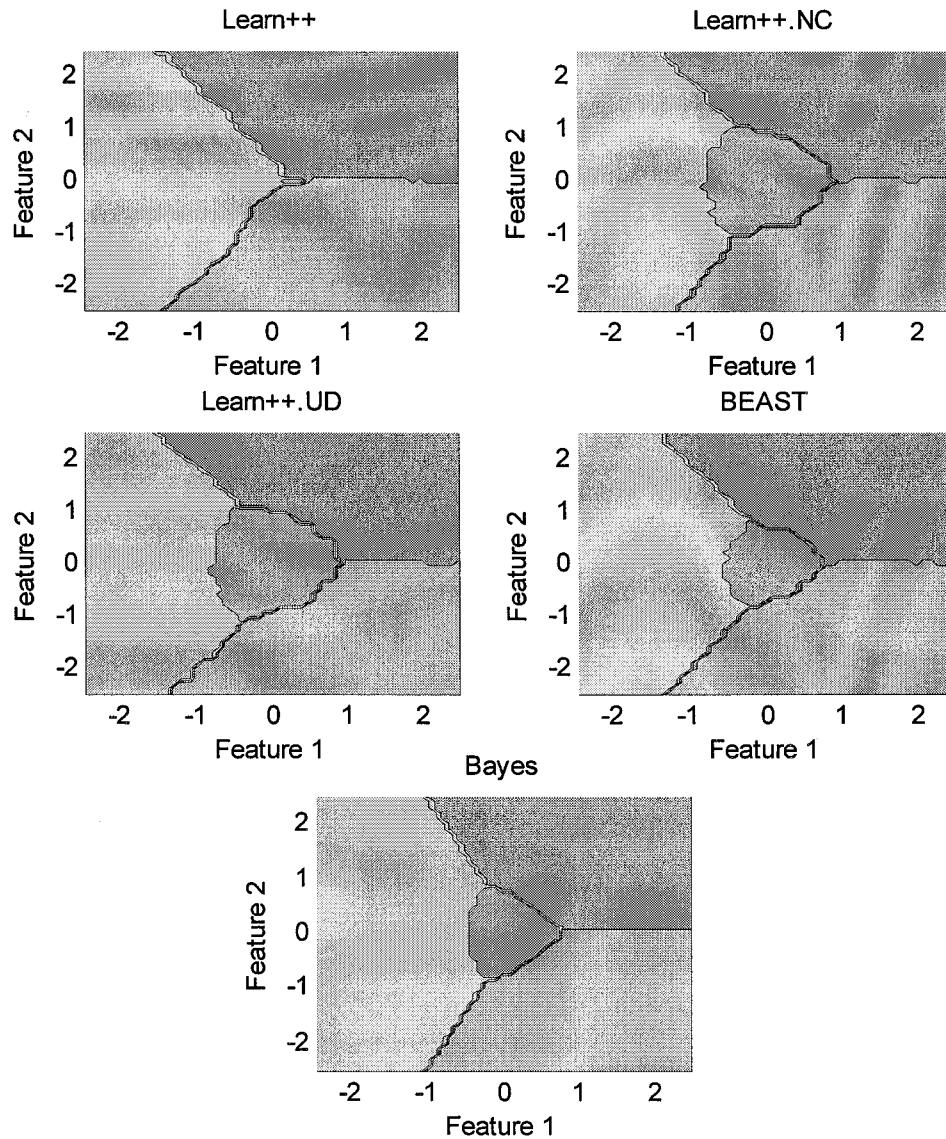


Figure 5.7 – *Decision boundaries over the entire feature space.*

Figure 5.7 shows the decision boundaries of each algorithm along with the decision boundary of the theoretically best performing classifier. Learn++.NC and Learn++.UD are very similar to the Bayes classifier; however, BEAST approximates the Bayes classifier most closely. To further analyze how each algorithm handles learning new

classes, it is helpful to investigate the performance on each class separately as new classifiers are generated.

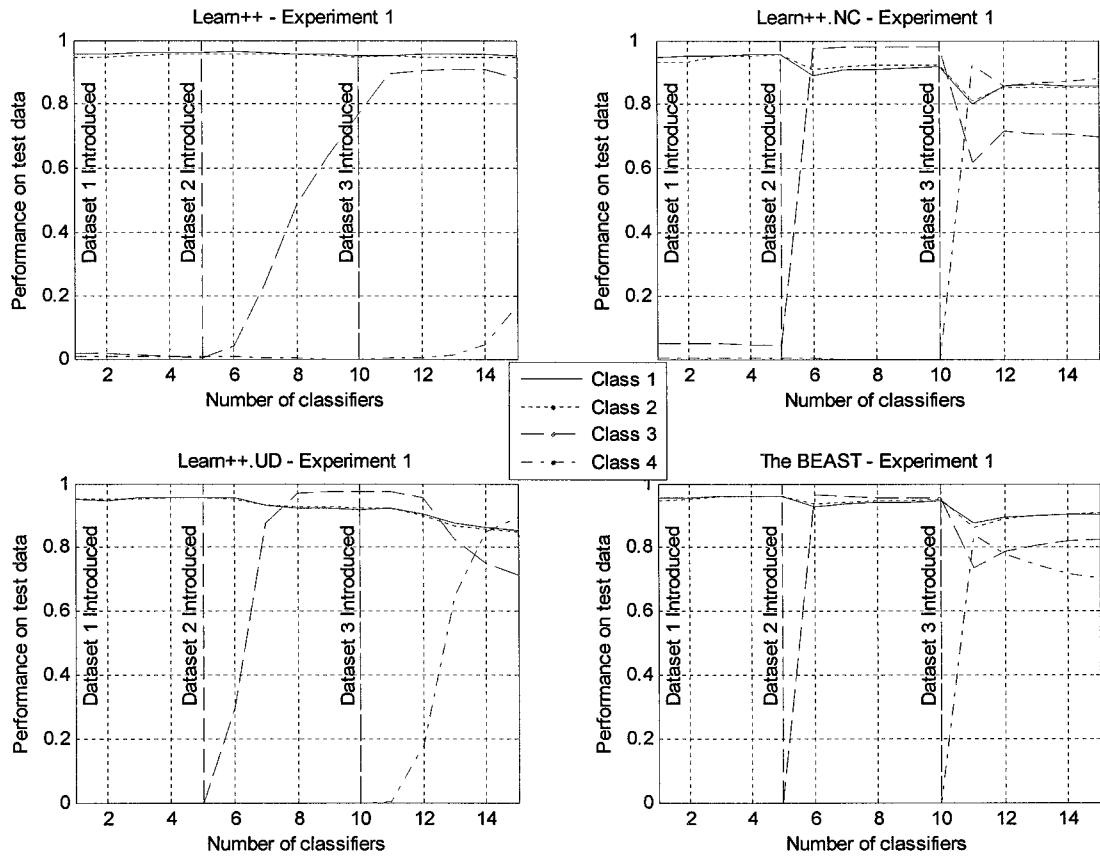


Figure 5.8 – *Class specific generalization performances from experiment 1.*

Figure 5.8 shows all four algorithm’s performance on each class. Again, it is evident that Learn++ is the most stable as it is very hesitant to allow newly generated classifiers to outvote the existing ensemble. Unfortunately, in many cases this will prevent the algorithm from learning properly. On the other hand, Learn++.NC and Learn++.UD are much more plastic and learn newly introduced classes much faster, especially in the case of Learn++.NC which is designed for this scenario. However, they both learn the new information at the expense of sacrificing previously learned information. Although these algorithms can learn extremely fast they are somewhat

volatile in nature. BEAST also learns extremely fast but it is not as volatile as its predecessors.

The second dataset introduces a new class that is easily separable from the previously learned classes. Under these circumstances BEAST and Learn++.NC learn the new class immediately and the other algorithms soon follow. The third dataset introduces a new class that conflicts with all previously learned classes. Again, Learn++.NC and BEAST learn the new information immediately. However, in this case the performance on the other classes drops after the first classifier is trained on the third dataset. Learn++.NC continues to perform better on the new class and worse on previously learned classes as classifiers are added. BEAST, strives to maximize the performance on all classes, and consequently performs only slightly worse on the newly introduced data in order to perform better on the other three classes.

5.2.2 Synthetic Experiment 2 – Unbalanced Data

The second synthetic experiment is designed to test the algorithms ability to continually learn information when presented in an unbalanced manner. Table 5.4 shows the information concerning the four Gaussian class distributions, all of which have a uniform variance. Figure 5.10 shows the probability density functions (PDF's) of these four classes.

Table 5.4 - Gaussian distribution information of the second experiment.

	Class 1	Class 2	Class 3	Class 4
μ_1	1	1	-1	-1
μ_2	-1	1	-1	1
Variance	0.35	0.35	0.35	0.35

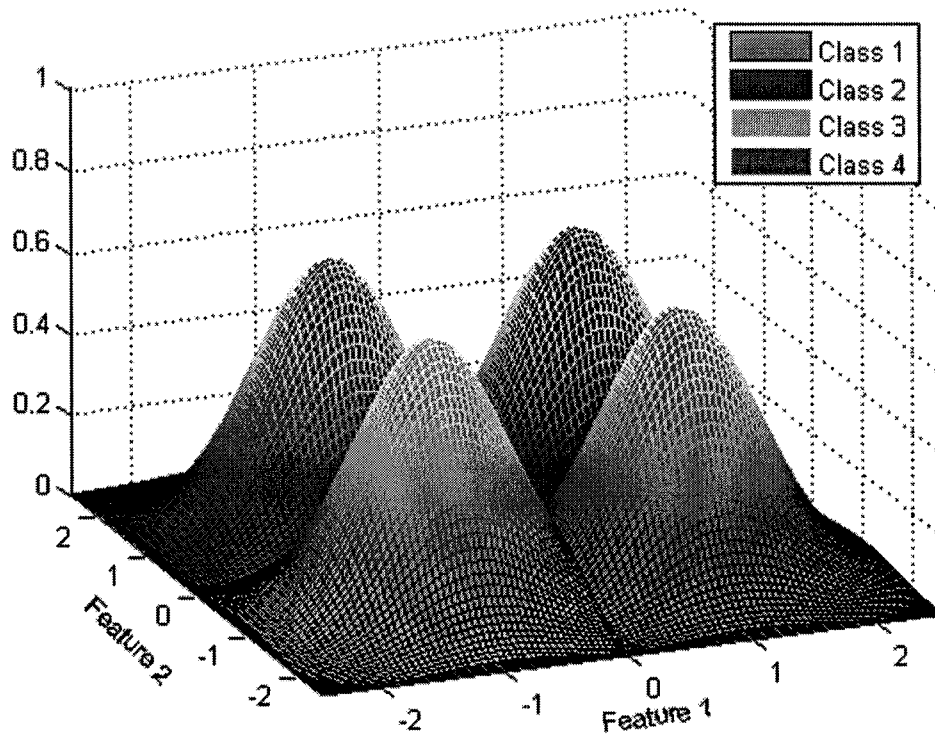


Figure 5.9 – *Example of the four PDF's corresponding to the classes described in Table 5.4.*

This database was split into four training sets and one test dataset. All training sets contain instances from all four classes; however they are severely unbalanced. All of the datasets contain a large number of instances from Class ω_1 and few instances from all of the other classes with the following exception, dataset 2-4 contain a large number of instances from classes 2-4 respectively. This experiment is designed to test the algorithm's abilities to learn from sequentially presented data in the presence of unbalanced data distributions. The actual instance distribution for this experiment can be found in Table 5.5. Furthermore, Figure 5.10 shows an example of the four training datasets and the test dataset

Table 5.5 - Instance distribution of the Experiment 2.

	Class 1	Class 2	Class 3	Class 4
Φ_1	100	10	10	10
Φ_2	100	100	10	10
Φ_3	100	10	100	10
Φ_4	100	10	10	100
Test Data	200	200	200	200

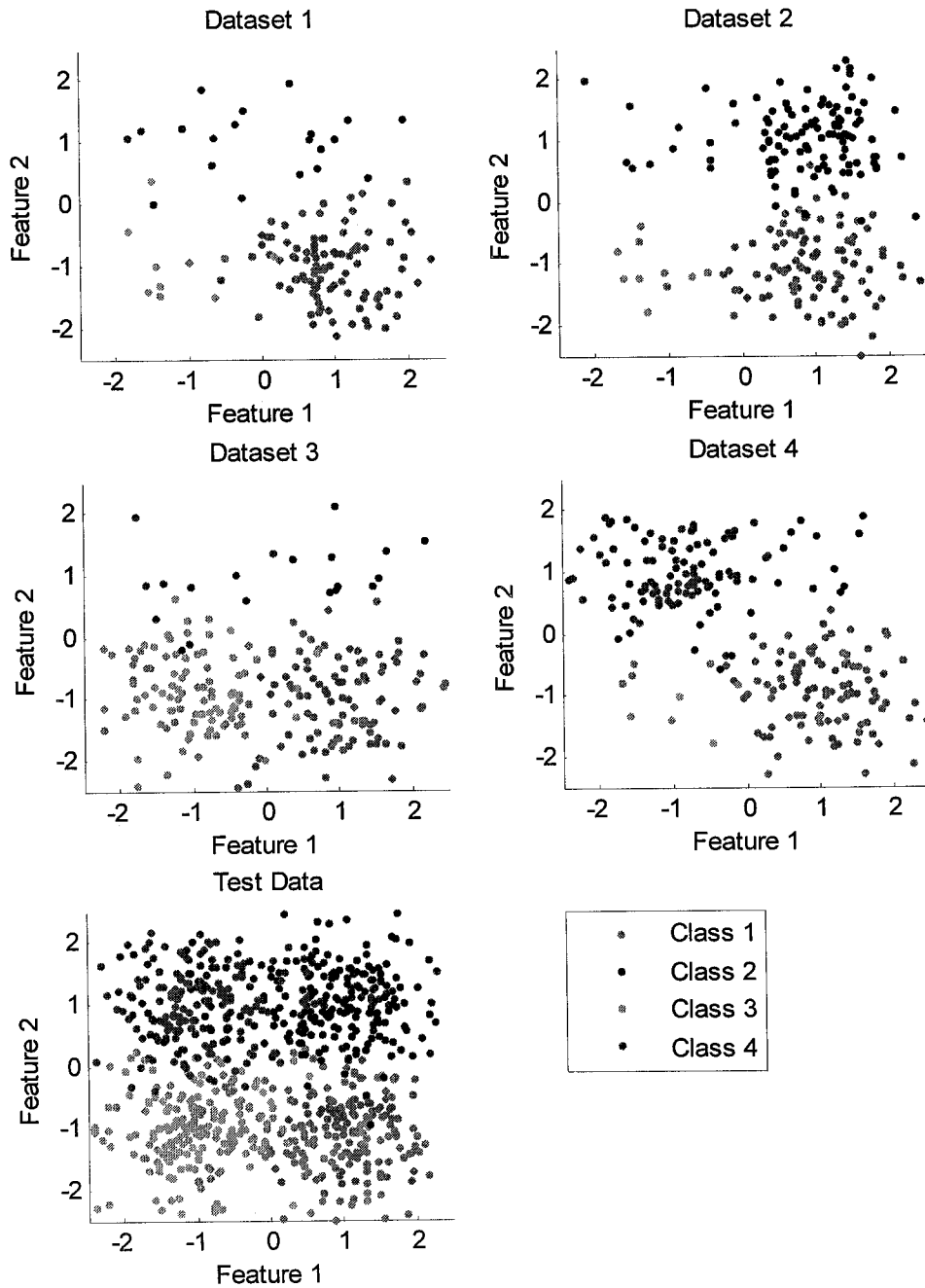


Figure 5.10 - Example training and testing data distributed according to Table 5.5.

Again, since we know the distribution information, we can also calculate the posterior probability for each class. Figure 5.11 show the maximum posterior probability of the Bayes classifier, given that the prior probabilities of all classes are equal.

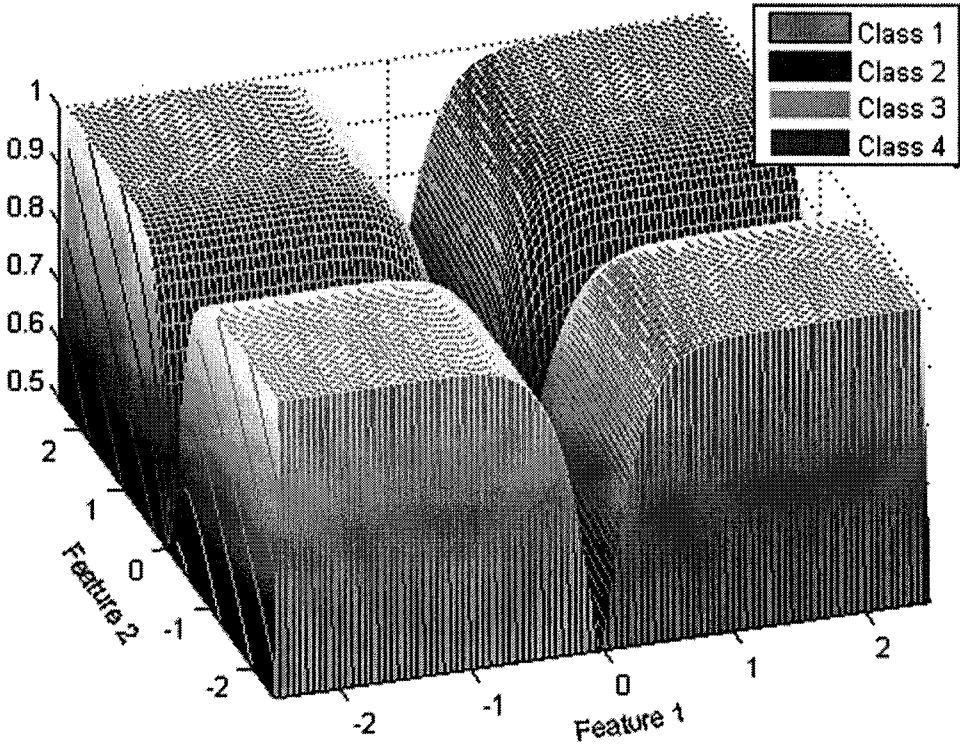


Figure 5.11 - Posterior probability output of the Bayes classifier over the entire feature space.

The **BaseClassifier** for all four algorithms is an MLP classifier with the following network architecture: 0.05 error goal and 25 hidden layer nodes. On each training dataset the algorithms are allowed to create 10 classifiers, 40 classifiers total. Table 5.6 shows the average performance results for this test.

Table 5.6 - Generalization performance from Experiment 2.

