INCREMENTAL LEARNING OF CONCEPT DRIFT FROM IMBALANCED DATA

by
Gregory Charles Ditzler

A Thesis
Submitted to the
Department of Electrical & Computer Engineering
College of Engineering
In partial fulfillment of the requirement
For the degree of
Master of Science
at
Rowan University
April, 2011

Thesis Advisor: Robi Polikar, Ph.D.

Acknowledgments

First of all, I would like to thank my advisor Dr. Robi Polikar for allowing me the opportunity to work with his team in the Signal Processing & Pattern Recognition Laboratory. He has been a constant source of encouragement throughout my graduate education. I would like to thank my committee members, Dr. Shreekanth Mandayam and Dr. Nancy Tinkham, for their patience and guidance throughout the thesis defense process. Furthermore, I would like to thank the graduate students in the SPPRL and virtual reality labs. Special thanks go out to Ryan Ewell for preparing the NOAA weather dataset. Finally, I thank my family for their continued support.

Abstract

Gregory Charles Ditzler
INCREMENTAL LEARNING OF CONCEPT DRIFT FROM IMBALANCED DATA
2009–2011
Robi Polikar, Ph.D.
Masters of Science

Learning data sampled from a nonstationary distribution has been shown to be a very challenging problem in machine learning, because the joint probability distribution between the data and classes evolve over time. Thus learners must adapt their knowledge base, including their structure or parameters, to remain as strong predictors. This phenomenon of learning from an evolving data source is akin to learning how to play a game while the rules of the game are changed, and it is traditionally referred to as learning concept drift. Climate data, financial data, epidemiological data, spam detection are examples of applications that give rise to concept drift problems. An additional challenge arises when the classes to be learned are not represented (approximately) equally in the training data, as most machine learning algorithms work well only when the class distributions are balanced. However, rare categories are commonly faced real-world applications, which leads to skewed or imbalanced datasets. Fraud detection, rare disease diagnosis, anomaly detection are examples of applications that feature imbalanced datasets, where data from category are severely underrepresented. Concept drift and class imbalance are traditionally addressed separately in machine learning, yet data streams can experience both phenomena. This work introduces Learn^{++}.NIE (nonstationary & imbalanced environments) and Learn^{++}.CDS (concept drift with SMOTE) as two new members of the Learn^{++} family of incremental learning algorithms that explicitly and simultaneously address the aforementioned phenomena. The former addresses concept drift and class imbalance through modified bagging-based sampling and replacing a class independent error weighting mechanism – which normally favors majority class – with a set of measures that emphasize good predictive accuracy on all classes. The latter integrates Learn^{++}.NSE, an algorithm for concept drift, with the synthetic sampling method known as SMOTE, to cope with class imbalance. This research

also includes a thorough evaluation of Learn^{++}.CDS and Learn^{++}.NIE on several real and synthetic datasets and on several figures of merit, showing that both algorithms able to learn in some of the most difficult learning environments.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Pattern recognition and machine learning is the process of taking in raw data and making a decision based on the category or class of the pattern [1]. Once we are provided a new (unlabelled) instance we wish to give a membership to the new data without the aid of a human. The ultimate goal in computational intelligence is to develop an algorithm that has the ability to mimic the brain-like intelligence, where in the context of machine learning is referred to as an adaptive and intelligent system (AIS).

The process of identifying patterns is crucial in the human thought process and has allowed for the evolution of our neural and cognitive systems. Our cognitive systems allow for the learning of new knowledge when it is presented to us. *Neural plasticity* is the learning and development of our neural systems in response to a new experience [2]. Plasticity allows for the acquisition of new information, so it seems intuitive that some form of plasticity be integrated into the learning machine if it is to react similar to human-like intelligence. *Neural stability* is the retaining of information that has been previously learned and is another valuable quality for a learning machine. Therefore, if we want the learning machine, or AIS, to have some brain-like intelligence properties, it is imperative to retain existing knowledge when relevant and learn new knowledge that becomes available over time.

In this thesis, the focus is on learning large amounts of data over time and the environments from that may be dynamic in nature with unbalanced data. Dynamic environments, concept drift and learning in nonstationary environments are used interchangeably throughout this thesis. Concept drift occurs when the statistical properties that govern the joint probability distribution change as a function of time. Unbalanced data, or imbalanced data, refers to an unequal representation of classes in a pattern recognition problem. There are typically two types on class in an imbalanced pattern recognition problem, majority (negative) and minority (positive). The majority (negative) class corresponds to a class or set of classes that is the large majority of the instances in a dataset. The minority (positive) class is under-represented in the training data. The

minority class is typically of greater importance than the majority class to the pattern recognition problem.

Traditional machine learning theory presented in the classical texts generally assumes the classification algorithm is learning from a fixed yet unknown distribution [1, 3, 4, 5]. Assuming such a distribution may not be a valid assumption when the data come from dynamic environments. While the classical texts provide a sound theory of machine learning, they do not address the problems faced in many real world applications such as semi-supervised learning [6], imbalanced data [7], biased training/testing distributions [8] and incremental learning [9, 10, 11]. In this thesis, we focus on incremental learning, concept drift and class imbalance.

1.1 Problem Statement

The problems addressed in this thesis cover three main topics: incremental learning, concept drift and class imbalance. Each one issue is studied individually as well as combining them into a single composite problem. Each problem requires its own specific methods for evaluation. We use the following setting: data are presented sequentially in batches with a set of labels for training. After training is complete, a testing data set is then presented to the algorithm where the class labels are not available until after the algorithm has been updated. Most importantly, we assume that the joint probability distribution of the data and labels at time t, $p_t(\mathbf{x}, \omega)$, is not the same as the joint probability distribution at time $t+1$, $p_{t+1}(\mathbf{x}, \omega)$.

1.1.1 Concept Drift

Concept drift occurs when the statistical properties that govern the distribution of the data change over time. This change could be caused by a hidden context, which may never be fully understood. Even though the reason why the distribution of the data are changing may not be fully known, an algorithm must take action in order to track and learn the drifting concepts. Traditional machine learning algorithms are not designed for learning in the presence of concept drift [1, 3, 4]. Thus, learning in nonstationary environments (i.e., in the presence of concept drift) requires that new methods be developed. Learning concept drift increases the difficulty of the classification task as the stability-plasticity dilemma needs to

be more directly addressed. This work extends the effort set forth by Polikar, Muhlbaier, and Elwell [12, 13], which is review in Chapter 3. Concept drift is enjoying increased attention as the applications that generate nonstationary data become more prevelant in the real-world.

1.1.1.1 Examples of Concept Drift

Applications that generate drifting data are increasingly becoming more prominent in real-world learning scenarios. This section presents some applications where concept drift is found.

Consumer Ad Relevance: Consider an application that tracks which ads are most relevant to a particular user's interest. Customer interests are known to change – or *drift* – over time, in part due to their changing needs [14, 15]. In such cases, certain ads the customer used to express interest may no longer be relevant. Thus, the algorithm designed to determine the relevant ads must be able to monitor the customer's browsing habits and determine when there is change in the customer's interest.

Spam E-mail Detection: E-mail spam, commonly referred to as junk e-mail, are identical e-mails sent in bulk to a large list of recipients. This form of e-mail is unsolicited and the number of spam e-mails has been steadily increasing since the inception of e-mail. In the pattern recognition realm, our goal is to identify e-mails that resemble spam. Spam e-mails, however are not all identical and change over time. For example, a user may receive spam e-mails trying to get them to buy pharmaceutical drugs; however, as time goes on the focus drifts to weight loss drugs and casino ads. This change in features of spam (pharmaceutical drugs $\rightarrow$ weight loss drugs $\bigoplus$ casino ads) must be identified.

Other applications that call for an effective change or drift detection algorithm can be expanded. Changes in electricity demands, financial data analysis, and climate data analysis are all examples where change or drift detection is needed such that the learner may take an appropriate action.

1.1.2 Class Imbalance in Data

Class imbalance occurs when a (rare) class is severely under-represented in the training and testing datasets. Often, the rare (minority) class is more important to the pattern

recognition task than the majority class. Learning from imbalanced datasets is very difficult for several reasons: (a) the primary class is under represented and may not provide information into the full feature space of the class, (b) traditional machine learning algorithms tend to bias themselves toward the majority class due to the optimization of an objective function, and (c) the misclassification of a minority class instance generally incurs more penalty than the majority class. Due to the issues associated with learning an imbalanced class problem, we need to evaluate different measures to analyze the algorithms performance. Overall accuracy is generally used to measure the performance of an algorithm for concept drift. In fact, error is typically used to track and detect when drift is present in a stream of data. However, overall accuracy is no longer an accurate assessment of how well the algorithm is performing. For example, consider a dataset that contain 20 minority (positive) and 980 majority (negative) instances. If a classifier obtains 2% error on the dataset, it tells us very little about how well the classifier identified 20 minority instances. Error will be biased towards the majority class and error will not be adequate enough to access an algorithms ability to recall the minority *as well as* majority class. Thus, we need to use statistical measures other than accuracy/error when we evaluate algorithms on imbalanced data.

1.1.2.1 Examples of Class Imbalance

Datasets that contain imbalanced data are found in many real-world applications. This section presents some applications where imbalanced data is found.

Credit Card Fraud Detection: Consider a computer software program that is required to label whether or not a customer's transaction is fraudulent or legitimate (as described in [16]). The fraudulent class in this application is severely under-represented. The majority class is likely be learned very well since the vast majority of transactions made every day by customers are legitimate. However, our primary task is to be able to identify the fraudulent transactions that are underrepresented in the training/testing data (imbalance is good for the customer / bad for the computer scientist).

Breast Cancer Classification from Mammograms: Consider the situation where a university is awarded a grant to improve the classification systems for breast cancer for younger patients as the detection is slightly more difficult in the early stages (say 30-39

years old). The primary objective of this project is to classify individuals as cancerous (positive) or healthy (negative). The data for the project are provided by a hospital in the region and comes from the last 10,000 women between the ages of 30-39 who received a mammogram. According to the National Cancer Institute, only 0.43% of patients between 30-39 test positive for cancer (about 1 in 233) [17]. Thus, approximately 43 out of the initial 10,000 women can be used to build a model of the cancerous population. We typically find that classifiers bias themselves to the majority class (in this case the healthy patients) and perform poorly on the *positive* (minority) class. Consider a classifier that obtains 20% accuracy on the positive class and 100% accuracy on the negative class. This means 35 patients who have breast cancer will be labelled as healthy and 9957 healthy patients are correctly identified, while the overall accuracy is $(8 + 9957)/100000 = 0.9965$. The high performance is quite deceiving when the the accuracy on the minority class becomes the focus of the pattern recognition.

1.1.3 Incremental Learning of Data

The learning machine solution presented in this thesis are geared towards learning from streaming data over time, where concepts that define the problem may change over time. However, it is clearly infeasible from a computational point of view to retain all of the data due memory limitations. Moreover, we may no longer have access to the previous (old) data thus rendering any algorithm that needs access to it useless in such an application. Incremental learning requires an algorithm that is capable of learning from new data (that may introduce new concept classes), while retaining the previously acquired knowledge without requiring access to old datasets. One simple solution would be to train a new model every time data are presented and discard the old model, but this solution is rather naïve in nature and incurs some detrimental results as discussed in the Background section of this thesis. Multiple classifier systems (MCS) or ensemble systems can be utilized to expand our knowledge base (i.e., add to the ensemble) and retain old knowledge from previous time (i.e., knowledge saved in the ensemble). This raises *stability-plasticity* dilemma hinted earlier [18]. The classifiers within the ensemble are models for an environment at a different point in time thus leading to the use of old models based on their relevance in recent time. Ensemble systems have been shown to provide a good balance between stability and plasticity, which

is one of the primary influences for using them in this work. The vast majority of the work done in incremental learning makes the assumption that the batches of data presented to an algorithm over time are coming from a static (i.e., fixed yet unknown) distribution. In other words the distribution is static.

1.2 Scope of Thesis

Combining concept drift and class imbalance into one learning problem has been under-explored in machine learning literature. Traditionally, either concept drift *or* class imbalance is addressed, rarely both. Furthermore, many concept drift algorithms perform poorly when imbalanced learning scenarios are present. Here we consider performing poorly as an indication of how well the algorithm does on *all* the classes.

It is important that research be carried out in this area for several reasons: many learning scenarios where concept drift is present may also contain class imbalance, and there are no truly incremental learning algorithms for this problem to the best of the author's knowledge.

1.2.1 Concept Drift + Class Imbalance Applications

The classification of fraudulent credit card charges as presented in Section 1.1.2.1 may also be converted into a problem that contains both concept drift and class imbalance. Consider detection of fraudulent credit card charges as a function of time. In this example let's assume we are provided labelled training data for updating our algorithm and field data for classifying charges every year. The raw features in this example are derived from the amount of the transaction, distance from the customers' residence, information on past purchases, time of the year and income of the customer.

Class imbalance is present in the training set simply because the fraudulent transactions occur far less often than a legitimate transaction. Now the features that describe the transactions may change over time and the change could be abrupt or gradual. For example, consider an individual who has a simple life-style and only buys essential goods. There may be a slight upward trend in the transaction amount because of inflation in the economy and varying prices in consumer products. Abrupt changes may occur when there is a crash in the stock market, change of occupation yield a change in the customers' income, or changing patterns in the customer's interests.

1.3 Summary of Contributions

This work presents an analysis of new algorithms that are capable of handling incremental learning, concept drift and class imbalance at the same time. Two new algorithms, Learn^{++}.NIE and Learn^{++}.CDS, have been developed specifically for this learning scenario. Three variations of Learn^{++}.NIE are presented as well. We then present a transductive learning ensemble for concept drift. Finally, a drift detection algorithm is presented. The core portions of this thesis can be summarized as:

1. Analysis of a new set of incremental learning algorithms that learn from drifting concepts and imbalanced classes, simultaneously. Two primary algorithms, Learn^{++}.NIE and Learn^{++}.CDS, are presented along with three variations of Learn^{++}.NIE.
2. Empirical analysis of various classifier weighting measures within the Learn^{++}.NIE algorithm.
3. Empirical analysis of Learn^{++}.CDS
4. A transductive learning algorithm for incremental learning in nonstationary environments. This ensemble algorithm attempts to estimate the weights of the Bayes optimal set of discriminant functions.
5. A drift detection algorithm for incremental learning scenarios using a difference in Hellinger divergence between two distributions.

Portions of the work presented in this thesis has appeared at the IEEE International Conference on Pattern Recognition (ICPR2010), the IEEE World Congress on Computational Intelligence (WCCI2010), and the IEEE Symposium on Computational Intelligence in Dynamic and Uncertain Environments (CIDUE2011).

1.4 Organization of this thesis

Chapter 2 provides a background into the problems encountered in incremental learning, concept drift and class imbalance. Chapter 3 follows with a comprehensive literature survey for algorithms that can handle incremental learning, concept drift, class imbalance and a fusion of the fields. Chapter 4 describes several new approaches to learning in such environments as well as a tranductive algorithm for concept drift and a drift detection

method utilizing a divergence measure. Chapter 5 presents a set of synthetic and real-world experiments to evaluate the strengths and weaknesses of the algorithms presented in Chapter 4. Finally, a summary of conclusions and suggestions for future work are laid out in Chapter 6.

Chapter 2

Background

This chapter introduces the fundamental issues associated with incremental learning from concept drift and classes that are severely under-represented. Each issue is evaluated individually before forming a fusion of the problems, which is the core portion of this thesis. Running examples are presented and used throughout this thesis to convey key problems in machine learning.

2.1 Incremental Learning

Incremental learning is a useful and practical technique of learning new data over time. Learning incrementally is not only useful because it allows us to refine our models/classifiers over time, but one can make the claim that we rarely get access to the entire feature space in one dataset. Rather we are presented with portions of the overall feature space in each batch of new data. Thus, an incremental learning algorithm allows learning of the entire feature space although the entire space has not been observed in any one dataset. Because of incremental learning, we may update, or add to, our knowledge base as more data are provided. Multiple classifier systems (MCS) provide a rational solution [11, 19]. A more Neolithic approach to learning from new data is to simply generate a new model and throw away the old knowledge. This is known as *catastrophic forgetting* and typically leads to undesirable results [20].

Consider the scenario where we are classifying weather data and the data are provided to the learning algorithm at evenly spaced intervals. At the end of every season, we must use this new information to update the mode. In the scenario described, not only is incremental learning vital, but also learning from concept drift as the feature describing the data drifts over of time. Concept drift in weather prediction is discussed in more detail in Section 2.2.

The definition or criteria for an incremental learning algorithm may vary from author to author and for this reason we clearly specify our criteria for an incremental learning algorithm [11, 21]. As an example, some authors do not consider holding onto minority

Table 2.1 : Incremental learning algorithm requirements

Requirement 1	*One pass learning*: The learning algorithm shall not require access to previous databases.
Requirement 2	*Learn new knowledge*: Build upon the current model when a new dataset is made available to the algorithm.
Requirement 3	*Preserve previous knowledge*: Knowledge shall not be discarded/forgotten from past learnt databases.
Requirement 4	*Limited processing*: Each batch should be processed in a small time regardless of number of the examples processed in the past.

class data as a violation of the one pass requirement in definition of incremental learning. Minority class data refer to the class that are under-represented in an imbalanced dataset. The concept of imbalanced data is discussed in more detail in Section 2.3. The authors' justification in [21] is that if the imbalance is large enough, then the physical memory usage required to store the data is extremely small and can therefore be accumulated over time. However, holding onto data, regardless of the class, for later use is in violation of the incremental learning definition in [11]. Therefore, we define an incremental learning algorithm as being able to learn additional information from new data when it becomes available, without requiring access to previous data, while preserving previously acquired knowledge [11]. Their criteria for incremental learning can be extend upon by including Kuncheva's suggestions for an algorithm that learns from concept drift, to include limited processing time and any-time learning [22]. Table 2.1 summarizes the typical incremental learning requirements for learning in the presence of concept drift [22, 11, 23].

The definition of incremental learning requires that an algorithm is able to learn new knowledge and retain old knowledge. This brings rise to the *stability-plasticity* dilemma, which are generally two conflicting objectives [18]. Carpenter and Grossberg proposed a solution in Adaptive Resonance Theory (ART) [24]. ART was originally developed as an unsupervised neural network for incremental learning of binary input patterns. The fundamental theory of ART has been expanded to develop unsupervised neural networks for continuous inputs (ART-2), implementing fuzzy logic into ART's pattern recognition (Fuzzy ART), implementing a supervised ART model for prediction (ARTMAP), and implementing fuzzy logic in ARTMAP (fuzzy ARTMAP) [25, 26, 10, 27].

The algorithms proposed and presented in this thesis are all incremental learning

algorithms, where new training data are presented in batches over time. We refer to each presentation of new data as *time stamps*. Incremental learning algorithms are different than on-line learning algorithms as on-line learning requires that a classifier is updated with each new instance presented. Examples of on-line learners are naïve Bayes [1, 5], Dynamic Weight Majority (DWM) [28], On-line Nonstationary Boosting (ONSBoost) [29], and Hoeffding decision trees [30, 31]. On-line learning algorithms typically maintain a good level of plasticity at the cost of stability while the opposite is true for an incremental learner.

2.2 Concept Drift

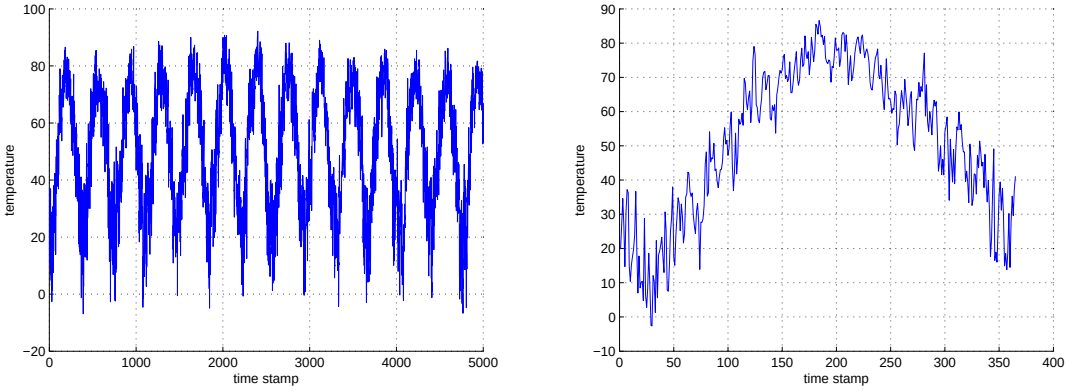
Traditional learning algorithms, such as Adaboost [32, 33] and Support Vector Machines (SVM) [34], assume the data defining the concepts are being sampled from a fixed yet unknown distribution. If the test data distribution varies as a function of time (i.e., training set is sampled from a different distribution than the testing set), the theoretical error bound of AdaBoost will not hold. Thus, we present a problem referred to as *learning under concept drift* or in *nonstationary environments*. Concept drift is the phenomenon of data changing over time. This drift/change can be caused by a number of different factors governing the learning problem, however, models that address this change must be adaptive in order to remain relevant predictors. Concept drift is a difficult problem in machine learning where the learning scenario that the concepts (classes) of interest may depend on some hidden context [35, 36]. The drifting concepts in the data may be slow, fast, abrupt, gradual, cyclical or a combination [37, 22] and studies performed in information retrieval have indicated that target concepts may change at several different speeds over time [38]. The nature of why drift is present may be understood, yet the algorithm being applied for a problem with concept drift must track the drift to be an accurate predictor on new concepts or probability distributions. Learning in nonstationary environments has been receiving increasing attention over the past several years as more Master's and Ph.D. theses focus on concept drift [13, 23, 30, 15, 39, 40, 41, 42].

Any algorithm that does not make the necessary adjustments to changes in data distribution will fail to provide satisfactory generalization on future data, if such data does not come from the distribution on which the algorithm was originally trained. Recall

the application that tracks a user's web browsing habits to determine which ads are most relevant for that user's interest. Tracking user interests have been studied in [43, 44, 45, 14]. Yandex.Direct is an advertising network that is designed specifically to take into account user interests when displaying ads [46]. Thus, an algorithm designed to determine the relevant ads must be able to monitor the customer's browsing habits and determine when there is change in the customer's interest. User interests may also be categorized as short-term or long-term as described by Žliobaitė [15]. Žliobaitė gives an example where an individual working on a class project will change their browsing habits for a short period of time until the project is finished. Long term interests may include following a sport like football. Applications that call for an effective change or drift algorithm can be expanded: analysis of electricity demands, financial data analysis, and climate data analysis are all examples of nonstationary applications where change or drift detection is needed so that the learner can take an appropriate action.

Changes in a hidden context may not be the only cause of a changing concept, but may also cause a change in the underlying distribution of the data. Consider developing a pattern recognition algorithm that predicts whether or not it rained on any given day. A team of informed meteorologist interns determine there are several discriminating features to aid in predicting if it rained on the day the data was collected. The interns job for the summer is to gather data and pass it along to a team of researchers involved in machine learning. The data are used by those involved in machine learning group to generate a classifier whose sole purpose is to predict rainy days. The concept of a rainy day does not change. Either it rained or it didn't rain on any given day when the data were collected. Now, new data are collected throughout the fall/winter months and classified with the rainy day model developed with the summer season data. It's very likely, depending on the physical environment, that the features of the new data could be radically different than those used for training due to changes in temperature, humidity, visibility, barometric pressure, etc*. The purpose of the classifier is still the same, predicting rainy days, yet new data from the fall/winter are presented to the classifier that could not have possibly been learned. Thus, the underlying distribution that governs the data has drifted as a function of time. This may cause the classifier's error to be unacceptable with the new distribution

*Some climates may not observe a large change in temperature over a yearly cycle.



(a) Drift over 5000 days

(b) Drift over 365 days

Fig. 2.1 : Drift in average daily temperature. Data is acquired from the NOAA [47].

that governs the data. Fig. 2.1 presents the average daily temperature in the National Oceanic and Atmospheric Administration (NOAA) weather database over 5,000 days and 365 days. In this example, the rainy day model must be adaptively updated to accurately predict on new data throughout the year.

The effects of concept drift can be analyzed through Bayes theorem in Eq. (2.1) & (2.3) where $p(\mathbf{x})$ is the evidence of a random variable $\mathbf{x}$, $p(\mathbf{x}|\omega_j)$ is the likelihood, $P(\omega_j)$ is the prior probability of ω_j and $P(\omega_j|\mathbf{x})$ is the posterior probability. This relationship is derived from the joint probability distribution of $\mathbf{x}$ and ω_j, namely: $p(\mathbf{x}, \omega_j) = p(\mathbf{x})P(\omega_j|\mathbf{x}) = p(\mathbf{x}|\omega_j)P(\omega_j)$. The subscript, t, in Eq. (2.1, 2.2, 2.3) denotes a time stamp when the Bayes posterior probability is computed. Simply, using any one of the terms in Bayes theorem is not enough to be certain that drift is present in the data.

$$P_t(\omega_j|\mathbf{x}) = \frac{\overbrace{p_t(\mathbf{x}|\omega_j)}^{likelihood}\overbrace{P_t(\omega_j)}^{prior}}{\underbrace{p_t(\mathbf{x})}_{evidence}} \tag{2.1}$$

$$p_t(\mathbf{x}) = \sum_{i=1}^{c} p_t(\mathbf{x}|\omega_i)P_t(\omega_i) \tag{2.2}$$

$$P_t(\omega_j|\mathbf{x}) = \frac{p_t(\mathbf{x}|\omega_j)P_t(\omega_j)}{\sum_{i=1}^{c} p_t(\mathbf{x}|\omega_i)P_t(\omega_i)} \tag{2.3}$$

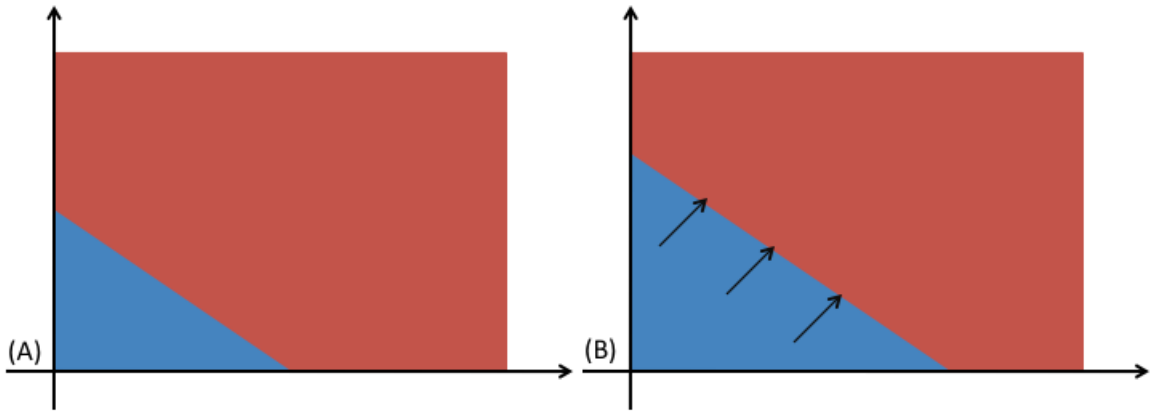


Fig. 2.2 : Simple drift example

The evidence in Bayes theorem, Eq. (2.2), is the probability that the random variable $\mathbf{x}$ (measurement) will even occur regardless of the class membership for $\mathbf{x}$. The evidence term may be written that as a summation over all classes of the product of the likelihood and prior probability as stated in Eq. (2.2). While it may be possible to use the evidence as a method to detect drift for a particular problem, it certainly is not enough for the more general drift detection scenario. For example, consider a uniform distribution of data over two features, $\mathbf{x} = [x_1, x_2]^T$, and there is a linear separation between the two concepts as shown in Figure 2.2(a). At a later point in time, the plane separating the two concepts shifts up to the location in Figure 2.2(b). The probability distribution of the data in this example never changed when the plane shifted its location thus, $p(\mathbf{x})$ never changed even though the concept drift is present in this simple example.

Another possibility is that not just a single component of Bayes theorem is changing; there may be drift where two of the components in Bayes theorem are drifting with time [48]. This makes it very difficult to know what is changing unless there access to a massive amount of the data at each time stamp, which is simply infeasible to process and estimate the components of Bayes theorem, particularly for high dimensional data [1].

Bayes theorem may be used to describe three different concept drift scenarios. Concept drift may appear in one of three ways:

1. Class priors, $P(\omega_i)$, change over time.
2. The likelihoods, $p(X|\omega_i)$, may change.

3. The posterior probability, $p(\omega_i|X)$, may change.

There are several terms for categories of concept drift, namely *real* and *virtual* drift. Real drift is a change in the posterior probability distribution given by $p(\omega_i|X)$. Virtual drift is change in the distributions of one or several classes given by $p(X|\omega_i)$. Dual change occurs when both $P(\omega_i)$ and $p(X|\omega_i)$ change as a function of time. Regardless, of real, virtual or dual change an algorithm must effectively process data to take an appropriate action when change is signaled or new data is processed.

2.2.1 Methods of Handling Concept Drift

A naïve approach to learning from concept drift is to simply discard a classifier when new data are presented and generate a new classifier. Therefore, data are classified only with the newest classifier. This approach does not save any information about old environments, which can be useful for a future classification task. Such an approach leads to *catastrophic forgetting* [20].

Learning from drifting environments is usually associated with a stream of incoming data, either one instance or one batch at a time. There are two types of approaches for drift algorithms in such streaming data: in *passive drift algorithms*, the learner assumes – every time new data become available – that some drift may have occurred, and updates the classifier according to the new data distribution, regardless whether drift actually occurred. In *active drift algorithms*, the algorithm continuously and explicitly monitors the data to detect if and when drift occurs. If – and only if – the drift is detected, the algorithm takes an appropriate action, such as updating the classifier with the most recent data or simply creating a new classifier to learn the current data.

2.2.2 Definitions of Concept Drift

In this section, the definitions of drift types are presented. The types of drift are distinct from one another and are used during the design of the synthetic experiments. We use the notion of a source S_1 generating data from a fixed distribution and source S_2 generating data from a fixed distribution that is different than S_1. Let S_1 be the initial source for generating data. This is the same notation used by Žliobaitė [15].

Sudden Drift or Concept Change: Concept change occurs at a point in time when source changes from S_1 to S_2. Fig. 2.2 is an example of abrupt concept change that contains an abrupt change when the hyperplane separating the two classes suddenly changes its location. The SEA experiment, described in Section 5.5.4, is a prime example of sudden drift [49].

Incremental Drift: Incremental drift contains multiple sources however the difference between the sources is very small. Thus, the drift is only realized when observed globally. The rotating checkerboard problem presented in [50], and further described in Section 5.5.1, uses incremental drift since each time stamp is a different source.

Gradual Drift: Gradual drift occurs when data are being drawn from two or more similar sources within one time stamp. Generally, as time passes the probability of sampling from S_1 decreases as the probability of sampling from S_2 increases. Consider sampling from a data stream as presented in [23] where S_1 & S_2 are two different sources for two different data streams, t is the time, t_0 is the point of change in the distribution, $c(t)$ is the stream generate by sampling streams S_1 & S_2, and $\mathcal{W}$ is the width of the change. Then Eq. (2.4) can be applied as the probability of sampling from source S_1 at time t and Eq. (2.5) as the probability of sampling from source S_2 at time t. The probability of sampling from either source is shown in Fig. 2.3.

$$P\left(c(t) = S_1(t)\right) = \frac{e^{-4(t-t_0)/\mathcal{W}}}{1 + e^{-4(t-t_0)/\mathcal{W}}} \tag{2.4}$$

$$P\left(c(t) = S_2(t)\right) = \frac{1}{1 + e^{-4(t-t_0)/\mathcal{W}}} \tag{2.5}$$

Reoccurring Concepts: Reoccurring concepts appear when several different sources are used to generate data over time (similar to incremental and gradual drift). However, unlike incremental and gradual drift, sources are used again to generate data at a future point in time. The checkerboard problem in [50] uses incremental drift with reoccurring concepts. A similar sampling scheme as described above may be applied to show a mixture of gradual drift with reoccurring concepts (see Fig. 2.4).

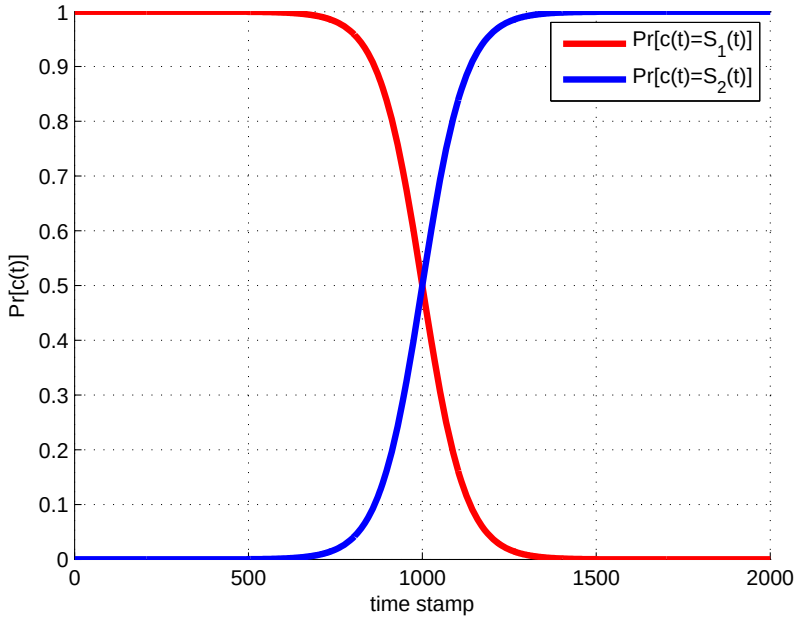


Fig. 2.3 : Probability of sampling from different sources under a gradual concept drift scenario as computed using Eq. (2.4) and Eq. (2.5) where $t_0 = 500$ and $\mathcal{W} = 250$

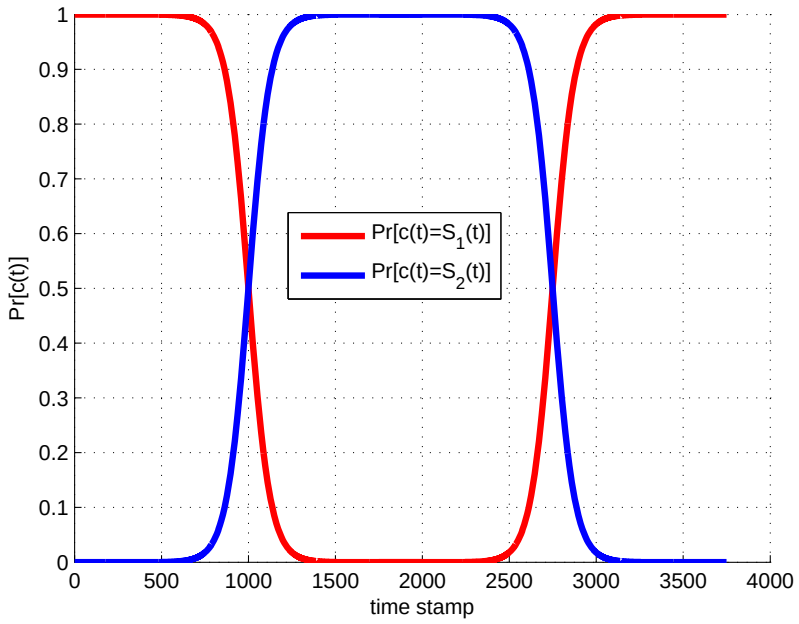


Fig. 2.4 : Probability of sampling from different sources under a gradual concept drift scenario with reoccurring environments

2.2.3 Ensembles for Concept Drift

Ensemble classifier based techniques have been widely studied since their inception [51, 52, 19]. The principle behind the ensemble decision is that the individual predictions combined appropriately, should have better overall accuracy, on average, than any individual ensemble member [53]. For the moment, consider the data are being drawn from a fixed yet unknown distribution. The concept of a varying distribution will be addressed after the following discussion on ensembles.

Consider a situation where T classifiers are generated on a binary classification problem where the classifiers have identical performance probabilities, and classifier outputs are independent of one another. Let the combination rule be a simple majority vote. If this is true then Eq. (2.6) is the error probability of the majority vote where p is the probability of error.

$$P(H(\mathbf{x}) \neq y) = \sum_{k>(T/2)}^{T} \begin{pmatrix} T \\ k \end{pmatrix} p^k (1-p)^{T-k} \tag{2.6}$$

Eq. (2.6) shows that the classifiers only need to be slightly better than a random guess for a binary classification problem ($p < 0.5$). If $p > 0.5$ then the ensemble error $P \to 1$ as $T \to \infty$ and if $p < 0.5$ then the ensemble error $P \to 0$ as $T \to \infty$. Therefore, classifiers that are *weak* in performance may be used and combined to create a very strong decision. The notion of using a set weak classifiers to form a strong hypothesis is the corner stone of Adaboost and boosting based approaches [33, 54]. Ensembles using boosting generate multiple classifiers on strategically chosen datasets, rather than random sampling as done in Bagging [55].

There are several reasons why an ensemble would be chosen over a single classifier solution, and the reasons have theoretical/practical motivations. First, which classifier to use in the absence of prior knowledge about a problem and is there a set of classifiers that will perform better than others on a classification scenario? The *No Free Lunch theorem* states that if no prior information is available then no classifier is universally better than any other classifier [56]. This includes random guessing. Second, ensembles may be extremely useful depending on the attributes or properties of the data. For example, ensembles may be used to simplify a problem by breaking a difficult problem into simpler

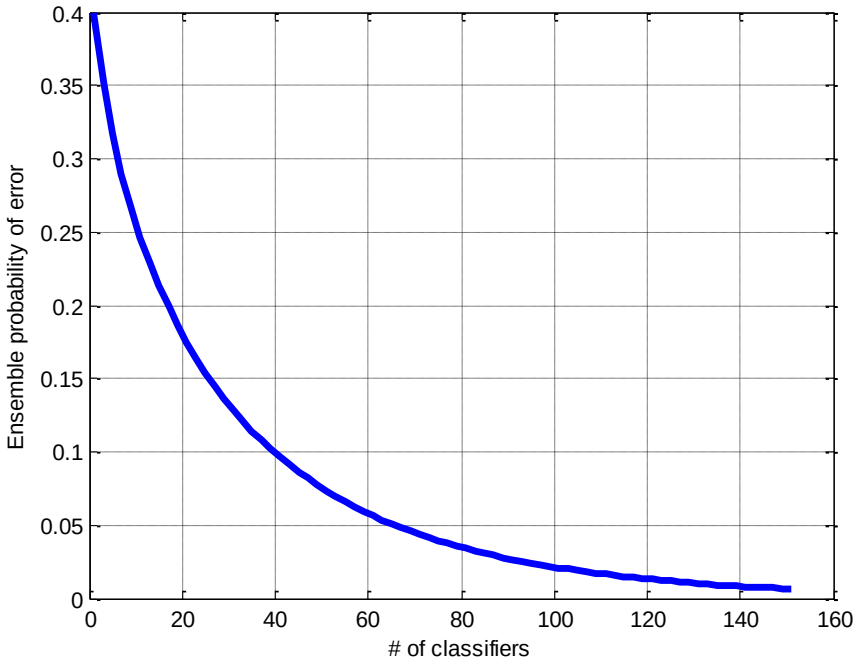


Fig. 2.5 : Theoretical error vs. the number of classifier for an ensemble combined using SMV. The error of each individual classifier is $p = 0.4$ and is computed using Eq. (2.6).

problems. Consider generating a classifier on an extremely high dimensional dataset. A single classifier's complexity may scale with the dimensionality of the data thus making the generation of a reliable single classifier on the data infeasible. Instead of a single classifier, generate multiple classifiers on different subsets of features thus reducing the complexity of the classifier trained on the subset. Third, single classifiers may not work well with data that are too little or too large in size. To work around this problem, ensembles can generate classifiers on multiple bootstrap datasets. A small dataset benefits from bootstrapping datasets because each classifier is generated on a different dataset sampled from the same distribution. If the cardinality of the data are too large then the entire dataset can be divided into smaller datasets and a classifier is generated in the smaller dataset. All classifiers must be combined to form the ensemble hypothesis. Fourth, generating a single strong classifier may be infeasible due to computational costs. Generating a classifier that performs slightly better than random guess is easier. Ensemble-based approaches have been widely used in earlier efforts of our group (SPPRL): including posterior estimation [57], early diagnosis of

Alzheimer's disease [58], missing features[59], data fusion [60], and learning in nonstationary environments [12].

The aforementioned reasons provide justification for using ensembles when the data are drawn from a fixed yet unknown distribution. With certain modifications, the ensemble system can also be used in concept drift problems. In each ensemble based approach, a method for classifier combination in needed along with method to generate classifiers in the ensemble. First, call the **BaseClassifier** to generate a new classifier when training data are presented. Instead of generating a weak classifier, as done in Adaboost, concept drift algorithms should have a fairly strong classifier that serves as a model of the distribution of data on which it was trained. Algorithms such as Learn^{++}.NSE use this approach because unlike Adaboost, Learn^{++}.NSE is designed to learn from sequential batch data and the classifier generated on each batch must have the ability to form a strong hypothesis on the data it was trained with.

Next, a combination rule should be selected to combine the outputs of the classifiers. Kuncheva presents an excellent analysis of ensembles and combination rules in her text, *Combining Pattern Classifiers* [52]. The choice of combination rule depends on the types of labels a classifier may return. Typically, classifiers outputs can be reduced down two types: a *hard* or *soft* label classifier. A classifier that returns a hard label provides support only to the class it has selected for a given instance. The CART and C4.5 decision trees are examples of classifiers that provide hard labels. A classifier that returns a soft label provides some support for each class the classifier was trained on. This level of support is generally an estimate of a posterior probability. The multi-layer perceptron neural network (MLPNN) and logistic regression are examples of classifier that provide a soft label.

Classifiers that return only hard labels limit the types of combination rules that can be used. For example, naïve Bayes, sum, median and majority rules are quite popular combination rules. However, the naïve Bayes rule cannot be used with classifiers that return hard labels as it needs a level of support for each class. The naïve Bayes combination rule is given by Eq. (2.7) where $\mu_j(\mathbf{x})$ is the ensemble support given to class j for instance $\mathbf{x}$, $P(\omega_j)$ is the prior probability of ω_j, and $p(d_k(\mathbf{x})|\omega_j)$ is the likelihood of the classifier decision $d_k(\mathbf{x})$ given class ω_j.

$$\mu_j(\mathbf{x}) \propto P(\omega_j) \prod_{k=1}^{T} p(d_k(\mathbf{x})|\omega_j) \tag{2.7}$$

In concept drift applications, a new classifier is typically generated at each time stamp. So, which voting scheme or combination rule is appropriate for concept drift problems? Since the distribution of data evolve over time, some classifiers will have lower error than others on the most recent environment. Therefore, one cannot assume that equal weights (i.e., sum or mean rule) would be the best selection if some are known to perform better than others on recent environments. Gao makes the claim that one cannot always assume that the distributions of most recent training data and the incoming testing data are alike, which is a true statement [61, 62]. However, if the data are evolving in a systematic pattern (not stochastic) then simply assuming there is no information about *how a classifier will perform on a future distribution* may not be entirely true. Gao reaches this conclusion by minimizing Eq. (2.8) subject to Eq. (2.9, 2.10).

$$\sum_{k=1}^{T} w_k^2 \tag{2.8}$$

$$1 - \sum_{k=1}^{T} w_k = 0 \tag{2.9}$$

$$0 \leq w_k \leq 1 \tag{2.10}$$

The problem described is a constrained convex optimization that can be easily carried out by forming a Lagrangian function, $L(\mathbf{w}, \alpha, \beta, \lambda)$. By computing $\frac{\partial L}{\partial w_k}$ is can be shown that $w_k = 1/T$ (i.e. uniform weighting).

However, using uniform weighting does not make practical sense with an ever-growing ensemble [13]. Gao's algorithm will be used as a comparison to the proposed weighted voting scheme described later in this thesis.

This thesis focuses on *weighted majority voting* approaches. This combination rule assigns a weight to classifier to use for voting. The weight is typically proportional to the classifier's performance. Thus, if a classifier is performing well on recent environments, the weight will be large and if the classifier is performing poorly, the weight will be small.

Since the concepts are evolving systematically, the classifier's error in recent time can serve as the basis for determining a classifier's weight, as implemented in Learn^{++}.NSE [12]. However, when dealing with imbalanced data, error is no longer a reliable measure. This is discussed later in Section 4.2 where new measures are presented for determining a classifier's voting weight.

2.2.4 Drift Detection

Concept drift algorithms generally fall into one of two categories: active or passive algorithms. In passive algorithms, the learner assumes that some drift has occurred since the last training session whereas active drift algorithms will seek to explicitly determine when the drift is present in the data. If drift is detected, the algorithm takes an appropriate action, such as updating the classifier with the most recent data or simply creating a new classifier to learn the current data. Drift detection algorithms typically fall into the active category of algorithm.

There are several schools of thought for designing a drift detection algorithm. First, one has to select a method or a set of drift descriptors to seek the presence of concept drift in data. Classifier error is one of the most popular indicators of drift in data [63, 64]. Classifier error is commonly used under two primary assumptions:

- A classifier trained on $\mathcal{D}^{(t)}$ will have a relatively constant error on $\mathcal{D}^{(t+1)}$ and $\mathcal{D}^{(t+t_q)}$ where $t + t_q$ is any arbitrary time stamp in the future.
- The error of a MCS will have increasing accuracy as classifiers are trained on new data.

where $\mathcal{D}^{(t)}$ is the dataset at time stamp t. Error may not be the only indicator of drift in data. Some algorithms use raw features to determine when drift is present. Using the raw features to determine drift requires that a parametric or non-parametric method to model the distribution of the data. A parametric model makes assumptions about the distribution from which the data are drawn. Gaussian distributions are common assumptions to simplify calculations and as properties of the Gaussian distributions are well known. However, what if the data are not actually drawn from a Gaussian, but a mixture of Gaussians or a different distribution all together? Non-parametric models, such as kernel density estimators, do not make any assumption about the distribution of the data.

Regardless of the selection of drift descriptors, an action is needed if drift is detected. After change is signaled, the algorithm (or a different algorithm run in conjunction with the drift detection method) should take an appropriate action. Actions can include discarding a classifier, generating a new classifier, updating classifier weights, or doing nothing.

2.3 Class Imbalance in Machine Learning

Class imbalance, sometimes referred to as unbalanced data, occurs when a dataset does not have an (approximately) equal number of instances from each class, which may be quite severe in some applications [65]. Unbalanced data research focuses on situations where the class balance is nowhere near 50%. Rather, severe class imbalance (1%, 2%, 5%, 10%, etc). The datasets with severe class imbalance requires that algorithms figure of merit is addressed in a different manor than if there were only concept drift in data. Consider datasets with imbalances of 1%, 2%, and 5% and a classifier is generated yielding performances of 99%, 98%, and 95%, respectively. The classifier used in this simple example can be a majority class classifier, which simply classifies a new instance with the label of the class that occurs most often. The majority classifier will have a high overall accuracy, which is generally a good quality, however it will not be able to identify any of the instances that belong to the minority class. From this very generic example, its clear that error is not a best statistic to identify how well the algorithm is performing across all classes. Therefore, researchers dealing with imbalanced datasets will typically present results with statistics other than overall error to access an algorithm on an experiment. Before analyzing measures other than error, let's focus on issues associated with learning from imbalanced data.

2.3.1 Why Do Classifiers Perform Poorly on a Minority Class?

Class imbalance arises from the under representation of at least one class in a learning problem. The minority class is unfortunately the target class for many classification tasks. Recall the example of credit card fraud in Chapter 1. The number of legitimate transactions essentially dwarfs the number of fraudulent transactions. Then how can the fraudulent transactions be learned when there are so few instances and how the classifier resist biasing towards to majority class? The fraudulent class in this scenario is difficult to learn for couple reasons:

- there are so few instances that there may not be a clear representation of the minority class feature space
- many classification algorithms tend to minimize an error function, which may not favor learning a minority class.

The first point is obvious and is partially a motivation for incremental learning, however the second point is worth further discussion. Classifiers typically minimize a global error function during the training process and do not take information about the distribution of the data into account. As a result, instances from the majority class are classified with high accuracy whereas examples from the minority class tend to be misclassified. For example, algorithms such as the multi-layer perceptron neural network (MLPNN) minimize an error function, generally mean-squared error (MSE), during the training phase of the neural network [66]. So, if a classifier is likely to bias its decisions towards a majority class, what can be reduce this effect? Sampling methods, cost-sensitive learning models, and ensemble have demonstrated favourable qualities to avoid bias towards the majority class as discussed in Section 2.3.2, 2.3.3, and 2.3.4.

2.3.2 Sampling Methods

There are several popular methods of handling class imbalance. Some of the more popular approaches to learn class imbalance occur at a *data* or *algorithmic* level [67]. The data level approaches generally employ some form of sampling to generate a new dataset that is similar to using bootstrap datasets with ensembles. Simple data level approaches use random over-sampling of a minority class or under-sampling of the majority class to reduce the imbalance. The random sampling must be done with care for several reasons. A simple random over/under-sampling of a dataset to create a less imbalanced dataset comes with repercussions. A simple random under-sampling of a dataset will discard instances from the majority class. However, by throwing out instances from the majority class, risks discarding information that can be useful to the classification problem. Over-sampling on the other hand does not discard majority class data, rather it adds exact replicates of the minority class. Using this approach to re-balance a dataset runs the risk of generating a classifier that will overfit the minority class.

Synthetic sampling can be used to reduce the undesirable qualities in random

over/under-sampling of minority/majority class data. Synthetic sampling methods generally oversample a data set; however the instances added into the new dataset are *synthetic*, and not exact replicates of the minority data as performed in random over-sampling. The synthetic data are generated such that they are "similar" to other instances in the minority class population. Some synthetic sampling methods not only focus on the generation of synthetic data but also the location of the synthetic instances. For example, synthetic sampling methods may generate synthetic instances that lie near a decision boundary. The synthetic sampling methods have been shown to be less prone to over-fitting classifiers to the minority class.

Over/under or synthetic sampling does not guarantee that the minority class can be adequately learned. Rather, sampling is a fast, cost-effective method of generally increasing the performance on a minority class. Studies have shown that some classifiers on particular datasets are not affected by sampling methods. Regardless, many imbalanced datasets benefit from sampling. Popular sampling approach can be found in Section 3.3.1.

2.3.3 Cost Sensitive Learning

The sampling methods described in the previous section attempt to develop a new dataset that contains less class imbalance. Note, that sampling methods may not convert the imbalanced learning problem into one that is balanced, rather they create a less imbalanced learning problem. Cost-sensitive learning algorithms assign penalties based on a cost matrix, which represents the penalties for different possible correct/incorrect classifications. The cost matrix can be considered as a numerical representation of the penalty of classifying examples from one class to another. The objective of cost-sensitive learning is to generate a classifier that minimizes the overall cost, not error, on the training data set, which is usually the Bayes conditional risk [7, 1]. The cost-sensitive learning problems generally lend themselves better to theoretical analysis than sampling methods.

2.3.4 Ensemble Methods

Ensemble methods are not only popular for reducing error of the final hypothesis, but are also employed to learn an under-represented class. The last two sections have focused on using sampling or cost-sensitive learning to increase the performance on a minority class.

Ensembles are widely used for learning class imbalance by combining multiple classifiers, sampling, and cost-sensitive learning. Several existing ensemble techniques minimize the overall cost during training, not the error. Several different ensemble methods are discussed in more depth as the literature review of the class imbalance is presented in the next chapter.

2.4 Learning Concepts Drift from Imbalanced Data

Both class imbalance and learning with concept drift have been independently studied by the machine learning community. In fact, there are a number of workshops held every year that focus specifically in this area of computational intelligence. Recently, the WCCI[†], CIDUE[‡], ECML[§], and PKDD[¶] have had sessions and/or tutorials dedicated to the problem of concept drift. The SIGKDD journal had a dedicated issue specifically for learning problems with class imbalance. The increasing interest in these fields can be attributed to their vast areas of application, because most modern day machine learning problems experience some underlying change with time and classes are rarely distributed equally. We begin this section by presenting application areas that experience class imbalance and concept drift simultaneously.

2.4.1 Real-World Scenarios

Recall, the example of climate prediction analysis where the features are going to drift as a function of time, hence the reason. This application also faces class imbalance in many of these learning scenarios. Let's consider an algorithm that is designed to predict whether or not it rained more than 3 inches on any given day. The algorithm must be able to track the drifting concept, but also must learn from a small amount of instance. How many days will experience rain fall greater than 3 inches? The specific number of days is not as important as the ratio between the number of days it rained more that 3 inches and the number of days it did not. It is very likely that the number of days it did not rain 3 inches is much larger than the number of days it did not.

[†]http://www.wcci2010.org/
[‡]http://www.ieee-ssci.org/2011/cidue-2011
[§]http://wwwis.win.tue.nl/hacdais2010/
[¶]http://www.cs.waikato.ac.nz/ abifet/PAKDD2011/

2.4.2 Learning in Harsh Environments

Typically, machine learning algorithms are simply not equipped to handle class imbalance or concept drift and the algorithms that are designed to handle harsh environments only focus on concept drift or class imbalance, not both simultaneously. Concept drift algorithms typically use error as a weighting measure or as an indicator of drift, but error is a biased measurement (biased towards a majority class). It is unreasonable to expect an algorithm designed for concept drift to be the ideal algorithm when dealing with class imbalance. In fact, the concept drift algorithm should be expected to have a very high overall accuracy while achieving a poor recall of a minority class. The poor performance on a minority class is simply unacceptable since the minority class is usually the *target class*. From the point of view for ensembles, error is generally the statistic used to determine a classifier's voting weight, hence the reason for an ensemble biasing its decision towards majority classes. Part of the motivation for this thesis is to answer the question, can statistics other than error be used to produce a classifier voting weight and still able to achieve the following: 1) tracking drift concepts in nonstationary environments, and 2) increase the accuracy on a minority class.

The evaluation of algorithms on data with concept drift and class imbalance must be done in a way to demonstrate that the proposed approach can track drift concepts and learn a minority class. Maximizing the performance of an algorithms is not the primary objective of this work. The objective of this work is to effectively learn in the presence on concept drift while maintaining a strong performance on a minority class.

2.5 Summary

In this chapter we have described incremental learning, concept drift and class imbalance in detail. Examples have been presented for real-world learning scenarios that motivate each area of research. We have also presented reasonable logic about why classifiers designed for concept drift may perform poorly in imbalanced learning scenarios. Against this background, the next section introduces the prior work that has been performed in the computational intelligence community for incremental learning, class imbalance, concept drift, and a fusion of concept drift & class imbalance.

Chapter 3

Literature Review

The section of the thesis covers some important algorithms and methods that have been used in machine learning to handle concept drift and class imbalance. Components of the previous work presented here have inspired future movements towards developing a viable and effective solution to learning from drifting concepts and imbalanced class distributions. This section specifically focuses on the following areas in machine learning:

- classification of data with drifting concepts
- drift detection methods and updating the classification model when drift is detected
- classification of rare classes
- classification of data that has drifting concepts and class imbalance

3.1 Incremental Learning of Data

3.1.1 Fuzzy ARTMAP

The (fuzzy) ARTMAP neural network structure has been widely used as an incremental learning algorithm [27]. Fuzzy ARTMAP incorporates two fuzzy ART modules that are linked via an inter-ART module known as a *map field*. The map field is used to form predictive categories for learning class association. Fuzzy ARTMAP will generate new decision clusters in response to new input patterns that are *sufficiently different* from previously seen instances. The 'sufficiently different' patterns are controlled using a free parameter of ARTMAP known as the vigilance parameter. One of the critiques of ARTMAP is the sensitivity of the vigilance parameter especially when there may be significant noise in the training data. Using stability and match tracking, fuzzy ARTMAP, automatically constructs as many categories as are needed to learn any static training set to 100%. Thus, fuzzy ARTMAP may overfit pending parameter selection leading to poor generalization.

3.1.2 Learn^{++}

The Learn^{++} family of algorithms have found their way into may applications including data fusion [60], missing features [59], posterior estimation [57] and out-voting reduction [68]. Learn^{++} is the original implementation of the ensemble based incremental learning algorithm. Learn^{++}, whose pseudo code is shown in Fig. 3.1, is an ensemble based incremental learning algorithm for learning from a fixed yet unknown distribution [11]. Inspired by AdaBoost.M1, Learn^{++} shares many similarities with AdaBoost.M1. Data is assumed to be presented in batches, $\mathcal{D}^{(k)}$, with $m^{(k)}$ labelled instances in each set. T_k classifiers are generated on each set, $\mathcal{D}^{(k)}$. Like AdaBoost, Learn^{++} maintains a distribution of instance weights, however Learn^{++} does not update the weights in the same manner as performed with AdaBoost. Learn^{++} uses the ensemble decision, rather than the decision of the latest classifier. When a new dataset arrives, the distribution is re-initialized by evaluating the entire ensemble and initializing the distribution.

The distribution of instance weights are always updated using the composite hypothesis and not the latest hypothesis. Learn^{++} guarantees convergence on any given training dataset, by reducing the classification error with each added hypothesis. The error of Learn^{++} is bound above Eq. (3.6) where E_t is the error of the tth composite hypothesis.

$$E \leq 2^T \prod_{t=1}^{T} \sqrt{E_t \left(1 - E_t\right)} \tag{3.6}$$

Furthermore, E_t is itself bounded above by Eq. (3.7) where ε_n is the error of the individual hypothesis.

$$E_t \leq 2^t \prod_{n=1}^{t} \sqrt{\varepsilon_n \left(1 - \varepsilon_n\right)} \tag{3.7}$$

3.2 Algorithms for Concept Drift

A number of different approaches have been developed to learn from concept drift, however not everyone directly influences the work in this thesis. In this section of the thesis, several algorithms are presented, mostly ensemble based, for learning in dynamic environments. Drift detection is covered for detecting a nonstationary change in a data stream.

