Poznan University of Technology
Faculty of Computing Science
Institute of Computing Science

Doctoral dissertation

BLOCK-BASED AND ONLINE ENSEMBLES FOR CONCEPT-DRIFTING DATA STREAMS

Dariusz Brzeziński

Supervisor
Jerzy Stefanowski, PhD Dr Habil.

Poznań, 2015

To my loving wife, Katarzyna,
for her patience and support

Acknowledgements

This thesis encapsulates my research carried out between October 2010 and February 2015 at the Faculty of Computing Science, Poznan University of Technology. My warmest gratitude goes to all the people who inspired me and helped me complete this dissertation.

I am extremely grateful to my supervisor, Professor Jerzy Stefanowski, for his time, patience, and challenging discussions throughout my scientific journey. His encouragement and constructive comments are what made this thesis possible. I would also like to thank Professor Tadeusz Morzy for the invaluable freedom I had in my research.

Furthermore, I am grateful to my family whose unconditional love and support have always motivated me to work hard and pursue my goals. I would also like to thank Maciej Piernik, Andrzej Stroiński, Dariusz Dwornikowski, Piotr Zierhoffer, and Mateusz Hołenko for making this endeavor a pleasant one.

Finally, I would like to acknowledge the financial support of the Polish National Science Center under Grant No. DEC-2011/03/N/ST6/00360.

Dariusz Brzeziński
Poznan, Poland
March 9, 2015

Contents

Notation

B_j	the j^{th} block of examples
$\mathbf{C}$	degree of consistency
C_i	classifier (the i^{th} ensemble member)
C'	candidate classifier
$\mathbf{D}$	degree of discriminancy
d	block or window size
δ	split confidence
$\mathcal{E}$	ensemble of classifiers
F_F	Friedman test statistic
$f_{iy}(\mathbf{x})$	the probability given by classifier C_i that $\mathbf{x}$ is an instance of class y
$G(\cdot)$	split evaluation function
K_c	class label (the c^{th} class)
k	number of ensemble members
κ	Cohen's Kappa
L	decision tree leaf
m	memory limit
ψ	tie threshold for Hoeffding Tree splits
$Q(\cdot)$	classifier quality measure; weighting function
$\mathcal{S}$	stream of examples
t	timestamp/example number
τ_i	time at which classifier C_i was created
W	window of examples
w_i	the weight of the i^{th} ensemble member
$\mathcal{X}$	attribute set
X_i	the i^{th} attribute
$\mathbf{x}^t$	the t^{th} example
y^t	label of $\mathbf{x}^t$

Acronyms

ACE	Adaptive Classifier Ensemble
ADWIN	Adaptive Windowing algorithm
AOC	Area Over the ROC Curve
ASHT	Adaptive-Size Hoeffding Trees
AUC	Area Under the ROC Curve
AUE	Accuracy Updated Ensemble
AWE	Accuracy Weighted Ensemble
Bag	Online Bagging
BWE	Batch Weighted Ensemble
CD	Critical Difference
CUSUM	Cumulative Sum
DDM	Drift Detection Method
DWM	Dynamic Weighted Majority
ECDD	EWMA for Concept Drift Detection
EDDM	Early Drift Detection Method
EWMA	Exponentially Weighted Moving Average
GMA	Geometric Moving Average
HOT	Hoeffding Option Tree
IFN	Information Network algorithm
Lev	Leveraging Bagging
MOA	Massive Online Analysis framework
MSE	Mean Square Error
MSRT	Multiple Semi-Random decision Trees
NB	Naive Bayes
NIP	Numerical Interleave Pruning
NSE	Learn++.NSE
OAUE	Online Accuracy Updated Ensemble
PH	Page-Hinkley test
ROC	Receiver Operating Characteristic
ROI	Return Of Interest
SEA	Streaming Ensemble Algorithm
UFFT	Ultra Fast Forest of Trees
VFDT	Very Fast Decision Tree
WWH	Weighted Windows with follow the leading History

Chapter 1

Introduction

Due to the growing number of applications of computer systems, vast amounts of digital data related to almost all facets of life are gathered for storage and processing purposes. From traffic control to stock indexes, from microblog posts to supermarket checkouts, modern societies record massive datasets which may contain hidden knowledge. However, due to the volume of the gathered data, that knowledge cannot be extracted manually. That is why, *data mining* methods have been proposed to automatically discover interesting, non-trivial patterns from very large datasets [56, 74, 157, 27]. Typical data mining tasks include association mining, classification, and clustering, all of which have been perfected for over two decades. Nevertheless, data mining algorithms are usually applied to static, complete datasets, while in many new applications one faces the problem of processing massive data volumes in the form of transient data streams.

A *data stream* can be viewed as a potentially unbounded sequence of instances (e.g., call records, web page visits, sensor readings) which arrive continuously with time-varying intensity. Due to the speed and size of data streams, it is often impossible to store instances permanently or process them more than once [3, 81, 62]. Examples of application domains where data needs to be processed in streams include: network monitoring [35], banking [163], traffic control [10], sensor networks [63], disaster management [113], ecology [156], sentiment analysis [152], object tracking [3], and robot vision [139]. The presence of streaming data in this new class of applications has opened an interesting line of research problems, including novel approaches to data mining, called *data stream mining*.

Learning from data streams faces three principal challenges [98]: *speed*, *size*, and *variability*. The speed and size of data streams force algorithms to process data using limited amounts of time and memory, while analyzing each incoming instance only once [49, 155, 99]. Variability, on the other hand, means learning in dynamic environments with changing patterns. The most commonly studied reason of variability in data streams is *concept drift*, i.e., changes in distributions and definitions of learned concepts over time [62]. Such unpredictable changes are reflected in the incoming learning instances and deteriorate the accuracy of algorithms trained from past instances. For example, consider the problem of analyzing a stream of microblog posts concerning a movie in production. Upon changing the actor responsible for the main role, the stream of opinions concerning the movie can quickly become unfavorable. This situation can be considered as a concept drift of the sentiment of several groups of people. An algorithm trained on all available posts will suggest an overly optimistic average opinion about the movie [92, 47]. Therefore, data mining methods that deal with concept drifts are forced to implement forgetting, adaptation, or drift detection mechanisms in order to adjust to changing environments. Moreover, depending on the rate of these changes, concept drifts are usually divided into sudden, gradual, incremental, and recurring ones, all of which require different reactions [159].

Out of several data mining tasks studied in the field of data stream processing [3, 63], classification has received probably the most research attention. The goal of *classification* is to generalize known facts, presented in the form of learning examples, and apply these generalizations to new data [56]. A classification algorithm produces a classifier (model) that can predict the class of new unlabeled instances, by training on instances whose class label is supplied. Although classification has been studied for several decades in the fields of statistics, pattern recognition, machine learning, and data mining [24, 27, 74, 82, 127], streaming applications require new, dedicated, learning techniques. This is caused by the aforementioned speed, size, and variability of data streams, with variability requiring special measures in the context of classification. To tackle these challenges, classifiers for evolving data streams make use of sliding windows, sampling methods, drift detection techniques, and adaptive ensembles [62].

Classifier ensembles are a common technique of enhancing prediction accuracy in static data mining, but were found additionally useful for evolving data. Ensemble algorithms are sets of single classifiers whose predictions are aggregated to produce a final decision [46]. However, due to their modularity, ensembles also provide a natural way of adapting to changes by modifying their structure [100, 101]. Notably, depending on whether they process the stream one example at a time or using larger portions of examples, adaptive ensembles can be divided into online and block-based approaches [62]. The properties, performance, and relation between block-based and online ensembles are the main topic of this thesis.

1.1 Motivation and Problem Statement

There are several real-world applications where data needs to be classified using limited resources. In many cases, this involves processing data incrementally rather than using the entire dataset at once. However, depending on the setting, class labels are available directly after each example or only in larger portions. For example, in traffic control true labels concerning information about congestion are available shortly after making predictions [128]. On the other hand, in the classification of combustion processes in cement plants, samples are accumulated throughout some time, sent to a laboratory, and labeled in blocks [154]. This distinction defines two common trends in data stream classification: one in which algorithms are optimized to work *online*, and another in which classifiers utilize the *block-based* nature of the processing environment.

Adaptive ensembles are among the most studied classifiers for both block-based and online environments. However, the way in which they are designed for each of these two settings differs. Most block-based ensembles periodically evaluate their component classifiers and substitute the weakest ensemble member with a new (candidate) classifier after each block of examples [155, 163]. Such approaches are designed to cope mainly with gradual concept drifts, as they passively forget old concepts rather than actively detect new ones. Furthermore, when training their components, block-based classifiers often take advantage of batch algorithms known from static classification. The main drawback of block-based ensembles is their delay in reactions to sudden concept drifts caused by analyzing true labels only after each full block of examples. Another disadvantage is the difficulty of tuning the block size to offer a compromise between fast reactions to drifts and high accuracy in periods of concept stability.

In contrast to block-based approaches, online ensembles are designed to learn in environments where class labels are available after each incoming example. With labels arriving online, algorithms have the possibility of reacting to concept drifts much faster than in environments where processing is performed in larger blocks of data. Many researchers tackle this problem by designing new online ensemble methods, which are incrementally trained after each instance and try to actively detect concept changes [16, 91, 132]. Most of these newly proposed ensembles are characterized by higher computational costs than block-based methods and the used drift adaptation mechanisms often require problem-specific parameter tuning [34]. Furthermore, online ensembles ignore weighting mechanisms known from block-based algorithms and do not introduce new components periodically and, thus, require specific strategies for frequent updates of incrementally trained components.

The conclusion which can be drawn from analyzing the available stream mining literature is that for both block-based and online settings we still lack classifiers capable of reacting simultaneously to various types of drifts. Algorithms developed for evolving data streams usually concentrate on a single type of change, e.g., only sudden or only gradual drift. Moreover, a clear separation can be noticed — classifiers developed for online environments concentrate on sudden drifts, while methods for reacting to gradual changes are predominant in algorithms for block-based environments.

In order to develop classifiers capable of reacting to several types of drift, factors responsible for the success of particular methods in both settings should be studied. Such an analysis could showcase the possibility of combining the most beneficial properties of each group of algorithms in a single classification method. Adaptive ensembles provide a natural ground for this kind of research, as they are among the most popular classifiers both in block-based and online settings. Furthermore, ensembles in both settings share several architectural similarities, which could facilitate the consolidation of learning mechanisms from different algorithms. To the best of our knowledge, there has been no systematic analysis of relations between block-based and online ensembles in scientific literature.

Based on the above analysis, we formulate the following hypothesis:

Hypothesis. *Novel methods for constructing adaptive data stream ensembles that react to several types of concept drift can be proposed. Such methods can work in block-based as well as online environments and perform comparably to state-of-the-art algorithms, in terms of accuracy, memory usage, and processing time.*

The hypothesis will be verified under the following assumptions. In the block-based setting, we consider each block of examples as a time step. The labels of all examples in a block are available directly after predictions are made. In the online setting, every new testing instance is a time step, the label is not delayed and becomes available after a prediction is made. Where not stated otherwise, we assume equal costs of errors in classification.

Apart from several types of concept drifts that are reflected by changes in class labels, we will also analyze drifts that do not involve modifications of class definitions. Such drifts, often called *virtual*, are connected with distribution changes rather than evolving class-label or attribute-value assignments. A special case of virtual drift involves class distribution changes, i.e., changes in the proportions of examples of each class. In the case of highly imbalanced class distributions, such changes can negatively affect the predictive performance of classifiers.

The problems of distribution changes and class imbalance have already been partially analyzed in traditional data mining. However, the speed and volume of data streams prohibit the use of many algorithms known from batch processing, which makes learning classifiers from imbalanced streams one of the most important challenges in data stream mining. In particular, the number of measures which can be currently used to evaluate classifiers on imbalanced streams is very limited and equivalents of the most popular batch metrics are unavailable. Notably, the area under the Receiver Operating Characteristic (ROC) curve, one of the most popular classifier evaluation measures in traditional data mining, cannot be used on large data streams due to time and memory requirements.

To verify the predictive performance of adaptive ensembles on streams with class distribution changes, we will survey available classifier evaluation methods for data streams. We will also propose and assess an algorithm for calculating the area under the ROC curve online with a forgetting mechanism. Consequently, the proposed algorithm will help verify the hypothesis of this dissertation for class ratio changes as a special case of virtual drift.

1.2 Main Contributions

The main contributions of this thesis to the field of data stream classification are:

1. The thesis advances methods for introducing elements of incremental learning in block-based ensembles. As a result, the Accuracy Updated Ensemble (AUE) algorithm is developed and experimentally validated. The proposed algorithm presents higher average predictive performance under sudden, gradual, incremental, recurring, and no drifts, as compared to competitive adaptive learning algorithms.
2. The thesis contributes to the understanding of adaptive block-based and online ensembles in general and the relations between their concept drift reaction mechanisms in particular. We verify if it is possible to transform block-based ensembles into online learners and propose three general strategies to achieve this goal:

    a) the use of a windowing technique which updates component weights after each example,

   b) the extension of the ensemble by an incremental classifier which is trained between component reweighting,
   c) and the use of an online drift detector which allows to shorten drift reaction times.

3. Based on the analysis of ensemble transformation strategies, we introduce a new error-based weighting function, which evaluates component classifiers online as they classify incoming examples. Furthermore, we put forward the Online Accuracy Updated Ensemble (OAUE), an algorithm which uses the proposed function to incrementally train and weight component classifiers. The OAUE algorithm is experimentally compared with selected online ensembles on several real and synthetic datasets simulating environments containing sudden, gradual, incremental, and mixed drifts.
4. We survey existing methods for evaluating data stream classifiers. The study highlights problems in applicability of existing evaluation measures in the event of class distribution changes, which can be regarded as a special case of virtual concept drift. In this context, we propose an efficient algorithm for computing a time-oriented area under the Receiver Operating Characteristic curve, called Prequential AUC. Finally, we analyze the properties of Prequential AUC as a new performance metric and use it as a complementary measure for evaluating the predictions of adaptive ensembles on drifting class-imbalanced streams.

Several of the contributions presented in this thesis have already been published in scientific journals. The author's publications related to this dissertation are listed in Appendix B.

1.3 Thesis Structure

The chapters that build the thesis are organized as follows.

Chapter 2 introduces basic definitions and terminology. We define the notion of classification, block-based and online processing, data streams, and concept drift. Moreover, we discuss related works in the field of drift reaction strategies and data stream classification, in particular ensemble classifiers for concept-drifting data streams.

Chapter 3 focuses on block-based processing of data streams and discusses limitations of existing ensemble classification algorithms. We propose a new data stream classifier, called the Accuracy Updated Ensemble, which aims at reacting equally well to several types of drift. The proposed algorithm is experimentally compared with state-of-the-art stream methods in different drift scenarios.

Chapter 4 analyzes if and how the characteristics of block and incremental processing can be combined to produce accurate ensemble classifiers. We propose and experimentally evaluate three strategies to transforming a block-based ensemble into an online learner: the use of a sliding window, an additional incrementally trained ensemble member, and a drift detector.

Chapter 5 focuses on online classification using adaptive ensembles. We analyze possible online component weighting schemes and their influence on drift reaction. As a result, we introduce and experimentally evaluate a new incremental ensemble classifier, called Online Accuracy Updated Ensemble, which uses an efficient weighting function based on the mean square error of components.

Chapter 6 analyzes the predictive performance of adaptive ensembles in the context of class distribution changes as special case of virtual concept drift. We highlight problems with existing evaluation methods used for streams with such changes and, more generally, streams with class imbalance. As a result, we suggest a complementary measure for scoring classifiers learning from class-imbalanced data, called Prequential AUC, which is later used to evaluate adaptive ensembles on streams with changing class distributions.

Chapter 7 summarizes the contributions of this thesis and concludes with a discussion on lines of future research in the field of data stream classification.

Chapter 2

Data Stream Classification

In recent years, a lot of research attention has been given to data streams and the problem of concept drift. Scientists have categorized concept changes based on their frequency, speed, and severity, and proposed several drift detection mechanisms. Furthermore, research on concept drift combined with efficient stream processing methods have led to the development of several classification algorithms designed to cope with evolving data, such as: sliding window approaches, online algorithms, drift detection techniques, and adaptive ensembles.

This chapter aims at providing basic definitions and reviewing existing works related to the field of data stream classification. The subsequent sections are organized as follows. Section 2.1 introduces basic terminology concerning classification, data streams, and online processing. In Section 2.2, we formally define the problem of concept drift, provide a taxonomy of drifts, and give real world examples of concept changes. Finally, in Section 2.3 we discuss state-of-the-art works in the field of drift reaction strategies and data stream classifiers.

2.1 Definitions and Terminology

The data mining task analyzed in this thesis is supervised classification, which can be described as the problem of assigning objects to one of several predefined classes. The input data for classification is a collection of objects, also called *examples* or *instances*. Each example is characterized by a tuple $\{\mathbf{x}, y\}$, where $\mathbf{x}$ is a set of attributes describing an object and y is the object's *class label*, i.e., a special attribute which falls into one of several categorical values ($y \in \{K_1, \ldots, K_c\}$, where c is the number of predefined classes). More formally, classification can be defined as follows [157]:

Definition 2.1. *Classification* is the task of learning a target function C that maps each attribute set $\mathbf{x}$ to one of the predefined class labels y.

Classification tasks are solved by means of induction using *classification algorithms*, also called *learning algorithms* or *learners*. In this thesis, we will discuss classification in the context of predictive modeling, where the discovered target function (also called a *model* or *classifier*) is used to predict class labels of unknown objects. Furthermore, we will use the term *concept* as a synonym of a description of a class that distinguishes it from other classes.

Real-world classification tasks include, for example, spam detection based on email text [43], hand movement predictions based on EEG signals [114], and anticipating flight delay based on the time of day, airline company, and route [22]. An illustrative dataset concerning this last example is presented in Table 2.1.

Table 2.1: Sample from the airlines dataset

Airline	**Flight**	**From**	**To**	**Day of week**	**Time**	**Length**	**Delayed?**
CO	269	SFO	IAH	Wednesday	15	205	yes
US	1558	PHX	CLT	Wednesday	15	222	yes
AA	2400	LAX	DFW	Wednesday	20	165	yes
AA	2466	SFO	DFW	Wednesday	20	195	yes
AS	108	ANC	SEA	Wednesday	30	202	no
CO	1094	LAX	IAH	Wednesday	30	181	yes
DL	1768	LAX	MSP	Wednesday	30	220	no
DL	2722	PHX	DTW	Wednesday	30	228	no
DL	2606	SFO	MSP	Wednesday	35	216	yes
AA	2538	LAS	ORD	Wednesday	40	200	yes
CO	223	ANC	SEA	Wednesday	49	201	yes
DL	1646	PHX	ATL	Wednesday	50	212	yes
DL	2055	SLC	ATL	Wednesday	50	210	no
AA	2408	LAX	DFW	Wednesday	55	170	no
AS	132	ANC	PDX	Wednesday	55	215	no
US	498	DEN	CLT	Wednesday	55	179	no
B6	98	DEN	JFK	Wednesday	59	213	no

In the presented data, the attribute set includes seven properties of a flight: its number, departure and arrival port, day of week, travel time, and distance. The class label is a discrete attribute stating if a given flight was delayed. The aim of the learning algorithm here is to find a function that is consistent with the presented dataset (usually called the *training* or *learning* dataset) and can also be used to provide a delayed/not-delayed prediction for future flights. By consistent we mean that the discovered function should, in most cases, agree with the flight status, given the attribute values provided in the training dataset.

From the learning dataset, the classification algorithm could infer that "any flight managed by CO airlines is delayed". Such a function can then be applied to any new flight described by the same seven attributes as those present in the training dataset. However, it is worth noticing that there are many different functions that could be inferred from the given data. Moreover, the discovered functions can be represented in many forms, such as: *rules*, *decision trees*, *associations*, *linear and nonlinear functions*, *conditional probabilities*, or *neural networks* [127, 24, 74, 115]. The discovered functions can be used to describe knowledge hidden in the data, but, as mentioned earlier, we will focus on predictive modeling where they are used to classify unseen examples.

With many learning algorithms and many possible output functions at hand, an evaluation criterion is needed to choose the best possible model for a given classification task. The main factor considered while choosing a classifier is its *predictive performance*. Predictive performance can be analyzed by a simple empirical *error-rate*, i.e. the fraction of misclassified examples, or its complement called *accuracy*, i.e., the fraction of correctly classified examples.

Accuracy can be measured directly on the training data, by verifying the number of examples for which the classifier output matches the true class label. For example, if we were to use the rule "any flight managed by CO airlines is delayed" and assume all flights that do not match this rule are on time, we would get 11 matches for 17 instances in Table 2.1, which would yield 64.7% accuracy. However, in order to avoid overfitting to the learning data, classifiers are usually evaluated on examples other than those on which they where trained. This involves separating the dataset into training and testing instances [82].

Traditionally, classification tasks are analyzed in the context of static datasets, where all training and testing examples are available at once, and can be analyzed multiple times. In contrast to such batch processing, in this thesis we will consider examples arriving in the form of a data stream.

Definition 2.2. A *data stream* $\mathcal{S}$ is an ordered, potentially infinite, sequence of instances $\mathbf{x}^t$ $(t = 1, 2, \ldots, T)$ that arrive at a rate that does not permit their permanent storage in memory.

We will consider a completely supervised framework, where an incoming example $\mathbf{x}^t$ is classified by a classifier C which predicts its class label. We assume that after some time the true class y^t of this example is available and the classifier can use it as additional learning information. Thus, we do not consider other forms of learning as, e.g., a semi-supervised framework where labels are not available for all incoming examples [120, 54, 89].

Due to their speed and size, data streams imply several constraints on classification algorithms [87, 11, 62]:

1. It is impossible to store all the data from the data stream in memory. Only small summaries of data streams can be computed and stored, and the rest of the information is disposed of.

2. The arrival speed of data stream examples forces each particular instance to be processed only once, in real time, and then discarded.

3. The distribution generating the examples can change over time, thus, data from the past may become irrelevant or even harmful for the current summary.

Constraint 1 limits the amount of memory that algorithms operating on data streams can use, while constraint 2 limits the time in which an item can be processed. The first two constraints led to the development of windowing and summarization techniques. On the other hand, constraint 3 is crucial primarily for learning algorithms, as they need to predict future examples, and outdated information deteriorates the accuracy of classifiers. Many of the first data stream mining approaches ignored this characteristic and formed the group of *stationary data stream learning* algorithms [3]. Other studies acknowledged the third constraint as a key feature and devoted their work to *evolving data stream learning*. In this thesis, we consider concept changes as a key characteristic of data streams and will focus mainly on algorithms and techniques designed for evolving data streams.

Examples can be read from a data stream either incrementally (*online*) or in portions (*blocks*). In the first approach, algorithms process single examples appearing one by one in consecutive moments in time, while in the second approach, examples are available only in larger sets called data blocks (or data chunks). Blocks $B_1, B_2, \ldots, B_j$ are usually of equal size and the construction, evaluation, or updating of classifiers is done when all examples from a new block are available.

In this thesis, we will assume that in online processing the true label y^t for example $\mathbf{x}^t$ is available before the arrival of $\mathbf{x}^{t+1}$. Conversely, in block processing, we will assume that instances are labeled in blocks and true labels for examples in B_j are available before the subsequent block B_{j+1} arrives. Online processing is sometimes called *instance incremental processing*, while block-based approaches can also be denoted as *batch incremental*. Figures 2.1 and 2.2 present the workflow of both processing schemes.

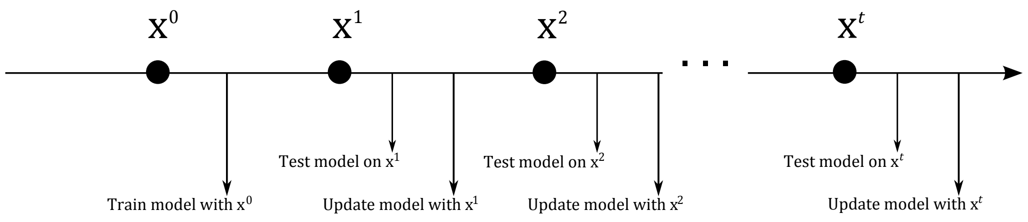


Figure 2.1: Online processing

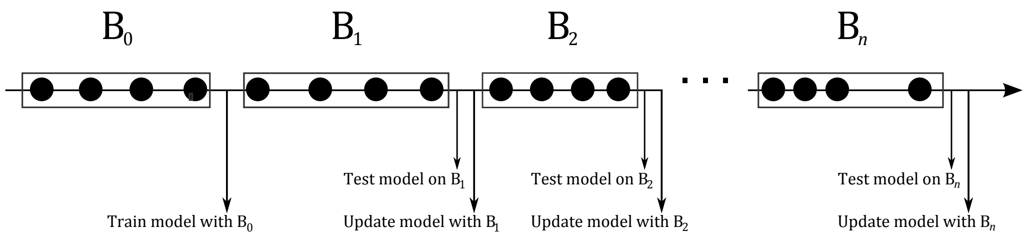


Figure 2.2: Block processing

Definition 2.3. A data stream $\mathcal{S}$ is processed *online* by a classifier C, *iff*, for each example $\mathbf{x}^t \in \mathcal{S}$, C classifies example $\mathbf{x}^t$ and updates its model before example $\mathbf{x}^{t+1}$ arrives.

Definition 2.4. A data stream $\mathcal{S}$ is processed in *blocks* by a classifier C, *iff*, for each block of examples $B_j \in \mathcal{S}$, C classifies all examples in B_j and updates its model before block B_{j+1} arrives.

Online processing can be regarded as a special case of block processing where the size of each block $|B_j| = 1$. However, it is worth noticing that contrary to online processing, in block processing several instances are available at the same time, thus, allowing to identify patterns in groups of consecutive examples.

2.2 Concept Drift

Standard batch classification algorithms assume that examples are generated at random according to some stationary probability distribution. However, one of the most important properties of data streams is that they can change over time. Therefore, classifiers for data streams need to be capable of predicting, detecting, and adapting to concept changes. In order to do so, the nature of changes needs to be studied, including their rate, cause, predictability and severity [70].

According to the Bayesian Decision Theory [51], a classification model can be described by the prior probabilities of classes $p(y)$ and class conditional probabilities $p(y|\mathbf{x})$, for all classes $y \in \{K_1, \ldots, K_c\}$, where c is the number of predefined classes. The dynamic nature of data streams is reflected by changes in these probability distributions in an event called *concept drift*. In practical terms, concept drift means that the concept about which data is being collected may shift from time to time after some minimal stability period [62]. Depending on the research area, concept drift can sometimes be referred to as *temporal evolution*, *population drift*, *covariate shift*, or *non-stationarity*. Most studies assume that concept drifts occur unexpectedly and are unpredictable, in contrast to seasonal changes. However, concept drift adaptation mechanisms often entail solutions for cases where changes can be anticipated in correlation with environmental events. Formally, concept drift can be defined as follows [70]:

Definition 2.5. For a given data stream $\mathcal{S}$, we say that *concept drift* occurs between two distinct points in time, t and $t + \Delta$, *iff* $\exists \mathbf{x} : p^t(\mathbf{x}, y) \neq p^{t+\Delta}(\mathbf{x}, y)$, where p^t denotes the joint distribution at time t between the set of input attributes and the class label.

Using this definition, changes in data can be characterized by changes in components of the above relation [86, 71]:

- prior probabilities of classes $p(y)$ can change,
- class conditional probabilities $p(\mathbf{x}|y)$ can change,
- as a result, posterior probabilities of classes $p(y|\mathbf{x})$ may (or may not) change.

Based on the cause and effect of these changes, two types of drift are distinguished: *real drift* and *virtual drift* [70].

Real drift is defined as changes in $p(y|\mathbf{x})$. It is worth noticing that such changes can occur with or without changes in $p(\mathbf{x})$, therefore, they may or may not be visible from the data distribution without knowing the true class labels. Such a distinction is crucial, as some methods attempt to detect concept drifts using solely attribute values [54]. Real drift has also been referred to as *concept shift* [148] and *conditional change* [71].

Virtual drift is usually defined as changes in the attribute-value $p(\mathbf{x})$ or class $p(y)$ distributions that do not affect $p(y|\mathbf{x})$ [44, 159, 167]. However, the source and therefore the interpretation of such changes differs among authors. Widmer and Kubat [167] attributed virtual drift to incomplete data representation rather than true changes in concepts. Tsymbal [159] on the other hand defined virtual drift as changes in the data distribution that change the decision boundary, while Delany [44] described it as a drift that does not affect the target concept. Furthermore, virtual drifts have also been called *temporary drifts* [106], *sampling shifts* [148], and *feature changes* [71].

To illustrate the difference between real and virtual drifts, let us recall the example classification problem from Table 2.1, where the task was to determine whether a given flight will be delayed or not. If an airline company changes flight hours, but it does not affect their delay, such a change is regarded as virtual drift. Similarly, if due to a crisis companies change the frequency of certain flights without any effect on their delays, this would also correspond to a virtual drift. However, if some flights become regularly delayed even though they used to be on time, real drift is occurring. It may happen that all of the aforementioned types of changes take place at the same time.

The difference between real and virtual drifts is also illustrated in Figure 2.3. The plot shows that only real concept drifts change the class boundary making any previously created model obsolete. The illustrated real drift occurs without any changes in the attribute space, however, in practice changes in prior probabilities may appear in combination with real drift.

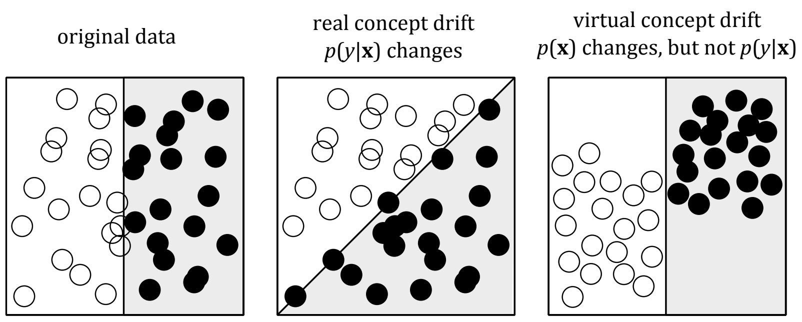

original data
real concept drift
$p(y|\mathbf{x})$ changes
virtual concept drift
$p(\mathbf{x})$ changes, but not $p(y|\mathbf{x})$

Figure 2.3: Types of drift [70]. Circles represent examples in a two-dimensional attribute space, different colors represent different classes.

As we will be mostly interested in the effect of concept drift on classification, we will focus on methods that use true class labels to detect drift. We will, therefore, concentrate mainly on real drifts regardless of whether they are visible from the input data distribution $p(\mathbf{x})$. However, we will also study classifier reactions to class distribution changes, as a special case of virtual drift. Specialized methods for tracking changes using solely attribute values are analyzed more thoroughly in the fields of novelty detection [116, 118, 119] and semi-supervised learning from data streams [2, 89, 120].

Apart from differences in the cause and effect of concept changes, researchers distinguish between several ways of *how* such changes occur. In this aspect, drifts can be further characterized, for example, by their permanence, severity, predictability, and frequency [106, 125, 97]. However, the most analyzed aspect of drifts is the way they manifest themselves over time [62, 99, 159, 166, 175].

Figure 2.4 shows six basic structural types of changes that may occur over time. The first plot shows a *sudden* (also called *abrupt*) drift that instantly and irreversibly changes the variable's class assignment. A sudden drift occurs when at a moment in time t the source distribution p^t is suddenly replaced by a different distribution in $t+1$. Abrupt drifts directly deteriorate the classification abilities of a classifier, as a once generated classifier

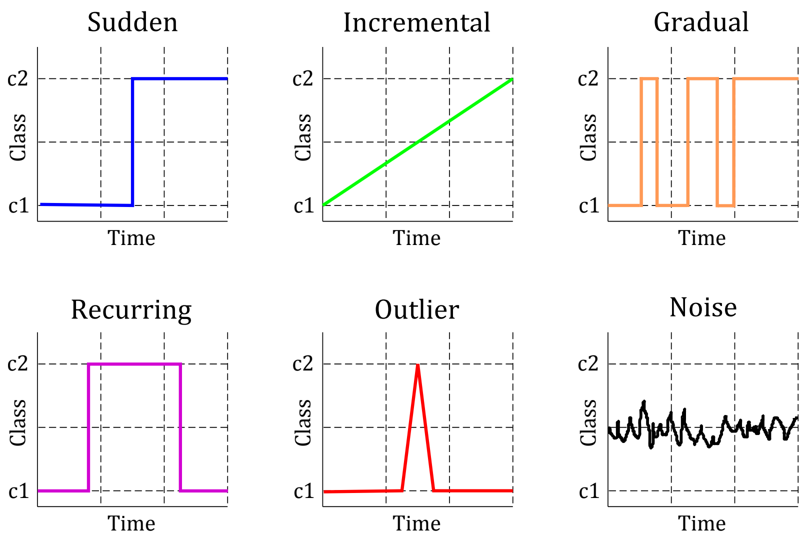


Figure 2.4: Types of changes over time [174]

has been trained on a different class distribution. *Gradual* drifts are not so radical and are connected with a slower rate of changes. More formally, gradual drift refers to a transition phase where examples from two different distributions p^t and $p^{t+\Delta}$ are mixed. As time goes on, the probability of observing examples from p^t decreases, while that of examples from $p^{t+\Delta}$ increases. A different type of moderate changes, which we will refer to as *incremental*, includes more than two sources, however the difference between them is small and the change is noticed only after a longer period of time [175, 125]. Yet another type of drift concerns *recurrent concepts*, i.e., previously active concepts that may reappear after some time. Moreover, some authors distinguish *outliers* (or *blips*), which represent "rare events" in a stable distribution. Outliers as well as noise are examples of anomalies, which are not considered as concept drift and should be ignored as the change they represent is random. Therefore, a good data stream classifier should be capable of combining robustness to noise with sensitivity to drifts.

It is important to note that the presented types of drift are not exhaustive and that in real life situations concept drifts are a complex combination of many types of drift. If a data stream of length t has just two data generating sources with distributions p and p', the number of possible change patterns is 2^t. Since data streams are possibly unbounded, the number of source distribution changes can be infinite. Nevertheless, it is important to identify structural types of drift, since assumptions about the nature of changes are crucial for designing adaptation strategies.

The problem of concept drift has not only been analyzed theoretically, but has also been recognized and addressed in multiple application areas. For example, concept drift is a common problem in monitoring systems, which need to distinguish unwanted situations from "normal behavior". This includes the detection of unwanted computer access, also called intrusion detection, where adversary actions taken by the intruder evolve with time, to surpass the also evolving security systems [102, 121, 135]. Similar systems are required in telecommunication [123, 77] and finance [50]. Drift detection techniques can also be employed to monitor and forecast traffic states and public transportation. Human driver factors and traffic patterns can evolve seasonally as well as permanently, thus the systems have to be able to handle concept drift [124]. Furthermore, there are several applications in the area of sensor monitoring where large numbers of sensors are distributed in the physical world and generate streams of data that need to be combined, tracked, and analyzed [5, 63, 9]. Such systems are used to control the work of machine operators and to detect system faults. In the first case, human factors are the main source of concept drift, while in the second, the change of the system's context [136, 61, 161].

Apart from monitoring applications, concept drift affects many personal assistance systems. This includes, for example, classifying news feeds, where drifting user interests can be a cause of reoccurring contexts in such systems [85, 23]. Similarly, spam filters need to evolve according to seasonality, adaptive adversaries, and changes in user preferences [111]. Although not strictly connected to data stream processing, modern recommender systems also suffer from drift, mainly due to the change of product popularity over time, the drift of users' rating scale, and changes in user preferences [8, 93]. Moreover, different types of changes affect the task of sentiment classification, where customer feedback is analyzed online based on streams of opinions posted on social media [12, 152, 18].

Finally, concept drifts occur in many decision support and artificial intelligence systems. Bankruptcy prediction or individual credit scoring are examples of applications where drift occurs due to hidden context [163]. Biomedical applications present another interesting field of concept drift research due to the adaptive nature of microorganisms. For example, as microorganisms mutate, their resistance to antibiotics changes [160]. Other medical applications include changes in disease progression, discovering emerging resistance, and monitoring nonsomnical infections [153, 164]. Concept drift also occurs in robot vision and image recognition applications, such as biometric authentication, road image classification, and robot navigation [158, 104]. Furthermore, intelligent household appliances need to be adaptive to changing environments and user needs [175]. Lastly, virtual reality requires mechanisms to take concept drift into account. Computer games and flight simulators should adapt to the skills of different users and prevent adversary actions like cheating [36].

The number of real-world applications that need to deal with concept drift showcases the demand for adaptive classification algorithms. The following section presents a review of classifiers designed to tackle concept-drifting data streams.

2.3 Classifiers for Concept-drifting Data Streams

Various categorizations of methods for handling concept drift in data streams have been proposed [62, 99, 159, 175, 70]. For the purposes of this thesis, we will discuss four categories most related to our research:

- single classifiers,
- windowing techniques,
- drift detectors,
- and ensemble methods.

Single classifiers are algorithms known from static learning that can be adapted to cope with evolving data streams. *Windowing techniques* provide a simple forgetting mechanism by selecting examples introduced to the learning algorithm, thus eliminating those examples that come from old concept distributions. A different idea stands behind trigger approaches, which are based on *drift detectors* that react to concept changes and alarm when the classifier should be rebuilt or updated. Lastly, *classifier ensembles* provide a way of adapting to changes by modifying ensemble components or their aggregation method. In the following sections, we discuss algorithms falling into all four categories.

2.3.1 Single Classifiers

Some of the popular classifiers proposed for stationary data fulfill basic stream mining requirements, i.e., they have the qualities of an online learner and some sort of forgetting mechanism. Moreover, some algorithms that are capable of processing data sequentially, but do not adapt, can be easily modified to react to changes. Below, we discuss fives types of learners that fall into these groups: neural networks, Naive Bayes, nearest neighbor methods, rule learners, and decision trees.

Neural networks

In static (batch) data mining applications, neural networks are incrementally trained using the epoch protocol. The entire set of examples is sequentially passed through the network a defined number of times (*epochs*) causing neuron weights to be updated; in the most popular multilayer network according to the backpropagation algorithm [168]. Presenting the data in several epochs allows the neural network to adjust to the presented concept and gradually improve classification accuracy.

By abandoning the epoch protocol, and presenting examples in a single pass, neural networks can be adapted to data stream environments. Because each example is seen only once and neuron weights are updated usually in constant time, such a modification fulfills time requirements set by data streams. Most neural networks are fixed, meaning they do not alter their number of neurons or architecture, thus the amount of memory necessary to use the learner is also constant. Furthermore, forgetting is a natural consequence of abandoning the epoch protocol. When not presenting the same examples multiple times, the network will change according to the incoming examples, thus reacting to concept drift. The rate of this reaction can be adjusted by the learning rate of the backpropagation algorithm. Examples of neural networks specialized for data streams include cluster-based neural networks [68] and evolving granular neural networks [107, 108].

Naive Bayes

The Naive Bayes algorithm is based on Bayes' theorem and computes class-conditional probabilities for each new example. Bayesian methods can learn incrementally and require constant memory. However, Naive Bayes is a *lossless classifier*, meaning it "produces a classifier functionally equivalent to the corresponding classifier trained on the batch data" [99].

To add a forgetting mechanism, sliding windows are usually employed to “unlearn” the oldest examples.

A single Naive Bayes model will generally not be as accurate as more complex models [34]. However, Bayesian networks, which are more sophisticated and give better results, are also suited to the data stream setting; it is only necessary to dynamically learn their structure [26]. Finally, the Naive Bayes algorithm is often a subcomponent of more complex methods such as decision trees for data streams [67, 66, 87].

Nearest neighbor classifiers

Nearest neighbor classifiers, also called *instance-based learners* or *lazy learners*, provide a natural way of learning data incrementally. Each processed example is stored and serves as a reference for new data points. Classification is based on the labels of the nearest historical examples. In this, lossless, version of the nearest neighbor algorithm called IB1 [4], the reference set grows with each example increasing memory requirements and classification time. A different method from this family called IB3 [4], limits the number of stored historical data points only to the most “usefull” for the classification process. Apart from reducing time and memory requirements, the size limitation of the reference set provides a forgetting mechanism as it removes outdated examples from the model.

A more recent example of using the nearest neighbor strategy to classify streaming data is the ANNCAD algorithm [105]. In ANNCAD, the authors propose to divide the feature space several times to create a multi-resolution data representation, where finer levels contain more training points than coarser levels. Predictions are made according to the majority of nearest neighbors starting at finer levels. If the finer levels give an inconclusive predictions, coarser levels are used. Concept drift is addressed by using a fading factor, which decreases the weight of older training examples.

Rule learners

Rule-based algorithms can also be adjusted to data stream environments. Decision rule classifiers consist of rules, i.e., disjoint components of the model that can be evaluated in isolation and removed from the model without major disruption. However, rules may be computationally expensive to maintain, as a drift of a single class can affect many decision rules. These observations served as a basis for developing complex data stream mining systems like FLORA [166], SCALLOP [57], and FACIL [63]. These systems learn rules incrementally and employ dynamic windows to provide a forgetting mechanism [40]. A different approach to creating classification rules from evolving data streams is the Adaptive Very Fast Decision Rules algorithm [94, 95, 96], which uses a structure similar to a decision tree to create rules, and rule-specific drift detectors to react to changes. Finally, one of the most recent rule-based learners called RILL [42], groups examples similarly to instance-based learners and generalizes these groups into rules which can evolve over time.

Decision trees

Decision trees were one of the first classical static learning algorithms to be adapted to data stream mining by using the Hoeffding bound. The Hoeffding bound states that with probability $1 - \delta$, the true mean of a random variable of range R will not differ from the estimated mean after n independent observations by more than:

$$\epsilon = \sqrt{\frac{R^2 ln(1/\delta)}{2n}}. \tag{2.1}$$

Using this bound, Domingos and Hulten [49] proposed a classifier called Very Fast Decision Tree (VFDT). Although the VFDT algorithm is among the most cited works in data stream mining, recent studies have shown that the Hoeffding bound in VFDT was used incorrectly [147, 122]. As a result, the number of samples required to make a proper split in the Hoeffding Tree (and several similar algorithms [80, 67, 94, 83]) is estimated imprecisely. However, the correct formulas for calculating split points, depending on the split function, can produce values close to the Hoeffding bound, which explains its practical efficiency despite its incorrectness [147].

In the following paragraphs, we will refer to the classical VFDT and its modifications, as these were the first algorithms used to adapt decision trees to data stream processing. However, it is important to remember that formulas for calculating the split criterion in these algorithms are imprecise and there are currently more accurate ways of creating decision trees from data streams [147, 122, 146, 145].

Algorithm 2.1 presents the pseudo-code for VFDT. As in this thesis we discuss algorithms that have the property of any-time learning, the pseudo-codes do not contain explicit return statements. We assume that the output classifier is available at any moment of the input stream and is able to provide a prediction after each example.

The algorithm induces a decision tree from a data stream incrementally, without the need for storing examples after they have been used to update the tree. It works similarly to the classic tree induction algorithm [141, 28, 142] and differs mainly in the selection of the split attribute. Instead of selecting the best attribute (in terms of a split evaluation function $G(\cdot)$) after viewing all the examples, it uses the Hoeffding bound (in more recent versions the McDiarmid bound [147]) to calculate the number of examples necessary to select the right split-node with probability $1 - \delta$.

Many enhancements to the basic VFDT algorithm have been proposed. Domingos and Hulten [49] introduced a method of limiting memory usage. They proposed to eliminate the statistics held by the "least promising" leaves. The least promising nodes are defined to be the ones with the lowest values of $p_L e_L$, where p_L is the probability that examples will reach a particular leaf L, and e_L is the observed error rate at L. To reduce memory usage even more, they also suggested the removal of statistics of the poorest performing attributes in each leaf.

The Hoeffding (and McDiarmid) bound holds true for any type of distribution. A disadvantage of being so general is that it is more conservative than a distribution-dependent bound and, thus, requires more examples than really necessary. Jin and Agrawal [83] pro-

Algorithm 2.1 The Hoeffding Tree algorithm [49]

```
Input: $\mathcal{S}$: data stream of examples
       $\mathcal{X}$: set of discrete attributes
       $G(\cdot)$: split evaluation function
       $\delta$: split confidence
Output: $H_T$: Hoeffding Tree

1: $H_T \leftarrow$ a tree with a single leaf $L_1$ (the root);
2: $\mathcal{X}_1 \leftarrow \mathcal{X} \cup \{X_0\}$; // where $X_0$ is the tree root
3: $G_1(X_0) \leftarrow G$ obtained by predicting the most frequent class in $\mathcal{S}$;
4: for all classes $K_k \in \{K_1, \ldots, K_c\}$ do
5:   for all values $x_{ij}$ of each attribute $X_i \in \mathcal{X}$ do
6:     $n_{ijk}(l_1) \leftarrow 0$; // $j$-th discrete value of $i$-th attribute
7:   end for
8: end for
9: for all examples $\mathbf{x}^t \in \mathcal{S}$ do
10:  Sort $\{\mathbf{x}^t, y^t\}$ into a leaf $L$ using $H_T$;
11:  for all attribute values $x_{ij} \in \mathbf{x}$ such that $X_i \in \mathcal{X}_L$ do
12:    $n_{ijk}(L) \leftarrow n_{ijk}(L) + 1$;
13:  end for
14:  label $L$ with the majority class among the examples seen so far at $L$;
15:  if the examples seen so far at $L$ are not all of the same class then
16:    compute $G_L(X_i)$ for each $X_i \in \mathcal{X}_L - \{X_0\}$ using the counts $n_{ijk}(L)$;
17:    $X_a \leftarrow$ the attribute with the highest $G_L$;
18:    $X_b \leftarrow$ the attribute with the second-highest $G_L$;
19:    compute Hoeffding bound $\epsilon$ using (2.1);
20:    if $G_L(X_a) - G_L(X_b) > \epsilon$ and $X_a \neq X_0$ then
21:      replace $L$ by an internal node that splits on $X_a$;
22:      for all branches of the split do
23:        add a new leaf $L_m$;
24:        $\mathcal{X}_m \leftarrow \mathcal{X} - \{X_a\}$;
25:        $G_m(X_0) \leftarrow$ the $G$ obtained by predicting the most frequent class at $L_m$;
26:        for all classes $K_k \in \{K_1, \ldots, K_c\}$ do
27:          for all values $x_{ij}$ of each attribute $X_i \in \mathcal{X}_m - \{X_0\}$ do
28:            $n_{ijk}(L_m) \leftarrow 0$;
29:          end for
30:        end for
31:      end for
32:    end if
33:  end if
34: end for
```

posed the use of an alternative bound which requires less examples for each split node. They also proposed a way of handling numerical attributes, which VFDT originally does not support, called Numerical Interleave Pruning (NIP). NIP creates structures similar to histograms for numerical attributes with many distinct values. With time, the number of bins in such histograms can be pruned allowing memory usage to remain constant.