		Class 1	Class 2	Class 3	Class 4	Gen. \pm CI
Learn++	Φ_1	98.8%	64.7%	61.5%	76.1%	75.3% \pm 1.6%
	Φ_2	97.9%	86.4%	62.0%	66.8%	78.3% \pm 1.0%
	Φ_3	95.0%	80.0%	80.0%	64.7%	80.7% \pm 0.9%
	Φ_4	98.5%	72.3%	73.3%	87.1%	82.8%\pm0.7%
Learn++.NC	Φ_1	98.9%	63.2%	64.2%	77.5%	76.0% \pm 1.7%
	Φ_2	95.9%	93.6%	52.3%	46.4%	72.1% \pm 2.1%
	Φ_3	97.2%	69.7%	88.4%	38.2%	73.4% \pm 2.0%
	Φ_4	98.8%	42.3%	68.5%	93.2%	75.7% \pm 1.2%
Learn++.UD	Φ_1	98.0%	67.7%	73.9%	80.6%	80.0% \pm 1.4%
	Φ_2	94.2%	95.2%	62.3%	43.3%	73.8% \pm 1.9%
	Φ_3	90.8%	91.9%	95.7%	28.7%	76.7% \pm 1.6%
	Φ_4	94.1%	43.4%	69.5%	98.2%	76.3% \pm 1.9%
BEAST	Φ_1	95.3%	81.2%	78.5%	81.0%	84.2% \pm 1.2%
	Φ_2	95.7%	89.1%	80.7%	82.2%	86.9% \pm 0.6%
	Φ_3	94.7%	88.6%	87.6%	80.3%	87.8% \pm 0.6%
	Φ_4	94.8%	86.1%	86.4%	89.1%	89.1%\pm0.3%
Bayes Classifier						91.3%

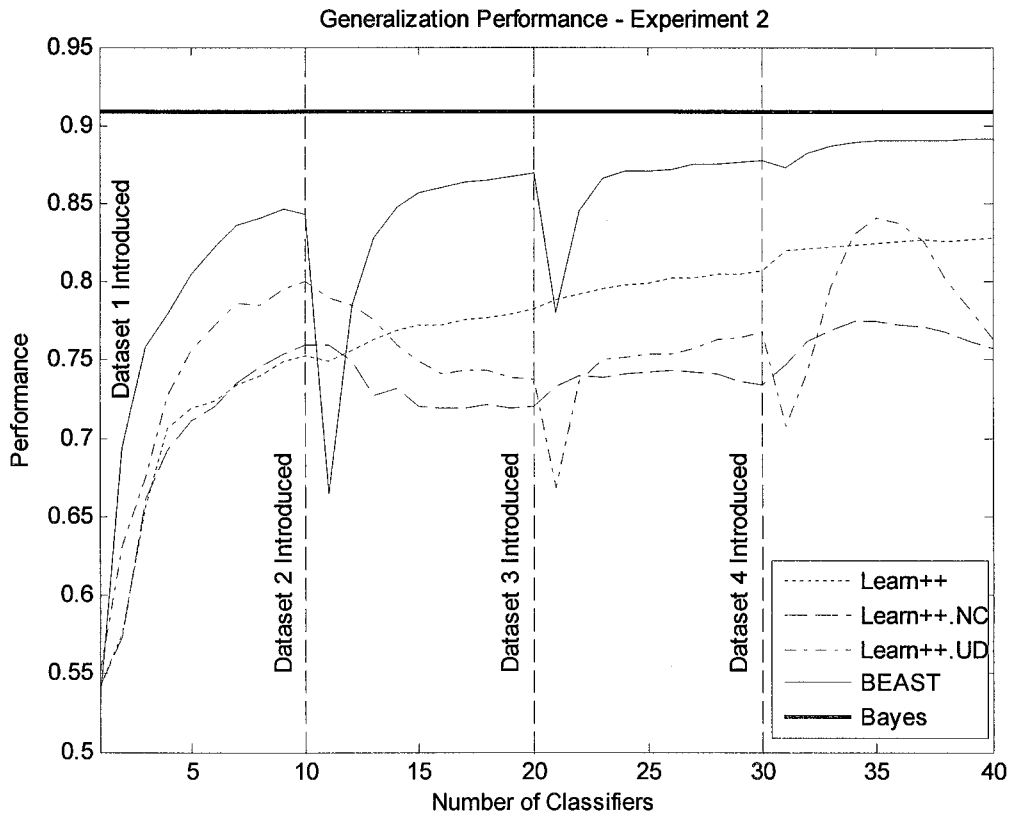


Figure 5.12 – Generalization performance vs. number of classifiers.

Table 5.6 and Figure 5.12 indicate that BEAST outperforms the other algorithms. Furthermore, Figure 5.13 shows the theoretical performance PDFs, of which the theoretical performance of the Bayes classifier and BEAST are extremely close. In fact they do not have a statistically significant difference, whereas, there is a very significant statistical difference between BEAST and the other three algorithms. Additionally, Figure 5.14 shows that the decision boundary of BEAST is very similar to that of the Bayes classifier. Note that the decision boundaries of other algorithms tend to favor Class 1, since they have seen much more data from this class.

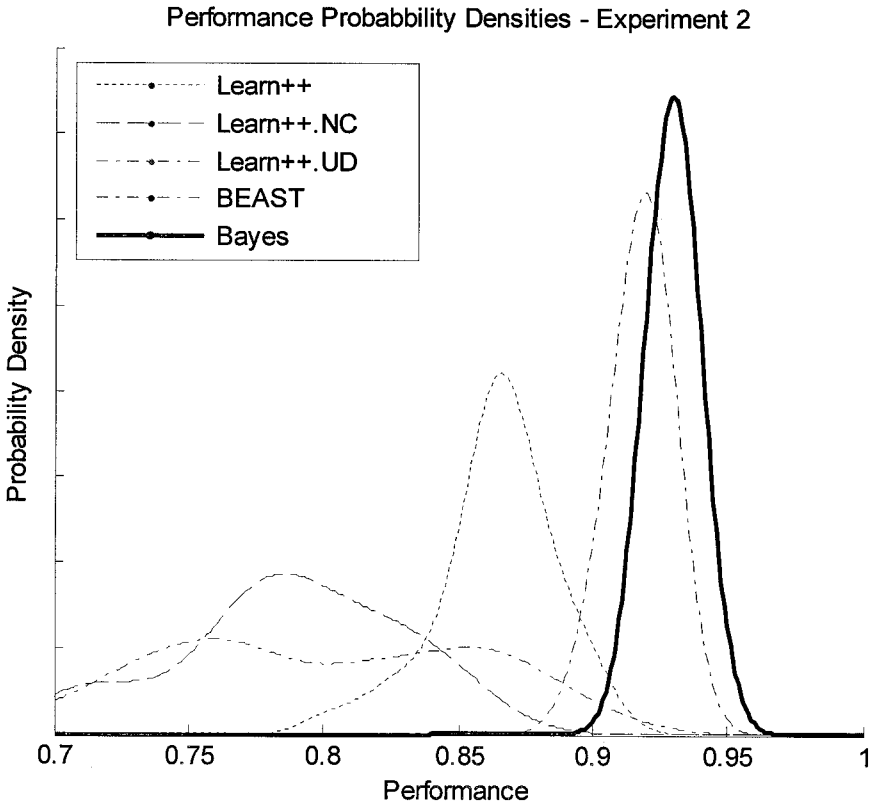


Figure 5.13 – Performance probability density functions from experiment 2.

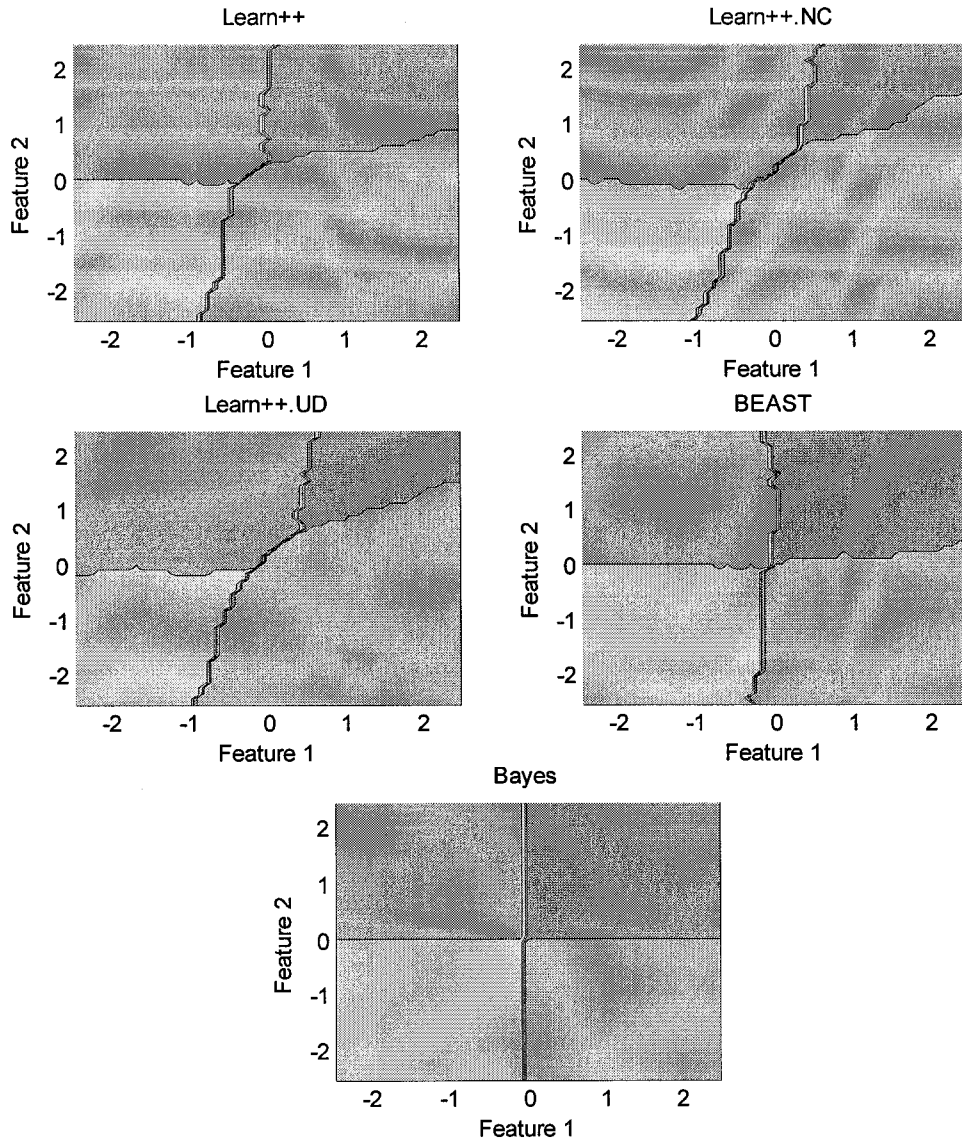


Figure 5.14 – *Decision boundaries over the entire feature space.*

Analyzing individual class performances as the new training datasets are introduced will help to better understand why each algorithm performs the way it does. Figure 5.12 indicates that the average performance of Learn++ steadily increases as classifiers are added and datasets introduced. One may be tempted to assume that this performance would continue increasing if more classifiers were added. However, Figure 5.15 shows that Learn++ learns from the abundance of information on classes 2-4 in

datasets 2-4 respectively, while simultaneously forgetting information on the other classes which don't have such a generous amount of data. Meanwhile, Learn++.NC reacts almost identically to Learn++ on this experiment but with more drastic changes. Learn++.UD might be expected to perform well on this problem considering that it is designed to handle unbalanced data. However, Learn++.UD is designed to reduce the adverse effects of unbalanced data between datasets, not between classes. Learn++.UD does perform well on the classes which have generous amounts of data, but performs poorly on those classes with fewer instances. BEAST is clearly superior in this experiment as it contains methods to handle unbalanced data both between classes in the same dataset and between datasets on the same class.

An interesting observation from Figure 5.12 is how the generalization performance on BEAST drops significantly upon the addition of the first classifier in each ensemble. BEAST combines the decisions from all classifiers generated on each dataset and then combines each of those decisions to get the final decision. Thus, when a new dataset is introduced, one classifier is trained on that data, that single classifier has the same amount of voting power as all the classifiers from the previous sections. In this experiment, upon creation, the 11th classifier is the sub-ensemble trained on \mathfrak{D}_2 and has as much voting power as classifiers 1-10, which constitute the sub-ensemble trained on \mathfrak{D}_1 . Since the 11th classifier performs well on class 1 and 2, and poorly on classes 3-4, the performance on class 2 shoots up and the performance on classes 3-4 drops significantly. However, as more classifiers are trained on the current dataset, BEAST recovers from the performance drop on classes 3-4. This situation occurs again with the

introduction of dataset 3 and 4, although, the effects are less noticeable since more sets of classifiers are then able to vote.

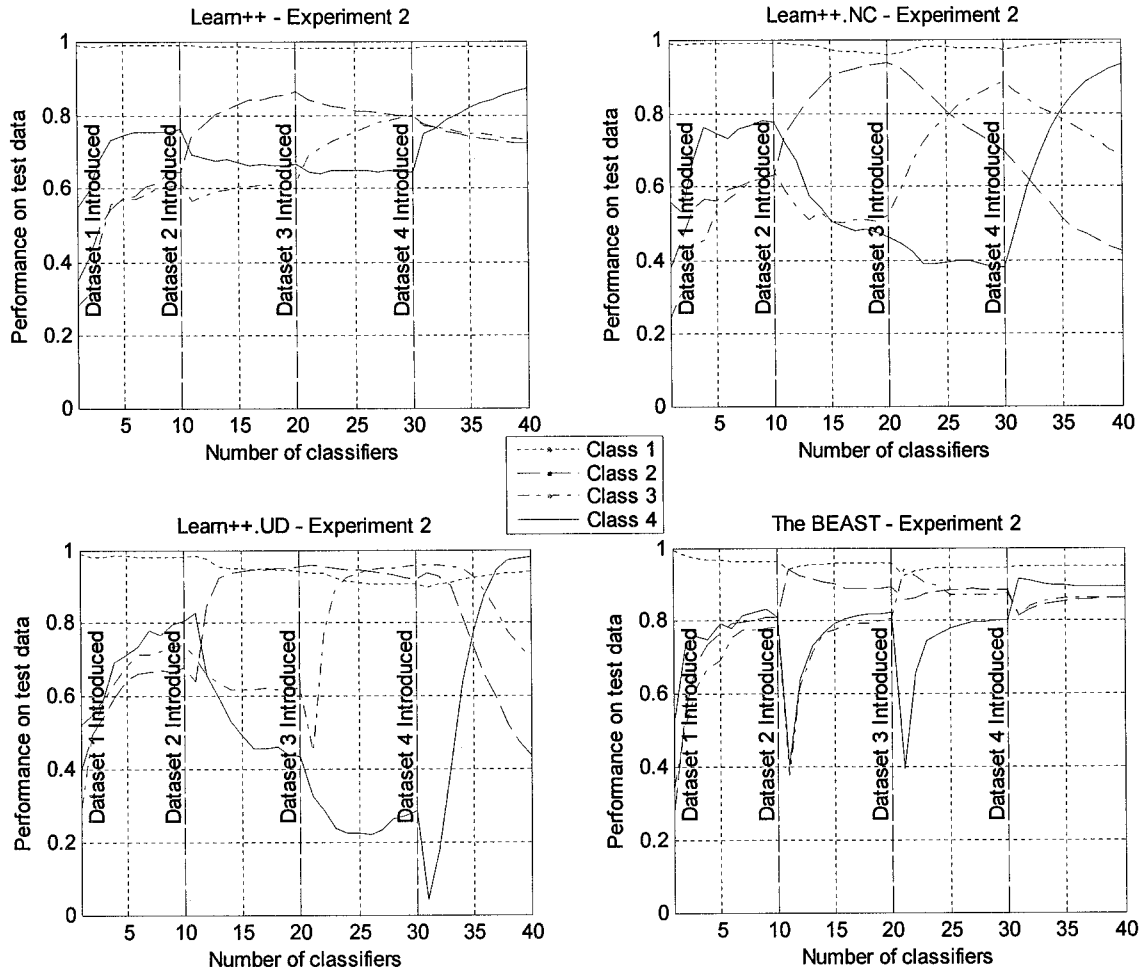


Figure 5.15 – *Class specific generalization performances.*

5.2.3 Synthetic Experiment 3 - Spiral Database

The third experiment is performed on the spiral database, which has been created and defined specifically for this research. The database can be created to have any number of spirals where each spiral can be defined in polar coordinates as follows:

$$\theta = r + \theta_0 \quad (5.4)$$

where θ_0 is the start angle of the spiral and r is the radius. Equation (5.4) defines the spiral; in order to draw samples from this distribution, a random variable is required. Ξ is

defined to generate a random number according to a normal distribution with zero mean and unit variance. The variance element must be a function of the radius of the spiral so that the data becomes more “noisy” as the spiral grows. Equations (5.5) - (5.7) provide a method of drawing n instances from the aforementioned spiral distribution.

$$r_i = t_i + \sigma_0 t_i \Xi, \quad \text{for } i = 1, 2, \dots, n \quad (5.5)$$

and

$$\theta_i = t_i + \theta_0 + \sigma_0 t_i \Xi, \quad \text{for } i = 1, 2, \dots, n \quad (5.6)$$

where

$$t_i = 2\pi \sqrt{i/n}, \quad \text{for } i = 1, 2, \dots, n \quad (5.7)$$

In equations (5.5) and (5.6) σ_0 represents the variance at a radius of one. Since n samples are drawn from n Gaussian distributions, each with its own mean and variance, we can compute the likelihood function

$$P([\theta, r] | \omega_j) = \frac{1}{n\sqrt{2\pi}} \sum_{i=1}^n \frac{1}{\sigma_i} e^{-0.5((\theta-\theta_i)^2 + (r-r_i)^2)/\sigma_i^2} \quad (5.8)$$

where

$$\sigma_i = \sigma_0 t_i, r_i = t_i, \theta_i = t_i + \theta_0 \quad (5.9)$$

Thus the posterior probability can be calculated according to equation (5.2). Both the sampled data and the posterior probability can then be converted to Cartesian coordinates using by

$$\begin{aligned} x_1 &= r \cos(\theta) \\ x_2 &= r \sin(\theta) \end{aligned} \quad (5.10)$$

Table 5.7 shows the parameters for the four spiral classes used in this experiment.

Table 5.7 – Distribution information of the spiral database.