Input: Training Data $\mathcal{D}^{(k)} = \{\mathbf{x}_i^{(t)} \in X, y_c^{(t)} \in \Omega\}$ where $i = 1, \ldots, m^{(k)}$
Weak learning algorithm **WeakLearn**
Integer T_k, specifying the number of iterations

for $k = 1, \ldots, K$ **do**
Initialize: $w_i = D(i) = 1/m^{(k)}$ unless there is prior knowledge to select otherwise.
for $t = 1, \ldots, T_k$ **do**

1. Assure that D_t is a distribution

$$D_t = w_t / \sum_{i=1}^{m^{(k)}} w_t(i) \tag{3.1}$$

2. Randomly sample training $\mathcal{D}_{tr}^{(t)}$ and testing $\mathcal{D}_{te}^{(t)}$ subsets according to D_t.
3. Call **WeakLearn**, providing it with $\mathcal{D}_{tr}^{(t)}$
4. Get back a hypothesis $h_t : X \to Y$ and calculate the pseudo error on $\mathcal{D}^{(k)}$

$$\varepsilon_t = \sum_{i=1}^{m^{(k)}} D_t(i) [\![h_t(\mathbf{x}_i) \neq y_i]\!] \tag{3.2}$$

5. If $\varepsilon_t > 1/2$, set $t = t - 1$, discard h_t and go to step 2. Otherwise, compute normalized error $\beta_t = \varepsilon_t / (1 - \varepsilon_t)$
6. Call weighted majority, obtain the overall hypothesis $\forall l$

$$H_t(\mathbf{x}_i) = \arg\max_{y \in \Omega} \sum_{l : h_l(\mathbf{x}_i) = y} \log \frac{1}{\beta_l} \tag{3.3}$$

7. Calculate the composite pseudo error

$$E_t = \sum_{i=1}^{m^{(k)}} D_t(i) [\![H_t(\mathbf{x}_i) \neq y_i]\!] \tag{3.4}$$

8. If $E_t > 1/2$, set $t = t - 1$, discard H_t and go to step 2. Otherwise, compute normalized error $B_t = E_t / (1 - E_t)$
9. Update weights

$$w_{t+1}(i) = w_t(i) B_t^{1 - [\![H_t(\mathbf{x}_i) = y_i]\!]} \tag{3.5}$$

end for
end for

Fig. 3.1 : Learn^{++} pseudo code

3.2.1 FLORA

Widmer & Kubat were some of the first to address concept drift and hidden contexts when the FLORA family of algorithms was introduced [35]. The FLORA algorithm is based on a moving window that deletes the oldest instances. FLORA then uses accepted descriptors (ADES) that contain information about only positive examples, negative descriptors (NDES) that summarize the negative examples and potential descriptors (PDES) that are general descriptors that match both ADES and NDES. The PDES contain instances that used to be ADES or NDES and are kept around because later in time it is possible they can be classified as an ADES or NDES again. So, adding a new instance into the window could add a descriptor into the ADES or it can move the descriptor from the NDES set to the PDES set. Similarly adding a negative instance into the window could add to the NDES or move an instance from the ADES set to the PDES set. Even forgetting instances can cause one of the existing descriptions to be weakened from losing a single instance. The representation of the hypothesis is in the form of the three description sets that summarize the ADES, NDES, and PDES. The idea behind FLORA is that forgetting should permit faster recovery after a context change by removing old and contradictory information.

The FLORA family of algorithms contains FLORA, FLORA2, FLORA3 and FLORA4. FLORA2 dynamically adapts the size of the window used during the learning process [69]. The idea is that a narrow window may not contain enough instances that allows an appropriate representation of a stable concept. However, a wide window slows down a learner's reaction to the concept drift. Therefore, if drift is present then FLORA reduces the size of the window, and if the environment appears to be stable then grow the size of the window to allow for a sufficient description of the concept. FLORA3 allows for the integration of reoccurring concepts by inspecting if concept descriptions that were useful on some old context. FLORA4 attempts to distinguish between what is real drift and noise that is inherent in data. FLORA4 does so by constructing confidence intervals around the accuracy estimates. A summary of the FLORA family of algorithms can be view in Table 3.1.

Table 3.1 : Summary of FLORA family of algorithms

Algorithm	Algorithm Purpose
FLORA	On-line algorithm used for learning in domains where concept drift is present in the data
FLORA2	Extends FLORA to adaptively adjust the size of the window used in the learning process
FLORA3	Extends FLORA2 by allowing the algorithm to avoid having to re-learn an old concept and saves old concepts until they are relevant again
FLORA4	FLORA4 attempts to determine if drift is actually present in the data or is it just noise

3.2.2 Dynamic Weighted Majority

Kolter & Maloof present the Dynamic Weighted Majority (DWM), whose pseudo code is shown in Fig. 3.2 [70, 28]. DWM is an online learning algorithm that generates an ensemble of classifiers capable of learning data streams that contain concept drift. The classifiers are combined using weighted majority where the weights are determine in a semi-heuristic manner similar to that of the Weighted Majority algorithm [71]. DWM maintains a weighted pool of classifiers (referred to as experts in the original paper) that are trained online. The experts are added and removed from the ensemble, based on the algorithms global performance on new data. The base classifiers used in DWM are the Incremental Tree Inducer (ITI) and online naïve Bayes classifier [1, 72].

DWM begins classifying a new instance, $\mathbf{x}_i$, using each individual classifier's hypothesis. If the classifier incorrectly labels an instance and the current time stamp is an update period, the classifier's voting weight is reduced and the ensemble hypothesis is updated ($\sigma_\lambda \leftarrow \sigma_\lambda + w_j$). If current time stamp is an update period, DWM removes poorly performing classifiers from the ensemble and a new classifier is generated if the ensemble hypothesis for $\mathbf{x}_i$ is incorrect. Normalizing the ensemble voting weights prevents newly added experts from dominating the predictions. All classifiers are then trained on $\mathbf{x}_i$.

DWM was shown to perform quite well on synthetic and real-world datasets containing concept drift. The results demonstrated that DWM maintains a comparable number of classifiers when tested against Blum's Weighted Majority [73] and the AQ algorithm [74,

Input: Training Data $\mathcal{D} = \{\mathbf{x}_i \in X; y_i \in \Omega\}$ where $i = 1, \ldots, n$
Update period, p
Weight decay factor, β
Pruning weight threshold, θ
Define: $\{e, w\}_1^m$: set of classifiers & weights
$\Lambda, \lambda \in \{1, \ldots, c\}$: global & local predictions

$m \leftarrow 1$
$e_m \leftarrow$ `CreateNewClassifier`()
$w_m \leftarrow 1$
for $i = 1, \ldots, n$ **do**
  $\sigma \leftarrow 0$
  for $j = 1, \ldots, m$ **do**
    $\lambda \leftarrow$ `Classify`$(e_j, \mathbf{x}_i)$
    if $\lambda \neq y_i$ **and** $i \mod p = 0$ **then**
      $w_j \leftarrow \beta w_j$
    end if
    $\sigma_\lambda \leftarrow \sigma_\lambda + w_j$
  end for
  $\Lambda \leftarrow \arg\max_j \sigma_j$
  if $i \mod p = 0$ **then**
    $w \leftarrow$`NormalizeWeights`(w)
    $\{e, w\} \leftarrow$`RemoveClassifiers`$(\{e, w\}, \theta)$
    if $\Lambda \neq y_i$ **then**
      $m \leftarrow m + 1$
      $e_m \leftarrow$ `CreateNewClassifier`()
      $w_m \leftarrow 1$
    end if
  end if
  for $j = 1, \ldots, m$ **do**
    $e_j \leftarrow$`Train`$(e_j, \mathbf{x}_i, y_i)$
  end for
end for

Fig. 3.2 : DWM pseudo code [70, 28]

75]. DWM achieves a higher overall accuracy and converged to those accuracies more quickly. However, DWM is not the ideal candidate for learning data where the classes are not distributed equally because there are not mechanisms built into DWM that ensures classifiers will not be biased towards a majority class.

3.2.3 ONSBoost

Online boosting, presented by Oza, is an ensemble-based approach using on-line learners and its tailored to be an on-line implementation of AdaBoost [33, 76]. Like AdaBoost, on-line boosting assumes the data are being drawn from a fixed yet unknown distribution, which is an assumption that is often not true. Pocock *et al.* present an extension to Oza's boosting and FloatBoost called On-line Nonstationary Boosting (ONSBoost) [29]. A floating search is integrated into on-line boosting that allows the addition of new classifiers and the removal of poorly performing classifiers [77]. The combination of the floating search and on-line boosting allows the resetting of outdated/inaccurate classifiers, which allows ONSBoost to adjust to a dynamically changing environment. ONSBoost is implemented in two phases. The first phase is that of Oza's on-line boosting. In fact, given the correct selection of parameters, ONSBoost reduces to on-line boosting. ONSBoost maintains a window of data used for testing the classifiers in the ensemble. The idea is that the window on data represents the most recent distribution of data that yields a classifier's performance on the most recent environment. Like DWM, ONSBoost uses an update parameter to determine when the algorithm needs to be updated. Preliminary results indicate the floating search extension to Oza's on-line boosting is capable of learning in dynamically changing environments. However, like DWM there is not mechanism built into ONSBoost to handle unbalanced data.

3.2.4 SEA

The Streaming Ensemble Algorithm (SEA) was one of the first batch based ensemble algorithms for concept drift [49]. SEA, whose pseudo code is shown in Fig. 3.3, processes batch data presented at a time stamp t by building classifiers until k classifiers have been generated on k different batches of data. Once the ensemble reaches size k, new classifiers are added only if they satisfy a quality criteria based on the estimated ability to improve

the ensemble performance.

Input: Training Data $\mathcal{D}^{(t)} = \{\mathbf{x}_i^{(t)} \in X, y_c^{(t)} \in \Omega\}$ where $c = 1, \ldots, C$
  BaseClassifier learning algorithm
  k: maximum ensemble size
  for $t = 1, 2, \ldots$ **do**
    Call **BaseClassifier** with $\mathcal{D}^{(t)}$
    if $t > k$ **then**
    - Evaluate ensemble on $\mathcal{D}^{(t)}$ for composite hypothesis $H(\mathbf{x}_n)$ using simple majority vote, $n = 1, 2, \ldots, m^{(t)}$
    - Train new classifier on data $\mathcal{D}^{(t)}$ and obtain individual classifier hypothesis: $h_j(\mathbf{x}_n)$ where $j = 1, 2, \ldots, k$ and $n = 1, 2, \ldots, m^{(t)}$

      for $n = 1, 2, \ldots, m^{(t)}$ **do**
        PC^*: classification percentage of true class $c^*(n)$ for instance n
        PC_C: classification percentage of all classes c for instance n
        P_1: top classification percentage among classifiers for instance n
        P_2: second highest classification percentage among classifiers for instance n
        for $j = 1, \ldots, k$ **do**
          if $h_j(\mathbf{x}_n) = c^*(n)$ **then**
            if $H(\mathbf{x}_n) = c^*(n)$ **then**
              Reward: $Q_j = Q_j + 1 - \|P_1 - P_2\|$
            else
              Reward: $Q_j = Q_j + 1 - \|P_1 - PC^*\|$
            end if
          else
            Penalty: $Q_j = Q_j - \left(1 - \|PC^* - PC_{h_j(\mathbf{x}_n)}\|\right)$
          end if
        end for
      end for
      if $Q_{k-1} > Q_j \; \forall j$ **then**
        Prune classifier j
      end if
    end if
  end for

Fig. 3.3 : SEA pseudo code [13]

Quality is defined as a classifier's ability to correctly classify data relative to the error of the ensemble. Classifiers are either rewarded for their quality or penalized based on their decision. Classifiers correctly labelling an instance $\mathbf{x}_i$ are rewarded proportional to the margin between the support of the two highest voted classes, namely P_1 and P_2. The

amount of reward provided to classifiers depends on whether the ensemble chose the correct class. Thus, the experts that perform well when the ensemble members do not agree are especially rewarded. If the classifier's decision for $\mathbf{x}_i$ is incorrect, the classifier is penalized proportional to the margin of error between the ensemble support for the true class PC^* and the support that the classifier provides for the wrong class.

The initial effort made by Street & Kim use a C4.5 decision tree as the base classifier [78]. SEA uses a simple majority vote rather than computing weights inversely proportional to the error of the classifier. Their justification is that there is little or no consistent ensemble performance increase using a weighted majority vote, thus the algorithm is left using a majority vote. SEA is shown to perform particularly well on abruptly changing concepts, namely the shifting hyperplane problem. One of the shortcomings of the SEA algorithm is the possibility that the ensemble may not be able to perform in recurring environments. This is a direct effect of the permanent pruning applied to the ensemble to remove members with a low quality score.

3.2.5 Bagging of Different Size Trees

Bifet *et al.* present an online bagging algorithm designed to learn from massive data streams using different size Hoeffding trees [79]. A Hoeffding tree is an incremental decision tree induction algorithm capable of learning massive data streams that are static with time. Decision tree algorithms such as ID3, CART or C4.5 are batch based learning algorithms and it is important to note that decision trees suffer from the horizon effect: a small adjustment to the training data can yield a significant change to the decision function made by the tree. Hoeffding trees have a theoretical proof that states the tree built in an incremental fashion will be very *similar* to a tree trained on batch data. Hoeffding trees exploit the fact that a small sample can often be enough to choose an optimal splitting attribute. The adaptive size Hoeffding tree (ASHT) method maintains the following properties:

- the tree has a maximum number of split node (referred to as the *size*)
- after one node splits, if the number of split nodes of the ASHT is higher than the *size*, then ASHT deletes some of the tree's nodes to reduce the size.

The motivation for this approach is that a smaller decision tree can adapt faster to changes than a large tree and a large tree can learn over periods of time where the concepts

are not changing. The ASHT with bagging algorithm then generates a set of Hoeffding trees of different sizes. Once a tree reaches its maximum size, the tree deletes its oldest node, the root, and all of its children except the node of the split or delete all nodes of the tree (i.e., train a new tree). The weights of the Hoeffding trees are inversely proportional to the exponentially weighted moving average (EWMA) of a classifier's squared error.

ASHT with bagging was shown to perform quite well on a variety of synthetic datasets compared to decision stumps, naïve Bayes, Oza boosting, OCBoost, and FLBoost. It is important to note that all algorithms tested in [79] are online algorithms.

3.2.6 Bagging Using `ADWIN`

`ADWIN` is an adaptive sliding window algorithm with an estimator and change detection capabilities [23]. `ADWIN` expects that data are presented in an incremental fashion like $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t, \ldots$ and a confidence $\delta \in [0, 1]$ where t indicates any arbitrary time stamp. The data are sampled from a distribution $\mathcal{D}^{(t)}$ where μ_t and σ_t^2 are the true mean and variance of $\mathcal{D}^{(t)}$, respectively.

The new data are placed into a window of size $\mathcal{W}$ and the observed mean of the window, $\hat{\mu}_{\mathcal{W}}^{(t)}$, is computed. As the window grows $\hat{\mu}_{\mathcal{W}}^{(t)}$ (observed) approaches $\mu_{\mathcal{W}}^{(t)}$ (true). `ADWIN` determines when two subwindows within $\mathcal{W}$ exhibit distinct enough averages that one can conclude that the expected values of the subwindows are different with confidence δ. Using a hypothesis test `ADWIN` drops the older window when the null hypothesis is no longer true (i.e., the two sub-windows are drawn from different distributions). `ADWIN` bagging implements Oza's online bagging with `ADWIN` as a change detector and as an estimator for the weights of the boosting method. When `ADWIN` detects change, a new classifier is added to the ensemble and poorest performing classifier is removed. Results have suggested that `ADWIN` bagging performs as well as ASHT bagging on many datasets. It should be noted that bagging, whether it be with `ADWIN` or ASHT, outperforms the single classifier in terms of overall accuracy [80]. A more rigorous theoretical analysis and description of `ADWIN` can be found in Bifet's Ph.D. thesis [23].

Table 3.2 : Weight update equations for the different boosting schemes [82]

Algorithm	$\alpha_d^{(t)}$	$D^{(t+1)}(\mathbf{x}_i^d)$
AdaC1	$\frac{1}{2}\log\frac{1+\sum_i y_d(i)h_t(\mathbf{x}_i^d)C_iD^{(t)}(\mathbf{x}_i^d)}{1-\sum_i y_d(i)h_t(\mathbf{x}_i^d)C_iD^{(t)}(\mathbf{x}_i^d)}$	$\frac{D^{(t)}(\mathbf{x}_i^d)\exp\left(-\alpha_d^{(t)}C_ih_t(\mathbf{x}_i^d)y_d(i)\right)}{Z}$
AdaC2	$\frac{1}{2}\log\frac{\sum_{i:y_d(i)=h_t(\mathbf{x}_i^d)}C_iD^{(t)}(\mathbf{x}_i^d)}{\sum_{i:y_d(i)\neq h_t(\mathbf{x}_i^d)}C_iD^{(t)}(\mathbf{x}_i^d)}$	$\frac{C_iD^{(t)}(\mathbf{x}_i^d)\exp\left(-\alpha_d^{(t)}h_t(\mathbf{x}_i^d)y_d(i)\right)}{Z}$
AdaC3	$\frac{1}{2}\log\frac{\sum_i C_iD^{(t)}(\mathbf{x}_i^d)+\sum_i y_d(i)h_t(\mathbf{x}_i^d)C_i^2D^{(t)}(\mathbf{x}_i^d)}{\sum_i C_iD^{(t)}(\mathbf{x}_i^d)-\sum_i y_d(i)h_t(\mathbf{x}_i^d)C_i^2D^{(t)}(\mathbf{x}_i^d)}$	$\frac{C_iD^{(t)}(\mathbf{x}_i^d)\exp\left(-\alpha_d^{(t)}C_ih_t(\mathbf{x}_i^d)y_d(i)\right)}{Z}$

3.2.7 Cost Sensitive Boosting

Cost sensitive boosting for concept drift is on a boosting algorithm that attaches costs to instances so the the performance of the minority class may be increased [81, 82]. The algorithm, whose pseudo code is shown in Fig. 3.4, is geared towards a binary classification task. The CSB for concept drift estimates costs of old data w.r.t. to new data. These cost are integrated into the boosting procedure.

The purpose of cost sensitive boosting for concept drift is to minimize the error on a similar (relevant) dataset and the reduction of the misclassification of a different dataset that comes from a different distribution. Let $T_d = \{\mathbf{x}_i^d, y_i^d\}_{i=1}^n$ and $T_s = \{\mathbf{x}_j^s, y_j^s\}_{j=1}^m$ represent the labelled different and same distribution data respectively. The objective is to generate a classifier ensemble that classifies unseen-same distribution data with minimum error, by training on T_s supplemented by relevant instances in T_d. Like AdaCost, which has three variants, cost-sensitive boosting for concept drift [82] has three variants as well where the weight update factor, $\alpha_s^{(t)}$, and distribution weights, $D^{(t+1)}(\mathbf{x}_i^d)$, are determined using Table 3.2. C_i are relevance costs attached to each of the instances in T_d. The re-weighting mechanism in cost-sensitive boosting continues to use the error computed over the different training distribution as the base for determining the weight update factor. The algorithm is looking not only to classify examples in the same distribution but also the important instances in the different dataset.

The results for CSB for concept drift demonstrate that the algorithm is effective at learning the different distribution [81]; however the results did not include any standard datasets that are used to demonstrate effectiveness on datasets with concept drift. We

Input: Labeled datasets T_d and T_s and the number of iterations T.

Compute a cost item $C_i \in [0, 1]$ for each instance $\left(\mathbf{x}_i^d, y_d(i)\right) \in T_d$
Initialize weight vector $D^{(1)}(\mathbf{x}_i^{s,d}) = 1/(n+m)$
for $t = 1, \ldots, T$ **do**

1. Train base learner using distribution $D^{(t)}$
2. Obtain hypothesis $h_t : X \to Y$ where $Y \in \{-1, 1\}$
3. Calculate weighted errors, $\varepsilon_s^{(t)}$ and $\varepsilon_d^{(t)}$ on T_s and T_d respectively
4. Choose $\alpha_d^{(t)}$ (refer to Table 3.2)
5. Update weight vectors $D^{(t+1)}$ augmented by the cost items C_i (refer to Table 3.2)
6. Set
$$\alpha_s^{(t)} = \frac{1}{2} \log \frac{1 - \varepsilon_s^{(t)}}{\varepsilon_s^{(t)}} \tag{3.8}$$
7. Update weight vectors
$$D^{(t+1)}\left(\mathbf{x}_j^s\right) = \frac{D^{(t)}(\mathbf{x}_j^s) \exp\left(-\alpha_d^{(t)} h_t\left(\mathbf{x}_j^s\right) y_s(j)\right)}{Z} \tag{3.9}$$
8. Obtain composite hypothesis
$$H(\mathbf{x}) = \texttt{sign}\left(\sum_{t=1}^{T} \alpha_s^{(t)} h_t\left(\mathbf{x}\right)\right) \tag{3.10}$$

end for

Fig. 3.4 : Cost-sensitive Boosting for Concept Drift

should note that one possible issue with this approach is there is no mention of how to integrate this algorithm to incrementally learn from data over time.

3.2.8 Relationship to Transfer Learning

Transfer learning is a field related to concept drift. In fact one can claim that knowledge transfer is an integral part to defining concept drift. Concept drift is really the fusion of three areas, namely: time series, knowledge transfer, and adaptivity. Transfer learning involves transferring knowledge between information in the training and testing datasets. In order words, if training data are generated by source S_1, the testing data are generated from source S_2 and $S_1 \neq S_2$. So, how can information in the training data be used to minimize the

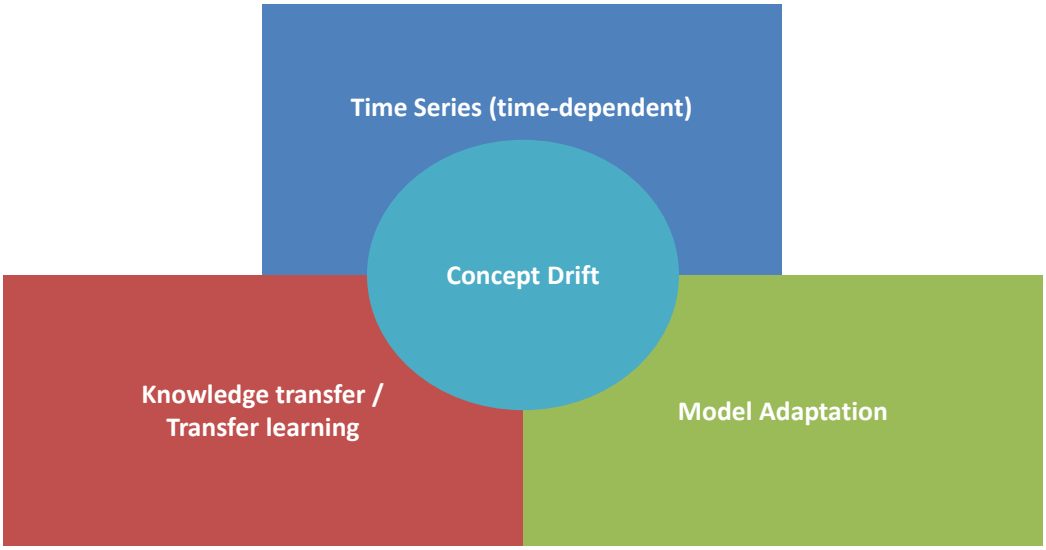


Fig. 3.5 : Relationship between knowledge transfer, time-series analysis, model adaptation and concept drift [15]

error on the testing data [83]. With this definition of transfer learning, it appears to be the same as concept drift however, there are subtle and very distinct differences between concept drift and transfer learning. First, transfer learning does not imply that we are dealing with a dataset that is potentially time series or dependent upon time, rather the training and testing sets are drawn from two sources. Second, concept drift requires adaptivity since it must learn the concepts that are dependent upon time. Fig. 3.5 depicts the relationship between concept drift, time series, knowledge transfer, and model adaptivity [15].

3.2.9 Drift Detection

Concept drift algorithms are usually associated with incremental learning of streaming data, where new datasets become available in batches or in an instance-by-instance basis, resulting in batch or on-line learning, respectively [84]. Given such data, the (active) drift detection itself can be parametric or non-parametric, depending on whether a specific underlying distribution is assumed [85]. Many parametric algorithms use a CUSUM (cumulative sum) based mechanism, which is traditionally used for control charts in detecting nonstationary changes in process data [85]. A series of successful implementations of this approach are proposed by Alippi & Roveri, including CI-CUSUM [86, 87, 88], a pdf free extension of the

Input: Training Data $\mathcal{D}^{(t)} = \{\mathbf{x}_i^{(t)} \in X, y_c^{(t)} \in \Omega\}$ where $c = 1, \ldots, C$
f: number of deviations
for $t = 1, 2, \ldots$ **do**
1. Evaluate classifier, h, on the current batch of data $\mathcal{D}^{(t)}$ to get the error ε_t.
2. Declare that change has been found if ε_t exceeds the f-sigma limit, $\varepsilon_t > p + f\sigma$.

end for

Fig. 3.6 : Shewhart drift detection algorithm [84]

traditional CUSUM, or more recently the intersection of confidence intervals (ICI) rule [89]. Some drift detection approaches such as the Early Drift Detection Method (EDDM) [63] and other similar approaches, do not make any assumptions on feature distribution, but rather monitor a classifiers' accuracy or some distance metric to detect drift.

3.2.9.1 Control Charts for Drift Detection

Control charts have been widely used and adapted for detection of concept drift in data streams. The algorithm shown in Fig. 3.6 is a basic Shewhart control chart presented by Kuncheva [84]. The algorithm simply computes the error on each batch of data then the mean of the previous errors, p, and the standard deviation, σ. If the error on the most recent data exceeds a threshold determined from mean and standard deviation then a changed is declared. Then it is up to the designer to determine what action to take once change is detected.

The CUmulative SUM (CUSUM) control chart is widely used by researcher. and most notably in Alippi's computational intelligence CUSUM (CI-CUSUM) [87, 88]. The CUSUM mechanism typically requires that a probability distribution is specified, which is typically a Gaussian. The CI-CUSUM is a *pdf* free extension of CUSUM (refer to the pseudo code in Fig. 3.7). CI-CUSUM builds a feature vector, $\boldsymbol{\varphi}$, derived from the data at each time stamp and principle component analysis (PCA) is applied to $\boldsymbol{\varphi}$. A mean vector and covariance matrix are computed from the PCA derived feature. The detection method uses a configuration sequence (size CS), beginning at time t_{drift}, to configure the parameters of the test. t_{drift} indicates the last time drift was detected. Drift is detected using a

threshold defined completely by the data. This threshold, calculated using the configuration parameters, is updated when $t - t_{drift} = CS$. The primary weakness of the CI-CUSUM is that the method only uses $p(X)$ to detect change, i.e., class information is not used, which could be a possible limitation (refer to Section 2.2). However, one can use CI-CUSUM on each class independently. Second, the Just-In-Time classifier uses a k-NN, which limits the classifier used in the design. The CI-CUSUM is known to react poorly to slow drift, however slow drift was addressed in [90].

3.2.9.2 Classifier Error for Drift Detection

Some drift detection approaches such as the *Early Drift Detection Method* (EDDM) [63] and other similar approaches [91, 92], do not make any assumptions on feature distribution, but rather monitor a classifiers' accuracy or some distance metric to detect drift. EDDM uses the distance between classification errors (number of instances between the two errors) to detect change instead of strictly using classification error as implemented with DDM. To implement EDDM, the authors calculate the distance between two errors (p'_i) and the standard deviation (s'_i). Each value of p'_i and s'_i are stored. Then, when $p'_i + 2 \cdot s'_i$ reaches a maximum value (obtaining p'_{max} and s'_{max}) the quantity $p'_{max} + 2 \cdot s'_{max}$ corresponds with the point where the distribution distance between two errors are maximum. Their implementation uses the following two thresholds:

1. $(p'_i + 2 \cdot s'_i)/(p'_{max} + 2 \cdot s'_{max}) < \alpha$ for a warning level
2. $(p'_i + 2 \cdot s'_i)/(p'_{max} + 2 \cdot s'_{max}) < \beta$ for a drift level

EDDM considers the thresholds and searches for concept drift when a minimum of 30 errors have occurred. The implementations of EDDM have shown that the approach is an easy to implement wrapper method to learning from concept drift and the algorithm appears to be quite robust to noise. EDDM also appears to detect when the base classifier begins to overfit to the training data.

3.2.9.3 Framework for Drift Detection

Cielslak & Chawla suggest Hellinger distance, not for detecting concept drift in an incremental learning setting, but rather to detect bias between training and test data

Input: Data $\mathcal{D}^{(t)}$
Configuration size CS
Initialize $t_{drift} = 0$
for $t = 1, \ldots, T$ **do**
1) Compute feature vector $\phi(t)$ from dataset $\mathcal{D}^{(t)}$
2) Apply PCA to $\phi(t)$ and obtain $\phi_{PCA}(t)$
if $t - t_{drift} == CS$ **then**
3) Compute mean $\hat{M}_0$ and covariance matrix $\hat{C}_0$ for null hypothesis

$$\hat{M}_0 = \frac{1}{CS} \sum_{k=t_{drift}}^{t_{drift}+CS} \phi_{PCA}(t) \tag{3.11}$$

$$\hat{C}_0 = \frac{1}{CS} \left(\phi_{PCA}(t) - \hat{M}_0\right) \left(\phi_{PCA}(t) - \hat{M}_0\right)^T \tag{3.12}$$

$H_0 : \theta_0 = \{\hat{M}_0, \hat{C}_0\}$
4) Compute confidence interval $M_{1,max}$, $M_{1,min}$, $C_{1,max}$, and $C_{1,min}$. Select alternative hypothesis $\theta_1 = \{M_1, C_1\}$
5) Configuration Parameters
Log-likelihood:

$$R(\tau) = \sum_{k=t_{drift}}^{\tau} \log \left(\frac{P_{\theta_1}\left(\phi_{PCA}(k)\right)}{P_{\theta_0}\left(\phi_{PCA}(k)\right)} \right) \tag{3.13}$$

Minimum: $m(\tau) = \min_\tau R(\tau)$ for $\tau = t_{drift}, \ldots, t$
CUSUM parameter: $g(\tau) = R(\tau) - m(\tau)$ for $\tau = t_{drift}, \ldots, t$
Threshold: $h(\tau) = \max_\tau g(\tau)$ for $\tau = t_{drift}, \ldots, t$
else if $t - t_{drift} > CS$ **then**
6) Update $R(t)$

$$R(t) = \sum_{k=t_{drift}}^{t} \log \left(\frac{P_{\theta_1}\left(\phi_{PCA}(k)\right)}{P_{\theta_0}\left(\phi_{PCA}(k)\right)} \right) \tag{3.14}$$

7) Compute $m(t)$ and $g(t)$
if $g(t) > m(t)$ **then**
$t_{drift} = t$
$drift = 1$
else
$drift = 0$
end if
end if
end for

Fig. 3.7 : Log-likelihood CUSUM test for drift detection [13]

distributions [93]. The Hellinger distance is used to quantify the similarity between two probability distributions. We will later extend this approach to learning incrementally from concept drift and use the Hellinger distance as a measure to help HDDDM (presented in Section 4.4) signal drift. In this framework, the authors generate a distribution of posterior probabilities on a validation set (carved out from training data) and a distribution of posterior probabilities on a corresponding test dataset. The non-parametric Kolmogorov-Smirnov statistical test (KS-test) is applied to measure the significance between the probability estimates of the validation and test datasets [94]. The output of the KS-test is a p-value, which can be used to determine if the probability estimates are being drawn from two different probability distributions. The Hellinger distance is used in conjunction with the KS-test to measure the divergence between the distribution of the raw features. The Hellinger distance is integrated into the framework by forming a baseline comparison that is the Hellinger distance between the original training and test dataset. Bias is then injected into the test set (see [8] for a summary of bias in data). Bias may be *missing completely at random* (MCAR), *missing at random* (MAR), or *missing not at random* (MNAR) [93]. A baseline Hellinger distance (i.e. when no bias is present between training/testing sets) and the distance after bias is injected into the dataset are observed. Using the baseline Hellinger distance (distance between training and testing set with not bias) with the biased Hellinger distance (distance between training and testing set with bias) allows the authors to show that the Hellinger distance is a viable measure for identifying various levels of bias between training/testing datasets.

3.3 Class Imbalance in Machine Learning

Typical learning algorithms (SVM, MLPNN, etc.) tend to bias themselves towards the majority class when faced with an imbalanced learning scenario. This is because the minority class is under-represented in the training data. The field of unbalanced data investigates methods to improve the *recall* (performance on the minority class) of the minority class without performing harm to the overall classification accuracy. This section of the thesis presents relevant research focused on learning a minority class in an imbalanced learning scenario.

3.3.1 Sampling Methods

Sampling methods are very popular for learning from imbalanced datasets because of their simplicity. These methods use some form of sampling, random or synthetic, to modify the original training dataset and generate a new training dataset that is not as imbalanced as the original set. Random sampling algorithms can add or remove randomly selected instances from the training data whereas a synthetic sampling method generates synthetic instances to *over-sample* the training data. Several studies have shown that the overall performance of a base classifier can be increased by using sampling methods [95]. For example, the C4.5 decision tree was evaluated on a variety of different imbalanced datasets in [96]. The summary of work demonstrated, among other results, that pruning the tree is usually detrimental to learning from imbalanced datasets, however after applying sampling methods, pruning can help as it improves the generalization of the decision tree. However, we should clearly state that the sampling heuristic is not guaranteed to work for every imbalanced learning problem.

3.3.1.1 Under/Over Sampling of Data

Random over-sampling aims to generate exact replicates of the minority class instances to balance the class distribution. Many times the new dataset created after over-sampling the post-balance ratio may not be $0.5 : 0.5$. Rather, the sampling heuristic creates a *less* imbalanced learning problem. Over-sampling provides a mechanism for varying the degree of class distribution balance to any desired level. Random under-sampling aims to balance the class distribution by discarding majority class instances from the training dataset [97].

Both of these methods are very intuitive. Unfortunately they come with a few consequences. Random over-sampling, as stated previously, uses exact replicates of the minority data, which tends to have classifiers over-fit to the minority class instances. Random under-sampling throws away data from the majority class. This causes a loss of information in the minority class. Even though the majority class may not be the most important class in the problem, it is still important information from the majority class is not thrown [7].

Input: Number of minority class examples T
Amount of SMOTE $N\%$
Number of nearest neighbors, k
number of attributes, m
for $i = 1, 2, \ldots, T$ **do**
Find k nearest neighbors of $\mathbf{x}_i$
$\hat{N} = \lfloor N/100 \rfloor$
while $\hat{N} \neq 0$ **do**
1. Randomly select one of the k nearest neighbors, call this $\bar{\mathbf{x}}$
2. Select a random number $\alpha \in [0, 1]$
3. $\hat{\mathbf{x}} = \mathbf{x}_i + \alpha\,(\bar{\mathbf{x}} - \mathbf{x}_i)$
4. Append $\hat{\mathbf{x}}$ to $\mathcal{S}$
5. $\hat{N} = \hat{N} - 1$

end while
end for
Output: Return synthetic data $\mathcal{S}$

Fig. 3.8 : SMOTE algorithm pseudo code

3.3.1.2 Synthetic Minority Oversampling Technique

The previous two methods of sampling the majority or minority data take place in data space. In other words, the original sampling domain is modified, and not the data instances in the minority set. The *Synthetic Minority Oversampling TEchnique* (SMOTE) over-samples the minority class by generating synthetic instances rather than over-sampling with exact replicates of minority instances [98]. SMOTE is one of the most popular imbalanced data techniques and is considered a benchmark to compare all other algorithms meant to combat class imbalance. This method of over-sampling the minority class occurs in feature space rather than modifying data space. It creates synthetic instances that lie on the line segment between a minority class instance and a randomly chosen nearest neighbor (also belonging to the minority class).

The SMOTE algorithm, whose pseudo-code is shown in Fig. 3.8, has two primary free parameters, k and N. The amount of SMOTE is controlled by N, a percentage and should be a multiple of 100, and the number of nearest neighbors is an integer used to create the synthetic instances that is controlled by k. The algorithm is implemented by looping through every minority class instance in a database and populating a set of synthetic

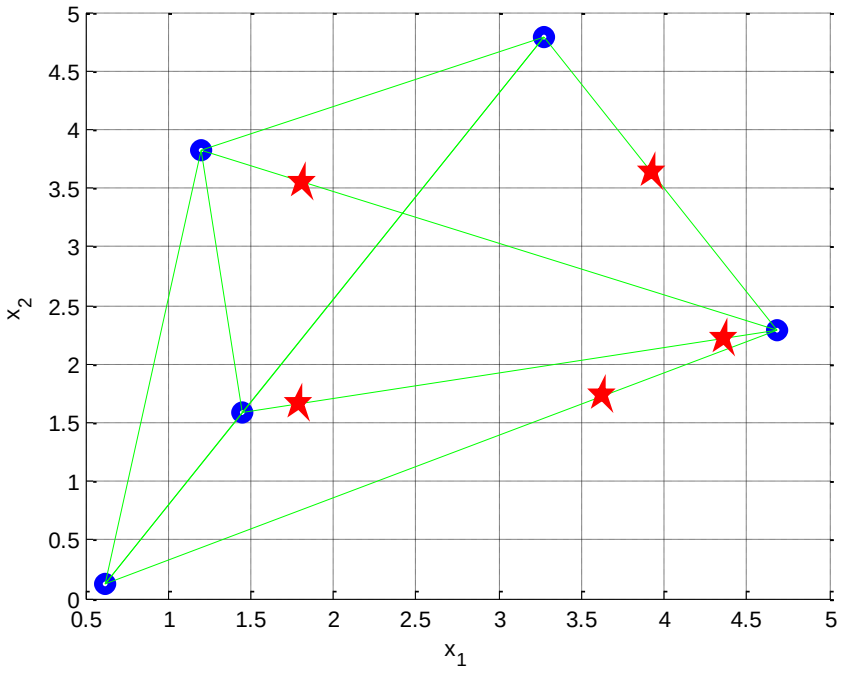


Fig. 3.9 : Demonstration of SMOTE sampling methods on synthetic data

instances that can be used for training. The synthetic instances are created by selecting a random nearest neighbor within the k nearest and generating a random number, *gap* (denoted by α in Fig. 3.8), that is bound by $[0, 1]$. The *gap* parameter determines the location of the synthetic instances on the line segment between the minority instance and the nearest neighbor. Finally, the synthetic instances are generated by multiplying the difference between the minority class instance and its selected nearest neighbor by *gap*. This result is then added to the minority class dataset under consideration. Thus, SMOTE forces the decision region of the minority class to become more general and avoid the issues of standard under/over-sampling.

A visual example of SMOTE can be found in Fig. 3.9. The blue points represent the original minority class data and the red points are the instances that have been generated using SMOTE. Notice that all instances generated with SMOTE lie on a line segment between the original minority class instances. One disadvantage of SMOTE is that it does not take into account any potential cost of generating synthetic instances in the majority class feature space because SMOTE requires no information about the majority class.

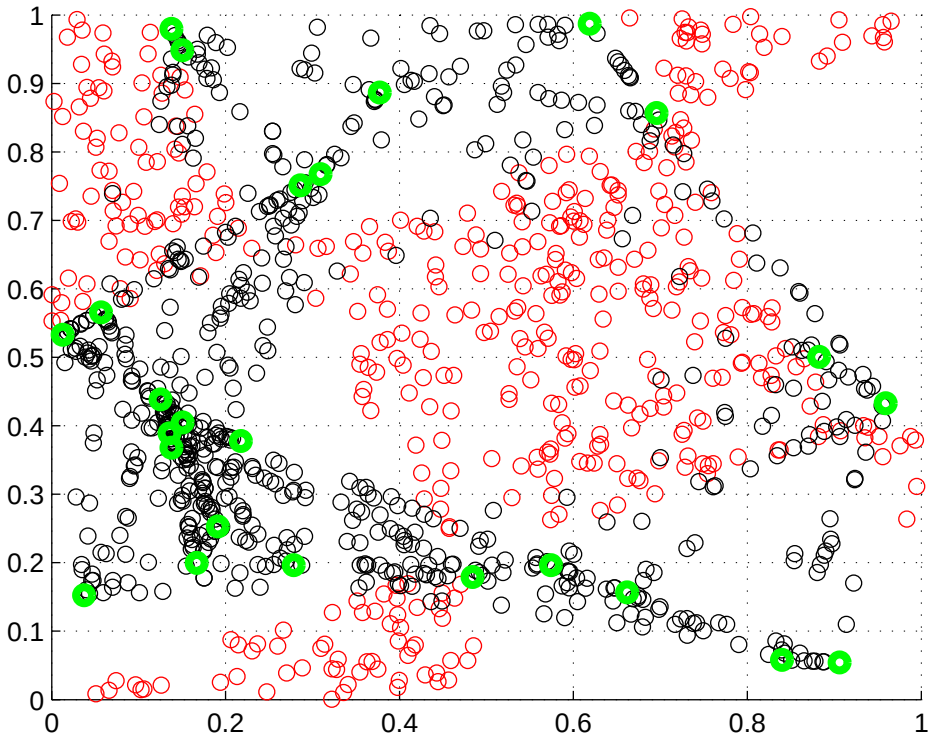


Fig. 3.10 : SMOTEs generation of synthetic minority class instances in the regions of the majority class feature space, thus leading to the degradation of the overall accuracy of a classifier. Green instances represent are the original minority class instances, red are the majority class instances, and black are the synthetic instances.

Therefore, SMOTE can populate regions of feature space that belong to the majority class with synthetic instances. Consider a rotated checkerboard dataset with a class balance of $\approx 1:40$ as shown in Fig. 3.10. In this figure, the green dots represent the locations of the original minority class instances, the red dots are the locations of the majority class instances and the black dots represent the locations of synthetic instances generated by calling SMOTE. We observe from this figure that SMOTE is populating the regions of the majority class feature space with synthetic instances. However, it should be noted that SMOTE has been shown to be a very robust algorithm and populating the regions of the majority class feature space is only a *possibility*.

3.3.1.3 *ADASYN*

ADAptive SYNthetic sampling (ADSYN) is a synthetic sampling algorithm that rectifies some of SMOTE's weaknesses. As mentioned earlier, SMOTE generates the same number of synthetic instances for each *minority class instance* in the training data. There is no consideration of the neighboring instances potentially from a different class when the synthetic instances are generated. Various methods of sampling, such as Borderline-SMOTE [99] and ADASYN [100], have been developed to work around this limitation.

ADASYN begins by determining the nearest neighbors of a minority class instance; however unlike SMOTE, ADASYN considers *majority class nearest neighbors* as well. Using this information ADASYN determines the number of synthetic instances that need to be generated for a minority class instance. The rest of the ADASYN algorithm follows SMOTE. The key difference is that ADASYN uses information in the majority class to determine the number of instances that need to be generated from each minority class instance rather than using the SMOTE mentality of keeping the number of synthetic instances generated around each minority class instance fixed.

3.3.2 Ensemble Methods for Class Imbalance

Ensembles of classifiers can be used when classes are imbalanced. In this section a variety of different approaches are presented for handling class imbalance with ensembles of classifiers.

3.3.2.1 *Bagging Variation*

The bagging ensemble variation (BEV) was developed to classify imbalanced data where concepts remain static (i.e. no concept drift) [101]. The goal is to develop a classification model based on bagging to maximally use information in the minority class without the need to generate synthetic data or making changes to the existing classification model.

BEV, whose pseudo code is shown in Fig. 3.11, begins by collecting the majority class (Ω_-) and dividing the set into k disjoint sets. Each of the disjoint sets of the majority class data is combined with the entire minority class population in $\mathcal{D}$. Fig. 3.12 provides insight to the formulation of the training sets for each of the individual classifiers. The **BaseClassifier** learning algorithm is called on each of the datasets to form k classifiers that have been trained on the entire minority class population and a portion of the majority class.

Input: Dataset $\mathcal{D} = \{\mathbf{x}_i \in X; y_i \in \Omega\}$ where $i = 1, \ldots, N$
BaseClassifier learning algorithm
k: number of classifiers to generate
Ω_+: minority class
Ω_-: majority class

1. Form k disjoint sets of Ω_-, call these sets $\mathcal{D}_m$ where $m = 1, \ldots, k$.
2. Combine Ω_+ with each dataset $\mathcal{D}_m$ to form $\hat{\mathcal{D}}_m$.
3. Call **BaseClassifier** on each dataset, $\hat{\mathcal{D}}_m$.
4. Combine classifiers using majority voting

Fig. 3.11 : Bagging ensemble variation algorithm pseudo code

BEV works at improving the accuracy on the minority class over traditional bagging by not using a simple bootstrap sample to train a classifier. For bagging, the bootstrap sample may, and most likely will, contain fewer minority class instances than the original dataset. By forming disjoint datasets of the majority class, BEV is able to have each classifier trained on all the minority class and a subset of the majority class, thus the ensemble decision can be confident about a minority class instance, as each classifier is trained on all Ω_+ and the combination of all classifiers will formulate the confidence in the majority class.

3.3.2.2 Learn^{++}.UDNC

Learn^{++}.NC was designed learning from new datasets than can include new classes without accessing previously seen data [68]. The approach was quite effective at reducing the outvoting problem with incremental learning systems. Learn^{++}.UDNC is an extension to Learn^{++}.NC that allows the algorithm to incrementally learn new concept classes from unbalanced datasets [102]. This version of Learn^{++} combines preliminary confidence measures introduced in Learn^{++}.NC, with a transfer function that adjusts the voting weights of classifiers based on the number of instances seen from each class, as well as the class imbalance in each dataset. This approach works well in situations where an incremental learning is required to classify data coming from moderately imbalanced data distributions and new classes are being added and / or removed incrementally. The ensemble decision itself, on the other hand, uses a preliminary confidence measure and adjusts this confidence

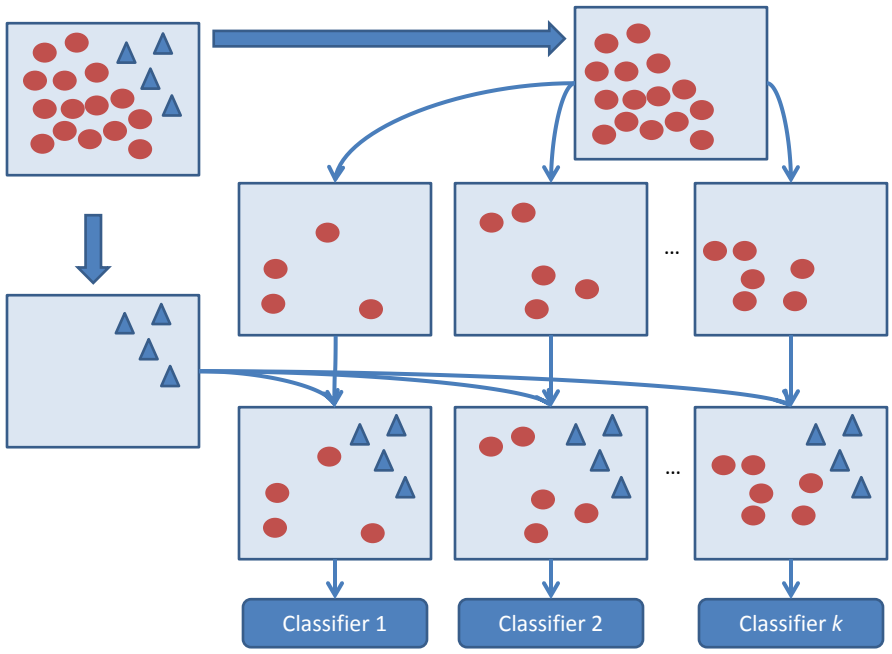


Fig. 3.12 : Disjoint set formulation used the bagging ensemble variation [101]. The majority class (red circles) are divided in disjoint sets and combined with the minority class (blue triangle) to form a new dataset used to generate a classifier.

measures by the cardinality of each class in the training set. Unlike traditional ensembles for learning imbalanced data, Learn^{++}.UDNC is designed to learn when the imbalance may vary drastically between the introduction of new training data.

Learn^{++}.UDNC was able to consistently outperform fuzzy ARTMAP under a variety of incremental learning scenarios and with a wide margin on unbalanced data problems. This was observed with both synthetic and real-world incremental learning problems. While not quite as effective as SMOTE on severely unbalanced data, Learn^{++}.UDNC performs comparably to SMOTE on minority class recall on moderately imbalanced datasets, but with the added advantage of learning incrementally, without applying any over-sampling (or under-sampling). This algorithm has shown the ability to perform well on a broad spectrum of incremental learning problems where the previous members of the Learn^{++} algorithms are not able to be reliable predictors on all classes.

3.3.2.3 *SMOTEBoost*

The success of AdaBoost and boosting in general has led to a vast set of algorithms derived from Freund & Schapire's original implementation. SMOTEBoost is a novel approach for learning a rare class based on a combination of SMOTE and AdaBoost.M2 [103]. However, unlike AdaBoost.M2 where all misclassified instances are given equal weights, SMOTEBoost creates synthetic examples from the minority class, thus indirectly changing the updating weights and compensating for skewed distributions. The motivation for this algorithm can be summarized as: 1) use SMOTE to improve the performance on a minority class, and 2) use boosting to avoid sacrificing the overall accuracy on the entire dataset.

SMOTEBoost, whose pseudo code is shown in Fig. 3.13, begins by initializing a uniform distribution over the instances. SMOTE is then called on the minority class, Ω_m, and added into the data for the generation of h_t, which is the classifier trained during the tth boosting iteration on $\mathcal{D}_t$. Note that $\mathcal{D}_t \in \{\mathcal{D} \cup \mathcal{S}_t\}$ where $\mathcal{S}_t$ is the synthetic data generated with SMOTE. The rest of the pseudo code follows exactly like AdaBoost.M2. The composite hypothesis is made via weighted majority voting. There are several notable points to make about SMOTEBoost:

- SMOTE introduces a broad representation of the minority class at each boosting iteration
- Introducing SMOTE at every boosting iteration increases the diversity among classifiers in the ensemble
- SMOTE should improve the performance on the minority class while boosting improves the overall accuracy of the ensemble

SMOTEBoost was shown to be quite effective at boosting the recall of the minority class while significantly increasing the overall performance compared to AdaBoost.M2. In many cases, the F-measure* was significantly increased for SMOTEBoost while little to no significant decrease was observed in the precision. While SMOTEBoost has been shown to be quite effective at learning a rare class, it is not designed to learn when the distribution of the data is dynamic.

*Statistical measures will be discussed in more detail in Section 5.2.2

Input: Dataset $\mathcal{D} = \{\mathbf{x}_i \in X; y_i \in \Omega\}$ where $i = 1, \ldots, N$
Ω_m: minority class
T: number of classifiers
Let $\mathcal{B} = \{(i, y) : i = 1, \ldots, N, y \neq y_i\}$
Initialize: $D_1(i) = 1/N$

for $t = 1, \ldots, T$ **do**

1. Modify distribution D_t by creating M synthetic examples from minority class Ω_m using SMOTE algorithm
2. Train a weak learner using distribution D_t and $\mathcal{D}$
3. Compute weak hypothesis, $h_t : X \rightarrow Y$
4. Compute the pseudo-loss of hypothesis h_t

$$\varepsilon_t = \sum_{(i,y)\in\mathcal{B}} D_t(i, y) \left(1 - h_t\left(\mathbf{x}_i, y_i\right) + h_t\left(\mathbf{x}_i, y\right)\right) \tag{3.15}$$

where $h_t\left(\mathbf{x}_i, y_i\right)$ is the confidence given to class y_i by h_t and $h_t\left(\mathbf{x}_i, y\right)$ is the average confidence given to all other classes $y_i \notin y$ by h_t.

5. Set $\beta_t = \varepsilon_t/(1 - \varepsilon_t)$
6. Update distribution

$$D_{t+1} = \frac{D_t}{Z} \beta_t^{\frac{1}{2}(1-h_t(\mathbf{x}_i,y)+h_t(\mathbf{x}_i,y_i))} \tag{3.16}$$

where Z is a normalization constant to assure D_{t+1} is a distribution

end for
Compute composite hypothesis

$$H(\mathbf{x}) = \arg\max_{y\in\Omega} \sum_{t=1}^{T} \log \frac{1}{\beta_t} \cdot h_t(\mathbf{x}, y) \tag{3.17}$$

Fig. 3.13 : SMOTEBoost pseudo code

3.3.2.4 DataBoost-IM

DataBoost-IM, whose pseudo code is shown in Fig. 3.14, combines data generation and boosting to improve the accuracies of the majority and minority class [104]. AdaBoost works by focusing on the hard to classify instances and iteratively attempts to have classifiers trained on this data. However, AdaBoost takes no class information into account when learning the hard to classify instances. DataBoost-IM considers hard to classify instances from the majority and minority classes at each boosting iteration.

Input: Dataset $\mathcal{D} = \{\mathbf{x}_i \in X; y_i \in \Omega\}$ where $i = 1, \ldots, N$
Ω_m: minority class
T: number of classifiers
Weak learning algorithm **WeakLearn**
for $t = 1, \ldots, T$ **do**

1. Identify hard examples from the original data set for all classes
2. Generate synthetic data to balance the training knowledge of different classes
3. Add synthetic data to the original training set to form a new training data set
4. Update and balance the total weights of the different classes in the new training data set
5. Call **WeakLearn**, providing it with the new training set with synthetic data and rebalanced weights
6. Get back a hypothesis, $h_t : X \rightarrow Y$
7. Calculate the error of h_t

$$\varepsilon_t = \sum_{i=1}^{N} D_t(i)[\![h_t(\mathbf{x}_i) \neq y_i]\!] \tag{3.18}$$

8. Set $\beta_t = \varepsilon_t/(1 - \varepsilon_t)$
9. Update distribution

$$D_{t+1} = \frac{D_t}{Z}\beta^{1-[\![h_t(\mathbf{x}_i)=y_i]\!]} \tag{3.19}$$

where Z is a normalization constant to assure D_{t+1} is a distribution

end for
Compute composite hypothesis

$$H(\mathbf{x}) = \arg\max_{y \in \Omega} \sum_{t=1}^{T} \log \frac{1}{\beta_t} \cdot h_t(\mathbf{x}, y) \tag{3.20}$$

Fig. 3.14 : DataBoost-IM pseudo code

DataBoost-IM begins by identifying the hard to classify instances in the dataset $\mathcal{D}$. In the case of AdaBoost the previous classifier is used to increase the weights of the instances that are still being misclassified. Next, synthetic data are generated for the hard to classify instances, which includes instances from both the majority and minority class. Algorithms like SMOTEBoost only use the minority class to generate synthetic data. The class frequencies in the new training set are rebalanced to alleviate the learning algorithm's bias toward the majority class by choosing a reduced number of representative instances

(seed) from both classes. The process of identifying "seed" instances is laid out in [104]. The rest of the algorithm follows as AdaBoost.

The DataBoost-IM approach performs well on a variety of imbalanced data sets found on the UCI machine learning repository [105]. DataBoost-IM performs quite comparably to SMOTEBoost on the datasets presented in the original paper. However, like SMOTEBoost, DataBoost-IM does not perform when faced with incremental learning scenarios and dynamically changing environments.

3.3.2.5 Cost Sensitive Learning

The weighting strategy for AdaBoost works by distinguishing between the correctly and incorrectly classified instances by the most recent classifier. The correctly classified instances have their weights decreased while the opposite is true for incorrectly classified instances. However, this strategy makes no differentiation between classes and decreases/increases the instance weights by a constant. The goal with classifying unbalanced data is to improve the identification performance on the minority class. Sun *et al.* introduce three ways to integrate cost sensitive learning into the AdaBoost algorithm [82]. The cost sensitive learning algorithms are referred to as AdaC1, AdaC2 and AdaC3 corresponding to the instance weight update equations in Eq. (3.21), (3.22), and (3.23).

AdaC1, AdaC2 and AdaC3 are cost sensitive boosting algorithms that attach a misclassification cost to each instance in the training dataset. The cost attachments affect the instance weight update as well as the classifier voting weights. The instance weights are updated using Eq. (3.21), (3.22), and (3.23) where $D^{(t)}(i)$ is the old instance weight for $\mathbf{x}_i$, $C_i \in [0, \infty)$ is the cost attached to $\mathbf{x}_i$, $h_t(\mathbf{x}_i) \in \{-1, +1\}$ is the tth classifier's prediction for $\mathbf{x}_i$, y_i is the true class label and α_t is the weight of the tth classifier.

$$D^{(t+1)}(i) = \frac{D^{(t)}(i) \exp\left(-\alpha_t C_i h_t(\mathbf{x}_i) y_i\right)}{Z_t} \tag{3.21}$$

$$D^{(t+1)}(i) = \frac{C_i D^{(t)}(i) \exp\left(-\alpha_t h_t(\mathbf{x}_i) y_i\right)}{Z_t} \tag{3.22}$$

$$D^{(t+1)}(i) = \frac{C_i D^{(t)}(i) \exp\left(-\alpha_t C_i h_t(\mathbf{x}_i) y_i\right)}{Z_t} \tag{3.23}$$

The classifier weights, α_t, are derived in [82] and will vary pending on which cost sensitive algorithm is applied. The results presented in [82] indicate that AdaC2 is superior to its rivals and experimental evidence was provided to back this claim. While the cost sensitive algorithms can work quite well at improving accuracy on a minority class, it is assumed that a cost-matrix is known for different types of errors. The authors of [82] point this out and indicate that efficient methods should be developed to determine cost factors.

3.4 Joint Problem: Learning Concept Drift from Imbalanced Data

Concept drift and class imbalance have been studied extensively, however they are typically studied independently. In fact, the literature review for concept drift and class imbalance in the previous sections is a very brief description of a few of the more popular approaches to learning data. Far less work has been performed on the joint problem of concept drift and class imbalance. This section of the thesis presents a review of the core approaches currently available for learning concept drift and class imbalance, simultaneously.

3.4.1 Uncorrelated Bagging

Uncorrelated bagging is an ensemble based algorithm for mining data and producing reliable posterior probability estimates when the class distribution in the stream is skewed [61]. The authors of the algorithm make the claim that there is no general correlation between the expected error of the previous model (ensemble) and any type of concept drift in the data stream. This mindset is the motivation behind the authors' choice of averaging votes (simple) rather than a weight majority vote.

The pseudo code for uncorrelated bagging is shown in Fig. 3.15. The algorithm maintains a fixed number of classifiers after k time stamps. Data are presented in batches, $\mathcal{D}^{(t)} = \{\mathbf{x}_i \in X; y_i \in \Omega\}$, at every time stamp t. The positive instances are accumulated as the batches are presented (positive instances refer to the minority class). Classifiers are generated on the *accumulated positive instances* and smaller randomly drawn samples of the negative instances that have been presented in the most recent batch of data. The posterior estimates of the classifiers are computed and averaged to get the posterior estimate of the ensemble. Only the k classifiers generated on the most recent batches are used in generating a composite hypothesis, thus implementing some form of forgetting, and making learning

Input: Current data chunk $\mathcal{D}^{(t)} = \{\mathbf{x}_i \in X; y_i \in \Omega\}$
Test data $\mathcal{T}^{(t)}$
Number of classifiers k
Distribution ratio r
Set of positive examples **AP**
for $t = 1, 2, \ldots$ **do**

1. Split $\mathcal{D}^{(t)}$ into $\mathcal{P}^{(t)}$ and $\mathcal{Q}^{(t)}$ corresponding to the minority and majority classes respectively
2. Update $\mathbf{AP} = \{\mathbf{AP}, \mathcal{P}^{(t)}\}$
3. Calculate the number of negative examples in the sample n_q based on the values of r and n_p
4. **for** $i = 1, 2, \ldots, k$
   (a) Draw a sample of size n_p from $\mathcal{Q}^{(t)}$ without replacement, $\mathcal{O}^{(t)}$
   (b) Train a classifier h_i on $\{\mathcal{O}^{(t)}, \mathbf{AP}\}$
   (c) Compute posterior probability estimates $\{f^i(\mathbf{x})\}_{\mathbf{x}\in\mathcal{T}}$ using h_i

   end for
5. Compute posterior probability estimates by combining ensemble outputs $\{f^E(\mathbf{x})\}_{\mathbf{x}\in\mathcal{T}}$ based on Eq. 3.24.

$$f^E(\mathbf{x}) = \frac{1}{k}\sum_{i=1}^{k} f^i(\mathbf{x}) \tag{3.24}$$

end for

Fig. 3.15 : Uncorrelated bagging pseudo code

in reoccurring environments difficult.