A different approach to dealing with numerical attributes was proposed by Gama et al. [67]. The authors use binary trees as a way of dynamically discretizing numerical values. The same paper also investigates the use of an additional classifier at leaf nodes, namely Naive Bayes. Other performance enhancements to Hoeffding Trees include the use of *grace periods*, *tie-breaking*, and *skewed split prevention* [80, 20, 67]. Because it is costly to compute the split evaluation function for each example, it is sensible to wait for more examples before re-evaluating a split node. After each example, leaf statistics are still updated, but the split nodes are evaluated after a larger number of examples dictated by a grace period parameter. Tie breaking involves adding a new parameter ψ, which is used in an additional condition $\epsilon < \psi$ in line 20 of the presented VFDT pseudo-code. This condition prevents the algorithm form waiting too long before choosing one of two, almost identically useful split attributes. To prevent skewed splits, Gama proposed a rule stating that "a split is only allowed if there are at least two branches where more than p_{min} of the total proportion of examples are estimated to follow the branch" [20].

The originally proposed VFDT algorithm was designed for stationary data streams and provided no forgetting mechanism. The problem of classifying time changing data streams with Hoeffding Trees was first tackled by Hulten et al. [80]. The authors proposed a new algorithm called CVFDT, which used a fixed-size window to determine which nodes are aging and may need updating. For fragments of the Hoeffding Tree that become old and inaccurate, alternative subtrees are grown that later replace the outdated nodes. It is worth noting, that the whole process does not require model retraining. Outdated examples are forgotten by updating node statistics and necessary model changes are performed on subtrees rather than the whole classifier.

Other approaches to adding a forgetting mechanism to the Hoeffding Tree include using the Exponentially Weighted Moving Average (EWMA) [143] or ADWIN as drift detectors [11]. The latter, gives performance guarantees concerning the obtained error rate and both mentioned methods are more accurate and less memory consuming than CVFDT. However, the EWMA and ADWIN tree extensions are more expensive in terms of average time required to process a single example.

Hoeffding Trees represent state-of-the-art in single classifiers for large-scale data streams. They fulfill all the requirements of an online learner presented in Section 2.1 and provide good interpretability. Their performance has been compared several times with traditional decision trees, Naive Bayes, kNN, and batch ensemble methods [49, 19, 67, 80, 83] and they proved to be much faster and less memory consuming while handling extremely large datasets.

It is worth mentioning that apart from algorithms based on the Hoeffding bound, a different approach to creating a decision tree for data streams was also put froward. Cohen et al. proposed to repeatedly apply the Information Network (IFN) algorithm [103] to a sliding window of examples and dynamically adjust the window size depending on the rate of concept changes. The resulting algorithm, called OLIN [38], produces a new decision tree with each fresh window of examples and does not use any error bounds to determine split nodes. The characteristic feature of trees produced by OLIN is that they aim at minimizing the total number of predicting attributes.

2.3.2 Windowing Techniques

Many popular approaches to dealing with time changing data involve the use of *sliding windows* [172, 88, 160, 13, 14, 67, 80]. Sliding windows provide a way of limiting the amount of examples introduced to the learner, thus eliminating those examples that come from an old concept. An important property of sliding windows is that they can transform traditional batch algorithms, known from static environments, into classifiers for concept-drifting data streams. The basic procedure of using sliding windows is presented in Algorithm 2.2.

Algorithm 2.2 Basic windowing algorithm

```
Input: $\mathcal{S}$: data stream of examples
       $W$: window of examples
Output: $C$: a classifier built on examples in window $W$

1: initialize window $W$;
2: for all examples $\mathbf{x}^t \in \mathcal{S}$ do
3:     $W \leftarrow W \cup \{\mathbf{x}^t\}$;
4:     if necessary remove outdated examples from $W$;
5:     rebuild/update $C$ using $W$;
6: end for
```

The basic windowing algorithm is straightforward. Each example updates the window and later the classifier is updated by that window. The key part of this algorithm lies in the definition of the window, i.e., in the way it models the forgetting process. In the simplest approach, sliding windows are of fixed size and include only the most recent examples from the data stream. With each new data point the oldest example that does not fit in the window is discarded. Unfortunately, when using windows of fixed size the user is caught in a trade-off. A classifier built on a small window of examples will react quickly to changes, but may lose on accuracy in periods of stability. On the other hand, a classifier built on a large window of examples will fail to adapt to rapidly changing concepts. For this reason, more dynamic ways of modeling the forgetting process, such as heuristic adjusting of the window size [166, 14, 88, 160, 172] or decay functions [37, 62], have been proposed. In the following paragraphs, we present algorithms that use dynamic sliding windows.

Weighted windows

A simple way of making the forgetting process more dynamic is providing the window with a decay function that assigns a weight to each example. Older examples receive smaller weights and are treated as less important by the base classifier. Cohen and Strauss [37] analyzed the use of different decay functions for calculating data stream aggregates. Equations 2.2 through 2.4 present the proposed functions.

$$w_{exp}(\tau) = e^{-\lambda\tau}, \quad \lambda > 0 \tag{2.2}$$

$$w_{poly}(\tau) = \frac{1}{\tau^{\alpha}}, \quad \alpha > 0 \tag{2.3}$$

$$w_{chord}(\tau) = 1 - \frac{\tau}{|W|} \tag{2.4}$$

Equation 2.2 presents an exponential decay function, 2.3 a polynomial function, and 2.4 a chordal function. For each of the functions, τ represents the age of an example. A new example will have $\tau = 0$ whilst the last example that fits chronologically in a window will have $\tau = |W| - 1$. The use of decay functions allows to gradually weight the examples offering a compromise between large and small fixed windows. Algorithm 2.3 presents the process of obtaining a window with decaying weights.

Algorithm 2.3 Weighted windows

Input: $\mathcal{S}$: a data stream of examples
d: window size
$w(\cdot)$: weight function
Output: W: a window of examples

```
1: for all examples $\mathbf{x}^t \in \mathcal{S}$ do
2:   if $|W| = d$ then
3:     remove the oldest example from $W$;
4:   end if
5:   $W \leftarrow W \cup \{\mathbf{x}^t\}$;
6:   for all examples $\mathbf{x}_i \in W$ do
7:     calculate example's weight $w(\mathbf{x}_i)$;
8:   end for
9: end for
```

FISH

Žliobaitė proposed a family of algorithms called FISH [173, 172, 175], that use time and space similarities between examples as a way of dynamically creating a window. To explain this approach, let us consider an illustrative example presented in Figure 2.5. A binary classification problem is represented by black and white dots. The data generating sources change with time, gradually rotating the optimal classification hyperplane. For a given fixed area in the attribute space, depicted with a red circle, the true class of examples changes as the optimal boundary rotates.

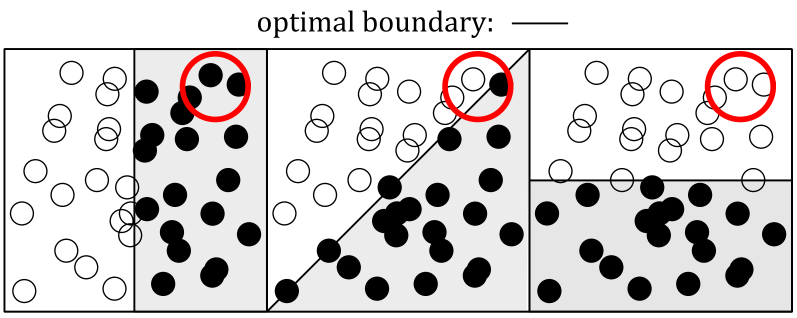


Figure 2.5: Rotating hyperplane example [172]: (left) initial source $S1$, (center) source $S2$ after 45° rotation, (right) source $S3$ after 90° rotation. Black and white dots represent the two classes.

As the example shows, concept definitions can change with time, but within a certain time frame examples from a given concept should be close to each other in the attribute space. Following this observation, the author proposed a method for selecting training examples based on a distance measure D_{ij} defined as follows:

$$D_{ij} = a_1 d_{ij}^{(s)} + a_2 d_{ij}^{(t)} \tag{2.5}$$

where $d^{(s)}$ indicates distance in attribute space, $d^{(t)}$ indicates distance in time, and $a1$, $a2$ are weight coefficients. In order to manage the balance between time and space distances, $d^{(s)}$ and $d^{(t)}$ need to be normalized. For two examples $\mathbf{x}^i$, $\mathbf{x}^j$ described by p attributes, the author proposes to use the Eucledian distance ($d_{ij}^{(s)} = \sqrt{\sum_{k=1}^{p} |\mathbf{x}_k^i - \mathbf{x}_k^j|^2}$) as distance in space and the difference between example numbers ($d_{ij}^{(t)} = |i - j|$) as distance in time. It is worth noticing that if $a_2 = 0$ then the measure D_{ij} turns into instance selection, and if $a_1 = 0$ then we have a simple window with linearly time decaying weights. Having discussed the proposed distance measure, we present the pseudo-code of FISH3 in Algorithm 2.4.

Algorithm 2.4 FISH3 [175]

Input: $\mathcal{S}$: a data stream of examples
k: neighborhood size
windowStep: optimal window size search step
proportionStep: optimal time/space proportion search step
b: backward search size
Output: W: window of selected examples

```
1:  for all examples $\mathbf{x}^t \in \mathcal{S}$ do
2:    for $\alpha \leftarrow 0$;  $\alpha \leq 1$;  $\alpha \leftarrow \alpha + proportionStep$ do
3:      $a_1 \leftarrow \alpha$;
4:      $a_2 \leftarrow 1 - \alpha$;
5:      for all remembered historical examples $\mathbf{x}^j \in \{\mathbf{x}^{t-b}, ..., \mathbf{x}^{t-1}\}$ do
6:        calculate distance $D_{tj}$ using (2.5);
7:      end for
8:      sort the distances from minimum to maximum;
9:      for $s = k$;  $s \leq b$;  $s \leftarrow s + windowStep$ do
10:       select $s$ instances having the smallest distance $D$;
11:       using leave-one-out cross-validation build a classifier $C_s$ using instances indexed
          $\{t_1, \ldots, t_s\}$ and test it on the $k$ nearest neighbors indexed $\{t_1, \ldots, t_k\}$;
12:       record the acquired testing error $e_s$;
13:     end for
14:   end for
15:   find the minimum error classifier $C_m$, where $m = \arg\min_{m=k,...,b}(e_m)$;
16:   $W \leftarrow$ instances indexed $t_1, ..., t_m$;
17: end for
```

For each new example $\mathbf{x}^t$ in the data stream, FISH3 evaluates different time/space proportions and window sizes. For each tested time/space proportion it calculates the similarities between target observation $\mathbf{x}^t$ and the past b instances, and sorts those distances from minimum to maximum. Next, the closest k instances to the target observation are selected as a validation set. This set is used to evaluate different window sizes from

k to b. FISH3 selects the training size m, which has given the best accuracy on the validation set. For window testing, leave-one-out cross-validation is employed to reduce the risk of overfitting. Without cross-validation the training set of size k is likely to give the best accuracy, because in that case the training set is equal to the validation set. The algorithm returns a window of m selected training examples that can be used to learn any base classifier.

FISH3 allows to dynamically establish the size of the training window and the proportion between time and space weights. The algorithm's previous version FISH2 [172] takes the time/space proportion as a parameter, while the first algorithm from the family, FISH1 [173], uses a fixed window of the nearest instances. To implement a variable sample size, FISH2 and FISH3 incorporated principles from two windowing methods proposed by Klinkenberg et al. [88] and Tsymbal et al. [160].

FISH3 is an algorithm that needs to iterate through many window sizes and time/space proportions, each time performing leave-one-out cross-validation. This is a costly process and may be unfeasible for rapid data streams. That is why the definition of parameters k, b, $proportionStep$, $windowStep$ is very important. It is also worth noticing that although the algorithm can be used with any base classifier, due to the way it selects instances, it will work best with nearest neighbor type methods.

ADWIN

Bifet [13, 14] proposed an adapting sliding window algorithm called ADWIN suitable for data streams with sudden drift. The algorithm keeps a sliding window W containing the most recently read examples. The main idea of ADWIN is as follows: whenever two "large enough" subwindows of W exhibit "distinct enough" averages, one can conclude that the corresponding expected values are different, and the older portion of the window is dropped. This involves answering a statistical hypothesis: "Has the average μ remained constant in W with confidence δ"? The pseudo-code of ADWIN is listed in Algorithm 2.5.

Algorithm 2.5 Adaptive windowing algorithm [14]

```
Input: $\mathcal{S}$: data stream of examples
       $\delta$: confidence level
Output: $W$: window of examples

1: initialize window $W$;
2: for all $\mathbf{x}^t \in \mathcal{S}$ do
3:     $W \leftarrow W \cup \{\mathbf{x}^t\}$;
4:     repeat
5:         drop the oldest element from $W$;
6:     until $|\mu_{\hat{W}_0} - \mu_{\hat{W}_1}| < \epsilon_{cut}$ holds for every split of $W$ into $W = W_0 \cup W_1$;
7: end for
```

The key part of the algorithm lies in the definition of ϵ_{cut} and the test used to determine if a window should be split. The authors state that different statistical tests can be used for this purpose, but propose only one specific implementation which is based on the Hoeffding bound. Let d denote the size of W, and d_0 and d_1 the sizes of W_0 and W_1

consequently, so that $d = d_0 + d_1$. Let $\mu_{\hat{W_0}}$ and $\mu_{\hat{W_1}}$ be the averages of the values in W_0 and W_1, and μ_{W_0} and μ_{W_1} their expected values. The value of ϵ_{cut} is proposed as follows:

$$\epsilon_{cut} = \sqrt{\frac{1}{2m} \cdot \frac{4}{\delta'}}, \tag{2.6}$$

where

$$m = \frac{1}{1/d_0 + 1/d_1}, \text{ and } \delta' = \frac{\delta}{d}.$$

The statistical test in line 6 of the pseudo-code checks if the observed average in both subwindows differs by more than threshold ϵ_{cut}. The threshold value is based on the Hoeffding bound, thus gives formal guarantees of the base classifier's performance. The phrase "holds for every split of W into $W = W_0 \cup W_1$" means that we need to check all pairs of subwindows W_0 and W_1 created by splitting W in two. The verification of all subwindows is very costly due to the number of possible split points. That is why, the authors proposed an improvement to the algorithm that allows to find a good cut point quickly [14].

The originally proposed ADWIN algorithm is also a lossless learner, thus, the window size W can grow infinitely if no drift occurs. This can be easily improved by adding a parameter that limits the window's maximal size. It is also worth noticing that in its original form, ADWIN works only for 1-dimensional data, e.g., the running error. For this method to be used for n-dimensional raw data, for example to track attribute value changes, a separate window should be maintained for each dimension.

2.3.3 Drift Detectors

Apart from sliding windows, another group of techniques that allow to transform almost any learner into an adaptive data stream classifier are *drift detectors* [70]. Their task is to detect concept drift and alarm a base learner that its model should be rebuilt or updated. This is usually done by a statistical test that verifies if the running error or class distribution remain constant over time.

For streams of numbers the first proposed tests where the Cumulative Sum (CUSUM) [133] and Geometric Moving Average (GMA) [143]. The CUSUM test raises an alarm if the mean of the input data is significantly different from zero, while GMA checks if the weighted average of examples in a window is higher than a given threshold. For populations more complex than numeric sequences statistical tests like the Kolmogorov-Smirnov test [117] have been proposed. Below, we discuss five recently proposed change detectors designed for drifting data streams.

DDM

Gama et al. [65] based their Drift Detection Method (DDM) on the fact, that in each iteration an online classifier predicts the decision class of an example. That prediction can be either *true* or $false$, thus, for a set of examples the error is a random variable from Bernoulli trials. Following this observation, the authors model the number of classification errors with a Binomial distribution.

Let us denote p_i as the probability of a $false$ prediction and s_i as its standard deviation calculated using Equation 2.7.

$$s_i = \sqrt{\frac{p_i(1-p_i)}{i}} \tag{2.7}$$

For a sufficiently large number of examples ($n > 30$), the Binomial distribution can be approximated by a Gaussian distribution with the same mean and variance. Using ideas from statistical process control [62], the authors propose to track the error rate of a classifier by updating two registers: p_{min} and s_{min}. These values are used to calculate a *warning level* condition presented in Equation 2.8 and an *alarm level* condition presented in Equation 2.9.

Algorithm 2.6 The Drift Detection Method [65]

```
Input: $\mathcal{S}$: a data stream of examples
       $C$: classifier
Output: $drift \in \{TRUE, FALSE\}$

1:  Initialize($i, p_i, s_i, ps_{min}, p_{min}, s_{min}$);
2:  $newConcept \leftarrow FALSE$;
3:  $W' \leftarrow \emptyset$;
4:  for all examples $\mathbf{x}^t \in \mathcal{S}$ do
5:    $drift \leftarrow FALSE$;
6:    if prediction $C(\mathbf{x}^t)$ is incorrect then
7:      $p_i \leftarrow p_i + (1.0 - p_i)/i$;
8:    else
9:      $p_i \leftarrow p_i - (p_i)/i$;
10:   end if
11:   compute $s_i$ using (2.7);
12:   $i \leftarrow i + 1$;
13:   if $i > 30$ (approximated normal distribution) then
14:     if $p_i + s_i \leq ps_{min}$ then
15:       $p_{min} \leftarrow p_i$;
16:       $s_{min} \leftarrow s_i$;
17:       $ps_{min} \leftarrow p_i + s_i$;
18:     end if
19:     if drift detected (2.9) then
20:       $drift \leftarrow TRUE$;
21:       Initialize($i, p_i, s_i, ps_{min}, p_{min}, s_{min}$);
22:       $W' \leftarrow \emptyset$;
23:     else if warning level reached (2.8) then
24:       if $newConcept = TRUE$ then
25:         $W' \leftarrow \emptyset$;
26:         $newConcept \leftarrow FALSE$
27:       end if
28:       $W' \leftarrow W' \cup \{x_i\}$
29:     else
30:       $newConcept \leftarrow TRUE$;
31:     end if
32:   end if
33: end for
```

Algorithm 2.7 DDM: Initialize()

Input: $i, p_i, s_i, ps_{min}, p_{min}, s_{min}$: window statistics
Output: initialized values of input parameters

```
1: $i \leftarrow 1$;
2: $p_i \leftarrow 1$;
3: $s_i \leftarrow 0$;
4: $ps_{min} \leftarrow \infty$;
5: $p_{min} \leftarrow \infty$;
6: $s_{min} \leftarrow \infty$;
```

Each time a warning level is reached, examples are remembered in a separate window. If afterwards the error rate falls below the warning threshold, the warning is treated as a false alarm and the separate window is dropped. However, if the alarm level is reached, the previously taught base learner is dropped and a new one is created, but only from the examples stored in the separate "warning" window.

$$p_i + s_i \geq p_{min} + \alpha \cdot s_{min} \tag{2.8}$$

$$p_i + s_i \geq p_{min} + \beta \cdot s_{min} \tag{2.9}$$

The values α and β in the above conditions decide about the confidence levels at which the warning and alarm signals are triggered. The authors proposed $\alpha = 2$ and $\beta = 3$, giving approximately 95% confidence of warning and 99% confidence of drift. Algorithm 2.6 shows the steps of the Drift Detection Method.

DDM works best on data streams with sudden drift, as gradually changing concepts can pass without triggering the alarm level. In its original form, when no changes are detected DDM works like a lossless learner, constantly training the base classifier. Therefore, for streams were drifts are rare or uncertain, it is advised to limit the maximum number of examples presented to a classifier or periodically rebuild classifiers even when no drifts are detected.

EDDM

Baena-García et al. [6] proposed a modification of DDM called EDDM. The authors use the same warning-alarm mechanism that was proposed by Gama, but instead of using the classifier's error rate, they propose the *distance error rate*. They denote p'_i as the average distance between two consecutive misclassifications and s'_i as its standard deviation. Using these values the new warning and alarm conditions are given by Equations 2.10 and 2.11.

$$\frac{p'_i + 2 \cdot s'_i}{p'_{max} + 2 \cdot s'_{max}} < \alpha \tag{2.10}$$

$$\frac{p'_i + 3 \cdot s'_i}{p'_{max} + 3 \cdot s'_{max}} < \beta \tag{2.11}$$

EDDM is designed to work better than DDM for slow gradual drift, but is more sensitive to noise. Another drawback of this method is that it searches for concept drift when a minimum of 30 errors have occurred (as opposed to a minimum of 30 examples). This is necessary to approximate the Binomial distribution by a Normal distribution, but can cause a significant delay in change detection.

ADWIN

Some of the algorithms that are used as sliding windows can also be used as drift detectors. For example, the ADWIN algorithm [13, 14] described in Section 2.3.2, apart from being a method for dynamically selecting a window of examples, can be used to predict concept changes. In ADWIN, a window of examples W grows until there has been a change in the average value inside the window. Therefore, when the algorithm succeeds at finding two distinct subwindows, the split point can be considered as a concept change.

ECDD

Recently, another popular windowing technique has been employed to detect drifts in an algorithm called ECDD (EWMA for Concept Drift Detection) [144]. The Exponentially Weighted Moving Average (EWMA) [143], apart from being a popular forgetting mechanism [20, 21], can be used to detect an increase in the mean of a sequence of random variables. ECDD calculates two estimators of the probability of misclassifying an example, one using EWMA and giving more weight to recent examples, and another with similar emphasis on recent and old data. Both estimations are compared and, when the difference between the two estimates exceeds a predefined threshold, a drift is signaled. Similarly to DDM, the authors also describe a warning level triggered when estimations are approaching drift level.

Page-Hinkley Test

The Page-Hinkley test (PH) [133] is a variant of CUSUM. Although originally used as a sequential analysis technique for change detection in signal processing [129], it has recently been proposed as a drift detector by Gama et al. [69]. It allows efficient detection of changes in the normal behavior of a process established by a model. The test variable m^t used in PH is defined as the cumulative difference between the observed values e^i and their mean up until the current moment in time:

$$m^t = \sum_{i=1}^{t} (e^i - \hat{e}^t - \delta) \tag{2.12}$$

where $\hat{e}^t = 1/t \sum_{i=1}^{t} e^i$ and δ corresponds to the magnitude of changes that are allowed [69]. For drift detection, Gama et al. propose to treat the classifier's error rate as the observed value. Additionally, the minimal m^t is defined as $M^t = min(m^i; i = 1 \dots t)$. The PH test calculates the difference between M^t and m^t ($PH^t = m^t - M^t$), and if this difference is higher than a user specified threshold (λ), a change is flagged.

As an alternative to tracking the classifier's mean error over time, Gama et al. propose to perform the PH test with the ratio between two error estimates: a long term error estimate (using a large window of examples or weak fading factor) and a short term error estimate (using a short window or strong fading factor). If the short term error estimator is significantly greater than the long term error estimator, a drift is signaled. For sliding windows, the procedure involves two windows of different sizes $W_1 = \{e^i | i \in (t - d_1, t]\}$ and $W_2 = \{e^i | i \in (t - d_2, t]\}$ (with $d_2 < d_1$) and computing the moving average w.r.t. both: $M_{W_1}(t) = 1/d_1 \sum_{i=t-d_1}^{t} e^i$ and $M_{W_2}(t) = 1/d_1 \sum_{i=t-d_2}^{t} e^i$. The PH test monitors the ratio $R(t)$ between both moving averages, as presented in Algorithm 2.8. It is worth noting that this test is designed to detect mainly abrupt drifts, with higher λ entailing fewer false alarms, but possibly causing to miss some changes [70].

Algorithm 2.8 Page-Hinkley Test [69]

```
Input: $\delta$: admissible change
       $\lambda$: drift threshold
       $e(\cdot)$: loss function
Output: $drift \in \{TRUE, FALSE\}$

1:  $SR(0) \leftarrow 0; m^t(0) \leftarrow 0; M^t \leftarrow 1;$
2:  for all examples $\mathbf{x}^t \in \mathcal{S}$ do
3:      update moving averages $M_{W_1}$ and $M_{W_2}$ using error $e(\mathbf{x}^t)$;
4:      $R(t) = \frac{M_{W_2}}{M_{W_1}};$
5:      $SR(t) \leftarrow SR(t-1) + R(t);$
6:      $m^T(t) \leftarrow m^T(t-1) + R(t) - \frac{SR(t)}{t} - \delta;$
7:      $M^T \leftarrow min(M^T, m^T(t));$
8:      if $m^T(t) - M^T \geq \lambda$ then
9:          $drift \leftarrow TRUE;$
10:     else
11:         $drift \leftarrow FALSE;$
12:     end if
13: end for
```

2.3.4 Ensemble Approaches

Classifier ensembles (also called *multiple classifiers* or *committees*) are a common way of improving classification accuracy, which has been studied in static data mining for many years [100]. Due to their modularity, they also provide a natural way of adapting to change by modifying ensemble members. In this section, we discuss the use of ensemble classifiers to mine evolving data streams.

Ensemble algorithms are sets of single classifiers, called *component classifiers* or *ensemble members*, whose decisions are aggregated into a final prediction. The combined decision of many single classifiers is usually more accurate than that given by a single component. However, in order to obtain this improvement in predictive performance, components have to be diversified. Components can differ due to the examples they have been trained on, the attributes they use, or the classification algorithm they use. The most common approach to aggregating decisions of component classifiers is by majority voting. A commonly used generic ensemble training scheme, based on selecting training examples for components, is presented in Algorithm 2.9.

Algorithm 2.9 Generic ensemble training algorithm [87]

Input: $\mathcal{S}$: set of examples
k: number of classifiers in ensemble
Output: $\mathcal{E}$: an ensemble of classifiers

```
1: $\mathcal{E} \leftarrow k$ classifiers;
2: for all classifiers $C_i$ in ensemble $\mathcal{E}$ do
3:     select a subset of examples $D_i$ from $\mathcal{S}$;
4:     build $C_i$ using $D_i$;
5: end for
```

In a data stream setting, the entire set of examples cannot be analyzed as a whole, and the presented ensemble training scheme needs to be modified. Furthermore, when concept drift is anticipated, a forgetting mechanism needs to be added to the process. There are many ways of achieving these goals, such as, adding a new classifier after each block of examples, changing component weights (vote importance), or replacing the weakest ensemble member. A comprehensive taxonomy of ensemble strategies for changing environments has been presented in [99, 62], however, for the purpose of this thesis, we will distinguish two general groups of ensembles:

- *online ensembles* which learn incrementally after processing single examples,
- *block-based ensembles* which process blocks of data.

In the following paragraphs, we describe ensemble methods most related to this thesis. We will start by analyzing four block-based algorithms: the Streaming Ensemble Algorithm, the Accuracy Weighted Ensemble, Learn^{++}.NSE, and the Adaptive Classifier Ensemble. After block-based ensembles, we will discuss online ensembles focusing on Online Bagging, the Dynamic Weighted Majority, and Hoeffding Option Trees. During the description of each algorithm, we will shortly mention classifiers, which are either wrappers or modifications of the aforementioned ensemble methods.

Streaming Ensemble Algorithm

Street and Kim [155] proposed an ensemble method called Streaming Ensemble Algorithm (SEA) that changes its structure to react to drifts. The authors propose a heuristic replacement strategy of the weakest component classifier based on two factors: accuracy and diversity. Accuracy is considered important because, as the authors suggest, an ensemble should correctly classify the most recent examples to adapt to drift. On the other hand, diversity is often desirable in such ensemble methods like bagging or boosting in static environments. The pseudo-code of SEA is listed in Algorithm 2.10.

The algorithm processes the stream in blocks of examples. Each data block is used to train a new (*candidate*) classifier, which is later compared with existing ensemble members. If any ensemble member is weaker than the candidate classifier, it is dropped and the candidate classifier takes its place. To select the classifiers, Street and Kim propose to use

Algorithm 2.10 The Streaming Ensemble Algorithm [155]

Input: $\mathcal{S}$: stream of examples
d: size of each data block B_j
$Q(\cdot)$: classifier quality measure
k: number of classifiers in an ensemble
Output: $\mathcal{E}$: ensemble of k classifiers

```
1: for all blocks $B_j \in \mathcal{S}$ do
2:   build classifier $C_j$ using $B_j$;
3:   evaluate classifier $C_{j-1}$ on $B_j$;
4:   evaluate all classifiers $C_i \in \mathcal{E}$ on $B_j$;
5:   if $|\mathcal{E}| < k$ then
6:     $\mathcal{E} \leftarrow \mathcal{E} \cup \{C_{j-1}\}$;
7:   else if $\exists i : C_i \in \mathcal{E}$ and $Q(C_{j-1}) > Q(C_i)$ then
8:     replace member $C_i$ with $C_{j-1}$;
9:   end if
10: end for
```

classification accuracy obtained on the most recent data block. They assign quality scores to components according to their accuracy and diversity as follows:

- if both the evaluated component C_i and the ensemble $\mathcal{E}$ are correct, then the score of C_i is increased by $1 - |P_1 - P_2|$;
- if C_i is correct and $\mathcal{E}$ incorrect, the score of C_i is increased by $1 - |P_1 - P_{correct}|$;
- if C_i is incorrect, the score of C_i is decreased by $1 - |P_{correct} - P_{C_i}|$,

where $P1$ and $P2$ denote the percentages of votes gained by two highest-voted decision classes by $\mathcal{E}$, $P_{correct}$ the percentage of votes of the correct decision class, and P_{C_i} the percentage of votes gained by the class predicted by the candidate classifier. Based on the obtained quality scores, the weakest component classifier is removed from the ensemble.

In the paper introducing SEA, the authors used C4.5 decision trees as base classifiers and compared the ensemble's accuracy with a pruned and unpruned decision tree. SEA performed almost as well as a pruned tree on static datasets and much better on datasets with concept drift. The authors also performed a series of experiments varying the number of operational parameters. They showed that SEA performed best when no more than 25 components were used, base classifiers were unpruned, and simple majority voting was used to combine member decisions.

Accuracy Weighted Ensemble

A similar way of restructuring an ensemble was proposed by Wang et al. [163]. In their algorithm, called Accuracy Weighted Ensemble (AWE), the authors also propose to train a new classifier C' on each incoming block of examples. Furthermore, the most recent block is also used to evaluate all of the existing ensemble members to select the best component classifiers. The difference between SEA and AWE lies in the way components are evaluated, selected, and combined. Wang et al. stated and proved that for an ensemble

$\mathcal{E}_k$ built from the k most recent data blocks and a single classifier G_k built using all of the examples in these k blocks, the following theorem stands:

Theorem 2.1. *$\mathcal{E}_k$ produces a smaller classification error than G_k, if classifiers in $\mathcal{E}_k$ are weighted by their expected classification accuracy on the test data.*

To explain the intuition behind this theorem, the authors discuss an illustrative example that presents the importance of accurate component weighting.

Let us assume a stream of 2-dimensional data partitioned into sequential blocks according to their arrival time. Let B_j be the data that came in between time t_j and t_{j+1}. Figure 2.6 shows the data distribution and optimum decision boundary during each time interval. Because the distributions in the data blocks differ, there is a problem in determining the blocks that should remain influential to accurately classify incoming data.

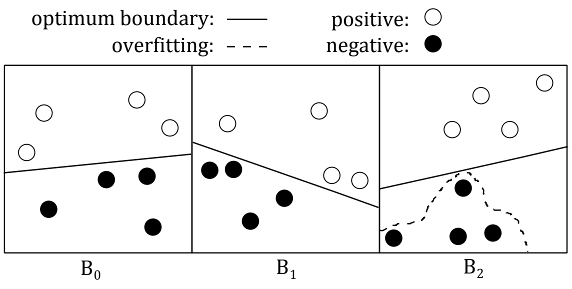


Figure 2.6: Example data distribution

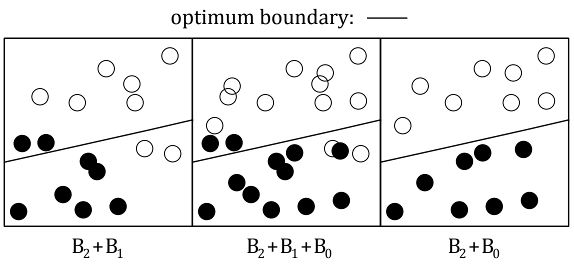


Figure 2.7: Training set selection

Figure 2.7 shows the possible block sets that can be selected for ensemble training. The best set consists of blocks B_0 and B_2, which have similar class distributions. This shows that decisions based on example class distributions are bound to be better than those based solely on data arrival time. Historical data whose class distributions are similar to that of current data can reduce the variance of the current model and increase classification accuracy.

It is worth noticing that the similarity of distributions in blocks largely depends on the size of the blocks. Bigger blocks will build more accurate classifiers, but can contain more than one change. On the other hand, smaller blocks are better at separating changes, but usually lead to poorer classifiers. The definition of block sizes is crucial to the performance of this algorithm.

According to Theorem 2.1, to properly weight the members of an ensemble we need to know the actual function being learned, which is unavailable. That is why the authors propose to derive weights by estimating the error rate on the most recent data block B_j, as shown in Equations 2.13–2.15.

$$MSE_{ij} = \frac{1}{|B_j|} \sum_{\{\mathbf{x},y\} \in B_j} (1 - f_{iy}(\mathbf{x}))^2 \tag{2.13}$$

$$MSE_r = \sum_y p(y)(1 - p(y))^2 \tag{2.14}$$

$$w_{ij} = MSE_r - MSE_{ij}, \tag{2.15}$$

Function $f_{iy}(\mathbf{x})$ denotes the probability given by classifier C_i that $\mathbf{x}$ is an instance of class y. This is an interesting feature, as most weighting functions use only the component's prediction rather than the probability of all possible classes. It is also important to note, that for the candidate classifier C' the error rate is calculated using cross-validation on the current block to avoid overfitting. Other ensemble members are evaluated on all the examples of the current block. The value of MSE_r is the mean square error of a randomly predicting classifier and is used to exclude components that do not contain any useful knowledge about the data. The pseudo-code of AWE is listed in Algorithm 2.11.

Algorithm 2.11 Accuracy Weighted Ensemble [163]

```
Input: $\mathcal{S}$: stream of examples
       $d$: size of each data block $B_j$
       $k$: number of classifiers in the ensemble
       $\mathcal{C}$: set of previously trained classifiers (optional)
Output: $\mathcal{E}$: set of $k$ classifiers with updated weights

 1: for all blocks $B_j \in \mathcal{S}$ do
 2:   train classifier $C'$ on $B_j$;
 3:   compute error rate of $C'$ via cross-validation on $B_j$;
 4:   derive weight $w'$ for $C'$ using (2.15);
 5:   for all classifiers $C_i \in \mathcal{C}$ do
 6:     apply $C_i$ on $B_j$ to derive $MSE_{ij}$;
 7:     compute $w_{ij}$ based on (2.15);
 8:   end for
 9:   $\mathcal{E} \leftarrow k$ of the top weighted classifiers in $\mathcal{C} \cup \{C'\}$
10:   $\mathcal{C} \leftarrow \mathcal{C} \cup \{C'\}$
11: end for
```

For the first k data blocks the algorithm outputs a set of all available classifiers, but when processing further blocks it selects only the k best components to form the ensemble. Wang et al. discussed that for a large data stream it is impossible to remember all the classifiers created during the ensemble's lifetime and the selection cannot be performed on an unbounded set of classifiers. That is why, for dynamic data streams it is necessary to introduce an additional parameter that limits the number of classifiers available for selection.

The AWE algorithm works well on data streams with recurring concepts, however, as with SEA it is crucial to properly define the data block size as it determines the ensemble's flexibility. It is also worth noticing, that AWE will improve its performance gradually over time and is best suited for large data streams.

Adaptive Classifier Ensemble

Both, SEA and AWE, are highly dependent on the data block size. Larger blocks promote more accurate ensemble members, but extend the period in which these algorithms cannot respond to sudden concept drifts. Small blocks, however, worsen the performance of each component classifier and in result the entire ensemble. To overcome these drawbacks, Nishida et al. proposed an online learning system, called Adaptive Classifier Ensemble (ACE) [130], which uses an online learner alongside an ensemble of batch classifiers. The basic concept of ACE is shown in Figure 2.8 and the full pseudo-code is presented in Algorithm 2.12.

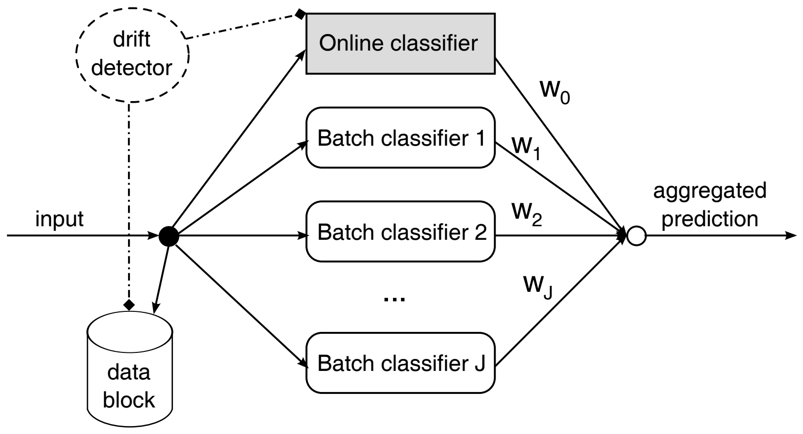


Figure 2.8: Basic architecture of ACE

ACE consists of one online classifier, many batch classifiers, and a drift detector. With each incoming example, the online classifier is incrementally trained and the block is extended. Furthermore, the drift detector checks the average accuracy of each batch classifier (calculated on the current block), and if the best performing component falls outside a $100(1-\alpha)\%$ confidence interval, where α is a user-specified parameter, a change is signaled. When concept drift is detected or the number of buffered examples exceeds the block size, a new batch classifier is created and the online learner is reset. ACE forms its final hypothesis by aggregating the predictions of the online learner and batch learners using a weighted majority vote, with each classifier C_i receiving at time t a weight w_i^t defined as:

$$w_i^t = \left(\frac{1}{1 - A_i^t} \right)^{\mu} \tag{2.16}$$

where A_i^t is the accuracy of the i-th classifier calculated on the current block of examples and μ is a normalization factor.

One of the characteristic features of ACE is that it does not limit the number of ensemble members. This property allows the algorithm to accurately react to recurring concepts by reusing previously trained classifiers. Furthermore, the addition of an online learner and drift detector offer quicker reactions to sudden concept changes compared

Algorithm 2.12 Adaptive Classifier Ensemble [130]

```
Input: $\mathcal{S}$: stream of examples
       $a$: short term memory size
       $d$: data block size ($> a$)
       $\alpha$: confidence level
       $\mu$: ensemble adjustment factor
Output: $\mathcal{E}$: set of classifiers with updated weights

 1: for all examples $\mathbf{x}^t \in \mathcal{S}$ do
 2:   $B \leftarrow B \cup \{\mathbf{x}^t\}$;
 3:   for all classifiers $C_i \in \mathcal{E} \cup \{C'\}$ do
 4:     compute average accuracy of $C_i$ on $B$ limited to $a$ examples;
 5:     compute confidence intervals for $C_i$;
 6:     compute weight $w_i^t$ based on (2.16);
 7:   end for
 8:   update (incrementally train) classifier $C'$ using $\mathbf{x}^t$;
 9:   if $|B| \geq d$ or $driftDetected$ (best batch classifier exceeds confidence interval) then
10:     create batch learner $C_{new}$ from examples in $B$;
11:     if $C_{new}$ is more accurate than a random classifier on $B$ then
12:       $\mathcal{E} \leftarrow \mathcal{E} \cup \{C_{new}\}$
13:     end if
14:     reset online classifier $C'$;
15:     $B \leftarrow \emptyset$;
16:   end if
17: end for
```

to most block-based ensembles. Finally, it is worth noticing that ACE detects changes based on the performance of a single batch classifier instead of the entire ensemble and uses a custom drift detection method. It is worth noting that the problem of quicker drift detections in block-based ensembles was also tackled in the Batch Weighted Ensemble algorithm, proposed by Magdalena Deckert [39].

Learn^{++}.NSE

Learn^{++}.NSE [52] is a block-based ensemble inspired by human learning theory. Several components of this algorithm relate to *schema theory* [59], which is a psychological model that describes the process of human knowledge acquisition and memory organization. For example, Learn^{++}.NSE retains, constructs, or temporarily discards knowledge depending on the nature of changes in the stream. Furthermore, the algorithm weights examples depending on their difficulty measured in terms of ensemble performance. The pseudo-code for Learn^{++}.NSE is given in Algorithm 2.13.

The training of Learn^{++}.NSE starts with evaluating the ensemble on a block of new examples. Next, the algorithm identifies which examples are correctly predicted by the existing ensemble and gives lower weights to these examples, as they are less difficult. Using the block of examples with updated weights, a new classifier is created and added to the ensemble. Later, all of the ensemble members are evaluated and their weights are calculated as log-normalized multiplicative inverses of their weighted errors. The weighting

Algorithm 2.13 Learn^{++}.NSE [52]

Input: $\mathcal{S}$: stream of examples
a: sigmoid slope
b: sigmoid infliction point
Output: $\mathcal{E}$: set of classifiers with updated weights

```
1:  for all blocks $B_j \in \mathcal{S}$ do
2:    compute error $E_j$ of $\mathcal{E}$ on $B_j$;
3:    for all examples $\mathbf{x}^t \in B_j$ do
4:      if $\mathbf{x}^t$ was predicted correctly by $\mathcal{E}$ then
5:        $w_j^t \leftarrow E_j$;
6:      end if
7:    end for
8:    create classifier $C_j$ using $B_j$;
9:    $\mathcal{E} \leftarrow \mathcal{E} \cup C_j$;
10:   for all classifiers $C_i \in \mathcal{E}$ do
11:     compute error $\epsilon_{ij}$ of $C_i$ on $B_j$ (with updated example weights);
12:     if $\epsilon_{ij} > 1/2$ and $i = j$ then
13:       create new classifier $C_j$;
14:     else if $\epsilon_{ij} > 1/2$ then
15:       $\epsilon_{ij} \leftarrow 1/2$;
16:     end if
17:     $\beta_{ij} \leftarrow \epsilon_{ij}/(1-\epsilon_{ij})$;
18:     $\omega_{ij} \leftarrow 1/(1+e^{-a(j-i-b)})$;
19:     $\omega_{ij} \leftarrow \omega_{ij}/\sum_{k=0}^{j-i}\omega_{i,j-k}$;
20:     $\bar{\beta}_{ij} \leftarrow \sum_{k=0}^{j-i}\omega_{i,j-k}\beta_{i,j-k}$;
21:     $W_{ij} \leftarrow log(1/\bar{\beta}_{ij})$;
22:   end for
23: end for
```

function is designed to temporarily block votes from component classifiers that do not match the current environment.

Learn^{++}.NSE has an interesting psychological inspiration, which is apparent mainly in the training and weighting of component classifiers. First of all, the algorithm tries to detect new concepts by analyzing the performance of the ensemble on new data, which can be connected to the process of problematizing known from tutoring theory [52]. Furthermore, the algorithm weights ensemble members using a sigmoid-based function, which takes into account recent performance of a given component classifier and draws inspirations from the human process of memory tuning. Finally, it is also worth mentioning that Learn^{++}.NSE does not permanently discard any component classifiers and is therefore particularly suitable for streams with recurring drifts.

Online bagging and boosting

Within the group of online ensembles, generalizations of static solutions are often considered. For example, Oza and Russel introduced an online version of bagging [132], where component classifiers are incremental learners that combine their decisions using a simple unweighted majority vote. The sampling, crucial to batch bagging, is performed incrementally by presenting each example to a component k times, where k is defined by the Poisson(1) distribution. The authors have proven that, under certain conditions, the classification function returned by online bagging converges to that returned by batch bagging as the number of base models and the number of training examples tends to infinity [131]. The pseudo-code of online bagging is given in Algorithm 2.14.

Algorithm 2.14 Online Bagging [131]

```
Input: $\mathcal{S}$: stream of examples
       $k$: number of classifiers in the ensemble
Output: $\mathcal{E}$: ensemble of classifiers

1: $\mathcal{E} \leftarrow k$ incremental classifiers;
2: for all example $\mathbf{x}^t \in \mathcal{S}$ do
3:   for all classifiers $C_i \in \mathcal{E}$ do
4:     set $l$ according to $Poisson(1)$;
5:     for 1 to $l$ do
6:       update $C_i$ using $\mathbf{x}^t$;
7:     end for
8:   end for
9: end for
```

Recently, Bifet et al. introduced two modifications of Oza's algorithm called Adaptive-Size Hoeffding Trees (ASHT) [17] and Leveraging Bagging [16], which aim at adding more randomization to the input and output of the base classifiers. More precisely, ASHT synchronously grows trees of different sizes, whereas Leveraging Bagging increases resampling from $Poisson(1)$ to $Poisson(\lambda)$ (where λ is a user-defined parameter) and uses output detection codes [16].

Furthermore, Oza and Russell also generalized the weighting procedure of boosting [132]. The authors noted that the AdaBoost algorithm actually divides the total example weight into two halves, i.e., half of the weight is assigned to the correctly classified examples, while the other half goes to the misclassified examples. To perform instance weighting online, the Poisson distribution is used once again, but with the parameter changing according to the boosting weight of the example as it is passed through each model in sequence. More recently, Pelossof et al. presented Online Coordinate Boosting [137], an online boosting algorithm, which yields a closer approximation to the AdaBoost algorithm. In this algorithm, the weight update procedure is derived by minimizing AdaBoost's loss when viewed in an incremental form.

Dynamic Weighted Majority

Another popular online ensemble is an algorithm called Dynamic Weighted Majority (DWM) [91]. In DWM a set of incremental classifiers is weighted according to their accuracy after each incoming example. With each mistake made by one of DWM's component classifiers, its weight is decreased by a user-specified factor β. Furthermore, after a period of predictions p the entire ensemble is evaluated and, if needed, a new classifier is added to the ensemble. If learned on a large stream, DWM can potentially generate extensive numbers of components, therefore, ensemble pruning is often considered as an extension [62]. The pseudo-code of the Dynamic Weighted Majority is given in Algorithm 2.14.

Algorithm 2.15 Dynamic Weighted Majority [91]

Input: $\mathcal{S}$: stream of examples
β: factor for decreasing weights ($0 \leq \beta < 1$)
θ: threshold for deleting component classifiers
p: period between expert removal, creation and weight update
Output: $\mathcal{E}$: ensemble of weighted classifiers

```
1:  $k \leftarrow 1$;
2:  $\mathcal{E} \leftarrow$ new incremental classifier $C_1$;
3:  $w_1 \leftarrow 1$;
4:  for all example $\mathbf{x}^t \in \mathcal{S}$ do
5:    for all classifiers $C_i \in \mathcal{E}$ do
6:      $\lambda \leftarrow$ class predicted by $C_i$ for example $\mathbf{x}^t$;
7:      if $\lambda \neq y^t$ and $t \mod p = 0$ then
8:        $w_i \leftarrow \beta w_i$;
9:      end if
10:     $\sigma_\lambda \leftarrow \sigma_\lambda + w_i$;
11:   end for
12:   $\Lambda \leftarrow \arg\max_i \sigma_i$; // global prediction
13:   if $t \mod p = 0$ then
14:     normalize weights $w_i, i = 1, 2, \ldots, k$;
15:     remove classifiers with weight lower than $\theta$;
16:     if $\Lambda \neq y^t$ then
17:       $k \leftarrow k + 1$;
18:       $\mathcal{E} \leftarrow$ new incremental classifier $C_k$;
19:       $w_k \leftarrow 1$;
20:     end if
21:   end if
22:   for all classifiers $C_i \in \mathcal{E}$ do
23:     train classifier $C_i$ with example $\mathbf{x}^t$;
24:   end for
25: end for
```

DWM is an extension of the Weighted Majority Algorithm [112] known from the field of online learning. However, DWM takes into account the dynamic nature of data streams and is designed to track concept drift. In contrast to its predecessor, DWM adds and removes component classifiers in response to global performance of the entire ensemble and local performances of individual components.