	Class 1	Class 2	Class 3	Class 4
θ_0	0	$\pi/2$	π	$3\pi/2$
σ_0	0.05	0.05	0.05	0.05

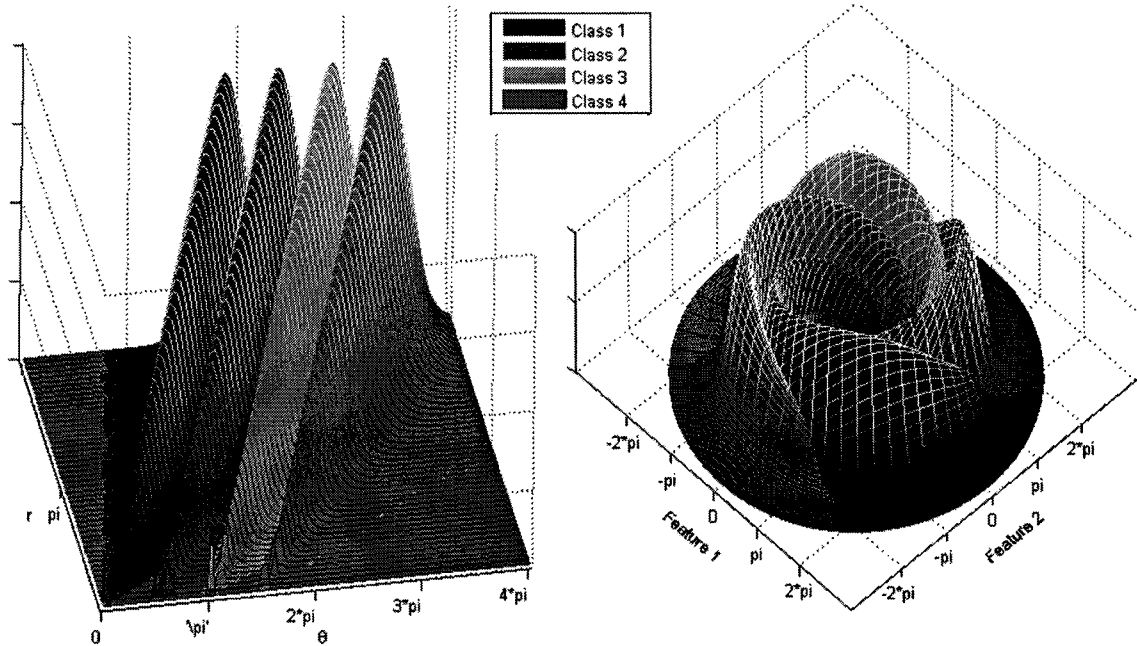


Figure 5.16 - Graph of the four PDF's corresponding to the spiral classes described Table 5.7. The PDF's are shows in polar coordinates (left) and cartesian coordinates (right).

The first two synthetic experiments tested the algorithms ability to incrementally learn new classes and learn from unbalanced data. Recall in experiment 1, that we only tested the algorithm's ability to learn newly introduced classes. In this experiment we test the algorithms ability to learn information from new classes, when information from previously seen classes are unavailable. The database will be split into four training and a test dataset. None of the four training sets will contain information on all four classes, more specifically, datasets 1-2 only contain data from two different classes, datasets 3-4 contain instances from three of the four classes. This is a particularly difficult problem since the algorithm is forced to learn from the new information without access to

instances from certain previously learned classes. Table 5.8 shows the actual instance distribution used for experiment 3.

Table 5.8 - Instance distribution of the spiral database.

	Class 1	Class 2	Class 3	Class 4
\mathcal{D}_1	200	0	200	0
\mathcal{D}_2	0	200	0	200
\mathcal{D}_3	50	150	0	150
\mathcal{D}_4	0	150	50	150
Test Data	200	200	200	200

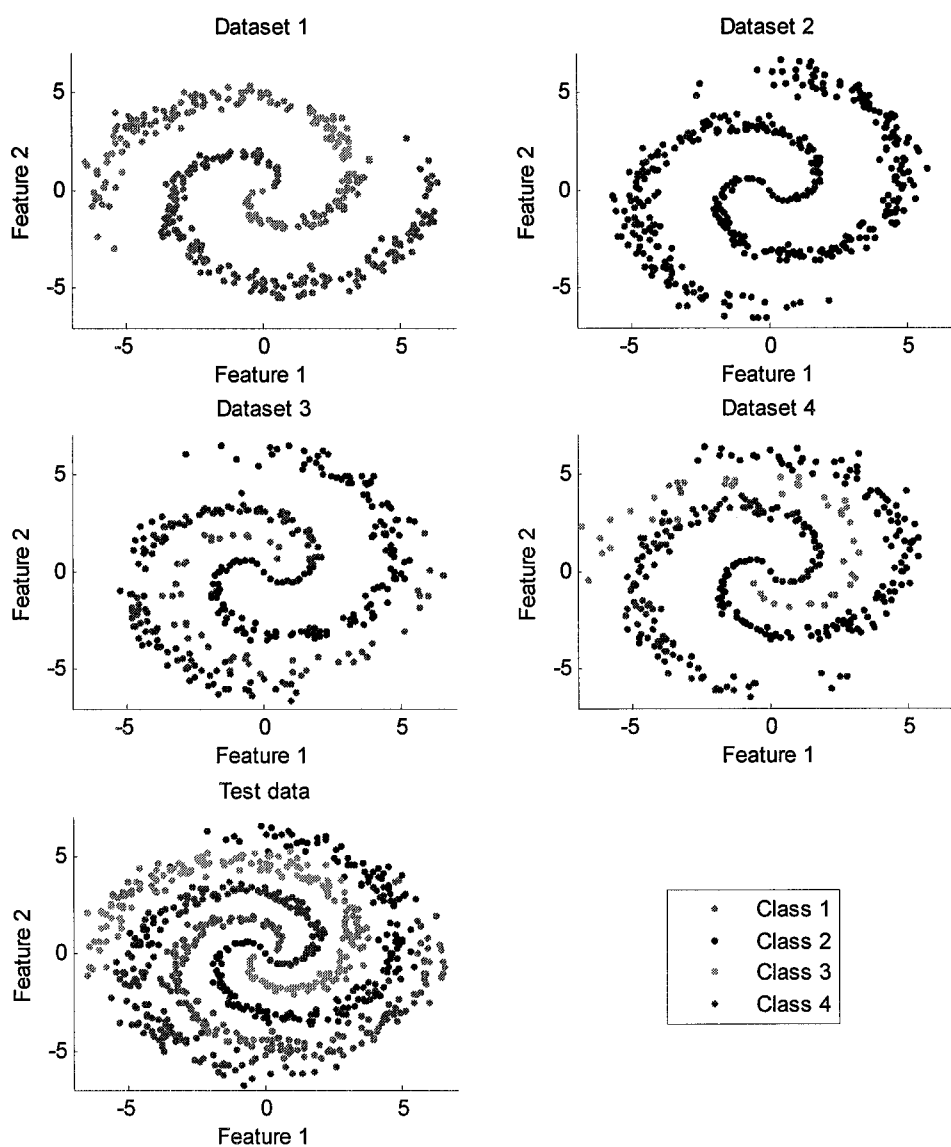


Figure 5.17 - Sample training and testing data distributed according to Table 5.8.

Using Equation (5.8) we can calculate the likelihood functions, which then can be used in Equation (5.2) to calculate the posterior probability for each class. Figure 5.18 shows the maximum posterior probability and classification of the Bayes classifier.

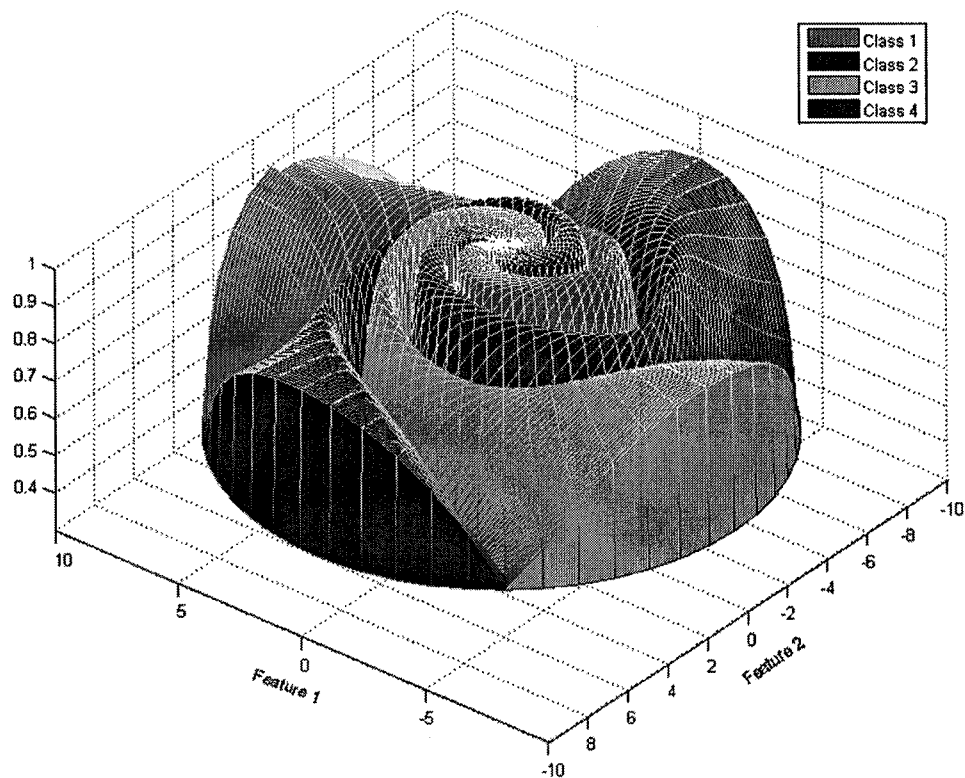


Figure 5.18 - *Posterior probability output of the Bayes classifier over the entire feature space.*

All four algorithms generated MLP classifiers with identical network architectures of, 25 hidden layer nodes and an error goal of 0.025. On each training dataset the algorithms are allowed to create 5 classifiers, 20 classifiers total. Table 5.9 and Figure 5.19 show the tabular and graphical results for this test.

Table 5.9 - Generalization performance on the swirl database.

		Class 1	Class 2	Class 3	Class 4	Gen. \pm CI
Learn++	Φ_1	97.9%	-	97.5%	-	48.9% \pm 0.1%
	Φ_2	63.6%	47.9%	62.5%	45.5%	54.9% \pm 1.3%
	Φ_3	83.4%	93.1%	9.5%	92.6%	69.7% \pm 1.5%
	Φ_4	56.1%	97.7%	42.3%	98.2%	73.6% \pm 3.7%
Learn++.NC	Φ_1	97.5%	-	97.3%	-	48.7% \pm 0.1%
	Φ_2	92.0%	24.2%	18.7%	25.4%	40.1% \pm 1.0%
	Φ_3	56.0%	66.2%	35.4%	57.4%	53.7% \pm 1.3%
	Φ_4	49.3%	98.6%	71.1%	98.8%	79.5% \pm 1.2%
Learn++.UD	Φ_1	97.6%	-	97.9%	-	48.9% \pm 0.1%
	Φ_2	9.7%	94.5%	7.6%	95.1%	51.7% \pm 0.9%
	Φ_3	84.4%	96.2%	15.6%	95.0%	72.8% \pm 0.6%
	Φ_4	83.1%	96.7%	83.4%	97.3%	90.1% \pm 0.5%
BEAST	Φ_1	97.5%	-	97.7%	-	48.8% \pm 0.2%
	Φ_2	91.8%	54.8%	52.0%	24.1%	55.6% \pm 0.9%
	Φ_3	91.4%	91.4%	66.1%	53.7%	75.6% \pm 0.7%
	Φ_4	89.5%	92.6%	91.1%	92.3%	91.4% \pm 0.4%
Bayes Classifier						98.5%

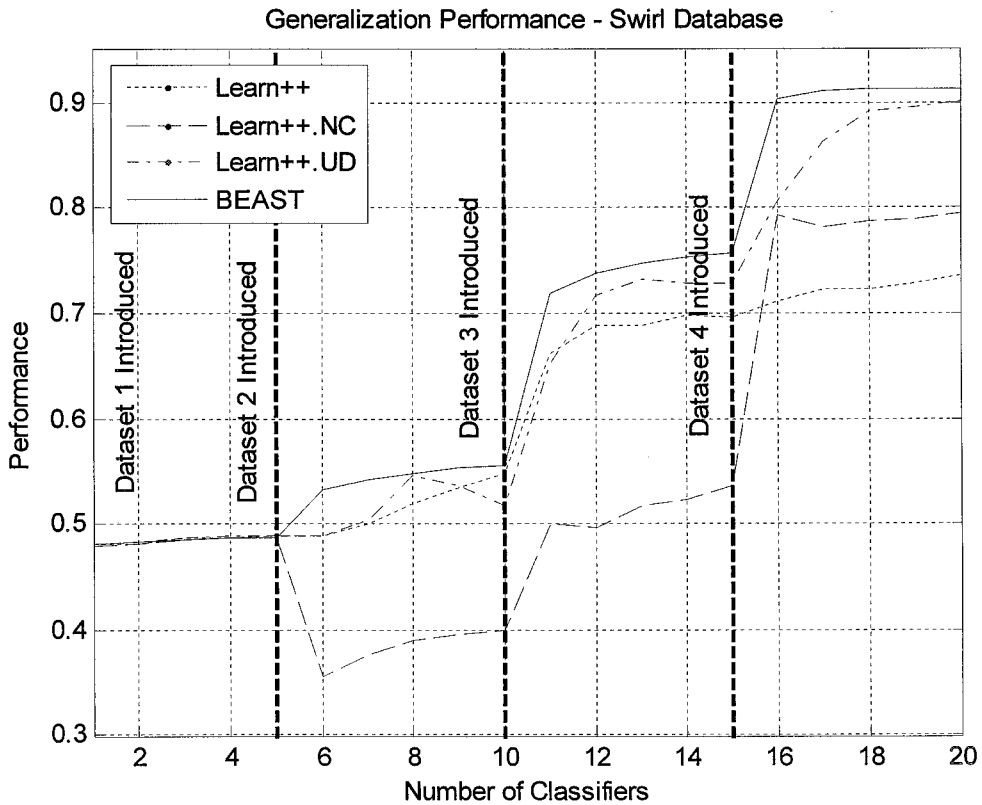


Figure 5.19 – Generalization performance vs. number of classifiers.

Table 5.9 and Figure 5.19 show the ability of BEAST to quickly and efficiently learn the information available from each dataset. Learn++.UD also displays desirable characteristics under these test conditions. The performance of Learn++ and Learn++.NC are relatively poor. Figure 5.20 illustrates that the performance PDF of Learn++ is inconsistent and does not generate repeatable results.

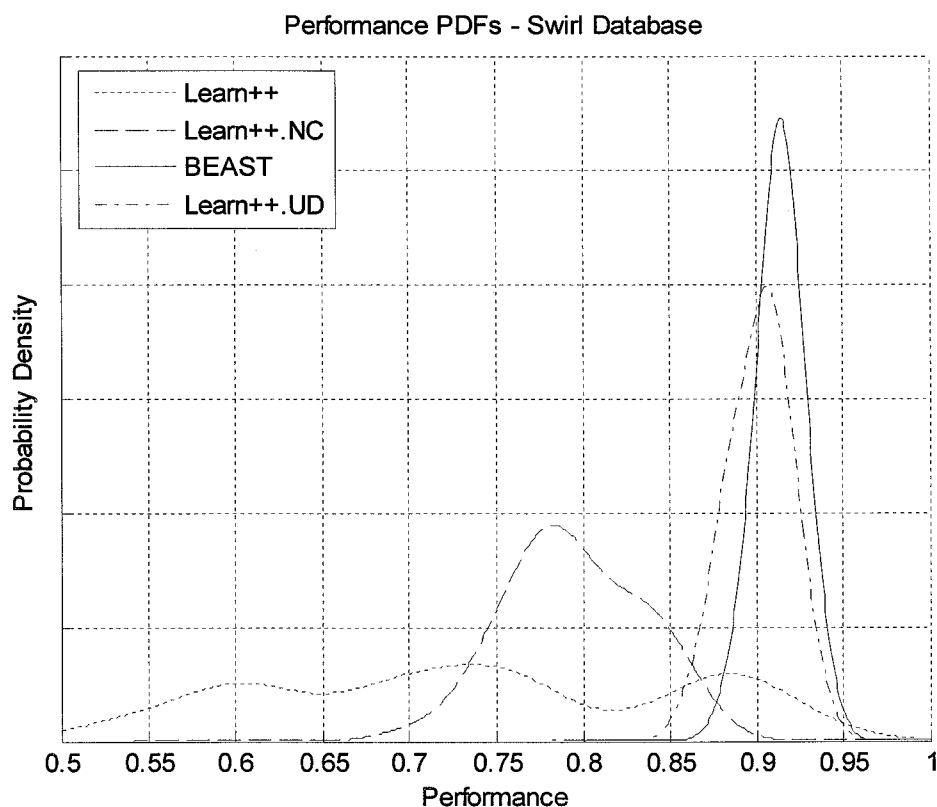


Figure 5.20 – Performance probability density functions from the swirl database.

Figure 5.21 shows the decision boundaries of the three algorithms along with the decision boundary of the Bayes classifier. This figure clearly illustrates that Learn++ and Learn++.NC have difficulty classifying two of the spirals. It is also clear that BEAST decision boundary is most similar to that of the Bayes classifier, in fact, there are very few differences between their boundaries.

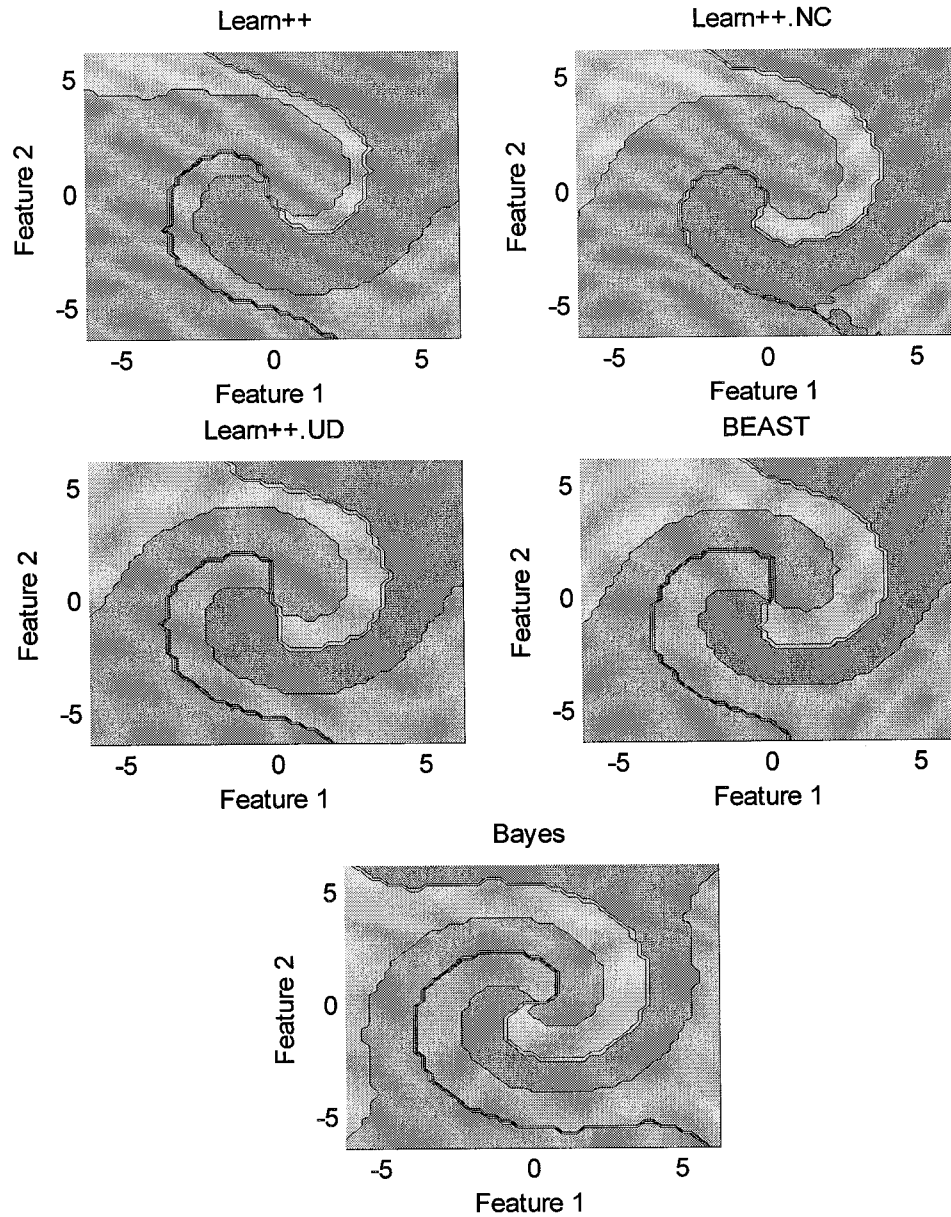


Figure 5.21 – *Decision boundaries over the entire feature space.*

Once again, it is interesting to analyze the class-specific-performance results of these algorithms, shown in Figure 5.22. Learn++ is only able to learn new class information at the cost of previously learned information. Learn++.NC appears to be biased towards the set of spirals introduced in \mathfrak{D}_2 and cannot properly learn information from the other two classes. While the results obtained using Learn++.UD appear to be

volatile, the end result is a good balance between individual class performances. The BEAST algorithm is the most robust and learns as much information as possible from each dataset without compromising too much existing knowledge.