Uncorrelated bagging differs from traditional bagging for several reasons: 1) in bagging, bootstrap samples are used as training sets, and 2) bagging uses simple voting while uncorrelated bagging generates average probabilities. The authors claim that the average vote is more effective than a weighted majority vote. Uncorrelated bagging is presented in the experiments section of this thesis to determine if one can always assume that there is no general correlation between the expected error of the previous model (ensemble) and any type of concept drift in the data stream.

Uncorrelated bagging implicitly makes the assumption that there is no drift minority class. If the minority data were to drift, then the classifiers would be trained on old and quite possibly irrelevant data. This method of training classifiers on all of the old data can lead to serious degradation of the overall accuracy of a classifier if the minority class does not remain stationary. Uncorrelated bagging is evaluated on several real-world and experimental datasets to observe the effect of training classifiers on the accumulated minority class data.

3.4.2 SERA

Uncorrelated bagging simply uses all of the accumulated minority class data to build a classifier, regardless of whether or not the data were relevant to the current environment. The selectively recursive approach (SERA) is developed to learn from concept drift and class imbalance [21]. Rather than using all accumulated minority class data, SERA judiciously selects instances from the accumulated minority data.

SERA, whose pseudo code is shown in Fig. 3.16, requires that a post-balancing ratio, f, is specified. SERA is presented with labelled batch data with a minority class and the imbalance ratio, r, is calculated. Like uncorrelated bagging, SERA maintains an accumulation of the minority class instances that have been presented to the algorithm, which is referred to as $\mathcal{C}^{(t-1)}$. The most recent data $\mathcal{D}^{(t)}$ is split into $\mathcal{P}^{(t)}$ and $\mathcal{N}^{(t)}$ containing minority and majority class training instances, respectively. If there is only a small amount of accumulated minority data then a new classifier is trained on $\{\mathcal{D}^{(t)}, \mathcal{C}^{(t-1)}\}$. If there is a sufficient amount of data accumulated then SERA selects the minority instances in $\mathcal{C}^{(t-1)}$ that are most *similar* to the current environment. Before the similarity of old and new data is determined, the mean vector, $\boldsymbol{\mu}$, and covariance matrix, $\boldsymbol{\Sigma}$, of $\mathcal{P}^{(t)}$ is calculated. The Mahalanobis distance, Eq. (3.25), is calculated for each instance in $\mathcal{C}^{(t-1)}$ using $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$.

$$\delta_M^2(i) = (\mathbf{x}_i - \boldsymbol{\mu})^t \, \boldsymbol{\Sigma}^{-1} \, (\mathbf{x}_i - \boldsymbol{\mu}) \tag{3.25}$$

The most relevant instances are selected by sorting $\delta_M(i)$ in ascending order and selecting $(f > (t-1)r)$ instances with the lowest Mahalanobis distance for training. A single classifier or a ensemble of classifiers trained with biased form of bagging are generated. It is important to note that SERA does not retain classifiers, rather the classifier, or ensemble if *BBagging* is implemented, is discarded at each time stamp and a new classifier is generated. *BBagging*

Input: Imbalance ratio $\rightarrow r$
Training data: $\mathcal{D}^{(t)} = \{\mathbf{x}_i \in X; y_i \in \Omega = \{+1, -1\}\}$ where $i = 1, 2, \ldots, m^{(t)}$
Current testing data chunk $\mathcal{T}^{(t)} = \{\mathbf{x}_i \in X; y_i \in \Omega = \{+1, -1\}\}$
Dataset $\mathcal{C}^{(t-1)}$ accumulating all minority instances until t
f: Post-balance ratio
BaseClassifier
for $t = 1, 2, \ldots$ **do**

1. Split $\mathcal{D}^{(t)}$ into $\mathcal{P}^{(t)}$ and $\mathcal{N}^{(t)}$ containing minority and majority class training instances, respectively.

2. **if** $(f > (t-1)r)$
   $\hat{\mathcal{D}}^{(t)} = \{\mathcal{D}^{(t)}, \mathcal{C}^{(t-1)}\}$
   else

   (a) Calculate the Mahalanobis distance $(\delta_M(i))$ between distribution of $\mathcal{P}^{(t)}$ and all the instances in $\mathcal{C}^{(t-1)}$

   (b) Sort $\delta_M(i)$ and select the first $(f > (t-1)r)$. Call this $\mathcal{M}^{(t)}$

   (c) $\hat{\mathcal{D}}^{(t)} = \{\mathcal{D}^{(t)}, \mathcal{M}^{(t)}\}$

   end if

3. Build the **BaseClassifier** on $\hat{\mathcal{D}}^{(t)}$

   (a) Generate single scoring hypothesis: h_t, or

   (b) Call BBagging function to generate ensemble hypotheses: $\{h_1, h_2, \ldots, h_L\}$

end for

Fig. 3.16 : SERA pseudo code

is a bias form a bagging that increase the minority class instance weights during sampling for training instances.

SERA uses the old minority instances effectively by selecting old instances that are most similar to the current minority concept. There are a few concerns about using this algorithm, however: 1) the use of the Mahalanobis distance as a similarity measure assumes the minority class data is being drawn from Gaussian distribution; 2) similar accumulated instances do not imply relevant instances; 3) old knowledge about the majority class is discarded after each time stamp; and 4) potential issues arise when $\boldsymbol{\mu}$ is constant and drift is present in the data stream. The last point will be demonstrated in Section 5.5.2. However, even with these concerns SERA is a fairly robust algorithm and capable of being used in nonstationary environments with class imbalance.

3.4.3 MuSERA

The multiple selectively recursive approach (MuSERA), which is an extension of SERA, is designed to learn from data streams containing concept drift when classes are imbalanced [106]. Like SERA, MuSERA accumulates all minority class instances and uses the Mahalanobis distance to select minority instances that are most similar to the current minority class distribution. However, MuSERA does not maintain a single hypothesis like SERA. MuSERA includes classifiers built on data from previous time stamps. The classifiers are combined using weighted majority voting unlike other approaches [61, 21].

MuSERA, whose pseudo code is shown in Fig. 3.17, is nearly identical to SERA with a few exceptions. First, MuSERA requires that a soft-hypothesis is generated at each time stamp. The soft-hypothesis could be any classifier that produces a probability output. The pseudo error of a classifier is computed based on the most recent batch of data as shown in Eq. (3.28) where $f^k_{y_i}(\mathbf{x}_i)$ is the posterior estimate for class y_i given by the kth classifier on instance $\mathbf{x}_i$. The weight of a classifier is proportional to the error on that most recent batch of data.

$$\varepsilon_k^{(t)} = \frac{1}{m^{(t)}} \sum_{i=1}^{m^{(t)}} \left(1 - f^k_{y_i}(\mathbf{x}_i)\right)^2 \tag{3.28}$$

Input: Imbalance ratio $\rightarrow r$
Training data: $\mathcal{D}^{(t)} = \{\mathbf{x}_i \in X; y_i \in \Omega = \{+1, -1\}\}$ where $i = 1, 2, \ldots, m^{(t)}$
Current testing data chunk $\mathcal{T}^{(t)} = \{\mathbf{x}_i \in X; y_i \in \Omega = \{+1, -1\}\}$
Dataset $\mathcal{C}^{(t-1)}$ accumulating all minority instances until t
f: Post-balance ratio
BaseClassifier
for $t = 1, 2, \ldots$ **do**

1. Split $\mathcal{D}^{(t)}$ into $\mathcal{P}^{(t)}$ and $\mathcal{N}^{(t)}$ containing minority and majority class training instances, respectively.
2. **if** $(f > (t-1)r)$
   (a) $\hat{\mathcal{D}}^{(t)} = \{\mathcal{D}^{(t)}, \mathcal{C}^{(t-1)}\}$

   else

   (a) Calculate the Mahalanobis distance $(\delta_M(i))$ between distribution of $\mathcal{P}^{(t)}$ and all the instances in $\mathcal{C}^{(t-1)}$
   (b) Sort $\delta_M(i)$ and select the first $(f > (t-1)r)$. Call this $\mathcal{M}^{(t)}$
   (c) $\hat{\mathcal{D}}^{(t)} = \{\mathcal{D}^{(t)}, \mathcal{M}^{(t)}\}$

   end if
3. Build soft-typed hypothesis, h_t, using **BaseClassifier** on $\hat{\mathcal{D}}^{(t)}$ and include h_t into the ensemble
4. Apply h_k on $\mathcal{D}^{(t)}$ to derive the weight $W_k^{(t)}$ based on Eq. (3.26) and (3.27):

$$\varepsilon_k^{(t)} = \frac{1}{m^{(t)}} \sum_{i=1}^{m^{(t)}} \left(1 - f_{y_i}^k(\mathbf{x}_i)\right)^2 \tag{3.26}$$

$$W_k^{(t)} = \log \frac{1}{\varepsilon_k^{(t)}} \tag{3.27}$$

end for

Fig. 3.17 : MuSERA pseudo code

3.5 Prior Work

3.5.1 Learn^{++}.NSE

Learn^{++}.NSE is an ensemble based incremental learning algorithm for nonstationary environments [12, 107, 50, 13]. Learn^{++}.NSE uses a unique method to dynamically combine classifiers in the ensemble, which allows the ensemble to temporarily remove voting power from classifiers that are poor predictors on the current environment. Learn^{++}.NSE has been shown to provide favorable results in nonstationary environments although there are a few issues worth addressing.

Learn^{++}.NSE, whose pseudo code is shown in Fig. 3.18, is presented with batches of labelled data incrementally over time [107, 12]. The database, $\mathcal{D}^{(t)}$, contain $m^{(t)}$ instances $\mathbf{x}$ and a true class label y. The algorithm maintains a set of weights over all the instances at time stamp t such that the sum of the weights is equal to 1. If $t = 1$ then there is no ensemble yet, our knowledge about previous environments does not exist. The initial distribution of instance weights to $D^{(1)}(i) = 1/m^{(1)}$ and continue to step 3. If $t > 1$, th old ensemble, $H^{(t-1)}$, is evaluated on the most recent data and the error on the new data is computed as shown in Eq. (3.29) (step 1). Note that $[\![\cdot]\!]$ is the indicator function that evaluates to 1 if the statement within is true and 0 if the statement is false. The indicator function evaluates to 1 if an instance, $\mathbf{x}_i$, is misclassified by the ensemble.

The weights of the instances, $w^{(t)}(i)$, that have been classified correctly are decreased proportional to the error of the old classifier ensemble as shown in Eq. (3.30). The instance weights are then normalized to create a distribution, Eq. (3.31).

A new classifier, h_t, is generated using $\mathcal{D}^{(t)}$. One major difference between the formulation of Learn^{++}.NSE and other implementations of Learn^{++} is that the distribution of instance weights is no longer used to sample the data. Previous implementations of Learn^{++} use $D^{(t)}$ as a probability distribution to form a new dataset, which is then used to generate a classifier; however Learn^{++}.NSE does not use the distribution in this way. $D^{(t)}$ is used as a penalty distribution to assign error to classifiers in the ensemble. The penalty distribution assigns a higher penalty to instances misclassified by the ensemble compared to instances correctly classified by the ensemble. On the other hand, AdaBoost/Learn^{++} uses the instance weight distribution for sampling instances for training a classifier. Another major difference between Learn^{++}.NSE and Learn^{++} is that we are no longer generating

Input: Training Data $\mathcal{D}^{(t)} = \{\mathbf{x}_i^{(t)} \in X, y_c^{(t)} \in \Omega\}$ where $i = 1, \ldots, m^{(t)}, c = 1, \ldots, C$
Supervised learning algorithm, **BaseClassifier**
$D^{(1)} = w^{(1)}(i) = 1/m^{(1)}$
for $t = 1, 2, \ldots$ **do**

1. Compute ensemble error

$$E^{(t)} = \sum_{i=1}^{m^{(t)}} \frac{1}{m^{(t)}} [\![H^{(t-1)}(\mathbf{x}_i) \neq y_i]\!] \tag{3.29}$$

2. Update and normalize distribution of weights

$$w_i^{(t)} = \frac{1}{m^{(t)}} \cdot \left(E^{(t)}\right)^{1-[\![H^{(t-1)}(\mathbf{x}_i)=y_i]\!]} \tag{3.30}$$

$$D^{(t)} = w^{(t)} / \sum_{i=1}^{m^{(t)}} w_i^{(t)} \tag{3.31}$$

3. Call **BaseClassifier** with $\mathcal{D}^{(t)}$, obtain $h_t : X \to Y$
4. Evaluate all exiting classifiers on new dataset, $\mathcal{D}^{(t)}$, where $k = 1, 2, 3, \ldots, t$

$$\epsilon_k^{(t)} = \sum_{i=1}^{m^{(t)}} D^{(t)}(i) \cdot [\![h_k(\mathbf{x}_i) \neq y_i]\!] \tag{3.32}$$

if $\epsilon_k^{(t)} > 1/2$, generate new h_t **else** $\epsilon_{k<t}^{(t)} > 1/2$, set $\epsilon_{k<t}^{(t)} = 1/2$ **endif**

$$\beta_k^{(t)} = \epsilon_k^{(t)} / (1 - \epsilon_k^{(t)})$$

5. Compute a weighted sum of all normalized error for the kth classifier, h_t

$$\sigma_k^{(t)} = 1/\left(1 + \exp\left(-a(t-k-b)\right)\right) \tag{3.33}$$

$$\hat{\sigma}_k^{(t)} = \sigma_k^{(t)} / \sum_{j=0}^{t-k} \sigma_k^{(t-j)} \tag{3.34}$$

$$\hat{\beta}_k^{(t)} = \sum_{j=0}^{t-k} \hat{\sigma}_k^{(t-j)} \beta_k^{(t-j)} \tag{3.35}$$

6. Calculate classifier voting weights

$$W_k^{(t)} = \log\left(1/\hat{\beta}_k^{(t)}\right) \tag{3.36}$$

7. Obtain composite hypothesis using Eq. (3.37).

end for

Fig. 3.18 : Learn^{++}.NSE pseudo code

weak classifiers. Learn^{++}.NSE generates classifiers that are quality (strong) models of the most recent data. This allows each classifier to strongly learn the most recent environment and not just on the instances that were misclassified by the previous classifier, which is done in algorithms like AdaBoost and Learn^{++}.

All existing classifiers are then evaluated on $\mathcal{D}^{(t)}$ and a pseudo error is computed from the distribution of instance weights, Eq. (3.32). This is a pseudo error, not overall error, because the non-uniform distribution of weights is used to determine the error measure. Thus, some instances will incur more of a penalty for a misclassification, namely the classifiers that incorrectly label instances misclassified by the ensemble.

The pseudo error of the classifier is compared to 1/2. If $\epsilon_{k=t} > 1/2$ then the classifier is discarded and a new classifier is generated from $\mathcal{D}^{(t)}$ until we have a classifier that has an error less than 1/2. If $\epsilon_{k<t} > 1/2$, then the pseudo error is set to 1/2. The error is then normalized as shown in step 4 and the reason for this double standard is two-folds: (a) the most recent classifier should have an error less than 1/2, which is a random guess for a two class problem, and (b) it is possible that a classifier ($h_{k<t}$) is performing poorly now may become a strong predictor at a later point in time. The condition of having a classifiers error greater than 1/2 may be difficult as the number of classes increases, however for this work we have kept the threshold at 1/2.

A normalized logistic sigmoid, as shown in the Eq. (3.33) & (3.34), is applied to the current and old errors of a classifier. This causes $\hat{\beta}_k^{(t)}$ to be influenced the most by the most recent time step errors. The slope and cut-off of the logistic sigmoid can be controlled using the a and b parameters. The cut-off essentially controls the number of classifier errors that will be weighted heavily (i.e., when the sigmoid function value is near 1). The slope of the sigmoid determines the rate at which the weights decay after the cut-off. We refer to this method of weighting as a time-adjusted weighted error.

The voting weight for each classifier is proportional to the weighted sum of a classifier's previous errors, Eq. (3.36). The voting weight for a classifier is updated every time labelled data are presented to the algorithm.

The final ensemble decision is a weighted majority vote, Eq. (3.37). The WMV with weights determined using a time-adjust sum of classifier errors allows classifiers that are performing well in recent time to have a larger weight than classifiers recently performing

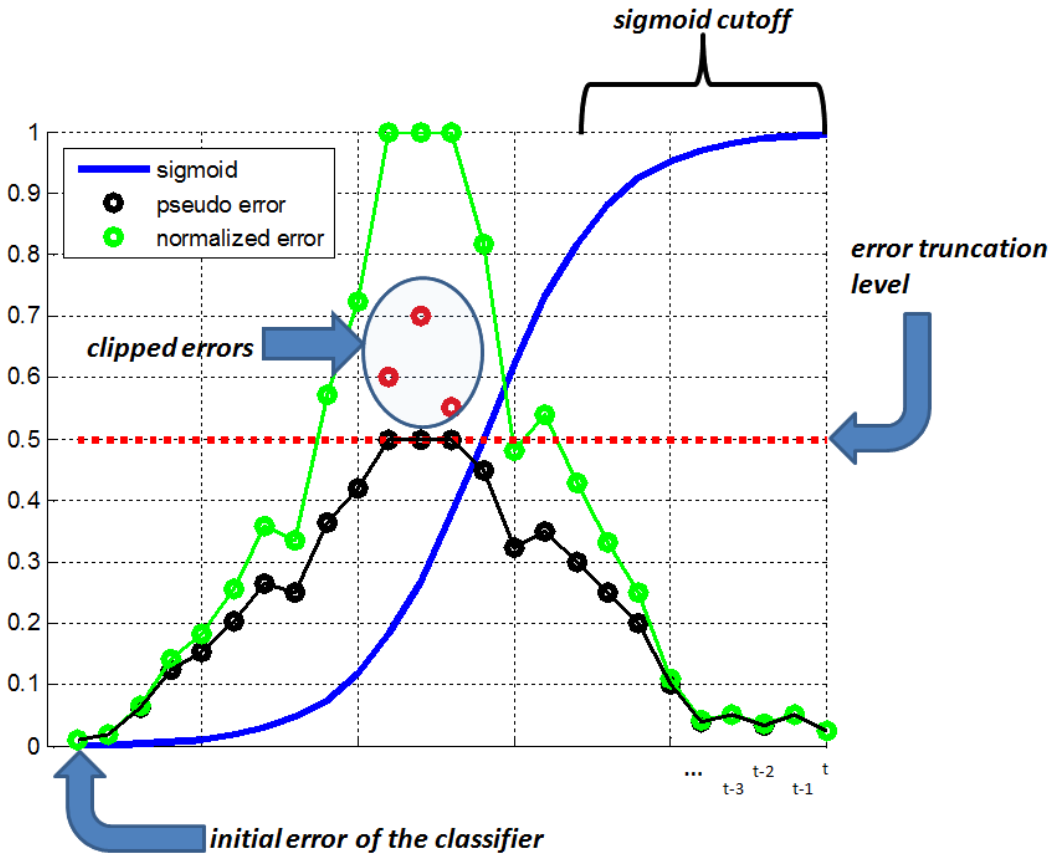


Fig. 3.19 : The more recent errors of a classifier are weighted heavier than errors ($\beta_k^{(t)}$) that have been committed a long time ago as indicated by the magnitude of the logistic function through time ($\sigma_k^{(t)}$)

poorly. The ensemble decision can be called at any time for a decision on any given instance.

$$H^{(t)}(\mathbf{x}) = \arg\max_c \sum_k^t W_k^{(t)} [\![h_k(\mathbf{x}) = c]\!] \tag{3.37}$$

3.6 Summary

So far, a variety of algorithms for concept drift, class imbalance and a combination the prior two problems have been presented for learning data over time. Many of them ensemble based, for learning concept drift over time or increasing the recall of a minority class. However, there are only a few algorithms that address concept drift and class imbalance explicitly. In the next chapter, a set of incremental learning algorithms for concept drift and class imbalance are presented along with a concept drift algorithm for transductive

learning and a drift detection method.

Chapter 4

Approach

We present several approaches for learning that are all related through concept drift and incremental learning. There are three primary sets algorithms presented in this thesis, namely:

- **Algorithms for Concept Drift and Class Imbalance**: Learn^{++}.CDS and Learn^{++}.NIE incrementally learn from concept drift with unbalanced data. These algorithms are presented in Sections 4.1 and 4.2.
- **Using Unlabelled Data in the Presence of Concept Drift**: A weight estimation algorithm (WEA) is presented for learning in the presence of concept drift when there is a large amount of unlabelled data present. WEA is presented in Section 4.3.
- **A Drift Detection Algorithm**: A drift detection algorithm for incremental learning is presented in Section 4.4.

4.1 Learn^{++}.CDS

4.1.1 Motivation for Learn^{++}.CDS

Learn^{++}.NSE, as described in Section 3.5, has been shown to work well in nonstationary environments [12, 50, 107]; however, Learn^{++}.NSE does not work well when there is class imbalance as it has not been designed to tackle this problem. On the other hand SMOTE – a popular sampling method for class imbalance – has been shown to work quite well at boosting the recall of a minority class. Therefore, an obvious solution is to combine the two algorithms, similar to SMOTEBoost. Learn^{++}.NSE is used to learn in nonstationary environments while SMOTE is used to oversample the minority class to reduce the imbalance ratio and enrich the minority class feature space.

4.1.2 Algorithm Description

Learn^{++}.CDS, whose pseudo code is shown in Fig. 4.1, integrates the incremental concept drift learning algorithm Learn^{++}.NSE with SMOTE [98, 108]. The initial motivation behind integrating these two algorithms is that each one works well for the problem for which they were designed to handle. The primary free parameters of Learn^{++}.CDS are the base classifier, the sigmoid weighting parameters and the SMOTE parameters. The base classifier can be any supervised learning algorithm that does not retain old data such as a decision tree (C4.5 or CART), MLPNN, or naïve Bayes classifier.

Learn^{++}.CDS is provided with data incrementally at time step t with dataset $\mathcal{D}^{(t)}$ that come from the joint probability distribution $p_t(\mathbf{x}, \omega_j)$, which may be different than $p_{t-t_0}(\mathbf{x}, \omega_j)$ where t_0 is any arbitrary earlier time stamp. Learn^{++}.CDS maintains a set of distribution weights over the instances. The weight of the instances, that are misclassified by the previous ensemble, $H^{(t-1)}(\mathbf{x})$ are increased and the distribution is renormalized (Step 1 & 2). The distribution of instance weights is not used for re-sampling as performed with AdaBoost [32, 33]. Rather, the distribution is used to determine the penalty, also known as the pseudo error, of a classifier for incorrectly classifying a particular instance in the labelled training data.

SMOTE is then called to re-balance the minority class data (denoted by $\mathcal{S}^{(t)}$). A new classifier is trained on the union of $\mathcal{S}^{(t)}$ and $\mathcal{D}^{(t)}$ (Step 3). Learn^{++}.CDS does not *sample* from $\mathcal{D}^{(t)}$ using $D^{(t)}$ because we are trying to learn the most recent environment and the examples unlearned by the ensemble. The new classifier and the all previously generated classifiers (denoted by $k = 1, \ldots, t$) are evaluated on the new dataset to obtain their pseudo errors on the new environment (Step 4). The distribution of instance weights, $D^{(t)}$, is used to determine the pseudo error. As explored in Learn^{++}.NSE, if the pseudo error for the new classifier is greater than 1/2, it is discarded and a new one is created; however if an older classifier's error is greater than 1/2 the error is set to 1/2.

A logistic sigmoid is then applied to errors of each classifier across all time steps (Step 5). Using the same mindset as Learn^{++}.NSE, this style of weighting will reward classifiers that are currently performing well on the most recent environments, even if such classifiers may have been generated a long time ago. The classifiers voting weights are computed at each time step proportional to the weighted sum of their pseudo errors over time (Step 6).

Input: Training Data $\mathcal{D}^{(t)} = \{\mathbf{x}_i^{(t)} \in X, y_i^{(t)} \in \Omega\}$ where $i = 1, \ldots, m^{(t)}, \Omega = 1, \ldots, C$
Supervised learning algorithm, **BaseClassifier**
SMOTE Parameters: k, N
$D^{(1)} = w^{(1)}(i) = 1/m^{(1)}$
for $t = 1, 2, \ldots$ **do**

1. Compute ensemble error

$$E^{(t)} = \sum_{i=1}^{m^{(t)}} \frac{1}{m^{(t)}} [\![H^{(t-1)}(\mathbf{x}_i) \neq y_i]\!] \tag{4.1}$$

2. Update and normalize distribution of weights

$$w_i^{(t)} = \frac{1}{m^{(t)}} \cdot \left(E^{(t)}\right)^{1-[\![H^{(t-1)}(\mathbf{x}_i)=y_i]\!]} \tag{4.2}$$

$$D^{(t)} = w^{(t)} / \sum_{i=1}^{m^{(t)}} w_i^{(t)} \tag{4.3}$$

3. Call SMOTE on minority class to create $\mathcal{S}^{(t)}$. Call **BaseClassifier** with $\mathcal{S}^{(t)}$ and $\mathcal{D}^{(t)}$, obtain $h_t : X \to Y$
4. Evaluate all exiting classifiers on new dataset, $\mathcal{D}^{(t)}$, where $k = 1, 2, 3, \ldots, t$

$$\epsilon_k^{(t)} = \sum_{i=1}^{m^{(t)}} D^{(t)}(i) \cdot [\![h_k(\mathbf{x}_i) \neq y_i]\!] \tag{4.4}$$

if $\epsilon_k^{(t)} > 1/2$, generate new h_t **else** $\epsilon_{k<t}^{(t)} > 1/2$, set $\epsilon_{k<t}^{(t)} = 1/2$ **endif**

$$\beta_k^{(t)} = \epsilon_k^{(t)} / (1 - \epsilon_k^{(t)})$$

5. Compute a weighted sum of all normalized error for the kth classifier, h_t

$$\sigma_k^{(t)} = 1/\left(1 + \exp\left(-a(t-k-b)\right)\right) \tag{4.5}$$

$$\hat{\sigma}_k^{(t)} = \sigma_k^{(t)} / \sum_{j=0}^{t-k} \sigma_k^{(t-j)} \tag{4.6}$$

$$\hat{\beta}_k^{(t)} = \sum_{j=0}^{t-k} \hat{\sigma}_k^{(t-j)} \beta_k^{(t-j)} \tag{4.7}$$

6. Calculate classifier voting weights

$$W_k^{(t)} = \log\left(1/\hat{\beta}_k^{(t)}\right) \tag{4.8}$$

7. Obtain composite hypothesis

end for

Fig. 4.1 : Learn^{++}.CDS pseudo code

The final ensemble decision is obtained using weighted majority voting (Step 7).

4.2 Learn^{++}.NIE

4.2.1 Motivation for Learn^{++}.NIE

Our primary goal is to develop an ensemble of classifiers-based model that can recognize instances of both the minority and the majority class, whose distributions may be experiencing concept drift. Since Learn^{++}.NSE has been shown to work well under various drift conditions, it was chosen as the foundation for its successor, Learn^{++}.NIE (*Nonstationary and Imbalanced Environments*). However, unlike Learn^{++}.CDS, the approach in this section does not rely on the generation of synthetic data. Learn^{++}.NIE uses measures, which would usually be used to evaluate the effectiveness of imbalanced data algorithm, to determine a voting weight for a classifier hypothesis.

4.2.2 Algorithm Description

Learn^{++}.NIE has many similarities to Learn^{++}.NSE. Both algorithms are incremental learning ensembles, employ age-adjusted weighting mechanism and compute the final hypothesis using a weighted majority vote [50, 109]. Significant differences exist between the algorithms, specifically in the method of generating new classifiers and the measure used to determine the voting weight of a classifier. Learn^{++}.NSE employs a pseudo error where a distribution of instance weights determines the penalty for a classifier misclassifying a particular instance. However, there is no method integrated into Learn^{++}.NSE that looks at the error of a minority class (or any specific class for that matter). Therefore, Learn^{++}.NIE employs a method of weighting that rewards a classifier that performs well across *all* classes. The idea behind using different measures for weighting classifiers is two-fold*: (1) can a measure other than error be selected that is capable of tracking drifting concepts, (2) to what extent can the recall of a minority class be increased while preserving a strong performance in nonstationary environments?

The pseudo code for Learn^{++}.NIE is shown in Fig. 4.2. The algorithm begins by calling a variation of bagging to generate new classifiers or sub-ensembles (Step 1). The classifiers in the `BaggingVariation` routine are generated by training classifiers on all the minority

*See [11, 33, 12, 110, 28, 71] for algorithms using voting weights derived from error

Input: Training Data $\mathcal{D}^{(t)} = \{\mathbf{x}_i^{(t)} \in X, y_i^{(t)} \in \Omega\}$ where $i = 1, \ldots, m^{(t)}, \Omega = 1, \ldots, C$
Supervised learning algorithm, **BaseClassifier**
Number of Classifiers in sub-ensemble, K
Error weight η, $0 \leq \eta \leq 1$
for $t = 1, 2, \ldots$ **do**

1. $H_t =$ `BaggingVariation`(**BaseClassifier**, K, $\mathcal{D}^{(t)}$)
2. Evaluate all sub-ensembles on new dataset , $\mathcal{D}^{(t)}$, where $k = 1, 2, 3, ..., t$

$$G_k^{(t)} = SMV\left(H_k, \mathcal{D}^{(t)}\right) \tag{4.9}$$

3. Compute $\epsilon_k^{(t)}$ using weighted average, F-measure or G-mean.
4. Compute a weighted sum of all $\epsilon_k^{(t)}$ for each sub-ensemble where $k = 1, 2, ..., t$

$$\sigma_k^{(t)} = 1/\left(1 + \exp\left(-a(t-k-b)\right)\right) \tag{4.10}$$

$$\hat{\sigma}_k^{(t)} = \sigma_k^{(t)} / \sum_{j=0}^{t-k} \sigma_k^{(t-j)} \tag{4.11}$$

$$\hat{\beta}_k^{(t)} = \sum_{j=0}^{t-k} \hat{\sigma}_k^{(t-j)} \beta_k^{(t-j)} \tag{4.12}$$

5. Calculate classifier voting weights

$$W_k^{(t)} = \log\left(1/\hat{\beta}_k^{(t)}\right) \tag{4.13}$$

6. Obtain composite hypothesis

$$H^{(t)}\left(\mathbf{x}_i\right) = \arg\max_c \sum_k W_k^{(t)} [\![h_k\left(\mathbf{x}_i\right) = c]\!] \tag{4.14}$$

end for

Fig. 4.2 : Learn^{++}.NIE pseudo code

class data in $\mathcal{D}^{(t)}$ and a randomly sampled subset of the majority class. The strategy here is to have each classifier generated exposed to all the minority class information and use the combination of the classifiers to learn the majority class information. The classifiers in the sub-ensemble are combined using a simple majority vote. The sub-ensemble is denoted by H_t is the pseudo code.

All existing sub-ensembles (H_k) are evaluated on $\mathcal{D}^{(t)}$ to produce a set of labels where

$G_k^{(t)}$ are the predicted labels of the kth classifier at time stamp t on $\mathcal{D}^{(t)}$. Learn++.NIE then computes an error measure, $\epsilon_k^{(t)}$, to be used in determining the weight of the sub-ensemble. As mentioned above, this measure is not based on the raw classification accuracy, but designed to accommodate the imbalanced nature of the data. Specifically, Learn^{++}.NIE uses one of three carefully selected measures, the first of which is the weighted average ($wavg$) for boosting recall and tracking drifting concepts. The $wavg$ measure is determined by computing the weighted average of recall on the majority class ($p_{k,n}^{(t)}$) and minority class ($p_{k,p}^{(t)}$). The weighted error, $\epsilon_k^{(t)}$, is computed using Eq. (4.15) from the performances of the majority and minority classes. The term η controls the weight given to a particular class error and is bound between zero and one ($\eta \in [0,1]$). Therefore, one can control the penalty incurred for the error of a class rather than penalizing for the misclassification of a particular instance.

$$\epsilon_k^{(t)} = \eta\left(1 - p_{k,p}^{(t)}\right) + (1-\eta)\left(1 - p_{k,n}^{(t)}\right) \tag{4.15}$$

$$= \eta\epsilon_{k,p}^{(t)} + (1-\eta)\,\epsilon_{k,n}^{(t)} \tag{4.16}$$

Selecting $\eta = 1$ (or $\eta = 0$) penalizes a classifier for error on the minority (or majority) class only. Thus, one would expect the ensemble to have a very high minority (or majority) class performance, but poor overall accuracy because a sub-ensemble is not penalized for not learning the majority (or minority) class. Selecting $\eta = 0.5$ typically provides a good balance between minority class performance and the overall accuracy.

There are additional error measures that can be used to accommodate imbalanced data. Two such measures are the geometric mean (G-mean) and F-measure, which are commonly used in assessing classifier performance on imbalanced data. In using the former, we acknowledge that sub-ensembles that perform well across all classes should have a larger geometric mean than classifiers that perform poorly on any given class. For example, a sub-ensemble may have a relatively high overall classification performance, but end up with a poor G-mean, if it performs poorly on a minority class. In other words, geometric mean will indicate if a classifier is performing well across all classes or just a majority class. When the G-mean measure is used, $\epsilon_k^{(t)}$ is computed as follows:

$$\epsilon_k^{(t)} = 1 - \prod_{c=1}^{C} \left(p_{k,c}^{(t)} \right)^{1/C} \tag{4.17}$$

Eq. (4.17) now becomes the sub-ensemble weighting measure that can replace the weighted error measure in Learn^{++}.NIE and may proceed with the rest of the steps to compute the classifier voting weights the same way that Learn^{++}.CDS computes the classifier voting weights.

The final experimental weighted measure is proportional to the F-measure given by:

$$\epsilon_k^{(t)} = 1 - 2\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = 1 - F_1 \tag{4.18}$$

where F_1 is the F_1-score or F-measure. The F-measure accounts for both precision and recall, which will be discussed more in the next chapter. The strategy behind using measures like the F-measure, G-mean and weighted average is to allow the algorithm to weigh classifiers on how well they are performing across all classes, majority and minority. Thus, if a sub-ensemble is consistently performing poorly on a minority (or majority) class, then the final voting weight, $W_k^{(t)}$, will reflect this sub-par performance (Step 5). Once the sub-ensemble errors $\epsilon_k^{(t)}$ are determined, the rest of Learn++.NIE follows as Learn^{++}.CDS, with individual classifiers h_t of Learn++.CDS replaced with the corresponding sub-ensemble H_t. The composite hypothesis is form via a weighted majority vote using the subensemble decisions as shown in Eq. (4.14).

4.3 Weight Estimation Algorithm for Learning in Nonstationary Environments

All learning scenarios described thus far have been supervised in nature, meaning that data are presented in pairs, $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i$ are the instances and y_i are the class labels. Obtaining labelled instances is generally an expensive task because instances, particularly in real-world scenarios, are label by a human. However, if a significant amount of unlabelled data is presented for testing, sampled from a different source than the training sources, than unlabelled data may provide information to update classifier voting weights for the purposes of labelling the unlabelled data. In this section, a weight estimation algorithm for learning concept drift is presented, which is based on a transductive learning framework for

updating classifier voting weights on unlabelled field data.

4.3.1 Determining Classifier Voting Weights

Consider the problem of estimating a set of Bayes-optimal discriminate functions based on the outputs of the classifiers in an ensemble [52]. The discriminate function can be formed by a weighted combination of the classifier outputs. However, we wish to determine a non-heuristic, optimal set of weights for member classifiers. The Bayes-optimal discriminate function for class ω_j on instance $\mathbf{x}$ is given by Eq. (4.19), where $\mathbf{h}$ is a vector of labels predicted by T classifiers in the ensemble. Assuming all classifiers in the ensemble provide independent outputs, the naïve Bayes rule may be applied on the on the conditional probability to formulate Eq. (4.20), which can be rewritten as in Eq. (4.21).

$$
\begin{aligned}
g_j(\mathbf{x}) &= \log\left\{P(\omega_j)p\left(\mathbf{h}|\omega_j\right)\right\} && (4.19)\\
&= \log\left\{P(\omega_j)\prod_{k=1}^{t}p\left(h_k|\omega_j\right)\right\} && (4.20)\\
&= \log P(\omega_j) + \sum_{k:h_k=\omega_j}\log\frac{1-\varepsilon_k}{\varepsilon_k} + \sum_{k=1}^{t}\varepsilon_k && (4.21)
\end{aligned}
$$

where

$$
\varepsilon_k = 1 - p(h_k|\omega_j) \tag{4.22}
$$

is the error of the kth classifier hypothesis h_k in predicting the true label ω_j.

The last term in Eq. (4.21) has no dependence on ω_j, and therefore it can be dropped from the discriminate. Eq. 4.21 may be further reduced by dropping the last term if all classes have equal prior probabilities. Thus, the weights may be given by Eq. (4.23).

$$
W_k \propto \log\frac{1-\varepsilon_k}{\varepsilon_k} \tag{4.23}
$$

This equation states the Bayes-optimal weight for each classifier can be computed if the classifier's error on a static dataset can be estimated. The error of the kth classifier, ε_k, is typically estimated by computing the error on a validation dataset. In order to compute the

weights, a significant amount of labelled data is needed to reliably estimate the error. In the current concept drift formulation, an algorithm has access to unlabelled data, possibly drawn from a different distribution (i.e., different from that generated the training data) and the data can be used to aid in assigning a weight to each classifier. Let S_k represent the source distribution whose data were used to generate a classifier h_k at time stamp k where $k = 1, 2, \ldots, t$ and t is the most recent time stamp for which training data is available. If the field data (i.e., test data) is sampled from a different source distribution, $S_{t+t_\beta} \neq S_t$, $W_k^{(t)}$ computed based on the errors of the classifiers on trained on S_t will not be optimal. Rather, the weights $W_k^{(t+t_\beta)}$ (as shown in Eq. (4.24)) are desired, however, labelled data at time stamp $t + t_\beta$ is needed to estimate $\varepsilon_k^{(t+t_\beta)}$. The problem is, then, how to estimate $W_k^{(t+t_\beta)}$ when (field) data are not yet labelled.

$$W_k^{(t+t_\beta)} \propto \log \frac{1 - \varepsilon_k^{(t+t_\beta)}}{\varepsilon_k^{(t+t_\beta)}} \tag{4.24}$$

A clustering algorithm is used to represent the distribution of the unlabelled data drawn from a source distribution S_{t+t_β}. Specifically, a mixture of Gaussians is generated with K components on the unlabelled data, and a separate mixture of Gaussians for each class in the labelled data set sampled from source S_t. Assuming the drift in the data is gradual or incremental, one can determine a correlation between the two sets of mixture models. This processed is described in detail in the next section.

4.3.2 Weight Estimation Algorithm

The *weight estimation algorithm* (WEA) is a batch based incremental learning algorithm for concept drift that receives batches of labelled and unlabelled data at each time stamp. However, in line with the typical streaming data applications, the unlabelled data are not available at the time of training, but only becomes available at the time of testing. Specifically, at time t a a batch of labelled data is obtained to train a new classifier. Upon completion of the training, there is unlabelled test data, which is used to adjust classifier voting weights. Furthermore, again in line the assumption of possibly continuous drift, the source distributions that generated the training data and the test data may be different, and access to the labels of the field data are not available until after they are evaluated by the ensemble. Using weights determined from the labelled data (per Eq. (4.23)) may result

in suboptimal performance if there is a significant amount of bias between the distributions of the labelled and unlabelled data sources. Thus, the goal is to use the unlabelled data and a set of mixtures models formed on the previous data sources to estimate the weights for all classifiers in the ensemble before classifying the unlabelled field data. After the field data have been classified, the true labels of the instances are obtained so that the performance of the ensemble can be computed.

WEA assumes that there is limited concept drift in the incremental learning scenario. By limited concept drift the assumption is that the drift is not completely random; rather there is a structure to the drift. Think of the underlying source generating the data evolving with time and not randomly selecting new sources to generate data. Thus, the sources generating the labelled and unlabelled data presented at a future time stamp are not radically different. Instead, the learning scenario experiences a gradual or incremental drift. After all, a source changing randomly between time stamps cannot be learned. Therefore, we make the limited drift assumption:

- **Limited Drift Assumption**: The Mahalanobis distance between a known (labelled) component and its future position (unlabelled at the time of testing) must be less than the Mahalanobis distance between the known component and every other future (unlabelled) component from a different class. This assumption requires that the drift of mixture components is not radical.

To describe this assumption more formally, let $\mathcal{M}_c^{(t)}(i)$ be a Gaussian mixture model at time stamp t for class c, which contains K_c components and $i, j, k = 1, \ldots, K_c$. Assuming Gaussian mixture models (GMM) are used in WEA, the limited drift assumption is given by Eq. (4.25), which states that the Mahalanobis distance between the ith component of the cth class in $\mathcal{M}_c^{(t)}$ and the jth component of the cth class in $\mathcal{M}_c^{(t+1)}$ must be less than the distance between ith component of the cth class in $\mathcal{M}_c^{(t)}$ and kth components of the cth class in $\mathcal{M}_{c'}^{(t+1)}$ where $c \neq c'$. The limited drift assumption requires that Eq. (4.25) should hold for all k and all $c \neq c'$.

$$\delta_M\left(\mathcal{M}_c^{(t)}(i), \mathcal{M}_c^{(t+1)}(j)\right) < \delta_M\left(\mathcal{M}_c^{(t)}(i), \mathcal{M}_{c'}^{(t+1)}(k)\right) \tag{4.25}$$

WEA, whose pseudo code is provided in Fig. 4.3, works iteratively as new data become

Input: Labeled training data $\mathcal{D}^{(t)} = \{\mathbf{x}_i \in \mathcal{X}; y_i \in \Omega\}$ where $i = 1, \ldots, m^{(t)}$
Unlabeled field data $\mathcal{B}^{(t)} = \{\mathbf{x}_j \in \hat{X}\}$ where $j = 1, \ldots, n^{(t)}$
K_c: number of centers in the cth GMM
$q^(t))$: number instances generated to estimate the classifier error
BaseClassifier learning algorithm

for t=1,2,... **do**

1. Call BaseClassifier on $\mathcal{D}^{(t)}$ to generate $h_t : \mathcal{X} \rightarrow \Omega$

2. Generate a GMM with K_c components for each class in $\mathcal{D}^{(t)}$. Refer to these mixture models as $\mathcal{M}_c^{(t)}$.

3. Generate a GMM with K centers from unlabeled $\mathcal{B}^{(t)}$, where $K = \sum K_c$. Refer to this mixture model as $\mathcal{N}^{(t)}$.

4. Compute Mahalanobis distance between each component of the unlabeled mixture $\mathcal{N}^{(t)}$ and all components of mixtures in $\mathcal{M}_c^{(t)}$. Assign each component in $\mathcal{N}^{(t)}$ with the label of the closest component in $\mathcal{M}_c^{(t)}$. Refer to this mixture as $\mathcal{N}_c^{(t)}$.

5. Generate synthetic data from $\mathcal{N}_c^{(t)}$ and compute the error for each classifier on synthetic data

$$\hat{\varepsilon}_k^{(t)} = \frac{1}{q^{(t)}} \sum_{l=1}^{q^{(t)}} [\![h_k(\hat{\mathbf{x}}_l) \neq \hat{y}_l]\!] \tag{4.26}$$

where $q^{(t)}$ is the number of synthetic instances generated and $k = 1, 2, \ldots, t$
if($\hat{\varepsilon}_k^{(t)} > 1/2$) **then**
$\hat{\varepsilon}_k^{(t)} = 1/2$
end if

6. Compute classifier voting weights for $\mathcal{B}^{(t)}$

$$W_k^{(t)} \propto \log \frac{1 - \hat{\varepsilon}_k^{(t)}}{\hat{\varepsilon}_k^{(t)}} \tag{4.27}$$

7. Classify field data using a weighted majority vote

$$H^{(t)}(\mathbf{x}_j) = \arg\max_{c \in \Omega} \sum_{k=1}^{t} W_k^{(t)} [\![h_k(\mathbf{x}_j) = c]\!] \tag{4.28}$$

end for

Fig. 4.3 : Weight Estimation Algorithm (WEA) pseudo code

available, as follows: At time t WEA is presented with a batch of labelled training data $\mathcal{D}^{(t)}$ and a batch of unlabelled field data $\mathcal{B}^{(t)}$. The source distribution that generated the data in $\mathcal{B}^{(t)}$ may be different from that of $\mathcal{D}^{(t)}$ due to concept drift in the data. A new classifier is generated using a supervised base classifier algorithm on $\mathcal{D}^{(t)}$. Because this is an incremental learning algorithm, only the current training data may be used for training at any time, that is, prior data are considered unavailable. The base classifier is not a weak learning algorithm as in AdaBoost and many other ensemble based approaches. Rather, a classifier is generated that is a good predictor on the most recent environment. Next, a Gaussian mixture model (GMM) is generated for each class c (where c is any arbitrary class), with K_c centers. K_c should be chosen – possibly based on prior knowledge or experience - that the GMM will be able to reasonably approximate the underlying data distribution. The GMM for class c, obtained from the labelled data at time t is referred to as $\mathcal{M}_c^{(t)}$. $\mathcal{M}_c^{(t)}$, for all classes, consists of K components where K_c of the components represent class c. Another Gaussian mixture model is generated from the unlabelled data in $\mathcal{B}^{(t)}$ when such data arrive. A GMM is generated with $\mathcal{B}^{(t)}$ with K components where K is given by Eq. (4.29).

$$K = \sum_{c \in \Omega} K_c \tag{4.29}$$

where the set Ω consists of all classes in the incremental learning problem. This mixture model obtained from the unlabelled data is referred to as $\mathcal{N}^{(t)}$. Thus, $\mathcal{N}^{(t)}$ is the model of the data sampled from an unknown source, which is an evolution of the previous model, $\mathcal{M}_c^{(t)}$. Since there are no labels for the components in $\mathcal{B}^{(t)}$, WEA seeks to determine where the mixtures in $\mathcal{M}_c^{(t)}$ have drifted to in $\mathcal{N}^{(t)}$ by measuring the similarity between the components in both models. To do this the Mahalanobis distances are computed between a component in $\mathcal{N}^{(t)}$ and all components in $\mathcal{M}_c^{(t)}$ (refer to Eq. (4.30)). The component in $\mathcal{N}^{(t)}$ is assigned to the label of the mixture in $\mathcal{M}_c^{(t)}$ with the smallest Mahalanobis distance. The covariance in the Eq. 4.30 is computed as the average covariance of the two components.

$$\delta_M^2 = (\boldsymbol{\mu}_i - \boldsymbol{\mu}_k)^T \left(\frac{\boldsymbol{\Sigma}_i + \boldsymbol{\Sigma}_j}{2} \right)^{-1} (\boldsymbol{\mu}_i - \boldsymbol{\mu}_j) \tag{4.30}$$

An unknown mixture is assigned the class label of the known mixture with the smallest

Mahalanobis distance. After each component in $\mathcal{N}^{(t)}$ has been associated with a class label, synthetic labelled data is sampled from $\mathcal{N}_c^{(t)}$ ($\mathcal{N}^{(t)}$ with components labelled). Classifiers with the synthetic data to get an estimate of what a classifier's error are on $\mathcal{B}^{(t)}$ using Eq. (4.31). The weights of the classifiers are then computed proportional to the estimated average error on $\mathcal{B}^{(t)}$ (refer to Eq. (4.31)). Finally, the unlabelled data is classified using a majority vote, where the final class labels are determined.

$$\hat{\varepsilon}_k^{(t)} = \frac{1}{q^{(t)}} \sum_{l=1}^{q^{(t)}} [\![h_k(\hat{\mathbf{x}}_l) \neq \hat{y}_l]\!] \tag{4.31}$$

The expectation in this process is that the error of the classifiers on the synthetic data will reasonably estimate the error on $\mathcal{B}^{(t)}$, if the model of $\mathcal{M}_c^{(t)}$ accurately represents the data in $\mathcal{D}^{(t)}$ and the assumptions are held. Thus, the weights computed by WEA should estimate the weights in Eq. (4.24).

4.3.2.1 Expectation Maximization for Mixture of Gaussians

This section describes the process of generating mixture model using Expectation Maximization. A Gaussian Mixture Model (GMM) is used to create the models used within the WEA algorithm. The GMM is an unsupervised learning algorithm (no training labels required). The *Expectation Maximization* (EM) algorithm is a common method for finding the maximum likelihood solutions with *latent*, or *hidden*, variables. GMMs contain Gaussian components that are combined using mixing coefficients (π_k). Let $k = 1, \ldots, K$ where K is the number of components in a GMM and $\mathbf{x}_n$ be the nth instance in a dataset. Each instance $\mathbf{x}_n$ has a random binary $1xK$ dimensional vector, $\mathbf{z}$, associated with it where $\sum_k z_k = 1$ and $z_k \in \{0, 1\}$. Using Bayes theorem for a given instance we have:

$$\gamma(z_k) = p(z_k = 1|\mathbf{x}) = \frac{p(z_k = 1)p(\mathbf{x}|z_k = 1)}{\sum_{j=1}^{K} p(z_j = 1)p(\mathbf{x}|z_j = 1)} \tag{4.38}$$

$$= \frac{\pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} \tag{4.39}$$

where $\gamma(z_k)$ becomes the responsibility that component k has for explaining $\mathbf{x}$.

Input: Unlabelled training data $\mathbf{x}_n \in X$
Number of centers, k
Initialize the means $\boldsymbol{\mu}_k$, covariances $\boldsymbol{\Sigma}_k$ and mixing coefficients π_k and evaluate the initial value of the log likelihood.

while log-likelihood has not converged **do**

1. **E-Step**: Evaluate the responsibilities using the current parameter values

$$\gamma\left(z_{nk}\right) = \frac{\pi_k \mathcal{N}\left(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\right)}{\sum_{j=1}^{K} \pi_j \mathcal{N}\left(\mathbf{x}_n | \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j\right)} \tag{4.32}$$

2. **M-Step**: Re-estimate the parameters using the current responsibilities.

$$\boldsymbol{\mu}_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma\left(z_{nk}\right) \mathbf{x}_n \tag{4.33}$$

$$\boldsymbol{\Sigma}_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma\left(z_{nk}\right) \left(\mathbf{x}_n - \boldsymbol{\mu}_k^{new}\right) \left(\mathbf{x}_n - \boldsymbol{\mu}_k^{new}\right)^T \tag{4.34}$$

$$\pi_k^{new} = \frac{N_k}{N} \tag{4.35}$$

where

$$N_k = \sum_{n=1}^{N} \gamma\left(z_{nk}\right) \tag{4.36}$$

3. Evaluate the log likelihood and check for the convergence of the parameters or the log likelihood. If the convergence criterion is not met, continue, else abort loop.

$$\log p\left(\mathbf{X} | \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}\right) = \sum_{n=1}^{N} \log \left(\sum_{k=1}^{K} \pi_k \mathcal{N}\left(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\right) \right) \tag{4.37}$$

end while

Fig. 4.4 : Expectation Maximization for Gaussian Mixtures

The implementation of the EM algorithm is shown in Fig. 4.4 [4]. The primary steps to the algorithm are the Expectation Step (E-Step) and the Maximization Step (M-Step). The EM algorithms assumes that a known number of mixtures in the model (K) is provided and the initialization of the mean vector ($\boldsymbol{\mu}_k$), covariance matrix ($\boldsymbol{\Sigma}_k$), and a set of mixture coefficients π_k. A few iterations of K-means is used to initialize $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$. The E-step begins by computing $\gamma(z_{nk})$, which is the the conditional probability of a latent variable given a data instance ($\gamma(z_{nk}) := p(z_k|\mathbf{x}_n)$). This term is the responsibility that a mixture component k takes for *explaining* the observation $\mathbf{x}_n$. Next, the M-step re-estimates the parameters (π_k, $\boldsymbol{\mu}_k$, $\boldsymbol{\Sigma}_k$) using the current responsibilities computed during the E-step. Finally, the evaluation of the log-likelihood is computed using Eq. (4.37). A check for convergence of the parameters or the log likelihood is performed and if the convergence criterion is met, then the algorithm concludes. Otherwise EM will continue until the parameters or log-likelihood converge.

4.4 Using Distributional Divergence to Detect Change in Features

A drift detection algorithm can be very useful when an active concept drift approach is being designed. One must be able to identify drift so an action can be taken to handle the nonstationary change in distribution of the data. This section of the thesis introduces the Hellinger distance as a measure that can be applied to drift detection in an incremental learning setting. The Hellinger Distance Drift Detection Method (HDDDM) was proposed in [111]. According to four criteria suggested by Kuncheva [84], HDDDM can be categorized as follows:

1. *Data chunks*: batch based
2. *Information used*: raw features
3. *Change detection mode*: explicit
4. *Classifier-specific vs. classifier-free*: classifier free (assuming an updateable classifier)

The algorithm uses a hypothesis testing-like statistical approach to determine if there is enough evidence to suggest that the data instances are being drawn different distributions.

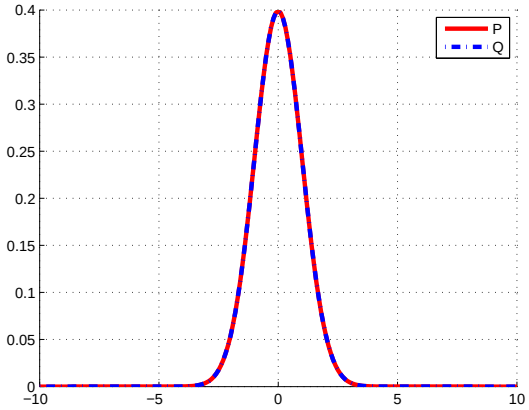
4.4.1 Motivation for HDDDM

In contrast to EDDM and other similar approaches that rely on classifier error [63], the proposed Hellinger distance drift detection method (HDDDM) is a feature based drift detection method, using the Hellinger Distance (see Eq. (4.40)) between current data distribution and a reference distribution that is updated as new data are received. The Hellinger distance is an example of f-divergence measure, similar to the Kullback-Leibler (KL) divergence. An f-divergence measures the difference between two probability distributions, which is what the Hellinger and KL-divergence are. However, unlike the KL-divergence, the Hellinger divergence is a symmetric metric.

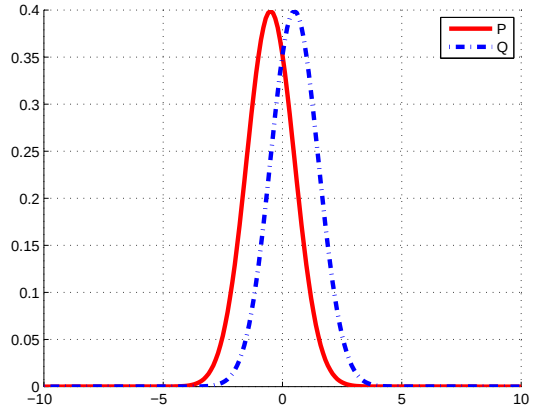
The Hellinger distance (shown in Eq. (4.40) for continuous variables) is a bounded distance measure: for two distributions with probability mass functions (or histograms representing these distributions) P and Q, the Hellinger distance is $\delta_H(P,Q) \in [0, \sqrt{2}]$. If $\delta_H(P,Q) = 0$, the two probability mass functions are completely overlapping and hence identical. If $\delta_H(P,Q) = \sqrt{2}$, the two probability mass functions are completely divergent (i.e., there is no overlap). This is demonstrated in Fig. 4.5.

$$\delta_H^2(P,Q) = \int_\Omega \left(\sqrt{\frac{dP}{d\lambda}} - \sqrt{\frac{dQ}{d\lambda}} \right)^2 d\lambda \tag{4.40}$$

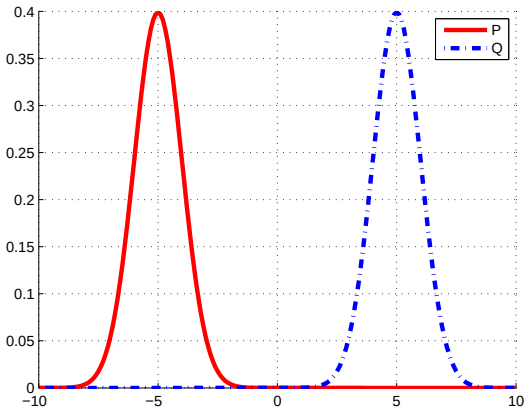
As an example, consider a two-class rotating mixture of Gaussians with class centers moving in a circular pattern (Fig. 4.6), with each distribution in the mixture corresponding to a different class label. The class means can be given by the parametric equations $\mu_1^{(t)} = [\cos(\theta_t), \sin(\theta_t)]^T$, $\mu_2^{(t)} = -\mu_1^{(t)}$, $\theta_t = \frac{2\pi c}{N} t$, with fixed class covariance matrices given as $\Sigma_1 = \Sigma_2 = 0.5 * \mathbf{I}$, where c is the number of cycles, t is the (integer valued) time stamp that iterates from zero to $N-1$, and $\mathbf{I}$ is a 2×2 identity matrix. Fig. 4.7(a) shows the evolution of the Hellinger distance computed between the datasets ($\mathcal{D}_k$) generated with respect to θ_1 and θ_k where $k = 2, \ldots, N-1$. The Hellinger distance is capable of displaying the relevance or the closeness of a new dataset ($\mathcal{D}_k$) to a baseline dataset ($\mathcal{D}_1$) as shown in Fig. 4.7(a). The Hellinger distance for the divergence of the datasets for class ω_1, class ω_2, and the entire data are plotted separately. The Hellinger distance varies as θ begins to evolve. It is observed that when θ_1 and θ_k are the same (or very similar) the Hellinger distance is small as observed at $t = \{0, 200, 400, 600\}$. This is when $\mu_1^{(t)} = \mu_1^{(1)}$ and $\mu_2^{(t)} = -\mu_1^{(t)} = -\mu_1^{(1)}$.