Hoeffding option trees

Kirkby [138, 87] proposed an Option Tree similar to that of Kohavi and Kunz [90] that allows each training example to update a set of option nodes rather than just a single leaf. Option nodes work like standard decision tree nodes with the difference that they can split the decision paths into several subtrees. Making a decision with an option tree involves combining the predictions of all applicable leaves into a single result.

Hoeffding Option Trees (HOT) provide a compact structure that works like a set of weighted classifiers, and just like regular Hoeffding Trees, they are built in an incremental fashion. The detailed pseudo-code for the Hoeffding Option Tree is listed below in Algorithm 2.16.

Algorithm 2.16 Hoeffding option tree [87]

```
Input: $\mathcal{S}$: a data stream of examples
       $G_l(\cdot)$: a split evaluation function
       $\delta$: split confidence
       $\delta'$: confidence for additional splits
       $\psi$: tie threshold
       $k$: maximum number of options that should be reachable by any single example
Output: $H_{OT}$: a Hoeffding option tree

1:  $H_{OT} \leftarrow$ a tree with a single leaf $l_1$ (the root);
2:  for all examples $\mathbf{x}^t \in \mathcal{S}$ do
3:    Sort $\mathbf{x}^t$ into a leaf/option $L$ using $H_{OT}$;
4:    for all option nodes $l$ of the set $L$ do
5:      update sufficient statistics in $l$;
6:      $n_l \leftarrow$ the number of examples seen at $l$;
7:      if $n_l \mod n_{min} = 0$ and examples seen at $l$ not all of same class then
8:        if $l$ has no children then
9:          compute $G_l()$ for each attribute of $\mathbf{x}^t$;
10:         $X_a \leftarrow$ the attribute with the highest $G_l$;
11:         $X_b \leftarrow$ the attribute with the second-highest $G_l$;
12:         compute Hoeffding bound $\epsilon$ using (2.1);
13:         if $X_a \neq X_\emptyset$ and $(G_l(X_a) - G_l(X_b) > \epsilon$ or $\epsilon < \psi)$ then
14:           add a node below $l$ that splits on $X_a$;
15:           for all branches of the split do
16:             add a new option leaf with initialized sufficient statistics;
17:           end for
18:         end if
19:       else
20:         if $optionCount_l < k$ then
21:           compute $G_l()$ for existing splits and (non-used) attributes;
22:           $s \leftarrow$ existing child split with highest $G_l$
23:           $X_s \leftarrow$ (non-used) attribute with highest $G_l$
24:           compute Hoeffding bound (2.1) using $\delta'$ instead of $\delta$;
25:           if $G_l(X_s) - G_l(s) > \epsilon$ then
26:             add an additional child option to $l$ that splits on $X_s$;
27:             for all branches of the split do
28:               add a new option leaf with initialized sufficient statistics;
29:             end for
30:           end if
31:         else
32:           remove attribute statistics stored at $l$;
33:         end if
34:       end if
35:     end if
36:   end for
37: end for
```

The algorithm works similarly to the Hoeffding Tree listed in Algorithm 2.1. The differences show from line 20 where a new option is created. Like in most ensemble approaches, there is a limit to the number of ensemble members denoted as k. If this limit has not been exceeded for a given leaf, a new option path can be trained. Option creation is similar to adding a leaf to a Hoeffding Tree with one minor difference concerning the split condition. For the initial split (line 13) the decision process searches for the best attribute overall, but for subsequent splits (line 25) the search is for attributes that are superior to existing splits. It is very unlikely that any other attribute could compete so well with the best attribute already chosen that it could beat it by the same initial margin (the Hoeffding bound practically insures that). For this reason, a new parameter δ', which should be much "looser", is used for the secondary split.

Other ensemble approaches

In the preceding sections, we describe ensemble methods most related to this thesis. However, there are many other approaches worth mentioning.

A set of several online bagging ensembles is used in the DDD algorithm proposed by Minku et al. [126]. DDD is a meta-classifier based on the analysis of levels of ensemble diversities. When a drift occurs, DDD tries to substitute pairs of differently diversified ensembles depending on the type of drift that occurred.

The WWH algorithm, from Yoshida et al. [171], builds component classifiers on overlapping windows to select the best learning examples and aggregates component predictions similarly to the Weighted Majority Algorithm. Therefore, WWH can be seen as a combination of an instance selection windowing technique with an adaptive ensemble.

Finally, several algorithms are dedicated to creating ensembles strictly from decision trees. For example, the Ultra Fast Forest of Trees (UFFT) [64] is an incremental algorithm that learns a forest of binary trees from data streams. Moreover, algorithms such as Multiple Semi-Random decision Trees (MSRT) [109] and Streaming Random Forests [1] create ensembles from decision trees built on randomized sets of attributes.

Although other adaptive data stream ensemble algorithms may exist, to the best of our knowledge, they are not directly related to the topic of this thesis.

Chapter 3

The Accuracy Updated Ensemble

Block-based ensembles are among the most popular classifiers for concept-drifting data streams. The popularity of ensembles, in general, is by virtue of their modularity, which allows them to achieve higher accuracy than single classifiers and facilitates their parallelization. Block-based ensembles additionally have the advantage that they can utilize static learners known from traditional data mining. Moreover, in many practical scenarios examples have to be processed in blocks, as incoming examples cannot be labeled online.

In this chapter, we propose a new block-based ensemble, called the Accuracy Updated Ensemble, which aims at reacting equally well to several types of drift. The proposed algorithm is experimentally compared with 11 state-of-the-art stream methods, including single classifiers, block-based and online ensembles, and hybrid approaches in different drift scenarios. The evaluation study shows that the Accuracy Updated Ensemble outperforms competitive algorithms in terms of average classification accuracy, while proving to be less memory consuming than other ensemble approaches.

3.1 Classification in Block-based Environments

As it was discussed in Section 2.3.4, block-based ensembles sequentially generate new component classifiers from consecutive fixed-size blocks of learning examples. Because several examples are available during the creation of component classifiers, standard algorithms known from static learning, such as, e.g., C4.5 decision trees, can be used during training. This feature differentiates block-based ensembles from online ensembles, which can only use incremental classifiers as ensemble members.

After a new classifier is learned from the most recent block of examples it is added to the ensemble, usually replacing the weakest of existing component classifiers. To decide which ensemble members should be kept and which should be discarded, each component is evaluated, commonly according to its predictive performance. This is possible due to the fact that each block of labeled examples constitutes a test set for existing ensemble members. Additionally, most block-based ensembles use the results of this evaluation not only to determine current ensemble members, but also to assign weights to each component. These weights are later used while making a combined prediction based on all components [155, 163, 130, 91, 52]. As mentioned in Section 2.3.4, the SEA algorithm [155] was the first of such adaptive ensembles and was soon followed by the Accuracy Weighted Ensemble [163], which is presently the most representative method of this type.

However, depending on the occurrence of concept drifts within a block of examples, the mentioned block-based ensembles may not react sufficiently to changes. In particular, for sudden drifts they may react too slowly, as classifiers generated from outdated blocks still remain valid components even though they have inaccurate weights. This situation is connected with the problem of proper tuning of the data block size. Using small-sized blocks can partly help in reacting to sudden changes, but doing so will damage the performance of the ensemble in periods of stability and increase computational costs. An unsatisfactory reaction of block-based ensembles to other types of drifts has already been noticed in several studies [130, 30, 52].

On the other hand, most online ensembles can react faster to sudden drifts and all their components evolve over time. This is in direct contrast to block-based ensembles, which often use static component classifiers that, once trained, never change. However, online ensembles do not take advantage of periodical component evaluations and do not weight or replace ensemble members [132, 16, 87, 137]. As a result, on data streams with gradual or incremental changes, online ensembles are often less accurate than block-based approaches. Furthermore, online ensembles are often characterized by higher computational costs than block-based methods. Such observations suggest that it could be profitable to combine characteristic features from both groups of approaches in order to sufficiently adapt to various types of changes.

Following these critical motivations, we propose a new hybrid algorithm, called Accuracy Updated Ensemble (AUE), which should react to various types of concept drift much better than related block-based ensembles. Our goal is to retain the simple schema of substituting component classifiers and weighting their predictions, characteristic for block-based algorithms, while adding elements known from online methods.

The main novel contribution of AUE is the introduction of incremental updating of component classifiers, which improves the ensemble's reactions to concept drifts, as well as reduces the impact of block sizes on the predictive performance of the ensemble. Incremental updates allow all ensemble members to adapt to the most recent concept simultaneously and, therefore, change the basic idea behind existing block-based algorithms. Additionally, we have performed an analysis of several component weighting procedures, which has led to interesting findings concerning incremental training of adaptive ensembles. In the following section, we discuss the details of the proposed algorithm and highlight its characteristic features.

3.2 The Accuracy Updated Ensemble

The Accuracy Updated Ensemble maintains a weighted pool of component classifiers and predicts the class of incoming examples by aggregating the predictions of components using a weighted voting rule. After each block of examples a new classifier is created. This new *candidate* classifier substitutes the poorest performing ensemble member. The performance of each component classifier is evaluated by estimating its expected prediction error on examples from the most recent data block. After substituting the poorest performing component, the remaining ensemble members are updated, i.e., incrementally trained, and their weights are adjusted according to their predictive performance. In this thesis, we will analyze the use of Hoeffding Trees as component classifiers, since they performed favorably compared to Naive Bayes in preliminary experiments. Nevertheless, the presented algorithm can be considered a general method, and in principle, one could use other online algorithms as base learners.

Let $\mathcal{S}$ be a data stream partitioned into evenly sized blocks $B_1, B_2, \ldots, B_j$, each containing d examples. For every incoming block B_j, the weight w_{ij} of each component classifier $C_i \in \mathcal{E}$ $(i = 1, 2, \ldots, k)$ is calculated by estimating the error rate on data block B_j, as presented in Equations 3.1–3.3.

$$MSE_{ij} = \frac{1}{|B_j|} \sum_{\{\mathbf{x},y\} \in B_j} (1 - f_{iy}(\mathbf{x}))^2, \tag{3.1}$$

$$MSE_r = \sum_y p(y)(1 - p(y))^2, \tag{3.2}$$

$$w_{ij} = \frac{1}{MSE_r + MSE_{ij} + \epsilon} \tag{3.3}$$

Function $f_{iy}(\mathbf{x})$ denotes the probability given by classifier C_i that $\mathbf{x}$ is an instance of class y. Following inspirations from the AWE algorithm [163], instead of single class predictions, probabilities of all classes are considered. The value of MSE_{ij} estimates the prediction error of classifier C_i on block B_j, while MSE_r is the mean square error of a randomly predicting classifier and is used as a reference point to the current class distribution (approximated on B_j). When changes occur in the class distribution, e.g., due to an ongoing concept drift, MSE_r rises causing component weights w_{ij} to have lower values and hindering the domination of single ensemble members during voting. Additionally, a very small positive value ϵ is added to the equation to ensure that w_{ij} can be calculated even when MSE_{ij} and MSE_r are equal to zero.

The weighting formula presented in Equation 3.3 aims at combining information about the classifier's accuracy and the current class distribution. Furthermore, by using a non-linear function, compared to the linear one used in AWE, we highly differentiate component classifiers. The final version of the component weighting function (as well as the candidate weighting function discussed further in this section) was chosen after performing a comparative study of several alternative approaches which will be discussed in Section 3.3.3.

Apart from assigning new weights to ensemble members, with each data block B_j a candidate classifier C' is created from examples within the most recent block of examples. As C' is trained on the most recent data, it is treated as a "perfect" classifier and assigned a weight according to Equation 3.4.

$$w_{C'} = \frac{1}{MSE_r + \epsilon} \tag{3.4}$$

Compared to the function used to weight existing ensemble members, the weight of the candidate classifier $w_{C'}$ does not take into account the prediction error of C' on B_j.

Such an approach is based on the assumption that the most recent block provides the best representation of the current and near-future data distribution. Since C' is trained on the most recent data it should be treated as the best possible classifier. Additionally, such an approach is computationally much cheaper than candidate cross-validation used in AWE.

If the ensemble $\mathcal{E}$ contains less than k components, the candidate classifier C' is simply added to the ensemble. Otherwise, out of the k existing ensemble members $C_i \in \mathcal{E}$, the poorest performing classifier, i.e., the component with the lowest weight, is substituted with the candidate classifier C'. After the substitution, remaining ensemble members are incrementally trained by presenting examples from the most recent data block B_j.

Our experiments (discussed in more detail in Section 3.3.3) have shown that for datasets which do not contain any drift, the incremental training of component classifiers in AUE can cause non-constant memory usage. For this reason, after each block the size of the ensemble is compared with a user-specified memory limit. If the memory limit is exceeded, then the least recently used leaves of component Hoeffding Trees are pruned to match the memory restriction. After pruning, the ensemble is ready to classify examples from the next incoming data block. The pseudocode of AUE is presented in Algorithm 3.1.

Algorithm 3.1 Accuracy Updated Ensemble (AUE)

```
Input: $\mathcal{S}$: stream of examples
       $d$: size of each data block $B_j$
       $k$: number of classifiers in an ensemble
       $m$: memory limit
Output: $\mathcal{E}$: ensemble of $k$ weighted incremental classifiers

1:  $\mathcal{E} \leftarrow \emptyset$;
2:  for all blocks $B_j \in \mathcal{S}$ do
3:      $C' \leftarrow$ new component classifier built on $B_j$;
4:      $w_{C'} \leftarrow \frac{1}{MSE_r + \epsilon}$ (3.4);
5:      for all classifiers $C_i \in \mathcal{E}$ do
6:          apply $C_i$ on $B_j$ to calculate $MSE_{ij}$;
7:          compute weight $w_{ij}$ based on (3.3);
8:      end for
9:      if $|\mathcal{E}| < k$ then
10:         $\mathcal{E} \leftarrow \mathcal{E} \cup \{C'\}$;
11:     else
12:         substitute classifier with lowest weight in $\mathcal{E}$ with $C'$;
13:     end if
14:     for all classifiers $C_i \in \mathcal{E} \setminus \{C'\}$ do
15:         incrementally train classifier $C_i$ with $B_j$;
16:     end for
17:     if $memory\_usage(\mathcal{E}) > m$ then
18:         prune (decrease size of) component classifiers;
19:     end if
20: end for
```

In contrast to earlier proposed block-based ensembles, such as AWE or SEA, the AUE algorithm is not designed to use static batch learners but, instead, incrementally updates component classifiers. In our opinion, this leads to better classification accuracy in the presence of slow gradual drifts and periods of stability. Additionally, since the components can be retrained, the algorithm should be less dependent on the block size and can use smaller blocks without deteriorating its accuracy [29, 30]. Compared to its predecessor AWE, AUE introduces several improvements (analyzed in detail in Section 3.3.3). In particular, AUE uses a new weighting function, does not require cross-validation of the candidate classifier, does not keep a classifier buffer, prunes its base learners, and always updates its components. That last property may be considered a costly modification since several classifiers need to be updated after each data block, however, as it will be shown in the experiments, this issue is mitigated by efficient candidate weighting.

The Accuracy Updated Ensemble also differs from other adaptive ensembles. AUE's component classifiers are weighted and can be removed, unlike in Online Bagging. Compared to VFDT-based ensembles like ASHT and HOT, we do not limit base classifier size and do not use any windows. Compared to Learn^{++}.NSE, the proposed algorithm incrementally trains existing component classifiers, retains only k of all the created components, and uses a different weighting function which ensures that components will have non-zero weights. In contrast to DWM, AUE processes the stream in blocks, weights components solely based on their prediction error, treats the candidate classifier as a perfect learner, and its weighting function does not require any user-specified parameters.

In a way, AUE can be considered as a hybrid approach — it can react to sudden drifts and it can gradually evolve with slow changing concepts. The relatively rapid adaptation after sudden drifts is achieved by repeatedly training and weighting components according to their prediction error, as well as giving the highest possible weight to the newest classifier. On the other hand, because components are updated after every block, they can react to gradual drifts. Additionally, the modular structure of AUE should protect the classifier from drastic accuracy losses in the presence of random blips, as a single "outlier" component can be outvoted when the target concept stabilizes. The performance of AUE in scenarios involving different types of drifts, as well as no drift, will be examined in the following section.

3.3 Experimental Evaluation

The proposed AUE algorithm was evaluated in several experiments to simulate scenarios involving various types of changes. We experimentally evaluated the main components of AUE, as well as compared its performance with that of related classifiers. In the following subsections, we describe all of the used datasets, discuss experimental setup, and analyze experiment results.

It is important to note that all of the experiments discussed in this thesis were performed using the MOA framework[1]. Massive Online Analysis (MOA) is a software environment for implementing algorithms and running experiments for online learning [15, 21, 20]. It is implemented in Java and contains a collection of data stream generators, online learning algorithms, and evaluation procedures. MOA is capable of reading ARFF dataset

[1]http://moa.cms.waikato.ac.nz/

files, which are commonly used in machine learning [165, 73]. It also allows to create data streams on the fly using generators, such as the Random Tree Generator [49], SEA [155], STAGGER [149], Hyperplane [163, 53, 54], Random RBF, LED [67], and Waveform [67]. Finally, the framework also allows users to add concept drift to stationary data streams. MOA uses the sigmoid function to model a drift as a weighted combination of two pure distributions that characterize the target concepts before and after the drift.

By implementing our algorithms and experiments using an open-source framework, we aim to ease the verification and dissemination of our work. Appendix A contains implementation details concerning source code, test scripts, and datasets used during experiments discussed in this thesis.

3.3.1 Datasets

Most of the common benchmarks for machine learning algorithms, e.g. gathered in the UCI Machine Learning Repository [60], contain too few examples to be considered suitable for evaluating data stream classification methods, especially in terms of processing time and memory usage. Furthermore, datasets used to test algorithms designed for static environments usually do not contain any type of concept drift. In terms of real-world data there is still a shortage of suitable and publicly available benchmark datasets. Some researchers have used private or confidential data that cannot be reproduced by others [49, 163, 53, 54]. For this reason, data stream classification algorithms are tested mostly on synthetic datasets, in which concept drifts are introduced in a controlled manner.

Following this evaluation approach, the AUE algorithm is compared with other classifiers on 11 synthetic and 4 real datasets. Artificial datasets were generated using the MOA framework and the real datasets are publicly available. A brief description of each dataset is provided below. Additionally, Table 3.1 summarizes the main dataset characteristics (detailed scripts are listed in Appendix A).

Table 3.1: Characteristic of datasets

Dataset	Instances	Attributes	Classes	Noise	Drifts	Drift type
$\texttt{Hyp}_S$	1M	10	2	5%	1	incremental
$\texttt{Hyp}_F$	1M	10	2	5%	1	incremental
$\texttt{RBF}_B$	1M	20	4	0%	2	blips
$\texttt{RBF}_{GR}$	1M	20	4	0%	4	gradual
$\texttt{RBF}_{ND}$	1M	20	2	0%	0	none
$\texttt{SEA}_S$	1M	3	4	10%	3	sudden
$\texttt{SEA}_F$	2M	3	4	10%	9	sudden
$\texttt{Tree}_S$	1M	10	4	0%	4	sudden recurring
$\texttt{Tree}_F$	100k	10	6	0%	15	sudden recurring
$\texttt{LED}_M$	1M	24	10	10%	3	mixed
$\texttt{LED}_{ND}$	10M	24	10	20%	0	none
`Elec`	45k	7	2	-	-	unknown
`CovType`	581k	53	7	-	-	unknown
`Poker`	1M	10	10	-	-	unknown
`Airlines`	539k	7	2	-	-	unknown

`Hyp`: Hyperplane is a popular dataset generator utilized in many stream classification experiments [54, 163, 172]. It assigns randomly generated examples to one of two classes divided by a hyperplane. Incremental concept drift is introduced by slightly rotating this hyperplane (the decision boundary) with each consecutive example, starting from the first instance. We set the hyperplane generator to create two datasets, each containing 1,000,000 instances described by 10 features. The first dataset ($\texttt{Hyp}_S$) contains incremental drift with the modification weight w_i changing by 0.001 with each example. The second dataset ($\texttt{Hyp}_F$) is similar to the first one but the change is more rapid with the weight changing by 0.1 with each example. Additionally, both datasets contain 5% of noise added to the concepts to randomly differentiate the instances. In this thesis, if not stated otherwise, by noise we shall refer to class noise, i.e., errors artificially introduced to class labels.

`RBF`: The RBF generator creates a user specified number of drifting centroids of radial basis functions. Each centroid is defined by a class label, position, weight, and standard deviation. With each example, the generator can force each centroid to slightly change its position in the attribute space creating a gradual drift. We use this generator to create three datasets, each conaining 1,000,000 examples described by 20 numeric attributes. The $\texttt{RBF}_{ND}$ dataset has two decision classes (two centroids) and no drift. The $\texttt{RBF}_B$ dataset contains 4 decision classes and 4 very short, sudden drifts (2 blips), which should be ignored by the tested classifier. The last dataset from this group, $\texttt{RBF}_{GR}$, is designed to contain 4 gradual recurring drifts with each concept containing 4 decision classes.

`SEA`: The SEA generator [155] is used to create two datasets with sudden concept drifts. Each concept is defined by a sum of two linear functions, which outputs a point belonging to one of four possible decision classes. Sudden drift is introduced by abruptly changing the function definitions at selected points in the stream. For our tests, we generate 1,000,000 instances with drifts occurring every 250,000 examples ($\texttt{SEA}_S$) and 2,000,000 instances with drifts occurring every 200,000 examples ($\texttt{SEA}_F$), with 10% class noise.

`Tree`: We use the Random Tree Generator to create two drifting datasets, each described by 5 nominal and 5 numerical attributes. The $\texttt{Tree}_S$ dataset contains 4 sudden recurring drifts evenly distributed over 1,000,000 examples. The $\texttt{Tree}_F$ dataset contains only 100,000 instances but is the fastest changing dataset with 15 sudden drifts. In both cases, drift is introduced by abruptly changing the concept (randomly generated tree) after a given number of examples.

`LED`: LED [28] is a popular artificial dataset, which consists of a stream of 24 binary attributes that define the digit presented on a seven-segment LED display. We use this generator to acquire two datasets. The first dataset, called $\texttt{LED}_M$, contains 1,000,000 instances with two gradually drifting concepts suddenly switching after 500,000 examples. Such a mixed type of drift is particularly difficult to learn. The second dataset ($\texttt{LED}_{ND}$) contains no drift but instead it is the largest and noisiest dataset with 10,000,000 examples and 20% of noise.

`Elec`, `CovType`, `Poker`, `Airlines`: The first of four utilized real datasets, called Electricity (`Elec`) [75], is one of the most widely used in data stream classification. It consists of energy prices from the electricity market, which were affected by market demand, supply, season, weather and time of day. `Elec` contains 45,312 instances each described by

7 features. The second real dataset, Covertype (`CovType`), contains cover type information about four wilderness areas. Examples are defined by 53 cartographic variables that describe one of seven possible forest cover types. The whole dataset consists of 581,012 instances and has been used in several papers on data stream classification [132, 19]. The third real benchmark dataset is `Poker` [19], which consists of 1,000,000 examples describing the suits and ranks of a hand of five playing cards. This gives a total of 10 predictive attributes per instance (5 cards $\times$ 2 attributes — suit and rank) with an additional class attribute that describes one of ten poker hands. Finally, `Airlines` is a real dataset containing 539,383 examples described by 7 attributes. `Airlines` encapsulates the task of predicting whether a given flight will be delayed, given the information of the scheduled departure.

The described synthetic datasets were chosen to evaluate all of the analyzed algorithms in different scenarios. As for the real datasets, we share the common assumption that we cannot unequivocally state when drifts occur or if there is any drift. The real datasets serve to compare the algorithms in a real-life scenario rather than a concrete drift situation. Furthermore, the real world datasets used in our experiments are publicly available and were used in several studies concerning data stream classification [132, 19, 177, 94, 126, 96].

3.3.2 Experimental Setup

The experiments involved AUE and 11 competitive data stream classifiers, whose selection will be discussed in more detail in Section 3.3.4. All of the tested algorithms were implemented in Java as part of the MOA framework. In particular, AUE was implemented for this study, the source codes of the Adaptive Classifier Ensemble and Learn++.NSE were provided courtesy of Dr. Kyosuke Nishida and Dr. Paulo Gonçalves respectively, while all the remaining classifiers were already a part of MOA. The experiments were conducted on a machine equipped with two 12-core AMD Opteron 6172, 2.1Ghz processors and 64 GB of RAM.

To make the comparison more meaningful, we set the same parameter values for all the algorithms. For ensemble methods we set the number of component classifiers to $k = 10$: AUE, AWE, DWM, ACE, Online Bagging, and Leveraging Bagging use ten Hoeffding Trees, while HOT has ten options. We decided to use ten component classifiers as, according to our preliminary study, using more classifiers (tested from 2 up to 40) linearly increased processing time and memory, but did not notably improve classification accuracy of the analyzed ensemble methods. The data block size used for block-based ensembles was equal $d = 500$ for all the datasets, as this size was considered the minimal suitable size for block-based ensembles such as AWE [155, 163], and lower values would drastically decrease AWE's accuracy. We set the static window size of Win to $10 \times d$ to make the number of examples seen by the windowed classifier similar to that seen by ensemble methods.

The parameters of the Hoeffding Tree used with the static window were the same as those of the option tree and the component classifiers (also Hoeffding Trees) of all the ensemble methods. More precisely, we used Hoeffding Trees enhanced with adaptive

Naive Bayes leaf predictions with a grace period $n_{min} = 100$, split confidence $\delta = 0.01$, and tie-threshold $\psi = 0.05$ [49]. Due to the fact that the only available implementation of ACE could not be fully adjusted to use classifiers from the MOA framework, we used ACE (as originally proposed by Nishida [130]) with ten C4.5 trees as batch learners and Naive Bayes as an online learner. As suggested in [52], Learn++.NSE does not use any pruning mechanism and has a sigmoid slope $a = 0.5$ and sigmoid crossing point $b = 10$.

According to the main characteristics of data streams [101, 155, 15], we evaluate the performance of algorithms with respect to time efficiency, memory usage, and classification accuracy. All the performance measures were calculated using the *block evaluation method* [29, 30], which works similarly to the test-then-train paradigm [87, 15, 11] with the difference that it uses blocks instead of single examples [29]. This method reads incoming examples without processing them, until they form a data block of size d. Each new data block is first used to evaluate the existing classifier, then it updates the classifier, and finally it is disposed. Such an approach allows to measure average block training and testing times and is less pessimistic than the test-then-train method. Moreover, since this evaluation method probes algorithm performance over time, it is suitable for evolving streams and provides a natural method of reducing result storage requirements. Where appropriate, apart from averaged metrics, we will also analyze plots of performance measures in time.

3.3.3 Component Analysis of the Proposed Algorithm

While constructing the AUE algorithm we decided to analyze the properties of AWE in search of improvements. In the following paragraphs, we summarize experiments conducted to investigate:

- the role of an additional classifier buffer,
- candidate classifier weighting schemes,
- the overall weighting function,
- refraining from component classifier updates.

One of the first analyzed properties was the use of a classifier buffer. With each block of examples, AWE creates a new classifier, but uses only k best classifiers to form an ensemble. To reduce memory usage, only n of all the constructed classifiers are stored until the next block is processed. The assumption behind such an approach is that a buffer of additional, out-of-ensemble, classifiers can prove profitable in the presence of recurring drifts. In the design phase of AUE, we decided to verify this assumption by analyzing the pros and cons of maintaining a buffer. Table 3.2 presents the results of comparing AUE with a buffer ($k = 10$ and $n = 30$) and AUE without one ($k = n = 10$).

As Table 3.2 shows, in terms of accuracy, AUE with a buffer appears to perform marginally better than AUE without one. Nevertheless, the difference in accuracy is minor or even negligible compared to the training time and memory cost. In the analyzed scenarios, using a buffer of 20 additional classifiers requires, on an average, over five times

Table 3.2: Comparison of AUE with and without a buffer in terms of average classification accuracy [%], average memory usage [MB], average block training and testing time [s]

	AUE with a buffer				AUE without a buffer			
	Acc.	Mem.	Train.	Test.	Acc.	Mem.	Train.	Test.
$\texttt{Hyp}_S$	88.59	1.86	0.23	**0.02**	**88.64**	**0.58**	**0.07**	**0.02**
$\texttt{RBF}_B$	**94.07**	2.73	0.66	**0.06**	94.06	**2.15**	**0.19**	**0.06**
$\texttt{RBF}_{GR}$	**93.37**	4.30	0.82	**0.06**	93.30	**3.91**	**0.19**	**0.06**
$\texttt{RBF}_{ND}$	**92.42**	121.51	1.33	**0.02**	92.41	**11.91**	**0.07**	**0.02**
$\texttt{SEA}_S$	89.00	1.46	0.16	**0.01**	**89.02**	**0.88**	**0.03**	**0.01**
`Elec`	**70.86**	0.39	0.05	**0.01**	70.76	**0.09**	**0.03**	**0.01**
`CovType`	**81.24**	1.56	0.78	0.12	81.19	**0.78**	**0.30**	**0.11**
`Poker`	**60.57**	0.29	0.13	0.03	59.86	**0.09**	**0.06**	**0.02**

more training time and twice as much memory compared to not using any buffer. For this reason, the buffer was excluded from AUE.

These results led to an additional conclusion. Although AUE does not require any pruning to restrict memory usage on datasets with drift [30], by testing the algorithm on a dataset without any drift ($\texttt{RBF}_{ND}$) we noticed that it requires such a mechanism in static environments. For this reason, AUE comes with a pruning mechanism that removes the least used leaves of each component Hoeffding Tree to fit a user specified memory limit.

Another costly property of AWE was the weighting of each newly created component classifier. AWE uses expensive 10-fold cross-validation (10cv) to weight the candidate classifier on the most recent block of examples [163]. We analyzed the impact of using other weighting schemes starting with other cross-validations, such as 4-fold (4cv) and 2-fold (2cv) cross-validation. We also considered the candidate's weight as a function of the remaining classifier weights. We investigated the performance of the candidate classifier with a weight equal to the maximum (Max), average (Mean), and minimum (Min) weight of the remaining classifiers, half of the sum of remaining classifier weights (Half), and half of the sum of remaining classifier weights minus a small positive value ϵ (Half$_\epsilon$).

Additionally, we experimented not only with the candidate weight but with the overall weight definition itself. We analyzed linear and non-linear functions, such as $w_L = \max(MSE_r - MSE_{ij}, 0) + \epsilon$ and $w_N = \frac{1}{MSE_r + MSE_{ij} + \epsilon}$. By using MSE_{ij} and MSE_r, we associate the component classifier's weight with its accuracy and the current class distribution. The ϵ in these functions is used to ensure that the ensemble will always be able to give a non-zero prediction.

In reference to functions w_L and w_N, we decided to treat the candidate component classifier C' as a "perfect classifier", i.e., one for which $MSE_{ij} = 0$. Such an approach is based on the implicit assumption that the most recent data block provides the best representation of the near-future data distribution. The resulting candidate weight functions for these methods are $w_{CL} = MSE_r + \epsilon$ and $w_{CN} = \frac{1}{MSE_r + \epsilon}$. It is worth noticing that the calculation of w_{CL} and w_{CN} does not require any cross-validation nor the analysis of remaining classifier weights and can be performed in constant time.

Table 3.3: Average classification accuracy of AUE with different candidate classifier weighting functions [%]

	10cv	4cv	2cv	Max	Mean	Min	Half	Half$_\epsilon$	w_{CL}	w_{CN}
`Hyp`$_S$	88.64	**88.70**	88.44	88.36	88.30	84.99	88.58	88.49	88.52	88.43
`RBF`$_B$	94.06	94.64	94.81	94.82	94.84	**95.87**	92.61	93.09	94.78	94.77
`RBF`$_{GR}$	93.30	93.98	94.10	94.21	94.23	74.73	63.38	63.64	94.15	**94.43**
`RBF`$_{ND}$	92.41	93.08	92.58	93.22	93.27	**93.40**	91.33	91.65	93.12	93.33
`SEA`$_S$	89.02	89.20	**89.21**	89.20	89.20	87.65	89.03	89.02	89.21	89.19
`Elec`	70.76	71.16	71.83	62.66	61.88	43.99	51.10	49.69	69.35	**77.32**
`CovType`	81.19	84.03	84.79	84.70	84.72	75.03	81.10	81.50	84.46	**85.20**
`Poker`	59.86	60.39	60.77	60.20	60.54	46.55	46.53	46.52	59.67	**66.23**

As Table 3.3 shows, treating the candidate classifier as a “perfect” classifier substantially increases accuracy, especially when combined with a non-linear weighting function. The most interesting results are achieved by w_{CN}, which proves best on most datasets and close to best on the remaining ones. The difference is especially visible on real datasets (`Elec`, `CovType`, `Poker`) where w_{CN} improves accuracy by a few percent compared to other solutions. What is worth noticing is that, compared to using a linear function, by using a non-linear weighting function more voting power is given to the candidate classifier. This is especially important in the presence of concept drift when the candidate is the only component of the ensemble with information about the incoming new concept. Giving such voting power to the candidate can prove inconvenient in the presence of sudden noise when the incoming concept should be treated as an outlier or when no drift occurs and the more experienced components should be more important. The obtained results seem to support this hypothesis, as for data with no drift (`RBF`$_{ND}$ and `RBF`$_B$) best results are achieved by the weighting mechanism that gives the most voting power to older components, i.e., the Min approach. Being the most accurate in different scenarios and much more computationally effective than cross-validation, we chose the w_{CN} function as the candidate classifier weighting mechanism for AUE.

In an attempt to further decrease memory usage and possibly improve classification accuracy via elements of diversification, we proposed and analyzed two alternative component updating mechanisms.

The first mechanism selects only the $b < k$ best weighted components for updating, i.e., for training with examples from the most recent block. We experimentally evaluated the effect of updating $b \in [4; 8]$ highest weighted components of an ensemble of 10 classifiers and denoted the obtained results as b_4–b_8 in Table 3.4. Since the 10th classifier is always the candidate classifier, updating 9 classifiers would actually mean updating all possible components, an option denoted in the results table as *All*.

The second mechanism involved directly using the mean square error of each component classifier. We proposed to stop updating a component classifier if the difference between the mean square error of that component obtained on the most recent data block (MSE_{ij}) and the error obtained on the previous block (MSE_{ij-1}) is greater than 0 and less than a user-defined threshold θ. Therefore, in this strategy component C_i is not updated if $0 < MSE_{ij} - MSE_{ij-1} < \theta$. We experimentally evaluated the effect of refraining from updating a component for $\theta \in [0.005; 0.05]$ and denoted the obtained results as $\theta_{0.5\%}$–$\theta_{5\%}$ in Table 3.4.

Table 3.4: Average classification accuracy of AUE with different refraining rules for component training [%]

	All	$\theta_{0.5\%}$	$\theta_{1\%}$	$\theta_{2\%}$	$\theta_{3\%}$	$\theta_{5\%}$	b_4	b_5	b_6	b_7	b_8
$\texttt{Hyp}_S$	88.44	88.51	88.58	88.43	87.74	**89.49**	87.72	87.96	88.23	88.39	88.48
$\texttt{RBF}_B$	**94.81**	94.34	93.99	92.57	88.61	78.30	93.83	94.12	94.31	94.62	94.60
$\texttt{RBF}_{GR}$	**94.10**	93.79	93.39	91.54	86.49	79.80	93.27	93.61	93.74	93.97	94.01
$\texttt{RBF}_{ND}$	92.58	92.99	92.59	91.55	89.40	77.08	92.03	92.49	92.82	92.97	**93.12**
$\texttt{SEA}_S$	**89.21**	89.16	89.12	88.61	88.00	87.19	88.67	89.01	89.04	89.07	89.19
`Elec`	**71.83**	70.61	70.52	70.60	70.81	70.93	69.29	69.29	69.29	69.29	69.29
`CovType`	**84.79**	84.57	84.19	83.36	82.57	81.17	83.29	83.60	83.99	84.21	84.57
`Poker`	**60.77**	59.67	59.68	59.69	59.79	59.85	59.82	59.82	59.82	59.82	59.82

Table 3.5: Percentage of memory used compared to updating all classifiers

	All	$\theta_{0.5\%}$	$\theta_{1\%}$	$\theta_{2\%}$	$\theta_{3\%}$	$\theta_{5\%}$	b_4	b_5	b_6	b_7	b_8
$\texttt{Hyp}_S$	100%	74%	55%	27%	20%	16%	47%	59%	84%	90%	104%
$\texttt{RBF}_B$	100%	81%	54%	32%	16%	7%	49%	60%	71%	79%	89%
$\texttt{RBF}_{GR}$	100%	80%	55%	27%	11%	4%	45%	57%	67%	77%	89%
$\texttt{RBF}_{ND}$	100%	88%	66%	31%	12%	1%	44%	59%	73%	88%	102%
$\texttt{SEA}_S$	100%	80%	60%	26%	14%	10%	43%	56%	68%	94%	96%
`Elec`	100%	100%	100%	96%	102%	99%	100%	100%	100%	100%	100%
`CovType`	100%	87%	75%	62%	53%	43%	56%	59%	69%	77%	82%
`Poker`	100%	100%	100%	100%	100%	99%	100%	100%	100%	100%	100%

The obtained results show that refraining from updating component classifiers is not the best strategy for streams with drifts. Not only does updating *All* components give best average accuracy, but one can clearly see that the less refraining was performed the better the results were. On the other hand, Table 3.5 shows that substantial savings in terms of memory can be achieved by not updating all of the component classifiers. Refraining from updating when MSE_{ij} settles at 0.5% of what it was on the previous block requires 14% less memory than always updating all components. Since results obtained by $\theta_{0.5\%}$ were very close to those achieved by *All*, this is a very interesting outcome if one needs to minimize the classifier's memory requirements.

In the tested scenarios, the proposed techniques allowed us to successfully reduce memory requirements, but did not increase accuracy. Such an outcome may suggest that the incremental creation of strong classifiers as ensemble members is of more value to the prediction of the ensemble. These results may therefore be considered concordant with the standpoint presented in [52], suggesting that drifting environments provide natural diversity and the premise of weaklearnability does not apply to them. As the main aim of the proposed algorithm is to react accurately to various types of drift, we decided to use the *All* updating option in AUE.

3.3.4 Comparative Study of Classifiers

After establishing the properties of AUE, a set of experiments was conducted to compare the newly proposed algorithm against 11 classifiers:

- the Hoeffding Option Tree (HOT),
- Adaptive Classifier Ensemble (ACE),
- a preliminary version of the AUE [30] that used a classifier buffer, weighting function $w_{ij} = 1/(MSE_{ij} + \epsilon)$, and candidate cross-validation (AUE$_{\text{pre}}$),
- the Accuracy Weighted Ensemble (AWE),
- Leveraging Bagging (Lev),
- Online Bagging (Bag),
- Dynamic Weighted Majority (DWM),
- Learn++.NSE (NSE),
- Drift Detection Method with a Hoeffding Tree (DDM),
- a single Hoeffding Tree with a static window (Win),
- and the Naive Bayes algorithm (NB).

We chose AWE and AUE$_{\text{pre}}$ as those are the classifiers we tried to improve upon. HOT and ACE were selected as they can be considered hybrid ensemble algorithms, combining elements of incremental learning. Bag, Lev, NSE, and DWM were chosen as strong representatives of online ensembles. The DDM algorithm and the windowed Hoeffding Tree were chosen as representatives of single classifiers. Additionally, the Naive Bayes algorithm is added to the comparison as a reference for using an algorithm without any drift reaction mechanism. All the studied algorithms were evaluated in terms of classification accuracy, memory usage, block training time and testing time. Average values of the analyzed measures are given in Tables 3.6–3.9.

Apart from analyzing the average performance of algorithms, we generated four graphical plots for each dataset depicting the algorithms' functioning in terms of training time, testing time, memory usage, and classification accuracy. By presenting the performance measure calculated after each data block on the y-axis and the number of processed training examples on the x-axis, one can examine the dynamics of a given classifier, in particular, its reactions to concept drift. Such graphical plots are the most common way of displaying results in data stream mining papers [6, 16, 49, 52, 65, 84, 91, 126, 130, 177]. In the following paragraphs, we will analyze the most interesting plots, which highlight characteristic features of the studied algorithms.

Figure 3.1 reports accuracies of the analyzed algorithms on the $\texttt{RBF}_{GR}$ dataset, which contains gradual recurring drifts. Looking at the plot one can see drops in accuracy around examples number 125 k, 250 k, 375 k, and 500 k. The most severely malfunctioning algorithm in the presence of gradual recurring drifts is NB, followed by Win, NSE, DWM and AWE. The subsequent drops in accuracy of the Naive Bayes algorithm suggest that

Table 3.6: Average classification accuracies in percentage [%]

	ACE	AUE$_{\text{pre}}$	AWE	AUE	HOT	DDM	Win	Lev	NB	Bag	DWM	NSE
Hyp$_S$	80.65	88.59	**90.43**	88.43	83.23	87.92	87.56	85.36	81.00	89.89	71.20	86.83
Hyp$_F$	84.56	88.58	89.21	**89.46**	83.32	86.86	86.92	87.21	78.05	89.32	76.69	85.39
RBF$_B$	87.34	94.07	78.82	94.77	93.79	88.30	73.07	**95.28**	66.97	93.08	78.11	73.02
RBF$_{GR}$	87.54	93.37	79.74	94.43	93.24	87.99	74.67	94.74	62.01	92.56	77.80	74.49
RBF$_{ND}$	84.74	92.42	72.63	**93.33**	91.20	87.62	71.12	92.24	72.00	91.37	76.06	71.07
SEA$_S$	86.39	89.00	87.73	**89.19**	87.07	88.37	86.85	87.09	86.18	88.80	78.30	86.23
SEA$_F$	86.22	88.36	86.40	**88.72**	86.25	87.80	85.55	86.68	84.98	88.37	79.33	85.07
Tree$_S$	65.77	84.35	63.74	**84.94**	69.68	80.58	50.15	81.69	47.88	81.67	51.19	49.37
Tree$_F$	45.97	**52.87**	45.35	45.32	40.34	42.74	41.54	33.42	35.02	43.40	29.30	33.90
LED$_M$	64.70	67.29	67.11	67.58	66.92	67.17	65.52	66.74	67.15	**67.62**	44.43	62.86
LED$_{ND}$	46.33	50.68	51.27	51.26	51.17	51.05	47.07	50.64	**51.27**	51.23	26.86	47.16
Elec	75.83	70.86	69.33	77.32	**78.21**	64.45	70.35	76.08	73.08	77.34	72.43	73.34
CovType	67.05	81.24	79.34	85.20	**86.48**	58.11	77.19	81.04	66.02	80.40	80.84	77.16
Poker	67.38	60.57	59.99	66.10	74.77	60.23	58.26	**82.62**	58.09	61.13	74.49	59.56
Airlines	66.75	63.92	63.31	**67.37**	66.18	65.79	64.93	63.10	66.84	66.39	61.00	63.83

Table 3.7: Average block training time in centiseconds [cs]

	ACE	AUE$_{\text{pre}}$	AWE	AUE	HOT	DDM	Win	Lev	NB	Bag	DWM	NSE
Hyp$_S$	26.83	14.82	12.15	4.41	0.69	0.33	0.17	6.04	**0.03**	3.94	7.26	116.20
Hyp$_F$	25.78	14.13	12.11	4.57	2.39	0.38	0.20	5.62	**0.03**	3.97	7.77	173.73
RBF$_B$	72.27	47.93	42.46	13.88	2.72	0.87	0.29	13.34	**0.05**	9.63	14.22	628.33
RBF$_{GR}$	72.72	54.51	42.63	14.08	3.45	0.88	0.29	13.83	**0.04**	9.96	14.57	679.69
RBF$_{ND}$	19.94	44.58	11.80	4.67	1.40	0.27	0.17	6.69	**0.03**	4.74	8.35	186.12
SEA$_S$	4.95	7.64	4.36	1.63	0.37	0.15	0.13	2.65	**0.01**	2.59	2.58	66.53
SEA$_F$	5.06	5.49	4.24	1.60	0.28	0.13	0.13	2.58	**0.01**	2.31	2.76	64.74
Tree$_S$	20.63	26.99	15.07	5.24	0.78	0.37	0.19	7.00	**0.02**	4.79	7.41	196.91
Tree$_F$	27.98	20.55	18.20	7.90	1.73	0.73	0.51	8.81	**0.02**	5.36	9.19	32.85
LED$_M$	7.75	31.47	25.48	8.81	4.44	0.62	0.26	9.49	**0.03**	7.98	8.21	402.41
LED$_{ND}$	7.73	27.72	25.30	8.99	10.11	1.28	0.22	10.14	**0.03**	11.60	7.98	932.29
Elec	4.47	5.00	5.57	3.26	1.89	1.00	1.00	3.75	**0.07**	2.70	5.90	6.74
CovType	23.35	40.87	41.29	14.83	6.64	1.23	0.38	10.21	**0.09**	9.66	18.24	425.72
Poker	2.78	9.57	6.56	4.20	1.91	0.39	0.18	3.07	**0.02**	2.83	7.69	108.00
Airlines	4.37	10.45	14.15	6.79	1.62	0.58	0.76	7.01	**0.02**	4.78	32.23	69.05

Table 3.8: Average block testing time in centiseconds [cs]

	ACE	AUE$_{\text{pre}}$	AWE	AUE	HOT	DDM	Win	Lev	NB	Bag	DWM	NSE
Hyp$_S$	0.60	1.82	1.72	1.75	0.33	0.19	0.18	2.29	**0.18**	1.97	0.80	7.48
Hyp$_F$	0.59	1.82	1.74	1.73	1.04	0.20	0.21	1.99	**0.18**	1.99	0.78	7.40
RBF$_B$	1.10	6.15	6.58	6.06	1.35	0.66	0.68	6.45	**0.66**	6.78	2.51	31.80
RBF$_{GR}$	1.11	6.44	6.53	6.29	1.69	0.70	0.71	6.81	**0.65**	7.11	2.68	20.10
RBF$_{ND}$	0.58	2.47	1.67	2.23	0.60	0.20	0.20	3.22	**0.19**	2.57	0.79	3.44
SEA$_S$	0.47	0.76	0.61	0.67	0.09	0.07	0.08	0.73	**0.07**	0.82	0.28	3.70
SEA$_F$	0.47	0.66	0.59	0.65	0.09	**0.07**	0.08	0.71	0.08	0.73	0.29	2.87
Tree$_S$	0.82	2.54	2.32	2.52	0.36	**0.22**	0.22	3.32	0.23	2.96	0.73	1.88
Tree$_F$	0.97	2.93	2.46	3.31	**0.27**	0.36	0.39	3.70	0.48	3.43	0.95	1.25
LED$_M$	2.10	4.92	4.05	3.83	**0.29**	0.48	0.41	4.58	0.39	5.62	2.02	4.49
LED$_{ND}$	2.05	4.15	4.01	3.90	**0.27**	0.97	0.42	4.95	0.40	9.28	2.03	4.08
Elec	0.62	0.85	0.40	1.18	0.73	**0.21**	0.27	1.29	0.35	1.30	0.46	2.05
CovType	0.84	6.17	6.34	6.74	4.33	**0.55**	0.71	5.22	1.26	7.45	2.29	15.36
Poker	0.57	1.79	0.37	1.92	1.31	**0.22**	0.20	1.09	0.47	1.68	0.48	2.69
Airlines	0.30	0.44	0.22	2.22	0.36	0.23	**0.21**	1.78	0.19	2.04	0.35	1.78

Table 3.9: Average classifier memory usage in megabytes [MB]

	ACE	AUE$_{\text{pre}}$	AWE	AUE	HOT	DDM	Win	Lev	NB	Bag	DWM	NSE
`Hyp`$_S$	0.14	1.97	0.28	0.63	2.94	0.24	**0.00**	4.30	0.01	0.71	0.15	16.85
`Hyp`$_F$	0.13	1.23	0.31	0.57	9.57	0.55	**0.00**	1.70	0.01	0.87	0.20	36.98
`RBF`$_B$	0.18	2.99	0.45	2.40	5.38	0.33	**0.01**	5.32	0.01	1.16	0.35	36.99
`RBF`$_{GR}$	0.19	4.67	0.43	4.65	5.94	0.30	**0.01**	6.84	0.01	1.94	0.33	36.99
`RBF`$_{ND}$	0.14	13.07	0.25	12.74	5.88	0.59	**0.00**	38.49	0.01	5.83	0.22	36.97
`SEA`$_S$	0.10	1.56	0.20	0.92	0.71	0.15	**0.00**	0.80	0.00	1.12	0.07	36.97
`SEA`$_F$	0.10	1.02	0.20	0.57	0.71	0.08	**0.00**	0.52	0.00	0.65	0.08	36.97
`Tree`$_S$	0.22	5.22	0.49	4.95	4.34	0.59	**0.00**	17.28	0.01	5.75	0.15	36.98
`Tree`$_F$	0.22	1.68	0.35	0.88	0.52	0.12	**0.01**	0.55	0.01	0.28	0.07	0.41
`LED`$_M$	0.27	0.62	0.61	0.22	2.06	0.17	**0.01**	0.62	0.03	1.50	0.04	36.99
`LED`$_{ND}$	0.27	0.62	0.61	0.22	15.74	4.73	**0.01**	0.29	0.03	6.16	0.03	180.68
`Elec`	0.10	0.39	0.27	0.46	0.75	0.03	**0.00**	0.34	0.01	0.14	0.11	0.10
`CovType`	0.20	1.57	0.68	0.85	17.17	0.08	**0.02**	0.82	0.05	0.32	0.48	12.59
`Poker`	0.14	0.33	0.27	0.20	8.05	0.14	**0.00**	1.23	0.01	0.12	0.31	25.47
`Airlines`	0.11	2.35	5.71	62.34	65.65	13.47	**0.05**	38.95	0.06	30.80	1.14	11.10

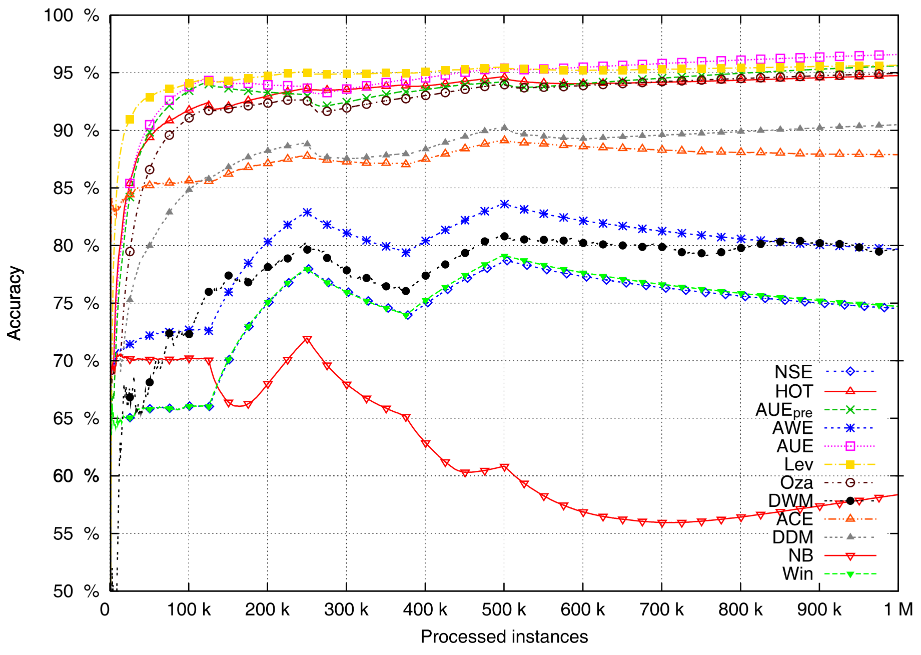


Figure 3.1: Classification accuracy on the $\texttt{RBF}_{GR}$ dataset

classifiers without any drift reaction mechanism fail to successfully learn from data with gradual recurrent drifts. On the other hand, Win, NSE, DWM and AWE appear to react too slowly. Additionally, DDM and ACE both use drift detectors which are designed to work best with sudden changes and for this reason the performance of these algorithms may not be as good as the performance of ensemble approaches. The two most accurate algorithms on this dataset are AUE and Lev. Both of these algorithms require similar training and testing time but Lev requires almost twice as much memory as AUE.