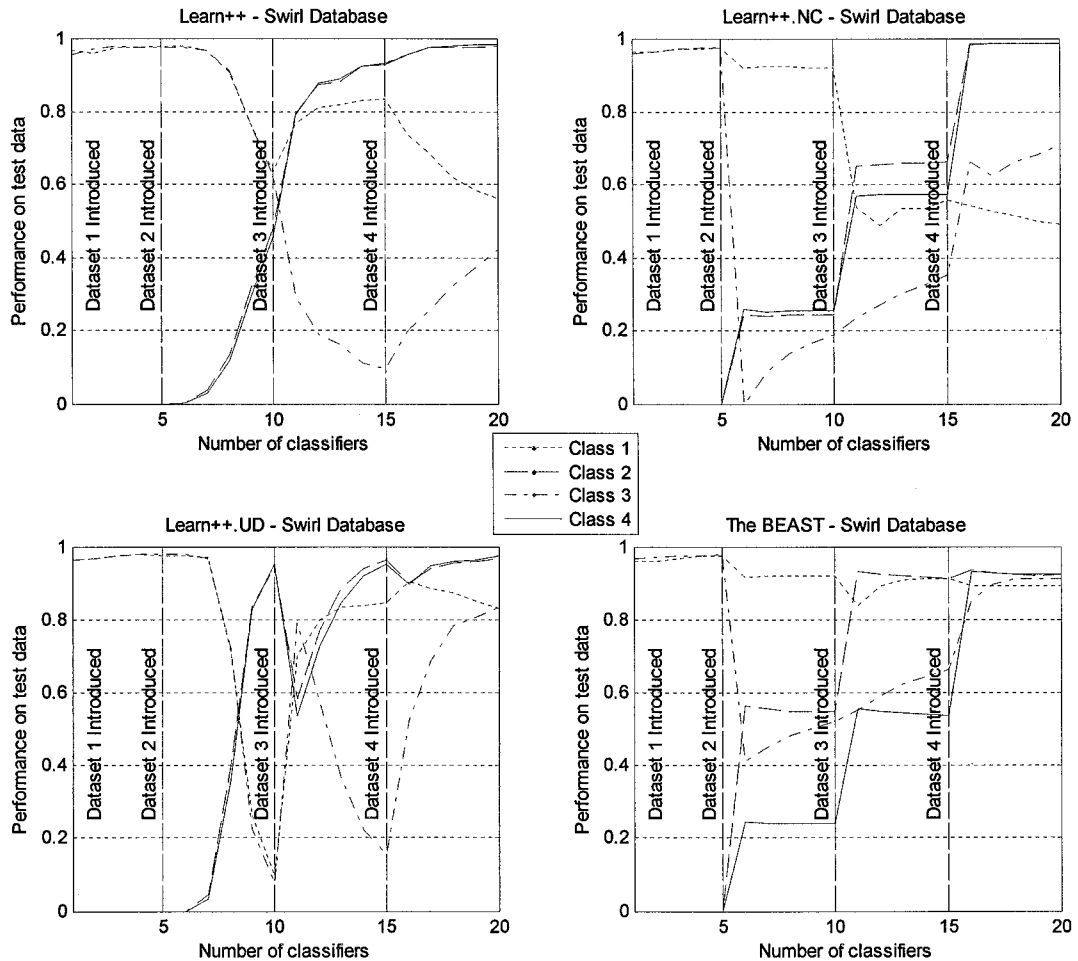


Figure 5.22 – Class specific generalization performances.

5.3 Simulation Results on Experimental Databases

All experimental results in this section are presented as explained in section 5.1.3. In the previous section, all results were calculated as an average of 40 independent trials. This technique is optimal for synthetic data since new training and testing data can be randomly sampled from the environment at the beginning of each trial. This luxury is not

available when working with real world data since the numbers of samples are limited. Consequently, k -fold cross validation is used to ensure the calculation of accurate and representative performance figures. Cross-validation breaks the entire database into k blocks of data; for each trial one block is used for testing, and the remaining $k-1$ blocks are combined for training. This procedure is repeated k times, such that every block of data is independently used for testing, resulting in more accurate figures of generalization performance. A graphical depiction of this cross validation process is seen in Figure 5.23.

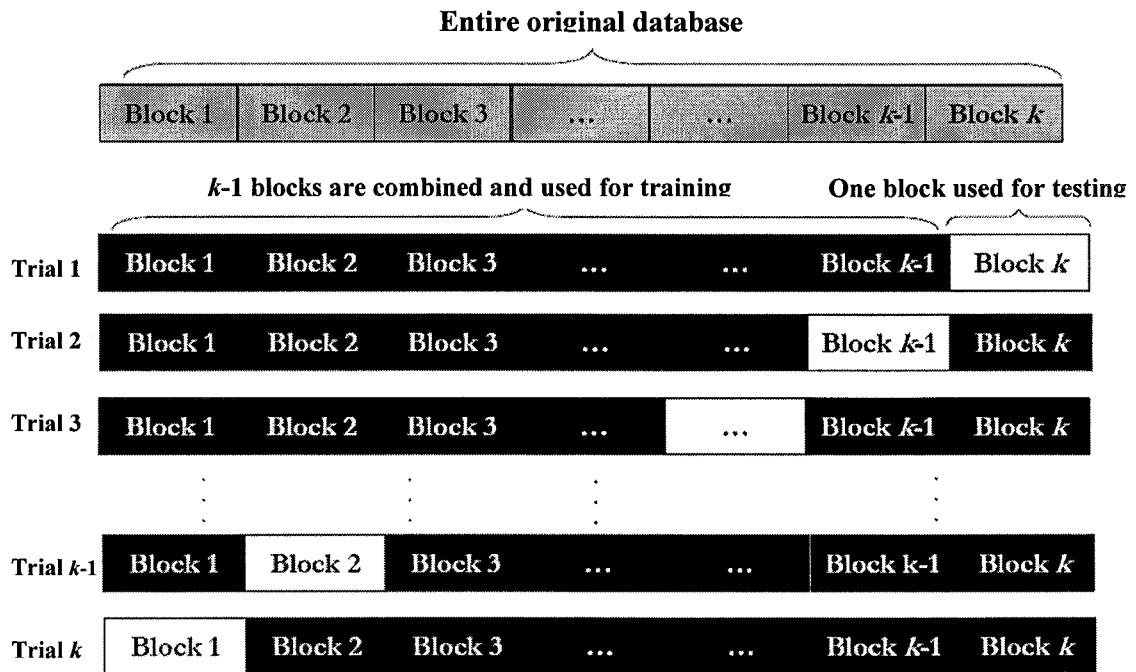


Figure 5.23 – k -fold cross validation diagram [54].

Note: all of the results presented in this section have been created using 10-fold cross validation.

5.3.1 Volatile Organic Compounds Recognition Database

The Volatile Organic Compounds (VOC) database is generated from a real world problem of identifying one of five VOCs based on the responses of 6 chemical sensors. Each sensor is coated with a different polymer, the collection of which constitutes the 6-feature instances. The individual VOCs were ethanol (ET), octane (OC), toluene (TL), trichloroethylene (TCE), and xylene (XL). Figure 5.24 shows an example of sensor data collected on each of these VOCs.

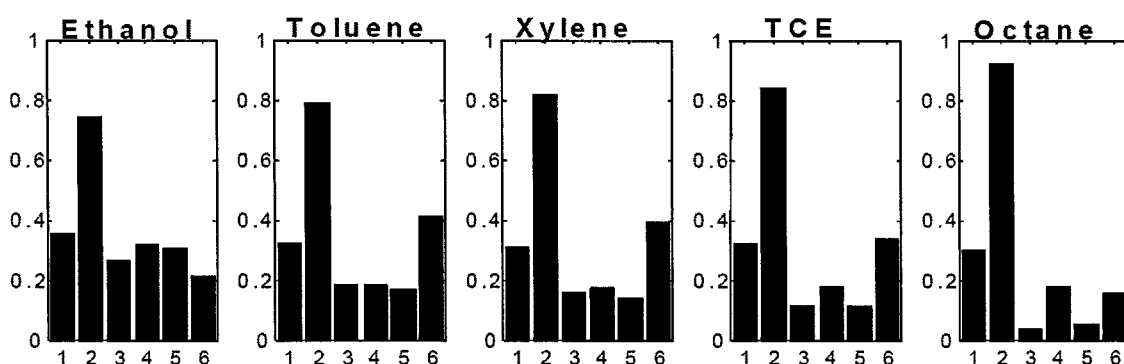


Figure 5.24 – Example instances drawn from the VOC recognition database.

The database was split into 10-blocks, 9 of which were combined and split into three training datasets, $\mathcal{D}_1 \sim \mathcal{D}_3$. The remaining block was used for testing, according to the abovementioned k -fold cross validation procedure. The instance distribution used in this experiment can be seen in Table 5.10; note that the number of test instances from any given class may vary by one since the database could not be evenly divided by 10. This experiment is designed to test the algorithms under some of the harshest unbalanced data conditions. \mathcal{D}_2 and \mathcal{D}_3 both introduce instances from new classes in addition to providing a very limited number of instances from previously learned classes.

Table 5.10 – Instance distribution for the VOC database experiment.

Class→	ET	OC	TL	TCE	XL
\mathcal{D}_1	0	40	0	40	40
\mathcal{D}_2	0	5	40	5	5
\mathcal{D}_3	40	5	5	5	5
Test	6	6	6	8	11

Each algorithm was sequentially trained on $\mathcal{D}_1 \sim \mathcal{D}_3$, creating five MLP classifiers on each dataset. The MLP classifiers all had 35 hidden layer nodes and were given an error goal of 0.025. The numerical performance results of this test can be found in Table 5.11. The graphical display of performance as classifiers are created is shown in Figure 5.25.

Table 5.11 – Generalization performance on VOC database.

		Ethanol	Octane	Toluene	TCE	Xylene	Gen. \pm CI
Learn++	\mathcal{D}_1	-	97.8%	-	93.6%	77.5%	55.4% \pm 1.1%
	\mathcal{D}_2	-	80.9%	81.4%	93.9%	66.9%	66.2% \pm 1.6%
	\mathcal{D}_3	36.9%	80.5%	92.6%	89.9%	68.1%	73.6% \pm 1.6%
Learn++.NC	\mathcal{D}_1	-	97.6%	-	92.7%	77.1%	58.1% \pm 1.1%
	\mathcal{D}_2	-	61.1%	98.8%	91.6%	67.5%	65.5% \pm 1.5%
	\mathcal{D}_3	96.4%	67.4%	94.4%	88.0%	62.4%	79.6% \pm 1.5%
Learn++.UD	\mathcal{D}_1	-	97.7%	-	92.7%	77.2%	58.1% \pm 1.2%
	\mathcal{D}_2	-	72.0%	96.9%	93.5%	71.6%	68.5% \pm 1.4%
	\mathcal{D}_3	93.8%	66.8%	96.0%	93.9%	64.0%	81.0% \pm 1.4%
BEAST	\mathcal{D}_1	-	97.5%	-	93.5%	76.6%	58.1% \pm 1.1%
	\mathcal{D}_2	-	87.8%	93.8%	94.4%	71.9%	70.9% \pm 1.3%
	\mathcal{D}_3	88.4%	87.6%	93.5%	94.2%	67.7%	84.3% \pm 1.4%

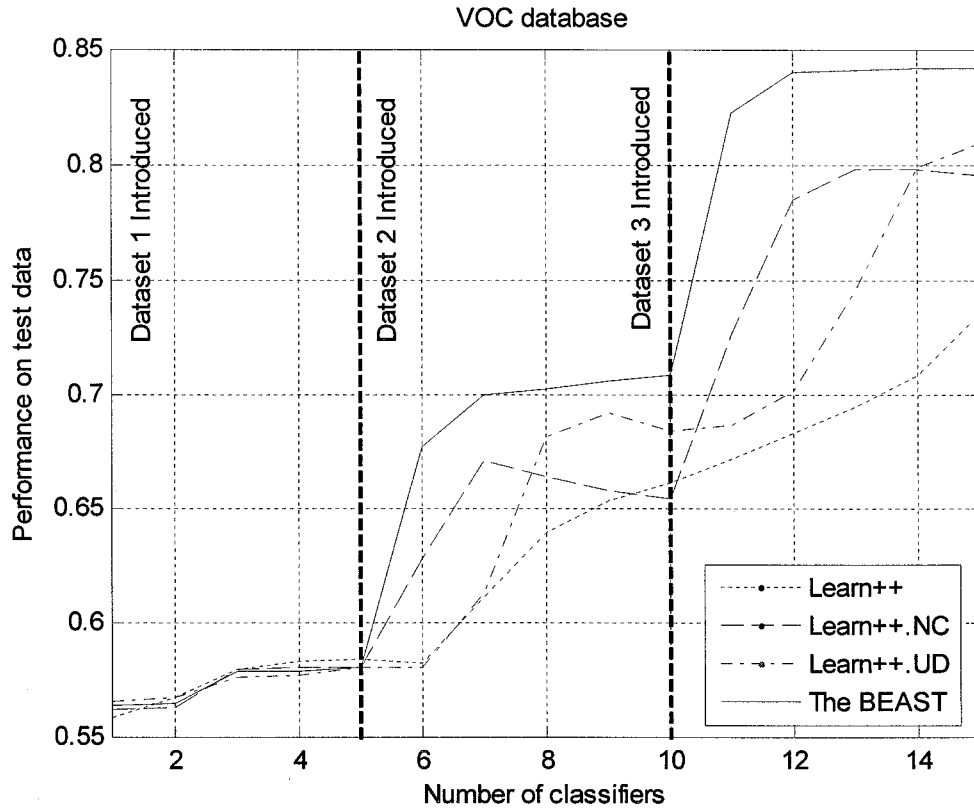


Figure 5.25 - Generalization performance vs. number of classifiers from VOC experiment.

These results show significant differences in generalization performance between the four algorithms. Learn++ is incapable of learning the required information about Ethanol from \mathfrak{D}_3 with only five classifiers. Learn++.NC and Learn++.UD display similar performances; however Learn++.UD learns at slower pace compared to the abrupt learning style of Learn++.NC. The BEAST algorithm exhibits a considerable performance increase over its predecessors, Learn++.NC and Learn++.UD; this increase is shown to be statistically significant from the final performance PDFs, as seen in Figure 5.26.

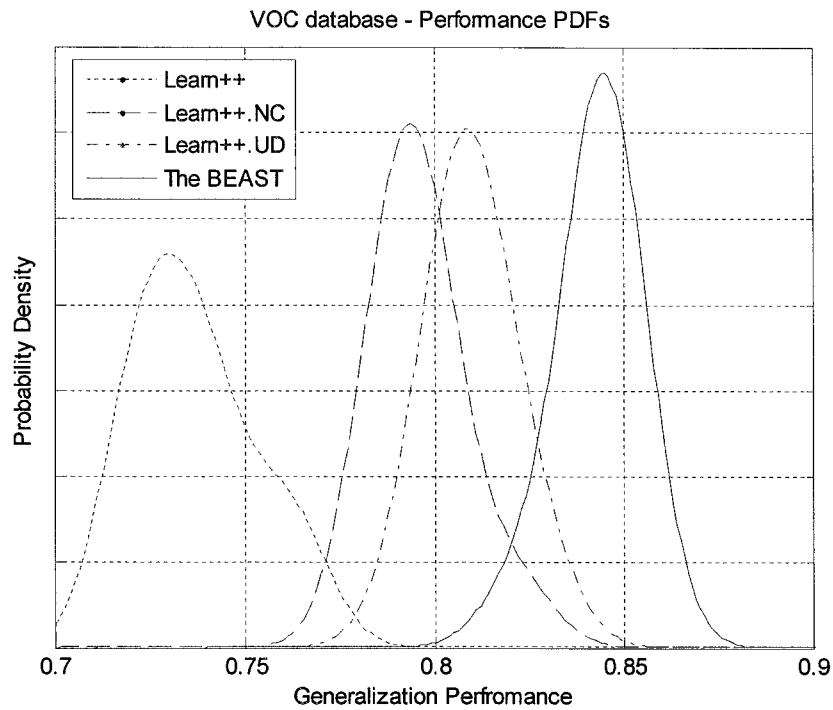


Figure 5.26 - Performance probability density functions from VOC experiment.