(a) $\delta_H(P,Q) \approx 0$



(b) $0 < \delta_H(P,Q) < \sqrt{2}$



(c) $\delta_H(P,Q) \approx \sqrt{2}$

Fig. 4.5 : Hellinger distance computed between two probability distributions P and Q.

The Hellinger distance computed for all data (entire dataset) repeats every 100 time stamps, twice as often compared to class specific distributions, which repeat every 200 time steps. This is because, as seen in Fig. 4.6, every 100 time steps, the data consists of the exact same instances at the exact same locations, but with their labels flipped, compared to the distribution at $t = 0$.

As the class centers evolve from $t = 0$ to $t = 200$, changes in Hellinger distances are observed, that follow the expected trends based on Fig. 4.6, reaching its maximum value at $t = 75$ and $t = 125$, with a slight dip at $t = 100$, for the class-specific datasets. The Hellinger distance then reduces to zero as the class distributions return to their initial state

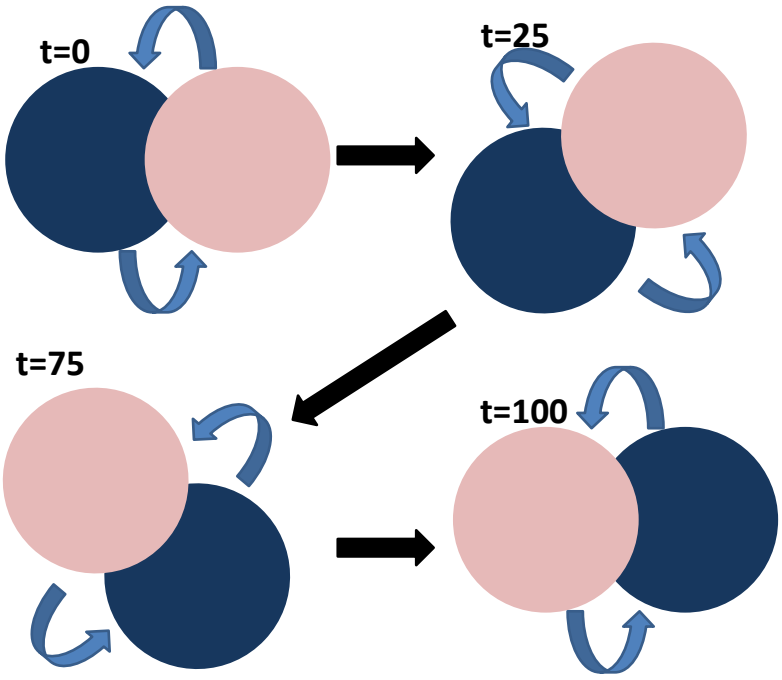
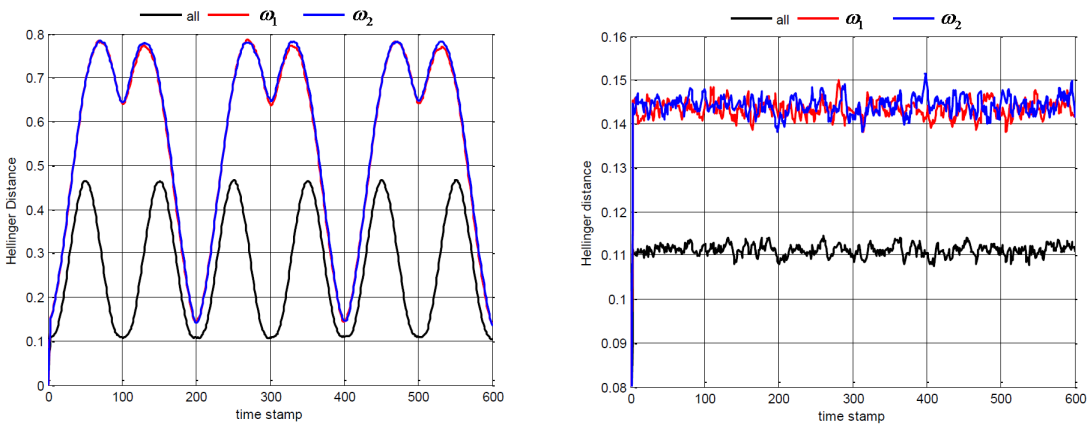


Fig. 4.6 : Evolution of a binary classification task with two drifting Gaussian distributions.



(a) Dynamic distribution

(b) Static distribution

Fig. 4.7 : Hellinger distance computed on a rotating Gaussian centers problem. The Hellinger distance is computed between datasets $\mathcal{D}_1$ and $\mathcal{D}_t$ where $t = 2, 3, \ldots, 600$ for ω_1, ω_2, and all classes. Recurring environments occur when the Hellinger distance is at a minimum.

at $t = 200$. The entire scenario then repeats two more times.

If the distributions governing the data at different time stamps is static (i.e., $\theta = constant$), then the Hellinger distance between the 1^{st} batch and subsequent batches remains constant as shown in Fig. 4.7(b). Note that a static distribution does not mean zero Hellinger distance. In fact, the Hellinger distance is non-zero due to differences in class means as well as the random nature of the data drawn for each sample. The Hellinger distance remains near constant, however, as the distribution itself does not change.

Having observed that the Hellinger distance does change between two distributions as these two distributions diverge from each other, and remain constant if they do not, this information can be used to determine when change is present in an incremental learning problem. The process of tracking the data distributions for drift is described below.

4.4.2 Assumptions of HDDDM

The proposed approach makes three assumptions: i) labelled training datasets are presented in batches to the drift detection algorithm, as the Hellinger distance is computed between two histograms (of distributions) of data. If only instance based streaming data is available, one may accumulate instances to form a batch to compute the histogram; ii) data distributions have finite support (range): $P(X \leq x) = 0$ for $x \leq T_1$ and $P(X \geq x) = 0$ for $x \geq T_2$, where $T_1 < T_2$ are finite real numbers. The number of bins in the histogram required to compute the Hellinger distance is fixed at $\lfloor N \rfloor$, where N is the number of instances at each time stamp presented to the drift detection algorithm. This can be manually adjusted if one has prior knowledge to justify otherwise. Under minimal or no prior knowledge, $\lfloor N \rfloor$ works well. iii) Finally, in order to follow a true incremental learning setting, HDDDM assumes that there is no access to old data [11, 23]. Each instance is only seen once by the algorithm.

4.4.3 Hellinger Distance Drift Detection Algorithm

The pseudo code of the proposed Hellinger distance drift detection method (HDDDM) is shown in Fig. 4.8. As mentioned above, data are assumed to arrive in batches, with dataset $\mathcal{D}_t$ becoming available at time step t. The algorithm initializes $\lambda = 1$ and $\mathcal{D}_\lambda = \mathcal{D}_1$ where λ indicates the last time stamp in where change was detected. $\mathcal{D}_1$ is established as the

first baseline reference dataset against we compare future datasets for possible drift. This baseline distribution, $\mathcal{D}_\lambda$, is updated incrementally as described below.

Input: Distribution of data, $\mathcal{D}_t$ corresponding to $p(\mathbf{x}, \omega)$ where $t = 1, 2, \ldots$
Initialize: $\mathcal{D}_\lambda = \mathcal{D}_1$
for $t = 2, 3, \ldots$ **do**

1. Generate a histogram, P, and from $\mathcal{D}_t$ and a histogram, Q, from $\mathcal{D}_\lambda$. each histogram has b bins.

2. Calculate the Hellinger distance between P and Q using Eq. (4.44). Call this $\delta_H(t)$.

3. Compute the difference in Hellinger distance

$$\varepsilon(t) = \delta_H(t) - \delta_H(t-1) \tag{4.41}$$

4. Update the adaptive threshold

$$\hat{\varepsilon} = \frac{1}{t-\lambda-1} \sum_{i=\lambda}^{t-1} |\varepsilon(i)| \tag{4.42}$$

$$\hat{\sigma} = \sqrt{\frac{\sum_{i=\lambda}^{t-1} \left(|\varepsilon(i)| - \hat{\varepsilon}\right)^2}{t-\lambda-1}} \tag{4.43}$$

Compute $\beta(t)$ using the standard deviation, Eq. (4.45), or the confidence interval method, Eq. (4.46).

5. **if** $|\varepsilon(t)| > \beta_t$ **then**
   $\lambda = t$
   Reset $\mathcal{D}_\lambda$ by setting $\mathcal{D}_\lambda = \mathcal{D}_t$
   Indicate change was detected
   else
   Update $\mathcal{D}_\lambda$ with $\mathcal{D}_t \rightarrow \mathcal{D}_\lambda = \{\mathcal{D}_\lambda, \mathcal{D}_t\}$
   end if

end for

Fig. 4.8 : Hellinger Distance Drift Detection Method

The algorithm begins by constructing histogram P from $\mathcal{D}_t$ and histogram Q from $\mathcal{D}_\lambda$ with $b = \lfloor N \rfloor$ bins, where N is the cardinality of the current data batch presented to the algorithm. The Hellinger distance between the two histograms P and Q is then computed using Eq. (4.44). The (intermediate) Hellinger distance is calculated first for each feature, and the average of distances for all features is then considered as the final Hellinger distance:

$$\delta_H(t) = \frac{1}{d}\sum_{k=1}^{d}\sqrt{\sum_{i=1}^{b}\left(\sqrt{\frac{P_{i,k}}{\sum_{j=1}^{b}P_{j,k}}} - \sqrt{\frac{Q_{i,k}}{\sum_{j=1}^{b}Q_{j,k}}}\right)^2} \tag{4.44}$$

where d is the dimensionality of the data, and $P_{i,k}$ ($Q_{i,k}$) is the frequency count in bin i of the histogram corresponding to the histogram P (Q) of feature k. Then compute $\epsilon(t)$, the difference in divergence between the most recent measure of the Hellinger distance, $\delta_H(t)$, and the Hellinger distance measured at the previous time stamp, $\delta_H(t-1)$. This differential between the current and prior distances is compared to a threshold, to determine whether the change is large enough to claim a drift. Rather than heuristically or arbitrarily selecting a threshold, an adaptive threshold that is automatically adjusted at each time stamp. To do this, first compute $\hat{\epsilon}$, the mean $|\epsilon(i)|$, and $\hat{\sigma}$, the standard deviation of differences in divergence, where $i = \lambda, \lambda+1, \ldots, t-1$. Note that the current time stamp and all steps before λ (last time a change was detected) are not included in the mean difference calculation.

The actual threshold $\beta(t)$ at time step t is then computed based on the mean and standard deviations of the differences in divergence. Two alternatives are proposed to compute this threshold: based on the standard deviation and on a confidence level. The first is computed simply by

$$\beta(t) = \hat{\epsilon} + \gamma\hat{\sigma} \tag{4.45}$$

where γ is some positive/real constant, indicating how many standard deviations of change around the mean are accepted as different enough. Note that $\hat{\epsilon} + \gamma\hat{\sigma}$ is used and not $\hat{\epsilon} \pm \gamma\hat{\sigma}$, as drift is flagged when the magnitude of the change is significantly greater than the average of the change in recent time (since the last detected change) with the significance controlled by the γ term and the standard deviation of the divergence differences.

The second implementation of HDDDM uses the t-statistic and scales by the square root of $t-\lambda-1$. Again, an upper-tailed test is used, $\hat{\epsilon} + t_{\alpha/2,df}\frac{\hat{\sigma}}{\sqrt{t-\lambda-1}}$ and not the two-tailed $\hat{\epsilon} \pm t_{\alpha/2,df}\frac{\hat{\sigma}}{\sqrt{t-\lambda-1}}$.

$$\beta(t) = \hat{\epsilon} + t_{\alpha/2,df}\frac{\hat{\sigma}}{\sqrt{t-\lambda-1}} \tag{4.46}$$

Once the adaptive threshold is computed, it is applied to observed current difference in

divergence to determine if drift is present in the most recent data. If magnitude $|\epsilon(t)| > \beta(t)$ then HDDDM signals that change has been detected in the most recent batch of data. As soon as change is detected, the algorithm resets $\lambda = t$ and $\mathcal{D}_\lambda = \mathcal{D}_t$. Note that the resetting of the baseline distribution is essential as the old distribution is no longer an accurate reference to determine how much the data distribution has changed (and whether that change is significant) as indicated by the large difference in Hellinger distance between time stamps. If drift is not detected, then HDDDM updates, rather than resets, the distribution $\mathcal{D}_\lambda$ with the new data $\mathcal{D}_t$. Thus, $\mathcal{D}_\lambda$ continues to be updated with the most recent data as long as drift is not detected. This histogram can be updated or reset using the following equation:

$$\begin{aligned} Q_{i,k} &\leftarrow Q_{i,k} + P_{i,k} && \text{if drift is not detected} \\ Q_{i,k} &\leftarrow P_{i,k} && \text{if drift is detected} \end{aligned}$$

Note that the normalization in Eq. (4.44) ensures that the correct density is obtained from these histograms.

The algorithm naturally lends itself to an incremental drift detection scenario since it does not need access to previously seen data. Rather, only the histogram of the current and reference distributions are needed. Therefore, the only memory required for this algorithm is the previous values of ϵ, $\delta_H(t-1)$ and the histogram of $\mathcal{D}_\lambda$ (Q).

4.4.4 Algorithm Performance Assessment

Determining the effectiveness of a drift detection algorithm can be best tested on carefully designed synthetic datasets using a variety of drift scenarios, such as abrupt or gradual drift. Since the drift is deliberately inserted into the dataset, the ability of the algorithm to detect a drift when one is known to exist (measured by sensitivity), as well as its ability to not signal one when there is in fact no drift (measured by specificity), can be easily determined for such datasets.

Detecting drift is only useful to the extent such information can be used to guide a learner track a nonstationary environment and learn the drift in the data. To do so, a naïve Bayes classifier is used, which can be easily updated as new data arrive, to determine whether detection of drift can be used to improve the classifier's performance on a drifting

classification problem. The following procedure is applied to determine the effectiveness of HDDDM:

- Generate two naïve Bayes classifiers with the 1^{st} database. Call them $\mathcal{N}_1$ and $\mathcal{N}_2$.
  for $t = 2, 3, \ldots$
  - Begin processing new databases for the presence of concept drift using HDDDM.
    - $\mathcal{N}_1$ is the target classifier that is updated or reset based on HDDDM decision. If drift is detected using HDDDM, reset $\mathcal{N}_1$ and train $\mathcal{N}_1$ only on the new data, otherwise incrementally update $\mathcal{N}_1$ with the new data.
    - Regardless of whether or not drift is detected, $\mathcal{N}_2$ is incrementally trained with the new data. $\mathcal{N}_2$ is therefore the control classifier that is not subject to HDDDM intervention.

  end for

One should expect $\mathcal{N}_1$ to outperform $\mathcal{N}_2$ on drifting environments. The on-line implementation of the naïve Bayes classifier is straightforward as the features are assumed class conditionally independent. The calculation of $p(x_i|\omega_j)$ required to implement naïve Bayes classifier, can be further simplified by assuming the data for the ith feature is normally distributed (though, this is not necessary for the algorithm to work). Thus, the parameters of naïve Bayes are computed as:

$$p(x_i|\omega_j) = \frac{1}{\sigma_{i,j}\sqrt{2\pi}} \exp\left(-\frac{(x_i - \mu_{i,j})^2}{2\sigma_{i,j}^2}\right) \tag{4.47}$$

$$\sigma_{i,j}^2 = \mathcal{E}[(X - \mu)^2] = \mathcal{E}[X^2] - \mathcal{E}[X]^2 \tag{4.48}$$

$$P(\omega_j|\mathbf{x}) \propto P(\omega_j) \prod_{i=1}^{d} p(x_i|\omega_j) \tag{4.49}$$

where x_i is the ith feature, σ_{ij}^2 is the variance of the ith feature of the jth class, μ_{ij} is the mean of the ith feature of the jth class, and ω_j is the label of the jth class.

4.5 Summary

In this chapter, a number of different algorithms were presented that all revolve around handling concept drift in data mining. Learn^{++}.CDS is an integration of Learn^{++}.NSE and SMOTE. The goal here is to use SMOTE to increase recall of minority class and Learn^{++}.NSE to track drifting concepts. SMOTE introduces a broad representation of the minority class at each time stamp. Learn^{++}.NIE use information in the F-measure, G-mean or weighted error along with a bagging variation to increase minority class recall and still track drifting concepts. Unlike Learn^{++}.NSE, Learn^{++}.NIE generates small ensembles of classifiers at each time stamp to reduce class imbalance, promote diversity, and assure all classifiers are trained on all of the minority class. WEA is designed to use information in unlabelled data to produce an accurate estimate of the Bayes-optimal weights derived from a discriminate function. Finally, a drift detection method that uses the raw features to detect drift in incremental learning problems was presented.

Chapter 5

Experiments

This chapter presents an empirical analysis of the algorithms discussed in Chapter 4 along with comparisons to some state-of-the-art algorithms in Chapter 3. The experiments are designed to answer the following research questions

Addressing Concept Drift from Imbalanced Data: Results are presented in Sections 5.5-5.7.

- To what extent, if any, will the performance on a minority class increase by integrating SMOTE-based sampling into Learn^{++}.NSE
- Can measures other than error be applied to weight classifiers in an ensemble? If so, can the algorithm still effectively track concept drift and boost the recall of a minority class?
- Does computing classifier voting weights with measures other than error provide reliable results?
- How do the new members of the Learn^{++} family of algorithms, Learn^{++}.CDS and Learn^{++}.NIE, compare to other approaches for learning concept drift and class imbalance?

Addressing Unlabelled Data and Concept Drift: Results are presented in Section 5.8.

- Can WEA effectively use unlabelled data to determine classifier voting weights? If so, how does WEA compare to Learn^{++}.NSE?

Addressing Drift Detection Using the Hellinger Distance Results are presented in Section 5.9.

- Is the Hellinger distance an effective metric that can be used to track changes in a dynamically changing environment?

A variety of different datasets are used to answer each of these questions. Synthetic datasets are primarily used to answer most of these questions as they allow control over

the experiments. Also, since such datasets can be created with low dimensionality, the boundaries can easily be visualized. Thus, we introduce a variety of different synthetic datasets that contain specific drift scenarios along with an under-represented class.

5.1 Algorithms Under Test

The following algorithms are used in the experiments. The selection of algorithms include some that are designed strictly for concept drift (†) and others that are designed to handle concept drift and class imbalance together (‡). SEA and Learn^{++}.NSE are included to serve as baseline comparison to demonstrate why new algorithms are needed to handle concept drift and class imbalance, and to demonstrate possible repercussions of applying concept drift problems to imbalance learning scenarios. Learn^{++}.NSE comparison allow us to determine the value added to Learn^{++}.CDS when SMOTE is applied for learning in a nonstationary environment with imbalanced classes.

- *SEA*†: Benchmark concept drift algorithm. See [49] for implementation details. Ensemble size was limited to 25 (same as suggested by the original paper authors) for all results presented here.
- *SERA*‡: Recently proposed for imbalanced datasets in concept drift problems. See [21] for implementation details. Sample size parameter, f, was set to 0.4 and $BBagging$ was implemented with five classifiers generated in $BBagging$, as recommended by the authors in [21].
- *Learn^{++}.NSE*†: Originally proposed in [12], Learn^{++}.NSE builds a single classifier at each database and dynamically weights classifier based on the pseudo error of the classifiers. The a and b parameters, which control the slope and cut off of the logistic sigmoid, were set to 0.5 and 15 receptively.
- *Uncorrelated Bagging*‡: Originally proposed in [61] is a bagging based algorithm that trains classifiers on all old minority data and a subset of the majority. The majority class sample parameter, r, was set to 0.3 for the experiments presented. Five classifiers are generated in the ensemble.
- *Learn^{++}.CDS* ‡: Originally presented in [108], this algorithm combines SMOTE and Learn^{++}.NSE as presented in the previous chapter.

- *Learn^{++}.NIE (wavg)*‡: Originally presented in [109]. The η parameter of Learn^{++}.NIE for the weighted error combination was set to 0.5, which gives equal weight to the error of each class. We keep this parameter constant over all experiments for consistency. The under-sampling in the bagging variation for Learn^{++}.NIE was set to approximately N/T where N is the number of instances in a batch and T is the number of classifiers generated at each time step.
- *Learn^{++}.NIE (gm)*‡: This implementation of Learn^{++}.NIE uses the geometric mean as the weighting measure. The number of classifier generated in the bagging variation algorithm is the same as Learn^{++}.NIE (*wavg*).
- *Learn^{++}.NIE (fm)*‡: This implementation of Learn^{++}.NIE uses the F-measure as the weighting method. The number of classifier generated in the bagging variation algorithm is the same as Learn^{++}.NIE (*wavg*).

The under-sampling in the bagging variation for Learn^{++}.NIE was set to N/K where N is the number of instances in a batch and K is the number of classifiers generated at each time stamp. However, the under-sampling ratio was set to 65% of the batch size for the NOAA dataset because this learning scenario does not contain a large class imbalance compared to the synthetic datasets. To avoid the accumulated minority data in UCB becoming a majority class (since this issue is not addressed in the original UCB algorithm) we apply a sliding window to the minority class data to ensure that we do not train on more positive (minority) instances than negative (majority) ones.

5.2 Evaluation Procedure & Evaluation

Raw classification accuracy is the traditional measure used to evaluate an algorithms' effectiveness on concept drift problems [49, 50, 112, 28]. However, error is not an appropriate measure to evaluate the algorithms, due to class imbalance in the learning problem. This section presents the processing of the data as well as the algorithm evaluation measures.

5.2.1 Batch Based Processing

All algorithms presented in the Section 5.1 are batch based processing algorithms, meaning that they require a group of examples for training a new classifier. This process is different

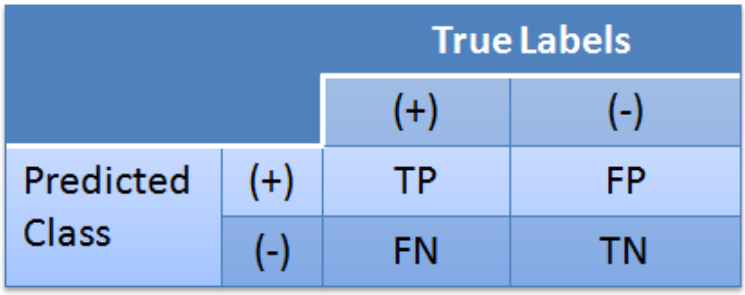


Fig. 5.1 : Confusion matrix

than on-line algorithms, which train on a single instance of data rather than a batch of data.

5.2.2 Algorithm Evaluation Measures

Raw classification accuracy (RCA), as shown in Eq. (5.1), is not a reliable metric in imbalanced datasets. The error can be computed using the information in a confusion matrix shown in Fig. 5.1. A confusion matrix is a $c \times c$ matrix. This matrix is obtained from test data and shows how many instances of each class are classified into different classes. The sum of the columns in Fig. 5.1 are the total number of instances from the testing data that have been labelled with the class indicated by the column. The sum of the rows are the total number of instances from the testing data that have been classified into the class indicated by the row. The sum of the diagonals is the number of instances that have been classified correctly. The overall accuracy is obtained by summing the diagonal components and dividing by the sum of all the entries. For a 2×2 confusion matrix there are four possible outcomes. For convenience the class labels are either positive (+) and negative (-). For the remainder of this paper the minority class is referred to as the positive class.

1. *True Positive (TP)*: is the number of instances that are in fact positive and correctly classified as such.
2. *True Negative (TN)*: is the number of instances that are in fact negative and correctly classified as such.

3. *False Positive (FP)*: is the number of instances that are in fact negative and incorrectly classified as positive.
4. *False Negative (FN)*: is the number of instances that are in fact positive and incorrectly classified as negative.

Raw classification accuracy, as indicated earlier, is not an accurate indicator for evaluating a classifier on datasets where the classes are imbalanced. For example, in a dataset in which the minority class constitutes only 1% of the instances, blindly choosing majority class for all instances provides 99% RCA, but 0% accuracy on the minority class, which is usually the more important class. Therefore, metrics other than error are used that will provide a better interpretation about how well the algorithm is performing on all classes. These measures must have the ability to interpret the performance of the minority class.

$$p_H^{(t)} = \frac{1}{N}\sum_{n=1}^{N}[\![H^{(t)}(\mathbf{x}_n) = y_n]\!] = \frac{TP+TN}{TP+TN+FP+FN} \tag{5.1}$$

The geometric mean (G-mean), given by Eq. (5.2), indicates the central tendency or typical value of a set of numbers. The geometric mean is computed using the class specific performances, p_k where k goes from 1 to c and c is the number of classes in the classification problem. Consider a situation where an algorithm performs extremely well on the negative class, say 95%, and the performance on the positive class is 10%. The classification error in this situation may be low assuming a large imbalance, however the G-mean in this example will be low because the algorithm performed very poorly on the positive class. The G-mean has been evaluated on imbalanced datasets by Kubat [65].

$$GM = \prod_{k=1}^{c} p_k^{1/c} \tag{5.2}$$

Precision, recall, and F_β-measure are also commonly used as performance measures. All of these measures can be obtained from the confusion matrix. The precision of a classifier, as shown in Eq. (5.3), is the ratio of the number of instances that are positive and are classified as positive to the number of instances that were classified as positive. The recall of a classifier, as shown in Eq. (5.4), is the ratio of the number of instances that are positive and are classified as positive to the number of instances that were classified that are truly

positive.

$$precision = \frac{TP}{TP + FP} \tag{5.3}$$

$$recall = \frac{TP}{TP + FN} \tag{5.4}$$

The F_β-measure is a test of accuracy that takes into account the precision and recall. When $\beta = 1.0$, the F-measure or balanced F-score (F_1 score) is the harmonic mean of precision and recall, as shown in Eq. (5.5). The harmonic mean is proportional to the squared G-mean divided by the arithmetic mean of the precision and recall.

$$F_\beta = \frac{(1 + \beta)^2 recall \times precision}{\beta^2 \times precision + recall} \tag{5.5}$$

$$F_1 = \frac{2 \times recall \times precision}{precision + recall} \tag{5.6}$$

The precision, recall, G-mean and F-measure are commonly used to evaluate algorithms that are designed for class imbalance. However, other measures like the area under the receiver operating characteristics (ROC) curves is also commonly used as an indicator of the performance of an algorithm on a target class.

5.2.3 ROC Curves and AUC

Receiver operating characteristics (ROC) curves are graphs for visualizing a classifier performance [113, 114, 115]. ROC curves have been used in signal detection to depict the tradeoff between true positive rates (*tpr*) and false positive rates (*fpr*) of classifiers tested on a binary classification problem. *tpr* is the recall while $fpr = FP/(FP + TN)$. The ROC curves may be used to analyze the tradeoff between *tpr* and *fpr*. The ROC curve is a 2-dimensional plot that depicts the relative tradeoffs between benefits and costs. In order to generate an ROC curve we need to have a classifier that is capable of providing a probabilistic output. A classifier that only has a label output only produces one (*tpr*,*fpr*) pair (i.e. one point in ROC space).

Generating an ROC curve requires labelled data and a classifier with a probabilistic output where the labels are given by $\omega \in \{0, 1\}$. The curve is generated by testing the

classifier on the labelled data and computing a posterior probability for all data. The true labels are considered to be 1 or 0 where 1 refers to the target (minority) class. A threshold, θ, is selected between 0 and 1. The threshold is compared to the posterior probabilities of the instance in the test dataset. If the posterior probability for the target class is greater than the threshold than the data are assign label $y_i = 1$ and if the posterior probability for the target class is less than the threshold the data are assigned to $y_i = 0$. From these labels the (*tpr*,*fpr*) pair is computed and this is one point in ROC space. This process is repeated by varying the threshold and collecting points in ROC space. The plot of *tpr* vs. *fpr* is the ROC curve. The area under the ROC curve is referred to as the AUC. AUC of a classifier is equivalent to the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance [113]. Classifiers that have the ROC curve in the upper left corner of the graph are desirable, thus yielding a high AUC. AUC=0.5 is equivalent to a classifier randomly assigning labels.

Let's consider an example of a naïve Bayes classifier trained on data sampled from a standardized Gaussian distribution with class labels randomly assigned to the data. The AUC should be very close to 0.5 since the classifier is trained on randomly labeled data. Fig. 5.2 contains the ROC curves for both classes in the training data. Only one of the classes is considered the target to generate the curves. The AUC for this example is 0.50185 for ω_1 and 0.50295 for ω_2. Fig. 5.3 considers the ROC curves for two of the classes in the tree cover type (covtype) dataset found in [105]. A naïve Bayes classifier is trained on 10,000 randomly sampled instances. The AUC for this dataset is significantly higher than the previous example since the data are not randomly labeled. The minority class will serve as the target class when ROC curves are generated.

5.2.4 Overall Performance Measure

Typically, one will find that no single algorithm outperforms all others in all measures, which justifies the use of the combined overall performance measure as well as the mean rank as a figure of merit. Therefore, we introduce an overall performance measure (OPM), a convex combination of raw classification accuracy, F-measure, AUC and minority class recall. For this study $\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = 1/4$.

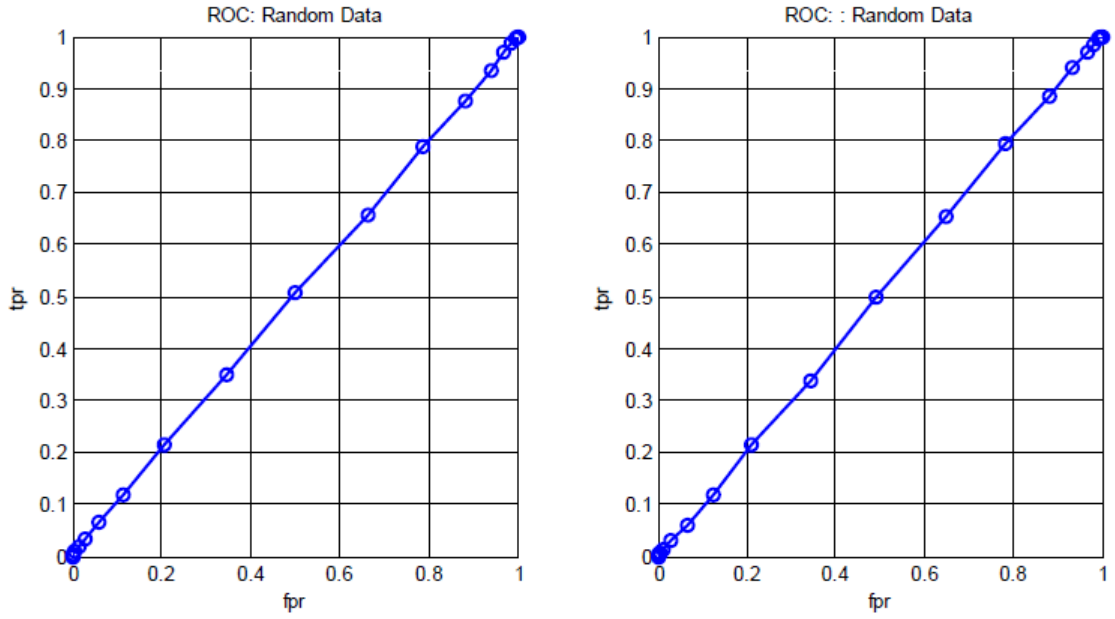


Fig. 5.2 : A naïve Bayes classifier with a Gaussian kernel was generated on 10,000 random instances drawn from a standardized Gaussian distribution. The class labels are produced by computing the $\texttt{sign}(\mathcal{N}(0,1))$. The AUC for ω_1 (left) is 0.50185 and ω_2 (right) is 0.50295.

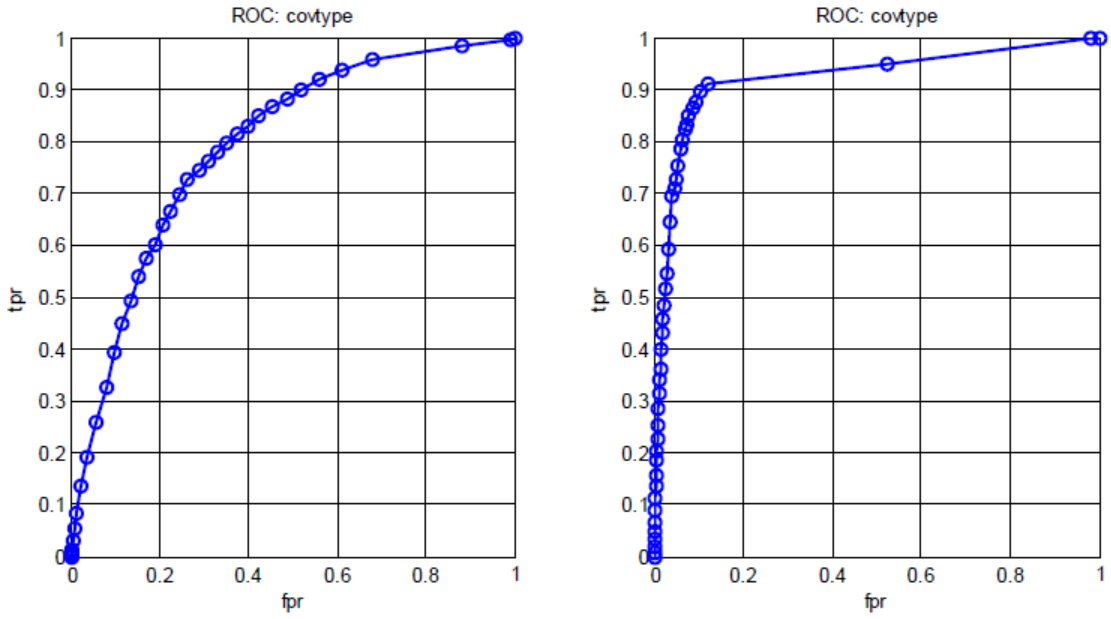


Fig. 5.3 : A naïve Bayes classifier with a Gaussian kernel was generated on 10,000 randomly selected instances and tested on 6,000 randomly selected instances. The ROC curve was generated using 200 thresholds. The AUC for ω_1 (right) is 0.7905 and ω_3 (left) is 0.9229.

$$\text{OPM} = \alpha_1 \times (1 - \varepsilon_H) + \alpha_2 \times F_1 + \alpha_3 \times \text{AUC} + \alpha_4 \times \text{Recall} \tag{5.7}$$

We also average each figure of merit (AUC, RCA, F-measure, and recall) over the course of the entire experiment to form a single value to represent an algorithm on a particular dataset. The average values are ranked and used to make comparisons between algorithms. The ranks can range from (1) to (8) where (1) is the best performing algorithm and (8) is the worst performing algorithm on any measure on a given dataset. We use the Friedman test as described in [116] to make comparisons between classifiers over multiple datasets after all datasets have been presented.

5.3 Base Classifier Selection

Learn^{++}.NSE was throughly evaluated on a variety of incremental learning scenarios that were both synthetic and real-world in [13]. Also in [13], an analysis of various base classifiers (naïve Bayes, SVM, MLPNN) and pruning methods (error/age based) were evaluated with Learn^{++}.NSE. Prior work has shown that Learn^{++}.NSE is fairly robust to the selection of the base classifier provided that the fundamental assumption of the classifier are not being violated. However, unbalanced data was not addressed in the work. This study uses a classifier not used in [12, 107, 13] and ensemble pruning is not addressed in this work.

In this study, a decision tree classifier is selected as the base classifier for several reasons: i) decision trees are quite possibly the most widely used classification algorithm (many implementations are available [117, 52, 118, 78, 119, 30]), ii) decision trees are commonly used with ensemble techniques, iii) decision trees have been shown to provide favorable qualities when data experience concept drift, and iv) the interpretation of a decision tree is intuitive compared to SVM or MLPNN. Tree classifiers are usually described in graph terminology.

The Classification and Regression Tree (CART) was selected using a Gini splitting criterion [117, 52]. The stopping criterion for growing the tree is based on a hypothesis test to determine whether the split is beneficial. Decision trees can be overly complex thus leading to over fitting, however a stopping criterion can be used to avoid this. The χ^2 statistic as presented in [52] is applied to stop growing a tree.

Decision trees are used widely throughout data mining and machine learning. Unlike

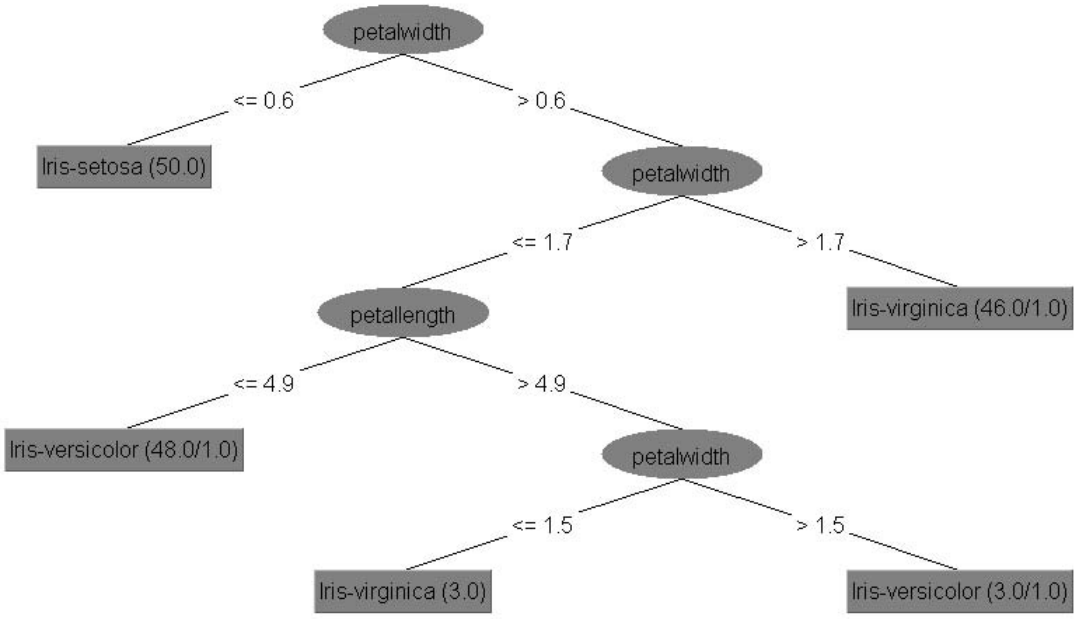


Fig. 5.4 : Visualization of a decision tree on the Fisher iris dataset

other base classifiers used with Learn^{++}.NSE, decision trees can easily handle nominal, ordinal, and cardinal features in the same dataset. Decision trees are comprised of a root node connected to internal nodes (descendants) via branches that represent the values of the attributes of the data. Each descendant is connected to a parent node and each decision made at a node splits the data through the branches. A node which does not have any descendants is a leaf node. The leaf node is associated with a category (label). The same label may appear at different leaves in the same tree. Decision trees are built recursively. A graphical model of a decision tree classifier for the Fisher iris dataset is shown in Fig. 5.4. A new instance is classified by starting at the root and evaluating if $x_{PL} < 0.6$ or if the pedal length is less than 0.6. If this condition is true the instance is classified as setosa otherwise we pass the instance down the right sub-tree until we reach a leaf.

5.4 Key Observations For Learning Concept Drift from Imbalanced Data

The results are presented with several different figures of merit, mentioned above, to analyze the algorithms' performance. The shading enveloping each curve in the figures below represents a 95% confidence interval based on 40 independent trials unless otherwise

noted (refer to the digital copy of the article for color figures). Due to the number of comparisons for each database, the results are split into three sets of figures: 1) comparing Learn^{++}.NSE and SEA; 2) comparing three versions of Learn^{++}.NIE with respect to three different error measures (F-measure (fm), G-mean (gm) and weighted average ($wavg$)); and 3) comparing Learn^{++}.CDS, Learn^{++}.NIE (fm), UCB and SERA. Among three versions of Learn^{++}.NIE, F-measure was used as the representative in comparison against other algorithms, since F-measure combines both recall and precision. The averaged figures of merit of all algorithms, along with their 95% confidence intervals and rankings are provided for all algorithms. Before presenting the empirical analysis on the synthetic and real-world datasets, lets summarize key observations that were consistent across a wide range of datasets and hence learning scenarios.

1. Learn^{++}.NIE (fm) and Learn^{++}.CDS consistently provide results, which rank them at or near the top three for OPM on nearly all datasets.
2. Learn^{++}.NIE (fm) and Learn^{++}.CDS typically provide a significant increase in recall, AUC, F-measure, and OPM compared to their predecessor Learn^{++}.NSE (as well as compared to SEA), while maintaining a reasonable overall accuracy, whereas UCB's increase in recall comes at the cost of lower overall accuracy and F-measure. In fact, UCB consistently maintains a good rank for recall, but holds the worst rank in terms of overall accuracy. Learn^{++}.NIE (fm) and Learn^{++}.CDS provide significant improvement in overall accuracy and F-measure compared to UCB.
3. The simple integration of SMOTE into Learn^{++}.NSE has improved the OPM rank on every dataset presented in this study.
4. Learn^{++}.NIE (fm) typically provides better results than the (gm) or ($wavg$). We find that Learn^{++}.NIE (fm) provides significant improvements over ($wavg$) in AUC, recall and OPM whereas only AUC is improved upon compared to (gm). However, the $wavg$ implementation provides a unique control over class specific errors.

5.5 Synthetic Experiments

Synthetic datasets are used to evaluate all algorithms under a variety of different drift scenarios and varying levels of class imbalance. The synthetic datasets allow us to observe the algorithm's behavior under controlled environments. Synthetic data can be used to

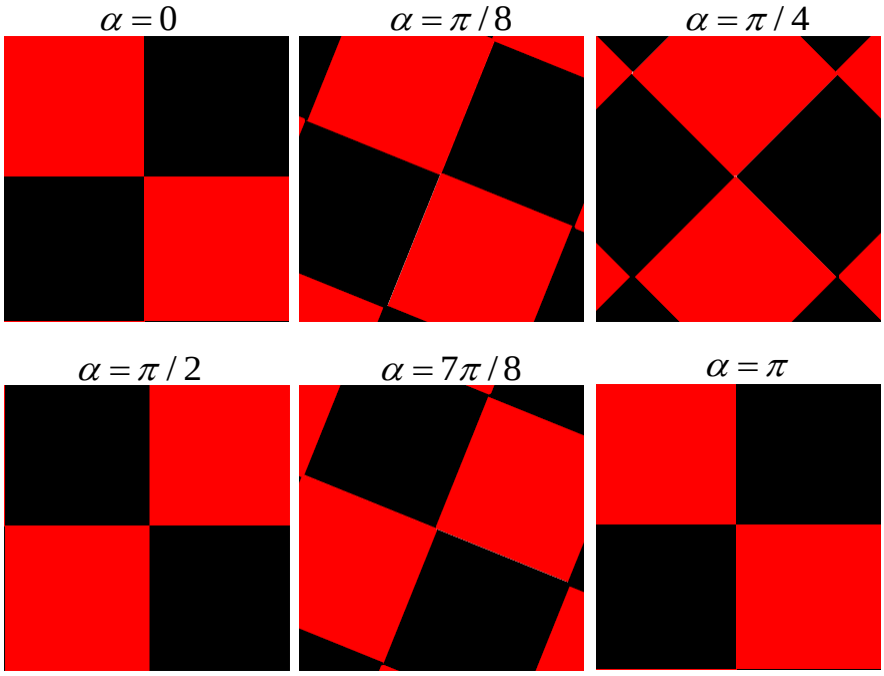


Fig. 5.5 : Rotating checkerboard dataset (half experiment, $\alpha \in [0, \pi]$)

simulate certain types of drift such as gradual or reoccurring. Unlike synthetic datasets, we cannot control the drift rate, class prior (although this can be adjusted with sampling), drift type or the time of the drift for real-world datasets.

5.5.1 Checkerboard Dataset

The rotating checkerboard problem is a challenging generalization of the classical non-linear XOR problem. The XOR problem is the special case when the angle of rotation equals 0, $\frac{\pi}{2}$, π, $\frac{3\pi}{4}$ and 2π. This simple experiment is particularly effective for observing an algorithms ability to learn data in recurring environments. Fig. 5.5 depicts half of the rotating checkerboard experiment. This experiment is controlled with two free parameters. The size of the checkerboard (a) or the angle of rotation (α) can be varied. Fig. 5.5 and the experiments presented is this thesis keep a fixed at 0.5. Fig. 5.5 shows half of an entire experiment ($\alpha \in [0, \pi]$), indexed to the parameter α. Note that after half a rotation, data are drawn from a recurring environment, as the $\alpha \in [\pi, 2\pi]$ interval will create an identical distribution drift to that of the $\alpha \in [0, \pi]$ interval.

Data from both classes are generated, however one of the classes is under-sampled to ensure there is an unbalanced data problem. In this thesis, datasets with 1000 instances are presented at each time stamp. The class balance may vary between $2.5 - 5\%$ due to randomness in the sample. Therefore, a majority class classifier will yield performances of $\approx 95 - 97.5\%$ on any dataset.

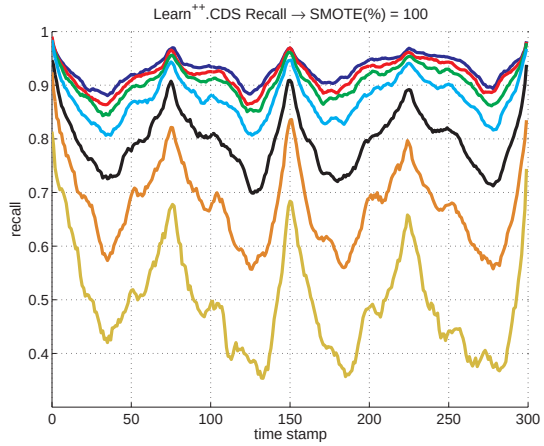
5.5.1.1 Learn^{++}.CDS Analysis on Checkerboard Data

This section presents an analysis of Learn^{++}.CDS under varying levels of class imbalance as well as varying levels of a percentage of SMOTE. The power of SMOTE is demonstrated the as well as potential issues with using SMOTE on certain problems. Recall from Fig. 3.10 that SMOTE generation of synthetic minority class instances in the regions of the majority class feature space.

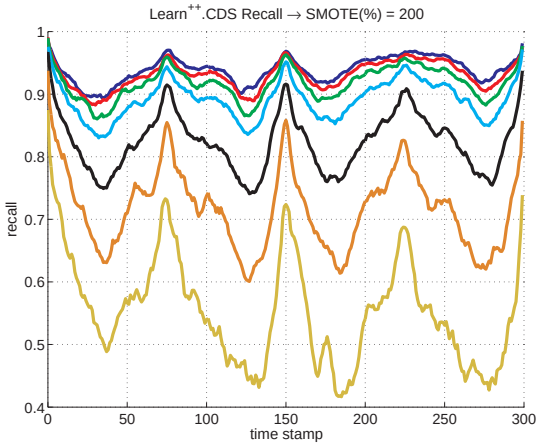
Fig. 5.6 shows the results using SMOTE with various levels of imbalance in the data stream. Each line in the subplots indicate a constant level of imbalance and each subplot is a different percentage of SMOTE applied to the training data. From Fig. 5.6(a), the observation can be made that data with 2.5% ($P(\omega_2) = 0.025$) has the least amount of recall, which is to be expected. Adding SMOTE into the training data from 100% to 1500% increases the recall of the minority class significantly from a time-averaged mean recall metric. However, the boost in recall comes at the cost of the overall performance as shown in Fig. 5.7.

5.5.1.2 Algorithm Analysis on Checkerboard Dataset

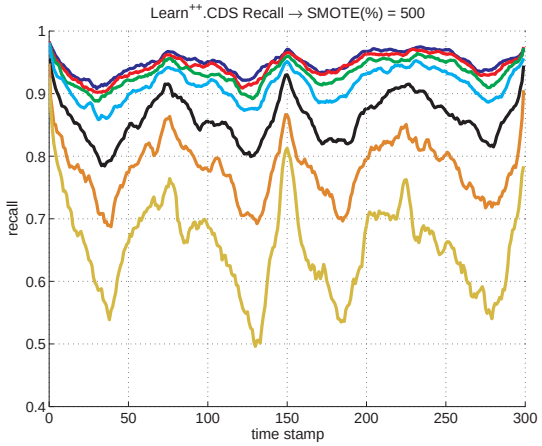
Fig. 5.8, 5.9, and 5.10 present the results on the rotating checkerboard dataset. The SMOTE percentage was set to 500% for this experiment. The mean values of all figures of merits used in the evaluation are tabulated in Table 5.1. Several trends can be observed, which appear to be common with those observed in the Gaussian experiment. First, Learn^{++}.NSE has very good accuracy, but primarily due to its performance on the majority class. Yet, Learn^{++}.CDS further improves the composite overall performance measure (OPM) by utilizing SMOTE to build more robust classifiers for unbalanced data. In fact, Learn^{++}.CDS performs consistently well across all figures of merit, with the best mean rank, closely followed by Learn^{++}.NIE. Learn^{++}.NIE (fm) and Learn^{++}.CDS provide significant



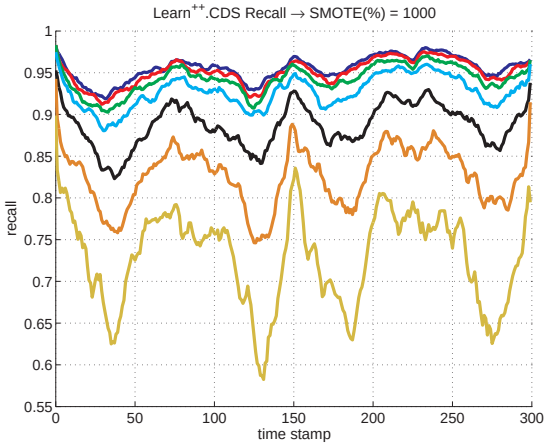
(a) SMOTE(%)=100



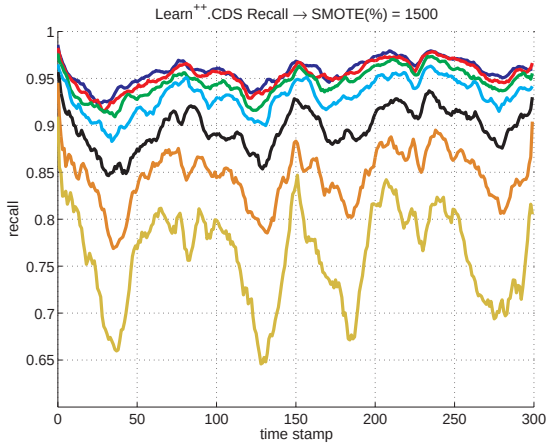
(b) SMOTE(%)=200



(c) SMOTE(%)=500



(d) SMOTE(%)=1000



(e) SMOTE(%)=1500

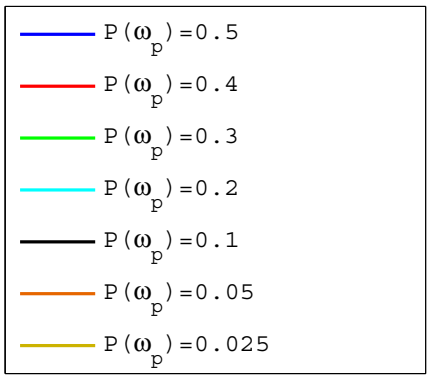
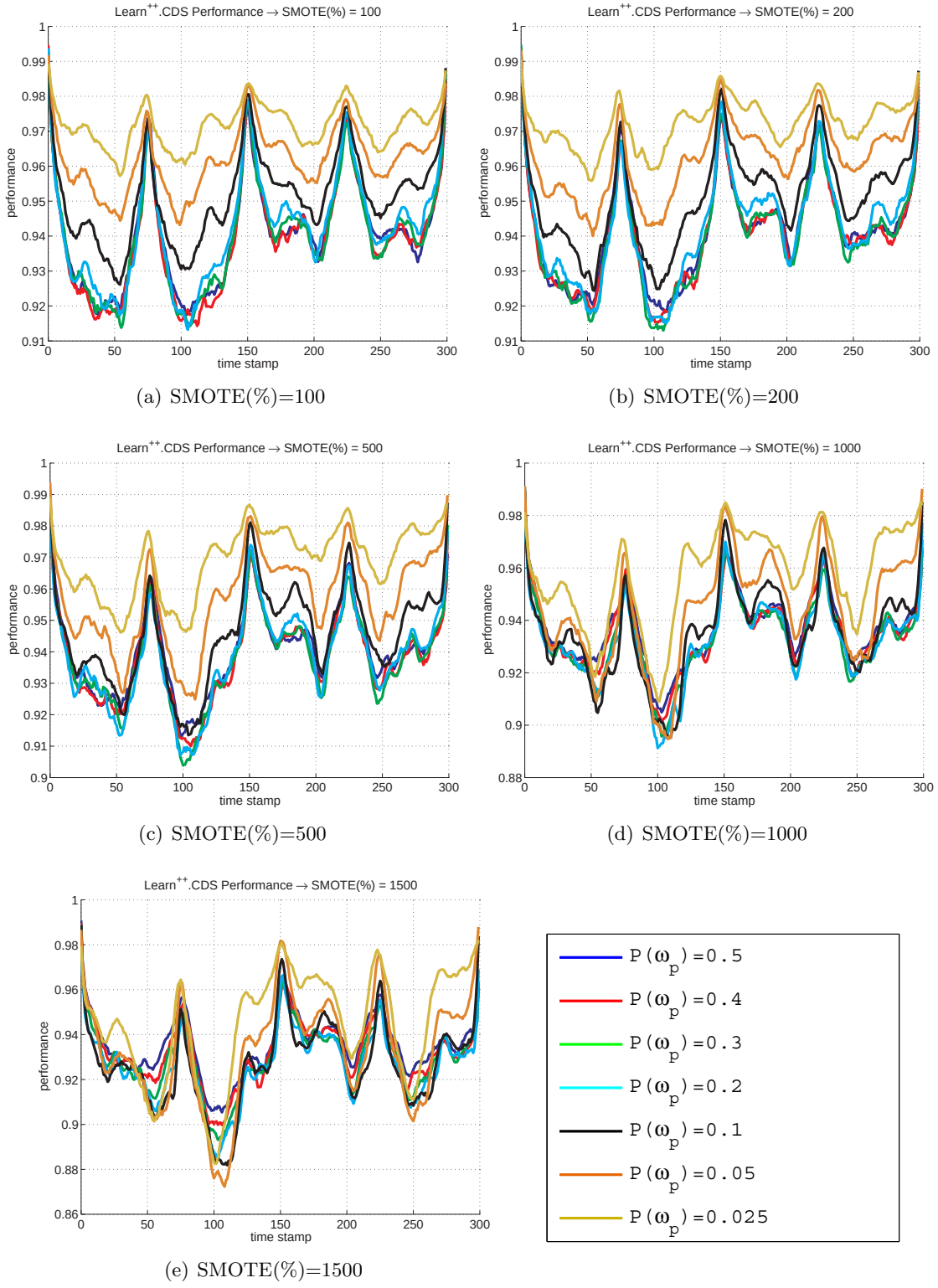


Fig. 5.6 : Recall for Learn^{++}.CDS with varying levels of SMOTE



(a) SMOTE(%)=100

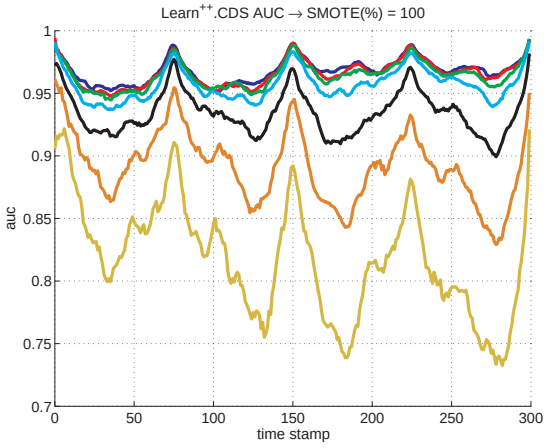
(b) SMOTE(%)=200

(c) SMOTE(%)=500

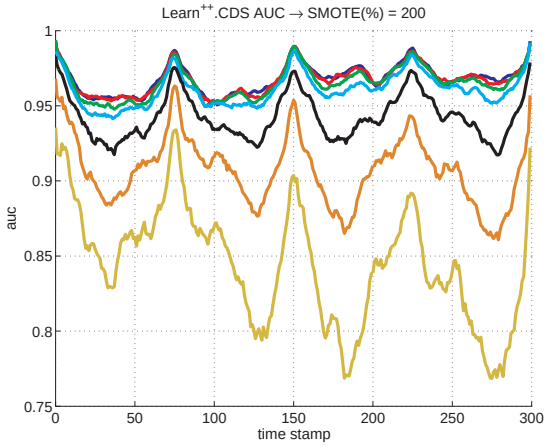
(d) SMOTE(%)=1000

(e) SMOTE(%)=1500

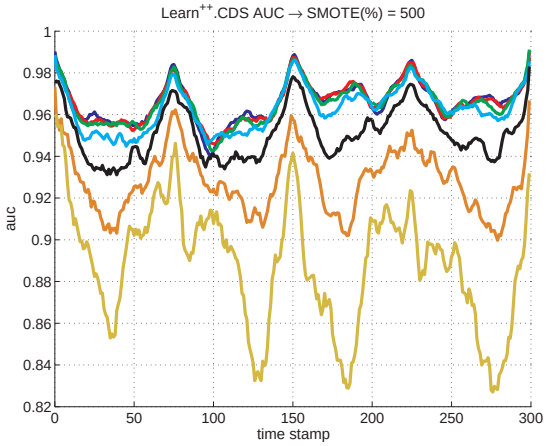
Fig. 5.7 : Raw Classification Accuracy for Learn^{++}.CDS with varying levels of SMOTE



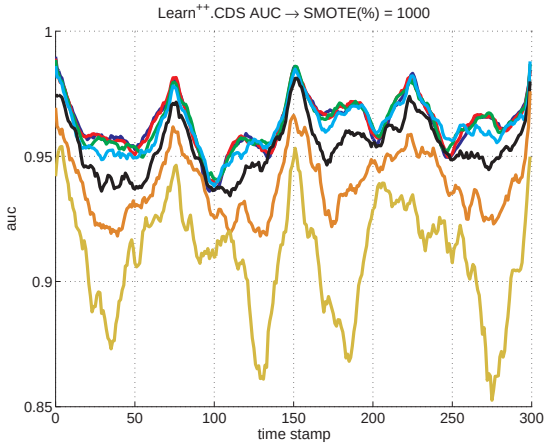
(a) SMOTE(%)=100



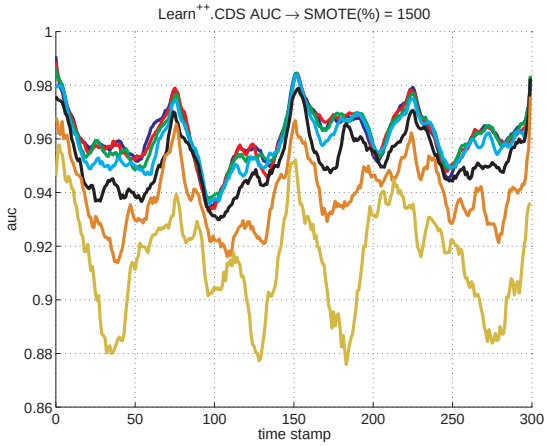
(b) SMOTE(%)=200



(c) SMOTE(%)=500



(d) SMOTE(%)=1000



(e) SMOTE(%)=1500

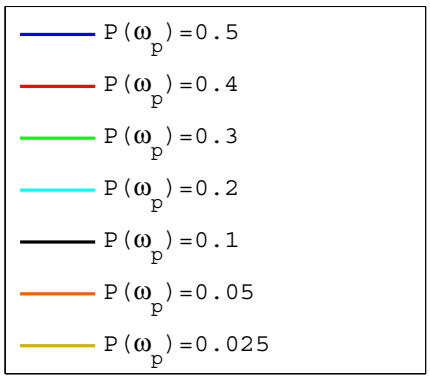


Table 5.1 : Algorithm Summary on Checkerboard Data

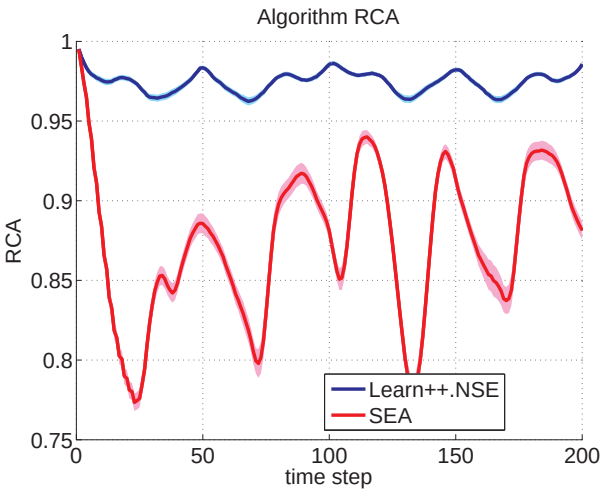
	RCA	**FM**	**AUC**	**Recall**	**OPM**	**Rank**
L++.NSE	97.45±0.17(1)	68.25±2.14(2)	83.76±1.17(4)	56.55±2.48(7)	76.50±1.49(3)	3.5
SEA	87.41±0.63(7)	21.93±1.63(8)	65.75±1.29(8)	31.87±2.18(8)	51.74±1.43(8)	7.8
NIE(fm)	95.06±0.47(3)	61.45±2.51(3)	92.62±0.85(1)	74.32±2.20(3)	80.86±1.51(2)	2.5
NIE(gm)	90.02±0.51(5)	42.11±1.94(5)	83.37±1.13(5)	66.76±2.20(5)	70.57±1.45(6)	5
NIE($wavg$)	89.89±0.51(6)	41.15±1.86(6)	82.75±1.12(6)	65.91±2.16(6)	69.93±1.41(7)	6
L++.CDS	97.18±0.21(2)	72.93±1.82(1)	90.89±0.96(3)	74.50±2.19(2)	83.88±1.30(1)	2
SERA	92.89±0.43(4)	52.57±2.29(4)	80.80±1.29(7)	67.39±2.55(4)	73.41±1.64(5)	4.8
UCB	85.78±0.51(8)	38.26±1.44(7)	91.89±0.70(2)	82.33±1.75(1)	74.57±1.10(4)	4.5

improvement to AUC, recall and F-measure compared to their predecessor. Second, as in the Gaussian dataset, UCB ranks best in terms of minority class recall but ranks lowest in terms of overall accuracy. UCB also experiences a large drop in rank when using the F-measure for evaluation. We do not observe this large drop in rank across measures for Learn^{++}.NIE (all implementations) or Learn^{++}.CDS. We also observe that Learn^{++}.NIE (fm) maintains a significantly better rank than the (gm) or ($wavg$) implementation of Learn^{++}.NIE in terms of OPM and other measures, similar to what was observed with the Gaussian problem. Finally, we note that the performance peaks that appear every 50 time steps coincide with the checkerboard being at right angle, corresponding to the simplest distribution for classification.