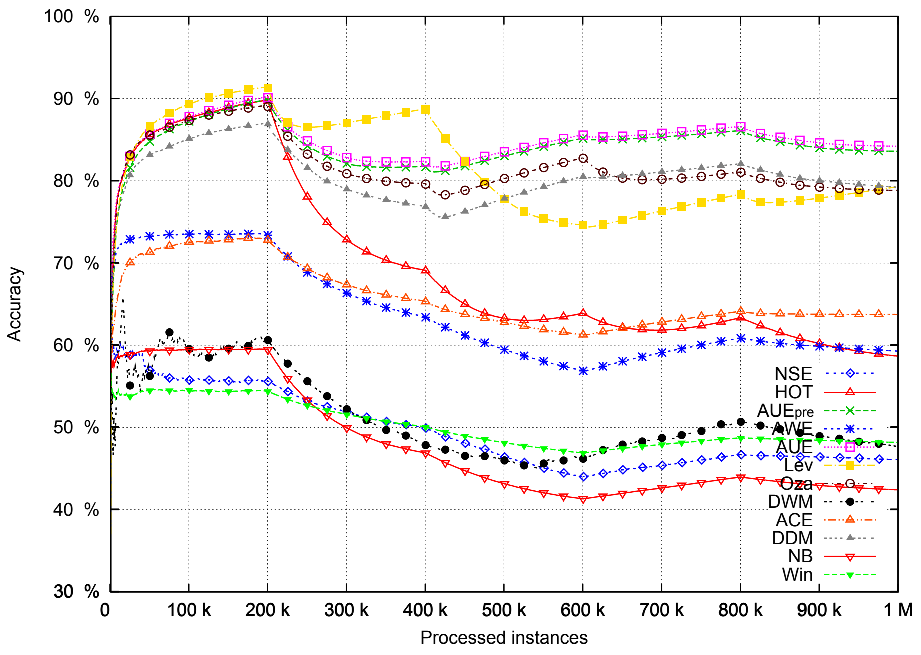


Figure 3.2: Classification accuracy on the $\texttt{Tree}_{S}$ dataset

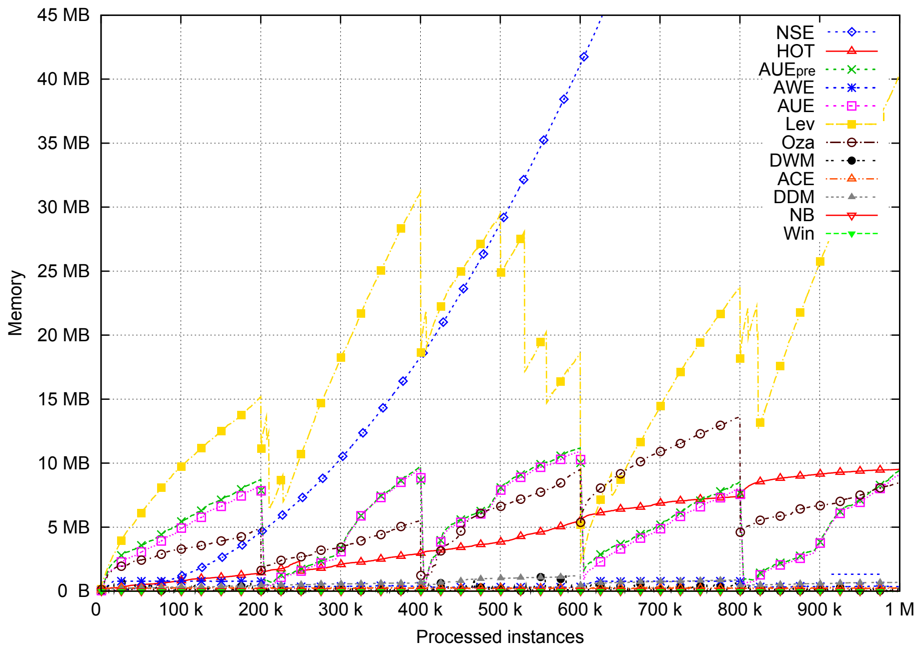


Figure 3.3: Memory usage on the $\texttt{Tree}_S$ dataset

Figures 3.2 and 3.3 show classification accuracy and memory usage on the $\texttt{Tree}_S$ dataset, which was designed to test the algorithms' reaction to recurring sudden drifts. The drifts occurring every 200 k examples are clearly visible both on the accuracy and memory plot. In the presence of sudden recurring drifts, AUE$_{\text{pre}}$ and AUE seem to perform best, with only the first drift having a major impact on their accuracy. Compared to recurring gradual drifts, the remaining algorithms are further behind in terms of accuracy. This is especially apparent with the HOT algorithm, which appears to lose accuracy with every consecutive drift. Looking at the memory plot in Figure 3.3, we can see that AUE$_{\text{pre}}$ and AUE abruptly reduce their memory usage when a drift occurs. The drop in accuracy of the previously learned components is reflected in their mean square error (MSE_{ij}), which forces one of the previously learned base classifiers to be disposed. Algorithms that seem not to have pruned their base classifiers after a sudden drift, such as HOT or Lev, lose accuracy. Similar behavior was observed in figures for the $\texttt{SEA}_S$ and $\texttt{SEA}_F$ datasets, which represent scenarios with sudden concept drifts.

It is worth noting that NSE requires much more time and memory than the remaining algorithms. This is only due to the fact that, following [52], no pruning was used to limit the number of NSE's component classifiers. On small datasets, like `Elec`, we can see that when only few components are created NSE uses less memory than other ensemble methods.

Although on the $\texttt{Tree}_S$ dataset AUE performed slightly better than AUE$_{\text{pre}}$, on the $\texttt{Tree}_F$ dataset AUE$_{\text{pre}}$ is clearly the winning algorithm. The characteristic feature of the $\texttt{Tree}_F$ dataset is the speed of recurring changes. The classifier buffer which was removed from AUE is the attribute that most probably helped AUE$_{\text{pre}}$ outclass other data stream learners on this dataset.

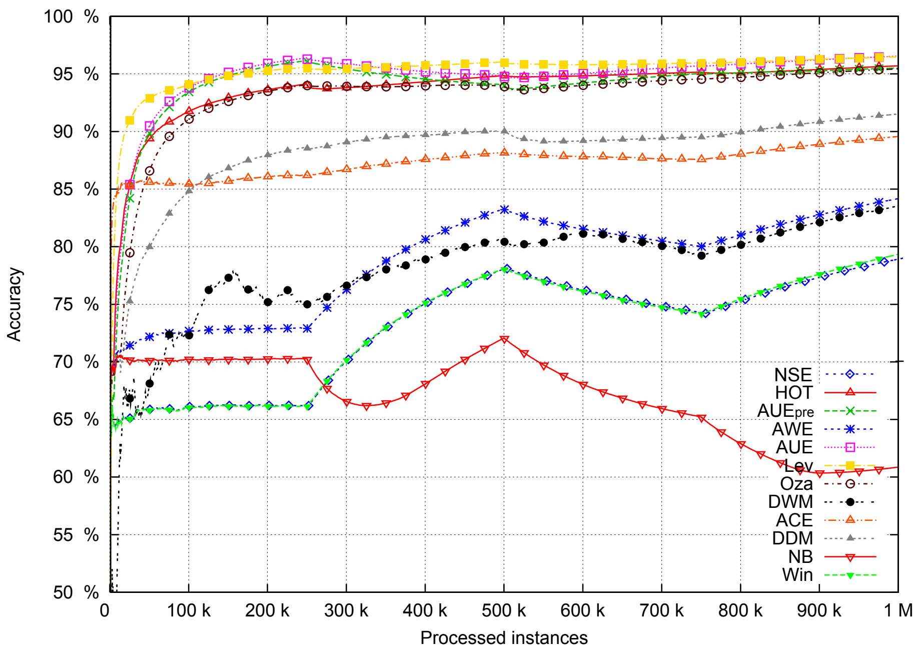


Figure 3.4: Classification accuracy on the RBF$_B$ dataset

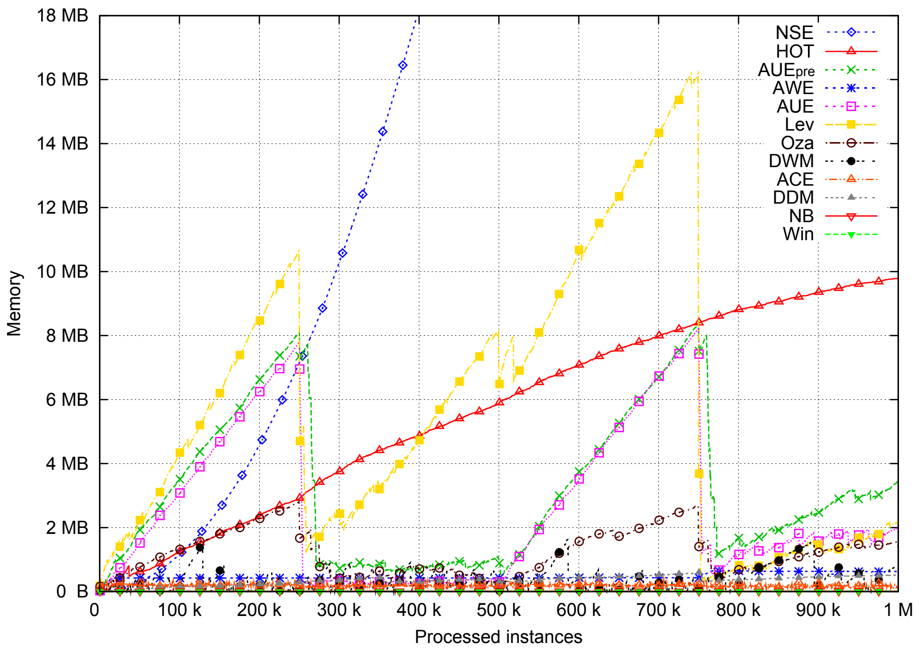


Figure 3.5: Memory usage on the RBF$_B$ dataset

A different experiment used the $\texttt{RBF}_B$ dataset, which incorporates very short, sudden concept changes (blips). Blips should be treated as outliers and should not have any long-term impact on the classifier's functioning. As Figure 3.4 shows, apart from NB, Win, NSE, DWM, and AWE, all the classifiers maintain stable accuracy throughout the entire dataset. Analyzing the memory plot in Figure 3.5, one can see that AUE_{pre} and AUE react to blips just like they reacted to sudden changes. The capability of sustaining accuracy by these two algorithms is possible due to the fact that only one ensemble component is removed per block. Even when a single component is removed in the occurrence of an outlier concept, AUE_{pre} and AUE still perform well after the blip. It is also worth noticing that the warning/alarm level mechanism used in DDM and ACE worked as expected and allowed these algorithms to stay accurate even though their classification error reached a warning level.

For datasets with incremental drifts, i.e., $\texttt{Hyp}_S$ and $\texttt{Hyp}_F$, the best performing algorithms are AWE, Bag, and AUE. AWE seems to perform particularly well on the $\texttt{Hyp}_S$ dataset. It is also worth noting that the Win classifier, usually performing rather poorly in terms of accuracy, reacts quite well to slow changes. The algorithms that perform worst are NB, ACE, DWM, and HOT. The Naive Bayes classifier has no drift reaction mechanism, the drift detector in ACE is not triggered, therefore, causing poor reaction to gradual changes, while HOT and DWM appear to not be pruning outdated data, with HOT additionally using too much memory.

When no drift is present, AUE and AUE_{pre} are the most accurate classifiers. On the $\texttt{RBF}_{ND}$ dataset, AUE has the highest accuracy followed by AUE_{pre} and Lev, while on $\texttt{LED}_{ND}$ AUE_{pre}, AUE, and NB achieve almost identical results.

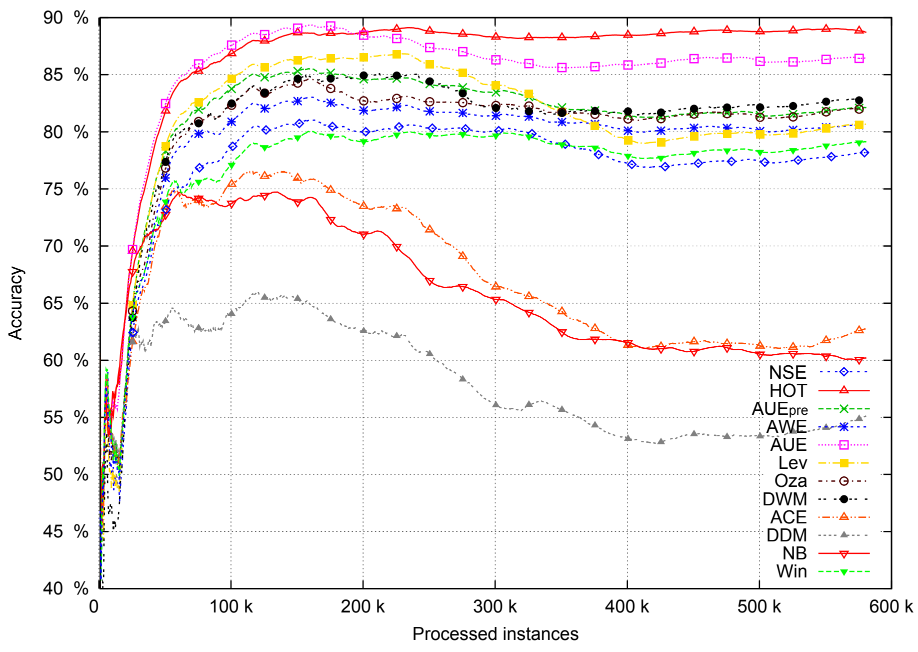


Figure 3.6: Classification accuracy on the CovType dataset

On real datasets (Elec, CovType, Poker, Airlines), HOT is the best performing learning algorithm followed by AUE. Additionally, on the Poker dataset Lev clearly outperforms all the other classifiers. It is worth mentioning that the accuracy of HOT comes at the price of high memory costs. It seems that for the analyzed real-world datasets the pruning mechanism, present in most adaptive ensembles, is not as important as the constant training of base classifiers, characteristic for HOT. The accuracy plot for the `CovType` dataset is presented in Figure 3.6. By looking at the performance of NB, DDM, and ACE, one can see that the analyzed dataset probably contains changes. The accuracy plots for `Elec`, `Poker`, and `Airlines` also contain fluctuations which were not present in the accuracy plots of artificial datasets without drift, i.e., $\texttt{RBF}_{ND}$ and $\texttt{LED}_{ND}$.

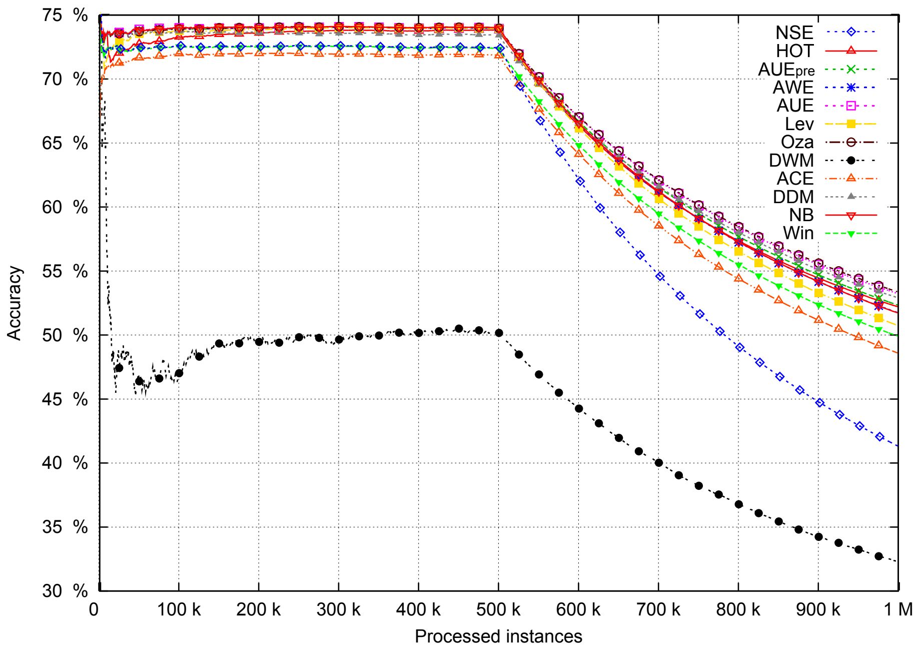


Figure 3.7: Classification accuracy on the $\texttt{LED}_M$ dataset

Finally, let us analyze the accuracy plot for the $\texttt{LED}_M$ dataset presented in Figure 3.7. In this dataset, we incorporated a complex change by joining two gradually drifting streams. After 500 k examples the target concept is suddenly switched but the gradual changes in the new concept prove to be very difficult to classify. Although Bag and AUE achieve best average accuracies, all algorithms seem to fail in reacting to the change. This shows that complex combinations of drifts can prove challenging for existing algorithms and constitute an interesting topic for further research.

3.3.5 Statistical Analysis of Results

To extend the analysis provided in Section 3.3.4, we carry out statistical tests for comparing multiple classifiers over multiple datasets [45, 82]. We use the non-parametric Friedman test combined with the Bonferroni-Dunn post-hoc test. The Friedman test is a non-parametric procedure, which ranks algorithms for each dataset separately, the best performing algorithm getting the rank of 1, the second best rank 2, etc. In case of ties average ranks are assigned. The null-hypothesis for this test is that there is no difference between the performance of all the tested algorithms. In case of rejecting this null-hypothesis, we proceed with the Bonferroni-Dunn post-hoc test. This post-hoc test verifies whether the ranked performance of AUE is statistically different from the remaining algorithms by calculating a critical difference CD that defines how much AUE must outperform another classifier.

Table 3.10: Average algorithm ranks used in the Friedman tests

	ACE	AUE$_{\text{pre}}$	AWE	AUE	HOT	DDM	Win	Lev	NB	Bag	DWM	NSE
Acc.	7.33	4.00	6.40	**2.20**	5.40	6.47	8.80	5.27	9.07	3.67	9.80	9.60
Train.	8.73	10.33	9.40	6.33	4.20	2.80	2.13	7.13	**1.00**	5.73	8.13	12.00
Test.	5.13	9.07	7.07	8.53	4.53	2.27	2.33	9.93	**2.13**	10.80	5.33	10.80
Mem.	4.00	9.33	6.27	8.07	10.13	4.93	**1.00**	9.40	2.00	7.73	4.33	10.80

The average ranks of the analyzed algorithms are presented in Table 3.10, providing a comparison in terms of accuracy, training and testing time, as well as memory usage. First, we perform the Friedman test to verify the statistical significance of the differences between accuracies of the algorithms. As the test statistic $F_F = 12.902$ and the critical value for $\alpha = 0.05$ is 1.851, the null hypothesis is rejected. Considering accuracies, AUE provides the best average achieving usually 1st or 2nd rank, regardless of the existence or type of drift. To verify whether AUE performs better than the remaining algorithms, we compute the critical difference (CD) chosen by the Bonferroni-Dunn test [45]. When the difference between corresponding average ranks of two classifiers is greater or equal to CD, one can state that they are significantly different.

As $CD = 3.736$, AUE performs significantly better than NSE, DDM, DWM, AWE, ACE, Win, and NB. As for the difference between AUE and the remaining algorithms, the test's power for the number of considered datasets is not sufficient to reach such a conclusion. Motivated by the fact that AUE has an accuracy rank much higher than AUE$_{\text{pre}}$, HOT, Lev, and Bag, we have decided to additionally perform the Wilcoxon signed rank test to get a better insight into the comparison of pairs of classifiers [45]. In contrast to the Friedmann test, in the Wilcoxon signed rank test the values of differences in performance of a pair of classifiers are taken into account. The p-values resulting from this test are: $p_{AUE_{\text{pre}}} = 0.006$, $p_{HOT} = 0.020$, $p_{Lev} = 0.009$, $p_{Bag} = 0.003$ for AUE$_{\text{pre}}$, HOT, Lev, and Bag, respectively. All these p-values support our observation that AUE is better in terms of accuracy than any of the compared algorithms.

We perform a similar analysis concerning average classifier training time, also presented in Table 3.10. Computing the test statistic we obtain $F_F = 133.834$. The null hypothesis can be rejected and by comparing average algorithm ranks with CD and performing additional Wilcoxon signed rank tests we can state that AUE is trained slower than Win, NB, but significantly faster than NSE, AUE$_{\text{pre}}$, Lev, ACE, AWE, and DWM ($p_{Lev} = 0.023$, $p_{ACE} = 0.004$, $p_{AWE} = 0.001$, $p_{DWM} = 0.002$). This is a positive outcome, since Win and NB are single classifiers, which are hard to compete against in terms of processing time, while the remaining algorithms are ensemble methods just as AUE.

Analogously, comparing average testing time we also reject the null hypothesis ($F_F = 74.549$) and state that AUE classifies slower than DDM, HOT, Win, and NB, but faster than Bag and NSE ($p_{Bag} = 0.003$, $p_{NSE} = 0.004$). Such an outcome is not surprising as

Win, NB, HOT, and DDM are single classifiers, while the rest of the analyzed algorithms are ensembles, each with 10 component classifiers.

Finally, we compare the average memory usage of each algorithm. The test value being $F_F = 60.307$, we reject the null hypothesis. By comparing average ranks we can state that AUE uses more memory than Win, NB, and ACE, but is more memory efficient than NSE, AUE$_{\text{pre}}$, HOT, Lev ($p_{NSE} = 0.007$, $p_{AUE_{\text{pre}}} = 0.007$, $p_{HOT} = 0.015$, $p_{Lev} = 0.050$).

3.4 Conclusions

The Accuracy Updated Ensemble is a block-based ensemble classifier designed to react to different types of concept drift. The main novelty of the proposed algorithm is the combination of an AWE-inspired ensemble weighting mechanism with incremental training of component classifiers. Such a hybrid approach allows AUE to react to various types of concept changes, such as sudden, gradual, recurring, short-term, and mixed drifts. Additional contributions of AUE include the proposal of a new component weighting function and a cost-effective candidate weight. By treating the candidate classifier as a "perfect" classifier, AUE ensures that the current concept is strongly reflected in the ensemble's prediction. The proposed algorithm is also optimized for memory usage by restricting ensemble size and incorporating a simple inner-component pruning mechanism.

As part of this study, we investigated different strategies concerning component classifier updates. Our experiments have shown that, in terms of accuracy, all component classifiers in AUE should be updated after each incoming data block. Such an approach promotes the incremental creation of strong classifiers as ensemble members and provides more accurate predictions of the ensemble. From this point of view, our results coincide with those presented in [52], therefore, suggesting that drifting environments provide natural diversity and the premise of weaklearnability does not apply to them.

We have also carried out an experimental study comparing AUE with 11 additional state-of-the-art data stream methods, including single classifiers, ensembles, and hybrid approaches, in different scenarios. The obtained results confirm that classifiers without any drift reaction mechanism fail to successfully learn from data with sudden, gradual, or recurrent drifts. They also seem to confirm that ensemble approaches that use batch classifiers, such as AWE, may suffer accuracy drops after sudden concept drifts [130], while drift detectors are less accurate on gradually drifting streams [6]. Novel findings include the reaction of algorithms to short random abrupt changes. The obtained results show that ensemble methods are more robust to random blips than single classifiers, as previously trained components allow them to recover from premature reactions. Furthermore, experiments on datasets with fast recurring drifts have showcased that the speed of changes is crucial to the decision whether a buffer of previously constructed component classifiers is useful or not. If recurrent changes are very frequent a buffer can improve accuracy but in other cases it only increases memory requirements and algorithm processing time.

Above all, the experimental study has demonstrated that AUE can offer very high classification accuracy in environments with various types of drift as well as in static environments. AUE provided best average classification accuracy out of all the tested algorithms, while proving less memory consuming than other ensemble approaches, such as Leveraging Bagging or Hoeffding Option Trees. It is worth noting that AUE's predictive performance was consistent among every dataset, achieving practically always the best or second-best rank in terms of accuracy. Finally, AUE showcased faster average training time compared to all the tested ensemble approaches.

Chapter 4

Strategies for Transforming Block-based Ensembles into Online Learners

In contrast to block-based approaches, online ensembles are designed to learn in environments were labels are available after each example. With class labels arriving online, algorithms have the possibility of adapting to changes as quickly as it is possible. Many researchers tackle this problem by designing new online methods, ignoring weighting mechanisms known from block-based algorithms. However, we argue that these weighting mechanisms, as well as component evaluations and periodically created candidate classifiers, could still be of much value in online ensembles. Experimental results presented in Chapter 3 suggest that by modifying block-based ensembles towards incremental classifiers one can improve classification accuracy on gradual and sudden drifts.

In this chapter, we examine existing block-based ensembles and seek ways of adapting them to online environments. We put forward three general strategies for transforming block-based ensembles into online learners:

I) a windowing technique which updates component weights after each example,

II) the extension of the ensemble by an incremental classifier which is trained between component reweighting,

III) an online drift detector which allows to shorten drift reaction times.

Finally, we experimentally compare these modifications using popular block-based ensembles and highlight the most important factors in transforming block-based ensembles into online learners.

4.1 Generalization of Block-based Ensembles

Before discussing different approaches to converting block ensembles into online learners, let us recall the basics of block-based processing and the generic ensemble training scheme, which will help describe the proposed strategies.

Let $\mathcal{S}$ be a data stream partitioned into evenly sized blocks $B_1, B_2, \ldots, B_j$, each containing d examples. For every incoming block B_j, the weights of component classifiers $C_i \in \mathcal{E}$ are calculated by a classifier quality measure $Q(\cdot)$, often called a weighting function. The function behind $Q(\cdot)$ depends on the algorithm being analyzed. For example, AWE calculates weights based on the mean square error of components, whereas SEA scores ensemble members based on accuracy and diversity. In addition to component reweighting, a candidate classifier is built from block B_j and added to the ensemble if the ensemble's size k is not exceeded. If the ensemble already contains k components, but the candidate's quality measure is higher than at least one member's weight, the candidate classifier substitutes the weakest ensemble member.

Algorithm 4.1 Generic block-based ensemble training scheme

```
Input: $\mathcal{S}$: data stream of examples
       $d$: size of each data block $B_j$
       $k$: number of classifiers in the ensemble
       $Q(\cdot)$: classifier quality measure
Output: $\mathcal{E}$: ensemble of $k$ weighted classifiers

1: for all data blocks $B_j \in \mathcal{S}$ do
2:   build and weight candidate classifier $C'$ using $B_j$ and $Q(\cdot)$;
3:   weight all classifiers $C_i$ in ensemble $\mathcal{E}$ using $B_j$ and $Q(\cdot)$;
4:   if $|\mathcal{E}| < k$ then
5:     $\mathcal{E} \leftarrow \mathcal{E} \cup \{C'\}$;
6:   else if $\exists i : Q(C') > Q(C_i)$ then
7:     replace weakest ensemble member with $C'$;
8:   end if
9: end for
```

The described training scheme, presented in Algorithm 4.1, can be used to generalize most popular block-based ensemble classifiers, such as the Streaming Ensemble Algorithm (SEA) [155], Accuracy Weighted Ensemble (AWE) [163], Learn^{++}.NSE [52], or Batch Weighted Ensemble [39, 41]. The following sections present three different strategies for modifying this generic algorithm to suit online environments.

4.2 Strategy I: Online Evaluation of Components

The first strategy converts a data block into a sliding window. Instead of evaluating component classifiers every d examples, ensemble members are weighted after each example using the last d training instances. This way component weights are incrementally updated and can follow changes in data faster. Because the creation of the candidate classifier can be a costly process, especially in block-based ensembles which use batch component classifiers, we propose to add new classifiers to the ensemble every d examples, just as in the original block processing scheme. The described strategy is presented in Algorithm 4.2.

Although no direct drift detection mechanism is used, this strategy should enable quicker reactions to sudden changes. Furthermore, since new classifiers are constantly added to the ensemble, algorithms modified using this strategy should also remain accurate

Algorithm 4.2 Windowing strategy

Input: $\mathcal{S}$: data stream of examples
k: number of ensemble members
W: window of examples
d: size of window
$Q(\cdot)$: classifier quality measure
Output: $\mathcal{E}$: ensemble of k weighted classifiers

```
1:  for all examples $\mathbf{x}^t \in \mathcal{S}$ do
2:    if $|W| < d$ then
3:      $W \leftarrow W \cup \{\mathbf{x}^t\}$;
4:    else
5:      replace oldest example in $W$ with $\mathbf{x}^t$;
6:    end if
7:    weight all classifiers $C_i$ in ensemble $\mathcal{E}$ using $W$ and $Q(\cdot)$;
8:    if $t > 0$ and $t$ mod $d = 0$ then
9:      build and weight candidate classifier $C'$ using $W$ and $Q(\cdot)$;
10:     if $|\mathcal{E}| < k$ then
11:       $\mathcal{E} \leftarrow \mathcal{E} \cup \{C'\}$;
12:     else if $\exists i : Q(C') > Q(C_i)$ then
13:       replace weakest ensemble member with $C'$;
14:     end if
15:   end if
16: end for
```

during gradual drifts. However, it is important to notice that component classifiers are not incrementally trained and, therefore, the window size d is still a crucial parameter.

4.3 Strategy II: Introducing an Additional Incremental Learner

The second strategy involves directly using an incremental classifier as an extension of a block-based ensemble. The ensemble works exactly like in the original algorithm, except that an additional online learner, which is trained with each incoming example, is taken into account during component voting. Such a strategy ensures that the most recent data is included in the final prediction.

Two factors are crucial for the incremental classifier to have an effect on the ensemble's performance: its weight and its accuracy. We propose to use the maximum weight of remaining ensemble members as the candidate's weight. Using such a value ensures that this strategy remains independent of the algorithm being modified and that the incremental learner will have substantial voting power. As for accuracy, to ensure accurate predictions in a time changing environment a classifier should be trained only on the most recent data. On the other hand, using too few examples will make the classifier inaccurate. That is why, we propose to initialize the incremental learner with the last full buffer of examples. Subsequently, the additional online classifier is incrementally trained for the next

d examples, after which it is reinitialized. The pseudocode of Strategy II is presented in Algorithm 4.3.

Algorithm 4.3 Additional incremental learner strategy

```
Input: $\mathcal{S}$: data stream of examples
       $C_o$: online learner
       $k$: number of ensemble members
       $B$: example buffer of size $d$
       $Q(\cdot)$: classifier quality measure
Output: $\mathcal{E}$: ensemble of $k$ weighted classifiers and one online learner

1:  for all examples $\mathbf{x}^t \in \mathcal{S}$ do
2:    incrementally train $C_o$ with $\mathbf{x}^t$
3:    $B \leftarrow B \cup \{\mathbf{x}^t\}$
4:    if $t > 0$ and $t$ mod $d = 0$ then
5:      build and weight candidate classifier $C'$ using $B$ and $Q(\cdot)$;
6:      weight all classifiers $C_i$ in ensemble $\mathcal{E}$ using $B$ and $Q(\cdot)$;
7:      if $|\mathcal{E}| < k$ then
8:        $\mathcal{E} \leftarrow \mathcal{E} \cup \{C'\}$;
9:      else if $\exists i : Q(C') > Q(C_i)$ then
10:       replace weakest ensemble member with $C'$;
11:     end if
12:     reinitialize $C_o$ with $B$;
13:     $B \leftarrow \emptyset$;
14:   end if
15: end for
```

4.4 Strategy III: Using a Drift Detector

The last strategy uses a drift detector attached to an online learner. In periods of stability, when no drifts occur, the algorithm works similarly to the second strategy, with an online learner serving as an additional ensemble member. When a drift is detected the algorithm creates an "early" data block, which consists of examples collected after the last periodical reweighting. Using this block, a candidate classifier is built, added to the ensemble, and weighted according to $Q(\cdot)$ along with existing ensemble members being reweighted. Finally, the online learner and drift detector are reinitialized. The complete pseudocode for this strategy is presented in Algorithm 4.4.

Drift detectors are often a part of online classifiers that ensures quick reactions to sudden changes. However, drift detectors are not capable of detecting gradual changes and block-based algorithms can outperform online approaches in slowly drifting environments [34]. Strategy III aims at faster, online, reactions to sudden changes, while maintaining properties responsible for accurate predictions during gradual drifts. Like with previous strategies, d remains an important parameter responsible for the accuracy of component classifiers.

The three presented strategies tackle different aspects of reacting to drifts in online environments. Consequently, gradual updates using online weighting, faster training by

Algorithm 4.4 Drift detector strategy

```
Input: $\mathcal{S}$: data stream of examples
       $D$: drift detector
       $k$: number of ensemble members
       $B$: example buffer of size $d$
       $Q(\cdot)$: classifier quality measure
Output: $\mathcal{E}$: ensemble of $k$ weighted classifiers and one classifier with a drift detector

1: for all examples $\mathbf{x}^t \in \mathcal{S}$ do
2:   incrementally train $D$ with $\mathbf{x}^t$
3:   $B \leftarrow B \cup \{\mathbf{x}^t\}$
4:   if $|B| = d$ or drift detected then
5:     build and weight candidate classifier $C'$ using $B$ and $Q(\cdot)$;
6:     weight all classifiers $C_i$ in ensemble $\mathcal{E}$ using $B$ and $Q(\cdot)$;
7:     if $|\mathcal{E}| < k$ then
8:       $\mathcal{E} \leftarrow \mathcal{E} \cup \{C'\}$;
9:     else if $\exists i : Q(C') > Q(C_i)$ then
10:      replace weakest ensemble member with $C'$;
11:    end if
12:    reinitialize $D$;
13:    $B \leftarrow \emptyset$;
14:  end if
15: end for
```

using incremental learners, and instant reactions to detected changes, all have a different impact on the modified block-based ensemble. The following section, provides an experimental analysis of the proposed transformations in various drift scenarios, and highlights the most interesting properties of each strategy.

4.5 Experimental Evaluation

The aim of combining mechanisms known from block-based ensembles with incremental learners is to provide accurate reactions to different types of changes. In this section, we verify if such a combination is profitable by evaluating the performance of three transformation strategies presented in Sections 4.2–4.4.

4.5.1 Experimental Setup

In our experiments, we evaluate four versions (the original algorithm and the three proposed modifications) of two block-based ensembles: the Accuracy Weighted Ensemble (AWE) and the preliminary version of the Accuracy Updated Ensemble (AUE$_{\text{pre}}$) [30]. We chose AWE and AUE$_{\text{pre}}$, because periodical component weighting is very important to the performance of these algorithms. Moreover, both algorithms use similar processing schemes, e.g., candidate classifier cross-validation, but different weighting functions. More precisely, AWE uses a linear weighting function ($w_{AWE} = MSE_r - MSE_{ij}$), while AUE$_{\text{pre}}$ uses a non-linear one ($w_{AUE_{\text{pre}}} = 1/(MSE_{ij} + \epsilon)$). Finally, AWE uses batch decision trees, whereas AUE$_{\text{pre}}$ has Hoeffding Trees as components. Both algorithms and all their modifications were implemented in Java as part of the MOA framework. Experiments were performed on a machine equipped with an Intel Core i7-2640M @ 2.80 GHz processor and 10.00 GB of RAM.

All the tested ensembles used $k = 10$ component classifiers; for AWE those classifiers were J48 trees with default WEKA parameters, while AUE_{pre}, used Hoeffding Trees with adaptive Naive Bayes leaf predictions with a grace period $n_{min} = 100$, split confidence $\delta = 0.01$, and tie-threshold $\psi = 0.05$ [49]. We decided to use ten component classifiers as using more classifiers (tested systematically from two up to forty) linearly increased processing time and memory, but did not notably improve classification accuracy of any of the analyzed ensemble methods. The AWE and AUE modifications which used drift detectors utilized the Drift Detection Method [65] with a Hoeffding Tree. The data block size used was equal $d = 1000$ for all the datasets, as this size was considered the best suitable for block-based ensembles [155, 163].

The analyzed algorithms were evaluated with respect to time efficiency, memory usage, and classification accuracy. Since in this chapter we analyze scenarios where examples are labeled online (contrary to block processing discussed in Chapter 3), all performance measures were periodically calculated using the prequential evaluation method [69, 15, 21]. In prequential evaluation, only the most recent examples are taken into account when calculating performance measures. This way, performance values at a given moment in time reflect only the most recent data, making them suitable for online algorithm evaluation and drift detection [69]. In our experiments, we used an evaluation window of $d = 1000$ examples, which is the default value suggested in MOA.

4.5.2 Datasets

We used data stream generators available in the MOA framework to construct 11 synthetic datasets. These datasets were created using the Hyperplane [54, 163, 172], SEA [155], Random Tree, RBF, LED, and Waveform [15] generators and were designed to simulate various drift scenarios; details concerning the functioning of each generator were discussed in Section 3.3. We used the Hyperplane generator to create two datasets, called $\texttt{Hyper}_S$ and $\texttt{Hyper}_F$, with a continuous slow and fast incremental drift, respectively. Two datasets created using the LED generator contained no drift ($\texttt{LED}_{ND}$) and a mixed drift ($\texttt{LED}_M$) involving gradually changing concept abruptly switched after 500 k examples. Furthermore, a similar set of changes was tested using the Waveform generator ($\texttt{Wave}$, $\texttt{Wave}_M$). Reactions to solely sudden changes were tested using the Tree ($\texttt{Tree}_{SR}$) and SEA ($\texttt{SEA}_S$) generators. Moreover, we introduced gradual and gradual recurring changes to datasets $\texttt{SEA}_G$ and $\texttt{RBF}_{GR}$, respectively. Finally, using the RBF generator, we tested algorithm performance in the presence of short concept blips by creating the $\texttt{RBF}_B$ dataset. Detailed generator scripts are available in Appendix A.

Additionally, we considered five publicly available real datasets previously used to test the related ensemble algorithms in several data stream mining papers [132, 19, 177, 134, 170]. The `CovType` and `Poker`, and `Airlines` datasets are commonly used benchmarks, which were also utilized during the evaluation of the Accuracy Updated Ensemble, discussed in more detail in Section 3.3. For these three datasets it is difficult to precisely state when drifts occur, however, for the remaining two more information is available. In particular, the `PAKDD` data was intentionally gathered to evaluate model robustness against performance degradation caused by market gradual changes and was studied by many research teams [134]. Similarly, the `Power` dataset contains hourly information about a company's power supply and contains several concepts with identified moments of changes [170]. The main characteristics of all the datasets are given in Table 4.1.

Table 4.1: Characteristic of datasets used to test transformation strategies

Dataset	#Inst	#Attrs	#Classes	Noise	#Drifts	Drift type
`Airlines`	539 k	7	2	-	-	unknown
`CovType`	581 k	53	7	-	-	unknown
`Hyper`$_F$	1 M	10	2	5%	1	incremental
`Hyper`$_S$	1 M	10	2	5%	1	incremental
`LED`$_M$	1 M	24	10	30%	3	mixed
`LED`$_{ND}$	250 k	24	10	20%	0	none
`PAKDD`	50 k	30	2	-	-	unknown
`Poker`	1 M	10	10	-	-	unknown
`Power`	30 k	2	24	-	-	mixed
`RBF`$_B$	1 M	20	4	0%	2	blips
`RBF`$_{GR}$	1 M	20	4	0%	4	gradual recurring
`SEA`$_G$	1 M	3	4	10%	9	gradual
`SEA`$_S$	1 M	3	4	10%	3	sudden
`Tree`$_{SR}$	100 k	10	6	0%	15	sudden recurring
`Wave`	1 M	40	3	random	0	none
`Wave`$_M$	500 k	40	3	random	3	mixed

4.5.3 Analysis of Ensemble Transformation Strategies

Tables 4.2–4.4 present average prequential accuracy, processing time, and memory usage of the proposed transformation strategies applied on the analyzed ensemble methods. Algorithms modified using the online evaluation, incremental candidate, and drift detector strategies are denoted with subscripts: $_W$, $_C$, and $_D$, respectively. Apart from tabular summaries, we generated graphical plots depicting performance measures of all algorithms in time. We will analyze the most interesting plots to highlight characteristic features of each strategy.