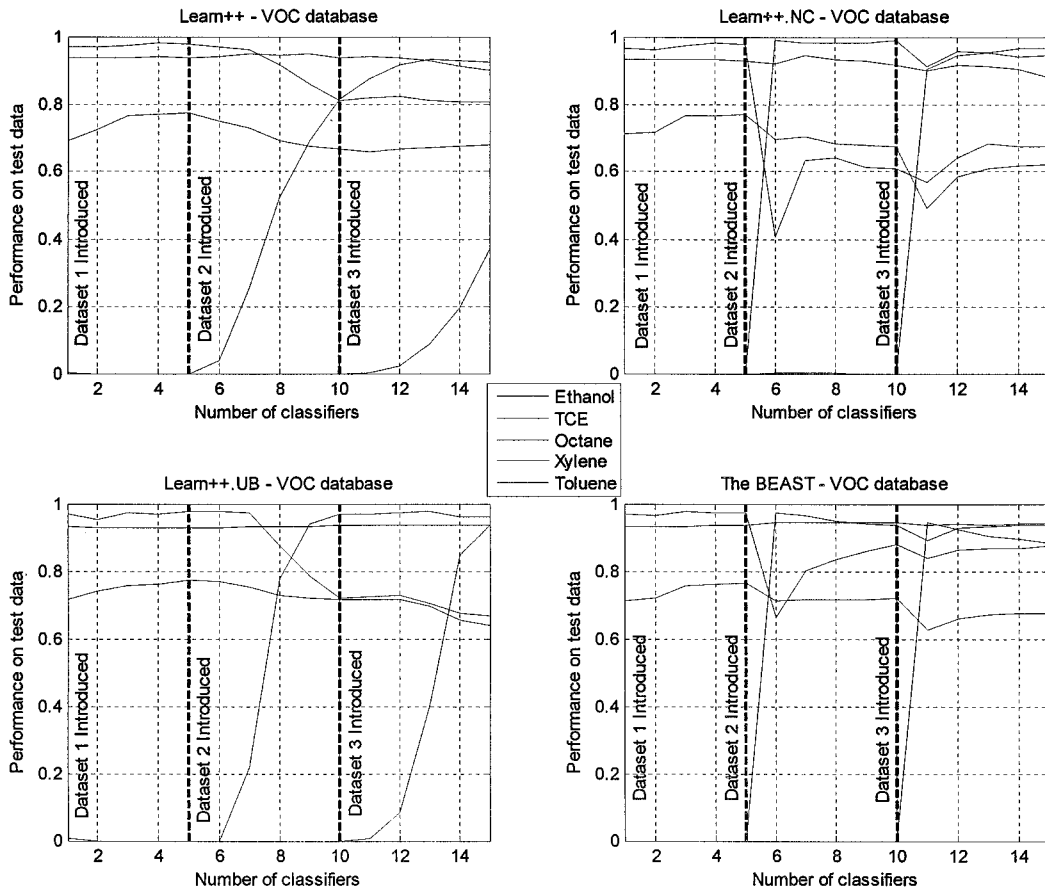


Figure 5.27 - Class specific generalization performances on the VOC database.

Figure 5.27 shows the class specific performance results. The performance plots from Learn++.UD and Learn++.NC clearly show their different learning characteristics although their end performances are statistically similar. BEAST exhibits more favorable characteristics; it is capable of rapidly learning the new class information, like Learn++.NC, while preventing performance drops from previously learned information. This is not to say that the performance on previously learned classes does not drop, but rather, the learning of new information in favor of existing knowledge is more properly balanced, resulting in the maximization of each class performance. Learn++ is clearly unable to learn instances from the Ethanol class with five classifiers; however, one may assume that with the addition of more classifiers trained on \mathfrak{D}_3 the performance on this class will increase. Thus an additional test was conducted where Learn++ was allowed to create 15 classifiers on \mathfrak{D}_3 , results of this test can be found in Figure 5.28.

Figure 5.28 shows the performance of the Learn++ algorithm when allowed to generate 15 classifiers on \mathfrak{D}_3 . The result is a significant increase in performance from the previous experiment with Learn++. This new performance is comparable to that of Learn++.NC and Learn++.UD but still fall significantly short of the performance attained using the BEAST algorithm.

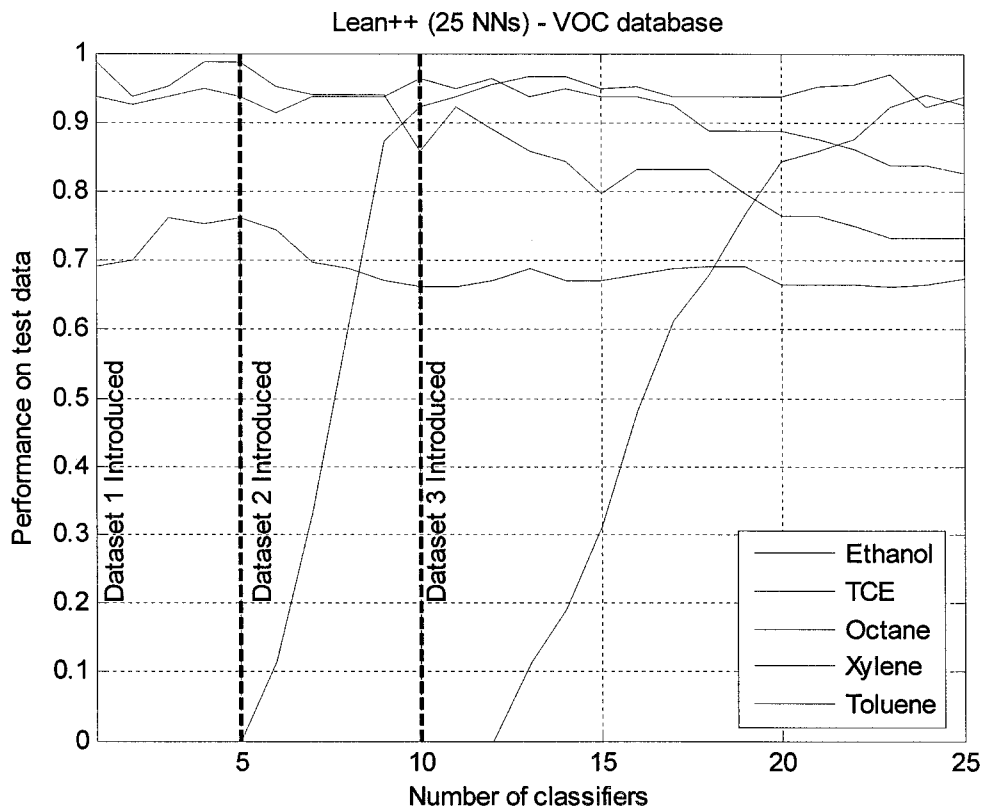


Figure 5.28 – Class specific generalization performance of Learn++ with 10 additional classifiers.

5.3.2 Optical Character Recognition Database

The optical character recognition (OCR) database consists of 10 classes with 64 attributes, obtained from handwritten numeric characters 0 ~ 9, digitized on an 8-by-8 grid. Several examples of these characters can be seen in Figure 5.29. The database was split into five subsets to create four training subsets, $\mathcal{D}^1 \sim \mathcal{D}^4$, and one test subset. The data distribution, shown in Table 5.12, was designed to determine the algorithms' ability to learn two new classes with each additional dataset, while retaining knowledge from previously learned classes.

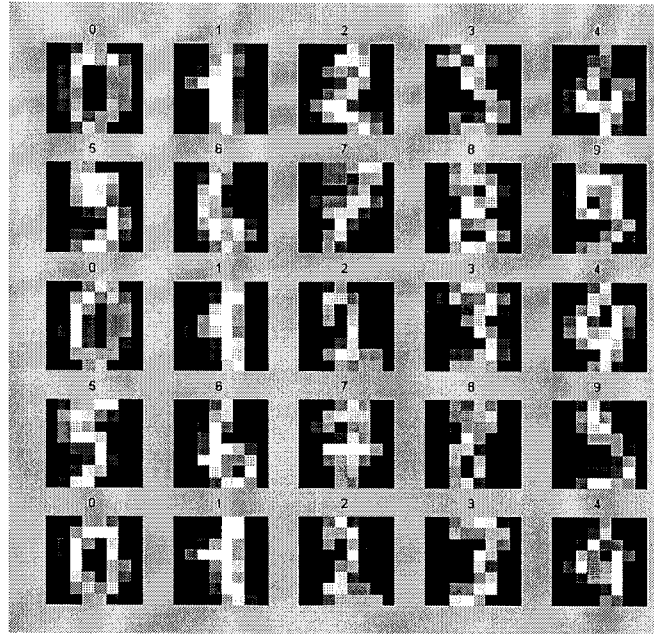


Figure 5.29 – Example data samples from the OCR database.

Table 5.12 – Instance distribution for the OCR database.

Class →	0	1	2	3	4	5	6	7	8	9
\mathcal{D}_1	250	250	250	0	0	250	250	0	0	250
\mathcal{D}_2	100	100	100	250	250	100	100	0	0	100
\mathcal{D}_3	0	0	50	150	150	50	50	400	400	0
Test	55	57	55	57	56	55	55	56	55	56

All algorithms were allowed to create five classifiers, for each dataset presented, for a total of 15 classifiers. Table 5.13 lists the class specific performance and the overall generalization performance after training with each dataset, \mathcal{D}_k . All performance numbers and the 95% confidence intervals were obtained through 10-fold cross validation. Figure 5.30 illustrates the generalization performance (on the test data) of each algorithm as new classifiers are added to the ensemble.

Table 5.13 - Generalization performance on OCR database.

Class →	0	1	2	3	4	5	6	7	8	9	Gen. ± CI	
Learn++	Φ_1	99%	97%	98%	-	-	98%	99%	-	-	98%	59%±0.4%
	Φ_2	99%	97%	99%	63%	63%	98%	99%	-	-	98%	71%±4.7%
	Φ_3	98%	97%	99%	95%	97%	97%	99%	36%	26%	95%	84%±3.6%
Learn++.NC	Φ_1	99%	96%	98%	-	-	97%	99%	-	-	98%	59%±0.3%
	Φ_2	99%	96%	98%	97%	98%	97%	99%	-	-	95%	78%±0.6%
	Φ_3	91%	51%	93%	96%	97%	95%	93%	97%	95%	80%	89%±2.3%
Learn++.UD	Φ_1	99%	96%	98%	-	-	97%	99%	-	-	97%	58%±0.5%
	Φ_2	99%	95%	97%	99%	98%	96%	97%	-	-	91%	77%±0.9%
	Φ_3	77%	43%	89%	98%	97%	91%	94%	99%	99%	60%	84%±2.8%
BEAST	Φ_1	99%	96%	99%	-	-	97%	99%	-	-	98%	58%±0.6%
	Φ_2	99%	96%	99%	90%	92%	97%	99%	-	-	97%	77%±1.1%
	Φ_3	98%	89%	98%	92%	93%	97%	98%	96%	89%	89%	94%±0.9%

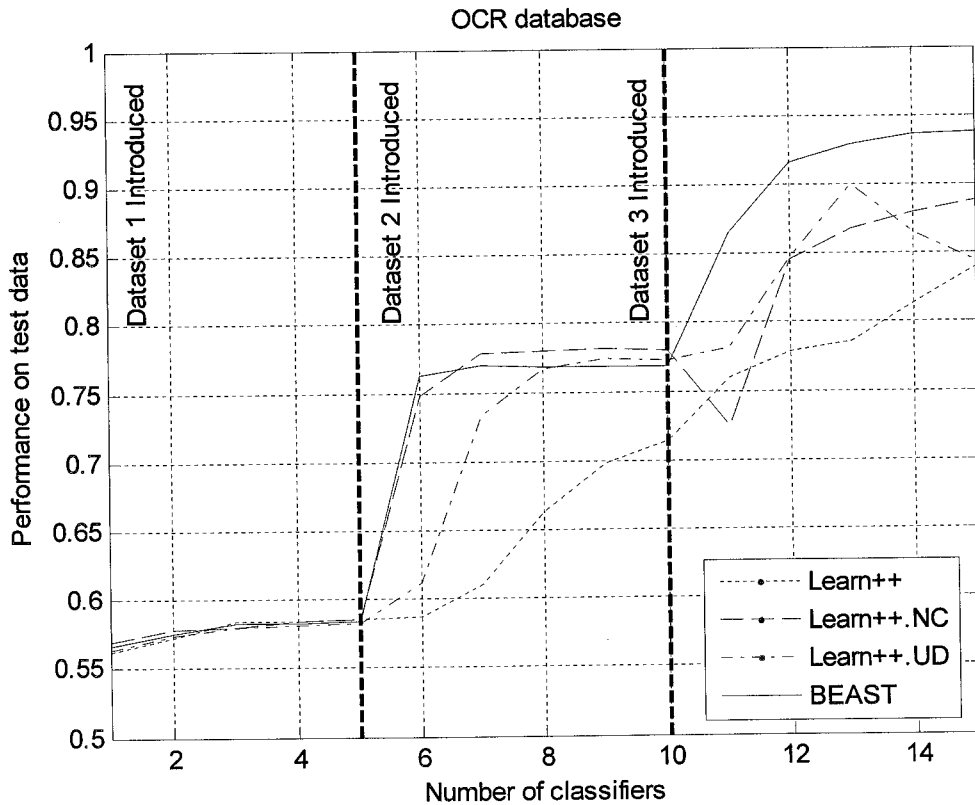


Figure 5.30 - Generalization performance vs. number of classifiers from OCR experiment.

These results show a considerable increase in the performance of BEAST over the other three algorithms. All four algorithms perform almost identically when training on the first dataset. The second training dataset introduces two new classes and provides some information on previously learned knowledge. Under this condition Learn++.NC, Learn++.UD and BEAST all perform in a similar manner, Learn++ performs worse on the newly introduced classes but would presumably finish learning the new classes if allowed to generate more classifiers, an experiment to validate this assumption is presented later in this section. The third training dataset presents a rather difficult problem: it introduces two new classes, completely removes instances from three previously learned classes, and contains a limited number of instances from the other classes. Under these harsh conditions, Learn++.UD becomes unsteady and only learns at the cost of previously learned information. Learn++.NC initially displays a drop in performance, due to its volatile nature when classes are introduced and removed; however, the algorithm begins to recover from this effect as classifiers are added to the ensemble. Learn++ is able to finish learning the classes introduced in \mathfrak{D}_2 but is unable to properly classify the classes introduced in \mathfrak{D}_3 . However, BEAST is able to learn the new information available from \mathfrak{D}_3 without sacrificing any significant information learned from \mathfrak{D}_1 and \mathfrak{D}_2 . Furthermore, the results offered by BEAST are extremely consistent, whereas the other algorithms perform better on some trials than on others, indicated by the 95% confidence intervals in Table 5.13 and the final performance probability density functions shown in Figure 5.31.

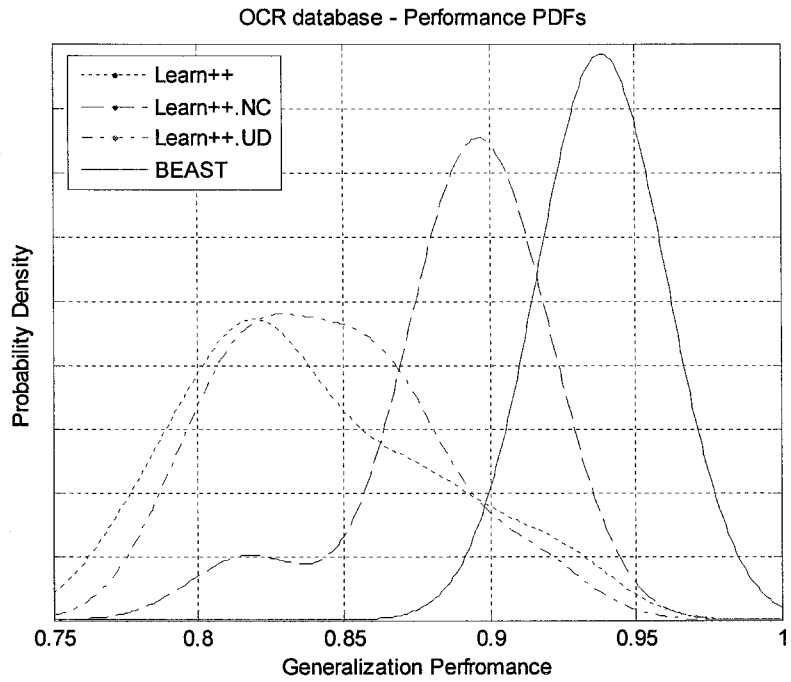


Figure 5.31 - Performance probability density functions from OCR experiment.

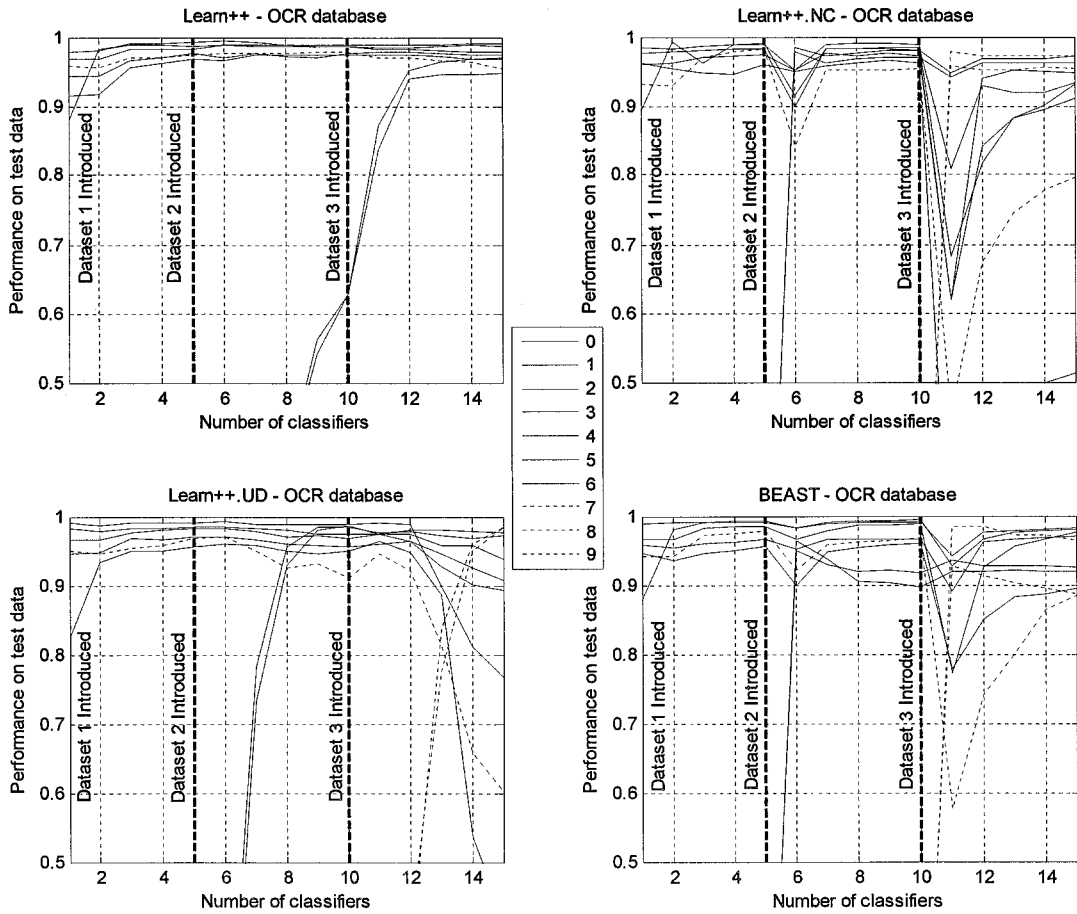


Figure 5.32 - Class specific generalization performances on the OCR database.