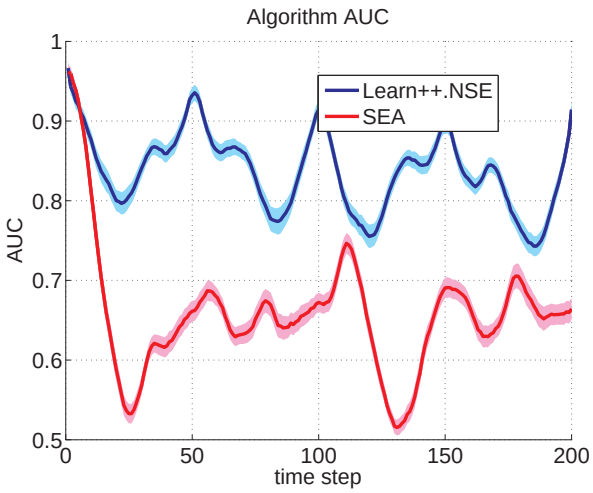
The weight evolution of Learn^{++}.NSE and Learn^{++}.NIE are shown in Fig. 5.11 for the rotating checkerboard dataset. There is little differentiation between the weights for Learn^{++}.NSE and Learn^{++}.NIE on this particular experiment. Simply note that both algorithms are capable of recalling environments on which the classifiers were trained on. This observation can be made because classifiers that become relevant again when they encounter environments on which they were trained (i.e., classifier weights increase when reoccurring environments are present).

5.5.2 Spiral Dataset

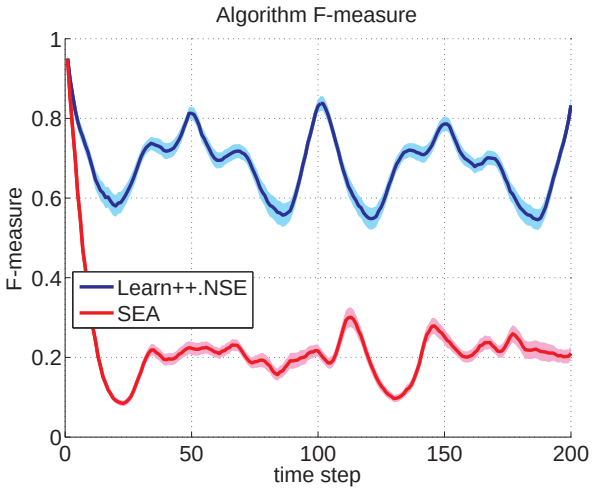
The rotating spiral dataset consists of four spirals, two for the minority class and two for the majority class as shown in Fig. 6. This figure shows the true decision (not boundary generated by a minimum error classifier) boundary at six different time stamps where the



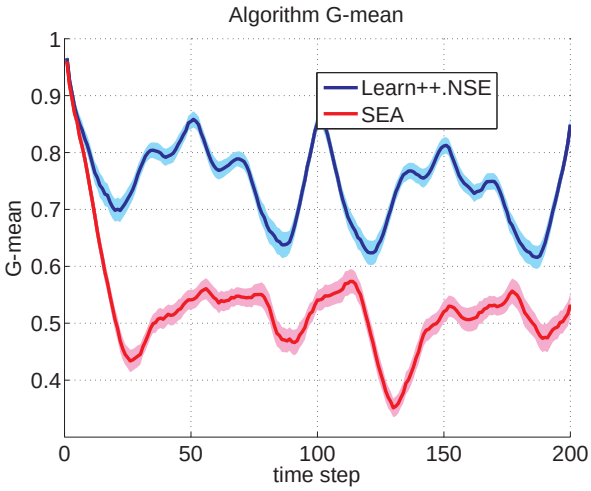
(g) Raw Classification Accuracy



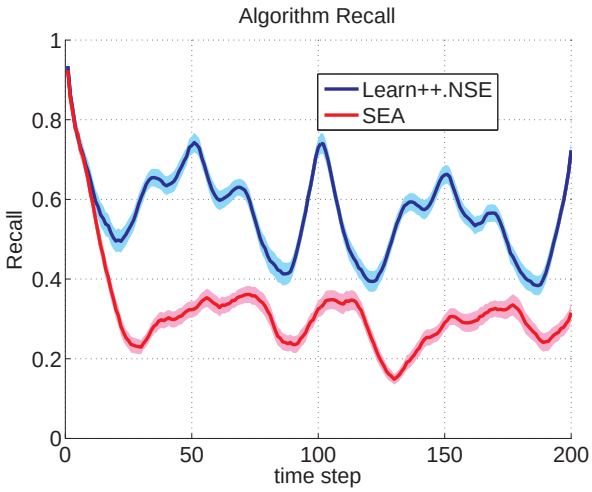
(h) AUC



(i) F-measure

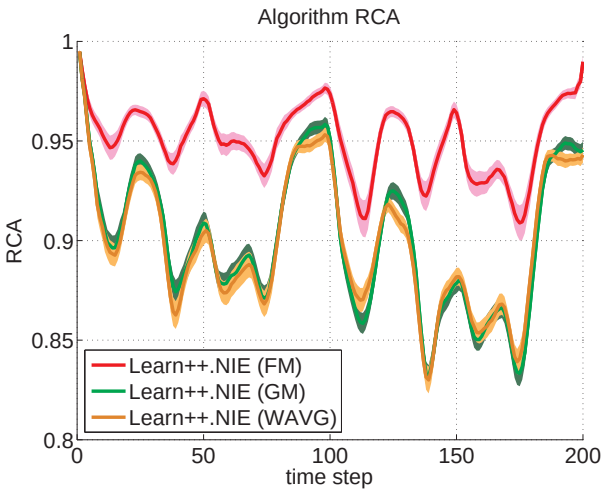


(j) G-mean

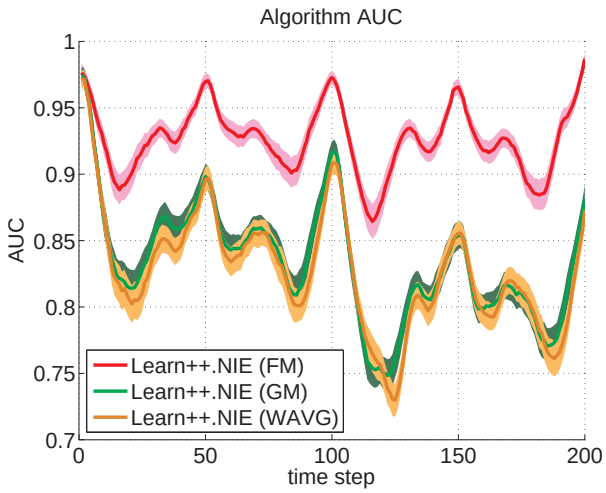


(k) Recall

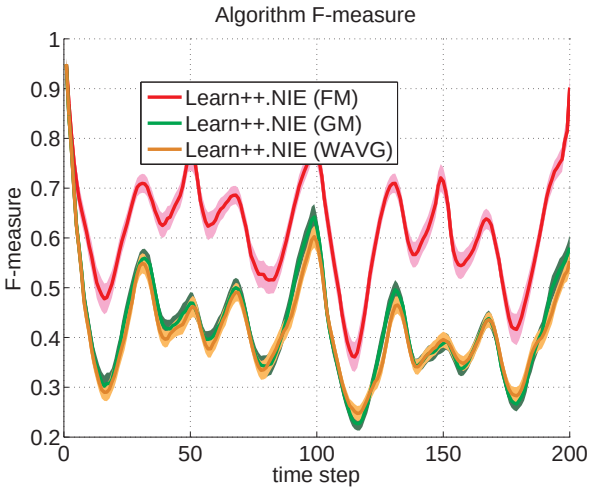
Fig. 5.8 : Learn^{++}.NSE and SEA evaluated on Checkerboard data



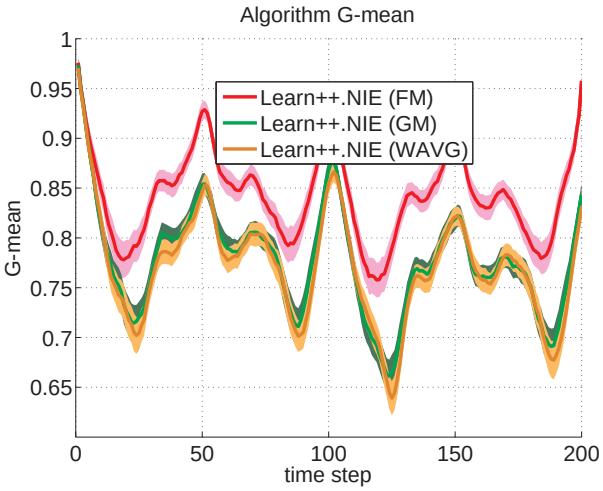
(a) Raw Classification Accuracy



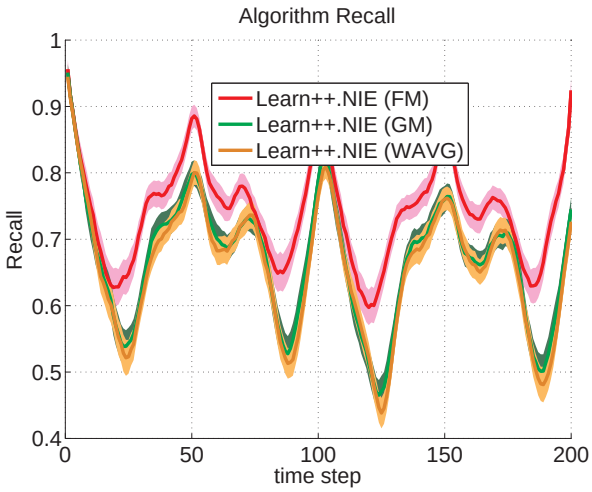
(b) AUC



(c) F-measure

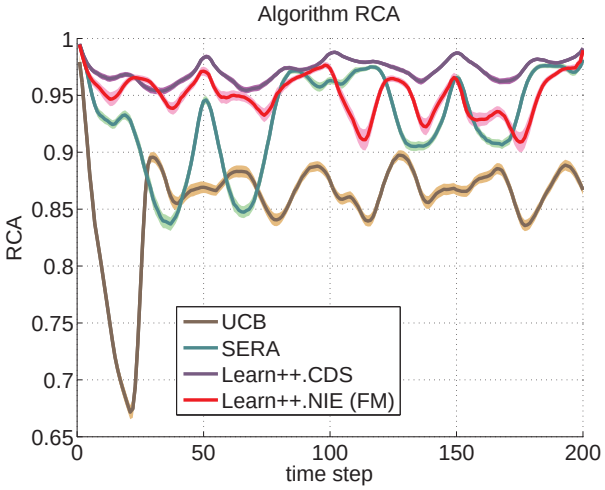


(d) G-mean

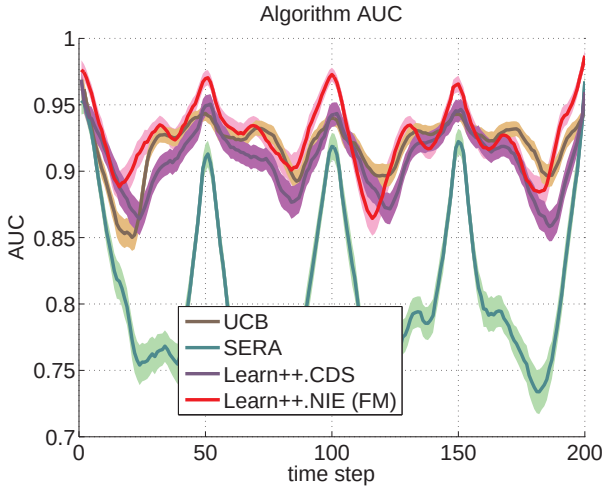


(e) Recall

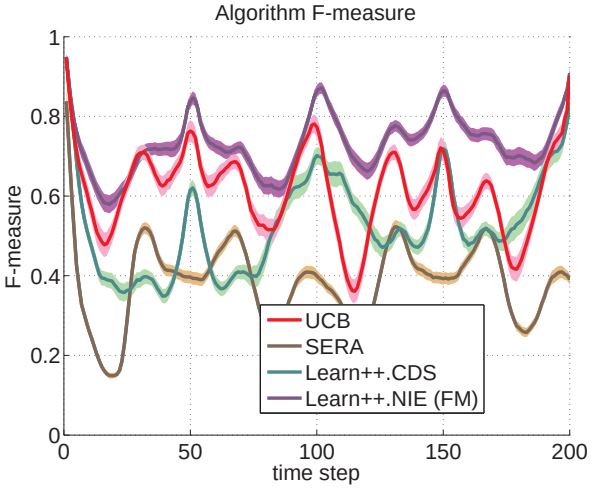
Fig. 5.9 : Learn^{++}.NIE family of algorithms evaluated on Checkerboard data



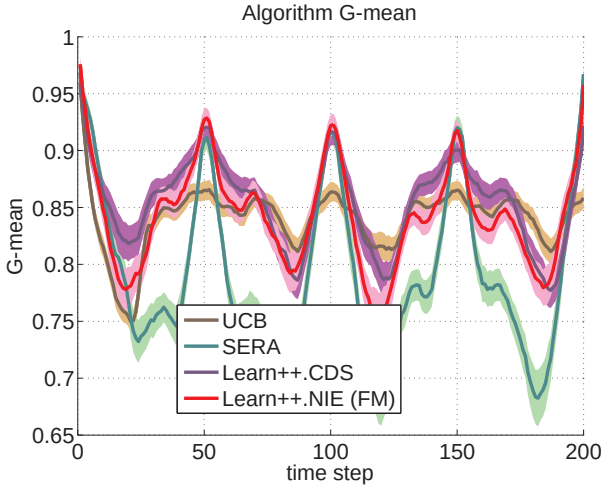
(a) Raw Classification Accuracy



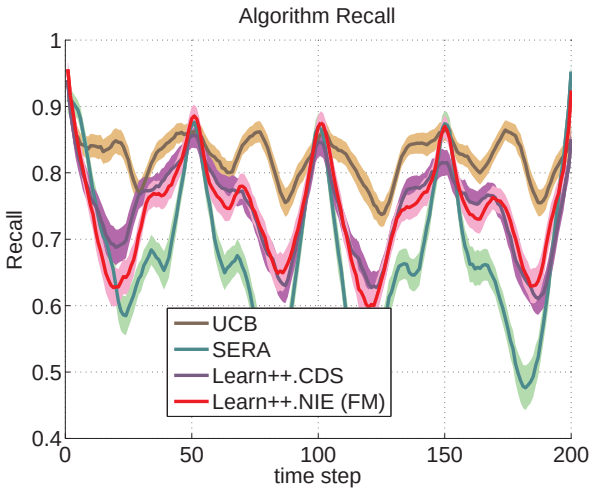
(b) AUC



(c) F-measure

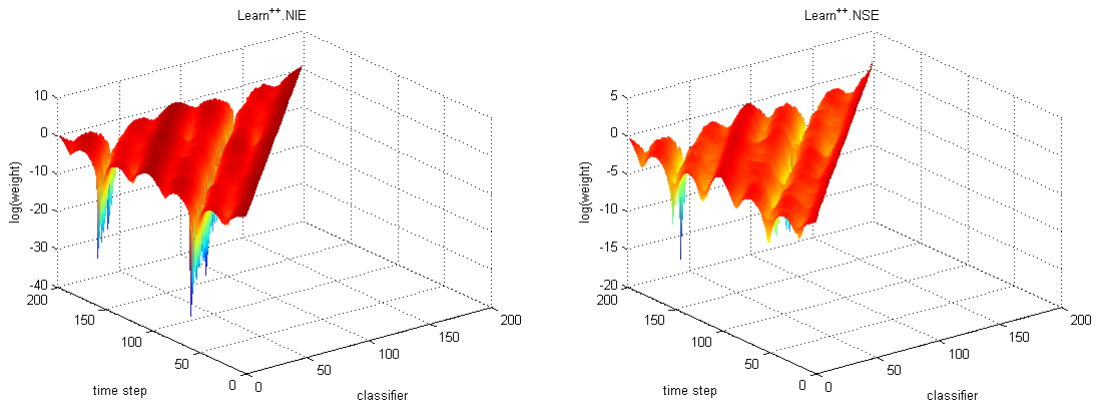


(d) G-mean



(e) Recall

Fig. 5.10 : Baseline algorithms evaluated on Checkerboard data



(a) Learn^{++}.NIE (wavg) (b) Learn^{++}.NSE

Fig. 5.11 : Learn^{++} weight evolution on Gaussian concept drift problem

angle, α is between $[0, \pi]$. The initial feature space for the rotating spiral dataset is shown in Fig. 5.12(a) and the spirals rotate over 300 time stamps at evenly spaced intervals. The environment repeats every 150 time stamps (i.e. $\alpha = \pi$), thus the experiment presented contains two full rotations, again to simulate a recurring environment. This is clearly shown in Fig. 5.12. The minority class consists of ≈5% of the total data size.

The spirals are created using two parametric equations shown below. The $\boldsymbol{\sigma}$ term is a small amount of random Gaussian noise to add in variability into different sets. As θ is increased, the spiral will move outward.

$$\mathbf{x} = \begin{pmatrix} \theta \cos \theta \\ \theta \sin \theta \end{pmatrix} + \boldsymbol{\sigma} \tag{5.8}$$

A rotation transformation matrix is used to evenly space the spirals around a grid. The β value calculated in the beginning and c is simply a counter going from 1 to C where C is the number of spirals. So the position on the cth spiral is calculated by using the equations below.

$$\beta = \frac{2\pi}{C} \tag{5.9}$$

$$\mathbf{T}_c = \begin{pmatrix} \cos((c-1)\beta) & \sin((c-1)\beta) \\ \sin((c-1)\beta) & -\cos((c-1)\beta) \end{pmatrix} \tag{5.10}$$

where C is the is the number of spirals to be generated and $\mathbf{T}_c$ is the transformation matrix for the cth spiral. The result of this matrix allows one to create any number of spirals that are evenly spaced apart on a grid. The instance pair becomes $(\mathbf{T}_c\mathbf{x}, c)$. Four spirals are generated and combined into pairs of two as shown in Fig. 5.12. The environment is rotated by using the $\mathbf{T}_c\mathbf{x}$ and another transform matrix computed using the rotation parameter α.

5.5.2.1 Learn^{++}.CDS Analysis on Spiral Data

Fig. 5.13 shows the effect of the amount of SMOTE added into the training dataset. As one would expect, the recall increases as a function of the SMOTE percentage (refer to Fig. 5.13(e)). However, Fig. 5.13(a) indicates that the boost in minority class recall is again at the cost of the overall performance and F-measure.

The RCA as well as the F-measure experience a constant drop every time more SMOTE is added into the dataset. On the other hand, the recall is increased with each increase in SMOTE percentage. However, there becomes a point where adding SMOTE into a dataset does not continue to significantly increase the minority class recall. Hence, the law of diminishing returns. Fig. 5.13(b) indicates there is little difference in AUC after the SMOTE percentage goes above 1000%.

5.5.2.2 Algorithm Analysis on Spiral Dataset

Fig. 5.14, 5.15 and 5.16 present the results on the rotating spiral dataset. The mean values of all figures of merits used in the evaluation are tabulated in Table 5.2. One prominent observation on this recurring environment is all implementations of Learn^{++} algorithms are able to use old information stored in the ensemble to increase several statistical measures when the environment reoccurs. This is a custom-designed property of all Learn^{++} family of algorithms, thanks to their weighting mechanisms that can deactivate and reactivate classifiers based on whether they are relevant to the current environment. Notice that all versions of Learn^{++} algorithms (though some – such as *wavg* implementation – more strongly than others) achieve a significant boost in RCA, minority recall, F-measure and

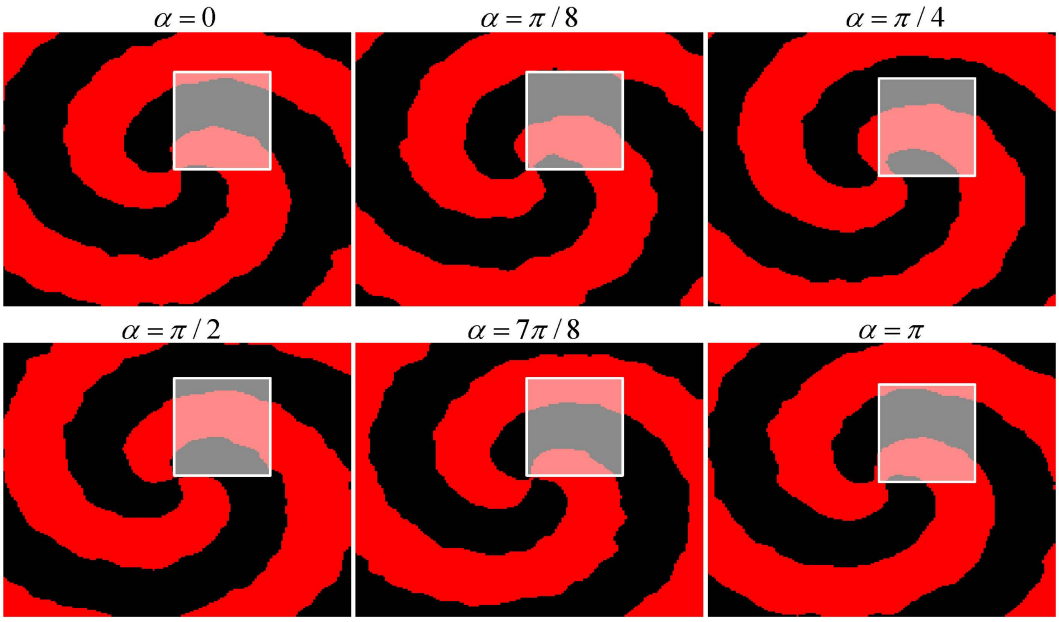
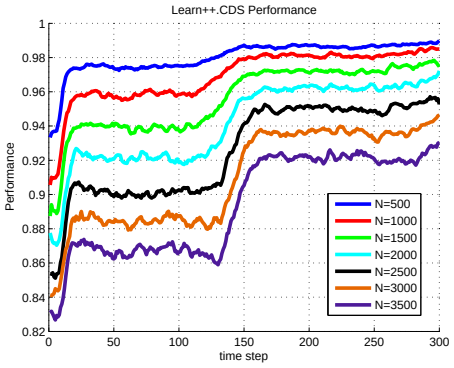


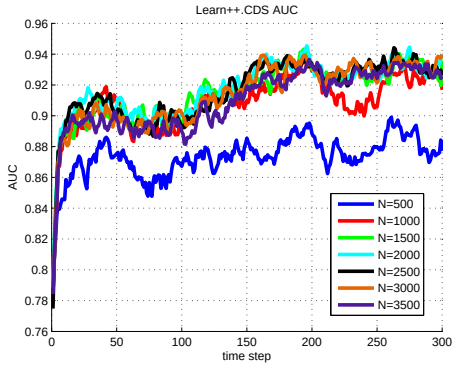
Fig. 5.12 : Rotating spiral experiment dataset (half experiment, $\alpha \in [0, \pi]$). The transparent region of the feature space represents a static window throughout the course of the experiment.

AUC as the recurring environment is encountered (time stamps ¿ 150) as shown in Fig. 5.15. Several other observations are also worth noting. First, while Learn^{++}.NSE has the best raw classification accuracy, again this is due to majority class classification. Lacking a mechanism to accommodate imbalanced data, Learn^{++}.NSE performs poorly on recall and AUC. Following Learn^{++}.NSE, Learn^{++}.NIE (fm) has the best raw classification accuracy, and unlike Learn^{++}.NSE, it maintains a high performance on all figures of merit. It is highly competitive with other algorithms across all measures. In fact, Learn^{++}.NIE (fm) outperforms all other algorithms in overall performance (OPM), and has the best mean rank across all measures among all algorithms, including Learn^{++}.NIE with $wavg$ and gm. This was observed on previous datasets as well. Second, a significant increase in OPM is observed and mean rank for Learn^{++}.CDS (which adds SMOTE to Learn^{++}.NSE) over its predecessor Learn^{++}.NSE, demonstrating the impact of adding SMOTE to Learn^{++}.NSE for learning unbalanced classes.

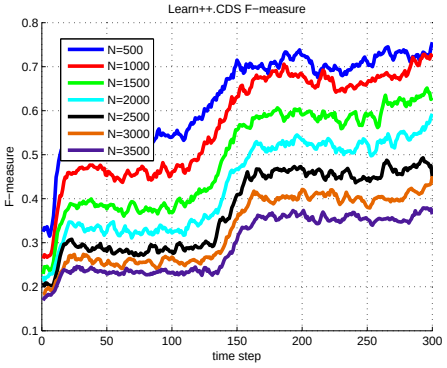
This dataset identifies a specific weakness of the SERA algorithm, which performs particularly poorly on this dataset with the poorest rank in every measure. A unique



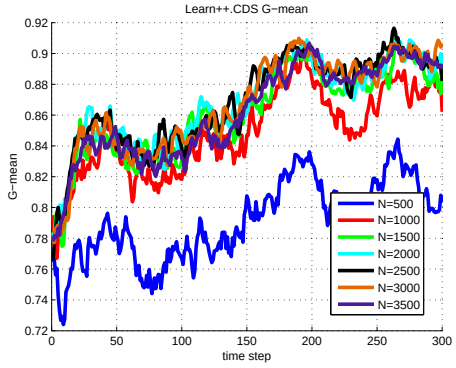
(a) Raw Classification Accuracy



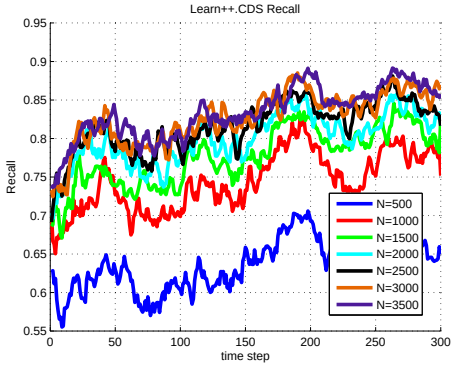
(b) AUC



(c) F-measure

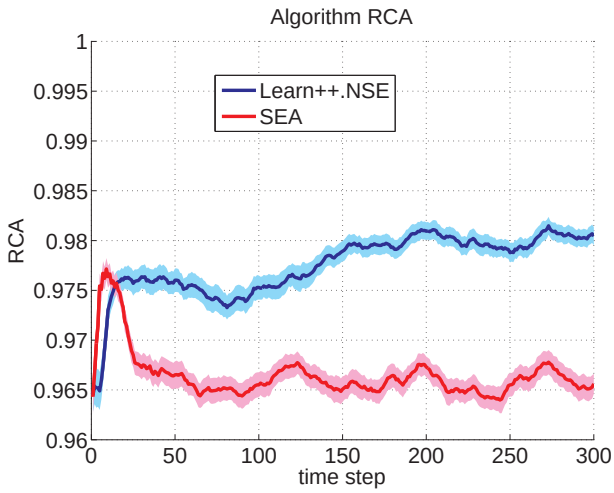


(d) G-mean

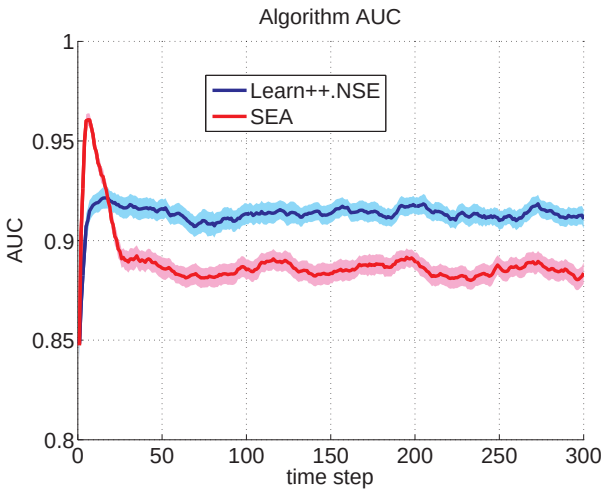


(e) Recall

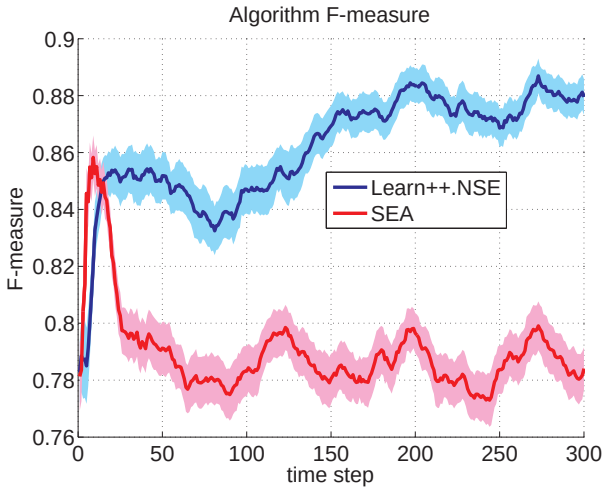
Fig. 5.13 : Learn^{++}.CDS evaluated on Spiral Data



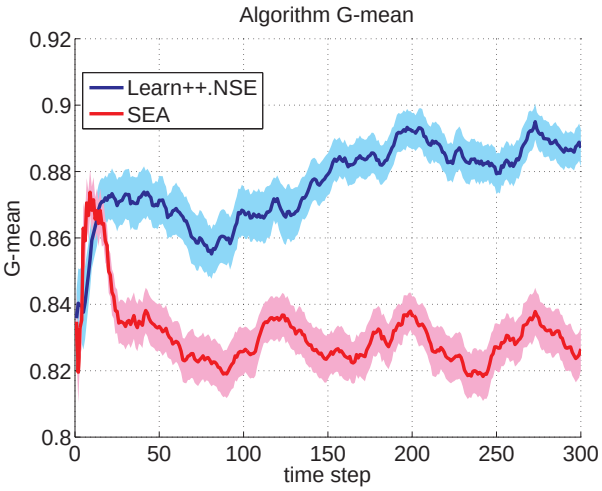
(a) Raw Classification Accuracy



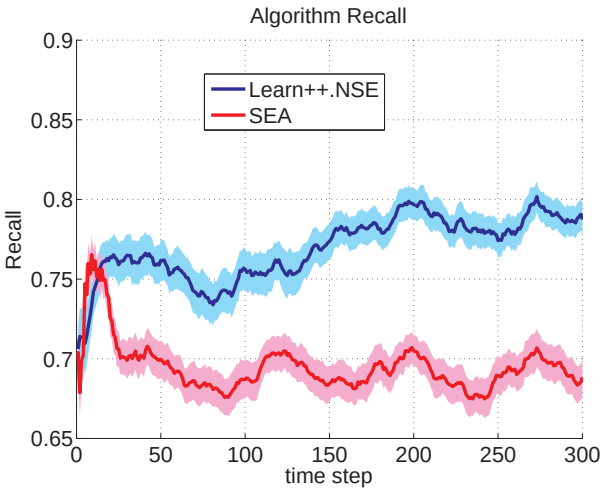
(b) AUC



(c) F-measure

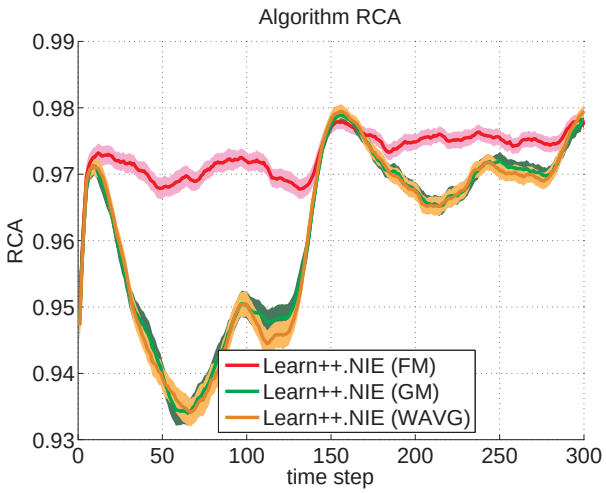


(d) G-mean

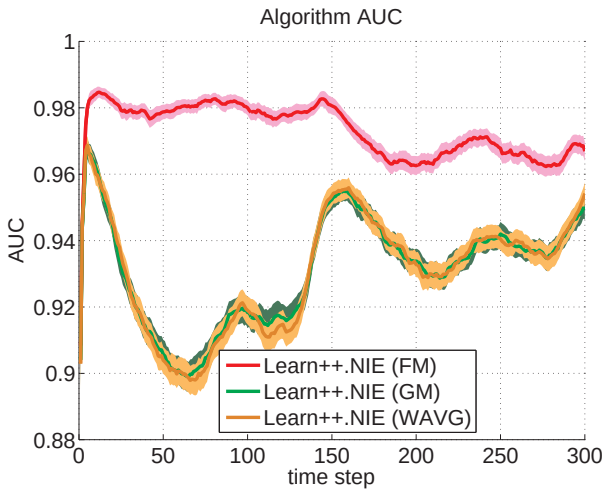


(e) Recall

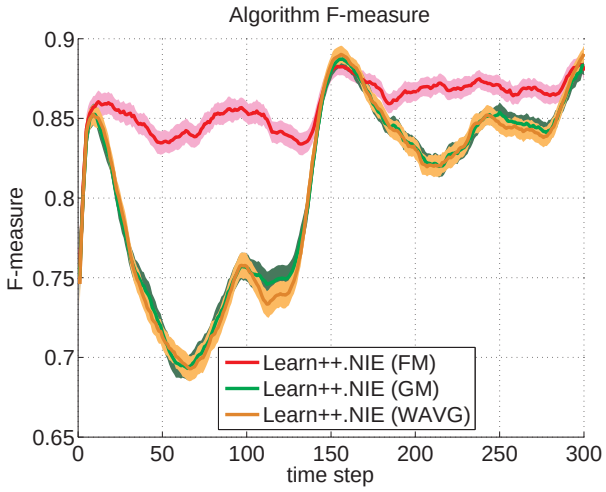
Fig. 5.14 : Learn^{++}.NSE and SEA evaluated on Spiral Data



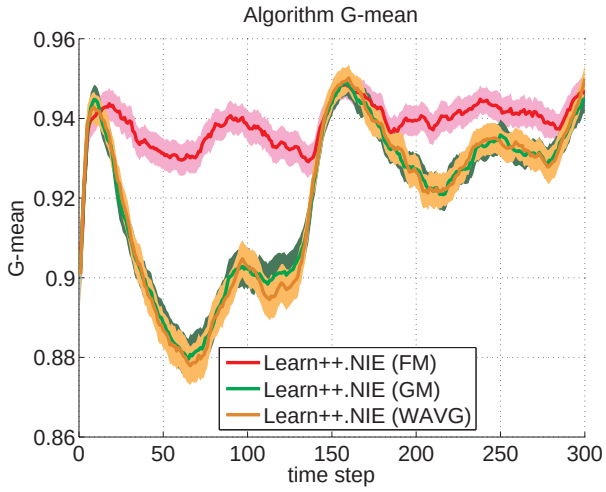
(a) Raw Classification Accuracy



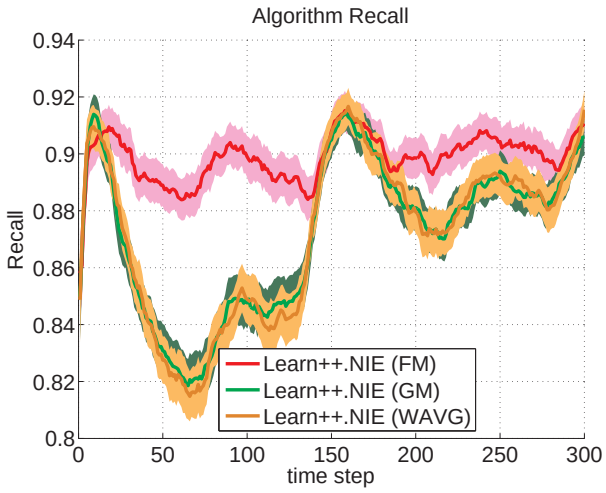
(b) AUC



(c) F-measure

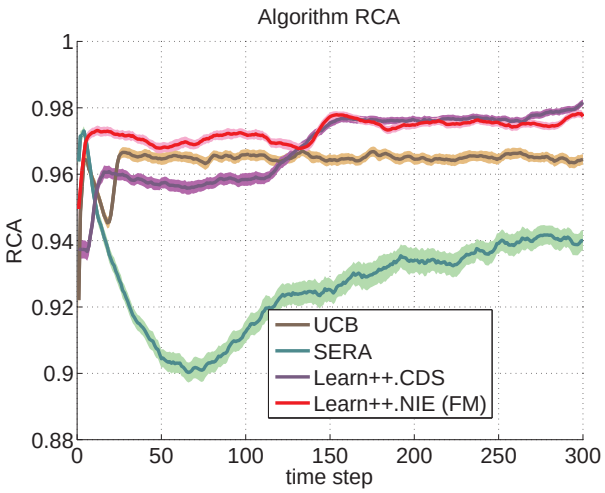


(d) G-mean

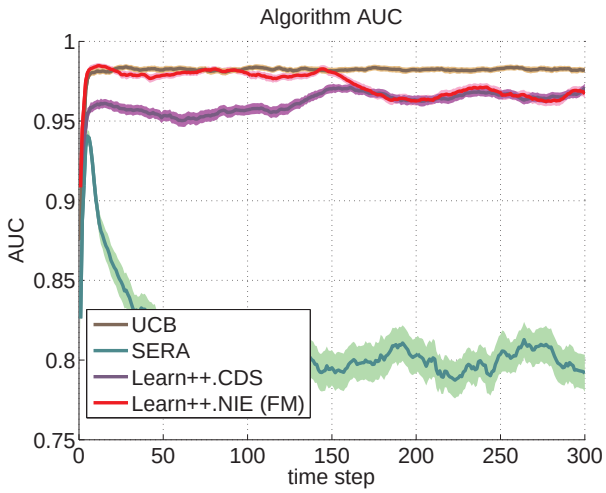


(e) Recall

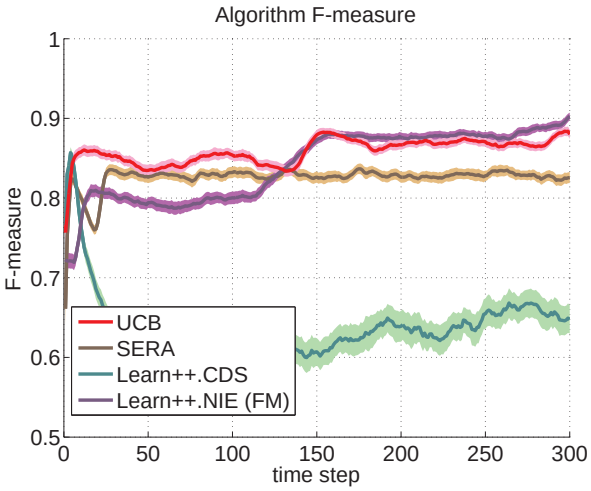
Fig. 5.15 : Learn^{++}.NIE family of algorithms evaluated on Spiral Data



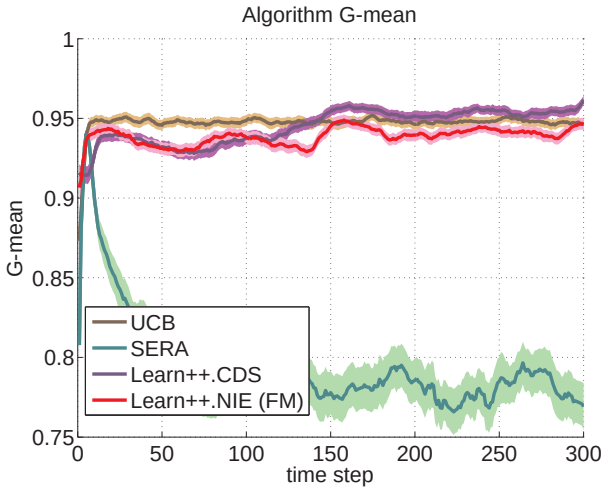
(a) Raw Classification Accuracy



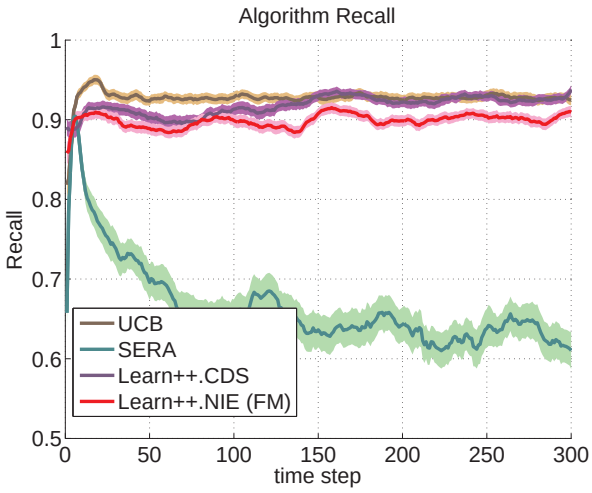
(b) AUC



(c) F-measure



(d) G-mean

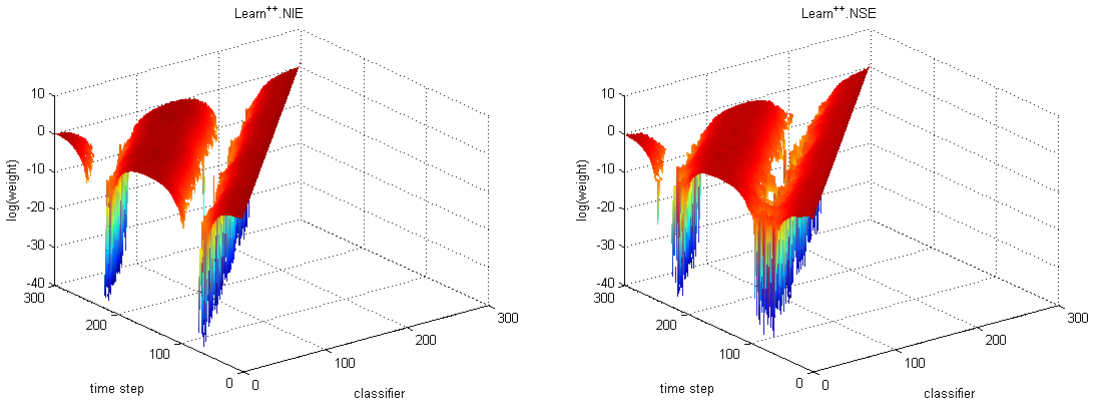


(e) Recall

Fig. 5.16 : Baseline algorithms evaluated on Spiral Data

Table 5.2 : Algorithm Summary on Spiral Data

	RCA	FM	AUC	Recall	OPM	Rank
L++.NSE	97.76±0.11(1)	86.13±0.76(1)	91.33±0.49(6)	76.96±1.17(6)	88.05±0.63(6)	3.5
SEA	96.65±0.12(4)	78.97±0.84(7)	88.91±0.50(7)	69.49±1.15(7)	83.51±0.65(7)	6.3
NIE(fm)	97.30±0.13(2)	85.87±0.65(2)	97.34±0.26(2)	89.87±0.73(3)	92.60±0.44(1)	2.3
NIE(gm)	96.11±0.16(6)	80.57±0.70(5)	93.11±0.38(4)	87.21±0.80(4)	89.25±0.51(4)	4.8
NIE($wavg$)	96.08±0.16(7)	80.46±0.70(6)	93.09±0.39(5)	87.20±0.80(5)	89.21±0.51(5)	5.8
L++.CDS	96.81±0.15(3)	84.15±0.65(3)	96.15±0.31(3)	91.77±0.71(2)	92.22±0.46(3)	2.8
SERA	92.73±0.32(8)	62.67±1.66(8)	80.96±1.10(8)	66.57±2.71(8)	75.73±1.45(8)	8
UCB	96.42±0.16(5)	82.57±0.69(4)	98.18±0.19(1)	92.74±0.65(1)	92.48±0.42(2)	2.8



(a) Learn^{++}.NIE (wavg) (b) Learn^{++}.NSE

Fig. 5.17 : Learn^{++} weight evolution on Spiral Data

feature of this dataset is that the mean as well as the covariance of the classes remain unchanged with drift. However, SERA's selection of minority class instances depends on (Mahalonobis-based) similarity measure. Since drift is present in the incremental learning problem and the mean of the minority data does not change, accumulated instances are select with the smallest distance when the selected instance may not be relevant to the most recent environment.

The weight evolution of Learn^{++}.NSE and Learn^{++}.NIE are shown in Fig. 5.17 for the rotating spiral dataset. There is little differentiation between the weights for Learn^{++}.NSE and Learn^{++}.NIE on this particular experiment. We shall simply note that both algorithms are capable of recalling environments on which the classifiers were trained on.

Table 5.3 : Mean and standard deviation Gaussian drift over time.

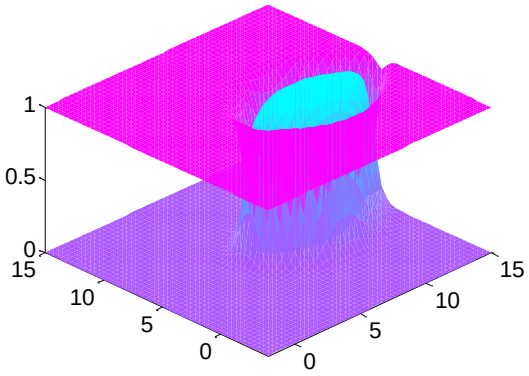
	$t=0$ to $t=1/3$				$t=1/3$ to $t=2/3$			
	σ_x	σ_y	μ_x	μ_y	σ_x	σ_y	μ_x	μ_y
$C_{1,1}$	1	$1+6t$	2	5	1	3	2	5
$C_{1,2}$	$3-6t$	1	5	8	1	1	$5+9(t-1/3)$	8
$C_{1,3}$	$3-6t$	1	5	2	1	1	$5+9(t-1/3)$	2
$C_{2,1}$	1	1	8	5	1	1	$8-9(t-1/3)$	5

	$t=2/3$ to $t=1$			
	σ_x	σ_y	μ_x	μ_y
$C_{1,1}$	1	$8-9(t-1/3)$	$8-9(t-1/3)$	$8-9(t-1/3)$
$C_{1,2}$	1	1	8	8
$C_{1,3}$	1	1	8	2
$C_{2,1}$	1	1	$8-9(t-1/3)$	$8-9(t-1/3)$

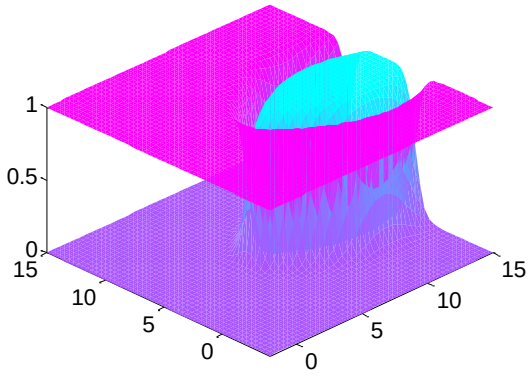
5.5.3 Gaussian Data

A Gaussian dataset is generated using a majority class containing a linear combination of three modes and a minority class with a single mode. Table 5.3 presents the parametric equations that govern the movement of the modes over $0 \leq t \leq 1$. The drift is controlled by varying the mean vectors and covariance matrices of each class. The off-diagonals of the covariance matrix for all classes are zero (i.e. features are independent of each other). The minority class is ≈3% of the training and testing datasets. The drift rate was set to 0.01, which corresponds to 100 time stamps. In this dataset, simply defaulting to the majority class yields ≈97% performance. Since this is a controlled experiment of relatively low dimension the Bayes classifier can easily be applied. The posterior probability of the Bayes classifier for the Gaussian dataset at six different time steps can be viewed in Fig. 5.18 where the z-axis represents the posterior probability. The light grey (cyan) colored areas of the feature space represent the posterior probability of the minority class. When $0 \leq t \leq t_1$, the means of the Gaussian modes remain constant while the covariance begins to drift. After a fixed amount of time, the means begin to drift and the location of the minority class eventually drifts between the center of the three majority class modes as shown in the posterior estimate of Fig. 5.18(a). Finally, the minority class mode moves out from the center of the three majority class modes.

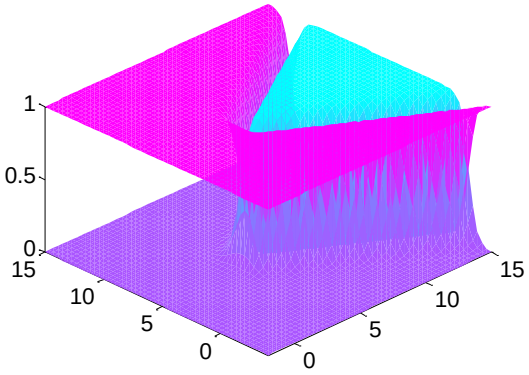
The results for the drifting Gaussians dataset are presented in Fig. 5.19, 5.20, and 5.21. The mean values of the figures of merits used in the evaluation are tabulated in



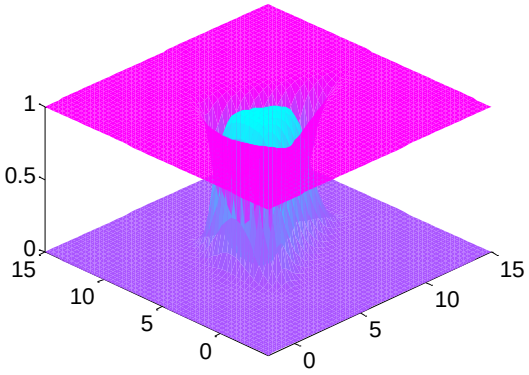
(a) time: $t = 1$



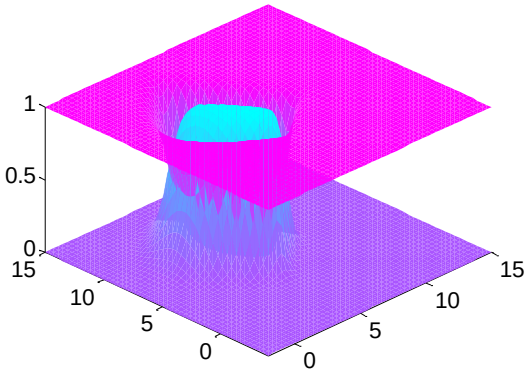
(b) time: $t = 20$



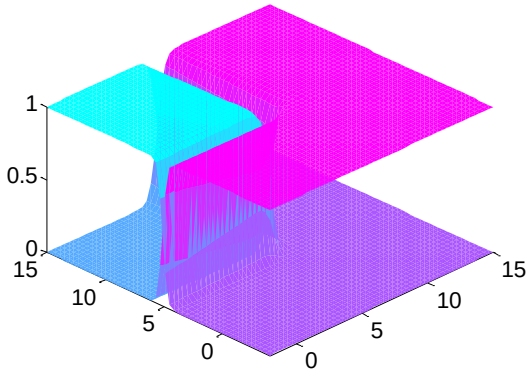
(c) time: $t = 40$



(d) time: $t = 60$



(e) time: $0 = 80$



(f) time: $t = 100$

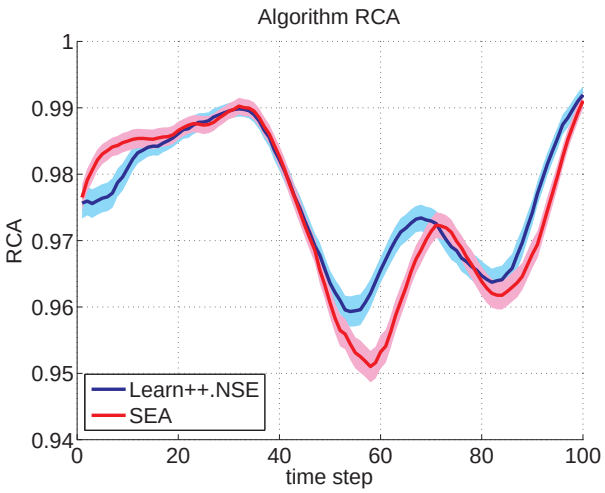
Fig. 5.18 : The pink region in the plots above indicate the posterior probability of the majority class ($p(-|\mathbf{x})$) and the cyan region indicates the posterior probability of the minority class ($p(+|\mathbf{x})$).

Table 5.4. There are several observations that can be made from these results. First, all algorithms experience a major drop in every measure around time step 40, which precisely corresponds to the minority class distribution moving into the middle of the majority class components, making prediction of minority class the most difficult. UCB appears to be the most robust in terms of recall and AUC, with Learn^{++}.NIE catching up in the latter part of the simulation. Learn^{++}.NIE (fm) maintains a better average rank (closely followed by Learn^{++}.CDS) compared to UCB, however, due to UCB's poor showing on RCA and F-measure. Specifically, the boost in minority class recall for UCB causes a large drop in RCA. While UCB has the best rank for minority class recall and AUC; it drops to rank 7 and 6 for RCA and F-measure, respectively. Second, Learn^{++}.CDS maintains a better rank compared to Learn^{++}.NSE and SEA in many of the measures in Table 5.4. This demonstrates the value and effectiveness of applying synthetic sampling (SMOTE) to Learn^{++}.NSE for learning unbalanced classes. Third, Learn^{++}.NIE and Learn^{++}.CDS algorithms are highly competitive with one another in terms of combined figure of merit, the overall performance measure (OPM). Fourth, while SERA does well on F-measure, but it performs rather poorly on other figures of merit. Hence, Learn^{++}.NIE (all implementations) and Learn^{++}.CDS out-rank SERA in terms of the OPM. Finally, Learn^{++}.NIE (fm) out ranks (gm) and ($wavg$) implementations (refer to Fig. 5.20 and Table 5.4).

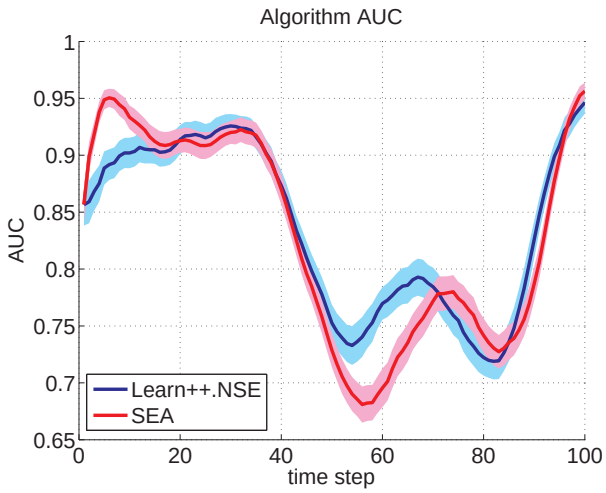
Fig. 5.22 compares the weights assigned to the classifiers for Learn^{++}.NIE ($wavg$) and Learn^{++}.NSE. Notice that the weight of a classifier is always the largest when the classifier was created then the weights begin to vary (generally decrease) as time goes on. Near time $t = 45$, we observe a decrease in classifiers 1-40 weight. Classifiers trained at a later time remain relevant for a shorter period of time after they are generated because of the nonstationarity of the data.