Comparing the performance of AWE and its first modification, AWE$_W$, we can see that the windowing technique seems to improve classification accuracy on six out of sixteen datasets. However, the improvement comes at the cost of much higher processing time, which is a direct result of recalculating component weights after each processed instance. In order to recalculate weights, the algorithm needs to know the accuracy of each ensemble member, which is achieved by testing each component's predictions on the entire window of examples. The AWE$_C$ modification on the other hand, increases accuracy on practically all the datasets and does not require that much additional processing time. Finally, the classification accuracy of the AWE$_D$ modification seems to show that a simple addition

Table 4.2: Average prequential accuracy of different transformation strategies [%]

	AWE	AWE$_W$	AWE$_C$	AWE$_D$	AUE$_{\text{pre}}$	AUE$_{\text{pre}W}$	AUE$_{\text{pre}C}$	AUE$_{\text{pre}D}$
`Airlines`	63.64	63.26	63.44	59.53	61.80	66.72	62.57	63.15
`CovType`	85.70	79.92	87.34	41.97	82.97	87.57	84.60	56.45
`Hyper`$_F$	87.75	88.26	88.48	54.31	89.44	90.34	89.03	90.35
`Hyper`$_S$	77.36	84.70	80.03	74.41	86.55	88.73	86.34	88.76
`LED`$_M$	45.43	50.63	46.97	48.37	53.03	53.38	52.99	53.39
`LED`$_{ND}$	38.46	46.42	44.13	44.94	51.42	51.42	51.42	51.44
`PAKDD`	80.28	80.28	79.78	79.76	80.26	80.24	80.21	80.27
`Poker`	79.36	74.02	80.87	51.89	60.34	75.67	67.14	62.88
`Power`	11.23	11.92	11.35	4.17	15.46	15.26	15.56	14.98
`RBF`$_B$	95.53	95.27	95.77	64.54	97.00	97.68	96.19	97.23
`RBF`$_{GR}$	94.81	94.25	95.08	32.30	96.19	97.26	95.54	97.22
`SEA`$_G$	88.44	88.34	88.45	85.07	87.45	88.47	86.44	88.73
`SEA`$_S$	88.60	88.58	88.58	83.92	88.39	89.06	87.03	89.11
`Tree`$_{SR}$	58.62	43.49	59.15	54.04	43.05	42.26	44.88	42.21
`Wave`	81.61	81.71	82.82	76.09	83.06	85.46	81.78	85.55
`Wave`$_M$	81.16	80.95	82.55	78.93	82.27	84.72	81.33	84.79

Table 4.3: Average time required to process $d = 1000$ examples by different transformation strategies [s]

	AWE	AWE$_W$	AWE$_C$	AWE$_D$	AUE$_{\text{pre}}$	AUE$_{\text{pre}W}$	AUE$_{\text{pre}C}$	AUE$_{\text{pre}D}$
`Airlines`	1.47	2.22	2.94	0.49	0.41	21.46	0.66	0.63
`CovType`	0.50	9.17	0.70	0.25	0.42	101.40	0.46	0.51
`Hyper`$_F$	0.49	5.87	0.48	0.26	0.81	20.16	0.33	0.42
`Hyper`$_S$	0.45	8.63	0.51	0.13	0.76	24.83	0.24	0.28
`LED`$_M$	0.19	25.82	0.75	0.17	0.30	69.83	0.24	0.29
`LED`$_{ND}$	0.44	20.45	0.90	0.20	0.78	87.33	0.25	0.29
`PAKDD`	6.43	45.66	6.51	3.93	0.48	8.23	0.39	0.33
`Poker`	0.41	20.08	0.46	0.05	0.07	25.89	0.09	0.12
`Power`	0.36	36.07	0.37	0.07	0.20	55.62	0.22	0.28
`RBF`$_B$	0.58	7.69	0.70	0.09	0.56	78.58	0.58	0.59
`RBF`$_{GR}$	0.65	9.97	0.73	0.10	0.70	75.19	0.73	0.82
`SEA`$_G$	0.18	2.70	0.34	0.11	0.12	7.08	0.12	0.13
`SEA`$_S$	0.31	2.73	0.31	0.10	0.31	5.90	0.20	0.23
`Tree`$_{SR}$	0.48	14.77	0.56	0.16	0.37	43.63	0.27	0.40
`Wave`	0.79	5.84	1.02	0.49	7.45	107.05	2.81	3.28
`Wave`$_M$	0.91	6.62	1.13	0.16	0.94	142.76	0.87	0.98

of a drift detector is not sufficient to improve reactions on sudden drifts without deteriorating the ensemble's ability to react to gradual changes. This issue is clearly visible on the accuracy plots for the `RBF`$_{GR}$ and `SEA`$_S$ datasets, presented in Figures 4.1 and 4.2, respectively. We can see that in the presence of frequent gradual changes (`RBF`$_{GR}$) the drift detector forces AWE to, needlessly, dispose existing components, whereas during sudden drifts (`SEA`$_S$) AWE is rebuilding the ensemble too slowly because of having to train static decision trees.

Table 4.4: Average ensemble memory usage for different strategies [MB]

	AWE	AWE_W	AWE_C	AWE_D	AUE_{pre}	$\text{AUE}_{\text{pre}W}$	$\text{AUE}_{\text{pre}C}$	$\text{AUE}_{\text{pre}D}$
Airlines	10.08	7.45	11.31	8.62	1.10	81.61	1.95	7.26
CovType	6.09	6.13	6.15	4.25	1.55	2.22	1.60	0.78
Hyper_F	2.43	2.47	2.50	6.59	1.85	1.99	1.88	6.39
Hyper_S	2.57	2.61	2.66	3.22	2.18	2.25	2.19	2.94
LED_M	5.44	5.48	5.83	4.90	0.25	0.49	0.29	1.84
LED_{ND}	7.10	7.13	7.53	5.47	0.25	0.51	0.29	0.96
PAKDD	26.16	26.20	26.17	11.31	1.60	2.76	1.60	2.10
Poker	2.45	2.49	2.51	0.45	0.14	0.75	0.16	0.34
Power	10.01	10.01	11.03	0.19	0.10	0.16	0.11	0.19
RBF_B	3.18	3.26	3.24	1.93	5.84	6.37	5.88	6.60
RBF_{GR}	3.23	3.29	3.30	1.81	7.78	9.91	7.82	10.58
SEA_G	1.52	1.55	1.55	1.81	1.62	1.70	1.63	1.88
SEA_S	1.50	1.54	1.54	1.91	3.13	3.21	3.14	3.60
Tree_{SR}	3.45	4.10	3.67	2.51	2.20	2.65	2.25	3.12
Wave	5.06	5.10	5.15	8.70	41.69	40.64	41.74	51.40
Wave_M	5.05	5.08	5.13	4.01	6.69	6.78	6.74	9.26

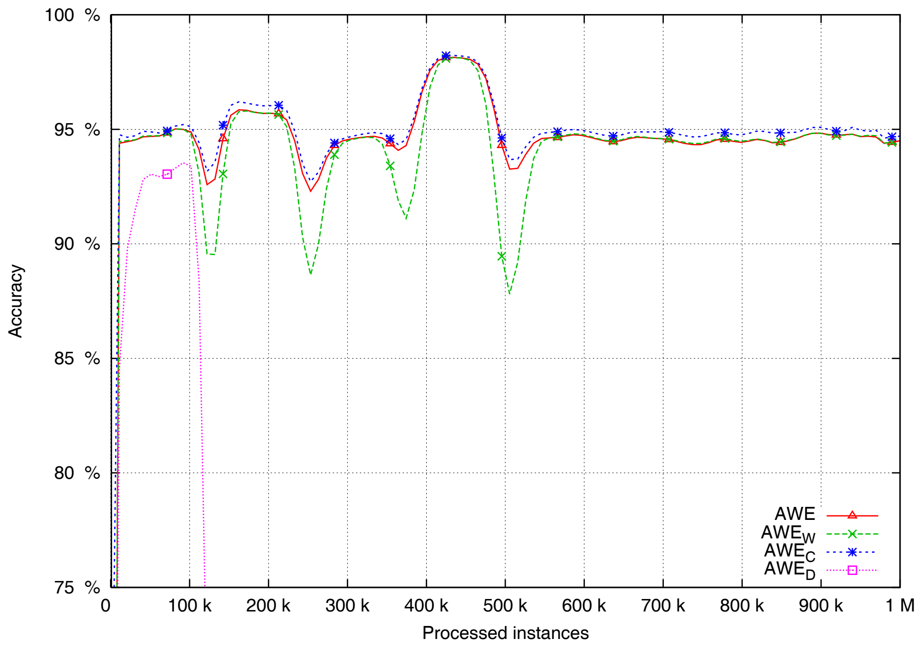


Figure 4.1: Prequential accuracy of AWE and its modifications on the RBF_{GR} dataset containing gradual recurring drifts

The differences between accuracies of AWE and its modifications were verified to be statistically significant by performing the Friedman test at $\alpha = 0.05$. Furthermore, by performing a series of Wilcoxon tests it was confirmed that AWE_C increases ($p_C = 0.002$) while AWE_D deteriorates ($p_D = 0.002$) the accuracy of AWE.

Looking at the results of AUE_{pre} and its modifications, we can see trends slightly different from those observed for AWE. The $\text{AUE}_{\text{pre}W}$ modification improves classification accuracy much more than AWE_W, but at higher processing costs. As $\text{AUE}_{\text{pre}W}$ updates

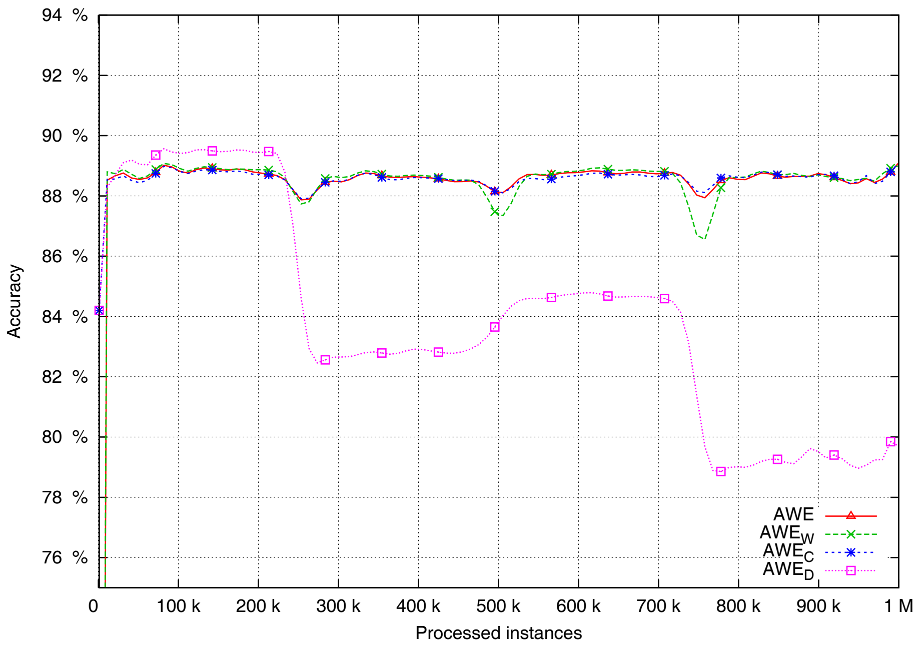


Figure 4.2: Prequential accuracy of AWE and its modifications on the $\texttt{SEA}_S$ dataset containing sudden changes

existing component classifiers, it can grow larger component Hoeffding Trees, which require more time to test on a window of examples. Thus, the windowing technique is much more time consuming when used to modify AUE_{pre} than it was on AWE.

The additional incremental classifier, present in $\text{AUE}_{\text{pre}C}$, allows to improve AUE_{pre}'s accuracy on fast changing datasets such as $\texttt{Tree}_{SR}$, `CovType`, `Poker`, and `Power`, but does not seem to be so useful on slower changing data. This is probably the effect of using a static (maximum) weight for the incremental candidate; in AWE which uses a linear weighting function it had a stronger impact than in AUE_{pre} which uses a nonlinear weighting function (the difference between using a linear and nonlinear function will be analyzed in Chapter 5). Nevertheless, the use of an additional incremental component gives comparable or better accuracy than the original AUE_{pre} at very small time and memory costs.

Finally, the use of a drift detector with AUE_{pre} proves more rewarding than its addition to AWE. Since, AUE_{pre} components can be incrementally updated after a drift is detected, $\text{AUE}_{\text{pre}D}$ manages to build strong component classifiers, while AWE is left with weak learners after each drift. This seems to show that when combined with periodical incremental component updates a drift detector can enhance sudden drift reactions without degrading performance on gradual changes. As Table 4.2 shows, accuracies of AUE_{pre} and its modifications are generally higher than AWE's which could also be caused by incremental updating of component classifiers.

Figures 4.3 and 4.4 present the performance of AUE_{pre} and its modification on $\texttt{RBF}_{GR}$ and $\texttt{SEA}_S$ datasets. By comparing these plots with Figures 4.1 and 4.2, we can see that AUE achieves higher accuracy than AWE. Furthermore, $\text{AUE}_{\text{pre}D}$ performs well on these datasets, which was not true for AWE. $\text{AUE}_{\text{pre}C}$ on the other hand seems to perform

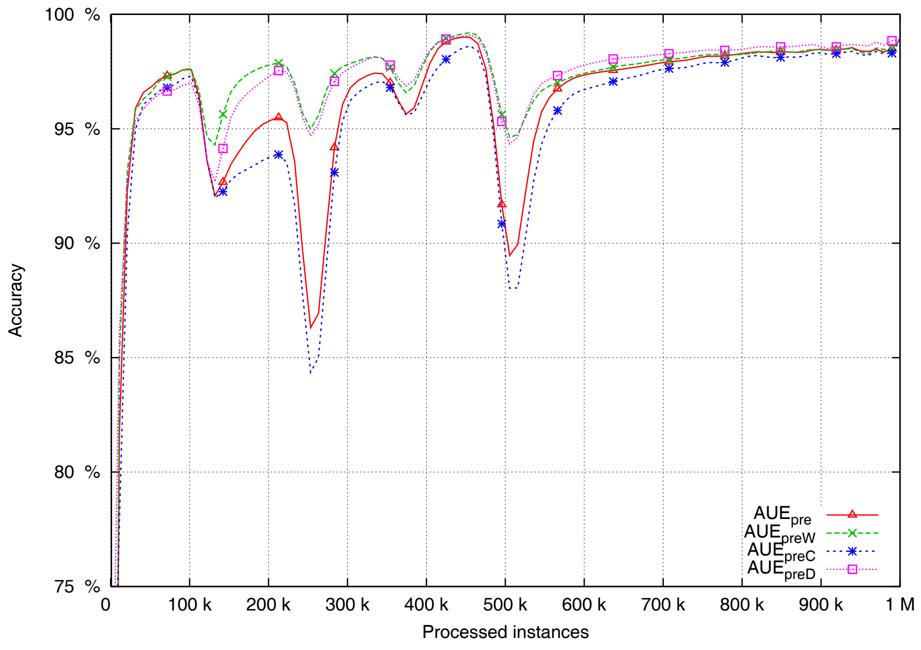


Figure 4.3: Prequential accuracy of AUE and its modifications on the $\texttt{RBF}_{GR}$ dataset containing gradual recurring drifts

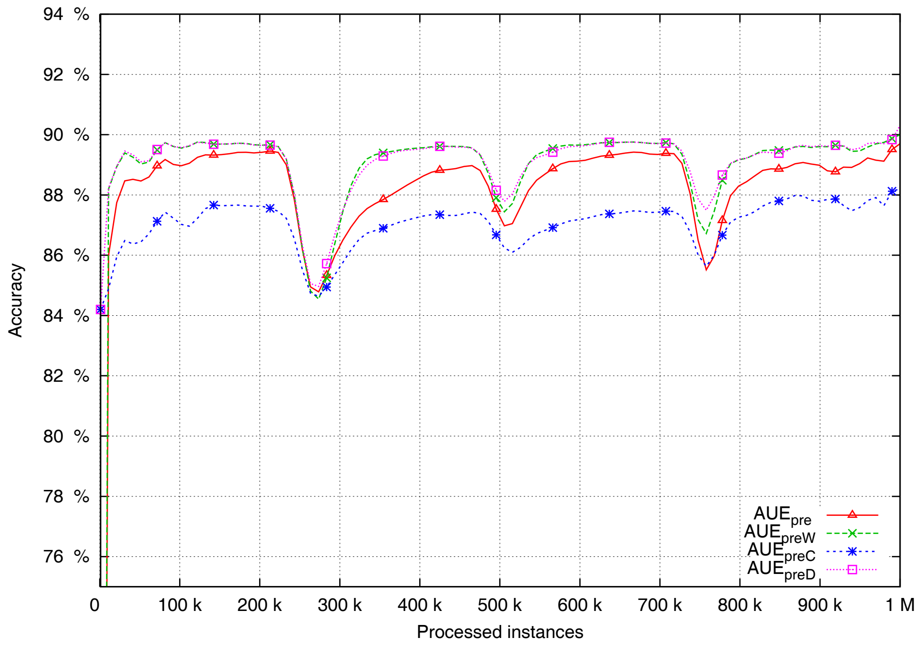


Figure 4.4: Prequential accuracy of AUE and its modifications on the $\texttt{SEA}_{S}$ dataset containing sudden changes

worse than the original algorithm, both during drift and in times of stability. As mentioned earlier, this is caused by using a static weight for the incremental candidate. Finally, it is worth noting that AUE$_{\text{pre}W}$ is performing better than the original algorithm, especially during gradual drifts. The trends visible in Figures 4.3 and 4.4 were common for most accuracy plots of AUE$_{\text{pre}}$ and its modifications.

As with AWE, the differences between accuracies of AUE$_{\text{pre}}$ and its modifications were verified to be statistically significant by performing the Friedman test at $\alpha = 0.05$. Additional Wilcoxon tests have shown that AUE$_{\text{pre}W}$ and AUE$_{\text{pre}D}$ significantly increase ($p_W = 0.001$, $p_D = 0.01$) the accuracy of AUE$_{\text{pre}}$.

4.6 Conclusions

In this chapter, we analyzed the problem of integrating weighting mechanisms and periodical component evaluations, known from block-based ensembles, into online classifiers. To verify the validity of such an approach, we proposed and evaluated three strategies to transforming block-based classifiers into online learners: a windowing technique, the use of an additional incremental learner, and the use of a drift detector. Experimental results demonstrated that all three strategies can be beneficial to the performance of a block-based ensemble using both static and incremental base classifiers.

However, not all strategies were equally effective. We have observed that online component reweighting is the best transformation strategy in terms of average prequential accuracy. Unfortunately, it is also the costliest strategy in terms of processing time, as it requires estimating each component's predictive performance after every example. Moreover, we have noticed that elements of incremental learning are crucial in improving classification accuracy — for an ensemble composed of static learners, adding an incremental learner was the best strategy. Finally, we found that drift detectors can also be beneficial in reacting to changes, but only for algorithms capable of quick ensemble training.

Above all, we have noticed that the transformation of a block-based ensemble to an online learner should be tailored to a given algorithm. Although the proposed strategies successfully introduced properties of online ensembles, they were too general to be computationally efficient. In particular, the online reweighting strategy, although profitable in terms of accuracy, drastically increased processing time. These findings were our motivation for creating an online ensemble, which tries to incorporate incremental learning with online component reweighting in a time and memory efficient manner. In the following chapter, we present a new algorithm called Online Accuracy Updated Ensemble, which aims at fulfilling these goals.

Chapter 5

The Online Accuracy Updated Ensemble

In environments where class labels are available after each example, ensembles which process instances in blocks do not react to sudden changes sufficiently quickly. On the other hand, ensembles which process streams online, do not take advantage of periodical adaptation mechanisms known from block-based ensembles, which offer accurate reactions to gradual changes.

In this chapter, we analyze whether the characteristics of block and online processing can be combined to produce new types of ensemble classifiers. In particular, we investigate the possibility of tailoring the strategies presented in Chapter 4 to a specific algorithm, in an attempt to minimize the additional processing cost introduced by these strategies. As a result, we put forward a new incremental ensemble classifier, called Online Accuracy Updated Ensemble, which weights component classifiers based on their error, in constant time and memory. The proposed algorithm is experimentally compared with four state-of-the-art online ensembles, providing best average classification accuracy on 16 datasets simulating various drift scenarios.

5.1 Block-based Weighting in Online Environments

In environments were examples arrive in portions, block-based ensembles offer the possibility of using batch algorithms and, if component classifiers are correctly weighted, achieve higher accuracy than a single classifier trained on all available examples [163]. However, in online environments batch algorithms can be too expensive in terms of required processing time and memory.

Additionally, determining the block size of a block-based ensemble is a non-trivial task, which requires finding a compromise between accurate predictions and fast reactions to changes [172, 30]. Theoretically, one could even use blocks containing single examples, thus, allowing a block-based ensemble to work online. However, component classifiers require more than one example to give satisfactory predictions and more than one example is also needed when evaluating the classification performance of a component. Therefore, in practice it is impossible to correctly weight an ensemble of 1000 classifiers built on one example each, and require it to be more accurate than a single classifier built on 1000 examples.

Strategies proposed in Chapter 4, showcased alternative approaches to adapting block-based ensembles to online processing. The analysis of these approaches has shown that the use of incremental learners and online weighting are important aspects in converting a block-based learner into an online ensemble. These results coincide with our experiences with the Accuracy Updated Ensemble, where periodical incremental updates of components using blocks of examples were a key factor in achieving high accuracy [30, 34].

However, the transformation strategies proposed in Chapter 4 were not tailored to any specific algorithm and, for this reason, were not always computationally efficient. In particular, online component weighting, although effective in terms of enhancing the predictive performance of the transformed algorithms, was found to be extremely costly in terms of time and memory overhead. This issue motivated the following research question: Can error-based weighting proposed for block-based methods be performed after each example, without the need of dividing data into blocks? In the following section, we try to answer this question by introducing a new ensemble classifier that uses incremental learners and block-based weighting mechanisms, but in an online, time and memory efficient manner.

5.2 The Online Accuracy Updated Ensemble

The proposed algorithm, called Online Accuracy Updated Ensemble, maintains a weighted set of component classifiers and predicts the class of incoming examples by aggregating the predictions of ensemble members using a weighted voting rule. After processing a new example, each component classifier is weighted according to its accuracy and incrementally trained. We retain an approach common for block-based ensembles, where every d examples a new candidate classifier is created which substitutes the poorest performing ensemble member. In our analysis and experiments, we will use Hoeffding Trees as component classifiers, but it is important to note that one could use any online learning algorithm as a base learner. The key novel element of the Online Accuracy Updated Ensemble, presented in Algorithm 5.1, is a block-based inspired weighting function, which we discuss in more detail in the following paragraphs.

Let $\mathcal{S}$ be a data stream. For each incoming example $\mathbf{x}^t$, the weights w_i^t of component classifiers $C_i \in \mathcal{E}$ $(i = 1, 2, \ldots, k)$ are calculated by estimating the prediction error on the last d examples as shown in Equations 5.1–5.5. Function $f_{iy}^t(\mathbf{x}^t)$ denotes the probability given by classifier C_i that $\mathbf{x}^t$ is an instance of class y^t. It is important to note that, instead of single class predictions, probabilities of all classes are considered. MSE_i^t estimates the prediction error of classifier C_i on the last d examples, τ_i denotes the time at which classifier C_i was created, whereas MSE_r^t is the mean square error of a randomly predicting classifier (also trained on the last d examples) which is used as a reference point to predictions made based on the current class distribution. Additionally, to ensure that each component classifier always receives a positive weight, ϵ is added to w_i^t. The described formula resembles the non-linear weighting function of AUE (discussed in Chapter 3), but aims at providing error estimates and weights after each example.

Algorithm 5.1 Online Accuracy Updated Ensemble (OAUE)

Input: $\mathcal{S}$: stream of examples
d: window size
k: number of ensemble members
m: memory limit
Output: $\mathcal{E}$: ensemble of k weighted incremental classifiers

```
 1: $\mathcal{E} \leftarrow \emptyset$;
 2: $C' \leftarrow$ new candidate classifier;
 3: for all examples $\mathbf{x}^t \in \mathcal{S}$ do
 4:   calculate the prediction error of all classifiers $C_i \in \mathcal{E}$ on $\mathbf{x}^t$;
 5:   if $t > 0$ and $t$ mod $d = 0$ then
 6:     if $|\mathcal{E}| < k$ then
 7:       $\mathcal{E} \leftarrow \mathcal{E} \cup \{C'\}$;
 8:     else
 9:       weight all classifiers $C_i \in \mathcal{E}$ and $C'$ using (5.5);
10:       substitute least accurate classifier in $\mathcal{E}$ with $C'$;
11:     end if
12:     $C' \leftarrow$ new candidate classifier;
13:     if $memory\_usage(\mathcal{E}) > m$ then
14:       prune (decrease size of) component classifiers;
15:     end if
16:   else
17:     incrementally train classifier $C'$ with $\mathbf{x}^t$;
18:     weight all classifiers $C_i \in \mathcal{E}$ using (5.5);
19:   end if
20:   for all classifiers $C_i \in \mathcal{E}$ do
21:     incrementally train classifier $C_i$ with $\mathbf{x}^t$;
22:   end for
23: end for
```

$$MSE_i^t = \begin{cases} MSE_i^{t-1} + \frac{e_i^t}{d} - \frac{e_i^{t-d}}{d}, & t - \tau_i > d \\ \frac{t-\tau_i-1}{t-\tau_i} \cdot MSE_i^{t-1} + \frac{e_i^t}{t-\tau_i}, & 1 \leq t - \tau_i \leq d \\ 0, & t - \tau_i = 0 \end{cases} \tag{5.1}$$

$$e_i^t = (1 - f_{iy}^t(\mathbf{x}^t))^2 \tag{5.2}$$

$$MSE_r^t = \begin{cases} MSE_r^{t-1} - r^{t-1}(y^t) - r^{t-1}(y^{t-d}) + r^t(y^t) + r^t(y^{t-d}), & t > d \\ \sum_y r^t(y), & t = d \end{cases} \tag{5.3}$$

$$r^t(y) = p^t(y)(1 - p^t(y))^2 \tag{5.4}$$

$$w_i^t = \frac{1}{MSE_r^t + MSE_i^t + \epsilon} \tag{5.5}$$

The presented formulas for calculating MSE_i^t and MSE_r^t are incremental versions of evaluation measures used by Wang et al. to weight component classifiers of the Accuracy Weighted Ensemble [163], which worked on blocks of examples B_j:

$$MSE_{ij} = \frac{1}{|B_j|} \sum_{\{\mathbf{x},y\} \in B_j} (1 - f_y^i(\mathbf{x}))^2 \tag{5.6}$$

$$MSE_r = \sum_y p(y)(1 - p(y))^2 \tag{5.7}$$

Instead of remembering a block of last d examples and performing component evaluations on the same instance multiple times (as described in the strategy presented in Section 4.2), we derived an incremental versions of Equations 5.6 and 5.7.

A newly added classifier ($t - \tau_i = 0$) is treated like an error-less classifier ($MSE_i^t = 0$). Such an approach is based on the assumption that the most recent block or window provides the best representation of the current data distribution and is a direct inspiration from our work concerning the Accuracy Updated Ensemble presented in Chapter 3. For $t - \tau_i \leq d$ we scale the mean-square error calculated on the previous example and add the prediction error calculated on $\mathbf{x}^t$. When a component classifier has been trained on more than d examples ($t - \tau_i > d$), the prediction errors used for weight calculation are limited to the last d, to evaluate only on the most recent data. The influence of different d values as well as the possible use of a linear weighting function will be discussed in Section 5.3.

The equation for MSE_r^t was built analogously, with the difference that instead of adding and removing errors, distributions (r^t) of the class of the newest (y^t) and oldest (y^{t-d}) example are updated, and that MSE_r^t is first calculated after creating the first component (after d examples).

Let us now analyze the complexity of the Online Accuracy Updated Ensemble. As Hoeffding Trees can be learned in constant time per example [49], the training of an ensemble of k Hoeffding Trees has a complexity of $O(k)$. Additionally, the weighting procedure defined by Equations 5.1–5.5 requires a constant number of operations, thus, for weighting k components $O(k)$ time is needed. Therefore, the training and weighting of OAUE has a complexity of $O(2k)$ per example and since k is a user-defined constant this resolves to a complexity of $O(1)$. It is worth noting that the same would be true for any other constant time per example component classifier, e.g., the Naive Bayes algorithm.

The memory requirements of an ensemble of Hoeffding Trees depend on the concept being learned and can be denoted as $O(kavcl)$, where a is the number of attributes, v is the maximum number of values per attribute, c is the number of classes, and l is the number of leaves in the tree [49]. The weighting mechanism of OAUE increases this value by d per component for calculating MSE_i^t and c for calculating MSE_r^t, which gives a total of $O(kavcl + k(d+c))$. Since k, d, and c are constants, the proposed weighting scheme does not increase the asymptotic space complexity compared to an ensemble without weighting.

In contrast to representative block-based ensembles like the Accuracy Weighted Ensemble [163] or Streaming Ensemble Algorithm [155], the proposed approach does not use static batch learners to construct component classifiers and does not divide the stream into blocks. OAUE utilizes the notion of accuracy-based weighting introduced in AWE, but it does not require a block of d examples and does not evaluate component classifiers on more than one example at a time. Instead, OAUE processes the stream one instance at a time and only requires each component classifier to remember its error on the last d examples. This means that the memory used by OAUE, apart from the memory used by its component classifiers, is dictated solely by the ensemble size k and the number of predictions used for weighting d and is, therefore, stream-invariant. Additionally, since component classifiers are incrementally trained after each example, OAUE is much less sensitive to the number examples between each new candidate classifier.

The Online Accuracy Updated Ensemble also differs from other data stream ensemble classifiers. Ensemble members of OAUE are incrementally weighted and periodically removed, unlike in Online Bagging [132] or Leveraging Bagging [16]. In contrast to the Adaptive Classifier Ensemble [130], OAUE does not use any drift detector or static batch learner and does not process the stream in blocks. In comparison with Learn++.NSE [52], the proposed algorithm incrementally trains all existing component classifiers after each example, retains only k of all the created components, and uses a different weighting function which ensures that all components will have positive weights. In contrast to the Dynamic Weighted Majority [91], OAUE weights components according to their prediction error, treats the candidate classifier as a perfect learner, and its weighting function does not require any user-specified parameters. It is also worth noting that the Dynamic Weighted Majority is only capable of penalizing component classifiers using a user-specified value, while OAUE, thanks to its memory of last d component errors, can penalize or reward components according to their mean square error.

Although weights of component classifiers in OAUE are calculated on a window of last d errors, it is not similar to sliding window algorithms used in data stream classification. In contrast to algorithms like ADWIN [19], OAUE does not aim at direct detection of drifts by analyzing windows of examples. OAUE also does not store, weight, or select past examples like FISH [172] or algorithms using decay functions [37]. Moreover, OAUE differs from algorithms that integrate sliding windows to calculate additional statistics to rebuild or prune parts of a single classifier, like in CVFDT [80] and other extensions of online decision trees. In contrast to the WWH algorithm from Yoshida et al. [171], we do not build component classifiers on overlapping windows to select the best learning examples or modify the Weighted Majority Algorithm. Finally, unlike algorithms using sliding windows, OAUE periodically reconstructs the ensemble by replacing component classifiers.

5.3 Experimental Evaluation

In this section, we summarize experiments conducted during the creation and evaluation of the proposed OAUE algorithm. In the first part, we study the impact of different elements of the proposed algorithm. Then, we compare OAUE with state-of-the-art online ensembles.

5.3.1 Experimental Setup

The proposed Online Accuracy Updated Ensemble (OAUE) is compared against four online ensembles, whose selection will be explained in Section 5.3.3. We chose exclusively ensembles that can be trained online, because in environments where labels are available after each example, block-based ensembles learn and react to drifts much slower than online approaches. The analyzed algorithms were implemented in Java as part of the MOA framework [15] and tested on a machine equipped with an Intel Core i7-2640M @ 2.80 GHz processor with 10 GB of RAM.

All the ensembles used $k = 10$ component classifiers; for ACE those classifiers were J48 trees with default WEKA parameters, while OAUE, Bag, Lev, and DWM used Hoeffding Trees with adaptive Naive Bayes leaf predictions with a grace period $n_{min} = 100$, split confidence $\delta = 0.01$, and tie-threshold $\psi = 0.05$ [49]. We decided to use ten component classifiers as using more classifiers (tested systematically from two up to forty) linearly increased processing time and memory, but did not notably improve classification accuracy of any of the analyzed ensemble methods. The data block size used for ACE was equal $d = 1000$ as this size was considered the best suitable for block ensembles [155, 163]. Analogously, OAUE and DWM used a window size/evaluation period of $d = 1000$.

The analyzed algorithms were evaluated with respect to time efficiency, memory usage, and accuracy. All evaluation measures were periodically calculated using the prequential evaluation method [69, 15, 21] with a window of $d = 1000$ examples. As in Chapter 4, we chose to perform prequential evaluations, as this method is more suitable for online algorithms than block or holdout evaluations [69].

The algorithms were tested using the same 11 synthetic and 5 real datasets that were described in Section 4.5.2. Table 5.1 recalls the main characteristics of each dataset.

Table 5.1: Characteristic of datasets used to evaluate the OAUE algorithm

Dataset	#Inst	#Attrs	#Classes	Noise	#Drifts	Drift type
Airlines	539 k	7	2	-	-	unknown
CovType	581 k	53	7	-	-	unknown
$\texttt{Hyper}_F$	1 M	10	2	5%	1	incremental
$\texttt{Hyper}_S$	1 M	10	2	5%	1	incremental
$\texttt{LED}_M$	1 M	24	10	30%	3	mixed
$\texttt{LED}_{ND}$	250 k	24	10	20%	0	none
PAKDD	50 k	30	2	-	-	unknown
Poker	1 M	10	10	-	-	unknown
Power	30 k	2	24	-	-	mixed
$\texttt{RBF}_B$	1 M	20	4	0%	2	blips
$\texttt{RBF}_{GR}$	1 M	20	4	0%	4	gradual recurring
$\texttt{SEA}_G$	1 M	3	4	10%	9	gradual
$\texttt{SEA}_S$	1 M	3	4	10%	3	sudden
$\texttt{Tree}_{SR}$	100 k	10	6	0%	15	sudden recurring
Wave	1 M	40	3	random	0	none
$\texttt{Wave}_M$	500 k	40	3	random	3	mixed

5.3.2 Analysis of OAUE Components

As in block-based ensembles the block size is a parameter which largely influences the accuracy of the ensemble [163], we decided to verify the impact of using different block/window sizes d for calculating the mean square error (MSE_i^t, MSE_r^t) in OAUE. It is worth noting that the d parameter in OAUE is responsible not only for the number of examples used to create candidate classifiers, but also the sliding window used to calculate the mean square error (MSE_i^t, MSE_r^t). Table 5.2 presents the average prequential accuracy of OAUE on different datasets with $d \in [500; 2000]$.

Table 5.2: Average prequential accuracy [%] of OAUE for different window sizes d

	Window size						
	500	750	1000	1250	1500	1750	2000
`Airlines`	67.50	66.93	67.03	67.12	66.72	66.33	66.23
`CovType`	90.07	90.85	90.91	91.08	91.43	91.51	91.58
`Hyper`$_F$	90.55	90.43	90.42	90.26	90.30	90.24	90.19
`Hyper`$_S$	89.05	89.04	88.94	89.00	88.98	88.92	88.97
`LED`$_M$	53.40	53.40	53.38	53.24	52.65	52.40	52.38
`LED`$_{ND}$	51.54	51.48	51.40	51.39	51.35	51.27	51.28
`PAKDD`	80.24	80.23	80.23	80.20	80.20	80.20	80.17
`Poker`	81.54	87.92	88.87	90.18	90.81	92.01	92.65
`Power`	15.73	15.58	15.54	15.34	15.27	15.23	14.87
`RBF`$_B$	96.78	97.59	97.83	97.84	97.96	98.00	97.90
`RBF`$_{GR}$	96.69	97.27	97.38	97.46	97.56	97.53	97.43
`SEA`$_G$	88.95	88.85	88.81	88.79	88.70	88.67	88.62
`SEA`$_S$	89.41	89.32	89.31	89.28	89.23	89.22	89.15
`Tree`$_{SR}$	46.23	46.05	45.86	45.21	44.39	43.66	43.28
`Wave`	84.34	85.25	85.47	85.58	85.53	85.50	85.49
`Wave`$_M$	83.86	84.75	84.85	84.87	84.86	84.73	84.66

Additionally, in Figure 5.1 we present three box plots summarizing the differences in accuracy, memory usage, and testing time of OAUE for different window sizes. The plots were created by calculating, for each dataset, the proportional differences between the mean performance over all window sizes and the performance for a certain window size d. For example, for the `Airlines` dataset the mean accuracy for all window sizes $d \in [500; 2000]$ is 66.83%. Therefore, the proportional difference for OAUE with $d = 500$, which has an average accuracy on `Airlines` equal 67.50%, is $\frac{67.50-66.83}{66.83} \approx +1.00\%$. This operation was repeated for all the datasets to create a box plot for a given d.

Analyzing the values in Table 5.2, one can see that differences in each row are small and no global dependency upon d can be seen. Furthermore, the box plot in Figure 5.1 (a) shows that most values are within 1% from the mean value on each dataset. Conversely, clear tendencies are visible on the plots of time and memory usage in Figures 5.1 (b) and (c). The larger the window size, the more time and memory consuming the classification. This dependency is a consequence of creating each new classifier using d examples. When d grows, so does each candidate classifier, which corresponds to higher time and memory requirements.

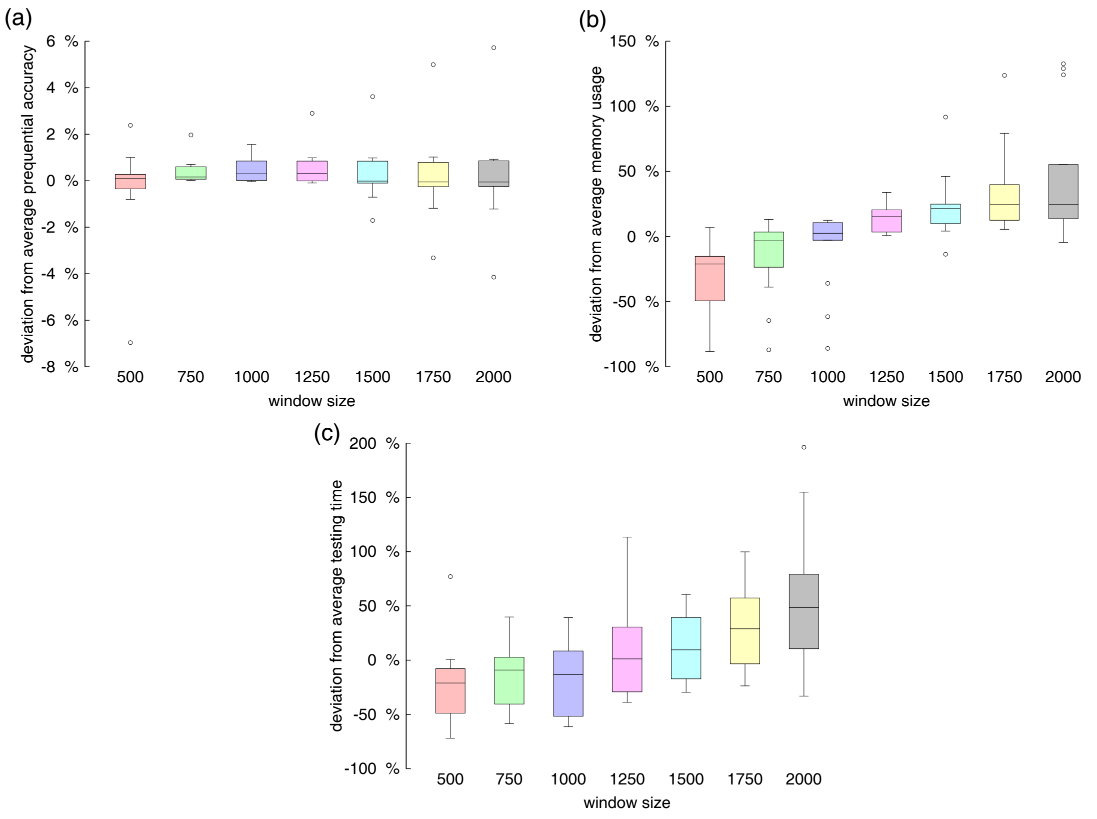


Figure 5.1: Box plot of average prequential accuracy (a), memory usage (b), and testing time (c) for window sizes $d \in [500; 2000]$. The depicted values are the percentage deviation from the average on each dataset.

Performing a Friedman test on the calculated deviations for $d \in [500; 2000]$ we obtain $F_{F_{Acc}} = 2.183$, $F_{F_{Mem}} = 34.594$, and $F_{F_{Time}} = 3.857$ for accuracy, memory usage, and evaluation time, respectively. As the critical value for comparing 7 values on 16 datasets with $\alpha = 0.05$ is 2.201, we reject the null-hypothesis for memory and time, but not accuracy. According to the Friedman test, we can state that there is a difference in average processing time and memory usage for different values of d, but concerning accuracy there is no significant difference. As there is no strong dependency upon d in terms of accuracy, but time and memory are proportional, in subsequent experiments we used $d = 1000$ as the value with least outliers and relatively low time and memory consumption.

Apart from studying the influence of d, we performed an analysis concerning the impact of using different functions for weighting OAUE's components. The experiments involved calculating average prequential accuracies for all test datasets using a nonlinear ($w^t_{NL} = \frac{1}{MSE^t_r + MSE^t_i + \epsilon}$) and linear function ($w^t_L = \max\{MSE^t_r - MSE^t_i, \epsilon\}$). The results, presented in Table 5.3, show that both functions perform almost identically on most datasets, with the linear function performing slightly better on datasets with very frequent changes and the nonlinear function being more accurate on datasets with noise and longer periods of stability. This dependency can be explained by analyzing component weights during a sudden concept drift. A practical example of such a situation is depicted in Figure 5.2 were proportional weight values of ten components are depicted for OAUE using w^t_{NL} (a) and w^t_L (b).

Table 5.3: Comparison of OAUE with a linear (w_L^t) and nonlinear (w_{NL}^t) weighting function in terms of average prequential accuracy [%], average memory usage [MB], and average testing time per $d = 1000$ examples [s]

	OAUE with w_L^t			OAUE with w_{NL}^t		
	Acc.	Mem.	Test.	Acc.	Mem.	Test.
`Airlines`	59.48	148.25	16.93	67.02	89.90	4.22
`CovType`	91.11	3.03	0.35	90.98	3.09	0.40
`Hyper`$_F$	90.44	3.23	0.24	90.43	3.23	0.22
`Hyper`$_S$	88.96	2.87	0.50	88.95	2.87	0.20
`LED`$_M$	53.41	0.32	0.13	53.40	0.32	0.15
`LED`$_{ND}$	51.47	0.32	0.39	51.48	0.32	0.15
`PAKDD`	80.14	4.98	1.39	80.23	3.39	1.01
`Poker`	92.34	3.62	0.56	88.89	2.18	0.18
`Power`	15.93	0.18	0.13	15.93	0.18	0.13
`RBF`$_B$	97.88	8.03	0.50	97.87	8.03	0.59
`RBF`$_{GR}$	97.43	11.06	0.65	97.42	11.06	0.70
`SEA`$_G$	88.90	1.45	0.09	88.83	1.45	0.09
`SEA`$_S$	89.40	2.66	0.14	89.33	2.66	0.14
`Tree`$_{SR}$	49.68	2.28	0.73	46.04	2.28	0.61
`Wave`	85.52	50.63	7.16	85.50	50.63	2.97
`Wave`$_M$	85.00	12.28	0.87	84.90	12.29	1.05

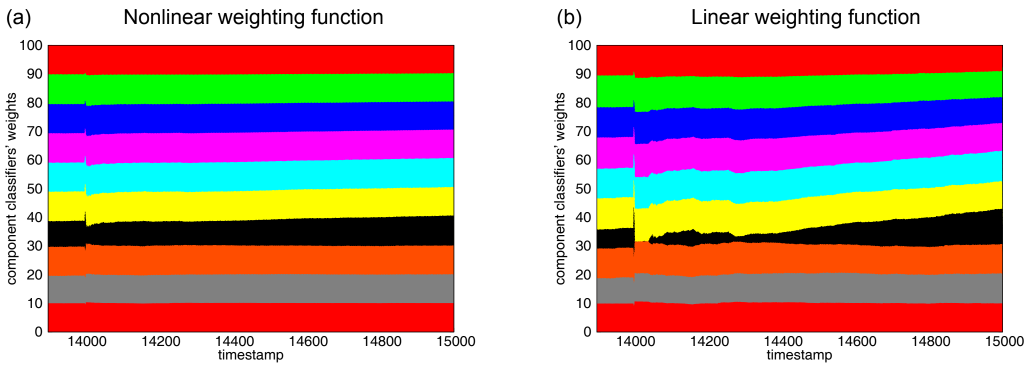


Figure 5.2: Percentage component weight proportions of OAUE with a nonlinear weighting function (a) and OAUE with a linear weighting function (b)

As Figure 5.2 shows, compared with the nonlinear function, the linear function introduces larger proportional weight changes after each example. This can have the effect of faster reactions to changes, but means that using w_L^t the algorithm is more sensitive to noise. We performed a Wilcoxon signed rank test to compare the accuracy of OAUE using w_{NL}^t and w_L^t. For $\alpha = 0.05$, we were unable to reject the null-hypothesis for any of the measured performance metrics, therefore, we cannot state any significant differences in using w_{NL}^t or w_L^t. In the comparative study presented below, OAUE is tested with the nonlinear function, as presented in Section 5.2. However, it is worth noting that using the presented linear function would yield a practically identical comparison, with identical ranks for the Friedman test presented in Table 5.7.

5.3.3 Comparison of OAUE and Other Ensembles

To evaluate the OAUE algorithm, we performed an experimental comparison involving four online ensembles:

- Online Bagging with an ADWIN change detector (Bag),
- Leveraging Bagging (Lev),
- Dynamic Weighted Majority (DWM),
- and the Adaptive Classifier Ensemble (ACE).

OAUE was implemented for this study, the source codes of the Adaptive Classifier Ensemble and Learn++.NSE were provided courtesy of Dr. Kyosuke Nishida and Dr. Paulo Gonçalves respectively, while the remaining classifiers were already a part of MOA.

Online Bagging and Leveraging Bagging were chosen as strong representatives of online ensembles, DWM was selected because it periodically evaluates an ensemble and incrementally changes component weights, and ACE represents a processing scheme with a drift detector similar to the third of the proposed modification strategies. It is worth pointing out that ACE is an algorithm that was used as a separate library and not originally written for the MOA framework. This means that ACE used different base classes and its time and memory usage measured by MOA are not fully comparable with the remaining algorithms. Additionally, on the `Wave`, `Wave`$_M$, and `PAKDD` datasets which contain a large number of attributes, ACE exceeded available memory and was unable to process the entire stream, while on other datasets ACE showcased very low memory usage. This confirms that the memory usage calculated by MOA for ACE was underestimated and, therefore, we do not present memory usage of ACE. Average prequential accuracy, memory usage, and processing time for all algorithms are given in Tables 5.4–5.6.

Table 5.4: Average prequential classification accuracies [%]

	ACE	DWM	Lev	Bag	OAUE
`Airlines`	64.89	64.98	62.84	64.24	**67.02**
`CovType`	69.60	89.87	**92.11**	88.84	90.98
`Hyper`$_F$	84.28	89.94	88.49	89.54	**90.43**
`Hyper`$_S$	79.59	88.48	85.43	88.35	**88.95**
`LED`$_M$	46.75	53.34	51.31	53.33	**53.40**
`LED`$_{ND}$	39.88	51.48	49.98	**51.50**	51.48
`PAKDD`	-	**80.24**	79.85	80.22	80.23
`Poker`	79.83	91.29	**97.67**	76.92	88.89
`Power`	**18.57**	15.45	16.84	15.96	15.93
`RBF`$_B$	84.62	96.00	**98.22**	97.87	97.87
`RBF`$_{GR}$	83.78	95.49	**97.79**	97.54	97.42
`SEA`$_G$	85.91	88.39	**89.00**	88.36	88.83
`SEA`$_S$	86.00	89.15	89.26	88.94	**89.33**
`Tree`$_{SR}$	43.20	42.48	47.88	**48.77**	46.04
`Wave`	-	84.02	83.99	**85.51**	85.50
`Wave`$_M$	-	83.76	83.46	**84.95**	84.90

Table 5.5: Average memory usage [MB]

	ACE	DWM	Lev	Bag	OAUE
`Airlines`	-	86.26	63.73	**60.27**	89.90
`CovType`	-	8.27	6.75	**1.19**	3.09
`Hyper`$_F$	-	4.37	63.41	7.19	**3.23**
`Hyper`$_S$	-	4.24	110.11	6.31	**2.87**
`LED`$_M$	-	0.61	1.76	2.56	**0.32**
`LED`$_{ND}$	-	0.41	0.98	1.32	**0.32**
`PAKDD`	-	**3.16**	38.26	6.24	3.39
`Poker`	-	2.25	4.62	**0.17**	2.18
`Power`	-	**0.09**	0.11	0.11	0.18
`RBF`$_B$	-	**6.36**	60.21	13.07	8.03
`RBF`$_{GR}$	-	**6.22**	52.94	13.15	11.06
`SEA`$_G$	-	1.73	31.05	4.06	**1.45**
`SEA`$_S$	-	**1.32**	67.33	7.32	2.66
`Tree`$_{SR}$	-	1.81	3.75	**1.10**	2.28
`Wave`	-	**6.18**	480.29	69.71	50.63
`Wave`$_M$	-	**6.42**	190.03	26.16	12.29

Table 5.6: Algorithm testing time per $d = 1000$ examples [s]

	ACE	DWM	Lev	Bag	OAUE
`Airlines`	**0.04**	2.50	11.20	2.64	4.22
`CovType`	**0.22**	0.49	2.84	0.35	0.40
`Hyper`$_F$	0.25	**0.21**	3.90	0.89	0.22
`Hyper`$_S$	0.26	**0.20**	9.16	0.38	**0.20**
`LED`$_M$	**0.09**	0.14	0.75	0.21	0.15
`LED`$_{ND}$	**0.08**	0.16	0.27	0.18	0.15
`PAKDD`	-	**0.28**	9.83	0.85	1.01
`Poker`	**0.03**	0.10	1.29	0.06	0.18
`Power`	0.19	**0.11**	0.24	0.15	0.13
`RBF`$_B$	0.62	**0.30**	11.36	0.75	0.59
`RBF`$_{GR}$	0.61	**0.31**	7.79	0.86	0.70
`SEA`$_G$	**0.05**	0.08	5.97	0.23	0.09
`SEA`$_S$	**0.04**	0.07	7.94	0.48	0.14
`Tree`$_{SR}$	0.25	**0.17**	0.62	**0.17**	0.61
`Wave`	-	**0.48**	33.65	5.04	2.97
`Wave`$_M$	-	**0.46**	33.46	1.63	1.05

As in previous chapters, we also generated graphical plots for each dataset depicting the algorithms' performance in time. We will analyze the most interesting accuracy and memory plots, which highlight characteristic features of the studied algorithms.