Figure 5.32 shows the instability of Learn++.UD when learning from \mathcal{D}_3 , the algorithm cannot retain previously learned information, while learning novel information. Learn++.NC performs well relative to Learn++.UD in that it does not completely forget previously learned classes to accommodate newly introduced classes. However, it does perform poorly on the classes whose instances are unavailable in \mathcal{D}_3 . BEAST also displays a drop in performance after the addition of the first classifier trained on \mathcal{D}_3 . As explained previously, this is due to the sub-ensemble fusion technique of the BEAST, where at the point where the performance drops the third sub-ensemble is comprised of only one classifier. Figure 5.32 clearly shows that the addition of classifiers to the third sub-ensemble allows BEAST to stabilize and learn the newly introduced information without sacrificing knowledge learned from \mathcal{D}_1 and \mathcal{D}_2 . Learn++ performs exceptionally well on 8 of the 10 classes, the two classes it performs poorly on are the two classes introduced in \mathcal{D}_3 . From looking at Figure 5.32 one may think to allow the Learn++ algorithm to generate more classifiers on \mathcal{D}_3 . Figure 5.33 shows the results of allowing Learn++ to generate 15 classifiers on \mathcal{D}_3 as opposed to 5.

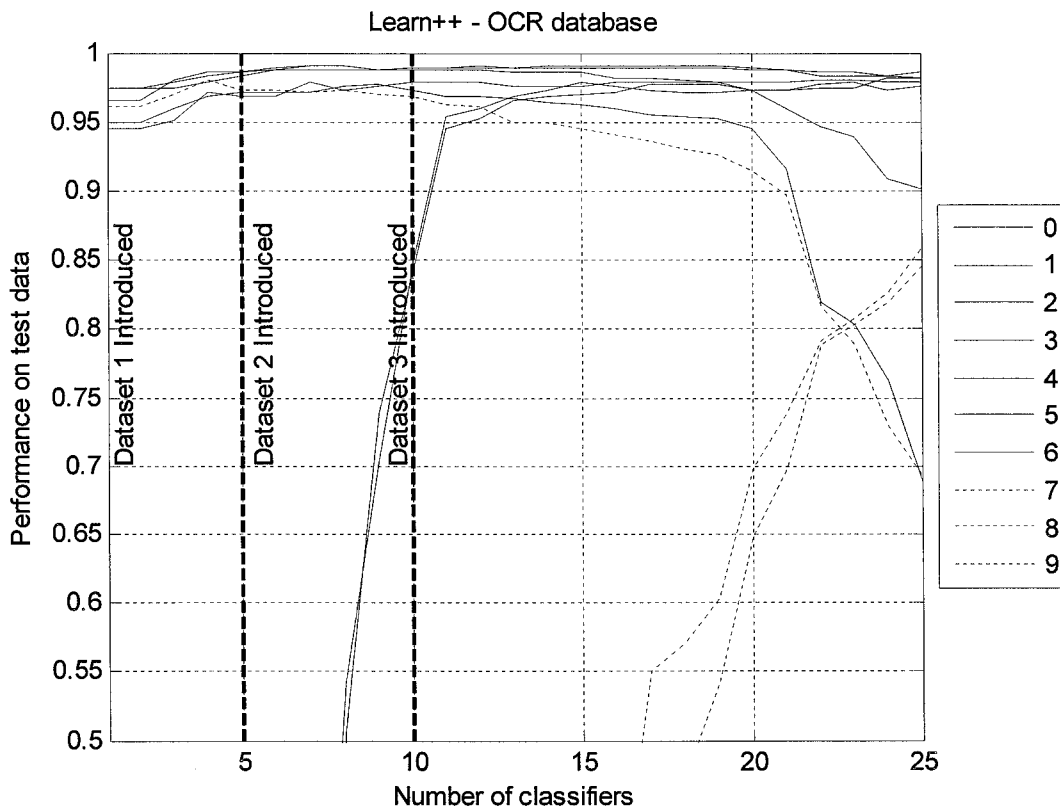


Figure 5.33 - Class specific generalization performance of Learn++ with 10 additional classifiers.

Figure 5.33 shows that Learn++ eventually learns the two classes introduced in \mathfrak{D}_3 . However, as well as learning the classes in \mathfrak{D}_3 Learn++ also loses information from the classes absent in \mathfrak{D}_3 .

CHAPTER 6

CONCLUSIONS

Three novel incremental learning algorithms, Learn++.NC, Learn++.UD, and BEAST are introduced in this thesis. The algorithms were all developed to solve various unbalanced data problems common to incremental learning. Learn++.NC uses preliminary decisions in assigning voting weights allowing the algorithm to dynamically adjust the classifier voting weights for each test instance. The approach overcomes the out-voting problem inherent in the original version of Learn++ and prevents proliferation of unnecessary classifiers. However, this efficiency is only prevalent in those cases where one or several new classes are introduced with subsequent datasets.

Learn++.UD is specifically designed to handle unbalanced data between datasets in incremental learning settings. Such imbalances in data distribution can cause the generalization performance to decrease with additional classifiers. The novelty of Learn++.UD is in its use of class conditional weight factors to assign voting weights to classifiers in the ensemble. For each classifier, this factor is the ratio of the number of instances from a particular class used for training, to the number of instances from that class used for training all classifiers in the ensemble. The actual voting weights are then determined as individual training performances of the classifiers, adjusted by the class conditional weight factors.

A primary goal of this research was to develop an algorithm capable of incremental learning in the harshest environments. This could include unbalanced data between classes, or between datasets, or the introduction and/or removal of all information from certain classes. This goal was only partially attained by Learn++.NC

and Learn++.UD. The BEAST algorithm introduced in this research meets the goals for learning under such harsh conditions. The success of the algorithm is due to the seamless combination of several contradicting and complimentary methods of solving unbalanced data problems. The first of these methods is the class specific weights found in Learn++.UD. Second, the concept of a normalized preliminary confidence, introduced by Learn++.NC. This preliminary confidence can then be used and modified to adapt to the current problem by calculating the preliminary confidences of each sub-ensemble and applying a transfer function to them. In this work, the function was designed to make it more difficult to classify data that was abundantly available during training. Finally, the sub-ensemble confidences are updated one last time such that the weights of sub-ensembles trained with more instances of a specific class are higher; allowing sub-ensembles with more “experience” on particular classes to be weighted higher when choosing that class. These methods by themselves are biased towards certain types of problems; however, when properly combined they balance each other out to create a robust algorithm that is suited for a wide variety of incremental learning problems.

6.1 *Summary of Experimental Findings*

The BEAST algorithm displays a number of favorable results through simulations on synthetic and real world databases. Even under some of the harshest incremental learning conditions, the algorithm continues to consistently perform well. However, it should be noted that most of the experiments in Chapter 5 were designed to simulate the conditions where previous algorithms fail and the BEAST prevails. There were a multitude of experiments conducted where other algorithms performed as well as, but not better than, the BEAST algorithm.

6.2 *Contributions of this Work*

The goal of this research was to design an algorithm that is capable of incrementally learning from severely unbalanced training datasets. This goal was obtained through the introduction of three novel incremental learning algorithms:

- 1) Learn++.NC, which uses the concept of dynamically adjusting classifier voting weights based on the hypotheses of other classifiers. This approach has been shown, through experimental results, to quickly and efficiently learn new classes that are introduced in subsequent training datasets.
- 2) Learn++.UD is designed to compensate for cardinality differences between training datasets by using class-conditional voting weights that are a function the amount of instances used in training.
- 3) BEAST adopts the novel concepts in Learn++.NC and Learn++.UD and combines them with additional methods for handling a variety of unbalanced data problems.

A comparison between Learn++ and the three aforementioned algorithms was carried out to characterize each algorithm's behavior on incremental learning problems complicated by different unbalanced data.

6.3 *Recommendations for Future Work*

Although the BEAST algorithm has shown tremendous increases in performance over its predecessors, there are several things that should be explored to further increase the algorithms capabilities.

6.3.1 Preliminary Confidence Transfer Function

During the course of this research only one transfer function was explored. Being one of the key elements of the algorithms success this is an important matter for future research.

Perhaps interesting results can be obtained by using class specific performances measures to modify these confidences. The algorithm would probably benefit more from performance measures similar to positive predicted value or negative predicted value. The goal would be to incorporate as much knowledge as can be inferred from the problem to intelligently fuse both the classifiers and the sub-ensembles.

6.3.2 Sub-Ensemble Combination Techniques

Once the preliminary confidence of each sub-ensemble is modified the current algorithm merely sums them and chooses the class with the maximum confidence. This leaves room for much expansion for existing classifier combination techniques to be used for combining the sub-ensembles. One worth noting is decision templates, where a decision profile for each class and for each sub-ensemble can be calculated. Then, when presented with a new dataset the previously generated sub-ensembles can create and/or update their profiles based on the newly available data, potentially allowing these sub ensembles to learn what they do not know and adjust themselves accordingly. Allowing the algorithm to learn from even harsher environments than those presented in this work.

Consider a problem where one dataset contains instances from one set of classes and a following dataset contains instances from a completely different set of classes without any information on previous classes. Decision templates could conceivably be used to solve this problem by allowing the first sub-ensemble to learn what it does not know and dynamically adjust its weights accordingly.

6.3.3 Incorporation of Prior Knowledge

The current version of the BEAST algorithm is designed to maximize the performance of each individual class. While this is a logical goal, there are many cases where

information about the problem would warrant the desire to weight one class's performance over another's. Furthermore, this could be extended to use information about the cost of different decisions. This prior information could easily be included in the preliminary confidence transfer function or in calculating the class specific weights. The theme of the algorithm is to extract as much information as possible from the environment and use it to dynamically and intelligently fuse both classifiers and sub-ensembles.

REFERENCES

- [1] R. O. Duda, P. E. Hart, and D. Stork, *Pattern Classification* Wiley-Interscience, 2001.
- [2] J. Murre, "Transfer of learning in back-propagation and in related neural network models," *Connectionist models of memory and language*, pp. 73-94, 1995.
- [3] A. V. Robins and M. R. Frean, "Learning and Generalization in a Stable Network," *Proceedings of the Conference on Neural Information Processing and Intelligent Information Systems*, Singapore: Springer-Verlag, 1998, pp. 314-317.
- [4] E. M. Gold, "Language identification in the limit," *Information and Control*, vol. 10, no. 5, pp. 447-474, May 1967.
- [5] K. P. Jantke, "Types of incremental learning," AAAI Press, 1993, pp. 26-32.
- [6] A. Sharma, "Note on batch and incremental learnability," *Journal of Computer and System Sciences*, vol. 56, no. 3, pp. 272-276, 1998.
- [7] S. Lange and G. Grieser, "On the power of incremental learning," in *Theoretical Computer Science*, 288 ed Tokyo, Japan: Elsevier Science B.V., 2002, pp. 277-307.
- [8] G. A. Carpenter and S. Grossberg, "Art of adaptive pattern recognition by a self-organized neural network," *Computer*, vol. 21, no. 3, pp. 77-88, 1988.
- [9] S. Grossberg, "Nonlinear neural networks: principles, mechanisms and architectures," *Neural Networks*, vol. 1, no. 1, pp. 17-61, 1988.
- [10] A. V. Robins, "Catastrophic forgetting, rehearsal, and pseudorehearsal," *Connection Science*, vol. 7, pp. 123-146, 1995.
- [11] D. L. Silver and R. E. Mercer, "The task rehearsal method of life-long learning: Overcoming impoverished data," *15th Conference of the Canadian Society for Computational Studies of Intelligence*, 2002, pp. 90-101.

- [12] R. French, "Catastrophic forgetting in connectionist networks: causes, consequences and solutions," *Trends in Cognitive Sciences*, vol. 3, no. 4, pp. 128-135, 1999.
- [13] R. Polikar, L. Udpa, S. S. Udpa, and V. Honavar, "Learn++: an incremental learning algorithm for multilayer perceptron networks," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 6 ed Istanbul, Turkey: IEEE, Piscataway, NJ, USA, 2000, pp. 3414-3417.
- [14] R. Polikar, "Learn++: An incremental learning algorithm based on psychophysiological models of learning," in *23rd Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, 1 ed Istanbul, Turkey: IEEE EMBS, 2001, pp. 672-675.
- [15] R. Polikar, L. Udpa, S. S. Udpa, and V. Honavar, "Learn++: An incremental learning algorithm for supervised neural networks," *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews*, vol. 31, no. 4, pp. 497-508, 2001.
- [16] R. Polikar, J. Byorick, S. Krause, A. Marino, and M. Moreton, "Learn++: A classifier independent incremental learning algorithm for supervised neural networks," 2 ed Honolulu, HI: IEEE, 2002, pp. 1742-1747.
- [17] R. Polikar, L. Udpa, S. Udpa, and V. Honavar, "An incremental learning algorithm with confidence estimation for automated identification of NDE signals," *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, vol. 51, no. 8, pp. 990-1001, 2004.
- [18] M. Muhlbaier, A. Topalis, and R. Polikar, "Learn++.MT: A New Approach to Incremental Learning," 5th Int. Workshop on Multiple Classifiers Systems Calagari, Italy: Springer, 2004, pp. 52-61.
- [19] M. Muhlbaier, A. Topalis, and R. Polikar, "Incremental learning from unbalanced data," in *IEEE International Conference on Neural Networks - Conference Proceedings*, 2 ed Budapest, Hungary: IEEE, Piscataway, NJ, USA, 2004, pp. 1057-1062.
- [20] M. Muhlbaier, A. Topalis, and R. Polikar, "Dynamically Updated Weighted Majority Voting for Efficient Incremental Learning of New Classes Using an Ensemble of Classifiers Approach," Glassboro, NJ: Rowan University, 2006.

- [21] Y. Kim and W. Nick Street, "A streaming ensemble algorithm (SEA) for large-scale classification," in *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* San Francisco, CA, United States: Association for Computing Machinery, 2001, pp. 377-382.
- [22] L. Fu, H. h. Hsu, and J. C. Principe, "Knowledge-based approach to supervised incremental learning," in *IEEE International Conference on Neural Networks - Conference Proceedings*, 3 ed Orlando, FL, USA: IEEE, Piscataway, NJ, USA, 1994, pp. 1793-1798.
- [23] C. A. Hung and S. F. Lin, "Incremental learning neural network for pattern classification," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 13, no. 6, pp. 913-928, 1999.
- [24] J. Mandziuk and L. Shastri, "Incremental class learning approach and its application to handwritten digit recognition," *Information Sciences*, vol. 141, no. 3-4, pp. 193-217, 2002.
- [25] V. R. de Angulo and C. Torras, "On-line learning with minimal degradation in feedforward networks," *IEEE Transactions on Neural Networks*, vol. 6, no. 3, pp. 657-668, 1995.
- [26] V. Ruiz de Angulo and C. Torras, "Sequential learning in feedforward networks: proactive and retroactive interference minimization," Barcelona, Spain: Institut de Robotica i Informatica Industrial (CSIC-UPC), 2002.
- [27] G. A. Carpenter and S. Grossberg, "A self organizing neural network for supervised learning, recognition, and prediction," 30 ed 1000, pp. 38-49.
- [28] 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 Transactions on Neural Networks*, vol. 3, no. 5, pp. 698-713, 1992.
- [29] P. Ramuhalli, R. Polikar, L. Udpa, and S. S. Udpa, "Fuzzy ARTMAP network with evolutionary learning," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 6 ed Istanbul, Turkey: IEEE, Piscataway, NJ, USA, 2000, pp. 3466-3469.

- [30] M. Georgiopoulos, A. Koufakou, G. C. Anagnostopoulos, and T. Kasparis, "Overtraining in Fuzzy ARTMAP: Myth or reality?," in *Proceedings of the International Joint Conference on Neural Networks*, 2 ed Washington, DC: IEEE, 2001, pp. 1186-1190.
- [31] I. Dagher, M. Georgiopoulos, G. L. Heileman, and G. Bebis, "Ordering algorithm for pattern presentation in fuzzy ARTMAP that tends to improve generalization performance," *IEEE Transactions on Neural Networks*, vol. 10, no. 4, pp. 768-778, 1999.
- [32] G. Tontini, "Robust learning and identification of patterns in statistical process control charts using a hybrid RBF Fuzzy Artmap neural network," in *IEEE International Conference on Neural Networks - Conference Proceedings*, 3 ed Anchorage, AK, USA: IEEE, Piscataway, NJ, USA, 1998, pp. 1694-1699.
- [33] R. E. Schapire, "The Strength of Weak Learnability," *Machine Learning*, vol. 5, no. 2, pp. 197-227, June 1990.
- [34] M. R. Ahmadzadeh, M. Petrou, and K. R. Sasikala, "Dempster-Shafer combination rule as a tool to classifier combination," in *International Geoscience and Remote Sensing Symposium (IGARSS)*, 6 ed Honolulu, HI, USA: IEEE, Piscataway, NJ, USA, 2000, pp. 2429-2431.
- [35] L. A. Alexandre, A. C. Campilho, and M. Kamel, "On combining classifiers using sum and product rules," *Pattern Recognition Letters*, vol. 22, no. 12, pp. 1283-1289, 2001.
- [36] R. Battiti and A. M. Colla, "Democracy in neural nets: voting schemes for classification," *Neural Networks*, vol. 7, no. 4, pp. 691-707, 1994.
- [37] M. G. Bello, "Enhanced training algorithms, and integrated training/architecture selection for multilayer perceptron networks," *IEEE Transactions on Neural Networks*, vol. 3, no. 6, pp. 864-875, 1992.
- [38] F. Roli, G. Giacinto, and G. Vernazza, "Methods for Designing Multiple Classifier Systems," 2nd Int. Workshop on Multiple Classifier Systems, Cambridge, UK, 2001, pp. 78-87.
- [39] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123-140, 1996.

- [40] H. Drucker, C. Cortes, L. D. Jackel, Y. LeCun, and V. Vapnik, "Boosting and other ensemble methods," *Neural Computation*, vol. 6, no. 6, pp. 1289-1301, 1994.
- [41] Y. Freund and R. E. Schapire, "Experiments with a new boosting algorithm," *Machine Learning: Proceedings of the Thirteenth International Conference*, 1996, pp. 148-156.
- [42] Y. Freund and R. E. Schapire, "Decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119-139, 1997.
- [43] Y. Freund, "Data Filtering and Distribution Modeling Algorithms for Machine Learning," University of California at Santa Cruz, 1990.
- [44] Y. Freund, "Boosting a weak learning algorithm by majority," *Information and Computation*, vol. 2, no. 121, pp. 256-285, Oct.1995.
- [45] N. Littlestone and M. Warmuth, "Weighted majority algorithm," *Information and Computation*, vol. 108, pp. 212-261, 1994.
- [46] A. Gangardiwala and R. Polikar, "Dynamically weighted majority voting for incremental learning and comparison of three boosting based approaches (in press)," *IEEE Int. Joint Conference on Neural Networks*, 2005.
- [47] M. Muhlbaier, A. Topalis, and R. Polikar, "Ensemble Confidence Estimates Posterior Probability," *6th Int. Workshop on Multiple Classifier Systems*, Monterey, CA: 2005, pp. 326-335.
- [48] L. Shapley and B. Grofman, "Optimizing group judgmental accuracy in the presence of interdependencies," *Public Choice (Historical Archive)*, vol. 43, no. 3, pp. 329-343, Jan.1984.
- [49] P. J. Boland, "Majority system and the Condorcet jury theorem," *Statistician*, vol. 38, no. 3, pp. 181-189, 1989.
- [50] D. Berend and J. Paroush, "When is Condorcet's Jury Theorem valid?," *Social Choice and Welfare*, vol. 15, no. 4, pp. 481-488, Aug.1998.