5.5.4 Shifting Hyperplane

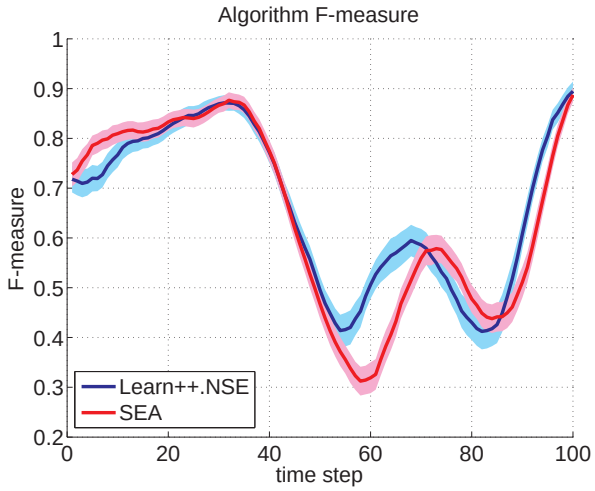
The shifting hyperplane dataset is derived from the original SEA experiment presented by Street & Kim [49]. The original dataset consist of three dimensions only two of which carry information relevant to the classification problem. Each instance is generated from a random number between zero and ten using Eq. (5.11). The class label for each feature only depends on features 1 and 2. This label is computed using Eq. (5.12). The parameter



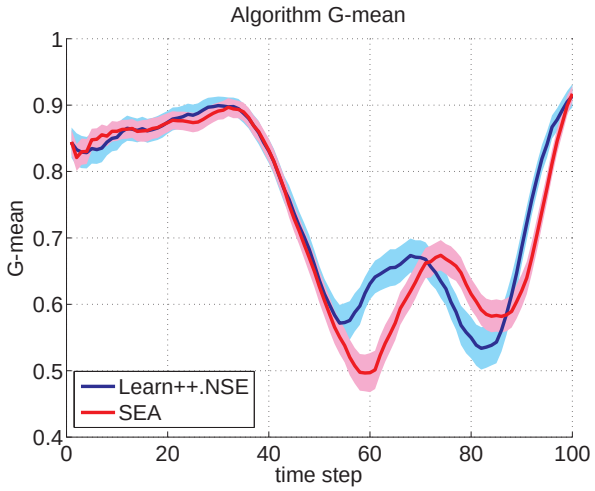
(a) Raw Classification Accuracy



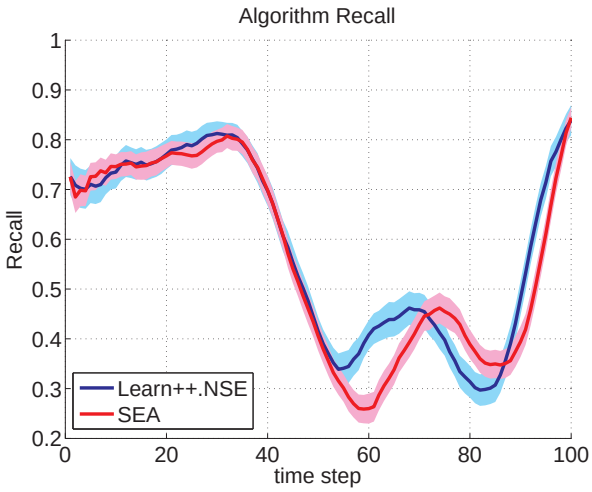
(b) AUC



(c) F-measure

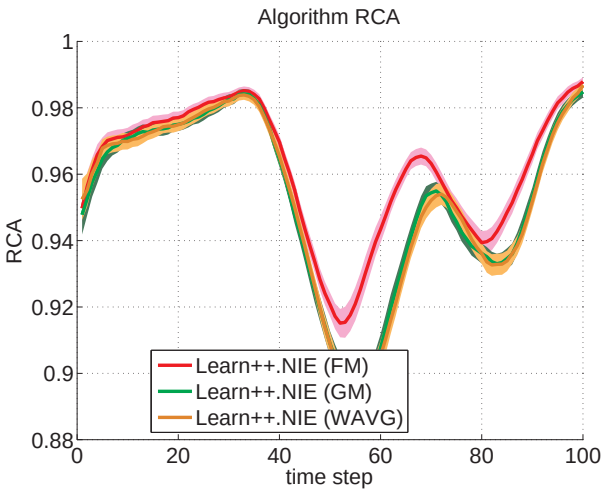


(d) G-mean

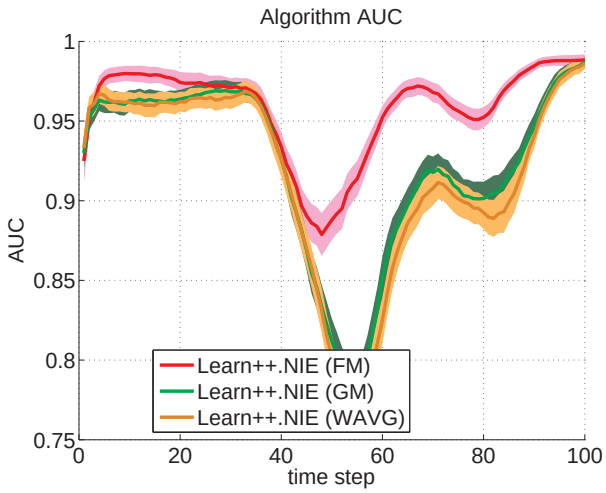


(e) Recall

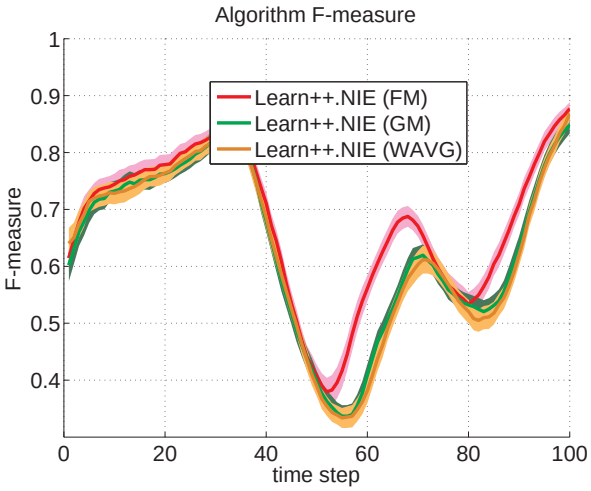
Fig. 5.19 : Learn^{++}.NSE and SEA evaluated on Gaussian data



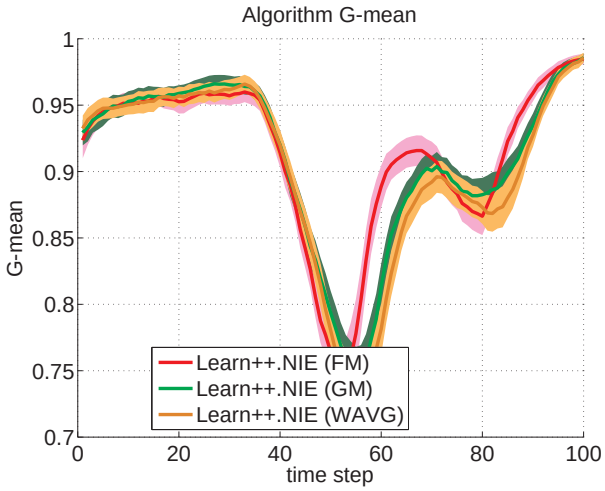
(a) Raw Classification Accuracy



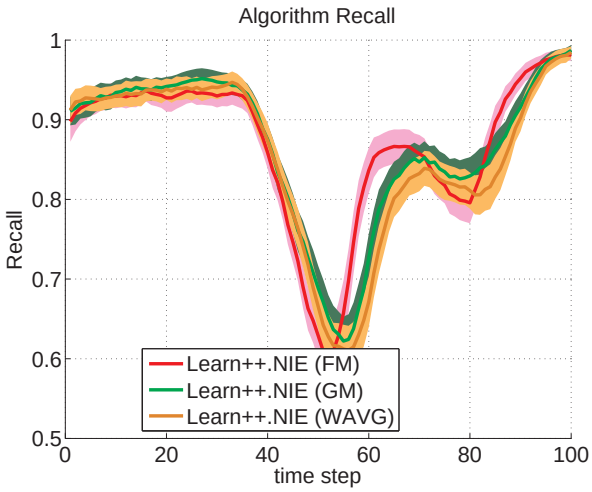
(b) AUC



(c) F-measure

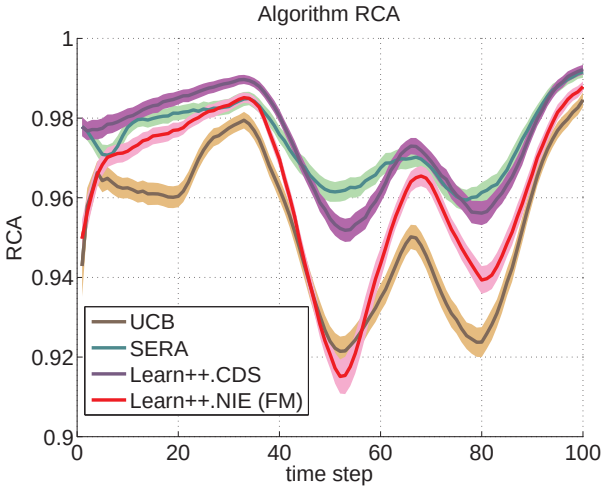


(d) G-mean

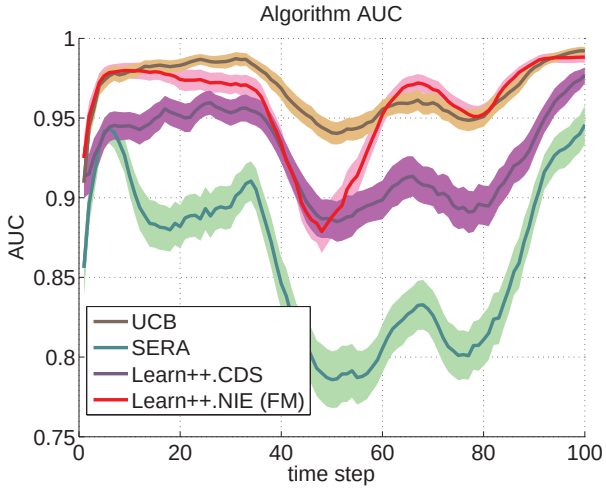


(e) Recall

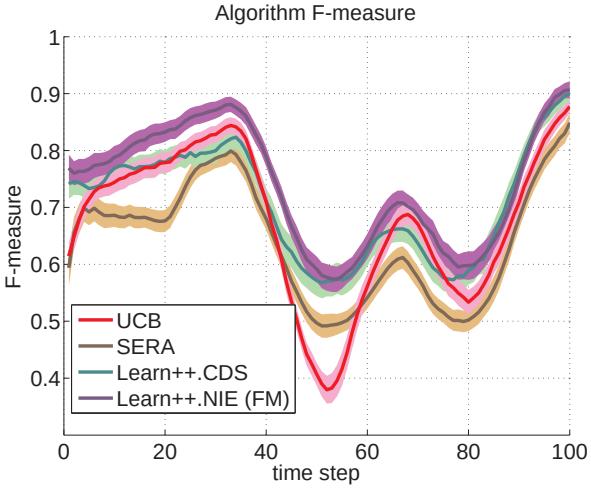
Fig. 5.20 : Learn^{++}.NIE family of algorithms evaluated on Gaussian data



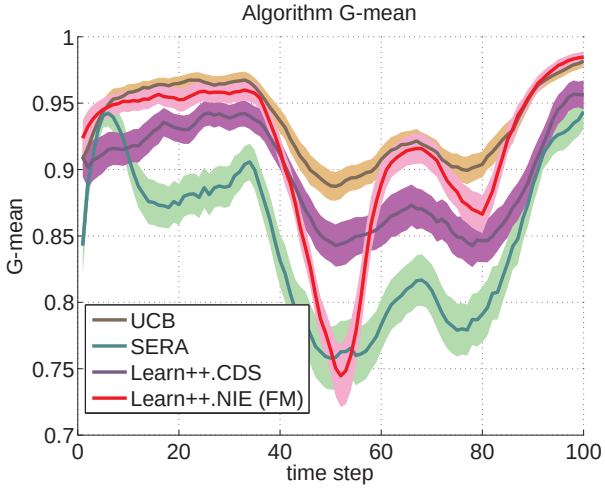
(a) Raw Classification Accuracy



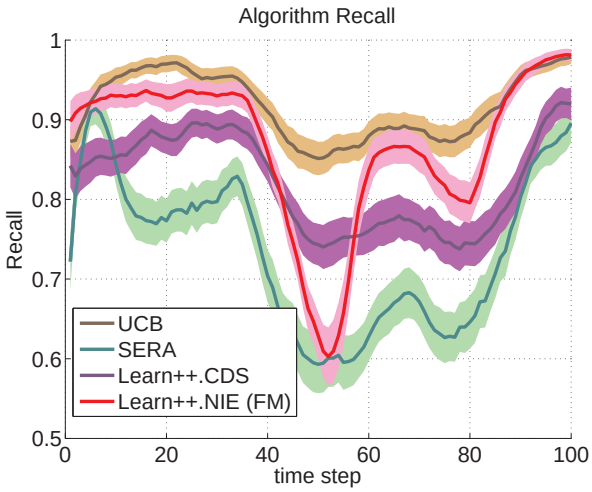
(b) AUC



(c) F-measure

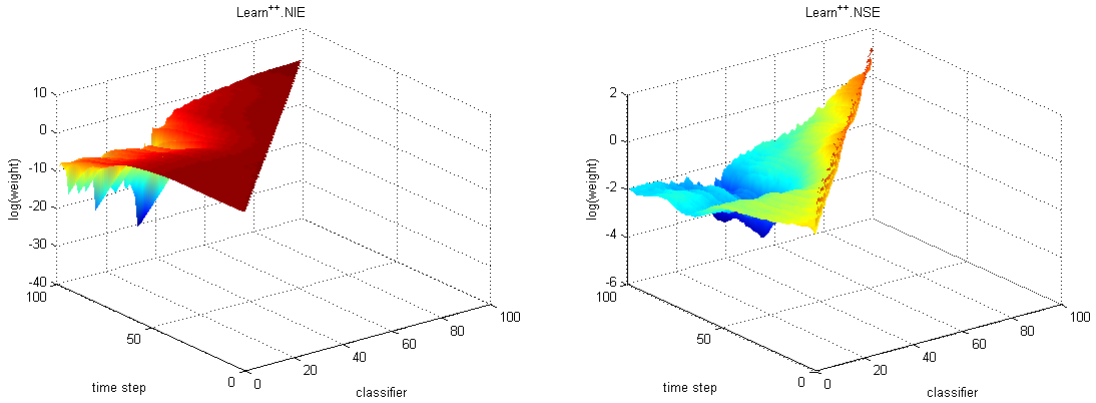


(d) G-mean



(e) Recall

Fig. 5.21 : Baseline algorithms evaluated on Gaussian data



(a) Learn^{++}.NIE (wavg) (b) Learn^{++}.NSE

Fig. 5.22 : Learn^{++} weight evolution on Gaussian concept drift problem

Table 5.4 : Algorithm Summary on Gaussian Data

	RCA	**FM**	**AUC**	**Recall**	**OPM**	**Rank**
L++.NSE	97.63±0.18(1)	66.30±2.62(4)	83.65±1.43(7)	58.33±3.15(7)	76.48±1.85(7)	4.8
SEA	97.46±0.18(3)	64.39±2.44(5)	82.97±1.31(8)	56.40±2.84(8)	75.31±1.69(8)	6
NIE(fm)	96.11±0.27(5)	67.30±1.00(3)	95.80±0.67(2)	86.74±2.01(2)	86.49±0.99(2)	3
NIE(gm)	95.24±0.27(6)	63.37±1.86(7)	92.12±0.89(4)	86.51±1.90(3)	84.31±1.23(4)	5
NIE($wavg$)	95.20±0.28(8)	62.93±1.91(8)	91.60±0.94(5)	85.42±1.97(4)	83.79±1.28(5)	6.3
L++.CDS	97.50±0.20(2)	74.21±1.90(1)	92.19±1.07(3)	80.85±2.45(5)	86.19±1.41(3)	2.8
SERA	97.37±0.22(4)	70.76±2.28(2)	85.99±1.46(6)	73.52±2.96(6)	81.91±1.73(6)	4.5
UCB	95.22±0.30(7)	63.74±1.94(6)	96.84±0.54(1)	92.02±1.56(1)	86.96±1.09(1)	3.8

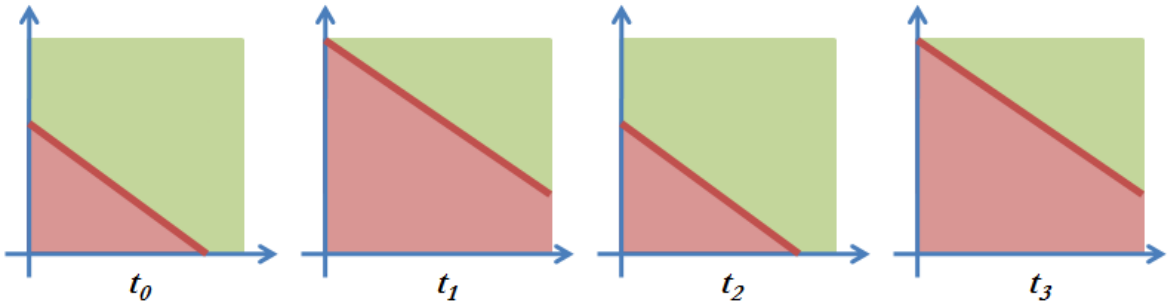


Fig. 5.23 : Demonstration of the shifting hyperplane problem presented by Street[49].

θ controls the location of the hyperplane. The location of the hyperplane is only computed using $x_i(1)$ and $x_i(2)$, leaving the 3^{rd} feature as random noise and carries no information for classification. The location of the hyperplane remains constant for 50 time stamps and abruptly changes. In addition, 10% noise is added to training/testing data.

$$x_i = 10 \times \texttt{rand}(3,1) \tag{5.11}$$

$$y_i = \begin{cases} 1 & \text{if } x_i(1) + x_i(2) \leq \theta; \\ 2 & \text{otherwise.} \end{cases} \tag{5.12}$$

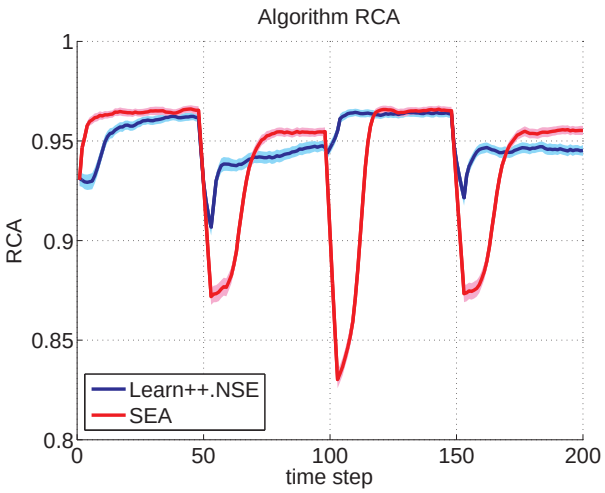
Fig. 5.23 provides a graphical representation of features 1 and 2 over the course of the experiment. The data contain three abrupt locations of *concept change*. The location of the hyperplane switches from $\theta = \{7, 9, 8, 9.5\}$ with the abrupt points of change occurring at $t = \{50, 100, 150\}$. The prior probabilities of classes ω_1 and ω_2 vary slightly over time, however some software suites provide options to balance the training/testing datasets during the duration of the experiment [118].

The generation of the shifting hyperplane dataset has been modified to be a recurring environment problem as well as containing an imbalanced class distribution unlike the original dataset in [49]. The hyperplane shifts back and forth between two locations with the class imbalance varying as the plane shifts ($\theta = \{4, 7, 4, 7\}$). The imbalance changes between $\approx$ 7% and $\approx$ 25% depending on the location of the hyperplane. The dataset contains 5% noise and was presented to the learning algorithms incrementally for 200 time steps with three different shifts occurring ever 50 time steps similar to the original dataset.

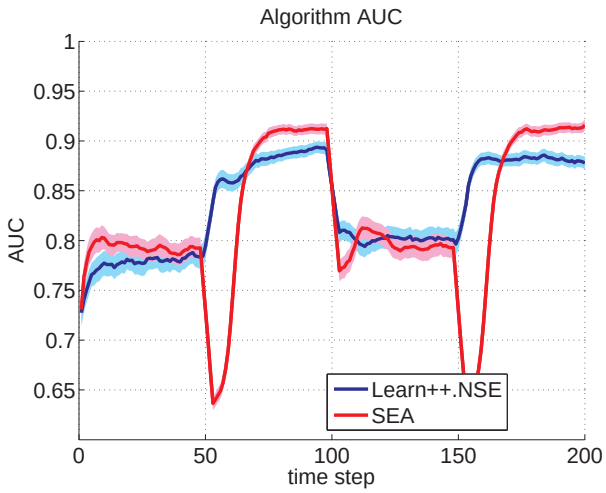
Each dataset contains 1000 instances for training and testing.

Fig. 5.24, 5.25, and 5.26 present the results on the shifting hyperplane problem (the standard benchmark for the SEA algorithm). The mean values of all figures of merits used in the evaluation are tabulated in Table 5.5. First, Learn^{++}.NIE (all implementations) significantly outperform Learn^{++}.NSE and SEA in terms of recall. Because this is an abrupt, sudden concept change problem, rather than a gradual concept drift problem, of primary importance is the speed of recovery after the sudden change, as well as maintaining a high performance during the steady state periods. From Fig. 5.25(e) it is observed that Learn^{++}.NIE (fm) maintains the best speedy recovery from a sudden change and a high steady state performance in comparison to other implementations of Learn^{++}.NIE. Also, Learn^{++}.NIE (fm) does not incur a sudden drop in recall when the hyperplane shifts when compared to other Learn^{++}.NIE implementations. Second, Learn^{++}.CDS and Learn^{++}.NIE (all implementations) are the top ranking algorithms in terms of the OPM and mean rank. Third, SERA maintains a constant RCA for nearly all time stamps as shown in Fig. 5.26(a). This is a result of SERA not using classifiers from previous time stamps, thus there is no prior knowledge retained about the majority class. Finally, the boost in minority class recall for UCB comes at the cost of the overall accuracy and F-measure.

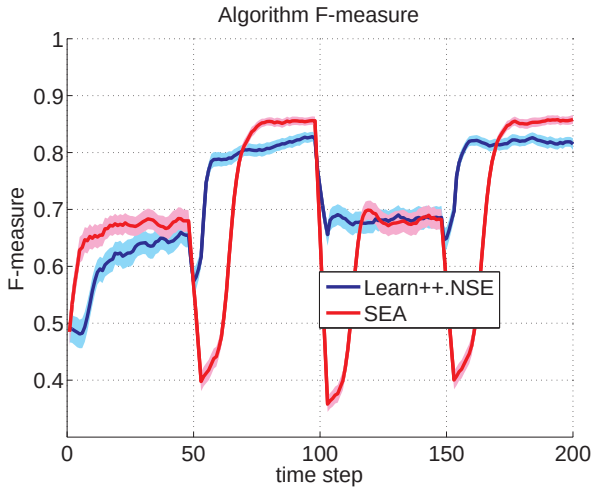
The classifier weights as a function of time for Learn^{++}.NIE and Learn^{++}.NSE are shown in Fig. 5.27. Several interesting observations can be made from these plots. First, Learn^{++}.NIE and Learn^{++}.NSE demonstrate the ability to use prior knowledge about previous environments to effectively decrease the weights of irrelevant classifiers in the ensemble. Secondly, Learn^{++}.NIE appears to have a flat response when the hyperplane shifts whereas Learn^{++}.NSE observes an abrupt drop in classifier weight then increasing slightly before reaching a flat response. This phenomenon using a simple example and knowing a little bit about the instance weighting scheme. First, assume that one can build the optimal classifier that learns the decision boundary in Fig. 5.23 at time stamp t_0. At some later point in time, t_1, the hyperplane shifts upwards. Assuming an optimal classifier, the new data that fall in between the optimal decision boundary at t_0 and t_1 will be misclassified by the classifiers generated before t_1. Now, when Learn^{++}.NSE uses the "old" ensemble to classify the new data at t_1, the instances that fall in the purple region of Fig. 5.28 will be misclassified by $H^{(t-1)}$. Learn^{++}.NSE then updates the distribution



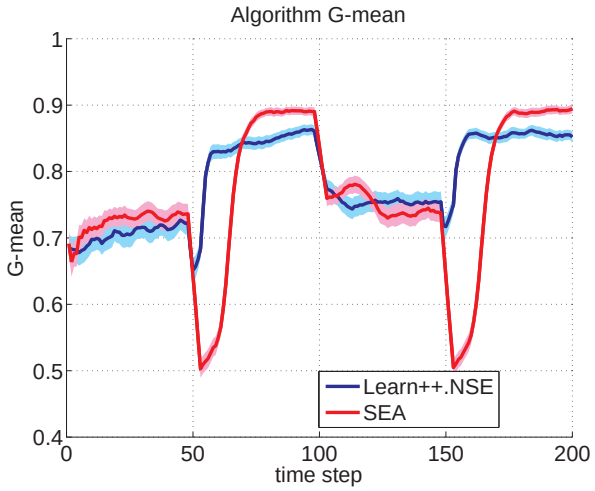
(a) Performance



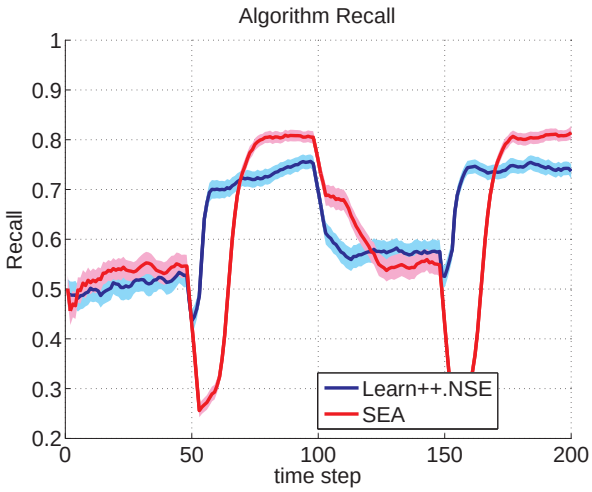
(b) AUC



(c) F-measure

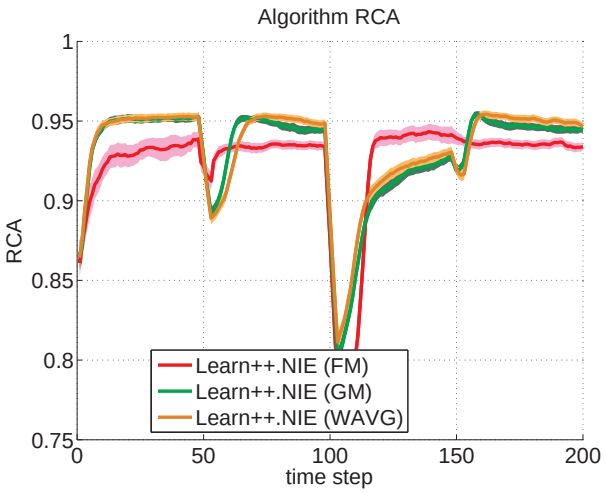


(d) G-mean

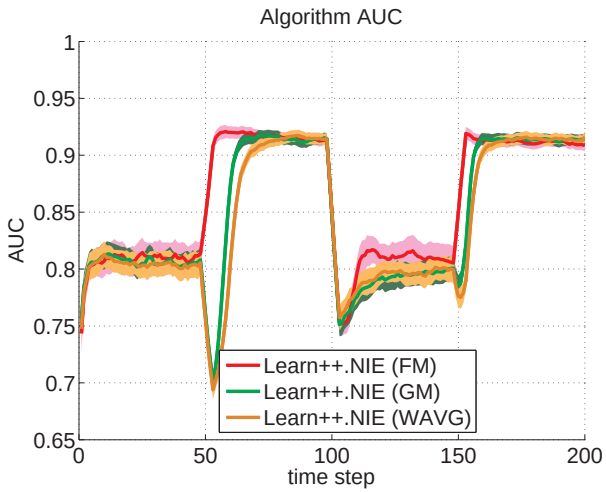


(e) Recall

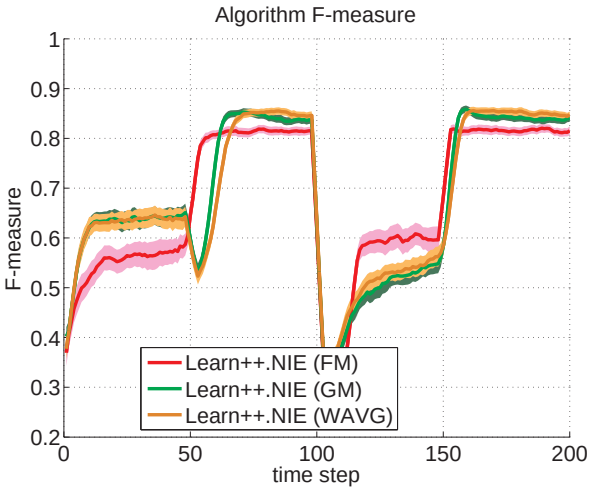
Fig. 5.24 : Learn^{++}.NSE and SEA evaluated on a Shifting Hyperplane Dataset



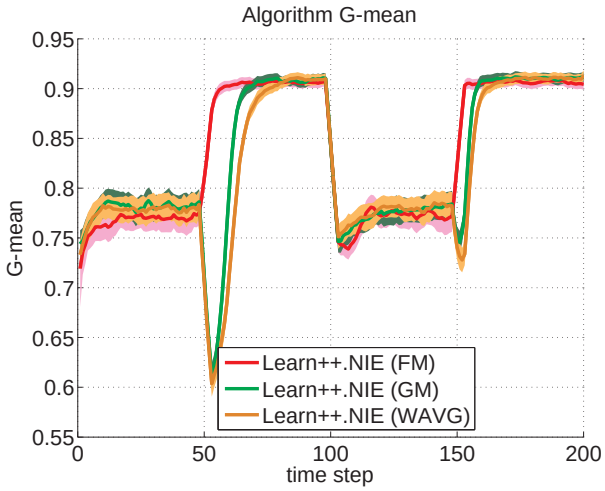
(a) Performance



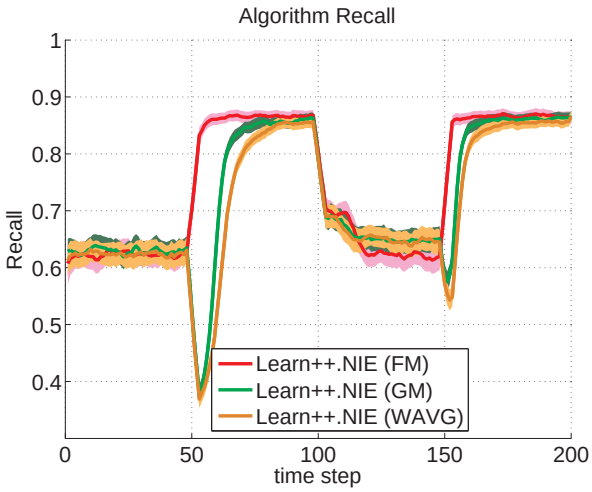
(b) AUC



(c) F-measure

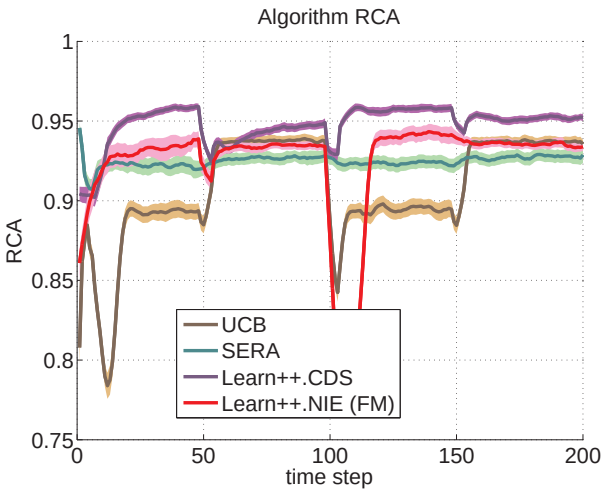


(d) G-mean

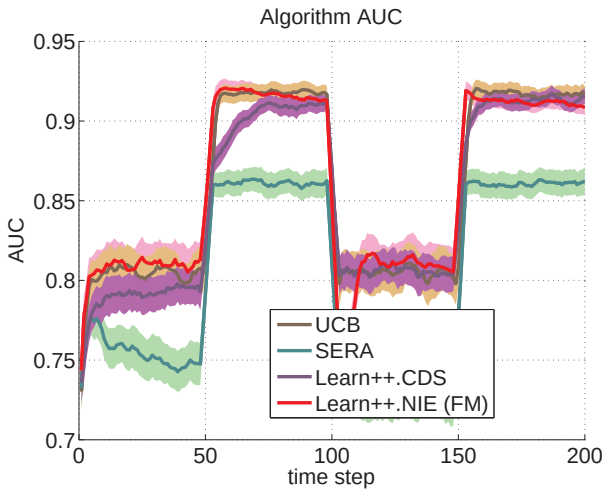


(e) Recall

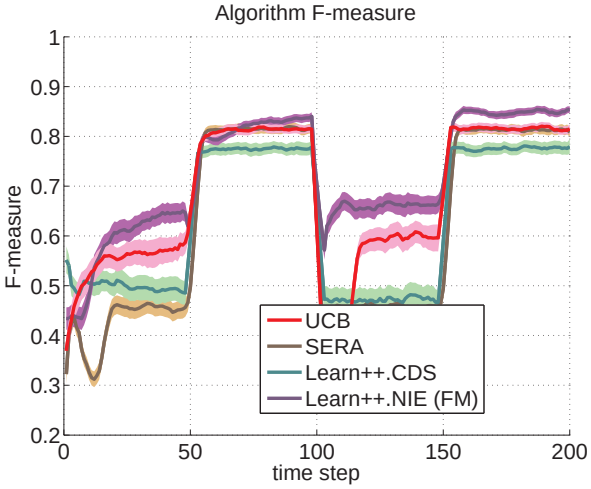
Fig. 5.25 : Learn^{++}.NIE family of algorithms evaluated on a Shifting Hyperplane Dataset



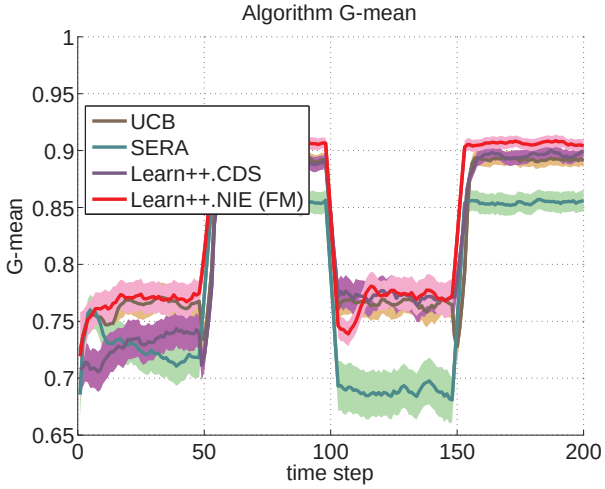
(a) Performance



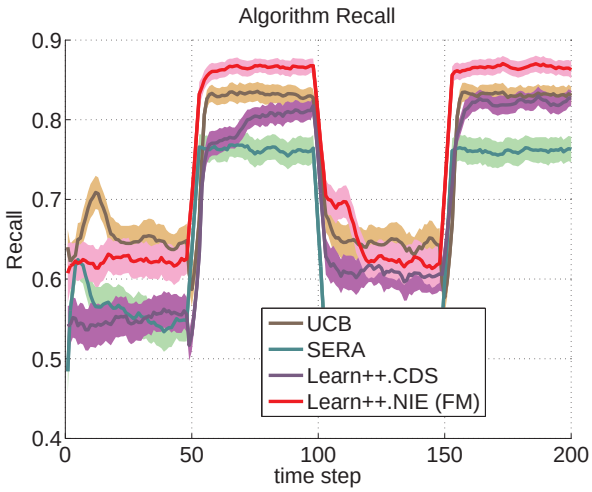
(b) AUC



(c) F-measure

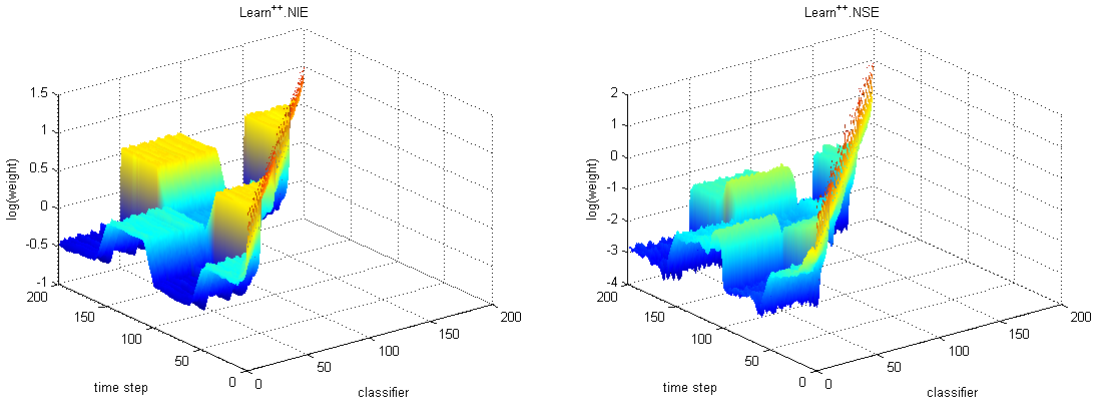


(d) G-mean



(e) Recall

Fig. 5.26 : Baseline algorithms evaluated on a Shifting Hyperplane Dataset



(a) Learn^{++}.NIE (wavg) (b) Learn^{++}.NSE

Fig. 5.27 : Learn^{++} weight evolution on SEA

of instance weights. Thus, the sum of the instances that fall into the green region is 1/2 and the sum the instances in the purple region will be 1/2 (even though there are fewer instances in this region).

$$\sum_{i=1}^{m^{(t)}} D^{(t)}[\![H^{(t-1)}(\mathbf{x}_i) \neq y_i]\!] = \frac{1}{2} \tag{5.13}$$

After several time stamps, the Learn^{++}.NSE ensemble will begin to correctly label the instances that fall into the purple region because the algorithm is learning the new environment. Since the ensemble is classifying the instances in the purple correctly, the penalty assigned to the classifiers that incorrectly classify instances will receive a smaller penalty then the time stamps just after the plane shift. Thus, Learn^{++}.NSE reduces the penalty applied to irrelevant classifiers as the ensemble correctly classifies instances from a new dataset.

5.6 Real-World Data

5.6.1 Electricity Pricing

The electricity pricing data is presented in the original Splice-2 paper [120] and has been used as a benchmark for concept drift problems [63, 28]. This dataset is a sequence of information related to time and demand fluctuations in the price of electricity in New

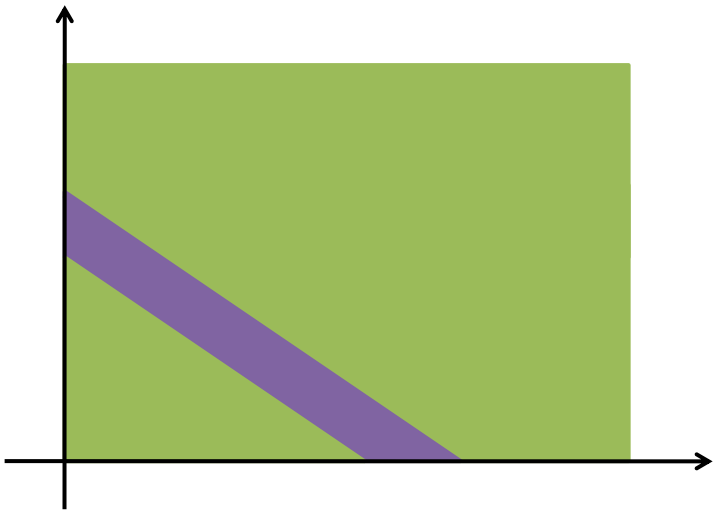


Fig. 5.28 : Learn^{++}.NSE weight distribution

Table 5.5 : Algorithm Summary on Shifting Hyperplane Data

	RCA	**FM**	**AUC**	**Recall**	**OPM**	**Rank**
L++.NSE	94.98±0.26(1)	71.98±1.57(2)	83.30±0.90(6)	62.87±1.96(7)	78.28±1.17(5)	4
SEA	94.00±0.26(3)	68.13±1.48(3)	82.00±0.85(7)	60.28±1.77(8)	76.10±1.09(7)	5.3
NIE(fm)	92.38±0.46(7)	67.27±1.62(6)	85.93±0.90(1)	74.83±1.60(1)	80.10±1.15(2)	3.8
NIE(gm)	93.03±0.31(5)	67.90±1.36(5)	84.51±0.81(4)	72.17±1.61(3)	79.40±1.02(3)	4.3
NIE($wavg$)	93.25±0.30(4)	67.94±1.39(4)	84.08±0.93(5)	70.65±1.65(4)	78.98±1.07(4)	4.3
L++.CDS	94.75±0.28(2)	72.24±1.46(1)	85.16±0.84(3)	68.80±1.79(5)	80.24±1.09(1)	2.8
SERA	92.47±0.44(6)	63.01±1.84(7)	80.11±1.08(8)	64.68±2.17(6)	75.07±1.38(8)	6.8
UCB	90.77±0.45(8)	62.05±1.44(8)	85.84±0.95(2)	73.34±1.66(2)	78.00±1.13(6)	5

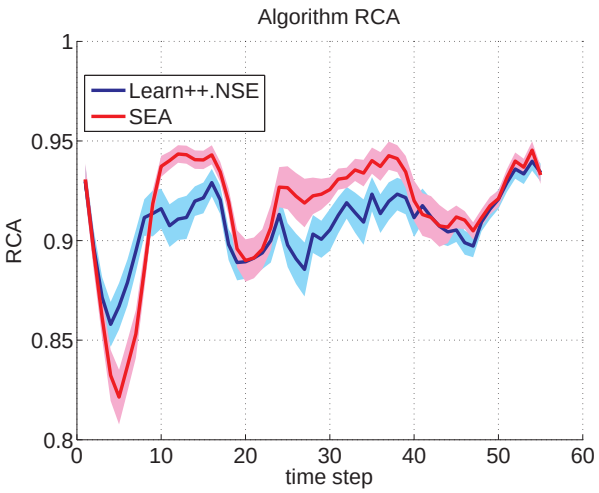
South Wales, Australia. The day, period, New South Whales (NSW) electricity demand, Victoria (VIC) electricity demand and the scheduled electricity transfer between the two states are used as features. All instances containing missing features have been removed from the database. The original dataset does not contain class imbalance so one of the classes was under sampled to create and imbalance ratio of approximately 1:18.

Fig. 5.29, 5.30, and 5.31 present the results of all algorithms on the electricity pricing dataset. The SMOTE percentage was set to 1500% for this experiment. The estimated mean of the figures on merit used in the evaluation have been tabulated in Table 5.6. First, Learn^{++}.NIE or Learn^{++}.CDS does not maintain the best OPM; however, they are high in the ranking of OPM following UCB. Second, UCB's strong recall measure comes at the cost of the overall accuracy whereas our proposed approaches have been able to find a well-suited balance between several measures. SERA and UCB are performing significantly lower than any of the algorithms in the Learn^{++} family (NSE, NIE or CDS) or SEA in terms of RCA (refer to Fig. 5.29(a), 5.30(a) and 5.31(a)). Third, all implementations of Learn^{++}.NIE perform approximately the same on this particular data set. SEA performs quite poorly on the minority class while it maintains the best RCA, but minority class recall of SEA nearly drops to 0% because there is no mechanism in SEA to enable the learning of an under-represented class.

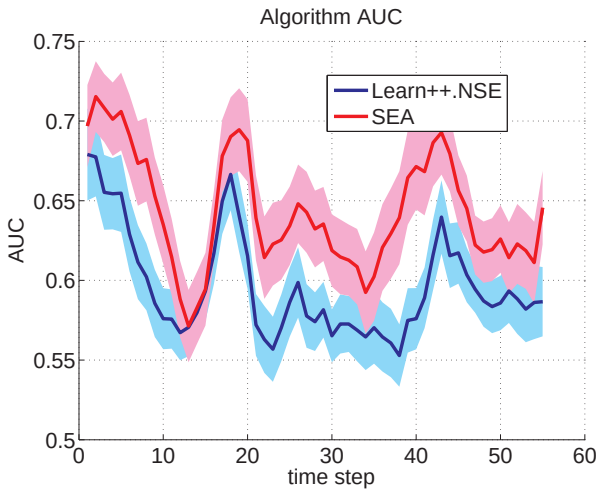
The variation of the classifier voting weights is quite clear and easy to interpret with the synthetic data problems because the generation of the data and the parameters of the experiment are controlled. However, the interpretation of the weight variation can become more complicated when we are facing real-world data like the electricity pricing dataset. Fig. 5.32 shows the variation of Learn^{++}.NIE classifier's voting weight variation with time.

5.6.2 NOAA

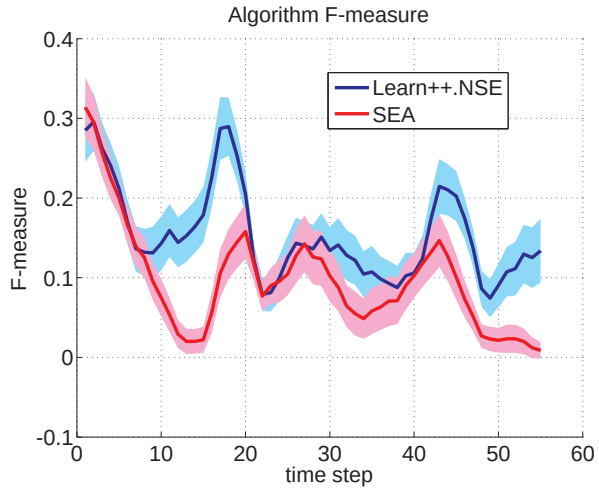
The NOAA dataset contains approximately 50 years of meteorological data obtained from a post at Offutt Air Force Base in Bellevue, Nebraska [47]. Daily measurements were taken for a variety of features like temperature, pressure, visibility, wind speed, etc. The number of features were reduced to eight features (average/minimum/maximum temperature, dew point, sea level pressure, visibility, average/maximum wind speed) and the classification task was to predict whether or not it rained on a particular day. It can be shown that the



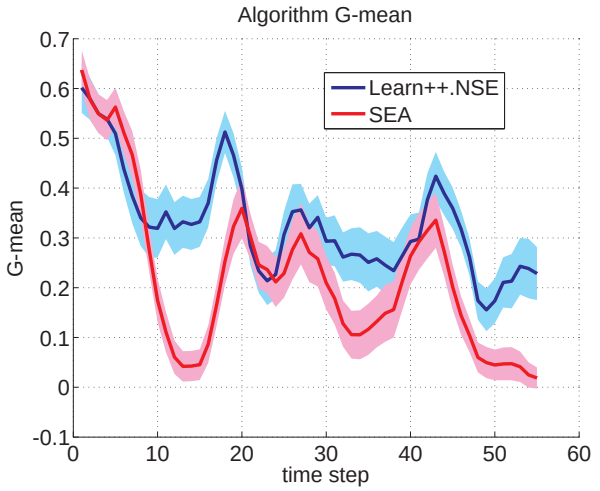
(a) Raw Classification Accuracy



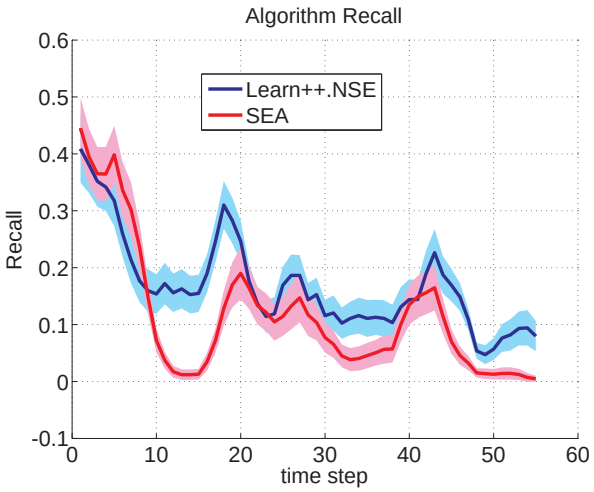
(b) AUC



(c) F-measure

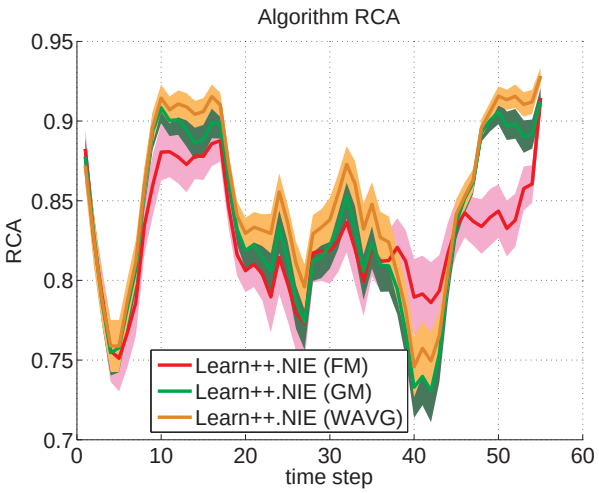


(d) G-mean

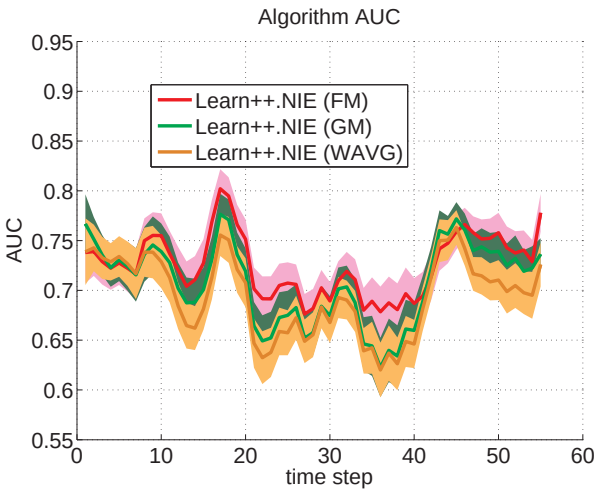


(e) Recall

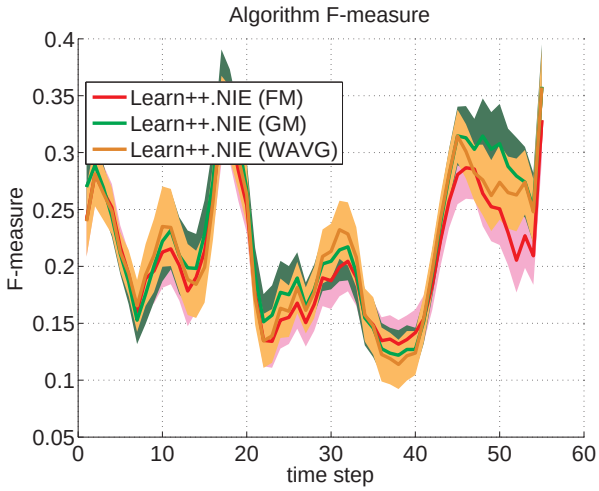
Fig. 5.29 : Learn^{++}.NSE and SEA evaluated on the Elec2 Dataset



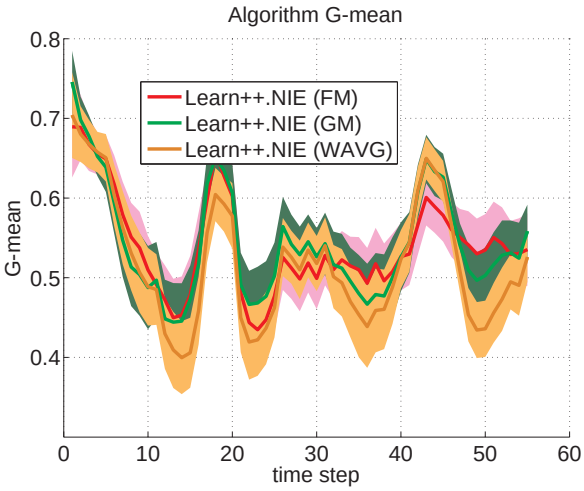
(a) Raw Classification Accuracy



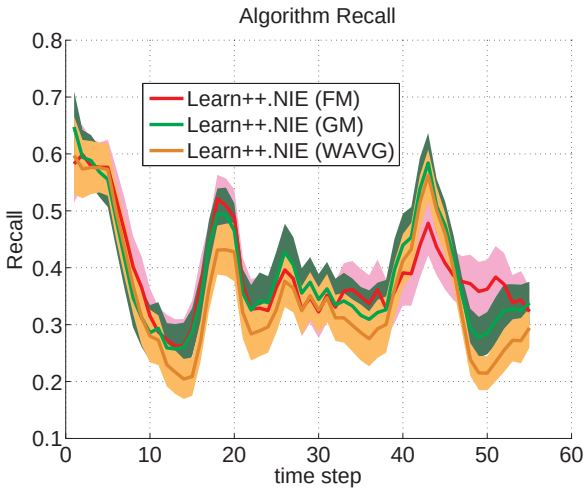
(b) AUC



(c) F-measure

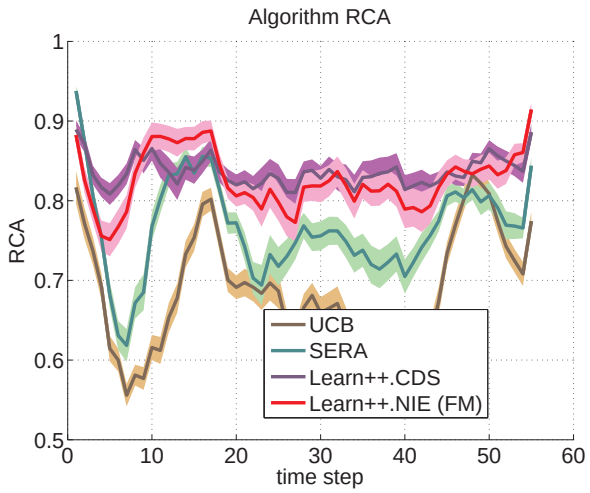


(d) G-mean

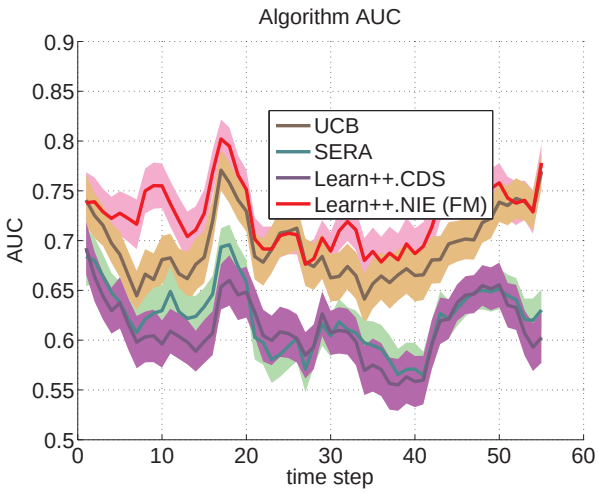


(e) Recall

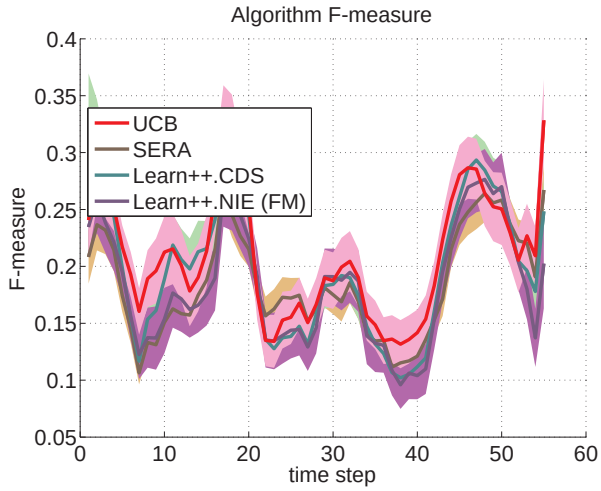
Fig. 5.30 : Learn^{++}.NIE family of algorithms evaluated on the Elec2 Dataset



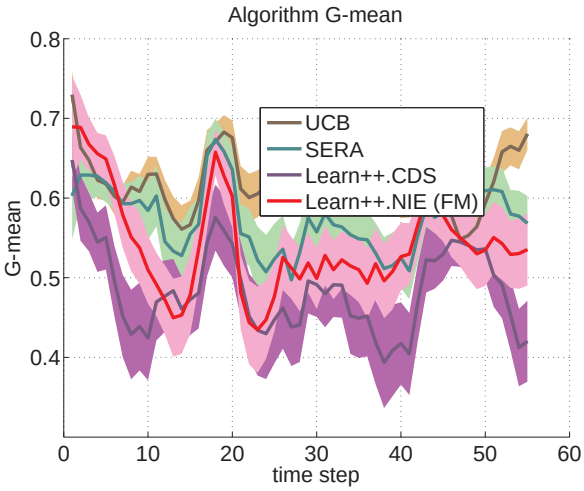
(a) Raw Classification Accuracy



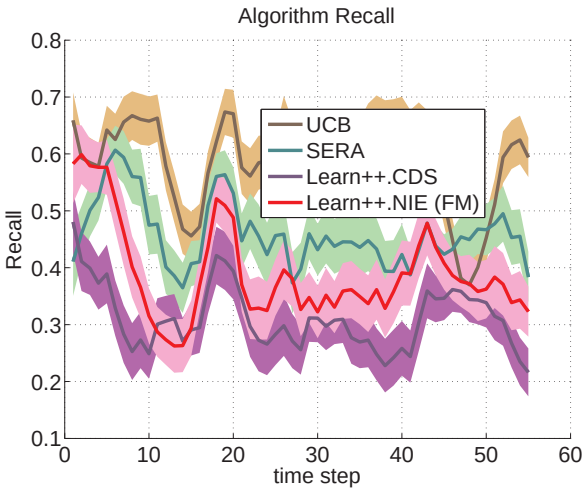
(b) AUC



(c) F-measure

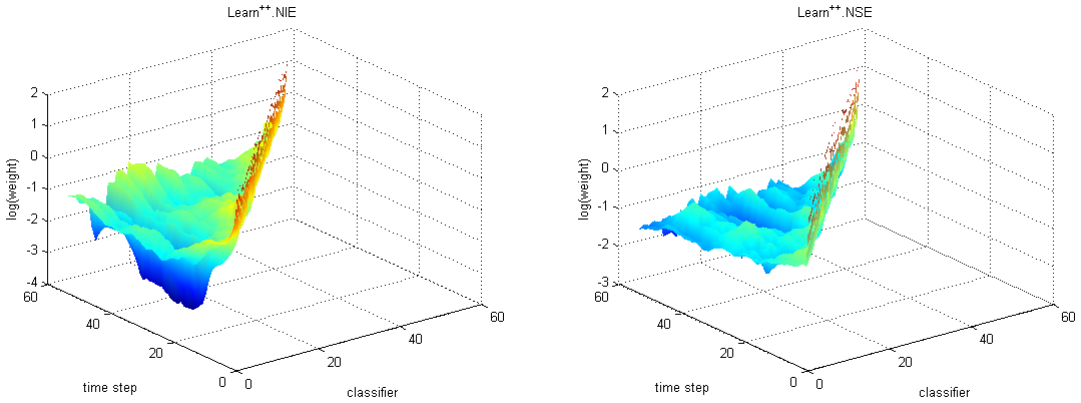


(d) G-mean



(e) Recall

Fig. 5.31 : Baseline algorithms evaluated on the Elec2 Dataset



(a) Learn^{++}.NIE (wavg) (b) Learn^{++}.NSE

Fig. 5.32 : Learn^{++} weight evolution on Elec2

Table 5.6 : Algorithm Summary on Electricity Pricing Data

	RCA	**FM**	**AUC**	**Recall**	**OPM**	**Rank**
L++.NSE	90.75±0.86(2)	15.40±3.05(7)	59.66±2.04(7)	16.87±3.31(7)	45.67±2.32(7)	5.8
SEA	92.15±0.60(1)	9.37±2.15(8)	58.48±1.55(8)	10.53±2.19(8)	42.63±1.62(8)	6.3
NIE(fm)	82.60±1.80(6)	20.79±2.55(3)	72.45±2.15(1)	38.72±4.93(3)	53.64±2.86(3)	3.3
NIE(gm)	83.60±1.30(5)	22.29±2.64(1)	70.70±2.34(2)	38.37±4.68(4)	53.74±2.74(2)	3
NIE($wavg$)	84.70±1.15(4)	21.88±2.61(2)	69.54±2.23(4)	35.61±4.28(5)	52.93±2.57(4)	3.8
L++.CDS	88.48±1.12(3)	18.09±3.05(6)	60.58±2.27(6)	22.91±4.07(6)	47.52±2.63(6)	5.3
SERA	76.42±1.70(7)	19.91±2.06(4)	62.42±2.22(5)	46.46±4.70(2)	51.30±2.67(5)	4.5
UCB	68.23±1.72(8)	18.68±1.75(5)	69.74±2.34(3)	58.87±4.47(1)	53.88±2.57(1)	4.3

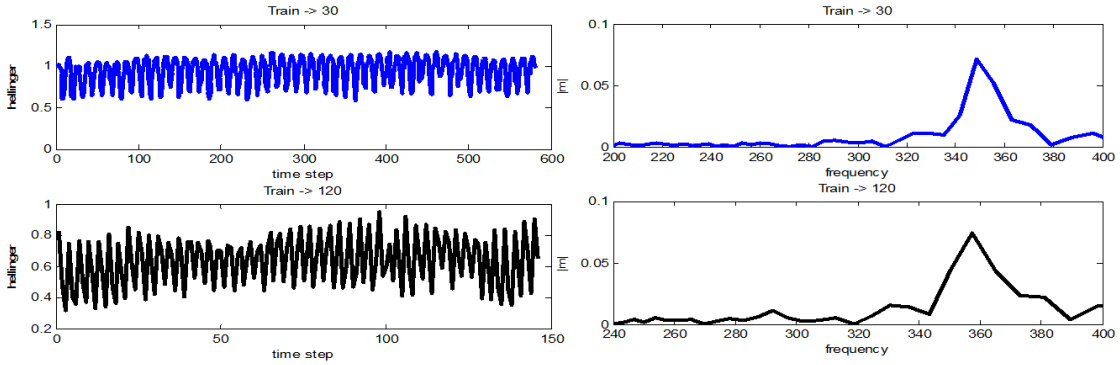
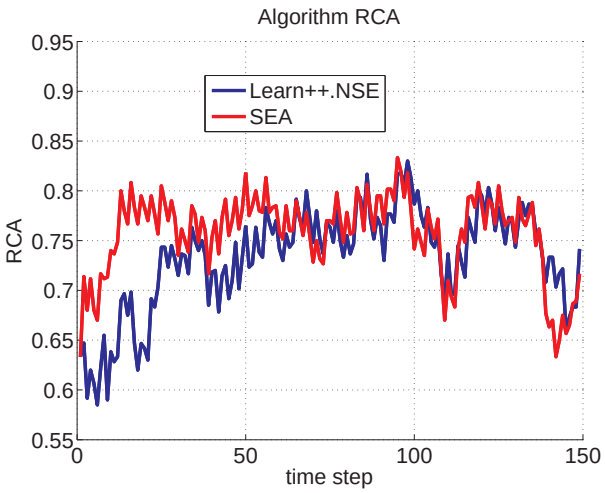


Fig. 5.33 : Distributional divergence measured between the 1^{st} batch and every subsequent batch of data in the entire NOAA databased (left). The FFT of the divergence (right) shows a large frequency component around 360 days.