Figures 5.3 and 5.4 present the prequential accuracy and memory usage of the analyzed algorithms on the $\texttt{Hyper}_F$ dataset. The two best performing algorithms for this dataset containing fast incremental drift are OAUE and DWM, which seem to react better to incremental changes than ACE, Lev, and Bag. The characteristic feature of DWM and OAUE is that they periodically change ensemble members, while the remaining three algorithms do that only when drift is detected. Similar behavior was observed in accuracy plots for the $\texttt{Hyper}_S$ dataset, which contains a continuous, slow, incremental drift.

Looking at the memory plot in Figure 5.4, we can see that Lev requires much more memory than the remaining algorithms, Bag is second, while OAUE and DWM are the least expensive in terms of memory. This observation was consistent among most memory plots. It is also worth noticing that OAUE is one of the fastest of the analyzed algorithms and, in contrast to the analyzed online evaluation strategy from Section 4.2, does not introduce any additional processing costs compared to its block-based predecessor AUE.

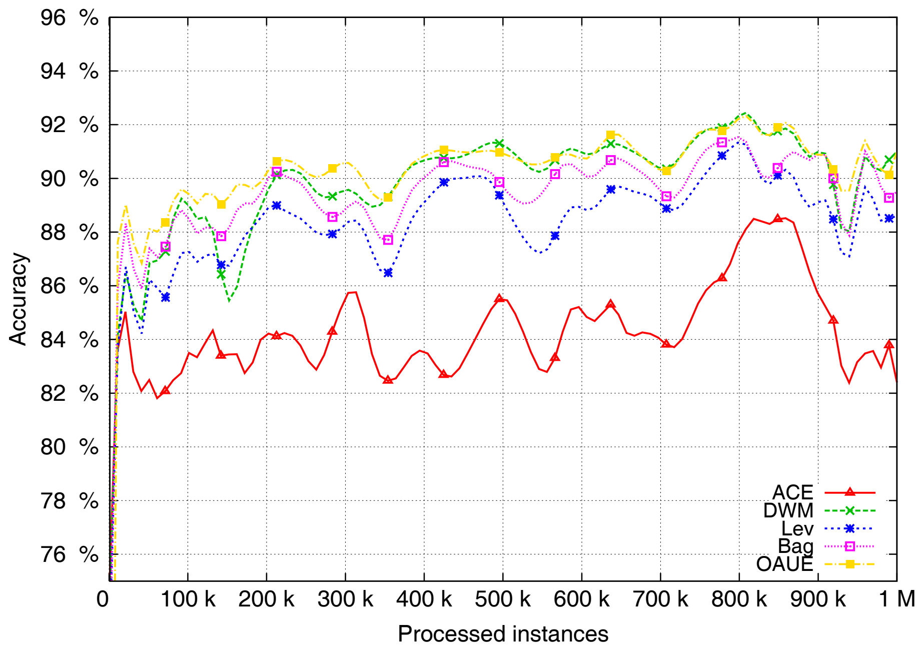


Figure 5.3: Prequential accuracy on the $\texttt{Hyper}_F$ dataset

For streams with gradual drifts ($\texttt{RBF}_{GR}$ and $\texttt{SEA}_G$) the best performing algorithm is Lev, with OAUE and Bag being close second. However, Lev is also the slowest and most memory consuming algorithm on these datasets, requiring on an average 13 times more memory and 38 times more processing time than OAUE. Figure 5.5 presents the accuracies of the analyzed algorithms on the $\texttt{RBF}_{GR}$ dataset. Gradual drifts created around examples number 125, 250, 375, 500 k have the worst impact on DWM and ACE. ACE uses static batch learners and, therefore, is not capable of reacting sufficiently quickly to changes. DWM on the other hand is probably performing slightly worse because it uses a penalty function which strongly diminishes component weights during prolonged drifts.

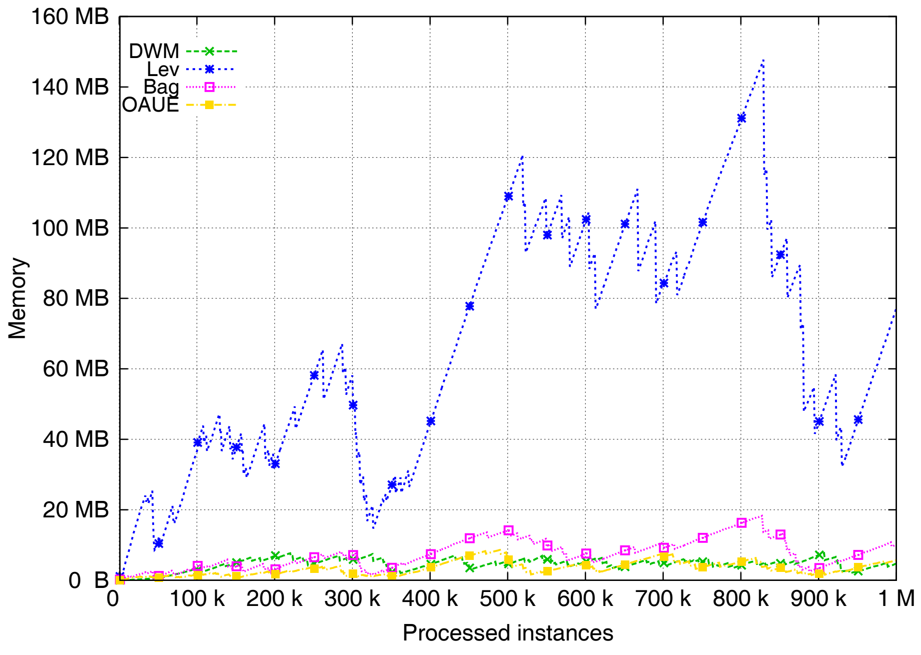


Figure 5.4: Memory usage on the $\texttt{Hyper}_F$ dataset

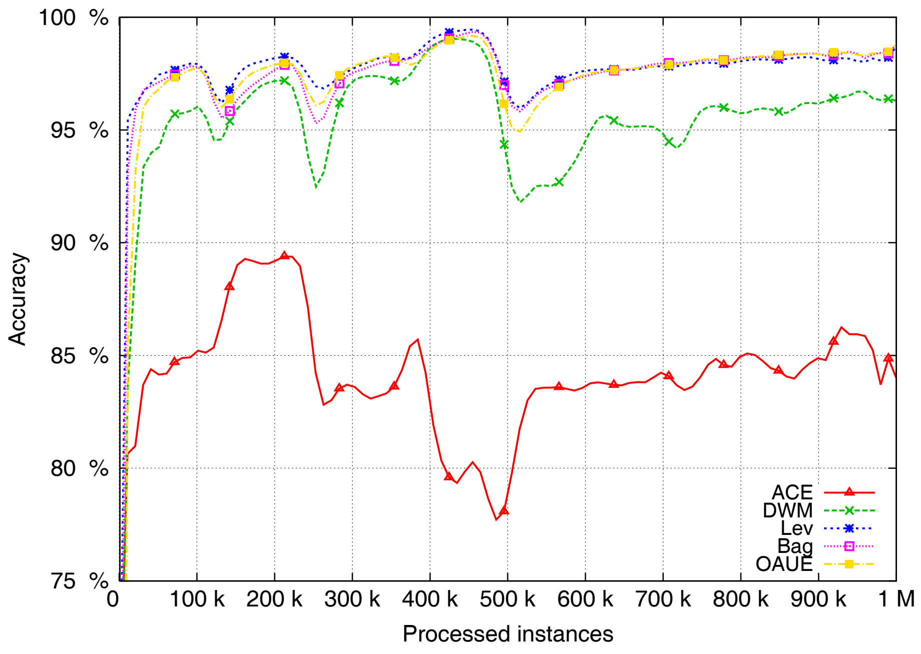


Figure 5.5: Prequential accuracy on the $\texttt{RBF}_{GR}$ dataset

Depending on the frequency of drifts, the best performing algorithms for streams with sudden changes were OAUE and Bag. For rare abrupt changes, such as in the $\texttt{SEA}_S$ dataset, OAUE suffered smallest accuracy drops. However, for very fast changes present in the $\texttt{Tree}_{SR}$ stream, algorithms with drift detectors, such as Lev and Bag, were more accurate. What is worth noting is that using OAUE with a linear weighting function (as presented in Section 5.3.2) would yield an accuracy of 49.68%, which would be the best result for this dataset. This shows that for very dynamic changes either drift detectors or very drastic component weight modifications are required to adapt in time.

On datasets with no drift ($\texttt{LED}_{ND}$), with drifting attribute values (`Wave`, $\texttt{Wave}_M$) or added noise ($\texttt{LED}_M$, $\texttt{RBF}_B$), OAUE and Bag perform almost identically, with Lev, DWM, and ACE being slightly less accurate. All algorithms react similarly to changes introduced in these datasets, and differences concern only the predictive performance of each ensemble.

Concerning real datasets, there is no single best performing algorithm in terms of accuracy. On `Poker`, Lev clearly outperforms all the other algorithms. On `CovType`, Lev is the most accurate followed by OAUE, while on `PAKDD` all the algorithms perform almost identically. On the other hand, OAUE is the most accurate on the `Airlines` dataset, while ACE is the best on `Power`.

To conclude the analysis, we carried out statistical tests for comparing multiple classifiers over multiple datasets. As in previous experiments, we used the non-parametric Friedman test coupled with the Bonferroni-Dunn post-hoc test [45, 82] to verify whether the performance of OAUE is statistically different from the remaining algorithms. The average ranks of the analyzed algorithms are presented in Table 5.7 (the lower the rank the better).

Table 5.7: Average algorithm ranks used in the Friedman tests

	ACE	DWM	Lev	Bag	OAUE
Accuracy	4.50	2.94	2.75	2.75	**2.06**
Memory	-	**1.81**	3.56	2.63	2.00
Testing time	2.50	**1.81**	4.81	3.19	2.69

By using the Friedman test to verify the differences between accuracies, we obtain $F_{F_{Acc}} = 7.248$. As the critical value for comparing 5 algorithms over 16 datasets for $\alpha = 0.05$ is 2.525, the null hypothesis can be rejected. Considering accuracies, OAUE provides the best average achieving usually a high rank on each dataset. To verify whether OAUE performs better than the remaining algorithms, we compute the critical difference chosen by the Bonferroni-Dunn test as $CD = 1.396$. This allows us to state that OAUE is significantly more accurate then ACE, but concerning the remaining algorithms the experimental data is not sufficient to reach such a conclusion. However, by performing additional one-tailed Wilcoxon signed rank tests for comparing pairs of classifiers, we can state that OAUE is more accurate than DWM with $p_{DWM} = 0.004$. The p-value for stating that OAUE is more accurate than Bag and Lev are $p_{Bag} = 0.089$ and $p_{Lev} = 0.163$ respectively. Overall, the conducted experiments seem to support our observation that in terms of accuracy OAUE is not only comparable to other systems in the literature, but in most cases achieves better performance.

Performing a similar analysis for memory usage and processing time we get $F_{F_{Mem}} = 6.793$ and $F_{F_{Time}} = 15.476$ respectively, which allows us to reject the null-hypothesis in both cases. Analyzing the CD and by performing Wilcoxon tests we can state that OAUE is significantly faster than Lev ($p_{Lev} = 0.0002$) and less memory consuming than Lev and Bag ($p_{Lev} = 0.001$, $p_{Bag} = 0.022$).

5.4 Conclusions

In this chapter, we combined the main results of Chapter 4 and proposed a new incremental stream classifier, called Online Accuracy Updated Ensemble (OAUE), which trains and weights component classifiers with each incoming example. The main novelty of the OAUE algorithm is the proposal of a cost-effective component weighting function, which estimates a classifier's error on a window of last seen instances in constant time and memory without the need of remembering past examples.

We also carried out experimental studies analyzing the effect of using different window sizes and functions for evaluating component classifiers. The obtained results showed that the accuracy of the proposed algorithm did not change depending on the window size, but larger windows induced higher time and memory costs. Concerning different error-based weighting functions, we have found that a linear function performed better on fast drifting streams, but a nonlinear function was more robust to noise.

Finally, we experimentally compared OAUE with four representative online ensembles: the Adaptive Classifier Ensemble, Dynamic Weighted Majority, Online Bagging, and Leveraging Bagging. The obtained results demonstrated that OAUE can offer high classification accuracy in online environments regardless of the existence or type of drift. OAUE provided best average classification accuracy out of all the tested algorithms and was among the least time and memory consuming ones.

The AUE algorithm, presented in Chapter 3, was designed to add elements of online learning to block-based ensembles. However, in environments where labels are available after each instance, these elements of online learning do not suffice if the ensemble still processes data in blocks. The OAUE algorithm aims at retaining the positive elements of AUE, while adding the capability of processing streams online. The fact that this capability was achieved with negligible overhead in terms of memory usage and processing time, makes OAUE a much better choice for streams with online labeling.

Chapter 6

Classifier Evaluation Methods for Imbalanced Streams with Class Distribution Changes

Thus far, the conducted analysis has concentrated on the performance of block-based and online ensembles in the presence of real drifts. However, as mentioned in Section 2.2, several types of changes fall into the category of virtual drift, i.e., changes in the data distribution $p(\mathbf{x})$ or $p(y)$ that do not necessarily affect class conditional probabilities $p(y|\mathbf{x})$. Although less frequently studied than real drifts in the context of data streams, virtual drifts resemble difficulties that have been more intensively analyzed for batch data. For example, the issue of class distribution changes between classifier training and performance has been investigated in the context of model selection [169, 82]. Furthermore, the problem of time-evolving *class imbalance*, i.e., the underrepresentation of certain classes, can also be considered a special type of virtual drift. In traditional methods for mining static data, the issues of class imbalance and class distribution changes have been studied due to complexities arising not only in classifier training but also evaluation [76]. Similarly, virtual drift introduces additional difficulties to the process of training and evaluating stream classifiers.

This chapter aims at analyzing how class distribution changes, as a special case of virtual drift, affect the performance of classifiers for evolving data streams. In order to perform this evaluation, we first summarize, in Section 6.1, existing methods for assessing the predictive performance of data stream classifiers. The main goal of this review is to underline deficiencies of currently used stream evaluation measures when applied to imbalanced data. In particular, we analyze if and how the most popular class distribution-independent measure for batch imbalanced data, the area under the ROC curve (AUC), is used for evaluating streaming classifiers. The resulting literature study indicates that traditional AUC computation methods are inapplicable to streaming data and the use of this measure for assessing adaptive classifiers remains an open challenge. Therefore, in Section 6.2, we put forward a novel online algorithm for calculating the area under the ROC curve with a forgetting mechanism, called prequential AUC. Section 6.3 analyzes the properties of this newly proposed evaluation measure, in particular, its consistency with traditional AUC in the context of stationary and drifting streams. Furthermore, in Section 6.4 we present experimental results, which demonstrate the time performance of prequential AUC and its applicability to drift detection for balanced and imbalanced data streams. Additionally, we use prequential AUC to assess the performance of online classifiers in the presence of class distribution changes. Finally, Section 6.5 concludes the chapter.

6.1 Classifier Evaluation Methods in the Context of Concept Drift

In this section, we discuss evaluation measures and error-estimation procedures used in data stream classification. Section 6.1.1, covers measures currently used to assess predictive performance and highlights their properties in the context of class distribution changes. Section 6.1.2 discusses error-estimation and randomization procedures, which try to take into account the existence of concept drift as well as problems of limited time and memory, during the calculation of evaluation measures.

6.1.1 Evaluation Measures

When evaluating the performance of a classifier for concept-drifting data streams, two factors are crucial: *prediction accuracy* and *the ability to adapt*.

The first factor could be analyzed by a simple empirical *error-rate*, i.e. the fraction of misclassified examples, or its complement, the fraction of correctly classified examples. This measure is referred to as *accuracy*, and is one of the most commonly used evaluation metrics in batch and data stream classification. Accuracy and error-rate effectively summarize the overall performance of a classifier in a single scalar metric, taking into account all data classes. Consequently, they can be efficiently calculated using all of the error-estimation and randomization techniques, which will be discussed in Section 6.1.2. Moreover, accuracy and error-rate can be used to assess simple discrete classifiers, as well as more complex scoring or probabilistic classifiers.

However, as a result of trying to combine knowledge about all data classes in a single metric, there are limitations concerning the encompassed information as well as the effectiveness of these measures in different scenarios [82]. Error-rate and accuracy do not convey information on the importance of the performance of different classes. Moreover, in case of skewed distributions these measures promote majority class predictions and effectively hide algorithm weaknesses concerning minority class examples. In other words, accuracy and error-rate are effective measures when the number of instances belonging to particular classes is more or less balanced. This last issue, limits the use of accuracy or error-rate on imbalanced or class distribution drifting streams.

In traditional, batch and stationary, data mining, several alternatives for evaluating classifiers on imbalanced data have been proposed [76, 82]. Most of these measures, such as the G-mean, sensitivity, specificity, precision, recall, or F-score, are calculated by aggregating correct and incorrect predictions, similarly to accuracy. However, the aforementioned metrics combine different parts of the confusion matrix, i.e., different types of classifier prediction errors [82]. Although these measures have computational requirements comparable to accuracy, they are still rarely used in evaluations of data stream classifiers. However, two other measures for imbalanced data are slowly gaining more attention in data stream processing — the κ statistic and AUC.

Bifet and Frank [12] proposed the use of the Cohen's κ to assess the predictive abilities of a classifier on imbalanced data streams. Cohen's κ is a statistic that measures the ratio of the difference between the observed and chance agreement that can be achieved over and beyond chance [82]. In the context of classification, this involves comparing the accuracy of the tested classifier with that of a chance classifier:

$$\kappa = \frac{p_0 - p_c}{1 - p_c} \tag{6.1}$$

where p_0 is the accuracy of the tested classifier and p_c is the accuracy of a chance classifier. The accuracy of a chance classifier is determined by the probability of giving a correct prediction by chance, which in turn is dependent on the current class distribution [82]. The κ statistic can achieve values from $(-\infty; 1]$, where zero means that p_0 is no better than a chance classifier and values above/below zero indicate how much better/worse p_0 performs compared to p_c. Additionally, in the context of data stream classification, this metric has been recently extended to take into account temporal dependence [22, 178]. The κ statistic, and other agreement measures, often yield more realistic estimations of predictive performance, as they take into account the marginal probability of label assignments to correct the estimated accuracy for chance. However, they are still sensitive to issues such as class imbalance and misclassification costs [82].

Several data stream researchers have also tried to use the area under the ROC (Receiver Operator Characteristic) curve, which is one of the most popular evaluation metrics for batch imbalanced data. ROC analysis investigates the relationship between the true and false positive rate of a binary classifier, for different decision thresholds [58]. If we denote examples of two classes distinguished by a binary classifier as *positive* and *negative*, the ROC curve is created by plotting the proportion of positives correctly classified (*true positive rate*) against the proportion of negatives incorrectly classified (*false positive rate*). If a classifier outputs a score proportional to its belief that an instance belongs to the positive class, decreasing the classifier's decision threshold (above which an instance is deemed to belong to the positive class) will increase both true and false positive rates. Varying the decision threshold results in a piecewise linear curve, called the ROC curve, which is presented in Figure 6.1. The area under this ROC curve, abbreviated as AUC, summarizes the plotted relationship in a single scalar metric and is one of the most popular evaluation metrics in data mining [82].

The popularity of AUC for batch imbalanced data, stems from the fact that it is invariant to changes in class distribution. Moreover, for scoring classifiers it has a very useful statistical interpretation as the expectation that a randomly drawn positive example receives a higher score than a random negative example [169]. Additionally, several

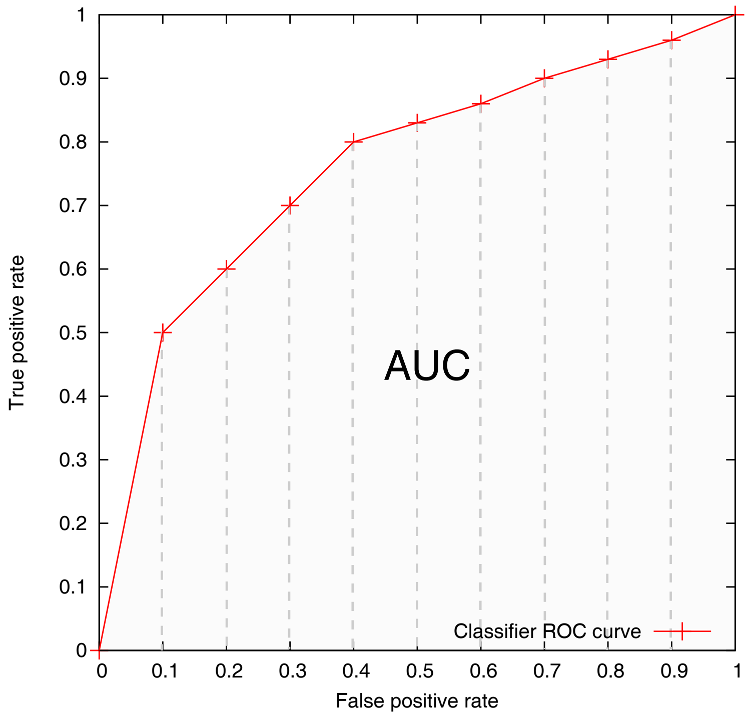


Figure 6.1: Example ROC curve

authors have shown that for stationary data AUC is more preferable for classifier evaluation than total accuracy [79]. However, the calculation of AUC is costly, as it requires that all examples be sorted according to scores assigned by the evaluated classifier. Moreover, after sorting, the calculation algorithm has to re-iterate through all the examples to sum the area of trapezoids for each pair of sequential points on the ROC curve. Therefore, researchers tried to compute AUC only once using entire streams [48, 110] or periodically using holdout sets [162, 78]. Nevertheless, it was noticed that periodical holdout sets may not fully capture the temporal dimension of the data, whereas evaluation using entire streams is neither feasible for large datasets nor suitable for drift detection. It is also worth mentioning that an algorithm for computing AUC incrementally has also been proposed [25], yet one which calculates AUC from all available examples and is not applicable to evolving data streams. Although the cited works show that AUC is recognized as a measure which should be used to evaluate classifiers for imbalanced data streams, up till now it has been computed the same way as for static data and its applicability to large drifting streams remains an open problem.

To evaluate the second crucial factor, *the ability to adapt*, separate methods are needed. Some researchers evaluate the classifier's adaptability by comparing drift reaction times [72]. This is done by measuring the time between the start of a drift and the moment when the tested classifier's accuracy recovers to level from before the drift. It is important to notice that in order to calculate reaction times, usually a human expert needs to determine moments when a drift starts and when a classifier recovers from it.

To automate the assessment of adaptability, Shaker and Hullermeier [150, 151] proposed an approach, called *recovery analysis*, which uses synthetic datasets to calculate classifier reaction times. The authors propose to divide a dataset with a single drift into two sets without drifts. Afterwards, they propose to plot the accuracy of the tested classifier on each of these datasets separately. The combination of these two plots is called the *optimal performance curve* and serves as a reference that can be compared with the accuracy plot of the classifier on the original dataset. Apart from a graphical comparison, the authors propose to calculate two metrics: the relative recovery duration and the maximum performance loss. The duration of recovery is defined as:

$$duration = \frac{t_2 - t_1}{T} \tag{6.2}$$

where t_1 is the time at which the performance of the classifier drops below 95% of its optimal performance curve before the drift, t_2 the time at which the classifier recovers to 95% of its optimal performance curve after the drift, and T the length of the entire stream. The maximum performance loss measures the maximum drop in accuracy as:

$$maxLoss = \max_{t \in T} \frac{min\{S_A(t), S_B(t)\} - S_C(t)}{min\{S_A(t), S_B(t)\}} \tag{6.3}$$

where $S_A(t)$ and $S_B(t)$ are optimal performance curves of the tested classifier before and after the drift respectively, and $S_C(t)$ is the performace curve of the classifier on the original dataset. Recovery analysis, bypasses the problem of hand labeling of the drift reaction period, but requires external knowledge about drifts in real streams or the use of synthetic datasets and, therefore, can only be used offline.

Finally, some researchers have put forward measures that take into account the cost of classifier adaptation. Brzezinski and Stefanowski [31] proposed to differentiate the importance of predictions made directly after the appearance of a concept drift and predictions during periods of stability. This is done by applying a user-defined weight to predictions during periods after a detected drift. Such an approach is inspired by cost-based learning for imbalanced datasets, where errors made on a minority class example cost more than errors made on examples from the majority class. In this approach, the authors treat examples directly after a detected drift as “minority” examples and assign a higher weight. With consecutive examples, the new concept slowly becomes the “majority” and the weight of examples converges back to a default value. There are several example weighting functions that fulfill these requirements, but as a practical example the authors proposed a logarithmic function defined as follows:

$$w(t) = max(-log_{e^{\frac{1}{w_{avg}-1}}} t + 1 + log_{e^{\frac{1}{w_{avg}-1}}} d, 1), \tag{6.4}$$

where t is the number examples after the detected drift, d is the average time between drifts, and w_{avg} is the average weight of examples during the d period. The w_{avg} parameter defines how much more important should predictions directly after a drift be compared to predictions in times of stability. The proposed measure can combine information about accuracy and adaptability in a single metric, but through the w_{avg} parameter assumes that the user can estimate the cost of not reacting to changes and makes the user responsible for parametrizing the evaluation.

A more business-oriented approach to adaptation costs was put forward by Žliobaitė et al. [179]. The authors propose to treat potential model updates as an investment decision, which should be based on performance gains in relation to computational costs. This gain-to-cost ratio can be used to resolve the practical trade-off between using accurate but computationally expensive models and not that accurate but cheap models. As a result, the authors propose practical assessment measures which can be used for offline and online model selection. More precisely they propose to calculate the return of interest (ROI) for quantifying the gain in performance per resources (RAM-hours) invested:

$$ROI = \frac{\gamma}{\varphi} \tag{6.5}$$

where γ is the absolute gain of adapting compared to not adapting to change and φ is overall cost of adaptation. Depending whether ROI is calculated on a single drift, averaged over several drifts, or computed incrementally, it can be used to analyze single drifts and select models offline or online. In terms of class distribution changes, ROI metrics have been used with error-rates and, therefore, inherit limitations concerning skewed and imbalanced data.

6.1.2 Error-estimation Procedures

Just as in traditional machine learning, in data streams the discussed evaluation measures are estimated on labeled testing examples. However, contrary to batch data scenarios, it is assumed that due to the size and speed of data streams repeated runs over the data are not necessary. In fact, due to their computational costs, resampling techniques such as cross-validation or bootstrapping are deemed too expensive to be worth applying in streaming scenarios [87]. Therefore, simpler error-estimation procedures have to be used, yet ones that enable to build a picture of *performance over time* or ensure *robustness* to concept drift.

When batch learning is performed on datasets involving too many examples to employ resampling, it is often accepted to measure performance on a single *holdout set*. This set is separated from the training instances, to ensure that the calculated measures indicate the generalization performance of the classifier [82]. Viewing data streams as large datasets, it then follows from batch learning that using holdout sets is appropriate. To track performance over time, the classifier can be evaluated periodically, for example, after every 50,000 examples. However, using a single holdout set is suitable only for data streams without concept drift, as each new concept requires appropriate testing examples.

An alternate scheme of estimating the performance of data stream classifiers involves interleaving testing with training [87]. Each individual example is first used to test the classifier before it is used for training. This evaluation procedure, often called *interleaved test-then-train*, has the advantage that it makes maximum use of the available data. Furthermore, when depicted over time, performance measures obtained using this procedure result in smoother and more detailed plots. However, the disadvantages of this approach are that it can only be used in online processing and it is difficult to separately measure training and testing times. With this issue in mind, some researchers propose to assess performance using *interleaved chunks* [29, 30], where, instead of single examples, larger blocks of examples are used for testing and training. The advantage of using interleaved chunks is that this procedure can be used in block processing environments and forms a compromise between testing on single examples or a holdout set.

More recently, Gama et al. [69] proposed a *prequential* procedure, which is similar to the test-then-train technique, but involves forgetting old examples. More precisely, prequential calculations use a sliding window, which limit the number of examples used during evaluation. When the window is full and a new example arrives, the oldest example is removed from the window. Such a procedure highlights the current rather than overall performance and as a result showcases changes in the stream more clearly, which is especially important for drift detection. Moreover, the authors have shown that computing accuracy using this method is more appropriate for continuous assessment and drift detection in evolving data streams than a holdout set or the interleaved test-then-train method. In terms of prequential accuracy, the authors have additionally shown that it can be computed either using a sliding window or a decay function. Nevertheless, prequential accuracy inherits the weaknesses of traditional accuracy, that is, variance with respect to class distribution and promoting majority class predictions. It is also worth noting that, despite several advantages of the prequential procedure, apart from accuracy and the κ statistic, other evaluation measures are not calculated prequentially on streams due to computational difficulties.

In terms of testing robustness, an interesting randomization technique involves permutating the test dataset several times to ensure measured drift reactions are not sensitive to the order of examples [176]. The author put forward three controlled permutation techniques that can create several variations of a given dataset, changing the time, speed, or shape of drifts. The key idea lies in the fact that these permutations are restricted and provide theoretical guarantees for preserving distributions. Therefore, they ensure that the new sets represent close variations of the original learning task. The drawback of this randomization technique is that it requires generating artificial datasets and, thus, is limited to offline use during model selection, rather than on deployed models working online on streams.

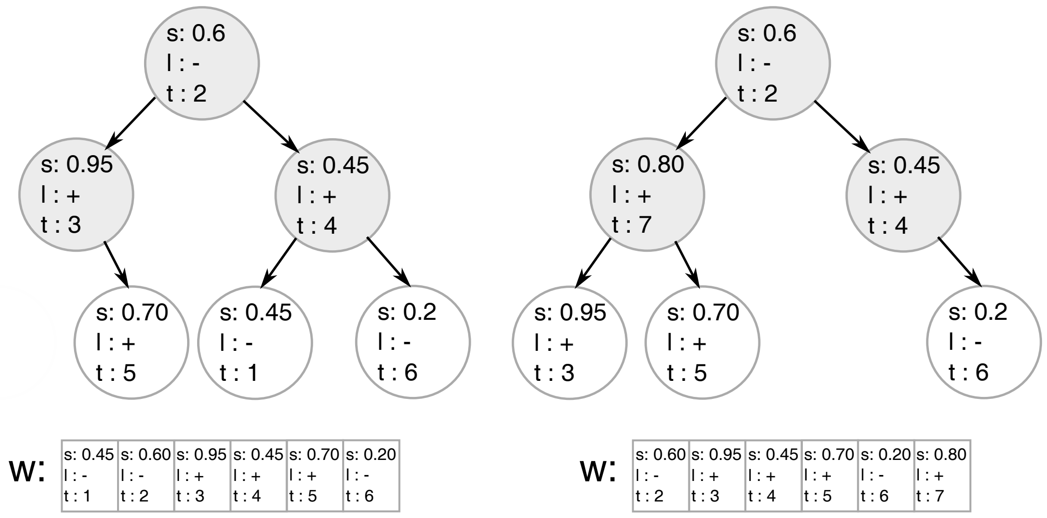
6.2 Prequential Area Under the ROC Curve

All of the measures presented in the previous section have drawbacks concerning the evaluation of data streams with class distribution changes. Total accuracy promotes majority class predictions and, thus, overestimates performance on imbalanced data. The κ statistic is also sensitive to skewed distributions. Finally, the area under the Receiver Operator Characteristic curve (AUC), has been used only on holdout sets or entire datasets, both of which are unacceptable for evolving data streams.

However, AUC is one of the main evaluation measures in traditional batch data mining, especially in the context of class imbalanced data. It is invariant to changes in class distribution, has a useful statistical interpretation, and has been shown to be preferable to accuracy in many scenarios. Therefore, the disadvantages of AUC as an evaluation measure stem from the fact that currently there is no memory and computationally efficient method for calculating this metric [33]. In this section we will tackle this open problem and verify if it is possible to efficiently adapt AUC to classifier evaluation on drifting data streams.

As AUC is calculated on ranked examples, we will consider *scoring classifiers*, i.e., classifiers that for each predicted class label additionally return a numeric value (*score*) indicating the extent to which an instance is predicted to be positive or negative. Furthermore, we will limit our analysis to binary classification. It is worth mentioning, that most classifiers can produce scores, and many of those that only predict class labels can be converted to scoring classifiers. For example, decision trees can produce class-membership probabilities by using Naive Bayes leaves or averaging predictions using bagging [140]. Similarly, rule-based classifiers can be modified to produce instance scores indicating the likelihood that an instance belongs to a given class [55].

We propose to compute AUC incrementally after each example using a special sorted structure combined with a sliding window forgetting mechanism. It is worth noting that, since the calculation of AUC requires sorting examples with respect to their classification scores, it cannot be computed on an entire stream or using fading factors without remembering the entire stream. Therefore, for AUC to be computationally feasible and applicable to evolving concepts, it must be calculated using a sliding window.


s: 0.6 l : - t : 2
s: 0.95 l : + t : 3
s: 0.45 l : + t : 4
s: 0.70 l : + t : 5
s: 0.45 l : - t : 1
s: 0.2 l : - t : 6
w:
s: 0.45 l : - t : 1 | s: 0.60 l : - t : 2 | s: 0.95 l : + t : 3 | s: 0.45 l : + t : 4 | s: 0.70 l : + t : 5 | s: 0.20 l : - t : 6
s: 0.6 l : - t : 2
s: 0.80 l : + t : 7
s: 0.45 l : + t : 4
s: 0.95 l : + t : 3
s: 0.70 l : + t : 5
s: 0.2 l : - t : 6
w:
s: 0.60 l : - t : 2 | s: 0.95 l : + t : 3 | s: 0.45 l : + t : 4 | s: 0.70 l : + t : 5 | s: 0.20 l : - t : 6 | s: 0.80 l : + t : 7

(a) Before adding a new instance (b) After adding a new instance

Figure 6.2: Example red-black tree of scored examples from window w, where l is the example's true label, s its assigned score, and t its timestamp

A sliding window of scores limits the analysis to the most recent data, but to calculate AUC, scores have to be sorted. To efficiently maintain a sorted set of scores, we propose to use the *red-black tree* data structure [7]. A red-black tree is a self-balancing binary search tree, which is capable of adding and removing elements extremely efficiently (in logarithmic time), while requiring minimal memory. An example red-black tree with a corresponding window of examples is presented in Figure 6.2. With these two structures, we can efficiently calculate AUC prequentially on the most recent examples. Algorithm 6.1 lists the pseudo-code for calculating prequential AUC.

Algorithm 6.1 Prequential AUC

Input: $\mathcal{S}$: stream of examples
d: window size
Output: $\hat{\theta}$: prequential AUC after each example

```
1:  $W \leftarrow \emptyset$; $n \leftarrow 0$; $p \leftarrow 0$; $idx \leftarrow 0$;
2:  for all scored examples $\mathbf{x}^t \in \mathcal{S}$ do
3:    // Remove oldest score from the window and tree
4:    if $idx \geq d$ then
5:      $scoreTree.remove(W[idx \textbf{ mod } d])$;
6:      if $isPositive(W[idx \textbf{ mod } d])$ then
7:        $p \leftarrow p - 1$;
8:      else
9:        $n \leftarrow n - 1$;
10:     end if
11:   end if
12:   // Add new score to the window and tree
13:   $scoreTree.add(\mathbf{x}^t)$;
14:   if $isPositive(\mathbf{x}^t)$ then
15:     $p \leftarrow p + 1$;
16:   else
17:     $n \leftarrow n + 1$;
18:   end if
19:   $W[idx \textbf{ mod } d] \leftarrow \mathbf{x}^t$;
20:   $idx \leftarrow idx + 1$;
21:   // Calculate AUC [169]
22:   $AUC \leftarrow 0$; $c \leftarrow 0$;
23:   for all consecutive scored examples $s \in scoreTree$ do
24:     if $isPositive(s)$ then
25:       $c \leftarrow c + 1$;
26:     else
27:       $AUC \leftarrow AUC + c$;
28:     end if
29:   end for
30:   $\hat{\theta} \leftarrow \frac{AUC}{pn}$;
31: end for
```

For each incoming labeled example the score assigned to this example by the classifier is inserted into the window (line 19) as well as the red-black tree (line 13) and, if the window of examples has been exceeded, the oldest score is removed (lines 5 and 19). The red-black tree is sorted in descending order according to scores and ascending order according to the arrival time of the score. This way, we maintain a structure that facilitates the calculation of AUC and ensures that the oldest score in the sliding window will be instantly found in the red-black tree. After the sliding window and tree have been updated, AUC is calculated by summing the number of positive examples occurring before each negative example (lines 23–29) and normalizing that value by all possible pairs pn (line 30), where p is the number of positives and n is the number of negatives in the window. This method of calculating AUC, proposed in [169], is equivalent to summing the area of trapezoids for each pair of sequential points on the ROC curve, but more suitable for our purposes, as it requires very little computation given a sorted collection of scores.

An example of using a sliding window and a red-black tree is presented in Figure 6.2. Window w contains 6 examples, all of which are already inserted into the red-black tree. As mentioned earlier, examples in the tree are sorted (depth first search wise) descending according to scores s and ascending according to arrival time t. When a new instance is scored by the classifier ($t = 7, l = +, s = 0.80$), the oldest instance ($t = 1$) is removed from the window and the tree. Since the tree is sorted according to scores and arrival times, finding the example to be removed involves finding the first example with the score of the oldest example ($s = 0.45$). After the new scored example is inserted, AUC is calculated by traversing the tree in a depth first search manner and counting labels as presented in lines 22–30 of Algorithm 6.1. In this example, the resulting AUC would be 0.875.

Let us now analyze the complexity of the proposed approach. For a window of size d, the time complexity of adding and removing a score to the red-black tree is $O(2 \log d)$. Additionally, the computation of AUC requires iterating through all the scores in the tree, which is an $O(d)$ operation. In summary, the computation of prequential AUC has a complexity of $O(d + 2 \log d)$ per example and since d is a user-defined constant this resolves to a complexity of $O(1)$. It is worth noticing that if AUC only needs to be sampled every k examples (a common scenario while plotting metrics in time) lines from 22 to 30 can be executed only once per k examples. In terms of space complexity, the algorithm requires $O(2d)$ memory for the red-black tree and the window, which also resolves to $O(1)$.

In contrast to error-rate performance metrics, such as accuracy [69, 62] or the Kappa statistic [12, 178], the proposed measure is invariant of the class distribution. Furthermore, unlike total accuracy, it does not promote majority class predictions. Additionally, in contrast to the Kappa statistic, AUC is a non-relative, $[0, 1]$ normalized metric with a direct statistical interpretation. As opposed to previous applications of AUC to data streams [48, 78, 110, 162], the proposed algorithm can be executed after each example using constant time and memory. Finally, compared to the method presented in [25], the proposed algorithm provides a forgetting mechanism and uses a sorting structure, making it suitable for evolving data streams and allowing for efficient sampling.

Prequential AUC assesses the ranking abilities of a classifier and is invariant of the class distribution. These properties differentiate it from common evaluation metrics for data stream classifiers and could be applied in an additional context. In particular, for streams with high class imbalance ratios simple metrics, such as accuracy, will suggest good performance (as they are biased toward recognizing the majority class) and may poorly exhibit concept drifts. Therefore, we propose to investigate AUC not only as an evaluation measure, but also as a basis for drift detection in imbalanced streams, where it should better indicate changes concerning the minority class.

For this purpose, we modify the Page-Hinkley (PH) test [69], however, generally other drift detection methods could also have been adapted. As discussed in Section 2.3.3, the PH test considers a variable m^t, which measures the accumulated difference between observed values e (originally error estimates) and their mean till the current moment, decreased by a user-defined magnitude of allowed changes δ: $m^t = \sum_{i=1}^{t} (e^t - \bar{e}^t - \delta)$. After each observation e^t, the test checks whether the difference between the current m^t and the smallest value up to this moment $\min(m^i, i = 1, \ldots, t)$ is greater than a given threshold λ. If the difference exceeds λ, a drift is signaled. In this thesis, we propose to use the area *over* the ROC curve ($AOC = 1 - AUC$) as the observed value. Hence, according to the statistical interpretation of AUC, instead of error estimates, we monitor the estimate of the probability that a randomly chosen positive is ranked *after* a randomly chosen negative. This way, the PH test will trigger whenever a classifier begins to make severe ranking errors regardless of the class imbalance ratio.

Prequential AUC aims at extending the list of available evaluation measures, particularly for assessing classifiers and detecting drifts in streams with evolving class distributions. However, we must verify if this measure is suitable for depicting performance changes over time, particularly in the context of class imbalance. Furthermore, we are also interested how prequential AUC averaged over time relates to AUC calculated periodically on blocks or once over the entire stream. In the following sections, we examine these characteristics of prequential AUC, as well as its performance in scenarios involving different types of drift and imbalance ratios.

6.3 Properties of Prequential AUC

As it was mentioned in Section 6.1, there have already been attempts to use AUC as an evaluation measure for data stream classifiers. Some researchers [162, 78] calculated AUC on periodical holdout sets (blocks of examples), while others [48, 110], for experimental purposes, treated entire data streams as a single batch of examples and calculated AUC traditionally. Furthermore, although not studied in the context of drifting data streams, there has been a proposal of an algorithm for computing AUC incrementally instance after instance [25]. With the proposed prequential estimation, this gives in total four ways of evaluating data stream classifiers using AUC. Let us analyze how these four alternatives perform when visualized in time and averaged over entire streams.

6.3.1 AUC Visualizations Over Time

Figures 6.3 and 6.4 present a visualization of four AUC calculation procedures:

- traditional batch AUC calculated once for the entire stream,
- incremental AUC calculated after each example,
- block AUC calculated every d examples on the last d examples,
- prequential AUC using last d examples.

More precisely, both plots present the performance of a single Hoeffding Tree classifier on a dataset with 20 k examples created using the `RBF` generator, discussed in Section 3.3.1. The forst dataset contained no drifts, whereas in the second dataset a sudden drift was added at the 10 k example. Both prequential and block-based AUC were calculated using a window of $d = 1000$ examples. Similar visualizations were previously done for accuracy [87, 69].

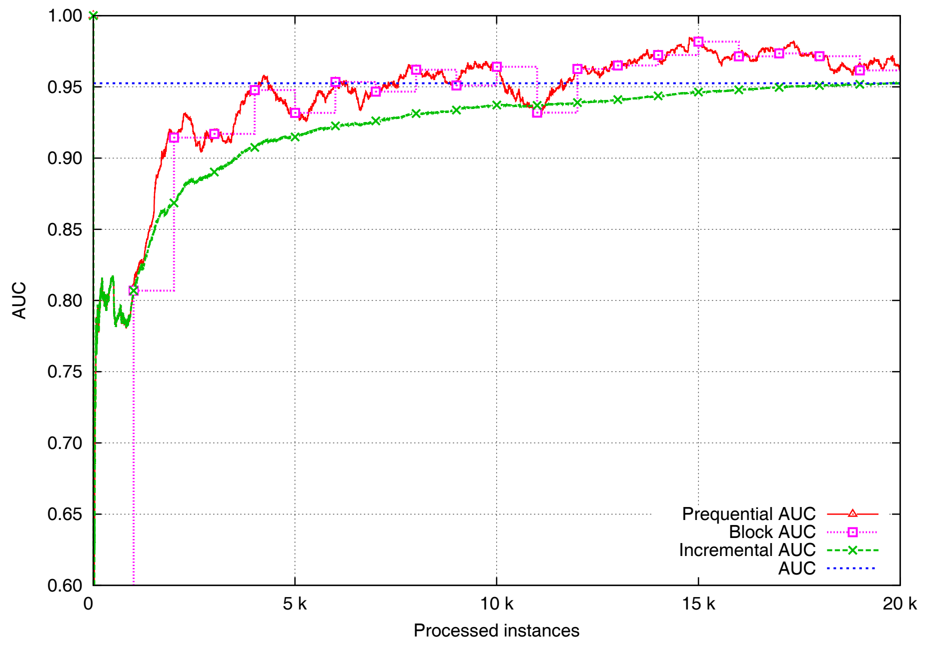


Figure 6.3: Batch, incremental, block-based, and prequential AUC on a data stream with no drifts ($\texttt{RBF}_{20k}$)

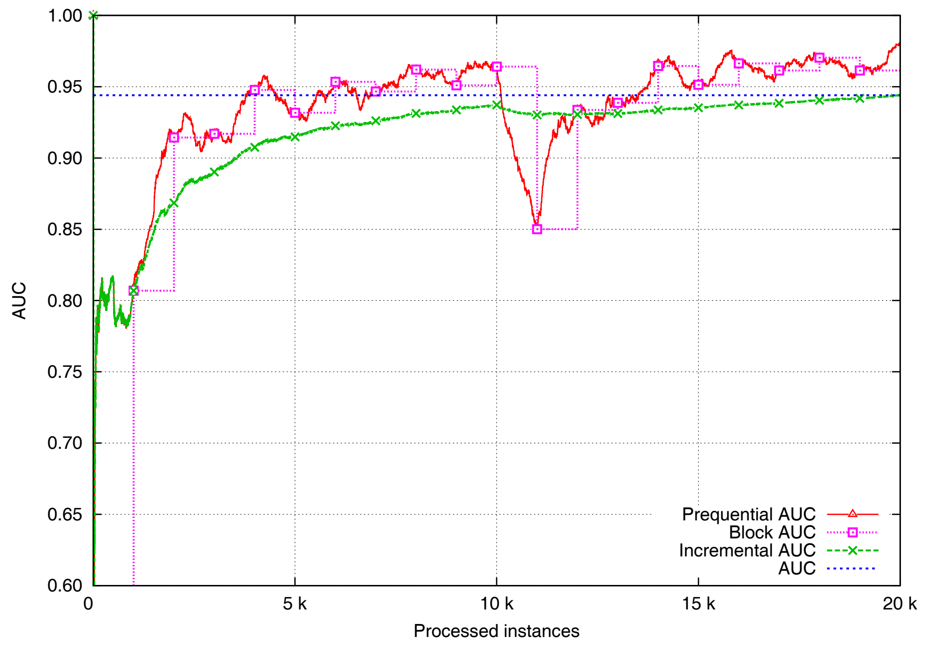


Figure 6.4: Batch, incremental, block-based, and prequential AUC on a data stream with a sudden drift after 10 k examples ($\texttt{RBF}_{20kSD}$)

On the stream without any drifts, presented in Figure 6.3, we can see that AUC calculated in blocks or prequentially is less pessimistic than AUC calculated incrementally. This is due to the fact that the true performance of an algorithm at a given point in time is obscured when calculated incrementally — algorithms are punished for early mistakes regardless of the level of performance they are eventually capable of, although this effect diminishes over time. This property has also been noticed by other researchers when visualizing classification accuracy over time [87, 69]. Therefore, if one is interested in the performance of a classifier at a given moment in time, prequential and block AUC give less pessimistic estimates, with prequential calculation producing a much smoother curve.