- [51] J. Kittler, M. Hatef, R. P. W. Duin, and J. Mates, "On combining classifiers," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 3, pp. 226-239, 1998.

- [52] L. I. Kuncheva, "A theoretical study on six classifier fusion strategies," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 2, pp. 281-286, 2002.

- [53] Blake C.L. and C. J. Merz, "UCI Repository of Machine Learning Database at Irvine CA," 2005.

- [54] R. Polikar, "Ensemble Based Systems in Decision Making," *IEEE Circuits and Systems Magazine*, In Press.

APPENDIX A

EXPERIMENT WITH *BEAST*

An experiment on a synthetic dataset will be shown to illustrate how the methods introduced in *BEAST* works. In particular, the steps taken by *BEAST*-ED to create and calculate the preliminary confidence are analyzed and discussed. The database used for this experiment is contained in a 2-D feature space with four classes with distribution information given in Table A.1 and shown in Figure A.1.

Table A.1 - Gaussian distribution information for *BEAST* experiment.

	Class 1	Class 2	Class 3	Class 4
μ_1	1	1	-1	-1
μ_2	-1	1	-1	1
Variance	0.35	0.35	0.35	0.35

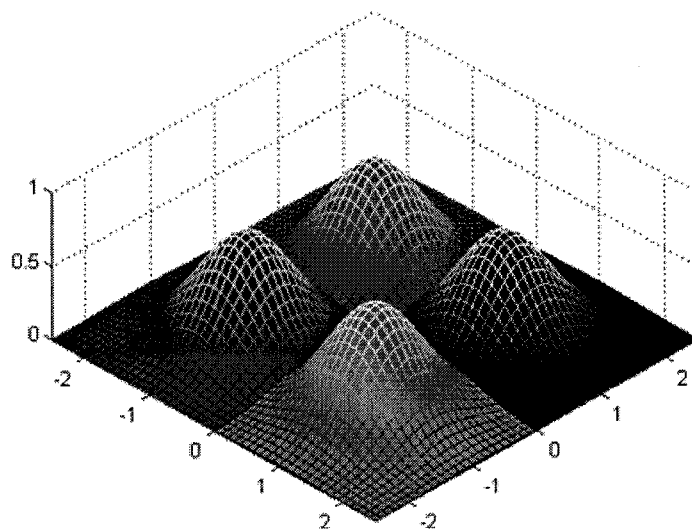


Figure A.1 – Example of the four PDF's corresponding to the classes described in Table A.1.

The instance distribution for this experiment was designed to create a difficult incremental learning problem with severely unbalanced data, shown in Table A.2. Each of the four training datasets contains a large number of instances from a particular class and few instances from other classes.

Table A.2 – Instance distribution for BEAST experiment.

	Class 1	Class 2	Class 3	Class 4
\mathcal{D}_1	100	10	-	-
\mathcal{D}_2	10	100	10	-
\mathcal{D}_3	10	10	100	10
\mathcal{D}_4	10	10	10	100
Test Data	200	200	200	200

Each training dataset, \mathcal{D}_1 - \mathcal{D}_4 , was used to create ten MLP classifiers, each with identical network architecture, 0.05 error goal and 25 hidden layer nodes. The primary purpose of this experiment is to illustrate how BEAST uses the novel methods introduced in this work to incrementally learn under hostile conditions. This illustration is done by displaying the preliminary confidence over the entire feature space, for each sub-ensemble, for each step in BEAST-ED. Recall that the first step in BEAST-ED is to calculate the preliminary confidence, $P_{k,c}$, for each sub-ensemble, k , and each class, c , Equation (A.1).

$$P_{k,c} = \frac{\sum_{t: h_t(x_i)=c} w_{t,c}}{Z_{k,c}}, \quad \text{for } t = eT_{k-1}, \dots, eT_k \quad c = 1, \dots, C \text{ and } k = 1, \dots, K \quad (\text{A.1})$$

where

$$Z_{k,c} = \sum_{t=eT_k}^{eT_k+T_k} w_{t,c}, \quad \text{for } c = 1, \dots, C \text{ and } k = 1, \dots, K \quad (\text{A.2})$$

The following figures show $P_{k,c}$ calculated over the entire feature space in **Ak**, i.e. **A2** shows $P_{2,c}$. The preliminary confidence of the entire ensemble is also calculated (Equation A.3) and shown in **A5**.

$$P_c = \sum_{k=1}^K P_{k,c}, \quad \text{for } c=1, \dots, C \quad (\text{A.3})$$

Next $\bar{P}_{k,c}$ is calculated by applying the preliminary confidence transfer function, Equation A.4.

$$\bar{P}_{k,c} = f(P_{k,c}) = (P_{k,c})^{N_{k,c}/\min(N_k)}, \quad \text{for } c=1, \dots, C \text{ and } k=1, \dots, K \quad (\text{A.4})$$

Figures A.2-A.5 show $\bar{P}_{k,c}$, calculated over the entire feature space, in **Bk**. The adjusted preliminary confidence of entire ensemble is also calculated (Equation A.5) and shown in **B5**.

$$\bar{P}_c = \sum_{k=1}^K \bar{P}_{k,c}, \quad \text{for } c=1, \dots, C \quad (\text{A.5})$$

Finally $\bar{\bar{P}}_{k,c}$ is calculated by adjusting the sub-ensemble preliminary confidence on each class according its relative experience on that class, (Equation A.6).

$$\bar{\bar{P}}_{k,c} = \bar{P}_{k,c} \frac{N_{k,c}}{\sum_{j=1}^K N_{j,c}}, \quad \text{for } c=1, \dots, C \text{ and } k=1, \dots, K \quad (\text{A.6})$$

Figures A.2-A.5 show $\bar{\bar{P}}_{k,c}$, calculated over the entire feature space, in **Ck**. The final confidence of the ensemble is then calculated according to Equation A.7 and shown in **C5**. Note that the final hypothesis of the algorithm is the class with the highest confidence.

$$\bar{\bar{P}}_c = \sum_{k=1}^K \bar{\bar{P}}_{k,c}, \quad \text{for } c=1, \dots, C \quad (\text{A.7})$$

Figures A.2-A.5 also show the actual training data, \mathfrak{D}_k , used for training each sub ensemble, in **Dk**. Last, D5 shows the posterior probability of the Bayes classifier, which can be used to compare with the preliminary confidence plots.

Figure Layout:

Row **A** shows the initial preliminary confidence for each sub-ensemble, row **B** shows the preliminary confidence after being updated by the transfer function, row **C** shows the preliminary confidence after being adjusted according to each sub-ensemble's relative experience, row **D** shows the data used to train each sub-ensemble. Columns **1-4** correspond to \mathfrak{D}_1 - \mathfrak{D}_4 and the associated sub-ensemble's, column **5** shows the combined preliminary confidence for the entire ensemble along with the posterior probability of the Bayes classifier.

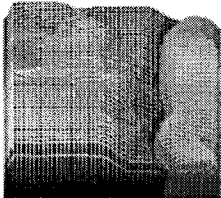
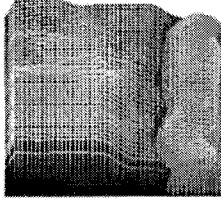
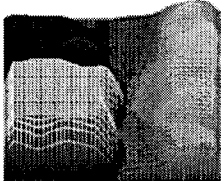
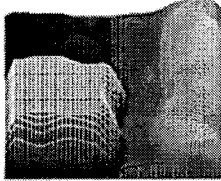
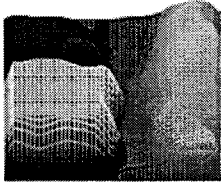
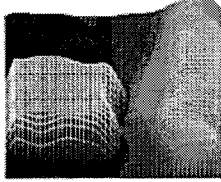
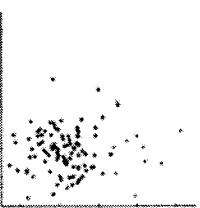
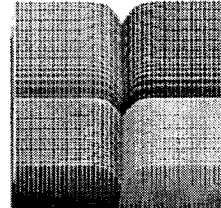
	\mathcal{D}_1	\mathcal{D}_2	\mathcal{D}_3	\mathcal{D}_4	Σ
P	A1 Classifier = 10 	A2	A3	A4	A5 
\bar{P}	B1 	B2	B3	B4	B5 
$\bar{\bar{P}}$	C1 	C2	C3	C4	C5 
Training Data	D1 	D2	D3	D4	D5 
					Bayes Classifier

Figure A.2 – Preliminary Confidence plots after training on \mathcal{D}_1 .

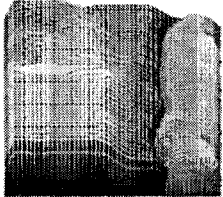
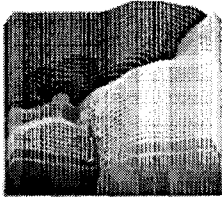
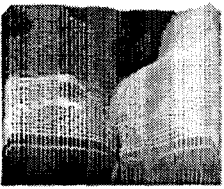
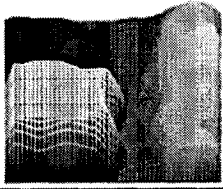

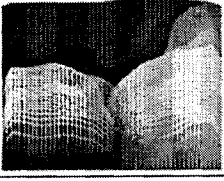
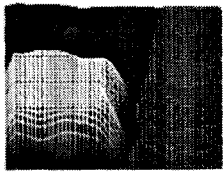
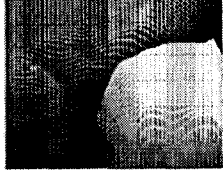
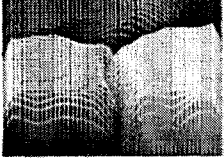
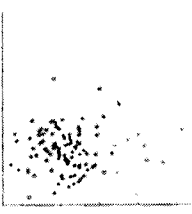
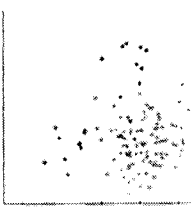
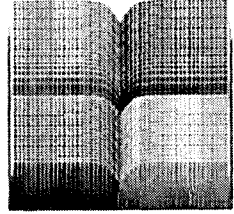
	\mathcal{D}_1	\mathcal{D}_2	\mathcal{D}_3	\mathcal{D}_4	Σ
P	A1 Classifier = 20 	A2 	A3	A4	A5 
\bar{P}	B1 	B2 	B3	B4	B5 
$\bar{\bar{P}}$	C1 	C2 	C3	C4	C5 
Training Data	D1 	D2 	D3	D4	D5 
					Bayes Classifier

Figure A.3 – Preliminary Confidence plots after training on \mathcal{D}_2 .

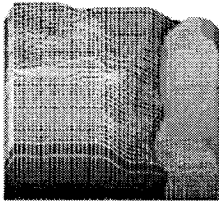
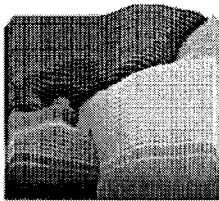
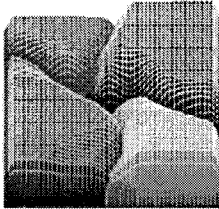
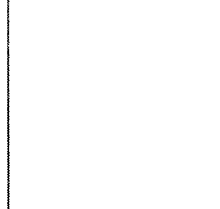
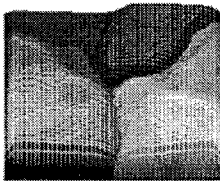
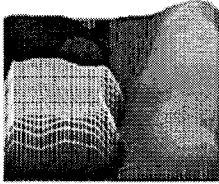
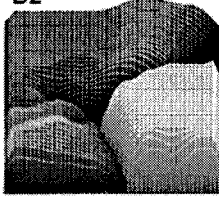
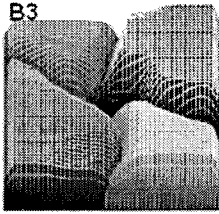
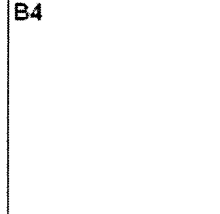
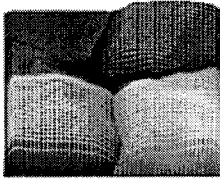
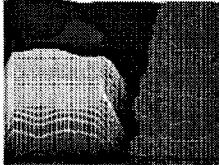
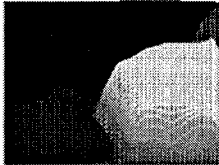
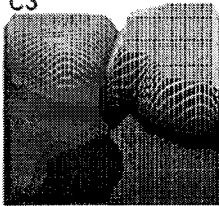
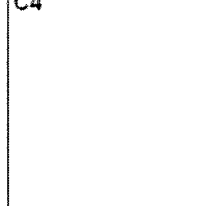
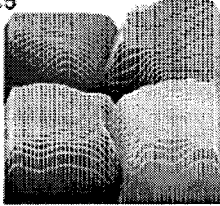
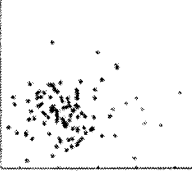
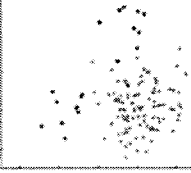
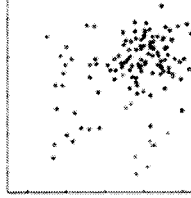
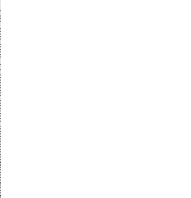
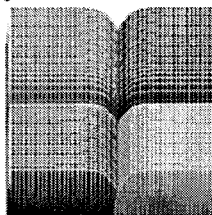
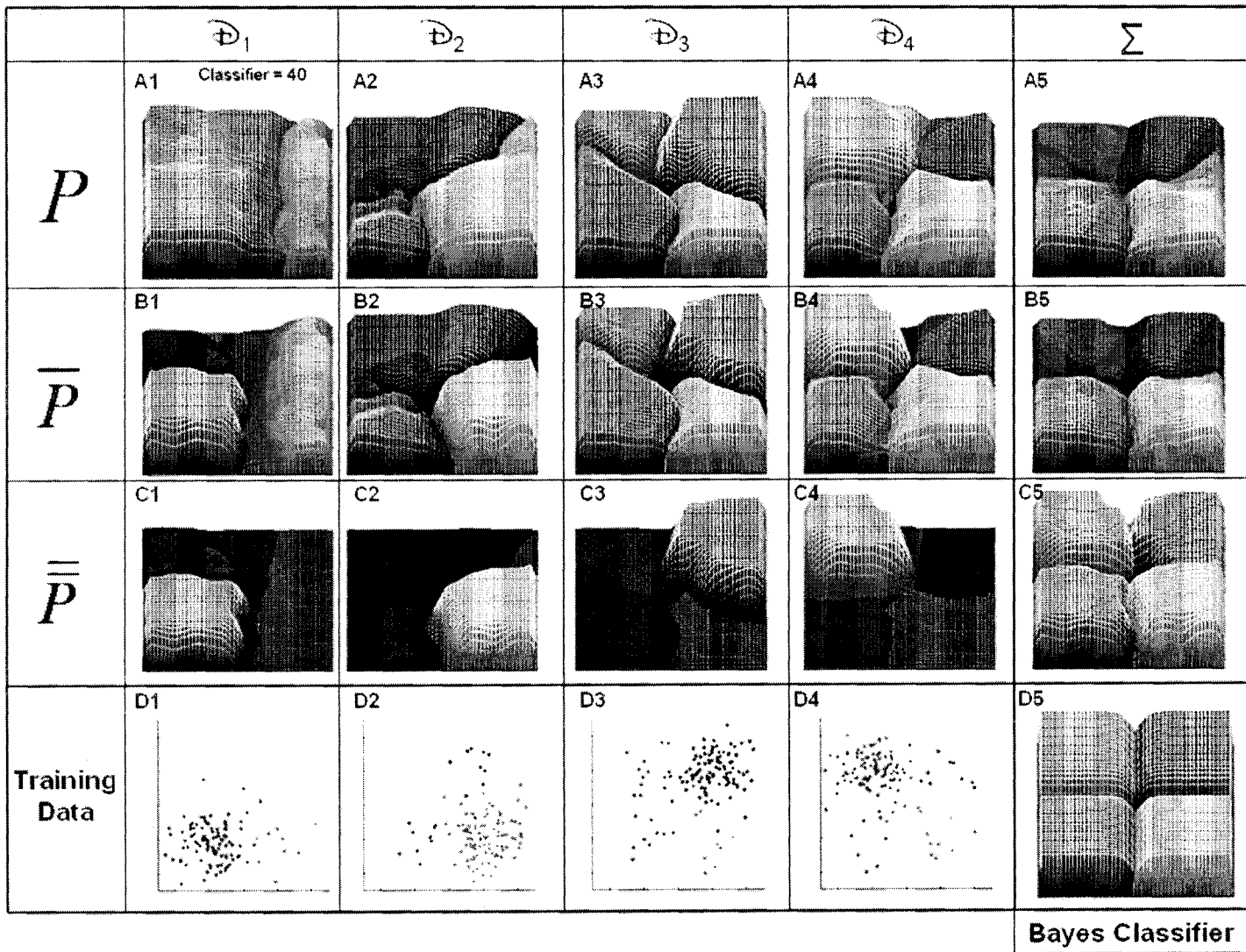
	\mathcal{D}_1	\mathcal{D}_2	\mathcal{D}_3	\mathcal{D}_4	Σ
P	A1 Classifier = 30 	A2 	A3 	A4 	A5 
\bar{P}	B1 	B2 	B3 	B4 	B5 
$\bar{\bar{P}}$	C1 	C2 	C3 	C4 	C5 
Training Data	D1 	D2 	D3 	D4 	D5 
					Bayes Classifier

Figure A.4 – Preliminary Confidence plots after training on \mathcal{D}_3 .

Figure A.5 – Preliminary Confidence plots after training on \mathcal{D}_4 .



Discussion of Figure A.2:

Note the preliminary confidence shown in **A1** is heavily biased towards the green class since there were a large number of samples from the green class available in training, **D1**. **B1** shows the preliminary confidence after it has been adjusted with the transfer function. The bias seen in **A1** is suppressed and the sub-ensemble is able to properly learn both classes despite unbalanced training data. **C1** shows no difference from **B1** because only one sub-ensemble existed, likewise, column **5** is the same as column **1**.

Discussion of Figure A.3:

There is a large drop in confidence on class green in **C1**, compared to **C1** in Figure A.2. This is because of the normalization step which adjusts the confidence of each sub-ensemble according to their relative experience. In this case the second sub-ensemble has been trained with ten times the number of instances from class green as the first sub-ensemble. The preliminary confidence of the second sub-ensemble is heavily biased towards class green, **A2**; however, once the transfer function is applied, which compensates for unbalanced data, the bias towards class green is suppressed, **B2**. Finally the confidence on class red is reduced, **C2**, since the first sub-ensemble has been trained with more instances from class red. The combined ensemble's preliminary confidence is shown in column **5**.