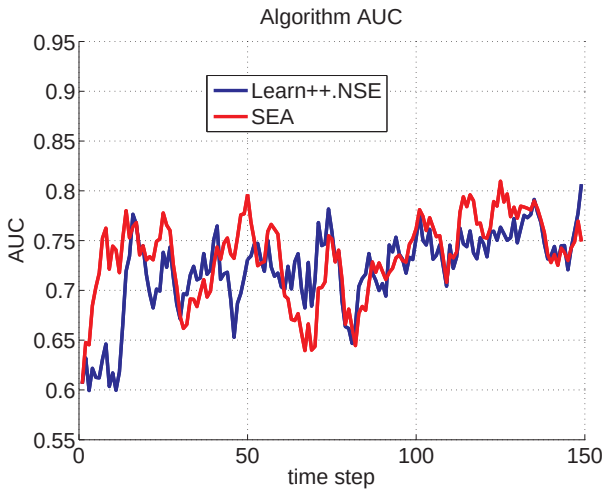
data is cyclical in nature by measuring the distributional divergence between the 1^{st} training batch and subsequent batches then computing the Fourier transform of this sequence (Fig. 5.33). The training size was set to 120 instances, which is approximately one season. The testing data is the next season (i.e. the next 120 instances after the training set).

Fig. 5.34, 5.35, and 5.36 present the results on the NOAA weather dataset. The mean values of all figures of merits used in the evaluation are tabulated in Table 5.37. Unlike previous datasets, Table 5.37 does not indicate any variation in Learn^{++}.NSE or SEA results because the generated base classifier (CART algorithm) is identical for each trial when we generate a classifier on any arbitrary batch of data. As in previous datasets, however, better rankings for Learn^{++}.NIE algorithms are observed. The differences between the Learn^{++}.NIE implementations were not statistically significant, whereas the improvement of any of the Learn^{++}.NIE implementations over any of the other algorithms were all significant, and often by wide margins. The only exception to this was again with the raw classification accuracy, where SEA performed the best, due to its majority class performance. Of course, SEA cannot accommodate imbalanced data, so its performances on all other metrics were – as expected – very poor. On the other hand, the three Learn^{++}.NIE implementations shared the top three spots in F-measure, AUC, recall, and OPM as well as mean rank. We also notice that both UCB and SERA perform rather poorly across all figures of merit on this dataset.

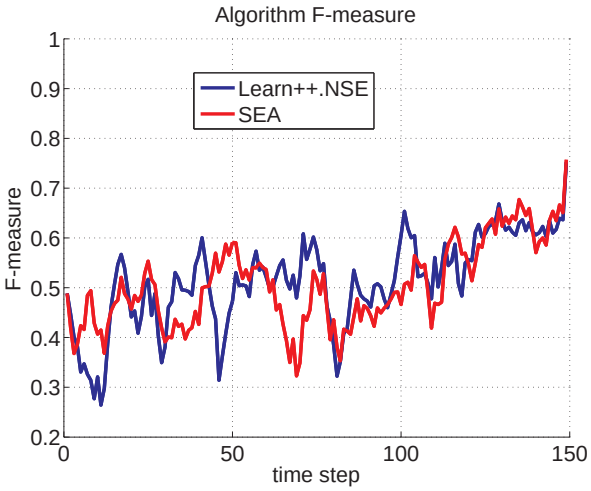
Once again, we observe a good set of ranks for the Learn++.NIE algorithms on the OPM



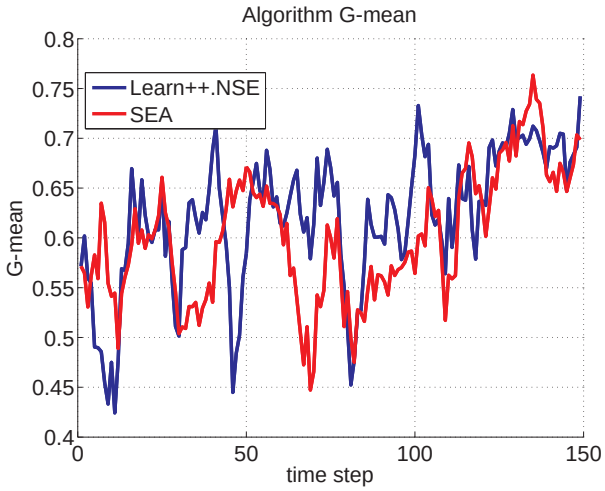
(a) Raw Classification Accuracy



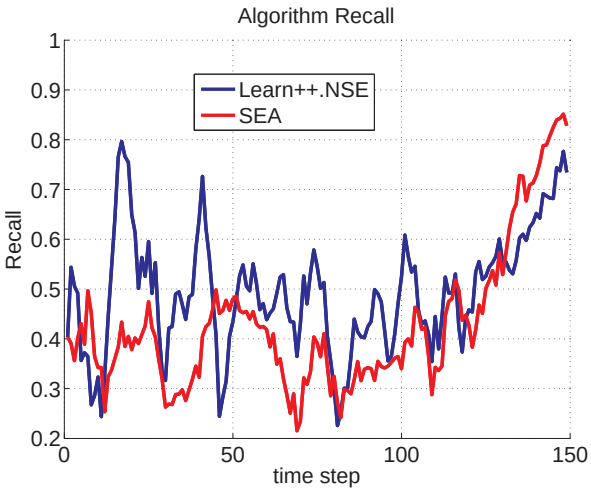
(b) AUC



(c) F-measure

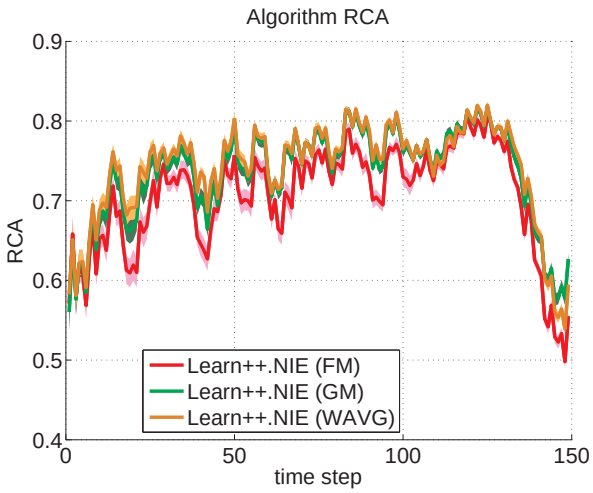


(d) G-mean

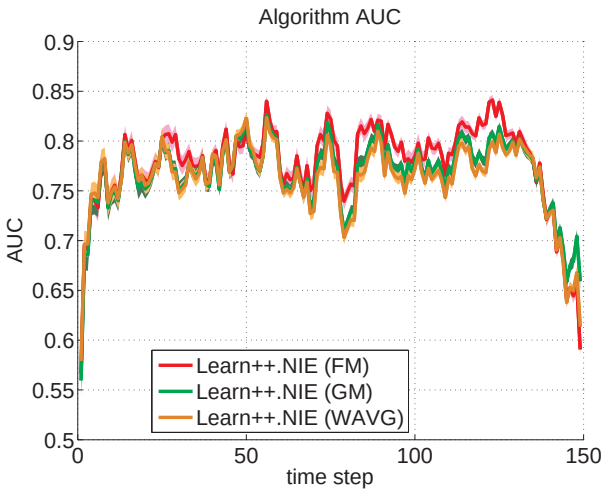


(e) Recall

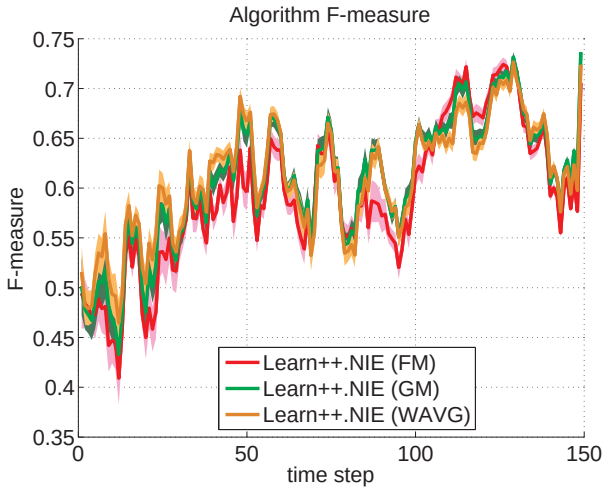
Fig. 5.34 : Learn^{++}.NSE and SEA evaluated on the NOAA Dataset



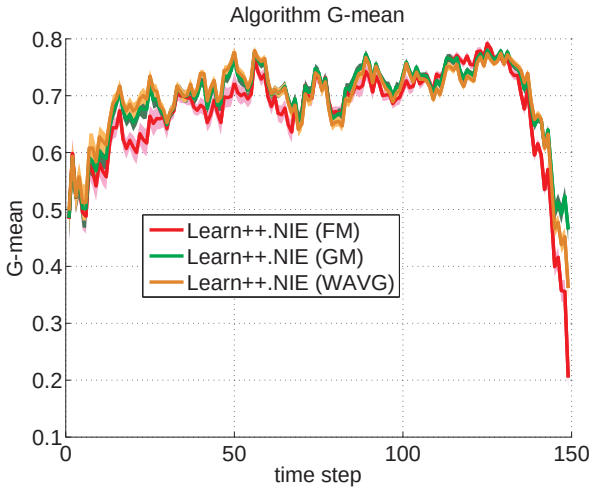
(a) Raw Classification Accuracy



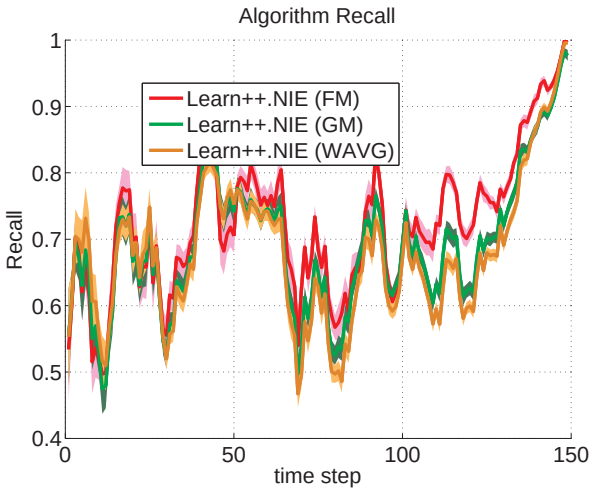
(b) AUC



(c) F-measure

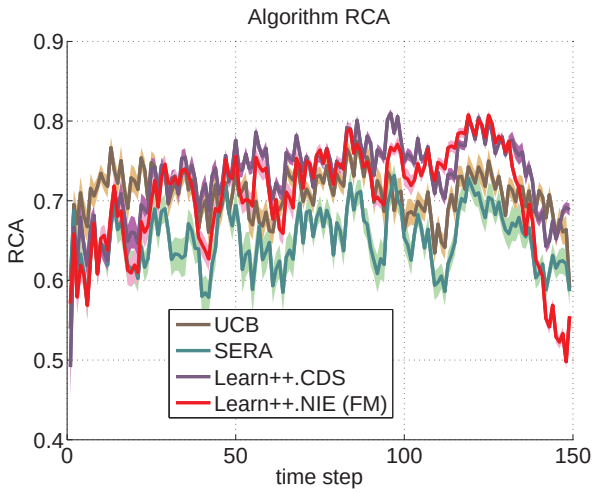


(d) G-mean

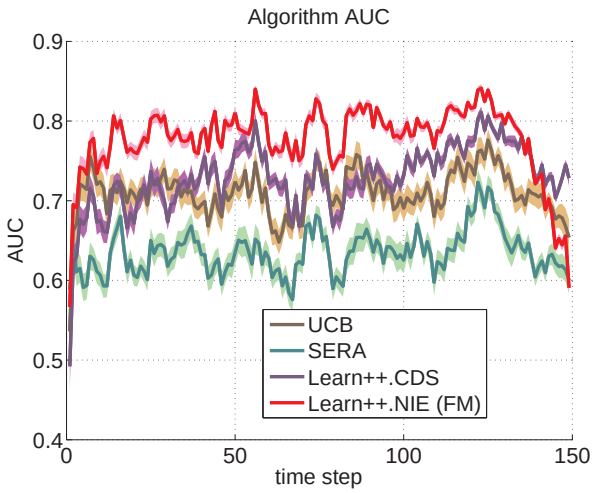


(e) Recall

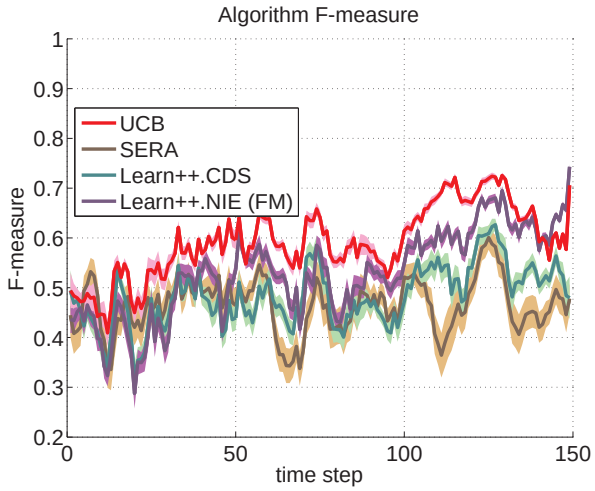
Fig. 5.35 : Learn^{++}.NIE family of algorithms evaluated on the NOAA Dataset



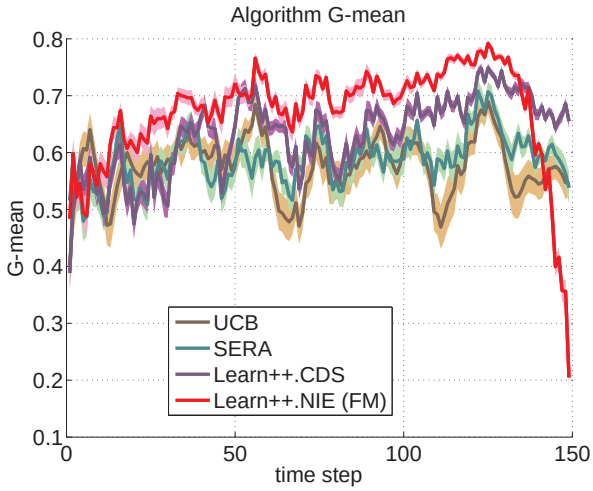
(a) Raw Classification Accuracy



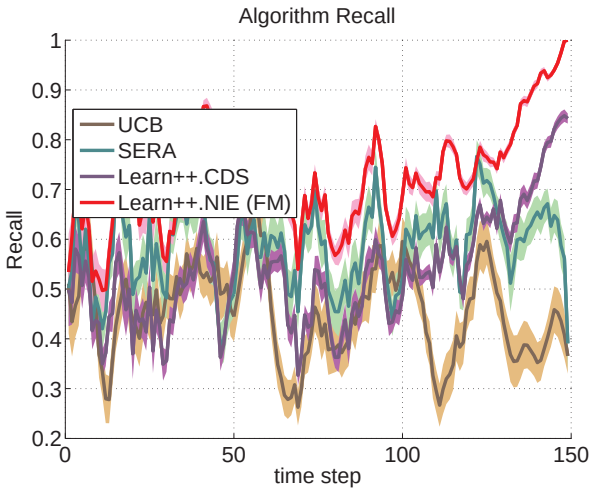
(b) AUC



(c) F-measure

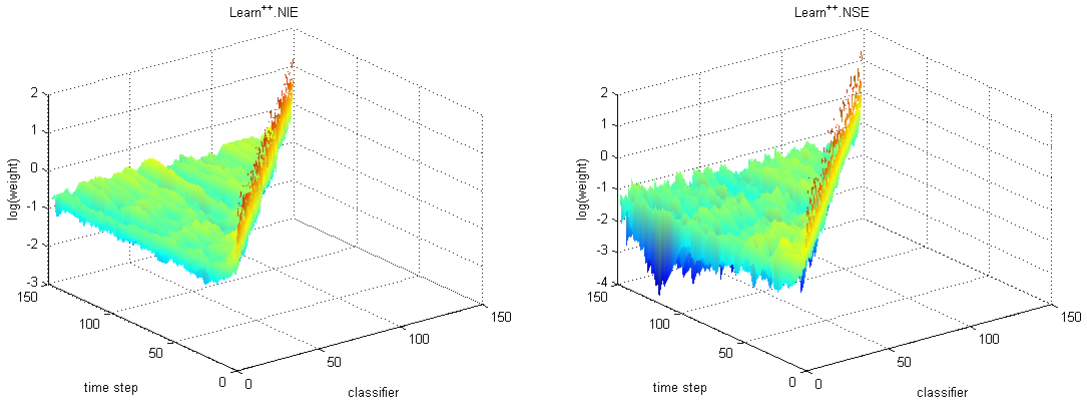


(d) G-mean



(e) Recall

Fig. 5.36 : Baseline algorithms evaluated on the NOAA Dataset



(a) Learn^{++}.NIE (wavg) (b) Learn^{++}.NSE

Fig. 5.37 : Learn^{++} weight evolution on NOAA

as indicated in Table 5.7. We find that Learn^{++}.NIE ($wavg$) and (gm) perform particularly well in regards to a variety of different measures. Similar to all the previous experiments we find that the simple integration of SMOTE and Learn^{++}.NSE can be beneficial to the improvement to the recall and overall performance measure.

The Learn^{++} family of algorithms can exploit the recurring concepts in the data because of the method behind computing the weights at each time stamp and saving classifiers. The algorithms that use old data, SERA and uncorrelated bagging, experience catastrophic forgetting. Catastrophic forgetting does not allow the algorithm to save knowledge and recall it at a later point in time.

The weights of the sub-ensemble classifiers in Learn^{++}.NIE can be seen in Fig. 5.37. Clearly, as classifiers are generated they later become irrelevant but many of them become relevant at later points in time. This once again demonstrates the power of saving classifiers (models) for use at a later point in time rather than employing methods such as catastrophic forgetting.

5.7 Summary of Learning for Concept Drift and Class Imbalance

Table 5.8 provides a summary of overall performance measure (OPM) ranks of each algorithm evaluated on all datasets along with the mean OPM rank averaged over all experiments. This table shows that Learn^{++}.NIE (fm), Learn^{++}.CDS and Learn^{++}.NIE (gm) occupy the top three spots in overall performance measure (OPM) as well as the mean

Table 5.7 : Algorithm Summary on NOAA Weather Data

	RCA	**FM**	**AUC**	**Recall**	**OPM**	**Rank**
L++.NSE	73.35±0.00(4)	51.27±0.00(5)	72.08±0.00(6)	49.38±0.00(6)	61.52±0.00(5)	5.3
SEA	75.81±0.00(1)	50.43±0.00(6)	73.37±0.00(4)	42.86±0.00(8)	60.62±0.00(6)	4.8
NIE(fm)	70.54±1.08(7)	59.19±1.31(3)	77.84±0.79(1)	72.48±2.19(1)	70.01±1.34(2)	3
NIE(gm)	73.53±0.80(3)	60.78±1.12(2)	76.83±0.69(2)	69.27±1.84(2)	70.10±1.11(1)	2.3
NIE($wavg$)	74.07±0.74(2)	60.94±1.04(1)	76.42±0.66(3)	68.04±1.71(3)	69.87±1.04(3)	2.3
L++.CDS	73.05±0.93(5)	52.89±1.74(4)	72.91±1.03(5)	53.75±2.69(5)	63.15±1.60(4)	4.8
SERA	65.17±1.83(8)	48.38±2.30(7)	63.54±1.48(8)	58.49±4.16(4)	58.90±2.44(7)	6.8
UCB	70.82±1.43(6)	46.40±3.18(8)	71.07±1.57(7)	45.54±4.77(7)	58.46±2.74(8)	7

Table 5.8 : Summary of the OPM ranks of all the algorithms on all datasets

	Gauss	**Checker**	**Spiral**	**SEA**	**Elec**	**NOAA**	**Average**	**Final**
L^{++}.NSE	7	3	5	6	7	5	5.50	6
SEA	8	8	7	7	8	6	7.33	8
NIE(fm)	2	2	2	1	3	2	2.00	1
NIE(gm)	4	6	3	4	2	1	3.33	3
NIE($wavg$)	5	7	4	5	4	3	4.67	5
L^{++}.CDS	3	1	1	3	6	4	3.00	2
SERA	6	5	8	8	5	7	6.50	7
UCB	1	4	6	2	1	8	3.67	4

rank, respectively. This table does not tell us whether one can claim – with any statistical significance – that any of the algorithms is actually better than the others.

When different algorithms provide varying performances on different datasets, one way to rigorously assess and compare the algorithms is the Friedman's test [116], which provides a rank based statistical significance test. This test, a nonparametric alternative to repeated measures ANOVA, tests whether there is a statistical difference among the ranks of different algorithms. To perform the Friedman test, begin by computing the average rank for each algorithm (R_j) across all datasets using Eq. (5.14) where r_{ji} is the rank of the jth of k algorithms on the ith of N datasets. We will have to do this for each measure and compare each separately. The Friedman statistic is then computed using Eq. (5.15).

$$R_j = \frac{1}{N}\sum_{i=1}^{N} r_{ji} \tag{5.14}$$

$$\chi_F^2 = \frac{12N}{k(k+1)} \left(\sum_{j=1}^{k} R_j^2 - \frac{k(k+1)^2}{4} \right) \tag{5.15}$$

The Friedman statistic is distributed according to χ_F^2 with $k-1$ degrees of freedom. However, the χ_F^2 has been shown to be too conservative, so [121] derived a better test statistic given by Eq. (5.16). F_f is distributed according to the F-distribution with $k-1$ and $(k-1)(N-1)$ degrees of freedom.

$$F_f = \frac{(N-1)\chi_F^2}{N(k-1) - \chi_F^2} \tag{5.16}$$

The Friedman test, when run over all algorithms, shows that the null hypothesis for each measure – that the rankings of the algorithms are not randomly distributed – is rejected, indicating that at least one algorithm is better than the others on each metric. Following the test description in [116], the z-scores can be computed as a test statistic for pair wise comparison of algorithm i vs. algorithm j where $i \neq j$, and $R_m(j)$, as shown in Eq. (5.17), is the average rank of algorithm j on measure m, k is the number of algorithms under test and N is the number of datasets used.

$$z(i,j) = \frac{R(i) - R(j)}{\sqrt{\frac{k(k+1)}{6N}}} \tag{5.17}$$

Since Table 5.8 lists Learn^{++}.NIE (fm) and Learn^{++}.CDS as the top ranking algorithms, we compare each of these two algorithms to all others on all measures – based on the pair wise Friedman test – to determine whether the performance increase – if any – is significant. The results are provided in Table 5.9. Significance is indicated by ♦ for Learn^{++}.NIE (fm) and by ■ for Learn^{++}.CDS. We also apply the Bonferroni-Dunn correction test to account for multiple comparisons of a group of algorithms to a control classifier (e.g., either Learn^{++}.NIE(fm) or Learn^{++}.CDS). We separate the Bonferroni-Dunn test into two groups, namely concept drift algorithms (baseline vs. SEA/Learn^{++}.NSE) and concept drift/class imbalance algorithms (baseline vs. Learn^{++}.NIE/CDS/SERA/UCB). We note that without this conservative correction, the results showed significant differences in favor of Learn^{++}.NIE and/or Learn^{++}.CDS even in the boxes that does not currently show significance. We observe that Learn^{++}.NIE (fm) and

Table 5.9 : Hypothesis testing comparing Learn^{++}.NIE (fm) (♦) and Learn^{++}.CDS (■) to other algorithms used during the presentation of results (only significant improvement is marked)

	L^{++}.NSE	**SEA**	**SERA**	**UCB**
RCA			■	■
FM		♦■		♦■
AUC	♦■	♦■	♦■	
Recall	♦■	♦■	♦	
OPM	♦■	♦■	♦■	

Learn^{++}.CDS are both significantly better than Learn^{++}.NSE and SEA on AUC, recall and OPM, but we cannot not claim significance on raw classification accuracy (RCA). This is not surprising, as Learn^{++}.NSE and SEA are designed for concept drift problems, and they perform very well on such problems, where their RCA performance is primarily due their accuracy on the majority class. Of course, we note that raw classification accuracy, alone, is never a good figure of merit on an imbalanced dataset. One or both of Learn^{++}.NIE(fm) and Learn^{++}.CDS also outperform SERA on all measures except F-measure. Learn^{++}.CDS significantly outperforms UCB in raw classification accuracy as well as F-measure, which itself is a combined measure of recall and precision. Learn^{++}.CDS has the best mean rank for F-measure, followed by Learn^{++}.NIE(fm). Both, Learn^{++}.NIE(fm) and Learn^{++}.CDS have a significantly better F-measure than UCB. While both Learn^{++}.NIE(fm) and Learn^{++}.CDS also outperform UCB on other measures, when averaged over all datasets, there was not sufficient evidence to determine if either Learn^{++} algorithms was better than UCB on those measures. Learn^{++}.NIE(fm) and UCB tie in terms of mean rank for recall across all datasets. Learn^{++}.NIE(fm) has the lowest (i.e., best) mean rank in AUC compared to UCB; however, there is not sufficient evidence to claim significance.

It is important to remember that in an imbalanced data environment, we seek a classifier that provides the best overall balance in classification accuracy, recall, precision and AUC. While no algorithm has absolute superiority on all others on all figures of merit, Learn^{++}.NIE (fm) and Learn^{++}.CDS outperform other algorithms far more often and with significance then their competitors when tested on a variety of synthetic and real world datasets that cover a broad spectrum of nonstationary environments.

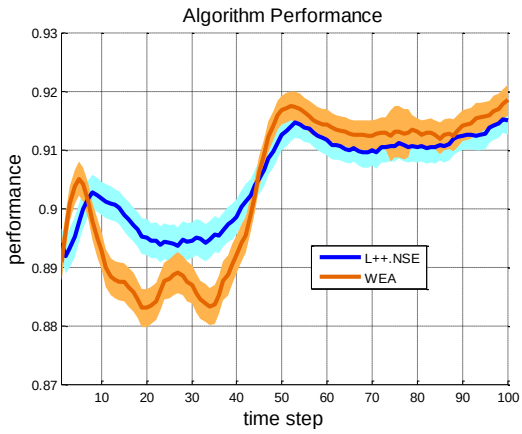
5.8 Weight Estimation Algorithm Experiments

In this section we present a set of results for WEA, presented in Section 4.3, on a few datasets. Recall that WEA takes advantage of unlabelled data to estimate the weights of classifiers in an ensemble. A set Gaussian and real-world datasets are presented to demonstrate the effectiveness of WEA.

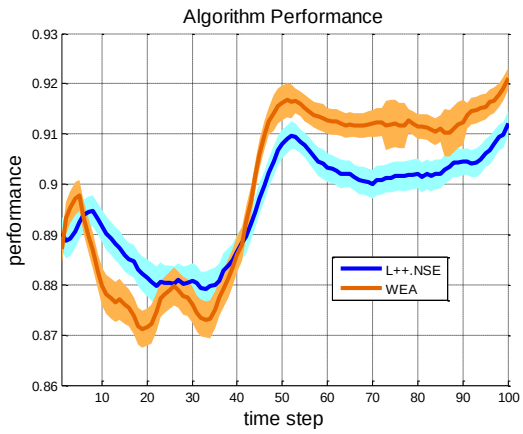
5.8.1 Rotating Circular Gaussian Drift

The rotating Gaussian dataset is comprised of two Gaussian mixtures rotating around one another. The class means can be given by the parametric equations $\mu_1^{(t)} = [\cos(\theta_t), \sin(\theta_t)]^T$, $\mu_2^{(t)} = -\mu_1^{(t)}$, $\theta_t = \frac{2\pi c}{N} t$, with fixed class covariance matrices given as $\Sigma_1 = \Sigma_2 = 0.5 * \mathbf{I}$, where c is the number of cycles, t is the (integer valued) time stamp that iterates from zero to $N-1$, and $\mathbf{I}$ is a 2×2 identity matrix. This is the same dataset presented in the description of the HDDDM algorithm (refer to Fig. 4.6). The experiment is run with 2 cycles, 1000 training/testing instances in each dataset, and a varying level of bias between the training and testing dataset. Bias is injected by sampling the testing data from a source at a future time stamp. This dataset is referred to as *GaussCir*.

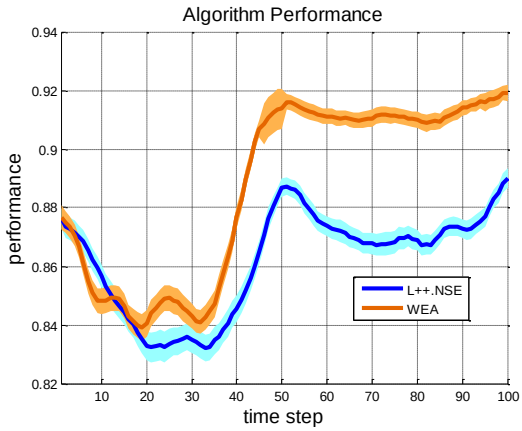
Fig. 5.38 shows the performance of WEA and Learn^{++}.NSE evaluated on the *GaussCir* dataset under varying levels of bias between the training and testing datasets. WEA uses one component for each class. WEA performs on par with Learn^{++}.NSE with zero bias between the training/testing batches. Both algorithms have a significant boost in performance when a reoccurring environment is encountered, which is at time step 50. WEA maintains nearly the same performance as it did without any bias when the bias is increased. However, Learn^{++}.NSE's performance begins to drop off rapidly as the bias increases. The effect of bias on Learn^{++}.NSE can be observed with small amount of bias as shown in Fig. 5.38(b) and 5.38(c), referring to a bias of 1 and 3 time stamps, respectively. This result is expected because Learn^{++}.NSE computes its classifier weights from a batch of data that was sampled from a significantly different distribution than the distribution used for the evaluation of Learn^{++}.NSE. Thus, the weights of Learn^{++}.NSE are not updated with respect to the most recent test data (i.e., unlabelled data). Many other concept drift algorithms may suffer the same drop in performance because information in the unlabelled data is not used to adjust classifier voting weights or any of the algorithm parameters. WEA maintains a dominant



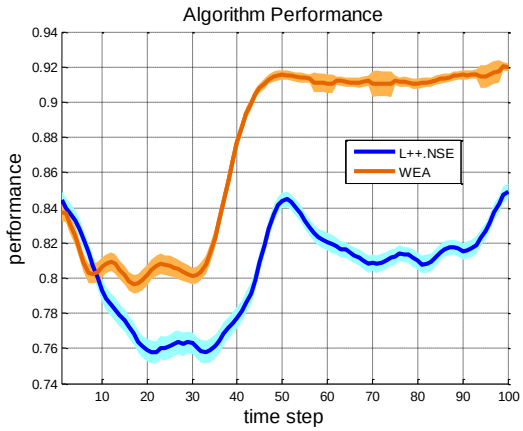
(a) Bias=0



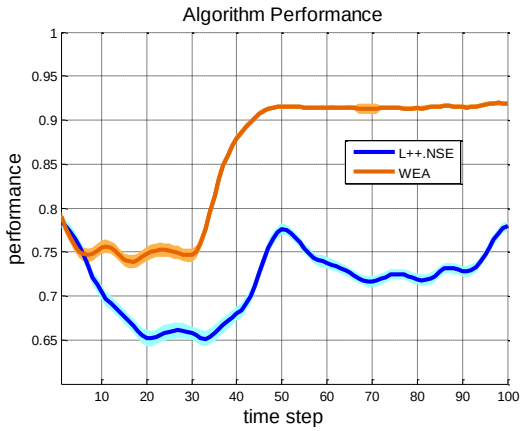
(b) Bias=1



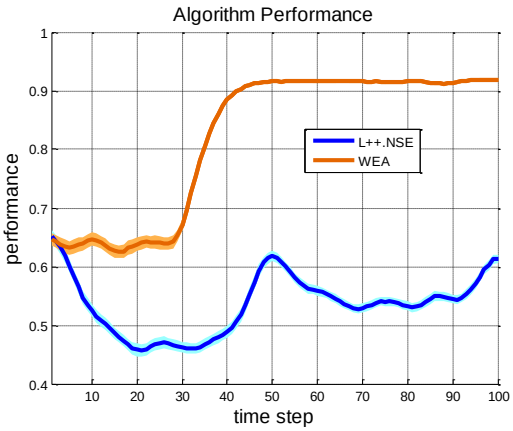
(c) Bias=3



(d) Bias=5



(e) Bias=7



(f) Bias=10

Fig. 5.38 : WEA vs. Learn^{++}.NSE on Rotating Circular Gaussian Drift

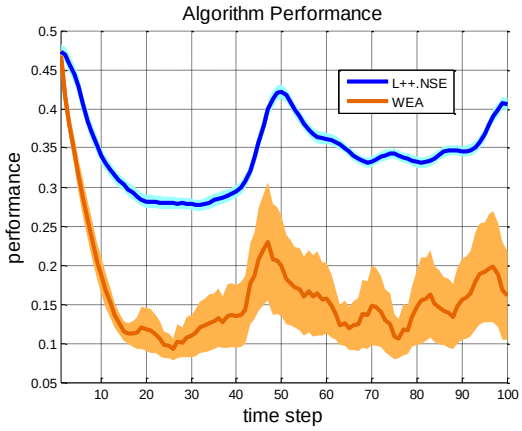


Fig. 5.39 : WEA vs. Learn^{++}.NSE on Rotating Circular Gaussian Drift with Failure

performance over Learn^{++}.NSE until the bias in the data becomes large enough that the limited drift assumption becomes violated (when bias=13 time steps) as shown in Fig. 5.39. Since this is a relatively easy problem (2 classes with one mixture each); violating the limited drift assumption becomes quite detrimental to the classification performance whereas a problem with a large number of mixture components may not experience the same degradation.

5.8.2 Rotating Triangular Gaussian Drift

A second Gaussian dataset was created, consisting of three components, each belonging to a different class moving in a triangular pattern. The environment experiences reoccurring concepts for a total of two rotations. The parametric equations that govern the mean and standard deviations for the x and y components are presented in Table 5.10. Each batch of training/testing data contains 1000 instances. The drift rate for the triangular Gaussian dataset as well as the circular Gaussian dataset remains constant throughout the duration of the experiment. This dataset is referred to as $GaussTri$.

The preliminary results of WEA on the GaussTri dataset are shown in Fig. 5.40. We run the experiment with two cycles and each GMM uses one component for each class, thus $K = 2$. We observe similar results on the $GaussTri$ dataset as with $GaussCir$. WEA is a very strong predictor when the bias between the training and testing datasets is 3, 5, 7, or 10 time stamps difference. The response of WEA becomes slightly less stable when the

Table 5.10 : Mean and standard deviation Gaussian drift over time.

	$0 < t < 1/6$ $1/2 < t < 2/3$		$1/6 < t < 2/6$ $2/3 < t < 5/6$	
	μ_x	μ_y	μ_x	μ_y
ω_1	$5 + 18t$	$8 - 36t$	$8 - 36t$	2
ω_2	$2 + 18t$	$2 + 36t$	$5 + 18t$	$8 - 36t$
ω_3	$8 - 36t$	2 2	$2 + 18t$	$2 + 36t$

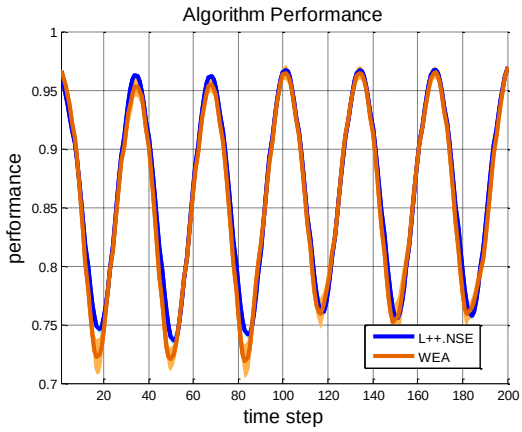
	$2/6 < t < 1/2$ $5/6 < t < 1$	
	μ_x	μ_y
ω_1	$2 + 18t$	$2 + 36t$
ω_2	$8 - 36t$	2
ω_3	$5 + 18t$	$8 - 36t$

bias increase is significant (Fig. 5.41). However, WEA performs quite well when the bias is "reasonable" and the limited drift assumption and held, which suggests WEA should perform well on problems where the distribution can be modeled with GMMs and the drift is gradual or incremental in nature.

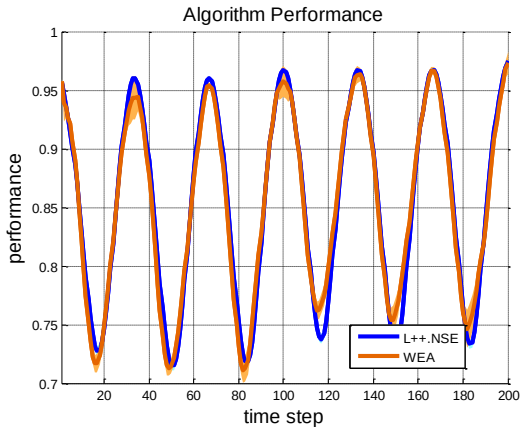
5.8.3 NOAA Dataset

Finally, the NOAA weather dataset that was presented in Section 5.6.2 is applied for learning concept drift. The entire dataset is divided into batches of 120 instances for training and testing. Once the batches are formed, synthetic data is generated for each class to obtain more data for training and testing. This is because EM typically requires a significant amount of data to properly estimate a complex distribution, which may be encountered with a real-world data mining problem. Three components were selected for each class after trial & error. A CART decision tree is used as the base classifier.

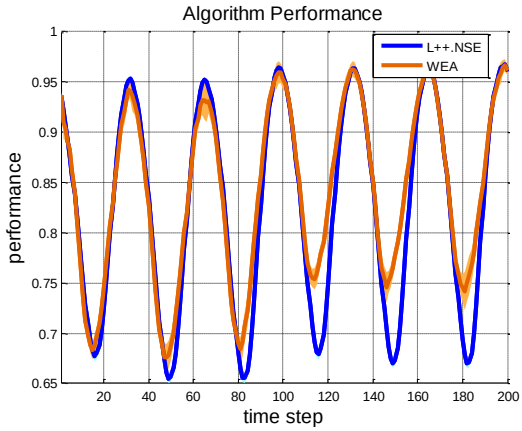
Fig. 5.42 contains the results from adding different levels of bias between the training and testing datasets. We typically find that WEA and Learn^{++}.NSE perform the same, statistically speaking. At times Learn^{++}.NSE outperform WEA however, one cannot make a clear distinction about an algorithm's superiority. WEA does not encounter any severe drops in performance as observed with the synthetic datasets when an assumption was violated.



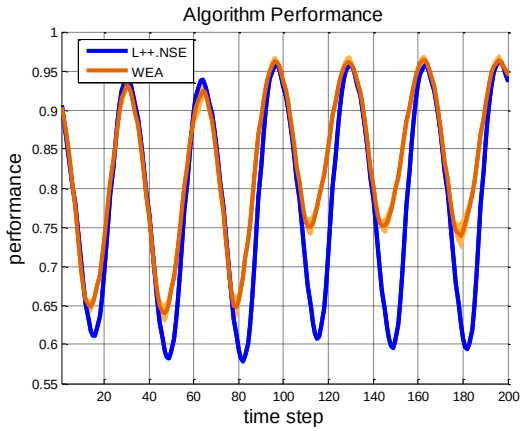
(a) Bias=0



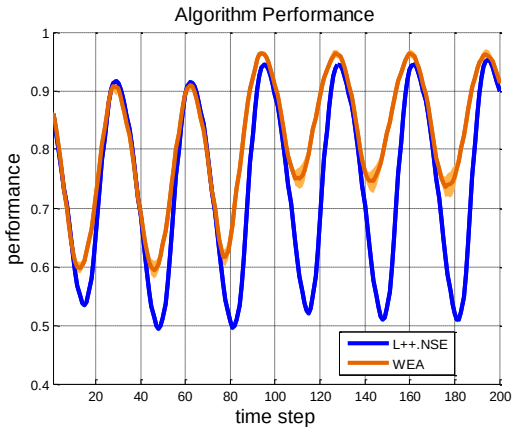
(b) Bias=1



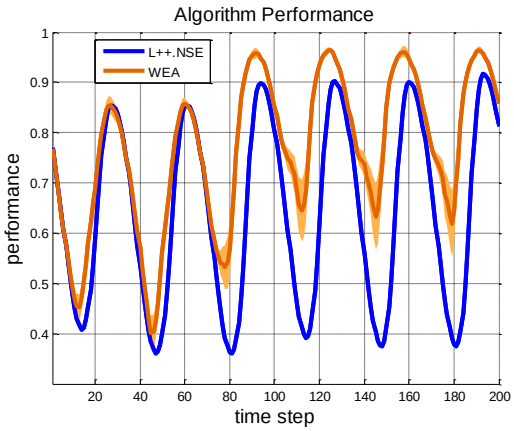
(c) Bias=3



(d) Bias=5



(e) Bias=7



(f) Bias=10

Fig. 5.40 : WEA vs. Learn^{++}.NSE on Rotating Triangular Gaussian Drift

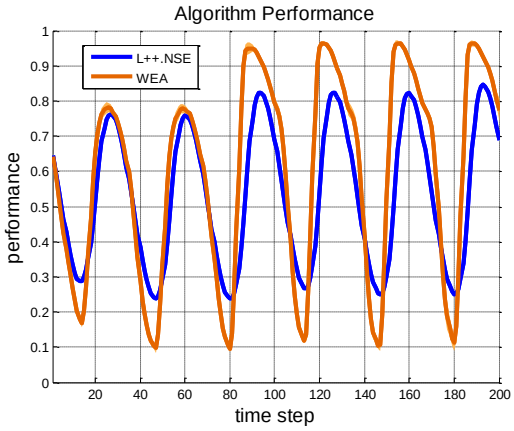
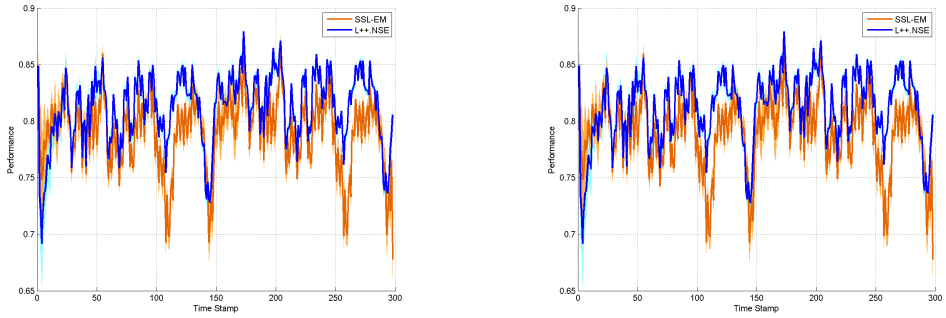


Fig. 5.41 : WEA vs. Learn^{++}.NSE on rotating triangular Gaussian drift with 13 time stamp bias.

5.9 Drift Detection using the Hellinger Distance

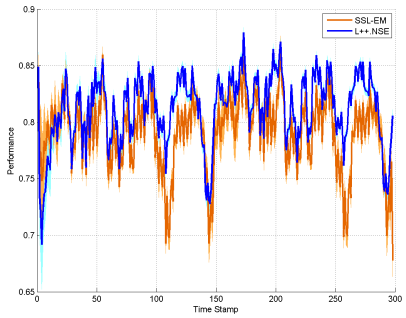
The final section of this chapter experiments with the Hellinger Distance Drift Detection Method (HDDDM) on several synthetic as well as real-world datasets. As summarized in Table 5.11, seven datasets were used. The rotating checkerboard dataset is generated using the Matlab source found in [52] and is explained in Section 5.5.1. The experiment is controlled by changing the rotation of the checkerboard over 400 time stamps. The drift is abrupt and has 15 different distributions in the dataset as depicted in Fig. 5.43, which contains a few snapshots of the checkerboard appearances over time. Electricity pricing (Elec) is a real-world dataset that contains natural drift within the database [120]. SEA dataset comes from the shifting hyper-plane problem presented by Street & Kim [49]. Magic dataset is from the UCI machine learning repository [105]. The original dataset, though being used in the past by others for concept drift problems, actually includes little if any drift. Therefore, this dataset has been modified by sorting a feature in ascending order and then generating incremental batches on the sorted data with a meaningful drift as an end-effect.

GaussCir and RandGauss are two synthetic datasets generated with a controlled drift scenario (whose decision boundaries are shown in Fig. 5.44 and 5.45, respectively). GaussCir is the example previously described in Section III. The NOAA dataset contains approximately 50 years of weather data obtained from a post at Offutt Air Force Base

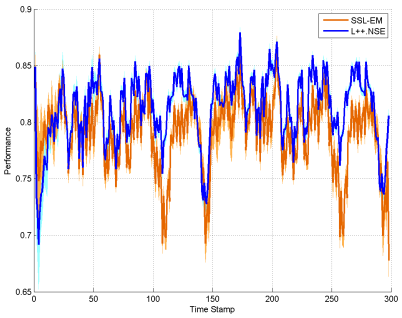


(a) Bias=0

(b) Bias=1



(c) Bias=3



(d) Bias=5

Fig. 5.42 : WEA vs. Learn^{++}.NSE on NOAA Dataset

in Bellevue, Nebraska. Daily measurements were taken for a variety of features like temperature, pressure, visibility, wind speed, etc. The number of feature vectors was reduced to eight features and the classification task was to predict whether or not there was rain on a particular day.

The preliminary results are summarized in Fig. 5.46 and 5.47. Each of the plots presents the error of a naïve Bayes classifier with no update or intervention from HDDM, and with various values of γ (0.5, 1.0, 1.5, 2.0) for the standard deviation implementation of the HDDDM, as well as with $\alpha = 0.1$ for the confidence interval based implementation of HDDDM. The primary observation to make is that the classifier that is continuously updated, disregarding the concept change (red curve, corresponding to $\mathcal{N}_2$, i.e., no drift detection being used), has an overall error that is typically greater than the error of the classifiers that are reset based on the intervention of the HDDDM (other colors, corresponding to different implementations of $\mathcal{N}_1$). Note that resetting an on-line classifier is

Table 5.11 : Description of datasets and and their drift properties used for the experimentation of HDDDM

Dataset	Instances	Source	Drift Type
Checkerboard	800,000	Matlab generated	Synthetic
Elec2	27,549[4]	Web Source[1]	Natural
SEA(5%)	400,000	Matlab generated	Synthetic
RandGauss	812,500	Matlab generated	Synthetic
Magic	13,065	UCI[2]	Synthetic
NOAA	18,159	NOAA[3]	Natural
GaussCir	950,000	Matlab generated	Synthetic

1. http://www.liaad.up.pt/∼jgama/ales/ales_5.html
2. UCI Machine Learning Repository [105]
3. National Oceanic and Atmospheric Administration(www.noaa.gov)
4. All missing instances with missing features have been removed

not necessarily the most ideal method to implement concept drift (as it causes catastrophic forgetting [20]), and there are several algorithms that can forget only what is no longer relevant, and retain the still-relevant information, such as the Learn^{++}.NSE algorithm [13, 12]. However, since the goal is to evaluate the drift detection mechanism, using the resetting approach, so that any improvement observed is not due to the innate ability of the learner to learn concept drift, but simply the impact and effect of intervention based on drift detection.

Of the seven datasets, the rotating checkerboard dataset provides the best way to determine the effectiveness of HDDDM since the drift occurs at evenly spaced intervals: approximately every 20 time steps, out of 300, the board rotates 0.49 radians, and remains static during the intermediate time stamps. This leads to a total of 15 concept changes (including time step 1) throughout the experiment. Table 5.12 presents the f-measure, sensitivity and specificity of the HDDDM's detections as averages of 10 independent trials. These quantities can be computed, precisely because the locations of the drift points are known for this dataset. In Table 5.12, sensitivity (ω_1) represents the ratio of the number of drifts detected to total number of real drifts that were actually present for ω_1, whereas specificity (ω_2) is the ratio of number of no-drift time steps to total number of no-drift steps in class ω_2. The performance measure in Table 5.12 indicates the detection rate across all data. Note that with class label removed, there is no change in checkerboard

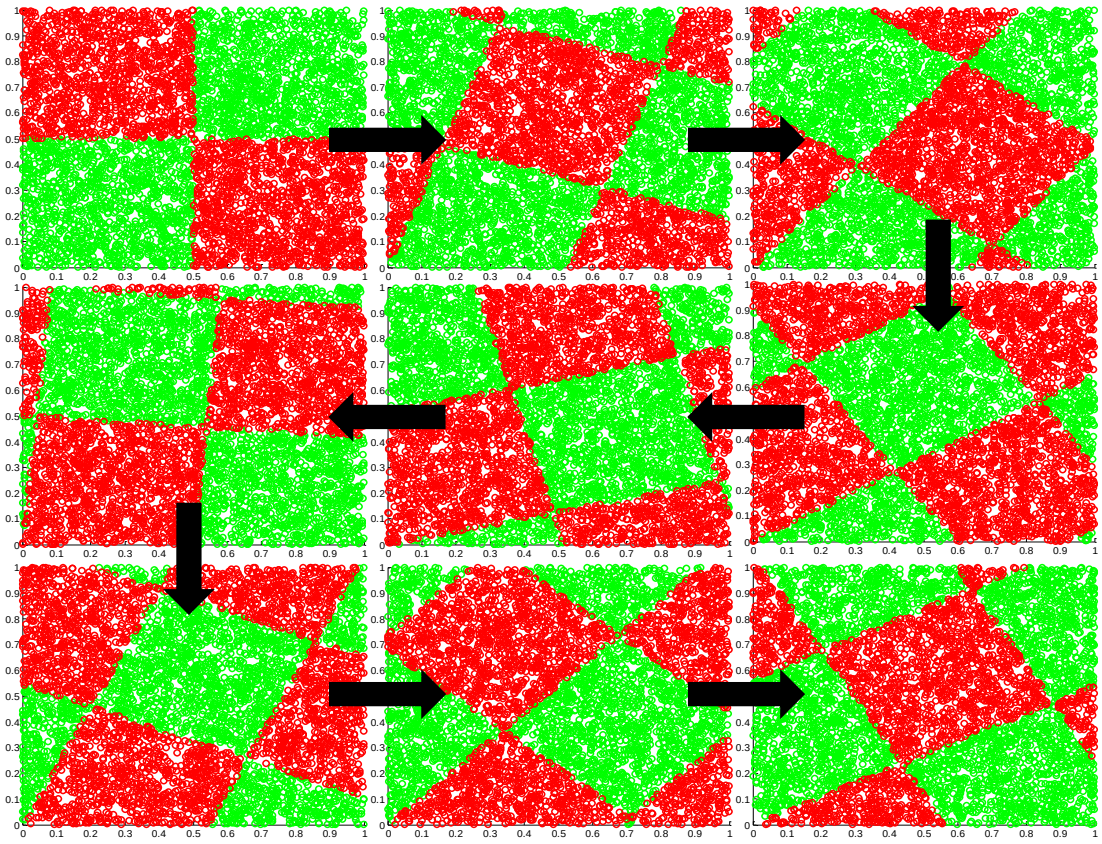


Fig. 5.43 : Evolution of the rotating checkerboard dataset

distribution, and hence there should be no change detected. The numbers are therefore de facto specificity figures for the entire data. Sensitivity cannot be measured on the entire data (when class labels are removed) since there are no actual drifts in such data.

Table 5.12 also shows the effect of the variation of γ and α on the checkerboard dataset. A smaller γ is better for sensitivity (and f-measure), whereas a larger γ is better for specificity – not a surprising outcome – with $\gamma = 1$ providing a good compromise. The standard deviation implementation of HDDDM generally maintains a higher performance than the t-statistic implementation, however the latter is more tolerant to changes in its parameter (of α values). By performance we are referring to the performance of the HDDDM algorithm in detecting change and not the performance of the naïve Bayes classifier.