Figure 6.4 presents the difference between all four AUC calculation methods in the presence of concept drift. As one can see, after a sudden drift, occurring after 10 k examples, the change in performance is most visible when looking at prequential AUC over time. AUC calculated on blocks of examples also depicts this change, but delayed according to the block size. However, the most relevant observation is that AUC calculated incrementally is not capable of depicting drifts due to a long memory of predictions. For this reason, prequential evaluations are favored over incremental and block-based assessment in drifting environments where class labels are available after each example [69]. Therefore, prequential AUC is preferable to batch and incremental AUC when monitoring classifier performance online on drifting data streams.

6.3.2 Prequential AUC Averaged Over Entire Streams

The above analysis shows that prequential AUC has several advantages when monitored over time, especially in environments with possible concept drifts. However, in many situations, particularly when comparing the performance of multiple algorithms over several datasets, it is easier to compare simple numeric values rather than entire performance plots. Therefore, in such situations researchers are more interested in averaged performance values over entire streams.

As it was shown in Figure 6.4, in the presence of concept drift AUC calculated prequentially will showcase stronger performance drops than AUC calculated incrementally. If one is to determine algorithms capable of reacting to drifts, prequential AUC averaged over the entire stream gives a better insight about predictive performance. This is especially true for sudden concept drifts and short blips, which will have almost no impact on AUC calculated on the entire stream without any forgetting mechanism.

However, if no drifts are expected, or one is not sure if there is a possibility of concept changes during classification, AUC calculated traditionally in a batch manner over all examples should give the best performance estimate. Since in a stationary stream all examples represent a single concept, all predictions can be simultaneously taken into account during evaluation. Although batch AUC computation is not feasible for large data streams, we are interested how prequential and block calculations averaged over the entire stream compare to AUC calculated once using all predictions.

It is worth noticing that if we simultaneously take all testing examples into account, their order of appearance does not affect the final AUC estimation, as long as an example receives a certain score regardless of its position in the stream. This is contrary to AUC calculated prequentially or in blocks where the order of examples affects the final averaged performance values. Let us analyze two examples that demonstrate this issue.

Table 6.1 presents a stream where two classifiers give the lowest score to positive instances and the highest to negative instances. As a result, AUC calculated on the entire dataset at once will be 0.00, regardless of the order of examples. However, if positive and negative instances are clearly separated (all positive instances appear before all negative, or vice versa), averaged AUC calculated in blocks or prequentially with a window of $d = 2$ instances will give an estimate of 1.00 and 0.93, respectively. Such high estimates are due to the fact that for most window positions all instances have the same class label (Classifier 1). On the other hand, if the same examples arrive in a different order (Classifier 2), prequential or block AUC with the same window of $d = 2$ when averaged over the stream will be 0.00 just as batch AUC.

Table 6.2 presents an additional issue. For Classifier 3 is batch AUC is 0.57, averaged block AUC is 1.00, and averaged prequential AUC is 0.53. For Classifier 4, on the other hand, batch AUC is 0.00, block AUC 1.00, and prequential AUC 0.93. This shows that prequential AUC does not have to be necessarily higher than batch AUC. It is worth noting, that if the sequence of interleaved positives and negatives for Classifier 4 started with a negative example, block AUC would yield 0.00.

Table 6.1: An example in which two classifiers have the same batch AUC but different prequential (and block) AUC (for a window of $d = 2$ examples)

Classifier 1	-	-	-	-	-	-	-	-	+	+	+	+	+	+	+	+
t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Classifier 2	-	-	-	-	-	-	-	-	+	+	+	+	+	+	+	+
t	1	3	5	7	9	11	13	15	2	4	6	8	10	12	14	16

Table 6.2: An example in which one classifiers has higher batch AUC but lower prequential (and block) AUC (for a window of $d = 2$ examples)

Classifier 3	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-
t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Classifier 4	-	-	-	-	-	-	-	-	+	+	+	+	+	+	+	+
t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The above examples show, that depending on the calculation procedure, one can obtain AUC estimates that are far away from each other. This might be beneficial for concept-drifting streams, but not for stationary distributions. Since we cannot ensure equal prequential, block-based, and batch AUC estimates, we are interested in how different these estimates are on average. More importantly, we are interested in how these differences affect classifier evaluation. To answer these questions, we will use criteria for comparing evaluation measures proposed by Huang and Ling [79].

When discussing two different measures f and g used for evaluating two learning algorithms A and B, we want f and g to be *consistent* with each other. That is, when f shows that algorithm A is better than B, then g will not say B is better than A [79]. Furthermore, if f is more *discriminating* than g, we expect to see cases where f can tell the difference between A and B but g cannot, but not vice versa. These intuitive descriptions of consistency and discriminancy were made precise by the following definitions [79].

Definition 6.1. *For two measures f, g and two classifier outputs a, b on a domain Ψ, f and g are strictly* consistent *if there exists no $a, b \in \Psi$, such that $f(a) > f(b)$ and $g(a) < g(b)$.*

Definition 6.2. *For two measures f, g and two classifier outputs a, b on a domain Ψ, f is strictly more* discriminating *than g if there exists $a, b \in \Psi$ such that $f(a) \neq f(b)$ and $g(a) = g(b)$, and there exists no $a, b \in \Psi$, such that $g(a) \neq g(b)$ and $f(a) = f(b)$.*

As we have already shown, counter examples on strict consistency and discriminancy do exist for average AUC calculated on an entire stream and using blocks or sliding windows. Therefore, it is impossible to prove consistency and discriminancy between batch and prequential or block AUC, based on Definitions 6.1 and 6.2. In this case, we have to rather consider the degree of being consistent and the degree of being more discriminating. This leads to the definitions of the *degree of consistency* and *degree of discriminancy* [79].

Definition 6.3. *For two measures f, g and two classifier outputs a, b on a domain Ψ, let $R = \{(a,b)|a,b \in \Psi, f(a) > f(b), g(a) > g(b)\}$, $S = \{(a,b)|a,b \in \Psi, f(a) > f(b), g(a) < g(b)\}$. The* degree of consistency *of f and g is $\mathbf{C}$ $(0 \leq \mathbf{C} \leq 1)$, where $\mathbf{C} = \frac{|R|}{|R|+|S|}$.*

Definition 6.4. *For two measures f, g and two classifier outputs a, b on a domain Ψ, let $P = \{(a,b)|a,b \in \Psi, f(a) > f(b), g(a) = g(b)\}$, $Q = \{(a,b)|a,b \in \Psi, g(a) > g(b), f(a) = f(b)\}$. The* degree of discriminancy *for f over g is $\mathbf{D} = \frac{|P|}{|Q|}$.*

As it was suggested by Huang and Ling, two measures should agree on the majority of classifier evaluations to be comparable. Therefore, we require prequential AUC averaged over the entire stream to be $\mathbf{C} > 0.5$ consistent with batch AUC . Furthermore, the degree of discriminancy $\mathbf{D}$ shows how many times it is more likely that a given measure can tell the difference between two algorithms, when the other measure cannot. In particular, for $\mathbf{D} > 1$ we will be able to say that one measure is more discriminating than the other.

Since prequential and block AUC are dependent on the ranking of examples, their order in the stream, and the used window size d, it is difficult to formally prove that prequential AUC is statistically consistent ($\mathbf{C} > 0.5$) and less (or more) discriminating ($\mathbf{D} \neq 1$) than AUC calculated using the entire stream at once. Therefore, we will experimentally verify whether prequential AUC is statistically consistent with batch AUC and whether we can decide which calculation procedure is more discriminating. More importantly, empirical evaluations on artificial datasets will give us an insight on the practical degree of consistency and discriminancy for different class imbalance ratios and window sizes.

We test datasets with 4, 6, 8, and 10 testing examples. For each case, we enumerate *all possible orderings* of examples of *all possible pairs* of ranked lists with 50%, 34%, 14%

minority class examples and calculate prequential (and block) AUC for *all possible window sizes* $(1 < d < n)$. For a dataset with n_p positive examples and n examples in total, there are $\binom{\binom{n}{n_p}+2-1}{2} \cdot n!$ such non-repeating pairs of ranked lists with different example orderings. Due to such a large number of ordering/ranking possibilities, we only test datasets up to 10 instances. The three class imbalance ratios were chosen to show performance on a balanced dataset (50%), 1:2 imbalance ratio (34%), and the extreme case of only one positive instance regardless of the dataset size (14%).

We exhaustively compare all orderings of examples for all possible example rankings to verify the degree of consistency and discriminancy for different window sizes d. To obtain the degree of consistency, we count the number of pairs for which "$AUC(a) < AUC(b)$ *and* $pAUC(a) < pAUC(b)$" and the number of pairs for which "$AUC(a) < AUC(b)$ *and* $pAUC(a) > pAUC(b)$", where $AUC(\cdot)$ denotes batch calculated AUC and $pAUC(\cdot)$ prequential AUC averaged over the entire stream. To obtain the degree of discriminancy, we count the number of pairs which satisfy "$AUC(a) < AUC(b)$ *and* $pAUC(a) = pAUC(b)$" and the number of pairs which satisfy "$AUC(a) = AUC(b)$ *and* $pAUC(a) < pAUC(b)$". Similar computations where done for block AUC, denoted as $bAUC(\cdot)$. Tables 6.3–6.8 show the experiment results.

Regarding consistency, the results show that both block-based and prequential AUC have a high percentage of decisions consistent with batch AUC. More precisely, both estimations usually achieve a degree of consistency between 0.80 and 0.90, which is much larger than the required 0.5. However, it is also worth noticing that, for all class imbalance ratios and all window sizes, prequential AUC is more consistent with batch AUC than its block-calculated competitor. What is even more important is that this difference is most apparent for smaller window sizes, where prequential AUC usually has a 0.05 higher degree of consistency. Finally, it is worth noting that larger windows persistently allow to achieve estimations that are more consistent with batch AUC.

In terms of the degree of discriminancy, the results vary. For very small window sizes batch AUC seems to be better at differentiating rankings, whereas windows of sizes $d > 3$ invert this relation. However, once again it is worth noticing that prequential AUC is usually more discriminant than block AUC regardless of the window size or class-imbalance ratio. Moreover, higher class-imbalance ratios appear to make the differentiation more difficult for prequential and block estimations, especially for smaller window sizes. This is understandable, as with such small datasets, for higher class imbalance ratios only a single positive example is available. If a window is too small compared to the class-imbalance ratio, no positive examples are available in a window. This situation is visible in Table 6.8, where for all dataset sizes n the number of positive examples is $n_p = 1$.

A direct conclusion can be drawn from this observation — window sizes used for prequential (or block) AUC estimations should be large enough to always contain at least one positive example to ensure higher discriminancy. Nevertheless, apart from very small window sizes compared to the class-distribution, prequential AUC is comparably discriminant with AUC calculated on the entire stream.

Table 6.3: Statistical consistency compared to batch-calculated AUC for balanced datasets (50% both classes)

(a) Prequential AUC

n	d	$AUC(a) < AUC(b)\&$ $pAUC(a) < pAUC(b)$	$AUC(a) < AUC(b)\&$ $pAUC(a) > pAUC(b)$	**C**
4	2	226	30	0.883
4	3	276	0	1.000
6	2	78,558	14,206	0.847
6	3	95,086	11,688	0.891
6	4	106,566	10,098	0.913
6	5	109,152	3,840	0.966
8	2	53,158,628	11,185,976	0.826
8	3	63,240,034	10,425,237	0.858
8	4	69,182,304	9,972,770	0.874
8	5	70,761,192	7,693,012	0.902
8	6	73,479,168	6,411,408	0.920
8	7	75,503,520	2,403,360	0.969
10	2	60,634,866,784	14,063,999,524	0.812
10	3	71,130,163,463	13,290,485,752	0.843
10	4	74,037,644,404	14,163,943,582	0.839
10	5	78,920,700,364	11,176,341,290	0.876
10	6	73,168,730,742	13,467,350,816	0.845
10	7	79,076,275,296	9,293,058,744	0.895
10	8	83,944,244,880	6,701,320,800	0.926
10	9	86,539,824,000	2,926,627,200	0.967

(b) Block AUC

n	d	$AUC(a) < AUC(b)\&$ $bAUC(a) < bAUC(b)$	$AUC(a) < AUC(b)\&$ $bAUC(a) > bAUC(b)$	**C**
4	2	208	56	0.788
4	3	236	28	0.894
6	2	67,230	17,428	0.794
6	3	87,150	14,164	0.860
6	4	92,930	19,356	0.828
6	5	101,520	7,608	0.930
8	2	45,405,586	13,358,810	0.773
8	3	53,850,202	14,391,887	0.789
8	4	66,567,040	8,624,167	0.885
8	5	63,572,560	12,086,730	0.840
8	6	64,611,792	14,232,192	0.819
8	7	72,136,080	4,448,880	0.942
10	2	51,487,887,104	17,000,441,762	0.752
10	3	59,490,757,935	16,883,741,696	0.779
10	4	64,512,419,513	20,509,708,299	0.759
10	5	76,714,147,735	9,078,493,566	0.894
10	6	71,855,703,700	13,934,726,872	0.838
10	7	70,312,608,720	16,552,837,632	0.809
10	8	74,378,685,600	15,590,530,080	0.827
10	9	83,365,107,840	4,817,836,800	0.945

Table 6.4: Statistical discriminancy compared to batch-calculated AUC for balanced datasets (50% both classes)

(a) Prequential AUC

n	d	$AUC(a) < AUC(b)\&$ $pAUC(a) = pAUC(b)$	$AUC(a) = AUC(b)\&$ $pAUC(a) < pAUC(b)$	**D**
4	2	80	8	10.00
4	3	60	8	7.50
6	2	26,756	3,500	7.64
6	3	12,746	4,044	3.15
6	4	2,856	4,724	0.60
6	5	6,528	4,032	1.62
8	2	16,214,756	2,353,182	6.89
8	3	6,894,089	2,738,339	2.52
8	4	1,304,298	2,996,764	0.44
8	5	2,104,792	2,870,944	0.73
8	6	356,712	2,891,136	0.12
8	7	2,652,480	2,349,360	1.13
10	2	16,474,070,244	2,325,714,636	7.08
10	3	6,751,652,794	2,692,735,085	2.51
10	4	1,135,333,025	2,782,776,394	0.41
10	5	1,523,949,099	2,693,165,272	0.57
10	6	205,901,522	2,464,416,666	0.08
10	7	837,730,392	2,688,388,944	0.31
10	8	163,838,880	2,513,982,960	0.07
10	9	1,707,148,800	2,113,695,360	0.81

(b) Block AUC

n	d	$AUC(a) < AUC(b)\&$ $bAUC(a) = bAUC(b)$	$AUC(a) = AUC(b)\&$ $bAUC(a) < bAUC(b)$	**D**
4	2	72	0	∞
4	3	72	4	18.00
6	2	34,862	2,010	17.34
6	3	18,206	2,802	6.50
6	4	7,234	4,122	1.75
6	5	10,392	2,712	3.83
8	2	21,794,964	1,827,868	11.92
8	3	12,317,271	2,152,120	5.72
8	4	5,268,165	2,258,632	2.33
8	5	4,899,706	2,262,634	2.17
8	6	1,403,304	2,475,192	0.57
8	7	3,974,400	1,751,040	2.27
10	2	22,684,607,686	1,764,461,488	12.86
10	3	14,797,802,378	2,108,350,183	7.02
10	4	4,314,793,199	2,648,700,613	1.63
10	5	5,828,349,452	2,133,672,486	2.73
10	6	1,051,552,508	2,577,780,700	0.41
10	7	2,341,618,080	2,381,208,000	0.98
10	8	840,188,880	2,230,413,840	0.38
10	9	2,990,655,360	1,607,114,880	1.86

Table 6.5: Statistical consistency compared to batch-calculated AUC for imbalanced datasets (34% minority class)

(a) Prequential AUC

n	d	$AUC(a) < AUC(b)$& $pAUC(a) < pAUC(b)$	$AUC(a) < AUC(b)$& $pAUC(a) > pAUC(b)$	**C**
4	2	96	8	0.923
4	3	112	8	0.933
6	2	44,328	7,430	0.856
6	3	53,318	7,312	0.879
6	4	58,774	6,904	0.895
6	5	62,064	3,312	0.949
8	2	34,250,730	6,980,006	0.831
8	3	40,699,497	6,793,142	0.857
8	4	44,193,331	6,982,899	0.864
8	5	45,463,956	5,426,998	0.893
8	6	47,516,640	4,326,864	0.917
8	7	49,281,120	2,127,600	0.959
10	2	14,056,095,136	2,991,723,204	0.825
10	3	16,506,930,025	3,058,609,668	0.844
10	4	17,397,851,565	3,458,536,944	0.834
10	5	17,857,902,489	3,077,583,820	0.853
10	6	17,368,267,282	3,208,774,634	0.844
10	7	18,541,658,208	2,198,965,392	0.894
10	8	19,469,178,720	1,653,883,920	0.922
10	9	20,338,174,080	834,825,600	0.961

(b) Block AUC

n	d	$AUC(a) < AUC(b)$& $bAUC(a) < bAUC(b)$	$AUC(a) < AUC(b)$& $bAUC(a) > bAUC(b)$	**C**
4	2	92	8	0.920
4	3	102	18	0.850
6	2	39,222	9,582	0.804
6	3	49,482	7,590	0.867
6	4	52,302	11,468	0.820
6	5	59,496	5,352	0.917
8	2	28,945,260	8,463,258	0.774
8	3	34,682,961	8,761,157	0.798
8	4	43,361,987	6,740,645	0.865
8	5	41,705,650	7,623,636	0.845
8	6	42,382,608	8,985,336	0.825
8	7	47,743,920	3,418,560	0.933
10	2	12,195,033,276	3,413,021,306	0.781
10	3	13,792,398,281	3,675,831,232	0.790
10	4	15,903,298,910	4,113,647,610	0.794
10	5	17,306,025,784	3,102,200,565	0.848
10	6	17,469,134,532	2,947,901,726	0.856
10	7	16,864,120,896	3,466,868,304	0.829
10	8	17,293,451,760	3,656,527,920	0.825
10	9	19,822,360,320	1,216,010,880	0.942

Table 6.6: Statistical discriminancy compared to batch-calculated AUC for imbalanced datasets (34% minority class)

(a) Prequential AUC

n	d	$AUC(a) < AUC(b)\&$ $pAUC(a) = pAUC(b)$	$AUC(a) = AUC(b)\&$ $pAUC(a) < pAUC(b)$	**D**
4	2	40	0	∞
4	3	24	0	∞
6	2	15,922	1,676	9.50
6	3	7,050	2,080	3.39
6	4	1,988	2,408	0.83
6	5	2,304	2,040	1.13
8	2	10,822,384	1,445,690	7.49
8	3	4,560,481	1,698,162	2.69
8	4	904,810	1,861,822	0.49
8	5	1,156,828	1,864,496	0.62
8	6	191,496	1,831,680	0.10
8	7	646,560	1,517,760	0.43
10	2	4,231,464,860	512,002,316	8.26
10	3	1,713,743,507	605,486,578	2.83
10	4	295,397,178	680,067,479	0.43
10	5	297,697,500	660,113,562	0.45
10	6	43,162,624	665,026,736	0.06
10	7	104,005,512	765,190,632	0.14
10	8	79,602,480	614,255,040	0.13
10	9	114,589,440	486,864,000	0.24

(b) Block AUC

n	d	$AUC(a) < AUC(b)\&$ $bAUC(a) = bAUC(b)$	$AUC(a) = AUC(b)\&$ $bAUC(a) < bAUC(b)$	**D**
4	2	44	0	∞
4	3	24	0	∞
6	2	18,876	1,050	17.98
6	3	10,608	1,042	10.18
6	4	3,896	2,156	1.81
6	5	2,832	1,416	2.00
8	2	14,644,602	1,061,404	13.80
8	3	8,609,002	1,206,625	7.13
8	4	1,978,408	1,835,340	1.08
8	5	2,718,496	1,620,150	1.68
8	6	667,056	1,522,224	0.44
8	7	892,800	1,100,880	0.81
10	2	5,671,228,618	384,816,444	14.74
10	3	3,811,053,687	451,977,687	8.43
10	4	1,134,839,167	602,144,260	1.88
10	5	824,957,460	625,114,249	1.32
10	6	203,168,282	669,160,838	0.30
10	7	513,639,912	557,695,848	0.92
10	8	252,685,440	474,176,160	0.53
10	9	249,217,920	341,107,200	0.73

Table 6.7: Statistical consistency compared to batch-calculated AUC for imbalanced datasets (14% minority class)

(a) Prequential AUC

n	d	$AUC(a) < AUC(b)\&$ $pAUC(a) < pAUC(b)$	$AUC(a) < AUC(b)\&$ $pAUC(a) > pAUC(b)$	**C**
4	2	96	8	0.923
4	3	112	8	0.933
6	2	6,888	840	0.891
6	3	8,068	1,240	0.867
6	4	8,568	1,220	0.875
6	5	9,264	816	0.919
8	2	701,568	96,768	0.879
8	3	841,168	145,632	0.852
8	4	868,680	164,584	0.841
8	5	891,960	164,384	0.844
8	6	936,144	139,056	0.871
8	7	1,006,560	82,080	0.925
10	2	99,792,000	14,636,160	0.872
10	3	121,113,792	21,683,232	0.848
10	4	125,448,672	25,481,808	0.831
10	5	127,000,512	27,198,096	0.824
10	6	128,755,248	27,134,520	0.826
10	7	132,687,744	24,559,776	0.844
10	8	139,400,640	19,555,920	0.877
10	9	148,861,440	10,805,760	0.932

(b) Block AUC

n	d	$AUC(a) < AUC(b)\&$ $bAUC(a) < bAUC(b)$	$AUC(a) < AUC(b)\&$ $bAUC(a) > bAUC(b)$	**C**
4	2	92	8	0.920
4	3	102	18	0.850
6	2	6,096	784	0.886
6	3	8,152	644	0.927
6	4	8,276	934	0.899
6	5	8,880	1,200	0.881
8	2	571,680	71,040	0.889
8	3	796,032	90,528	0.898
8	4	914,944	64,888	0.934
8	5	923,656	77,832	0.922
8	6	940,368	99,312	0.904
8	7	982,800	105,840	0.903
10	2	81,250,560	11,632,320	0.875
10	3	105,151,848	16,303,848	0.866
10	4	126,349,656	12,907,656	0.907
10	5	139,374,336	8,251,152	0.944
10	6	142,754,112	12,808,368	0.918
10	7	141,324,576	10,824,720	0.929
10	8	142,572,240	12,827,520	0.917
10	9	146,603,520	13,063,680	0.918

Table 6.8: Statistical discriminancy compared to batch-calculated AUC for imbalanced datasets (14% minority class)

(a) Prequential AUC

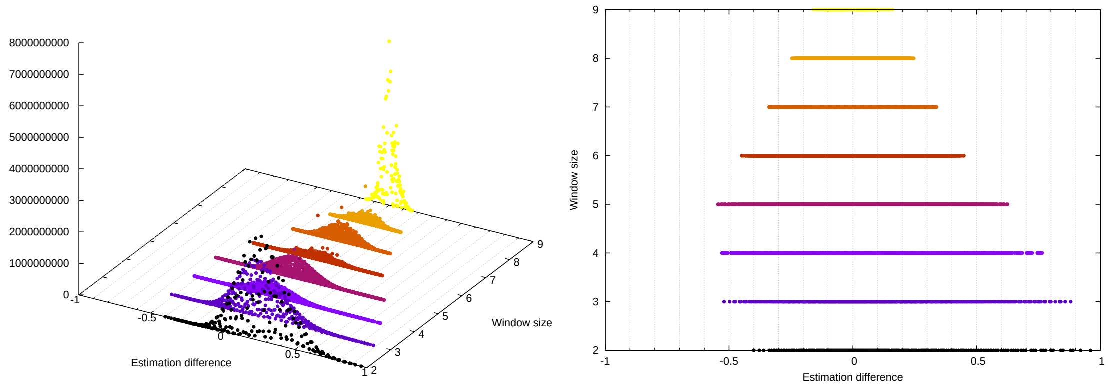
n	d	$AUC(a) < AUC(b)$& $pAUC(a) = pAUC(b)$	$AUC(a) = AUC(b)$& $pAUC(a) < pAUC(b)$	**D**
4	2	40	0	∞
4	3	24	0	∞
6	2	3,072	0	∞
6	3	1,492	0	∞
6	4	1,012	0	∞
6	5	720	0	∞
8	2	330,624	0	∞
8	3	142,160	0	∞
8	4	95,696	0	∞
8	5	72,616	0	∞
8	6	53,760	0	∞
8	7	40,320	0	∞
10	2	48,867,840	0	∞
10	3	20,498,976	0	∞
10	4	12,365,520	0	∞
10	5	9,097,392	0	∞
10	6	7,406,232	0	∞
10	7	6,048,480	0	∞
10	8	4,339,440	0	∞
10	9	3,628,800	0	∞

(b) Block AUC

n	d	$AUC(a) < AUC(b)$& $bAUC(a) = bAUC(b)$	$AUC(a) = AUC(b)$& $bAUC(a) < bAUC(b)$	**D**
4	2	44	0	∞
4	3	24	0	∞
6	2	3,920	0	∞
6	3	2,004	0	∞
6	4	1,590	0	∞
6	5	720	0	∞
8	2	486,240	0	∞
8	3	242,400	0	∞
8	4	149,128	0	∞
8	5	127,472	0	∞
8	6	89,280	0	∞
8	7	40,320	0	∞
10	2	70,413,120	0	∞
10	3	41,840,304	0	∞
10	4	24,038,688	0	∞
10	5	15,670,512	0	∞
10	6	7,733,520	0	∞
10	7	11,146,704	0	∞
10	8	7,896,240	0	∞
10	9	3,628,800	0	∞

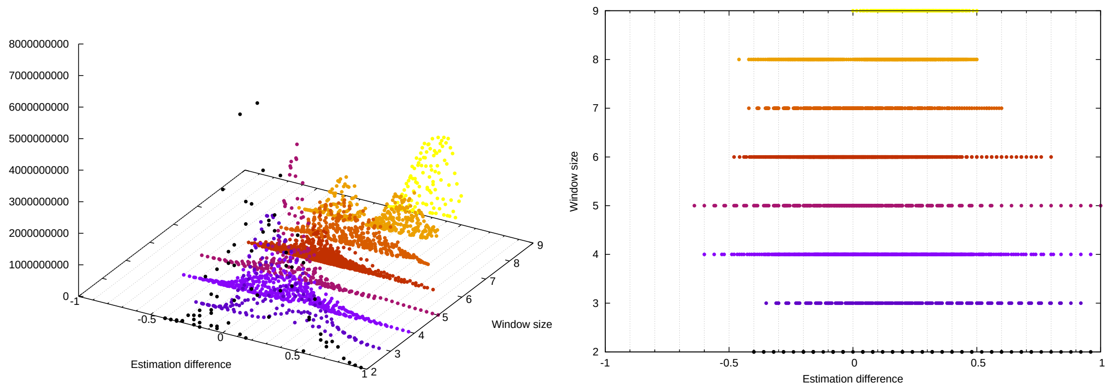
Apart from the number of pairs when prequential or block estimations are consistent or more discriminating than batch AUC, one may be interested in the absolute difference between the values of these estimations. To verify these differences, we have calculated and plotted the values of $pAUC(a) - AUC(a)$ and $bAUC(a) - AUC(a)$ for different class imbalance ratios. Figures 6.5–6.10 present these differences on 3-dimendional plots.

The left-hand side of each figure presents a 3-dimensional plot, where the x-axis denotes the difference between prequential (or block) AUC and batch AUC, the y-axis describes window sizes, and the z-axis shows the number of rankings for which a given difference was observed. The right-hand side of each figure shows a 2-dimensional top view of the same plot. The left plots are intended to demonstrate the dominating difference values and their variation for each window size. It is also worth noticing that these plots usually demonstrate peaks around the 0.0 difference. The right plots, on the other hand, clearly show the range of possible differences for each window size.



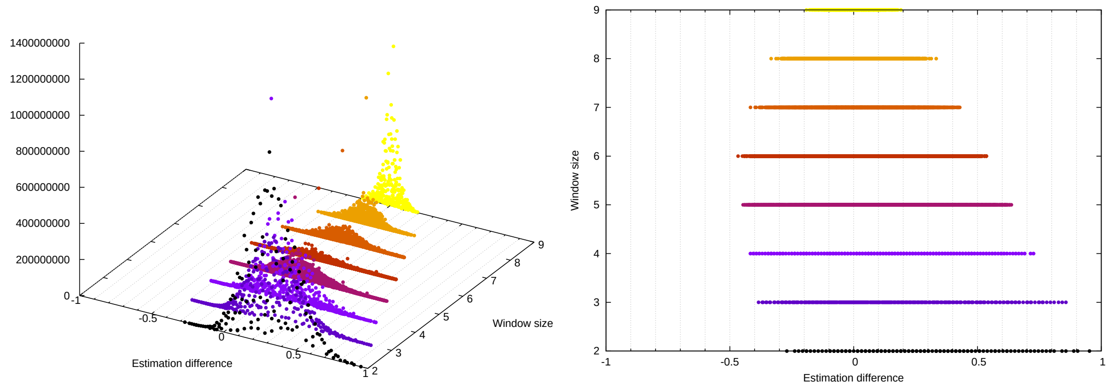
(a) 3-dimensional plot (b) 2-dimensional view

Figure 6.5: Differences between prequential and batch AUC for different window sizes on the largest balanced dataset (50% examples of both classes)



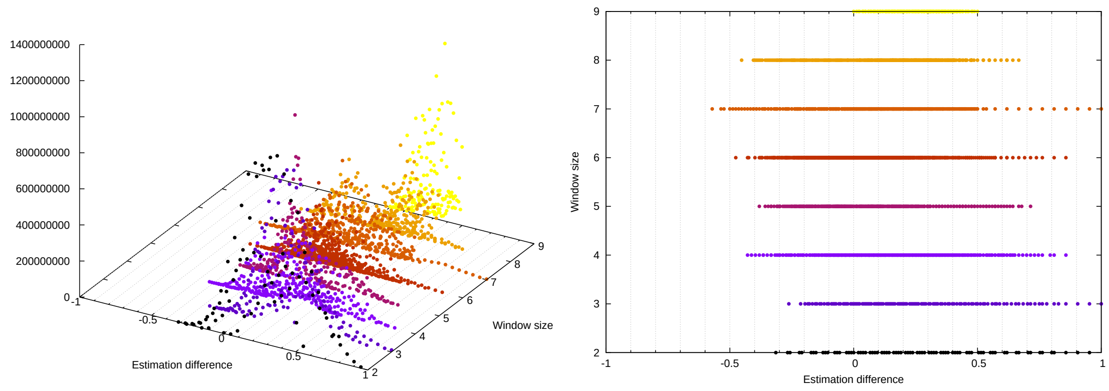
(a) 3-dimensional plot (b) 2-dimensional view

Figure 6.6: Differences between block and batch AUC for different window sizes on the largest balanced dataset (50% examples of both classes)



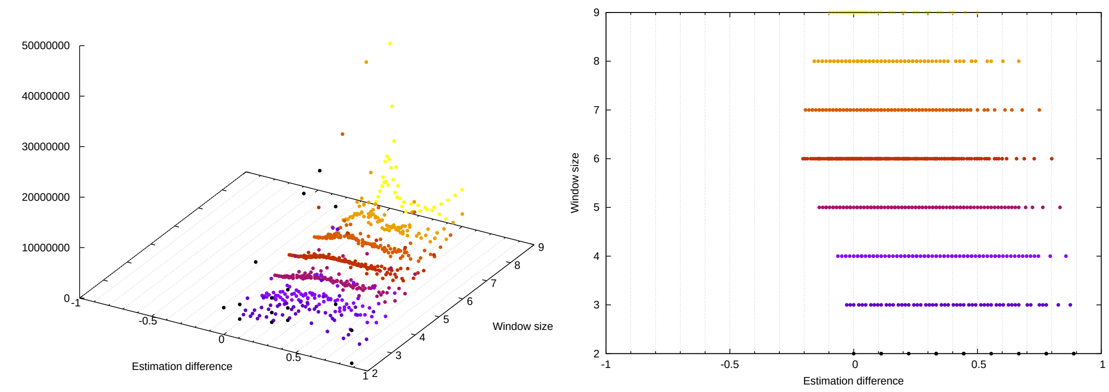
(a) 3-dimensional plot (b) 2-dimensional view

Figure 6.7: Differences between prequential and batch AUC for different window sizes on the largest dataset with medium class imbalance (34% minority class examples)



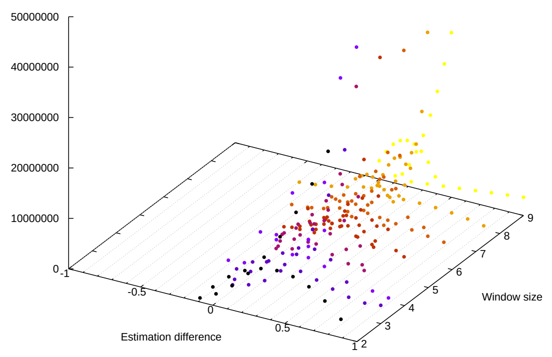
(a) 3-dimensional plot (b) 2-dimensional view

Figure 6.8: Differences between block and batch AUC for different window sizes on the largest dataset with medium class imbalance (34% minority class examples)

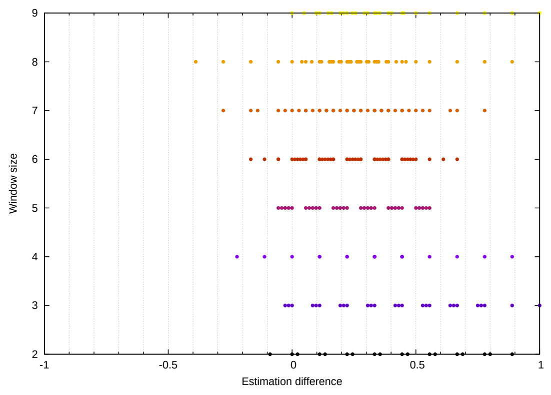


(a) 3-dimensional plot (b) 2-dimensional view

Figure 6.9: Differences between prequential and batch AUC for different window sizes on the largest dataset with high class imbalance (14% minority class examples)



(a) 3-dimensional plot



(b) 2-dimensional view

Figure 6.10: Differences between block and batch AUC for different window sizes on the largest dataset with high class imbalance (14% minority class examples)

As the above plots show, most prequential estimates of AUC are very close AUC calculated on the entire dataset. For all three class imbalance ratios, one can notice single points above the bell curve directly above the zero difference value. This showcases, that the most common difference between batch and prequential AUC is zero. This is not so obvious for block-based estimates. When compared with prequential AUC, block estimates have much "wider" bell curves without such strong peaks around zero.

Looking at 2-dimensional plots, it is worth noticing that for small window sizes, small compared to the class imbalance ratio, prequential AUC gives a more optimistic estimate compared to batch AUC. This issue is related to that of lower discriminancy for small windows or high class imbalance ratios (plots for the 14% minority class dataset have much fewer distinct points). When there are no positive examples in the window, AUC for that window is equal to 1, which can lead to overestimating AUC over time. However, as the plots show, larger windows clearly mitigate this problem. The 2-dimensional plots also show that, compared to prequential computation, block -based calculations are more prone to over- and under- estimation of AUC.

The above analyses have shown that prequential AUC averaged over time is highly consistent and comparably discriminant to AUC calculated on the entire stream. We have also seen that the absolute difference between these two measures is very small for most example orderings. Furthermore, we have noticed that the window size used for prequential calculation should be large enough to contain at least one positive example at all time, to avoid overly optimistic AUC estimates and, as an effect, low discriminancy. Finally, prequential AUC proved to be more consistent, more discriminant, and closer in terms of absolute values to AUC than block-calculated AUC. The following section, evaluates AUC on real and synthetic data streams, including streams with class distribution changes.

6.4 Experimental Analysis

In this section, we summarize two groups of experiments: one showcasing the processing time and monitoring capabilities of AUC as an evaluation measure, and another assessing online ensembles on data streams with class distribution changes.

In the first group, we compare the processing time required to evaluate a stream prequentially using AUC and the κ statistic, which has been deemed more appropriate than AUC due to computational costs [12]. Moreover, we analyze the effectiveness of prequential AUC as a basis for drift detection, both for real and virtual sudden drifts.

The second group of experiments compares prequential accuracy and AUC on real and synthetic datasets containing various types of drift. We concentrate on the effect of class imbalance and class distribution changes during evaluation. As a result, we evaluate online ensembles analyzed in Chapter 5 on data streams with virtual drift.

6.4.1 Experimental Setup

During the comparison of prequential accuracy and AUC on real and synthetic datasets, we tested seven different classifiers:

- Online Bagging with an ADWIN change detector (Bag),
- Leveraging Bagging (Lev),
- Dynamic Weighted Majority (DWM),
- Adaptive Classifier Ensemble (ACE),
- Online Accuracy Updated Ensemble (OAUE),
- Naive Bayes (NB),
- and a Very Fast Decision Tree with Naive Bayes leaves (VFDT).

The first five algorithms are online ensembles, whose selection was discussed in Section 5.3.3. By using the same algorithms as in the previous chapter, we aim at extending the analysis of online ensembles by studying their performance in the presence of class imbalance and virtual drift. Naive Bayes and VFDT were additionally chosen as incremental algorithms without any forgetting mechanism, to showcase the differences between prequential AUC and accuracy also for single classifiers. For experiments concerning drift detection and the evaluation time of prequential AUC, we only utilized VFDT with Naive Bayes leaves, similarly as was done in [69].

All the algorithms and evaluation methods were implemented in Java as part of the MOA framework [15]. The experiments were conducted on a machine equipped with a dual-core Intel i7-2640M CPU, 2.8Ghz processor and 16 GB of RAM. For all the ensemble methods (Bag, Lev, DWM, ACE, OAUE) we used 10 Very Fast Decision Trees as base learners, each with a grace period $n_{min} = 100$, split confidence $\delta = 0.01$, and tie-threshold $\psi = 0.05$ [62].

6.4.2 Datasets

Classifier comparison

In experiments using prequential AUC as an evaluation metric, we used 2 real and 12 synthetic datasets (generator scripts are available in Appendix A). For the real-world datasets we chose two data streams which were used in the previous chapters; see Sections 3.3.1 and 4.5.2 for a detailed description. In particular, we chose `Airlines` (`Air`) as a large, balanced dataset and `PAKDD` as a smaller but imbalanced dataset.

Additionally, we used the MOA framework [15] to generate 12 artificial datasets with different types of concept drift. We used modified versions of generators discussed in Section 3.3.1, which were capable of controlling the class imbalance ratio. More specifically, the SEA generator [155] was used to create a stream without drifts ($\texttt{SEA}_{ND}$), as well as three streams with sudden changes and a constant 1:1 ($\texttt{SEA}_{1}$), 1:10 ($\texttt{SEA}_{10}$), 1:100 ($\texttt{SEA}_{100}$) class imbalance ratio. Similarly, the Hyperplane generator [163] was used to simulate three streams with different class ratios, 1:1 ($\texttt{Hyp}_{1}$), 1:10 ($\texttt{Hyp}_{10}$), 1:100 ($\texttt{Hyp}_{100}$), but with a continuous incremental drift rather than sudden changes. We also tested the performance of prequential accuracy and AUC in the presence of very short, temporary changes in a stream (`RBF`) created using the RBF generator [15].

Apart from data containing real drifts, we additionally created four streams with virtual drifts, i.e., class distribution changes over time. $\texttt{SEA}_{RC}$ contains three sudden class ratio changes (1:1/1:100/1:10/1:1) appearing every 250 k examples, while $\texttt{SEA}_{RC+D}$ contains identical ratio changes combined with real sudden drifts. Analogously, $\texttt{Hyp}_{RC}$ simulates a continuous ratio change from 1:1 to 1:100 throughout the entire stream, while $\texttt{Hyp}_{RC+D}$ combines that ratio change with an ongoing incremental drift. It is worth mentioning that all the synthetic datasets, apart from `RBF`, contained 5% to 10% examples with class noise to make the classification task more challenging.

Drift detection

For experiments assessing prequential AUC as a measure for monitoring drift, we created 7 synthetic datasets using the SEA (`SEA`), RBF (`RBF`), Random Tree (`RT`), and Agrawal (`Agr`) generators [15]. Each dataset tested for a single reaction (or lack of one) to a sudden change. $\texttt{SEA}_{NoDrift}$ contained no changes, and should not trigger any drift detector, while `RT` involved a single sudden change after 30 k examples. The $\texttt{Agr}_{1}$, $\texttt{Agr}_{10}$, $\texttt{Agr}_{100}$ datsets also contained a single sudden change after 30 k examples, but had a 1:1, 1:10, 1:100 class imbalance ratio, respectively. Finally, $\texttt{SEA}_{Ratio}$ included a sudden 1:1/1:100 ratio change after 10 k examples and $\texttt{RBF}_{Blips}$ contained two short temporary changes, which should not trigger the detector.

Evaluation time

Additionally, two small data streams created using the RBF generator ($\texttt{RBF}_{20k}$, $\texttt{RBF}_{20kSD}$) were used to showcase the evaluation speed of prequential AUC. The characteristics of all the datasets are given in Table 6.9.

Table 6.9: Characteristic of datasets

Dataset	#Inst	#Attrs	Class ratio	Noise	#Drifts	Drift type
$\mathtt{SEA}_{ND}$	100 k	3	1:1	10%	0	none
$\mathtt{SEA}_{1}$	1 M	3	1:1	10%	3	sudden
$\mathtt{SEA}_{10}$	1 M	3	1:10	10%	3	sudden
$\mathtt{SEA}_{100}$	1 M	3	1:100	10%	3	sudden
$\mathtt{Hyp}_{1}$	500 k	5	1:1	5%	1	incremental
$\mathtt{Hyp}_{10}$	500 k	5	1:10	5%	1	incremental
$\mathtt{Hyp}_{100}$	500 k	5	1:100	5%	1	incremental
RBF	1 M	20	1:1	0%	2	blips
$\mathtt{SEA}_{RC}$	1 M	3	1:1/1:100/1:10/1:1	10%	3	virtual
$\mathtt{SEA}_{RC+D}$	1 M	3	1:1/1:100/1:10/1:1	10%	3	sud.+virt.
$\mathtt{Hyp}_{RC}$	500 k	3	1:1 → 1:100	5%	1	virtual
$\mathtt{Hyp}_{RC+D}$	500 k	3	1:1 → 1:100	5%	1	inc.+virt.
Air	539 k	7	1:1	-	-	unknown
PAKDD	50 k	30	1:4	-	-	unknown
$\mathtt{SEA}_{NoDrift}$	20 k	3	1:1	10%	0	none
$\mathtt{Agr}_{1}$	40 k	9	1:1	1%	1	sudden
$\mathtt{Agr}_{10}$	40 k	9	1:10	1%	1	sudden
$\mathtt{Agr}_{100}$	40 k	9	1:100	1%	1	sudden
RT	40 k	10	1:1	0%	1	sudden
$\mathtt{SEA}_{Ratio}$	40 k	3	1:1/1:100	10%	1	virtual
$\mathtt{RBF}_{Blips}$	40 k	20	1:1	0%	2	blips
$\mathtt{RBF}_{20k}$	20 k	20	1:1	0%	0	none
$\mathtt{RBF}_{20kSD}$	20 k	20	1:1	0%	1	sudden

6.4.3 Prequential AUC Evaluation Time

First, we performed a simple comparison of classifier evaluation time using AUC and the κ statistic, both calculated prequentially. Although in Section 6.2 we have shown that the time per example required to calculate prequential AUC is constant, we want to verify evaluation time on a concrete data stream. Furthermore, some researchers have argued that AUC is too computationally expensive to be used for evaluating data stream classifiers. In particular, the inability to calculate AUC efficiently made authors suggest, for example, that "the Kappa statistic is more appropriate for data streams than a measure such as the area under the ROC curve" [12]. Therefore, to verify whether AUC calculated prequentially overcomes these limitations, we compare its processing time with that of the κ statistic.

We compare the average evaluation time required per example for two small data streams, one without any drift ($\mathtt{RBF}_{20k}$) and one with a single sudden drift ($\mathtt{RBF}_{20kSD}$). We use the MOA framework to compare the time required to evaluate a single Hoeffding Tree using both measures. Originally, the MOA framework calculates several evaluation metrics per example, therefore, we created two separate evaluation functions which calculate only prequential AUC and prequential κ, respectively. Both measures require only a window of past predictions and the number of attributes does not affect calculation time. Therefore, we calculate evaluation time only on two datasets. However, we average the acquired results over ten runs to minimize the effect of random measurement variability. Table 6.10 presents evaluation time per example using prequential AUC and prequential κ on a window of $d = 1000$ examples.

Table 6.10: Evaluation time per example [ms] using prequential AUC and prequential κ on a window of $d = 1000$ examples (averaged over 10 runs $\pm$ standard deviation)

	Prequential AUC	Prequential κ
$\texttt{RBF}_{20k}$	7.230 ± 0.158	7.189 ± 0.246
$\texttt{RBF}_{20kSD}$	7.344 ± 0.519	7.287 ± 0.243

As results in Table 6.10 show, evaluation using prequential AUC is only slightly slower than using the κ statistic on both datasets. The difference is very small and proportionally AUC is 0.57% slower for $\texttt{RBF}_{20k}$ and 0.79% for $\texttt{RBF}_{20kSD}$. It is worth noting that this difference could be larger for a larger window size, however, due to the properties of red-black trees it would grow logarithmically with the window size.

Therefore, prequential AUC offers similar evaluation time compared to the κ statistic for a reasonable window size of $d = 1000$ examples. Furthermore, this difference can grow only logarithmically for larger window sizes. Thus, prequential AUC, contrary to AUC calculated on entire streams, should not be deemed too computationally expensive for evaluating data stream classifiers.