Discussion of Figure A.4:

A large confidence drop on class blue can be seen between **C2** in Figure A.3 and Figure A.4. This drop is due to the algorithm adjusting the weight of each sub-ensemble according to their relative experience on each class. **A5** shows the preliminary decision of the ensemble, at this step there are no instances that would be classified as magenta.

After compensating for the unbalanced data the ensemble is able to classify some instances from class magenta, **B5**. Finally, the last step is taken to normalize the preliminary confidence, **C5**. This final confidence shows a strong resemblance to the posterior probability of the Bayes classifier, **D5**, which is quite an achievement considering the data used to train the ensemble **D1-D3**.

Discussion of Figure A.5:

The preliminary confidence of each sub-ensemble, **A1-A4**, is biased and clearly needs to be adjusted to compensate for the unbalanced training data. After applying the transfer function, the preliminary confidence, of each sub-ensemble, is able to better represent the true underlying distribution, **B1-B4**. **C1-C4** clearly shows the effects of normalizing the preliminary confidence according to each sub-ensembles experience with particular classes. The final confidence, **C5**, is remarkably close to that of the Bayes classifier, **D5**. After training on all datasets, $\mathcal{D}_1-\mathcal{D}_4$, the ensemble is able to efficiently learn all four classes, even under the harsh nature of this incremental learning problem.

APPENDIX B

RESULTS USING VALIDATION DATA

Chapter 5 presented results on three synthetic databases and two experimental databases. For all simulations the number of classifiers generated by each ensemble was determined before training. However, in many applications the number of classifiers needed to learn from a dataset varies between trials and between algorithms. For example, on scenarios which suffer from the outvoting problem, the number of classifiers needed by Learn++ to outvote existing classifiers is entirely dependent on the weights of the classifiers; thus each algorithm will be allowed to create as many classifiers as necessary on each training data, \mathcal{D}_k , and automatically determine the optimal number of classifiers to retain based on its performance on validation, V_k . Learn++, Learn++.NC, and Learn++.UD retain the number of classifiers which yield in the maximum ensemble performance on V_k . BEAST works in a similar way; however, it calculates the maximum performance of sub-ensemble k on V_k .

Synthetic Experiment 1 – Incremental Learning

The database used in this experiment is described in detail in Section 5.2.1. Table B.1 shows the instance distribution of this experiment, which is identical to Table 5.2 with the addition of validation data, V_1 - V_3 .

Table B.1 – Instance distribution for experiment 1, with validation data.

	Class 1	Class 2	Class 3	Class 4
\mathcal{D}_1	100	100	0	0
\mathcal{D}_2	50	50	150	0
\mathcal{D}_3	50	50	50	200
V_1	50	50	0	0
V_2	25	25	75	0
V_3	25	25	25	100
Test Data	200	200	200	200

Each algorithm was allowed to generate a generous number of classifiers on each dataset and then determined the number of classifiers to retain, T_k , according to the performance on the validation data. The results of this test are shown in Table B.2, the format of this table is the same as Table 5.3 except there is an additional column to the left displaying the average number of classifiers retained from each training session along with a column to the right showing the maximum performance out of the 40 independent trials.

Table B.2 - Generalization performance from Experiment 1, with validation data

		$T_k \pm \text{CI}$	Class 1	Class 2	Class 3	Class 4	Gen. \pm CI	Max
Learn++	\mathcal{D}_1	6 ± 0.9	96.0%	96.3%	-	-	$49.2\% \pm 0.8\%$	62.1%
	\mathcal{D}_2	10 ± 1.6	95.0%	94.8%	94.4%	-	$71.1\% \pm 0.7\%$	72.4%
	\mathcal{D}_3	36 ± 2.5	93.5%	93.5%	83.6%	52.3%	$80.7\% \pm 1.7\%$	86.6%
Learn++.NC	\mathcal{D}_1	5 ± 0.9	95.1%	96.2%	-	-	$48.3\% \pm 0.8\%$	64.0%
	\mathcal{D}_2	4 ± 1.0	91.5%	92.7%	97.8%	-	$70.5\% \pm 0.3\%$	72.6%
	\mathcal{D}_3	6 ± 1.4	84.1%	85.5%	69.1%	91.3%	$82.5\% \pm 0.5\%$	85.3%
Learn++.UD	\mathcal{D}_1	4 ± 0.5	95.9%	96.3%	-	-	$48.0\% \pm 0.2\%$	49.1%
	\mathcal{D}_2	4 ± 0.8	92.0%	92.6%	97.8%	-	$70.6\% \pm 0.3\%$	72.0%
	\mathcal{D}_3	6 ± 1.3	83.5%	85.0%	71.5%	91.9%	$83.0\% \pm 0.4\%$	86.6%
BEAST	\mathcal{D}_1	5 ± 0.8	95.3%	96.4%	-	-	$47.9\% \pm 0.2\%$	49.4%
	\mathcal{D}_2	5 ± 0.9	93.5%	95.2%	95%	-	$70.9\% \pm 0.3\%$	72.8%
	\mathcal{D}_3	5 ± 1.3	87.7%	89.2%	80%	80.4%	$84.3\% \pm 0.4\%$	87.1%

Table B.2 shows a similar performance between all four algorithms; however, there is a large difference between the numbers of classifiers retained by Learn++. Recall that this

experiment was designed to test each algorithm’s ability to incrementally learn new classes; \mathfrak{D}_2 introduced a new class that was separable from the others, where \mathfrak{D}_3 introduced a new class that overlapped with previously learned classes. Learn++, on average, needed to generate 10 classifiers on \mathfrak{D}_2 to outvote the 6 classifiers trained on \mathfrak{D}_1 . However, Learn++ retained 36 classifiers, trained on \mathfrak{D}_3 , and still was not completely able to learn class 4. This observation is due to the nature of the outvoting problem.

Classifier voting weights are based on the performance on their own training data, thus the performance of classifiers on more separable problems tends to be higher; alternatively, classifiers trained on inseparable problem will be assigned lower voting weights. For example, classifiers generated on easier problems, such as those found in \mathfrak{D}_1 and \mathfrak{D}_2 , will have much higher weights than classifiers generated on a more difficult problem, as seen in \mathfrak{D}_3 . As a consequence of this weighting scheme, a large number of classifiers need to be generated on \mathfrak{D}_3 in order to outvote the previously trained classifiers.

Synthetic Experiment 2 – Unbalanced Data

The database used in this experiment is described in detail in Section 5.2.2. Again the instance distribution is identical to the original problem except for the validation data shown in Table B.3.

Table B.3 – Instance distribution for experiment 2, with validation data.

	Class 1	Class 2	Class 3	Class 4
\mathcal{D}_1	100	10	10	10
\mathcal{D}_2	100	100	10	10
\mathcal{D}_3	100	10	100	10
\mathcal{D}_4	100	10	10	100
V_1	50	5	5	5
V_2	50	50	5	5
V_3	50	5	50	5
V_4	50	5	5	50
Test Data	200	200	200	200

The results of this simulation using validation data are shown in Table B.4. All performance figures are very close to those presented in Table 5.6. The number of classifiers retained by each algorithm is similar to one another with the exception of Learn++ which retains several more classifiers on each dataset. BEAST significantly outperforms the other three algorithms using the fewest number of classifiers.

Table B.4 - Generalization performance on Experiment 2, with validation data

		$T_k \pm CI$	Class 1	Class 2	Class 3	Class 4	Gen. $\pm CI$	Max
Learn++	\mathcal{D}_1	8 ± 0.7	98.7%	66.2%	68.3%	80.1%	$78.3\% \pm 1.8\%$	86.6%
	\mathcal{D}_2	6 ± 0.9	98.0%	85.4%	64.7%	71.4%	$79.9\% \pm 1.5\%$	88.8%
	\mathcal{D}_3	7 ± 1.4	98.1%	79.0%	82.1%	68.0%	$81.8\% \pm 1.0\%$	87.4%
	\mathcal{D}_4	5 ± 1.7	98.6%	72.7%	76.0%	86.5%	$83.4\% \pm 0.8\%$	88.5%
Learn++.NC	\mathcal{D}_1	6 ± 0.7	98.7%	63.5%	64.6%	83.1%	$77.5\% \pm 1.8\%$	89.6%
	\mathcal{D}_2	6 ± 1.0	96.3%	92.9%	51.0%	49.2%	$72.3\% \pm 2.5\%$	85.4%
	\mathcal{D}_3	7 ± 1.2	97.0%	70.2%	90.0%	41.2%	$74.6\% \pm 1.7\%$	84.6%
	\mathcal{D}_4	7 ± 1.5	98.8%	45.2%	66.9%	93.6%	$76.1\% \pm 1.7\%$	86.1%
Learn++.UD	\mathcal{D}_1	6 ± 0.8	98.4%	69.1%	66.0%	83.4%	$79.2\% \pm 1.6\%$	89.4%
	\mathcal{D}_2	5 ± 1.0	94.2%	95.2%	61.0%	53.0%	$75.9\% \pm 2.2\%$	89.0%
	\mathcal{D}_3	6 ± 1.5	91.6%	88.0%	95.6%	33.7%	$77.2\% \pm 1.4\%$	86.8%
	\mathcal{D}_4	6 ± 1.7	94.1%	42.0%	69.9%	97.6%	$75.9\% \pm 2.2\%$	87.6%
BEAST	\mathcal{D}_1	6 ± 0.7	96.3%	76.3%	78.5%	84.6%	$83.9\% \pm 1.3\%$	90.3%
	\mathcal{D}_2	6 ± 1.0	96.0%	89.5%	77.8%	82.2%	$86.4\% \pm 0.7\%$	89.9%
	\mathcal{D}_3	5 ± 1.4	94.7%	88.9%	89.4%	78.7%	$87.9\% \pm 0.5\%$	90.9%
	\mathcal{D}_4	5 ± 1.6	94.8%	85.1%	86.5%	90.1%	$89.1\% \pm 0.5\%$	92.3%

Synthetic Experiment 3 - Spiral Database

The database used in this experiment is described in detail in Section 5.2.3. Table B.5 shows the validation data distribution along with the original data distribution.

Table B.5 – Instance distribution for swirl experiment, with validation data.

	Class 1	Class 2	Class 3	Class 4
\mathcal{D}_1	200	0	200	0
\mathcal{D}_2	0	200	0	200
\mathcal{D}_3	50	150	0	150
\mathcal{D}_4	0	150	50	150
V_1	100	0	100	0
V_2	0	100	0	100
V_3	25	75	0	75
V_4	0	75	25	75
Test Data	200	200	200	200

Table B.6 - Generalization performance on Experiment 2, with validation data

		$T_k \pm CI$	Class 1	Class 2	Class 3	Class 4	Gen. $\pm CI$	Max
Learn++	\mathcal{D}_1	6±0.9	98.3%	-	98.2%	-	49.1%±0.1%	49.9%
	\mathcal{D}_2	7±1.7	48.5%	69.5%	46.4%	70.1%	58.6%±1.3%	64.9%
	\mathcal{D}_3	7±1.4	84.1%	97.9%	6.5%	96.9%	71.3%±0.4%	74.4%
	\mathcal{D}_4	7±2.0	50.1%	98.4%	55.3%	98.3%	75.6%±2.0%	89.1%
Learn++.NC	\mathcal{D}_1	6±0.7	98.4%	-	98.1%	-	49.1%±0.1%	49.9%
	\mathcal{D}_2	12±1.2	0.2%	57.1%	65.7%	58.3%	45.3%±0.8%	50.7%
	\mathcal{D}_3	14±1.3	37.2%	62.5%	67.9%	58.4%	56.5%±1.5%	65.4%
	\mathcal{D}_4	6±1.6	34.9%	98.5%	76.0%	98.7%	77.0%±1.2%	84.6%
Learn++.UD	\mathcal{D}_1	6±0.7	98.1%	-	98.4%	-	49.1%±0.1%	49.8%
	\mathcal{D}_2	11±1.5	0.5%	98.1%	1.2%	97.7%	49.4%±0.3%	54.3%
	\mathcal{D}_3	12±1.9	77.8%	99.1%	2.1%	97.7%	69.2%±0.5%	72.4%
	\mathcal{D}_4	13±2.3	37.0%	98.0%	70.6%	98.8%	76.1%±2.7%	91.3%
BEAST	\mathcal{D}_1	6±0.8	98.2%	-	98.2%	-	49.1%±0.2%	49.9%
	\mathcal{D}_2	7±1.4	92.8%	55.5%	52.8%	25.1%	56.5%±1.3%	64.6%
	\mathcal{D}_3	6±1.8	91.5%	91.7%	66.0%	54.4%	75.9%±1.0%	80.8%
	\mathcal{D}_4	6±2.2	90.8%	92.8%	89.5%	93.3%	91.6%±0.4%	94.1%

The results from this simulation are shown in Table B.6. There is a large drop in performance for Learn++.NC and Learn++.UD compared to the results presented in

Table 5.9. This is because the validation data is subset of the training data and does not necessarily reflect the true distribution. In this experiment V_2 only contains instances from classes 2 and 4; thus, each algorithm will retain the number of classifiers which cause the ensemble to perform best on classes 2 and 4. A good performance on the validation data does not necessarily translate to a good performance on the test data. The BEAST algorithm is designed to work best when each sub-ensemble beast represents the data it is trained on. Therefore, the algorithm will not suffer when the distribution of the validation data does not represent the true underlying distribution of the environment.

Volatile Organic Compounds Recognition Database

The VOC database is described in detail in Section 5.3.1, the instance distribution for this experiment is shown in Table B.7, and the results shown in Table B.8.

Table B.7 – Instance distribution for the VOC database, with validation data

Class→	ET	OC	TL	TCE	XL
\mathcal{D}_1	0	25	0	25	25
\mathcal{D}_2	0	5	25	5	5
\mathcal{D}_3	25	5	5	5	5
V_1	0	15	0	15	15
V_2	0	3	15	3	3
V_3	15	3	3	3	3
Test	6	6	6	8	11

While the performances of all algorithms are similar, BEAST still shows a statistically significant increase, in final performance, over the other three algorithms.

Table B.8 - Generalization performance on VOC database, with validation data

		$T_k \pm CI$	Eth.	Oct.	Tol.	TCE	Xyl.	Gen. $\pm CI$	Max
Learn++	Φ_1	5 \pm 0.4	-	94.7%	-	92.5%	78.4%	58.1% \pm 0.9%	70.0%
	Φ_2	5 \pm 0.7	-	83.5%	95.8%	93.4%	68.6%	69.4% \pm 0.9%	80.5%
	Φ_3	9 \pm 1.1	82.2%	82.8%	93.7%	92.0%	64.5%	81.1% \pm 1.3%	92.7%
Learn++.NC	Φ_1	5 \pm 0.4	-	93.7%	-	91.0%	76.7%	57.0% \pm 0.9%	64.9%
	Φ_2	4 \pm 0.5	-	69.1%	98.6%	89.9%	67.2%	66.3% \pm 1.3%	80.5%
	Φ_3	5 \pm 0.8	95.6%	75.1%	91.8%	92.1%	57.2%	79.6% \pm 1.0%	90.2%
Learn++.UD	Φ_1	5 \pm 0.4	-	92.3%	-	90.8%	76.4%	56.6% \pm 0.9%	64.9%
	Φ_2	4 \pm 0.7	-	82.6%	98.2%	91.6%	70.5%	69.8% \pm 1.2%	81.1%
	Φ_3	6 \pm 1.0	96.3%	78.6%	95.4%	91.4%	59.1%	81.3% \pm 1.0%	92.7%
BEAST	Φ_1	5 \pm 0.4	-	93.2%	-	90.9%	75.1%	56.4% \pm 0.9%	64.9%
	Φ_2	4 \pm 0.5	-	90.6%	92.6%	91.5%	71.2%	70.4% \pm 0.9%	81.1%
	Φ_3	5 \pm 0.7	81.3%	90.5%	92.3%	92.1%	66.8%	83.8% \pm 1.0%	95.1%

Optical Character Recognition Database

The VOC database is described in detail in Section 5.3.2. The instance distribution for this problem is shown in Table B.9.

Table B.9 - Instance distribution for the OCR database, with validation data.

Class \rightarrow	0	1	2	3	4	5	6	7	8	9
Φ_1	200	200	200	0	0	200	200	0	0	200
Φ_2	100	100	100	250	250	100	100	0	0	100
Φ_3	0	0	50	150	150	50	50	400	400	0
V_1	50	50	50	0	0	50	50	0	0	50
V_2	20	20	20	50	50	20	20	0	0	20
V_3	0	0	10	30	30	10	10	80	80	0
Test	55	57	55	57	56	55	55	56	55	56

Table B.10 shows the results on the OCR database using validation data to determine the ensemble size. Learn++, Learn++.NC, and Learn++.UD perform worse compared to the results presented in Table 5.13.

Table B.10 - Generalization performance on OCR database, with validation data

		$T_k \pm \text{CI}$	1	2	3	4	5	6	7	8	9	0	Gen. \pm CI	Max
Learn++	\mathcal{D}_1	7 \pm 0.4	99	98	99	-	-	98	99	-	-	97	58.8% \pm 0.1%	59.6%
	\mathcal{D}_2	9 \pm 0.5	99	98	99	80	83	97	99	-	-	97	75.2% \pm 0.9%	79.2%
	\mathcal{D}_3	10 \pm 0.5	98	96	99	96	97	97	99	45	39	94	86.1%\pm1.0%	95.7%
Learn++.NC	\mathcal{D}_1	7 \pm 0.4	99	98	98	-	-	97	99	-	-	97	58.7% \pm 0.1%	59.2%
	\mathcal{D}_2	8 \pm 0.6	99	97	98	96	96	96	99	-	-	95	77.8% \pm 0.2%	79.0%
	\mathcal{D}_3	10 \pm 0.6	33	17	86	96	97	87	92	25	98	85	71.6%\pm0.6%	79.4%
Learn++.UD	\mathcal{D}_1	7 \pm 0.4	99	97	99	-	-	97	99	-	-	97	58.7% \pm 0.1%	59.4%
	\mathcal{D}_2	6 \pm 0.7	99	97	97	98	98	96	98	-	-	93	77.8% \pm 0.2%	79.4%
	\mathcal{D}_3	7 \pm 0.9	81	49	91	97	97	92	96	99	98	69	86.7%\pm0.8%	93.4%
BEAST	\mathcal{D}_1	7 \pm 0.4	99	98	99	-	-	98	99	-	-	97	58.7% \pm 0.1%	59.6%
	\mathcal{D}_2	7 \pm 0.6	99	98	99	93	94	97	99	-	-	97	77.6% \pm 0.2%	79.4%
	\mathcal{D}_3	7 \pm 0.7	97	86	98	95	94	97	99	94	85	89	93.4%\pm0.5%	96.9%