Fig. 5.48(a) and 5.48(b) display the location of drift detection on ω_1 and ω_2 from the rotating checkerboard dataset, as an additional figure of merit. Recall that this dataset experiences a change every ≈20 time steps. There are 15 distinct points of drift (including

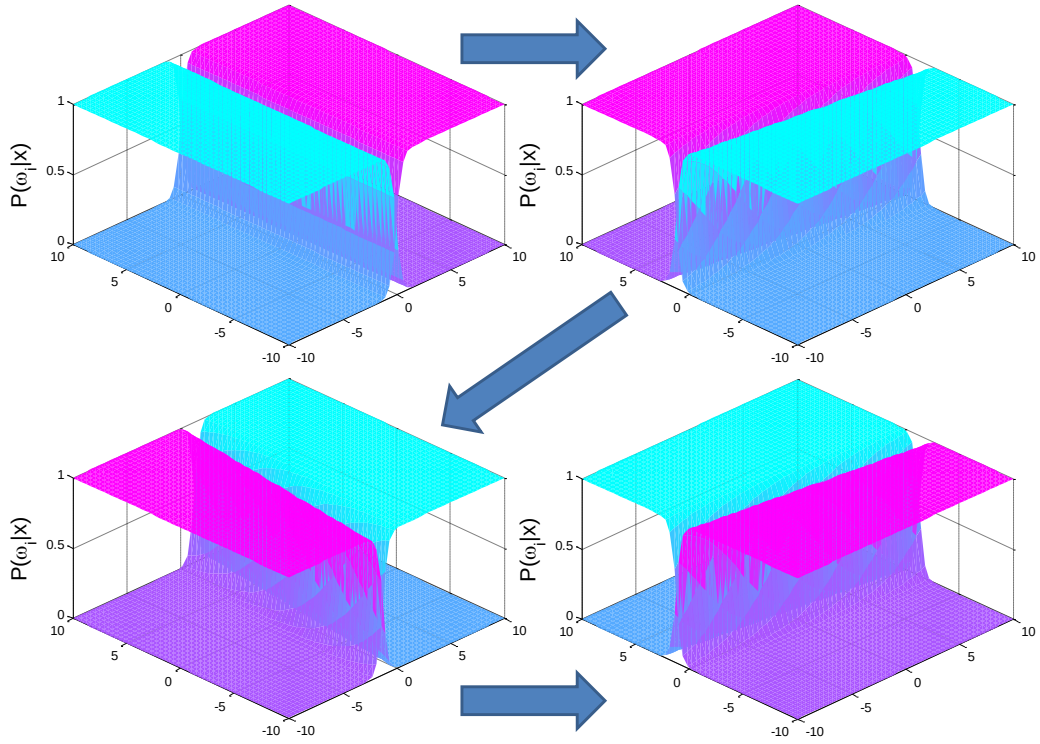


Fig. 5.44 : Evolution of the circular Gaussian drift by observing the posterior probability. The drift is continuous from database to database and not abruptly changing. There are 100 time stamps for each cycle.

first time step) in the checkerboard dataset each indicated by the vertical grid lines, whereas each horizontal grid line indicates a different selection of the free parameters of the HDDDM algorithm.

These plots provide a graphical display on the algorithm's ability to detect the changes. Every marker that coincides with the vertical lines is a correct detection of drift. Every marker that falls off a vertical grid is a false alarm of a non-existing drift, whereas every missing marker on a vertical grid is a missed detection of an actual drift. HDDDM was able to make the correct call in vast majority of the cases. However, a few cases are worth further discussion: for example, there are certain times when drift is detected in one of the classes but not the other, even though the drift existed on both classes. Consider the situation for $\gamma = 1.5$, where a change was detected in ω_1 at time stamp 102, while drift was not detected in ω_2. However, the HDDDM algorithm will correctly detect the change in the data in this case, because our implementation decides on a change when drift is detected in either one

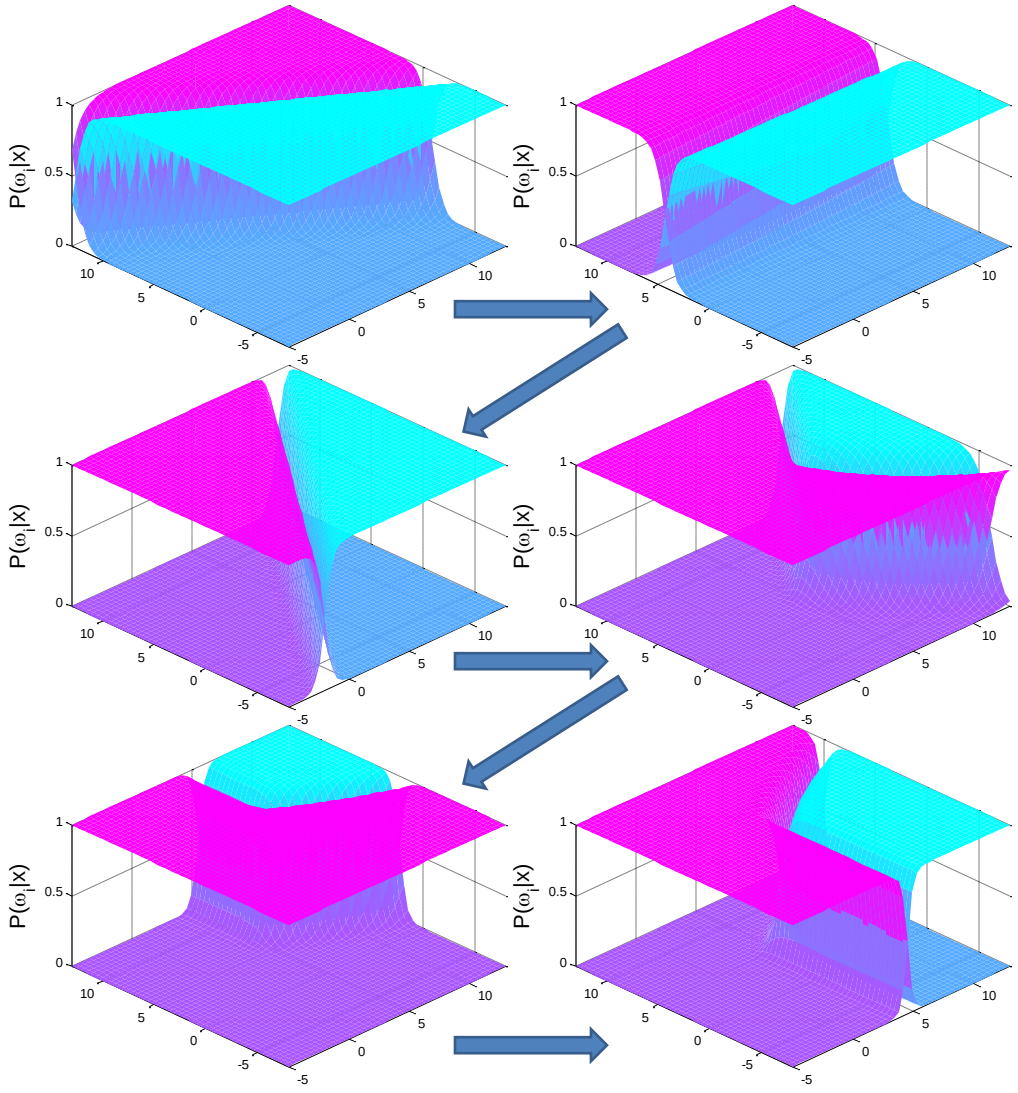
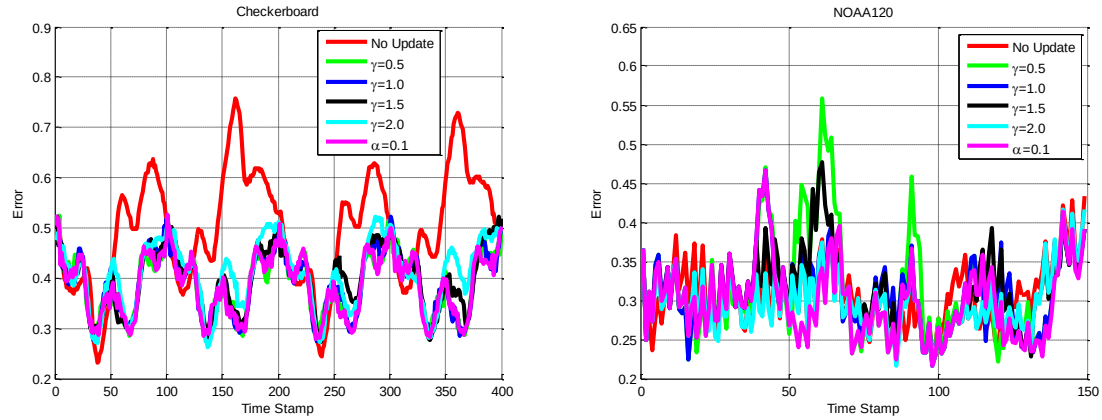


Fig. 5.45 : Evolution of the RandGauss dataset by observing the posterior probability. The dataset begins with 2-modes (one for each class) and begin slowly drifting. The drift stops at the third evolution point and remains static for 25 time stamps followed by the introduction of a new mode for the magenta class. The dataset continues to evolve with slow change followed by abrupt changes.

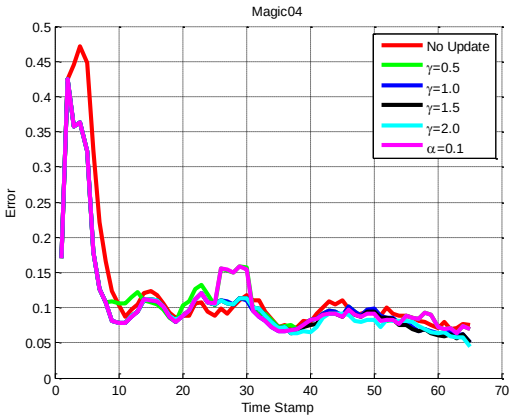
of the classes. This approach will accommodate all cases where the drift is detected in at least one of the classes. The drawback, however, is a potential increase in false alarm rate: if the algorithm incorrectly detects drift for any of the classes, it will indicate that the drift exists, even though it actually may not.

A table similar to Table 5.12, or figures similar to Fig. 5.48(a) and 5.48(b) cannot be

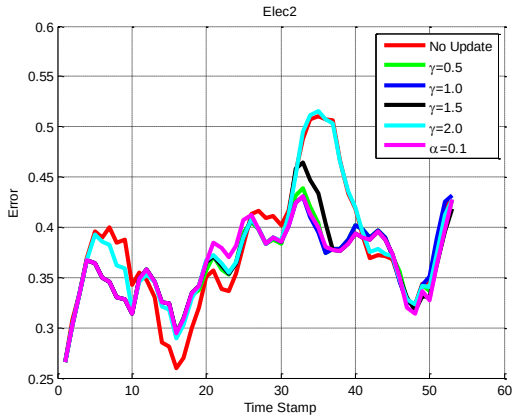


(a) CB2

(b) NOAA



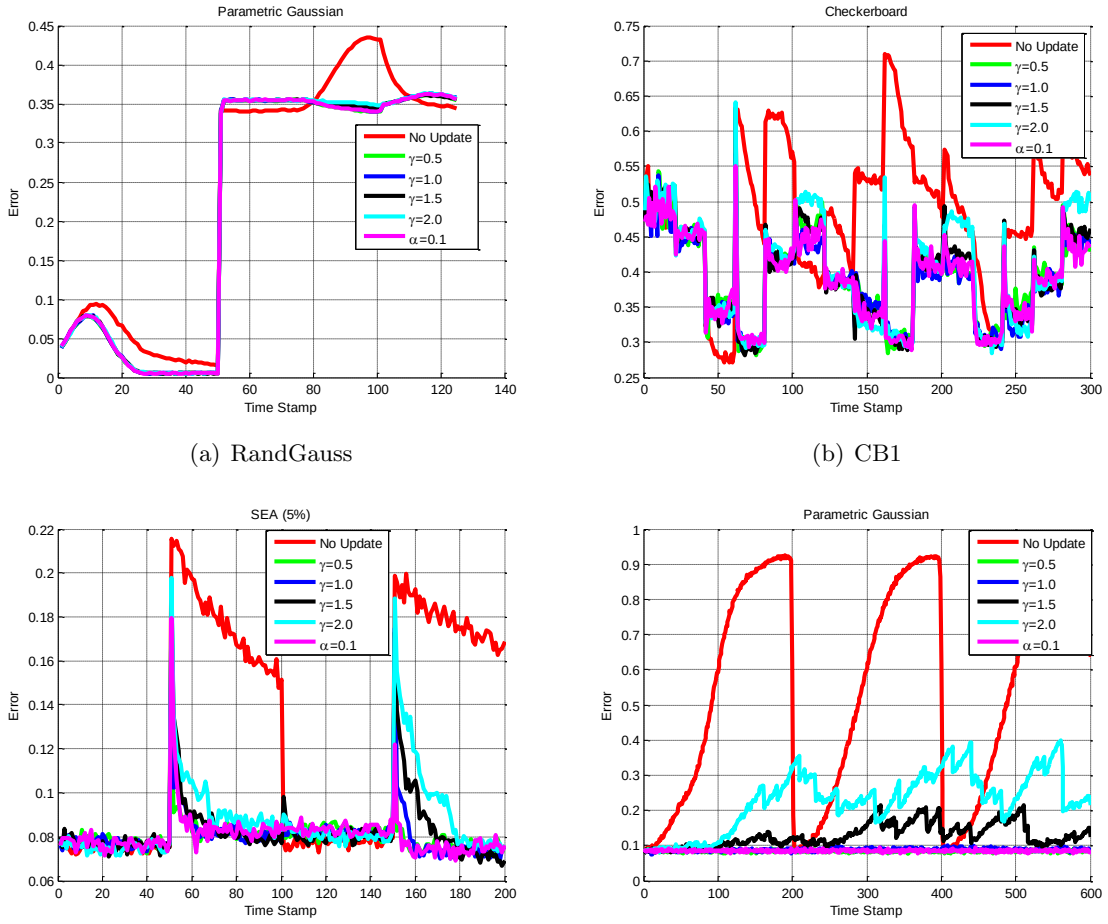
(c) Magic04



(d) Elec2

Fig. 5.46 : Error evaluation of the on-line naïve Bayes classifiers (updated and dynamically reset) with a variation in the parameters of the Hellinger Distance Drift Detection Method (HDDDM).

generated for other databases, either because the drift is continuous, or the exact locations of the drift are actually not known.



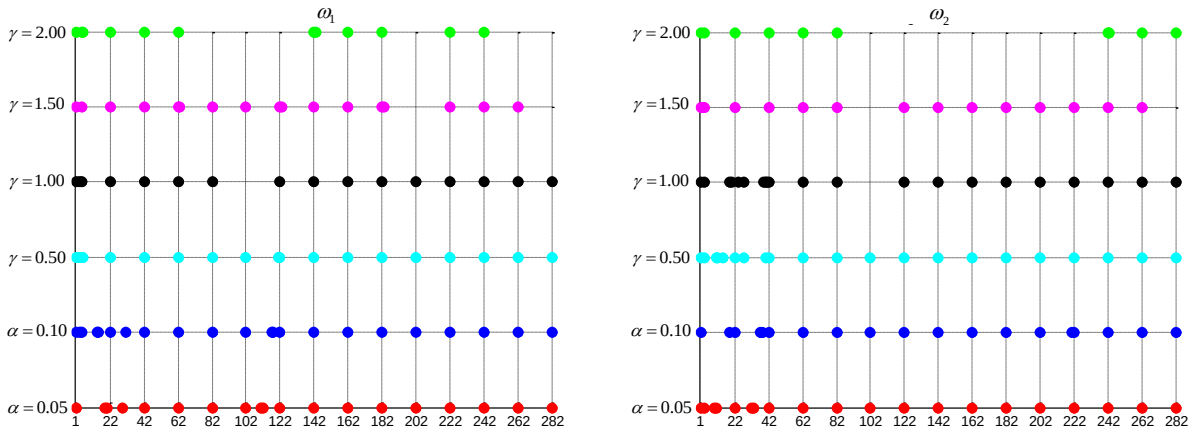
(a) RandGauss (b) CB1

(c) SEA (d) GaussCir

Fig. 5.47 : Error evaluation of the on-line naïve Bayes classifiers (updated and dynamically reset) with a variation in the parameters of the Hellinger Distance Drift Detection Method (HDDDM).

Table 5.12 : F-measure, sensitivity and specificity measures on the rotating checkerboard dataset averaged of 10 independent trials. True positives correspond to the true points of change in the data

Parameter $\rightarrow$	γ				α	
Measure $\downarrow$	0.5	1.0	1.5	2.0	0.05	0.10
f-measure (ω_1)	0.80	0.84	0.78	0.64	0.79	0.82
sensitivity (ω_1)	1.00	0.97	0.81	0.61	0.98	1.00
specificity (ω_1)	0.97	0.98	0.98	0.99	0.97	0.97
f-measure (ω_2)	0.79	0.81	0.78	0.64	0.82	0.80
sensitivity (ω_2)	0.99	0.94	0.86	0.58	0.97	0.98
specificity (ω_2)	0.97	0.98	0.98	0.98	0.89	0.97
performance	0.81	0.87	0.91	0.92	0.85	0.82



(a) ω_1 drift detections

(b) ω_2 drift detections

Fig. 5.48 : Drift detection locations for the rotating checkerboard problem with abrupt points of change for ω_1 (Fig. 5.48(a)) and ω_2 (Fig. 5.48(b))

Chapter 6

Conclusions

The primary scope of this thesis was to develop algorithms capable of learning concept drift and class imbalance simultaneously. All algorithms proposed in this thesis are designed for incremental learning and concept drift. The definitions of concept drift and incremental learning have be clearly defined and upheld in the design of the proposed approaches. The core portions of this thesis are summarized below.

1. Several algorithms have been proposed that focus specifically on incremental learning, concept drift and class imbalance. The algorithms are evaluated using a variety of different statistical measures to verify the functionality and benefits of each of the approaches to learning in harsh environments.
2. An weight estimation algorithm was presented that uses information in unlabelled data to dynamically compute classifier voting weights.
3. A drift detection algorithm that uses the Hellinger distance was presented to detect a nonstationary change in data for incremental learning scenarios.

6.1 Contributions of this Work

Learn^{++}.NIE (*nonstationary and imbalanced environments*) is an algorithm tailored to learning incrementally from concept drift and class imbalance in data mining problems. Learn^{++}.NSE's sigmoid weighting scheme is applied to determine classifiers performing well in recent time, however the measures used to derive the classifier weights are significantly different than that of Learn^{++}.NSE. We have presented several variations of Learn^{++}.NIE:

- Learn^{++}.NIE (fm): A classifier's voting weight is derived from the F-measure on the labelled data over time. A normalized logistic is applied to the classifier's F-measure over-time.
- Learn^{++}.NIE (gm): The geometric mean is applied to indicate if a classifier is performing well across all classes or just a majority class. The geometric mean of

the recalls of the classes. A normalized logistic is applied to the classifier's geometric mean over-time.

- Learn^{++}.NIE ($wavg$): The true positive rate (tpr) and true negative rate (tnr) is computed for each classifier in the ensemble at each new dataset. The classifier weighted error is calculated using Eq. (4.15). A normalized logistic is applied to the classifier's weighted error over-time.

Learn^{++}.CDS (*Concept Drift with SMOTE*) was presented as quick work around to learning concept drift and class imbalance simultaneously. The simple integration of Learn^{++}.NSE and SMOTE provided a meaningful increase in minority class recall while maintaining a good compromise between F-measure, G-mean and AUC.

We have presented WEA (*weight estimation algorithm*), which is a transductive learning ensemble to estimate the Bayes-optimal weights derived from a discriminate function. Unlabelled data are used along with Gaussian mixture models to estimate the weights and predict labels for field data.

Finally, we presented a drift detection algorithm, inspired in part by [93], that relies only on the raw data features to estimate whether drift is present in a supervised incremental learning scenario. This approach utilizes the Hellinger distance with an adaptive threshold as a measure to infer whether drift is present between two batches of training data. The adaptive threshold was analyzed using a standard deviation and t-statistic approach. This threshold is computed based on the information obtained from the divergence between training distributions.

6.2 Summary of Findings

This thesis has presented several methods to learn from batch data incrementally that experiences concept drift and imbalanced classes. The focus was placed on binary classification problems. The proposed approaches are truly incremental in that they do not require access to previous data. The algorithms have the ability to learn new knowledge and preserve prior knowledge about the environment, which is particularly useful for reoccurring concepts.

6.2.1 Learning Concept Drift and Imbalanced Data

We have presented new members to the Learn^{++} suite of algorithms that are designed to handle the fusion of concept drift and class imbalance. Learn^{++}.CDS is a combination of an algorithm designed for concept drift (Learn^{++}.NSE [13]) and an algorithm for sampling and rebalancing imbalanced data (SMOTE [98]). This straight forward combination is very robust when analyzed on several synthetic and real-world datasets.

Learn^{++}.NIE has been expanded upon our preliminary work presented in [109]. Learn^{++}.NIE uses measures other than a class independent error to weigh classifiers to allow for concept drift to be tracked, and boost the recall of a minority class. Sub-ensembles are combined using measures other than a class independent error, such as the F-measure, weighted error or the geometric mean, to track concept drift and boost the recall of a minority class. The measures presented in this paper were selected to reward classifiers in the ensemble that perform well on all classes rather than an error measure that may be biased towards a majority class or do not examine the error on a minority class. The η parameter of Learn^{++}.NIE allows control over how much penalty is given to the error of the majority and minority class recall separately. The term effectively allows choosing a balance between recall of the minority class and overall performance of the algorithm. This was demonstrated in a series of experiments.

The proposed approaches have been compared to other algorithms specifically designed for class imbalance and concept drift. Each of these approaches retains some data from the minority class at each time step. During the experimentation process we have kept all algorithm parameters constant in order to maintain a fair comparison. The proposed Learn^{++} based incremental learning algorithms have the ability to recall old environments from the models saved in the ensemble unlike approaches UCB, SERA or MuSERA [61, 21, 106], which require access to old data.

A reasonable question that now needs to be answered is: which algorithm should be applied to a particular task if there is some amount of prior knowledge about the problem? In an imbalanced data concept drift scenario, we have several criteria and constraints that are sometimes conflicting in their nature: of course, we want good classification accuracy in general, but we also want to be able to recall the minority class data, maintain good performance on majority data, and maintain a healthy ROC characteristic. We observed,

while not significantly outperforming others on all datasets and all figures of merits, Learn^{++} based approaches typically provided a better-balanced performance. Based on our observations on several different datasets and figures of merit, we reach the following set of guidelines:

- *Learn^{++}.NSE*: Use Learn^{++}.NSE when there is concept drift in the data and the classes are relatively balanced. If classes are imbalanced, Learn^{++}.NSE may still obtain a relatively good classification accuracy - a potentially misleading result – as it is based on its majority class performance. The recall of minority class will suffer.
- *Learn^{++}.NIE*: Learn^{++}.NIE is the better overall algorithm, if both minority and majority concepts are drifting and a strong balanced performance is needed on both minority and majority classes. Learn^{++}.NIE uses a weight that reflects the performance on weighted recall measure ($wavg$), F-measure, or G-mean. The F-measure weighting scheme typically provided the best results on a broad array of learning scenarios. Note that while Learn^{++}.NIE with $wavg$ came third in overall ranking behind other Learn^{++}.NIE variants, it has the distinct feature to control performance for recall and precision through its η parameter (see [109]).
- *Learn^{++}.CDS*: Use Learn^{++}.CDS if both minority and majority concepts are drifting, classes contain imbalance and memory considerations are important. Learn^{++}.CDS has a smaller memory requirement than Learn^{++}.NIE, as it does not need to generate subensembles.
- *UCB*: This algorithm, came fourth in our overall ranking, is most suitable if the minority class does not drift, and it is the minority class recall that is the most important figure of merit. UCB generally provided the best minority recall performance, though at the cost of classification accuracy of the majority class.

6.2.2 Transductive Learning Ensembles

The Weight Estimation Algorithm (WEA) was presented for determining classifier-voting weights when concept drift is present with a large amount of unlabelled data. WEA is an incremental ensemble based algorithm that uses both labelled and unlabelled data to determine the classifier voting weights before the data is classified. WEA was compared to Learn^{++}.NSE and empirical results indicate that WEA performs similarly to Learn^{++}.NSE

when there is no bias between the labelled (training) and unlabelled (field) data. However, WEA showed significant improvement when bias was present between the distributions of labelled and unlabelled batches of data.

6.2.3 Drift Detection using Raw Features

HDDDM utilizes the Hellinger distance as a measure to infer whether drift is present between two batches of training data using an adaptive threshold. The adaptive threshold was analyzed using a standard deviation and t-statistic approach. This threshold is computed based on the information obtained from the divergence between training distributions. Preliminary results show that the Hellinger distance drift detection method (HDDDM) presented in this thesis can improve the performance of an incremental learning algorithm by resetting the classifier when a change has been detected. The HDDDM can then be employed with any active concept drift algorithm. The drift detection in HDDDM is not based on classifier error and is a classifier independent approach that can be used with other supervised batch learning algorithms.

6.3 Recommendations for Future Work

6.3.1 Online learning of Under-represented classes in Data Streams

The algorithms presented in this thesis are incremental batch learning algorithms. The assumption is that data will always be presented in batches to learn a specific environment. This condition can not always be met as data may be arriving in a very rapid stream making it difficult to store the data in memory. There is also the issue of generating a classifier a massive dataset. Consider training a support vector machine on an extremely large dataset with 100,000,000 instances. The training of the SVM will become more time consuming as a function of the size of the database. Other classifiers will have similar issues with training on massive datasets (MLPNN, CART, C4.5, etc.). Therefore, the use of an on-line classification algorithms can allow for the ability to perform as well as the algorithms presented in this thesis would be extremely valuable. The Massive Online Analysis (MOA) software suite maintains many different online classification algorithms that are geared towards data stream mining [118]. Most research has focused on a batch based processing scheme for learning an under-represented class; however, little has been done in the way of

online learning of an imbalanced class. Therefore, on-line learning of imbalanced classes in massive data streams would be a meaningful contribution to machine learning.

6.3.2 Semi-Supervised/Transductive Learning in Nonstationary Environments

Semi-supervised learning in nonstationary environments is vastly under-explored in machine learning. The data mining community could benefit from a semi-supervised learning algorithm for nonstationary environments. Although, WEA was presented in Chapter 4, there are areas for possible improvement of semi-supervised learning in nonstationary environments by answering the following questions

- Can theoretical error bounds be produced for WEA?
- How well does WEA perform on a wide array of problems?
- How can imbalanced classes be handled in a semi-supervised/transductive learning scenario?

One possible way to cope with imbalanced classes could be to use approaches in active learning. Active learning will request that a small amount of data be labelled so that an algorithm can continue to labelled unknown data. Using active learning can allow the learn to query a human for the identification of a very small number of minority class instances in the unlabelled set to better learn the most recent minority class data.

References

[1] R. O. Duda, P. E. Hart and D. G. Stork, *Pattern Classification*, 2nd ed. John Wiley & Sons, Inc., 2001.

[2] S. Pinker, *The Blank Slate.* Penguin, 2003.

[3] T. Mitchell, *Machine Learning.* McGraw-Hill, 1997.

[4] C. Bishop, *Pattern Recognition and Machine Learning.* Springer, 2006.

[5] E. Alpaydin, *Introduction to Machine Learning.* MIT Press, 2004.

[6] O. Chapelle, B. Scholkopf and A. Zien, *Semi-Supervised Learning.* MIT Press, 2006.

[7] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on Data and Knowledge Discovery*, vol. 12, no. 9, pp. 1263–1284, 2009.

[8] J. Quinonero-Candela, M. Sugiyama, A. Schwaighofer and N. Lawrence, *Dataset Shift in Machine Learning.* The MIT Press, 2009.

[9] G. A. Carpenter, S. Grossberg, and J. H. Reynolds, "A self-organizing ARTMAP neural architecture for supervised learning and pattern recognition," *Neural Networks: Theory and Applications*, vol. Eds., pp. 43–80, 1991.

[10] G. A. Carpenter, S. Grossberg, and J. Reynolds, "ARTMAP: A self–organizing neural network architecture for fast supervised learning and pattern recognition," in *Proceedings of the International Joint Conference on Neural Networks*, 1991, pp. 863–868.

[11] 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*, vol. 31, no. 4, pp. 497–508, 2001.

[12] M. Muhlbaier and R. Polikar, "Multiple classifiers based incremental learning algorithm for learning nonstationary environments," in *IEEE International Conference on Machine Learning and Cybernetics*, 2007, pp. 3618–3623.

[13] R. Elwell, "An ensemble-based computational approach for incremental learning in non-stationary environments related to schema and scaffolding-based human learning," Master's thesis, Rowan University, 2010.

[14] I. Koychev, "Changing user interests through prior-learning of context," in *Adaptive Hypermedia and Adaptive Web Based Systems.* Springer-Verlag, 2002, pp. 223–232.

[15] I. Žliobaitė, "Adaptive training set formation," Ph.D. dissertation, Vilnius University, 2010.

[16] B. Schlköpf and A. J. Smola, *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*, 1st ed. The MIT Press, 2001.

[17] National Cancer Institute, "Probability of breast cancer in american women," U.S. National Institutes of Health, 2010. [Online]. Available: http://www.cancer.gov/cancertopics/factsheet/Detection/probability-breast-cancer

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

[19] R. Polikar, "Bootstrap inspired techniques in computational intelligence," *IEEE Signal Processing Magazine*, vol. 24, pp. 57–72, 2007.

[20] F. H. Hamker, "Life-long learning cell structures continuously learning without catastrophic forgetting," *Neural Networks*, vol. 14, no. 5, pp. 551–573, 2001.

[21] S. Chen and H. He, "SERA: Selectively recursive approach towards nonstationary imbalanced stream data mining," in *International Joint Conference on Neural Networks*, 2009, pp. 552–529.

[22] L. I. Kuncheva, "Classifier ensembles for changing environments," in *Proceedings 5th International Workshop on Multiple Classifier Systems*, 2004, pp. 1–15.

[23] A. Bifet, "Adaptive learning and mining for data streams and frequent patterns," Phd Thesis, Universitat Politecnica de Catalunya, 2009.

[24] G. A. Carpenter and S. Grossberg, "A massively parallel architecture for a self-organizing neural pattern recognition machine," *Computer Vision, Graphics, and Image Processing*, vol. 37, pp. 54–115, 1987.

[25] G. A. Carpenter and S. Grossberg, "ART 2: Self-organization of stable category recognition codes for analog input patterns," *Applied Optics*, vol. 26, no. 23, pp. 4919–4930, 1987.

[26] G. A. Carpenter, S. Grossberg, and J. H. Reynolds, "Fuzzy ART: Fast stable learning and categorization of analog patterns by an adaptive resonance system," *Neural Networks*, vol. 4, pp. 759–771, 1991.

[27] G. A. Carpenter, S. Grossberg, N. Markuzon, J. Reynolds and D. Rosen, "Fuzzy ARTMAP: A neural network architecture for incremental supervised learning of analog multidimensional maps," *IEEE Transaction on Neural Networks*, vol. 3, pp. 698–713, 1992.

[28] J. Z. Kolter and M. A. Maloof, "Dynamic weighted majority: An ensemble method for drifting concepts," *Journal of Machine Learning Research*, vol. 8, pp. 2755–2790, 2007.

[29] A. Pocock, P. Yiapanis, J. Singer, M. Lujan, and G. Brown, "Online nonstationary boosting," in *International Workshop on Multiple Classifier Systems*, 2010.

[30] R. Kirkby, "Improving Hoeffding trees," PhD Thesis, University of Waikato, 2007.

[31] P. Domingos and G. Hulton, "Mining high-speed data streams," in *Knowledge and Data Discovery*, 2000.

[32] Y. Freund and R. E. Schapire, "A short introduction to boosting," *Journal of Japanese Society for Artificial Intelligence*, vol. 14, no. 5, pp. 771–780, 1999.

[33] Y. Freund and R. Shapire, "A decision-theoretic generalization of online learning and an application to boosting," *Journal of Computer and System Sciences*, vol. 55, pp. 119–139, 1997.

[34] V. Vapnik, *The Nature of Statistical Learning Theory*, 2nd ed. Springer-Verilag, 1999.

[35] G. Widmer and M. Kubat, "Learning in the presence of concept drift and hidden contexts," *Machine Learning*, vol. 23, no. 1, pp. 69–101, 1996.

[36] J. C. Schlimmer and R. H. Granger, "Incremental learning from noisy data," *Machine Learning*, vol. 1, no. 3, pp. 317–384, 1986.

[37] M. Nunez, R. Fidalgo and R. Morales, "Learning in environments with unknown dynamics: Towards more robust concept learners," *Journal of Machine Learning Research*, vol. 8, pp. 2595–2628, 2007.

[38] D. H. Widyantoro, T. R. Iorger, and J. Yen, "An adaptive algorithm for learning changes in user interests," in *Conference on Information and Knowledge Management*, 1999, pp. 405–412.

[39] G. Castillo, "Adaptive learning algorithms for bayesian network classifiers," Ph.D. dissertation, University of Aveiro, 2006.

[40] K. Nishida, "Learning and detecting concept drift," Ph.D. dissertation, Hokkaido University, Japan, 2008.

[41] E. Spinosa, "Novelty detection in data streams," Ph.D. dissertation, University of Sao Paulo, 2008.

[42] D. Widyantoro, "Concept drift learning and its application to adaptive information filtering," Ph.D. dissertation, Texas A & M University, 2003.

[43] D. H. Widyantoro, T. R. Ioerge and J. Yen, "Tracking changes in user interests with a few relevance judgments," in *ACM International Conference on Information and Knowledge Management*, 2003.

[44] S. J. Gong and G. H. Cheng, "Mining user interest change for improving collaborative filtering," in *International Symposium on Intelligent Information Technology Application*, 2008, pp. 24–27.

[45] Z. Chen, Y. Jiang, and Y. Zhao, "A collaborative filtering recommendation algorithm based on user interest change and trust evaluation," *International Journal of Digital Content Technology and its Applications*, vol. 4, no. 9, pp. 106–113, 2010.

[46] Yandex, "Yandex.direct," March 2011. [Online]. Available: http://direct.yandex.com/

[47] United States Department of Commerce, "National oceanic and atmospheric administration," 2010. [Online]. Available: www.noaa.gov/

[48] J. Gao, B. Ding, W. Fan, J. Han, and P. S. Yu, "Classifying data streams with skewed class distributions and concept drifts," *IEEE Internet Computing*, vol. 12, no. 6, pp. 37–49, 2008.

[49] W. N. Street and Y. Kim, "A streaming ensemble algorithm (SEA) for large scale classification," in *Proceedings to the 7th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2001, pp. 377–382.

[50] R. Elwell and R. Polikar, "Incremental learning variable rate concept drift," in *Multiple Classifier Systems (MCS 2009), Lecture Notes in Computer Science, J.A. Benediktsson et al, eds.*, 2009, pp. 142–151.

[51] R. Polikar, "Ensemble based systems in decision making," *IEEE Circuits and Systems Magazine*, vol. 6, no. 3, pp. 21–45, 2006.

[52] L. I. Kuncheva, *Combining Pattern Classifiers: Methods and Algorithms*. John Wiley & Sons, Inc., 2004.

[53] G. Brown, "Ensemble learning," *Encyclopedia of Machine Learning*, vol. ., p. ., 2010.

[54] R. Shapire, "The strength of weak learnability," *Machine Learning*, vol. 5, no. 2, pp. 197–227, 1990.

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

[56] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997.

[57] M. D. Muhlbaier, A. Topalis and R. Polikar, "Ensemble confidence estimates posterior probability," in *Multiple Classifier Systems*, L. N. in Computer Science, Ed., vol. 3541, 2005, pp. 326–335.

[58] M. Ahiskali, "Decision based data fusion of complementary features for the early diagnosis of alzheimer's disease," Master's thesis, Rowan University, 2009.

[59] R. Polikar, J. DePasquale, H. S. Mohammed, G. Brown and L. I. Kuncheva, "Learn++.MF: A random subspace approach for the missing feature problem," *Pattern Recognition*, vol. 43, pp. 3817–3832, 2010.

[60] D. Parikh and R. Polikar, "An ensemble-based incremental learning approach to data fusion," *IEEE Transaction on Systems, Man and Cybernetics*, vol. 37, no. 2, pp. 437–450, 2007.

[61] J. Gao, W. Fan, J. Han, and P. S. Yu, "A general framework for mining concept-drifting data streams with skewed distributions," in *Proceedings of the 7th SIAM International Conference on Data Mining*, 2007, pp. 203–208.

[62] J. Gao, W. Fan and J. Han, "On appropriate assumptions to mine data streams: Analysis and practice," in *7th IEEE International Conference on Data Mining*, 2007, pp. 143–152.

[63] M. Baena-Garcia, J. del Campo-Avila, R. Fidalgo, A. Bifet, R. Gavaldua and R. Morales Bueno, "Early drift detection method," in *International Workshop on Knowledge Discovery from Data Streams*, 2006.

[64] K. Nishida and K. Yamauchi, "Adaptive classifiers-ensemble system for tracking concept drift," in *Proceedings of the Sixth International Conference on Machine Learning and Cyberne tics*, 2007, pp. 3607–3612.

[65] M. Kubat and S. Matwin, "Addressing the curse of imbalanced data sets: One-sided sampling," in *Proceedings of the 14th International conference on Machine Learning*, 1997, pp. 179–186.

[66] S. Haykin, *Neural Networks and Learning Machines*. Pearson, 2009.

[67] N. V. Chawla, N. Japkowicz, and A. Kolcz, "Editorial: Special issue on learning from imbalanced data sets," *Sigkdd Explorations*, vol. 6, no. 1, pp. 1–6, 2004.

[68] M. D. Muhlbaier, A. Topalis and R. Polikar, "Learn++.NC: Combining ensemble of classifiers with dynamically weighted consult-and-vote for efficient incremental learning of new classes," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 152–168, 2009.

[69] G. Widmer and M. Kubat, "Learning flexible concepts from streams of examples: FLORA2," in *Proceedings of the 10th European Conference on Artificial Intelligence*, 1992, pp. 463–467.

[70] J. Z. Kolter and M. A. Maloof, "Dynamic weighted majority: A new ensemble method for tracking concept drift," in *Proceedings of the Third International IEEE Conference on Data Mining*, 2003, pp. 123–130.

[71] N. Littlestone and M.K. Warmuth, "The weighted majority algorithm," *Information and Computation*, vol. 108, pp. 212–261, 1994.

[72] P. E. Utgoff, N. C. Berkman, and J. A. Clouse, "Decision tree induction based on efficient tree restructuring," *Machine Learning*, vol. 29, pp. 5–44, 1997.

[73] A. Blum, "Empirical support for Winnow and Weighted-Majority algorithms: Results on a calendar scheduling domain," *Machine Learning*, vol. 26, pp. 5–23, 1997.

[74] M. A. Maloof and R. S. Michalski, "Incremental learning with partial instance memory," *Artificial Intelligence*, vol. 154, pp. 95–126, 2004.

[75] M. A. Maloof and R. S. Michalski, "Selecting examples for partial memory learning," *Machine Learning*, vol. 41, pp. 27–52, 2000.

[76] N. Oza, "On-line ensemble learning," Ph.D. dissertation, University of California, Berkeley, 2001.

[77] S. Z. Li, Z. Q. Zhang, H. Y. Shum and H. J. Zhang, "Floatboost learning for classi?cation," in *Advances in Neural Information Processing Systems*, 2003.

[78] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.

[79] A. Bifet, G. Holmes, B. Pfahringer, R. Kirkby and R. Gavalda, "New ensemble methods for evolving data streams," in *Knowledge and Data Discovery*, 2009.

[80] A. Bifet, G. Holmes, B. Pfahringer, and R. Gavalda, "Improving adaptive bagging methods for evolving data streams," in *Proceedings of the 1st Asian Conference on Machine Learning: Advances in Machine Learning*, 2009, pp. 27–37.

[81] A. Venkatesan, N. C. Krishnan, and S. Panchanathan, "Cost-sensitive boosting for concept drift," in *European Conference on Machine Learning (ECML/PKDD)*, 2010.

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

[83] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.

[84] L. I. Kuncheva, "Using control charts for detecting concept change in streaming data," Bangor University, Tech. Rep. BCS-TR-001-2009, 2009.

[85] B. Manly and D. Mackenzie, "A cumulative sum type of method for environmental monitoring," *Environmetrics*, vol. 11, pp. 155–166, 2000.

[86] C. Alippi and M. Roveri, "An adaptive CUSUM-based test for signal change detection," in *ISCAS 2006*, 2006, pp. 5752–5755.

[87] C. Alippi and M. Roveri, "Just-in-time adaptive classifiers–part I: Detecting nonstationary changes," *IEEE Transactions on Neural Networks*, vol. 19, no. 7, pp. 1145–1153, 2008.

[88] C. Alippi and M. Roveri, "Just-in-time adaptive classifiers–part II: Designing the classifier," *IEEE Transactions on Neural Networks*, vol. 19, no. 12, pp. 2053–2064, 2008.

[89] C. Alippi and M. Roveri, "Change detection tests using the ICI rule," in *International Joint Conference on Neural Networks*, 2010, pp. 1190–1196.

[90] C. Alippi, G. Boracchi and M. Roveri, "Just in time classifiers: managing the slow drift case," in *International Joint Conference on Neural Networks*, 2009, pp. 114–120.

[91] C. Mesterham, "Tracking linear-threshold concepts with winnow," *Journal of Machine Learning Research*, vol. 4, pp. 819–839, 2003.

[92] J. Gama, P. Medas, G. Castillo, and P. Rodrigues, "Learning with drift detection," *Lecture Notes in Computer Science*, vol. 3741, pp. 286–295, 2004.

[93] D. Cieslak and N. V. Chawla, "A framework for monitoring classifiers performance: When and why failure occurs?" *Knowledge and Information Systems*, vol. 18, no. 1, pp. 83–109, 2009.

[94] F.J. Massey, "The Kolomogorov-Smirnov test for goodness of fit," *Journal of the American Statistical Association*, vol. 18, no. 1, pp. 83–108, 1951.

[95] G. M. Weiss and F. Provost, "The effect of class distribution on classifier learning: An empirical study," Department of Computer Science, Rutgers University, Tech. Rep. ML-TR-44, 2001.

[96] N. V. Chawla, "C4.5 and imbalanced data sets: Investigating the effect of sampling method, probabilistic estimate, and decision tree structure," in *International Conference on Machine Learning*. Workshop on Learning from Imbalanced Datasets, 2003.

[97] G. Batista, R. C. Prati, and M. C. Monard, "A study of the behavior of several methods for balancing machine learning training data," *Sigkdd Explorations*, vol. 6, no. 1, pp. 20–29, 2004.

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

[99] H. Han, W. Y. Wang, and B. H. Mao, "Borderline-smote: A new over-sampling method in imbalanced data sets learning," in *Proceedings of the International Conference on Intelligent Computing*, 2005, pp. 878–887.

[100] H. He, Y. Bai, E. A. Garcia, and S. Li, "Adasyn: Adaptive synthetic sampling approach for imbalanced learning," in *IEEE/INNS Int. Joint Conf. on Neural Networks (IJCNN)*, 2008, pp. 1322–1328.

[101] C. Li, "Classifying imbalanced data using a bagging ensemble variation (BEV)," in *ACMSE*, 2007, pp. 203–208.

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

[103] N. V. Chawla, A. Lazarevic, L. O. Hall and K. W. Bowyer, "SMOTEBoost: Improving prediction of the minority class in boosting," in *7th European Conference on Principles and Practice of Knowledge Discovery in Databases*, 2003, pp. 1–10.

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

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

[106] S. Chen, H. He, K. Li, and S. Sesai, "MuSERA: Multiple selectively recursive approach towards imbalanced stream data mining," in *IEEE/INNS Int. Joint Conf. on Neural Networks / World Congress on Computational Intelligence (IJCNN / WCCI)*, 2010, pp. 2857–2864.

[107] M. Karnick, M. Ahiskali, M. Muhlbaier and R. Polikar, "Incremental learning in non-stationary environments with concept drift using a multiple classifier based approach," in *International Conference on Pattern Recognition (ICPR2008)*, 2008, pp. 1–4.

[108] G. Ditzler, N. V. Chawla, and R. Polikar, "An incremental learning algorithm for nonstationary environments and class imbalance," in *International Conference on Pattern Recognition (ICPR'10)*, 2010, pp. 2997–3000.

[109] G. Ditzler and R. Polikar, "An incremental learning framework for concept drift and class imbalance," in *IEEE/INNS Int. Joint Conf. on Neural Networks / World Congress on Computational Intelligence (IJCNN / WCCI 2010)*, 2010, pp. 736–473.

[110] K. Nishida, K. Yamauchi and T. Omori, "Ace: Adaptive classifiers-ensemble system for concept-drifting environments," in *Multiple Classifier Systems*, 2005, pp. 176–185.

[111] G. Ditzler and R. Polikar, "Hellinger distance based drift detection for nonstationary environments," in *IEEE Symposium on Computational Intelligence in Dynamic and Uncertain Environments (CIDUE)*, 2011, p. to appear.

[112] R. Elwell and R. Polikar, "Incremental learning in nonstationary environments with controlled forgetting," in *International Joint Conference on Neural Networks*, 2009, pp. 771–778.

[113] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Letters*, vol. 27, pp. 861–874, 2006.

[114] C. X. Ling, J. Huang and H. Zhang, "AUC: A statistically consistent and more discriminating measure than accuracy," in *International Conference on Artificial Intelligence*, 2003, pp. 329–341.

[115] J. Hanley and B. J. McNeil, "The meaning and use of the area under a receiver operating characteristic (ROC) curve," *Radiology*, vol. 143, pp. 29–36, 1982.

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

[117] L. Breiman, J. Friedman, R. Olshen and C. Stone, *Classification and Regression Trees*. CRC Press, 1984.

[118] A. Bifet, G. Holmes, R. Kirkby and B. Pfahringer, "MOA: Massive online analysis," *Journal of Machine Learning Research*, vol. 11, pp. 1601–1604, 2010.

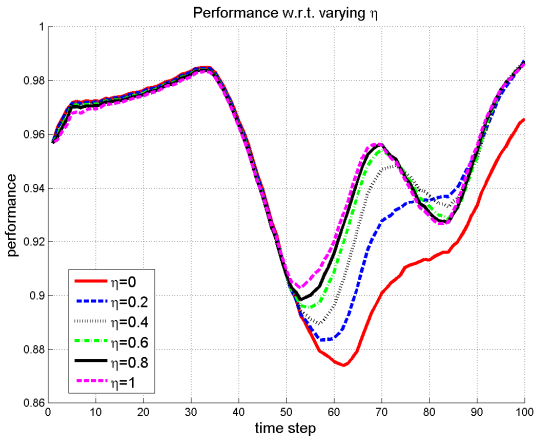
[119] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*. Elsevier, 2005.

[120] M. Harries, "Splice-2 comparative evaluation: Electricity pricing," The University of South Wales, Tech. Rep., 1999.

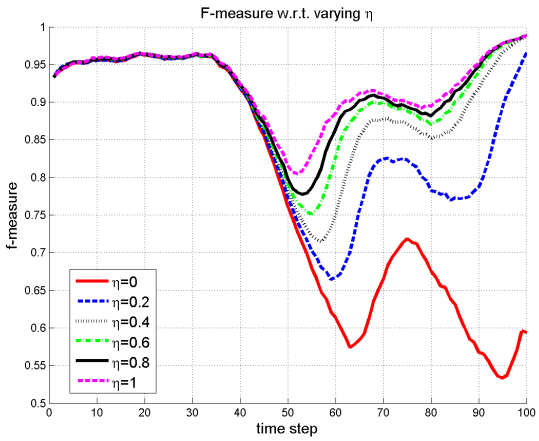
[121] R. L. Iman and J. M. Davenport, "Approximations of the critical region of the friedman statistic," *Communications in Statistics*, pp. 571–595, 1980.

Appendix

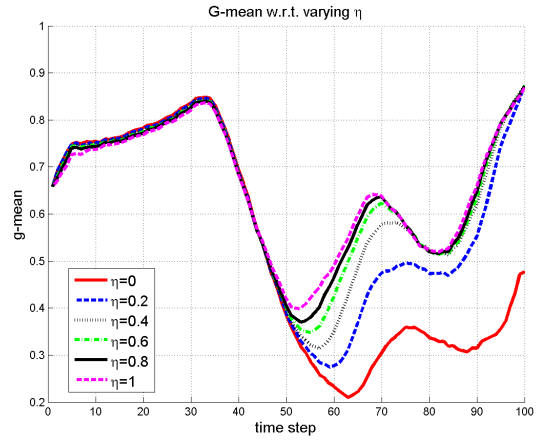
Appendix A: Learn^{++}.NIE η variation



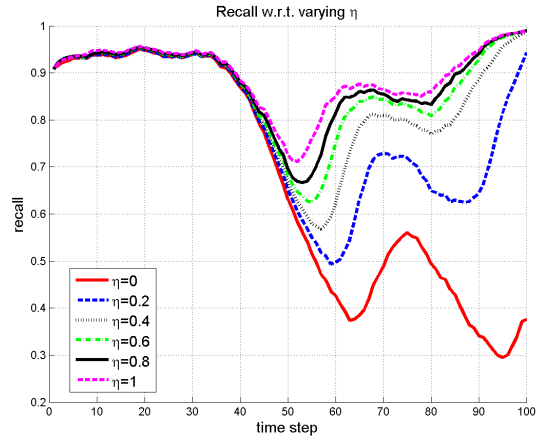
(a) Performance



(b) F-measure

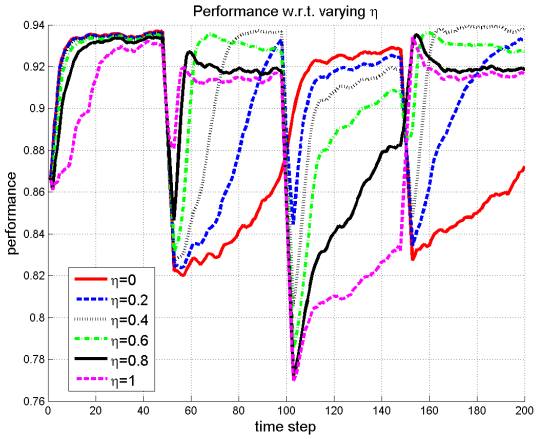


(c) G-mean

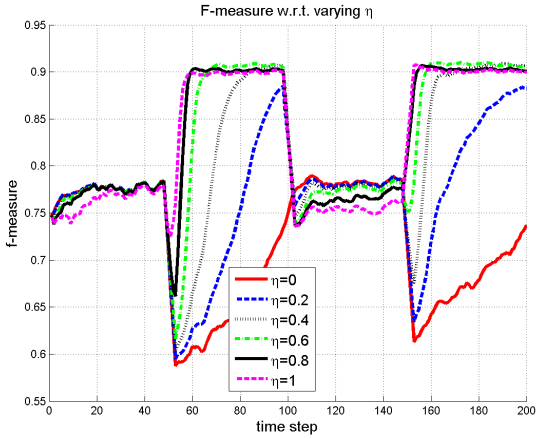


(d) Recall

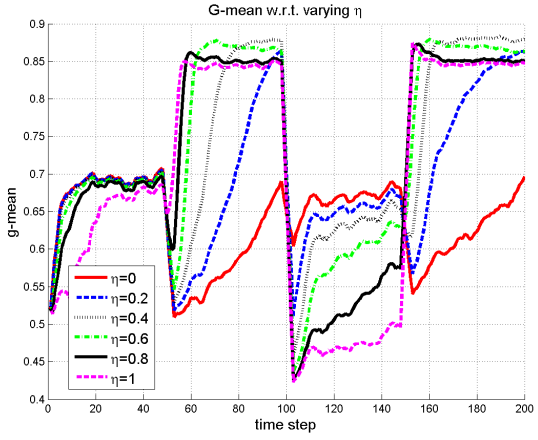
Fig. A.1 : Learn^{++}.NIE η variation on Gaussian data



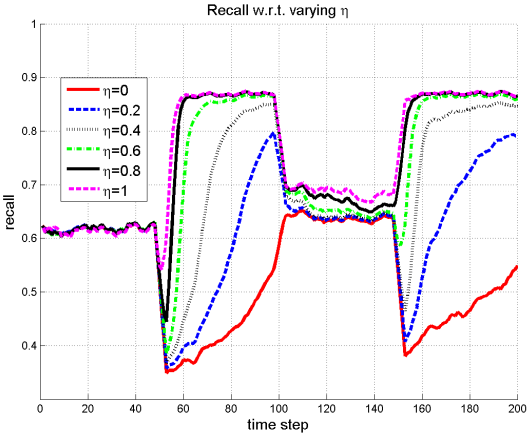
(a) Performance



(b) F-measure

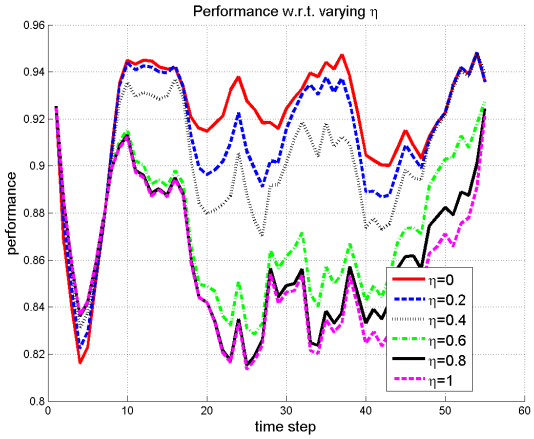


(c) G-mean

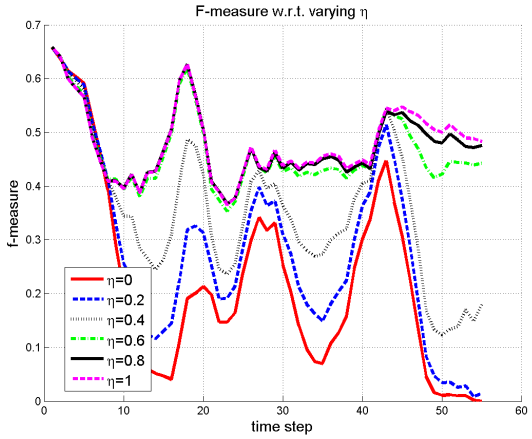


(d) Recall

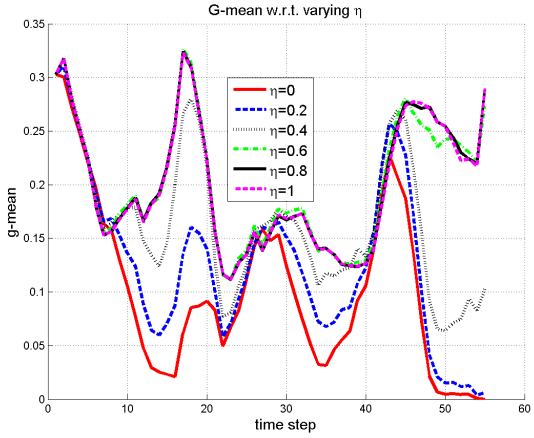
Fig. A.2 : Learn^{++}.NIE η variation on SEA data



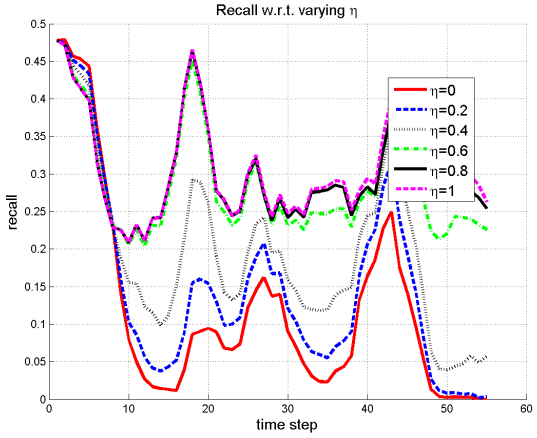
(a) Performance



(b) F-measure

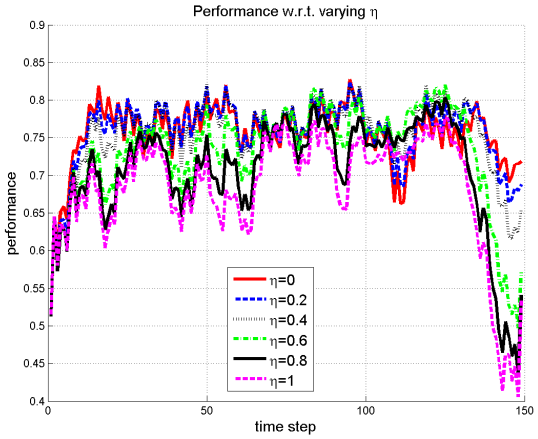


(c) G-mean

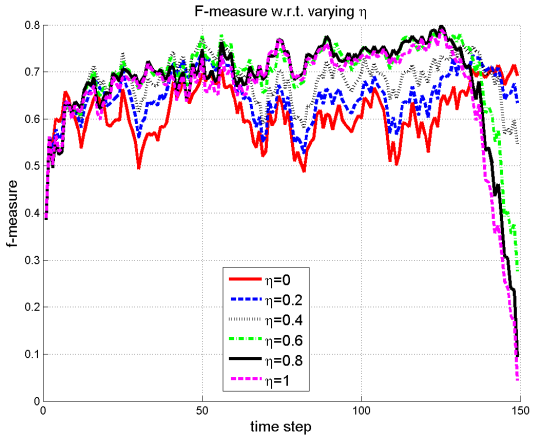


(d) Recall

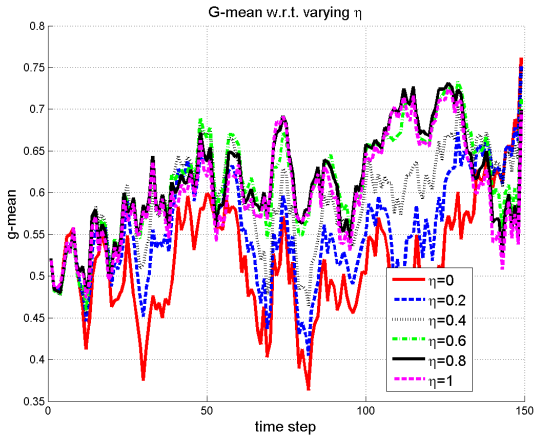
Fig. A.3 : Learn^{++}.NIE η variation on Elec2 data



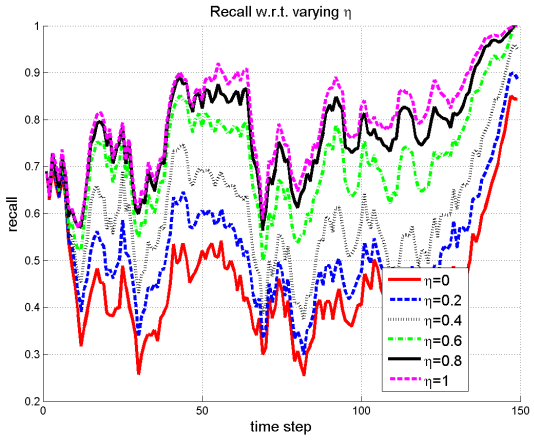
(a) Performance



(b) F-measure



(c) G-mean



(d) Recall

Fig. A.4 : Learn^{++}.NIE η variation on NOAA data