6.4.4 Drift Detection Using Prequential AUC

The next group of experiments involved using the PH test to detect drifts based on changes in prequential accuracy and prequential AUC. To compare both metrics, we used window sizes (1000–5000) and test parameters $\lambda = 100$, $\delta = 0.1$, as proposed in [69]. Table 6.11 presents the number of missed versus false detection counts, with average delay time for correct detections. The results refer to total counts and means over 10 runs of streams generated with different seeds.

Concerning datasets with balanced classes ($\texttt{SEA}_{NoDrift}$, $\texttt{RT}$, $\texttt{Agr}_1$, $\texttt{RBF}_{Blips}$), both evaluation metrics provide similar drift detection rates and delays. However, for datasets with high class imbalance ($\texttt{Agr}_{10}$, $\texttt{Agr}_{100}$) the PH test notes more missed detections for accuracy. This is probably due to the plot "flattening" caused by promoting majority class predictions. On the other hand, detectors which use AUC have less missed detections for highly imbalanced streams, but still suffer from a relatively high number of false alarms. This suggests that detectors using AUC should probably be parametrized differently than those using accuracy.

However, the most visible difference is for the stream with class ratio changes ($\texttt{SEA}_{Ratio}$). The PH test misses all virtual drifts when using accuracy as the base metric, but detects all the drifts when prequential AUC is used. This confirms, that in imbalanced evolving environments the use of AUC as an evaluation measure could be of more value than standard accuracy.

Table 6.11: Number of missed and false detections (in the format missed:false) obtained using the PH test with prequential accuracy (Acc) and prequential AUC (AUC). Average delays of correct detections are given in parenthesis, where (-) means that the detector was not triggered or the dataset did not contain any change. Subscripts in column names indicate the number of examples used for estimating errors.

	Acc_{1k}	Acc_{2k}	Acc_{3k}	Acc_{4k}	Acc_{5k}
$\texttt{SEA}_{NoDrift}$	0:0 (-)	0:0 (-)	0:0 (-)	0:0 (-)	0:0 (-)
$\texttt{Agr}_1$	0:2 (1040)	0:1 (1859)	0:0 (2843)	1:0 (4033)	5:0 (4603)
$\texttt{Agr}_{10}$	0:9 (1202)	0:3 (1228)	0:2 (1679)	0:2 (2190)	0:2 (2817)
$\texttt{Agr}_{100}$	2:12 (1610)	2:17 (2913)	2:10 (3136)	3:12 (3903)	3:10 (4612)
RT	6:0 (1843)	7:0 (2621)	8:0 (2933)	8:0 (3754)	8:0 (4695)
$\texttt{SEA}_{Ratio}$	10:0 (-)	10:0 (-)	10:0 (-)	10:0 (-)	10:0 (-)
$\texttt{RBF}_{Blips}$	0:2 (-)	0:1 (-)	0:0 (-)	0:0 (-)	0:0 (-)
	AUC_{1k}	AUC_{2k}	AUC_{3k}	AUC_{4k}	AUC_{5k}
$\texttt{SEA}_{NoDrift}$	0:0 (-)	0:0 (-)	0:0 (-)	0:0 (-)	0:0 (-)
$\texttt{Agr}_1$	2:2 (1042)	3:1 (1760)	4:1 (2726)	4:0 (3773)	7:0 (4640)
$\texttt{Agr}_{10}$	0:5 (868)	0:5 (1539)	0:1 (1506)	0:1 (1778)	1:1 (2197)
$\texttt{Agr}_{100}$	0:19 (1548)	0:18 (2461)	1:9 (2664)	1:11 (3563)	2:9 (4835)
RT	3:0 (1815)	5:0 (2407)	6:0 (3105)	6:0 (4121)	7:0 (4725)
$\texttt{SEA}_{Ratio}$	0:0 (1339)	0:0 (2249)	0:0 (3152)	0:0 (4057)	0:0 (4959)
$\texttt{RBF}_{Blips}$	0:3 (-)	0:1 (-)	0:0 (-)	0:0 (-)	0:0 (-)

6.4.5 Classifier Comparison

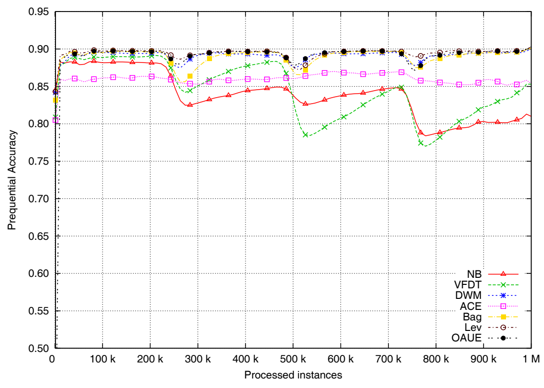
Finally, we compared the predictive performance of five online ensembles discussed in Chapter 5 (DWM, ACE, Bag, Lev, OAUE) and two additional single classifiers (NB, VFDT). All of the analyzed algorithms were tested in terms of accuracy (Acc.) and the area under the ROC curve (AUC). The results were obtained using the prequential evaluation procedure [62], with a sliding window of $d = 1000$ examples. Table 6.12 presents a comparison of average prequential accuracy and prequential AUC.

By comparing average values of the analyzed evaluation measures, we can see that for datasets with a balanced class ratio (SEA, $\texttt{SEA}_1$, $\texttt{Hyp}_1$, RBF, Air) both measures have similar values. However, as expected, for datasets with class imbalance ($\texttt{SEA}_{10}$, $\texttt{SEA}_{100}$, $\texttt{Hyp}_{10}$, $\texttt{Hyp}_{100}$, PAKKD, $\texttt{SEA}_{RC}$, $\texttt{SEA}_{RC+D}$, $\texttt{Hyp}_{RC}$, $\texttt{Hyp}_{RC+D}$) accuracy does not demonstrate the difficulties the classifiers have with recognizing minority class examples, while AUC clearly showcases this problem. The differences between accuracy and AUC are even more visible on graphical plots depicting algorithm performance in time. Figures 6.11–6.16 present selected plots, which best characterize the differences between both measures.

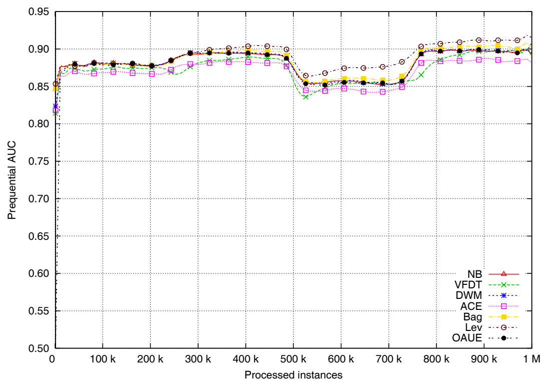
Comparing Figures 6.11 and 6.12, we can notice how the class imbalance ratio affects both prequential accuracy and AUC. The accuracy plot visibly flattens when the class imbalance ratio rises, but absolute values almost do not change. AUC on the other hand flattens but its value drastically changes, showing more clearly the classifiers' inability to recognize the minority class.

Table 6.12: Average prequential accuracy (Acc.) and AUC (AUC)

	NB		VFDT		DWM		ACE		Bag		Lev		OAUE	
	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC
$\texttt{SEA}_{ND}$	0.86	0.90	0.89	0.89	0.88	0.90	0.86	0.89	0.89	0.90	0.90	0.90	0.89	0.90
$\texttt{SEA}_{1}$	0.84	0.88	0.85	0.87	0.89	0.88	0.86	0.87	0.89	0.88	0.89	0.89	0.89	0.88
$\texttt{SEA}_{10}$	0.84	0.74	0.87	0.73	0.89	0.74	0.87	0.72	0.89	0.74	0.89	0.75	0.89	0.74
$\texttt{SEA}_{100}$	0.89	0.54	0.89	0.54	0.90	0.54	0.88	0.52	0.90	0.54	0.90	0.57	0.90	0.54
$\texttt{Hyp}_{1}$	0.78	0.85	0.81	0.87	0.88	0.92	0.78	0.83	0.88	0.93	0.86	0.92	0.88	0.93
$\texttt{Hyp}_{10}$	0.88	0.80	0.89	0.75	0.91	0.76	0.88	0.71	0.91	0.81	0.91	0.80	0.91	0.82
$\texttt{Hyp}_{100}$	0.94	0.57	0.93	0.53	0.94	0.52	0.89	0.50	0.94	0.56	0.94	0.55	0.94	0.55
RBF	0.74	0.83	0.97	0.99	0.98	1.00	0.87	0.89	0.99	1.00	0.99	1.00	0.99	1.00
$\texttt{SEA}_{RC}$	0.86	0.77	0.89	0.77	0.89	0.77	0.87	0.75	0.90	0.77	0.90	0.78	0.90	0.77
$\texttt{SEA}_{RC+D}$	0.82	0.77	0.85	0.76	0.89	0.77	0.86	0.75	0.89	0.77	0.89	0.77	0.89	0.77
$\texttt{Hyp}_{RC}$	0.93	0.67	0.93	0.63	0.93	0.61	0.89	0.55	0.94	0.66	0.93	0.66	0.93	0.66
$\texttt{Hyp}_{RC+D}$	0.92	0.64	0.92	0.61	0.93	0.63	0.88	0.59	0.93	0.65	0.93	0.64	0.93	0.65
Air	0.65	0.66	0.64	0.66	0.65	0.65	0.65	0.61	0.64	0.65	0.62	0.60	0.67	0.68
PAKKD	0.56	0.64	0.73	0.57	0.80	0.50	-	-	0.80	0.63	0.80	0.62	0.80	0.62

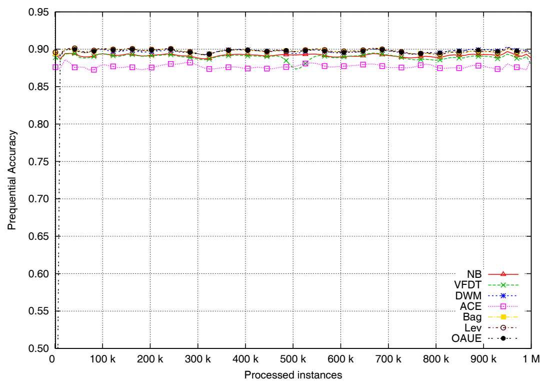


(a) Prequential accuracy

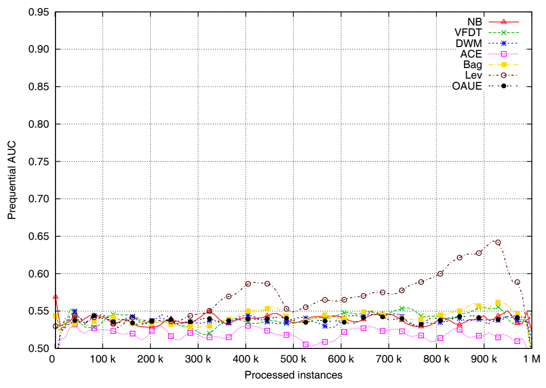


(b) Prequential AUC

Figure 6.11: Comparison of prequential accuracy and AUC on a data stream with sudden drifts and a balanced class ratio ($\texttt{SEA}_1$)

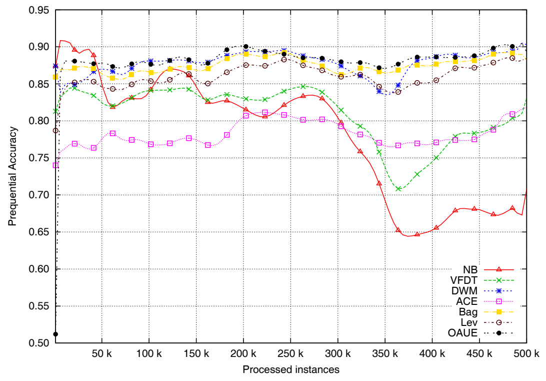


(a) Prequential accuracy

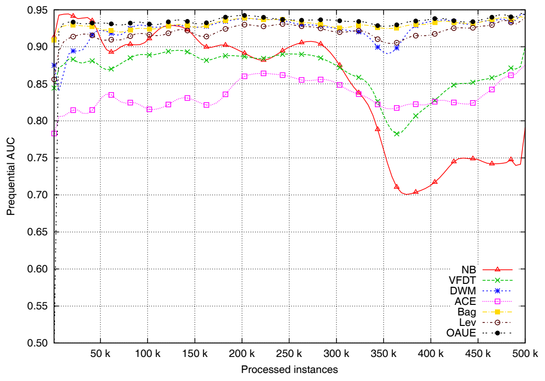


(b) Prequential AUC

Figure 6.12: Comparison of prequential accuracy and AUC on a data stream with sudden drifts and a 1:100 class imbalance ratio ($\texttt{SEA}_{100}$)

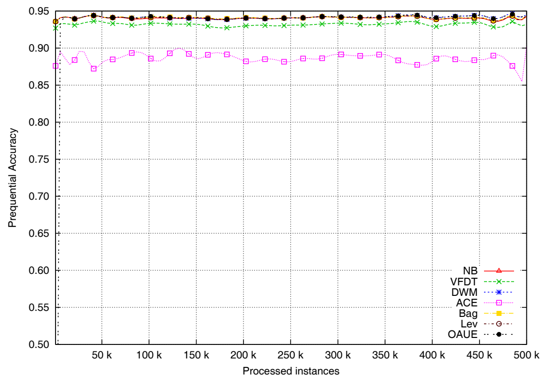


(a) Prequential accuracy

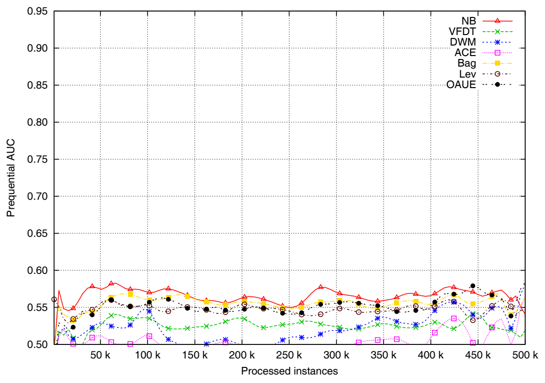


(b) Prequential AUC

Figure 6.13: Comparison of prequential accuracy and AUC on a data stream with incremental drift and a balanced class ratio ($\texttt{Hyp}_1$)

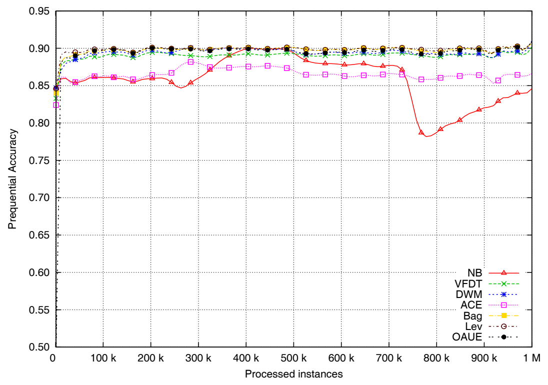


(a) Prequential accuracy

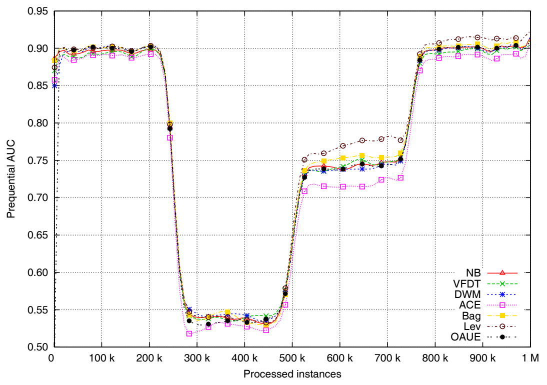


(b) Prequential AUC

Figure 6.14: Comparison of prequential accuracy and AUC on a data stream with incremental drift and a 1:100 class imbalance ratio ($\texttt{Hyp}_{100}$)

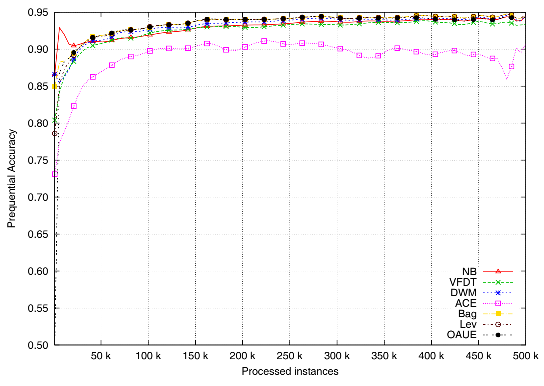


(a) Prequential accuracy

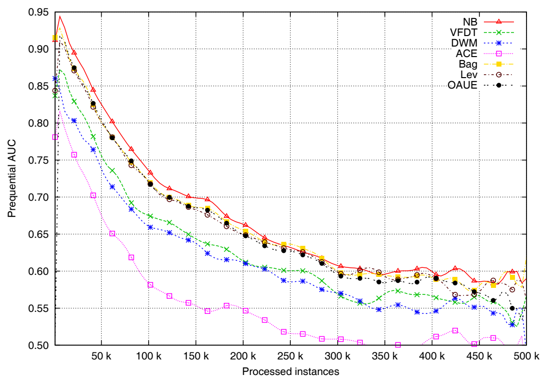


(b) Prequential AUC

Figure 6.15: Comparison of prequential accuracy and AUC on a data stream with sudden class ratio changes ($\texttt{SEA}_{RC}$)



(a) Prequential accuracy



(b) Prequential AUC

Figure 6.16: Comparison of prequential accuracy and AUC on a data stream with a gradual class ratio change ($\texttt{Hyp}_{RC}$)

A similar situation is visible on Figures 6.13 and 6.14, where the classifiers were subject to an ongoing slow incremental drift. When classes are balanced, the plots are almost identical, both in terms of shape and absolute values. However, when the class ratio is 1:100, the accuracy plot flattens and its average value rises, while the AUC plot still clearly differentiates classifiers and additionally its average value signals poor performance. These results coincide with the study performed by Huang and Ling, who have proven and experimentally shown that AUC is statistically more discriminant than accuracy, especially on imbalanced datasets [79].

Finally, Figures 6.15 and 6.16 depict classifier performance on data streams with class ratio changes. During sudden changes, all the tested classifiers, apart from NB, kept the same accuracy after each drift making the changes invisible on the performance plot. However, on the AUC plot, ratio changes are clearly visible providing valuable information about the ongoing processes in the stream. In fact, the absolute values of AUC hint the severity of class imbalance in a given moment in time. Similarly, during gradual ratio changes the accuracy plot does not signal any changes in the stream while on the AUC plot the drift is clearly visible. Plots for datasets containing simultaneously real and virtual drifts ($\texttt{SEA}_{RC+D}$, $\texttt{Hyp}_{RC+D}$) showcased identical properties — AUC plots depicted both real and virtual drifts while accuracy plots were only capable of showing real drifts. These scenarios clearly illustrate the advantages of prequential AUC as a measure for indicating class ratio changes.

Apart from analyzing single performance values and plots, we decided to check whether, for the analyzed algorithms and datasets, choosing AUC over accuracy would change the classifier ranking in terms of predictive performance. In order to verify this, we performed the non-parametric Friedman test [45]. The average ranks of the analyzed algorithms are presented in Table 6.13 (the lower the rank the better).

The null-hypothesis of the Friedman test, that there is no difference between the performance of all the tested algorithms, can be rejected both for accuracy and AUC at $p < 0.0001$. To verify which algorithms perform better than the other, we compute the critical difference chosen by the Bonferroni-Dunn post-hoc test [45] as $CD = 2.15$.

Table 6.13: Average algorithm ranks used in the Friedman tests

	NB	VFDT	DWM	ACE	Bag	Lev	OAUE
Accuracy	6.08	5.46	3.15	6.00	2.85	2.46	**2.00**
AUC	3.62	5.23	4.54	6.84	**2.23**	2.38	3.15

As we can see, in our experiments AUC and accuracy suggest different algorithm rankings. If we take into account accuracy, OAUE has the highest rank with Lev, Bag, and DWM achieving comparable results. This ranking is concordant with results on balanced datasets [32], discussed in Chapter 5. However, if we take into account AUC, we can see that the differentiation between algorithms is not so clear. In particular, NB achieves results comparable to Bag, Lev and OAUE. This promotion in the ranking is probably due to the fact that the Naive Bayes algorithm is usually capable of producing more diverse scores than decision trees. Nevertheless, the ranking based on prequential AUC seems to suggest that none of the analyzed online ensembles was prepared for handling imbalanced data or reacting to class distribution changes, since their results are comparable with single classifiers without any drift reaction mechanisms.

6.5 Conclusions

In case of static data, AUC is a useful measure for evaluating classifiers both on balanced and imbalanced classes. However, up till now it has not been sufficiently popular in data stream mining, due to its costly calculation. In this chapter, we introduced an new efficient method for calculating AUC incrementally with forgetting on evolving data streams.

The proposed algorithm, called prequential AUC, proved to be computationally efficient and useful for evaluating classifier performance over time. In particular, we have compared its processing speed against the κ statistic and positively assessed its visualizations and drift detection with that of prequential accuracy. Furthermore, we have shown that for stationary streams prequential AUC averaged over the entire stream is statistically consistent and comparably discriminating to AUC calculated using all examples at once. This comparison also involved AUC calculated in blocks, which proved inferior in terms of the degree of consistency and discriminancy.

Finally, experiments involving real and synthetic datasets have shown that prequential AUC is capable of correctly identifying poor classifier performance on imbalanced streams and detecting virtual drifts, i.e., changes in class ratio over time. As a result we have noticed that existing online ensembles are not prepared to deal with class imbalanced data and class distribution changes. Therefore, the topic of classifiers for drifting imbalanced data streams constitutes an interesting line of future research.

Chapter 7

Conclusions and Future Work

This dissertation concerned learning ensemble classifiers from concept-drifting data streams. As stated in Section 1.1, our main goal was to propose efficient block-based and online ensembles, which are capable of reacting to various types of concept drift. This involved analyzing the properties of both processing schemes as well as tackling the problem of accurate classifier evaluation in the context of real and virtual drift. In our opinion, the main goal as well as the aforementioned sub-tasks have been accomplished. To support this claim, we summarize the main contributions of this dissertation.

Reacting to various types of drift in block-based environments

In Chapter 3, we have analyzed the implications of introducing elements of incremental learning in block-based ensembles. As a result, we have proposed and experimentally validated the Accuracy Updated Ensemble (AUE), a block-based ensemble classifier designed to react to different types of concept drift. The main novel contribution of AUE is the introduction of incremental updating of component classifiers, which improves the ensemble's reactions to concept drifts, as well as reduces the impact of block sizes on the predictive performance of the ensemble. Incremental updates allow all ensemble members to adapt to the most recent concept simultaneously and, therefore, change the basic idea behind existing block-based algorithms. Such a hybrid approach allows AUE to react to various types of concept changes, such as sudden, gradual, recurring, short-term, and mixed drifts, which was one of the main goals of this thesis.

Additional contributions of AUE include the proposal of a new component weighting function and the investigation of different strategies concerning ensemble member updates. Our experiments have shown that, in terms of classification accuracy, all component classifiers in AUE should be updated after each incoming data block. Such an approach promotes the creation of strong component classifiers, which is concordant with results presented in [52]. This suggests that concept-drifting data streams provide natural diversity and the premise of weaklearnability does not fully apply to them.

To verify the performance of the proposed approach, we carried out an experimental study comparing AUE with 11 state-of-the-art data stream methods, including single classifiers, ensembles, and hybrid approaches, on a large set of streams simulating different drift scenarios. The experimental study has demonstrated that AUE can offer very high classification accuracy in environments with various types of changes, as well as in stationary environments. Notably, AUE provided best average classification accuracy out of all the tested algorithms, while proving to be faster and less memory consuming than competitive ensemble approaches.

Relations between block-based and online ensembles

The thesis has also contributed to the study of relations between drift reaction mechanisms of block-based and online ensembles. In particular, in Chapter 4 we verified if it is possible to transform block-based ensembles into online learners and proposed three general strategies for this purpose:

I) the use of a windowing technique which updates component classifier weights online,

II) the extension of the ensemble by an incremental classifier,

III) and the use of an online drift detector.

Experimental results have demonstrated that all three strategies can be beneficial to the performance of a block-based ensemble using both static and incremental base classifiers, however, not all of them are equally effective. In particular, we have observed that online component reweighting is the best transformation strategy in terms of average prequential accuracy, but is also the costliest one in terms of processing time. Moreover, we have noticed that elements of incremental learning are crucial to the improvement of classification accuracy in online environments. Finally, we have noticed that the transformation of a block-based ensemble to an online learner should be tailored to a given algorithm.

Reacting to various types of concept drift in online environments

Based on the analysis of ensemble transformation strategies, in Chapter 5 we have proposed a new incremental stream classifier, called Online Accuracy Updated Ensemble (OAUE). The main novelty of the OAUE algorithm is the proposal of a cost-effective component weighting function, which estimates a classifier's error on a window of last seen instances in constant time and memory without the need of remembering past examples. By weighting components according to their mean square error on the most recent predictions, OAUE aims at retaining the positive elements of AUE, while adding the capability of processing streams online. The fact that this capability was achieved with negligible overhead in terms of memory usage and processing time, makes OAUE a much better choice for streams with online labeling.

The predictive performance of OAUE, was experimentally compared with four representative online ensembles: the Adaptive Classifier Ensemble, Dynamic Weighted Majority, Online Bagging, and Leveraging Bagging. The obtained results have demonstrated that OAUE can offer very accurate predictions in online environments, regardless of the existence or type of drift. In particular, OAUE provided best average classification accuracy out of all the tested algorithms and was among the least time and memory consuming ones.

The demonstrated ability to react comparably well regardless of the type and severity of concept drift was also one of the main goals of this dissertation.

Prequential AUC as a measure for evaluating data stream classifiers

In Chapter 6, we have extended the analysis of adaptive ensembles to streams with class-distribution changes as a special type of virtual drift. To perform this analysis, we first surveyed existing methods for evaluating data stream classifiers, concentrating on their applicability to imbalanced data with changing class distributions. In particular, we showcased that existing methods for computing of the most popular evaluation measure for batch imbalanced data, the area under the ROC curve (AUC), are unfeasible for data streams. These limitations were overcome by introducing a novel algorithm for computing a time-oriented area under the ROC curve.

The proposed algorithm, called prequential AUC, proved to be computationally efficient and suitable for evaluating classifier performance over time. We have compared its visualizations, computation time, and drift detection accuracy with that of the currently most popular measures, prequential accuracy and the κ statistic. More importantly, we have shown that for stationary data, prequential AUC averaged over the entire stream is statistically consistent and comparably discriminating to AUC calculated using all examples at once. Finally, experiments involving real and synthetic datasets have demonstrated that prequential AUC is capable of correctly identifying poor classifier performance on imbalanced streams and detecting class distribution changes.

All these results have shown that, contrary to prior assumptions, AUC, which is one the most popular classifier evaluation measures in traditional data mining, can also be adapted to data stream processing. In the context of adaptive classifiers, by using prequential AUC we have noticed that existing online ensembles are not prepared to deal with class imbalanced data and class distribution changes.

Lines of further research

The above contributions open several directions for future studies. As it was signaled in Chapter 3, combinations of different types of drift constitute an interesting line of further research. Current works in the field of data stream classification concentrate mostly on single, separated changes. However, as it was shown during experiments involving block-based ensembles, combinations of gradual and sudden drifts are particularly difficult to react to. This is only one possible combination, and many others might be worth analyzing.

Concerning different types of drift, we have also seen that class distribution changes pose challenges to current data stream classifiers. To the best of our knowledge, class distribution changes have not been previously analyzed in the literature. However, in the context of analyzing multiple distributed data streams, such types of virtual drift can be seen in practice. For example, if several sensors in a monitored environment fail or malfunction, the classification algorithm might be left with only part of examples of a given class [63]. This in turn might be reflected in a change in the class distribution, which, as we have seen in Chapter 6, can easily deteriorate the predictive performance of a classifier. Therefore, we believe that methods for handling class distribution changes are also worth investigating.

Finally, the proposal and positive assessment of prequential AUC should open many new possibilities anticipated in the literature [98]. First of all, we believe that the proposed measure should facilitate the evaluation of classifiers learning from imbalanced streams and help visualize their performance over time. Furthermore, the computational feasibility of prequential AUC may lead to its application in classifier optimization. For example, many current adaptive ensembles use classification accuracy (or prediction errors) to weight or select ensemble members. With prequential AUC such operations could also be done using the ranking, rather than predictive, performance of a classifier. This is particularly interesting as AUC has been shown to be a better optimization criterion than accuracy for many scenarios in traditional data mining [79].

Appendix A

Experiment scripts

Below we present *excerpts* from experiment scripts used for classifier comparisons in this thesis. The aim of these excerpts is to give insight into basic algorithm and stream generator parameters. Full scripts, including additional experiments, source code, Java Runtime parameters, and precompiled MOA packages with the discussed algorithms, are available at http://www.cs.put.poznan.pl/dbrzezinski/software.php. Were appropriate, we will indicate direct links to software packages for concrete experiments.

A.1 Accuracy Updated Ensemble

Full scripts used for tests comparing of the Accuracy Updated Ensemble with other classifiers are available at: http://www.cs.put.poznan.pl/dbrzezinski/software/AUE2DatasetScripts.zip. The real-world datasets are available at: http://moa.cms.waikato.ac.nz/datasets/, while the synthetic datasets were generated by MOA. Listing A.1 presents the parameters of each generated dataset with non-default parameters described in comments. Descriptions of generator parameters are listed in the MOA Manual (http://moa.cms.waikato.ac.nz/documentation/).

Listing A.1: AUE experiment datasets

```
#RBF_ND (size=1M, drift=none)
generators.RandomRBFGenerator

#LED_ND (size=10M, noise=20%, drift=none)
generators.LEDGenerator -n 20

#Hyp_S (size=1M, classes=4, noise=5%, rotationSpeed=0.001,
   drift=gradual)
generators.HyperplaneGenerator -c 4 -k 5 -t 0.001

#Hyp_F (size=1M, classes=4, noise=5%, rotationSpeed=0.1,
   drift=gradual)
generators.HyperplaneGenerator -c 4 -k 5 -t 0.10
```

```
#Tree_S (size=1M, classes=4, driftPointInEachStream=200000,
   driftWidth=1, drift=sudden recurring)
ConceptDriftStream -s (generators.RandomTreeGenerator -c 4) -
   d (ConceptDriftStream -s (generators.RandomTreeGenerator -
   i 2 -r 2 -c 4) -d ... repeated 4 more times with different
    -i and -r... -a 90.0 -p 200000 -w 1 -r 2) -a 90.0 -p
   200000 -w 1

#SEA_S (size=1M, driftPointInEachStream=250000, driftWidth
   =50, drift=sudden)
ConceptDriftStream -s (generators.SEAGenerator -f 1) -d (
   ConceptDriftStream -s (generators.SEAGenerator -f 2) -d (
   ConceptDriftStream -s (generators.SEAGenerator -f 3)   -d
   (generators.SEAGenerator -f 4) -w 50 -p 250000 ) -w 50 -p
   250000 ) -w 50 -p 250000

#SEA_F (size=1M, driftPointInEachStream=100000, driftWidth
   =50, drift=sudden)
ConceptDriftStream -s (generators.SEAGenerator -i 1 -f 1 -b)
    -d (ConceptDriftStream -s (generators.SEAGenerator -i 2 -
   f 2 -b) ... repeated 7 more times with different -i and -f
   ... -w 50 -p 100000 ) -w 50 -p 100000

#RBF_GR (size=1M, classes=4, attrs=20, driftPointInEachStream
   =125000, driftWidth=250000, drift=gradual)
ConceptDriftStream -s (generators.RandomRBFGenerator -c 4 -a
   20) -d (ConceptDriftStream -s (generators.
   RandomRBFGenerator -r 5 -i 5 -c 4 -a 20 -n 25) -d ...
   repeated 4 more times with different -i and -r... -a 45.0
   -p 125000 -w 250000 -r 2) -a 45.0 -p 125000 -w 250000

#LED_M (size=1M, attrs=20, driftPointInStream=500000,
   driftWidth=10,  noiseAfterDrift=30%, drift=gradual drifts
   with a sudden concept change)
ConceptDriftStream -s (generators.LEDGeneratorDrift -d 3) -d
   (generators.LEDGeneratorDrift -d 2 -i 4 -n 30) -a 90.0 -p
   500000 -w 10

#RBF_B (size=1M, classes=4, attrs=20, driftPointInEachStream
   =249900, driftWidth=200, drift=blips)
```

```
ConceptDriftStream -s (generators.RandomRBFGenerator -c 4 -a
   20) -d (ConceptDriftStream -s (generators.
   RandomRBFGenerator -r 5 -i 5 -c 4 -a 20 -n 25) -d ...
   repeated 3 more times with different -i, -r, and -n... -a
   80.0 -p 249900 -w 200 -r 2) -a 80.0 -p 249900 -w 200

#Tree_F (size=100k, classes=6, driftPointInEachStream=3000,
   driftWidth=1, drift=sudden recurring)
ConceptDriftStream -s (generators.RandomTreeGenerator -c 6) -
   d (ConceptDriftStream -s (generators.RandomTreeGenerator -
   i 2 -r 2 -c 6) -d ...repeated 17 more times with different
    -i and -r... -a 90.0 -p 3000 -w 1 -r 2) -a 90.0 -p 3000 -
   w 1
```

Most of the tested algorithms, including the proposed Accuracy Updated Ensemble, are a part of the MOA framework and therefore their parameter meanings are also described in the aforementioned MOA manual. However the DWM and Learn++.NSE algorithms are only available as extensions which need to be added to MOA. They are already included in the precompiled version of MOA attached to the scripts for these testes on the authors website, but additional information can also be found at: https://sites.google.com/site/moaextensions/.

Listing A.2 presents algorithm parameters used for the experimental comparison in Chapter 3. It is worth noting taht in newer versions of MOA certain algorithm names might have changed and the presented script is compatible with the precompiled version of MOA attached to the scripts.

Listing A.2: AUE experiment algorithms

```
meta.AccuracyUpdatedEnsemble1 -n 10 -c 500
meta.AccuracyUpdatedEnsemble2 -n 10 -c 500
meta.AccuracyWeightedEnsemble -n 10
meta.OzaBagAdwin -l (trees.HoeffdingTree -e 1000 -g 100 -c
   0.01)
meta.LeveragingBag -l (trees.HoeffdingTree -e 1000-g 100 -c
   0.01)
trees.AdaHoeffdingOptionTree -o 10 -c 0.01
drift.SingleClassifierDrift -l (trees.HoeffdingTree -e 1000 -
   g 100 -c 0.01)
bayes.NaiveBayes
StaticWindow -l (trees.HoeffdingTree -e 1000 -g 100 -c 0.01)
   -w 500
meta.DynamicWeightedMajority -l (trees.HoeffdingTree -e 1000
   -g 100 -c 0.01) -p 500
meta.LearnNSE -l (trees.HoeffdingTree -e 1000 -g 100 -c 0.01)
    -p 500
```

A.2 Transformation strategies

The transformations of AUE$_{pre}$ and AWE are not a part of the MOA framework. However, their source code is available under: http://www.cs.put.poznan.pl/dbrzezinski/software/Strategies_20120808.zip. Datasets used for the comparison of these strategies were the same as for evaluation of the Online Accuracy Updated Ensemble, which is described in the following section. Listing A.3 prestens algorithm parameters.

Listing A.3: Transformation strategies

```
meta.AccuracyWeightedEnsemble -l (meta.WEKAClassifier -l (
   weka.classifiers.trees.J48 -C 0.25 -M 2)) -n 10.0 -r 10.0
   -c 1000
meta.IncrementalAWECandidate -l (meta.WEKAClassifier -l (weka
   .classifiers.trees.J48 -C 0.25 -M 2)) -n 10.0 -r 10.0 -c
   1000
meta.IncrementalAWEDetector -b (trees.HoeffdingTree -e
   2000000 -g 100 -c 0.01) -l (meta.WEKAClassifier -l (weka.
   classifiers.trees.J48 -C 0.25 -M 2)) -n 10.0 -r 10.0 -c
   1000
meta.AccuracyUpdatedEnsemble1 -l (trees.HoeffdingTree -e
   2000000 -g 100 -c 0.01) -n 10.0 -r 10.0 -c 1000
meta.IncrementalAUE1Candidate -l (trees.HoeffdingTree -e
   2000000 -g 100 -c 0.01) -n 10.0 -r 10.0 -c 1000
meta.IncrementalAUE1Detector -b (trees.HoeffdingTree -e
   2000000 -g 100 -c 0.01) -l (trees.HoeffdingTree -e 2000000
    -g 100 -c 0.01) -n 10.0 -r 10.0 -c 1000
meta.IncrementalAUE1Window -l (trees.HoeffdingTree -e 2000000
    -g 100 -c 0.01) -n 10.0 -r 10.0 -c 1000
meta.IncrementalAWEWindow -l (meta.WEKAClassifier -l (weka.
   classifiers.trees.J48 -C 0.25 -M 2)) -n 10.0 -r 10.0 -c
   1000
```

A.3 Online Accuracy Updated Ensemble

Full test scripts for the experimental comparison from Chapter 5 are available at: http://www.cs.put.poznan.pl/dbrzezinski/software/OAUEScripts_20130910.zip. The real-world datasets were downloaded from http://moa.cms.waikato.ac.nz/datasets/ and http://www.cse.fau.edu/~xqzhu/stream.html, while the synthetic datasets were generated by MOA.

Listings A.4 and A.5 present the parameters of datasets and algorithm, respectively. Datasets not described in this listing were identical with those used for AUE (Listing A.1). As with previous experiment scripts, descriptions of generator parameters are described in the MOA Manual.

Listing A.4: OAUE experiment datasets

```
#Hyp_F (size=1M, classes=4, noise=5%, rotationSpeed=0.25,
   drift=gradual)
generators.HyperplaneGenerator -c 4 -k 5 -t 0.25

#Wave (size=1M, driftingAttributes=20, instanceRandomSeed=5,
   noise=random, drift=none)
generators.WaveformGeneratorDrift -d 20 -i 5 -n

#SEA_G (size=1M, driftPointInEachStream=100250, driftWidth
   =50000, driftAngle=45deg, drift=gradual)
ConceptDriftStream -s (generators.SEAGenerator -i 1 -f 1 -b)
    -d (ConceptDriftStream -s (generators.SEAGenerator -i 2 -
   f 2 -b) -d ...repeated 6 more times with different -i and
   -f... -w 50000 -a 45.0 -p 100250 ) -w 50000 -a 45.0 -p
   100250

#Wave_M (size=500k, driftingAttributes=20/30,
   instanceRandomSeed=5/11, noise=random, driftPoint=250000,
   driftWidth=250000, driftAngle=45deg, drift=none)
ConceptDriftStream -s (generators.WaveformGeneratorDrift -d
   20 -i 5 -n) -d (generators.WaveformGeneratorDrift -d 30 -i
    11 -n) -a 45.0 -p 250000 -w 250000
```

Listing A.5: OAUE experiment algorithms

```
meta.OzaBagAdwin -l (trees.HoeffdingTree -e 2000000 -g 100 -c
    0.01) -s 10
meta.LeveragingBag -l (trees.HoeffdingTree -e 2000000 -g 100
   -c 0.01) -s 10
meta.DynamicWeightedMajority -l (trees.HoeffdingTree -e
   2000000 -g 100 -c 0.01) -e 10 -p 1000
meta.OnlineAccuracyUpdatedEnsemble -l (trees.HoeffdingTree -e
    2000000 -g 100 -c 0.01) -n 10 -w 1000
AdaptiveClassifiersEnsemble -n 10 -c 1000
```

A.4 Prequential AUC

Finally, scripts used for testing prequential AUC are available at: `http://www.cs.put.poznan.pl/dbrzezinski/software/AUCScripts_20150128.zip`. Since algorithm parameters did not differ from previous experiments, we only present excerpts from data generator scripts. It is worth noting taht in order to control the class imbalance ratio, data stream generators available in MOA had to be changed. Therefore, some parameters presented in Listing A.6 apply only to modified versions of generators available in the software package.

Listing A.6: Prequential AUC experiment datasets

```
#Hyp_1
generators.HyperplaneGenerator -c 2 -k 5 -t 0.001
#Hyp_10
generators.HyperplaneGenerator -c 2 -k 5 -t 0.001 -r 10 -b
#Hyp_100
generators.HyperplaneGenerator -c 2 -k 5 -t 0.001 -r 100 -b
#Hyp_RC
generators.HyperplaneGenerator -c 2 -k 5 -t 0.0 -r 100 -b -g
   500 -l 499000
#Hyp_RC+D
generators.HyperplaneGenerator -c 2 -k 5 -t 0.1 -r 100 -b -g
   500 -l 499000

#SEA_ND
generators.SEAGenerator -i 111 -f 1 -b

#SEA_1
ConceptDriftStream -s (generators.SEAGenerator -f 1) -d (
   ConceptDriftStream -s (generators.SEAGenerator -f 2) -d (
   ConceptDriftStream -s (generators.SEAGenerator -f 3)   -d
   (generators.SEAGenerator -f 4) -w 50 -p 250000 ) -w 50 -p
   250000 ) -w 50 -p 250000
#SEA_10
ConceptDriftStream -s (generators.SEAGenerator -f 1 -r 10 -b)
    -d (ConceptDriftStream -s (generators.SEAGenerator -f 2 -
   r 10 -b) -d (ConceptDriftStream -s (generators.
   SEAGenerator -f 3 -r 10 -b)   -d (generators.SEAGenerator
   -f 4 -r 10 -b) -w 50 -p 250000 ) -w 50 -p 250000 ) -w 50 -
   p 250000
#SEA_100
ConceptDriftStream -s (generators.SEAGenerator -f 1 -r 100 -b
   ) -d (ConceptDriftStream -s (generators.SEAGenerator -f 2
   -r 100 -b) -d (ConceptDriftStream -s (generators.
   SEAGenerator -f 3 -r 100 -b)   -d (generators.SEAGenerator
    -f 4 -r 100 -b) -w 50 -p 250000 ) -w 50 -p 250000 ) -w 50
    -p 250000
#SEA_RC
ConceptDriftStream -s (generators.SEAGenerator -f 1 -r 1 -b)
   -d (ConceptDriftStream -s (generators.SEAGenerator -f 1 -r
    100 -b) -d (ConceptDriftStream -s (generators.
   SEAGenerator -f 1 -r 10 -b)   -d (generators.SEAGenerator
   -f 1 -r 1 -b) -w 50 -p 250000 ) -w 50 -p 250000 ) -w 50 -p
    250000
#SEA_RC+D
ConceptDriftStream -s (generators.SEAGenerator -f 1 -r 1 -b)
   -d (ConceptDriftStream -s (generators.SEAGenerator -f 3 -r
    100 -b) -d (ConceptDriftStream -s (generators.
   SEAGenerator -f 2 -r 10 -b)   -d (generators.SEAGenerator
   -f 4 -r 1 -b) -w 50 -p 250000 ) -w 50 -p 250000 ) -w 50 -p
    250000
```

Appendix B

List of publications

This Appendix presents a list of author's publications related to research carried out in this dissertation. The papers are grouped according to Chapters which they relate to.

Chapter 3:

- Dariusz Brzezinski and Jerzy Stefanowski. Reacting to different types of concept drift: The accuracy updated ensemble algorithm. *IEEE Transactions on Neural Networks and Learning Systems*, 25:81–94, 2014.
- Dariusz Brzezinski and Jerzy Stefanowski. Accuracy updated ensemble for data streams with concept drift. In *Proceedings of the 6th International Conference on Hybrid Artificial Intelligence Systems*, volume 6679 of *Lecture Notes in Computer Science*, pages 155–163. Springer, 2011.

Chapters 4 and 5:

- Dariusz Brzezinski and Jerzy Stefanowski. Combining block-based and online methods in learning ensembles from concept drifting data streams. *Information Sciences*, 265:50–67, 2014.
- Dariusz Brzezinski and Jerzy Stefanowski. From Block-based Ensembles to Online Learners In Changing Data Streams: If- and How-To, *Proceedings of the 2012 ECML PKDD Workshop on Instant Interactive Data Mining*, available at: http://adrem.ua.ac.be/iid2012/.

Chapter 6:

- Dariusz Brzezinski and Jerzy Stefanowski. Prequential AUC for classifier evaluation and drift detection in evolving data streams. In *Proceedings of the 3rd International Workshop on New Frontiers in Mining Complex Patterns*, 2014.
- Georg Krempl, Indrė Žliobaitė, Dariusz Brzezinski, Eyke Hüllermeier, Mark Last, Vincent Lemaire, Tino Noack, Ammar Shaker, Sonja Sievi, Myra Spiliopoulou, and Jerzy Stefanowski. Open challenges for data stream mining research. *SIGKDD Explorations*, 16(1):1–10, 2014.

- Dariusz Brzezinski and Maciej Piernik. Adaptive XML Stream Classification using Partial Tree-edit Distance, In *Proceedings of ISMIS 2014, the 21st International Symposium on Methodologies for Intelligent Systems*, volume 8502 of *Lecture Notes in Computer Science*, pages 10–19. Springer, 2014.
- Dariusz Brzezinski and Jerzy Stefanowski. Classifiers for concept-drifting data streams: Evaluating things that really matter. In *Proceedings of the 1st International Workshop on Real-World Challenges for Data Stream Mining*, pages 10–14. Otto-von-Guericke University Magdeburg, 2013.

Bibliography

[1] Hanady Abdulsalam, David B. Skillicorn, and Patrick Martin. Classification using streaming random forests. *IEEE Trans. Knowl. Data Eng.*, 23(1):22–36, 2011.

[2] Charu C. Aggarwal. On change diagnosis in evolving data streams. *IEEE Trans. Knowl. Data Eng.*, 17(5):587–600, 2005.

[3] Charu C. Aggarwal, editor. *Data Streams - Models and Algorithms*, volume 31 of *Advances in Database Systems*. Springer, 2007.

[4] David W. Aha, Dennis F. Kibler, and Marc K. Albert. Instance-based learning algorithms. *Machine Learning*, 6:37–66, 1991.

[5] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–16. ACM, 2002.

[6] Manuel Baena-García, José del Campo-Ávila, Raúl Fidalgo, Albert Bifet, Ricard Gavaldá, and Rafael Morales-Bueno. Early drift detection method. In *Proceedings of the 4th International Workshop on Knowledge Discovery from Data Streams*, 2006.

[7] Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Inf.*, 1:290–306, 1972.

[8] Robert M. Bell, Yehuda Koren, and Chris Volinsky. The bellkor solution to the netflix prize, 2008. http://www.research.att.com/~volinsky/netflix/.

[9] Ricardo J. Bessa, Vladimiro Miranda, and João Gama. Entropy and correntropy against minimum square error in offline and online three-day ahead wind power forecasting. *IEEE Trans. Power Syst.*, 24(4):1657–1666, 2009.

[10] Alain Biem, Eric Bouillet, Hanhua Feng, Anand Ranganathan, Anton Riabov, Olivier Verscheure, Haris N. Koutsopoulos, Mahmood Rahmani, and Baris Güç. Real-time traffic information management using stream computing. *IEEE Data Eng. Bull.*, 33(2):64–68, 2010.

[11] Albert Bifet. *Adaptive learning and mining for data streams and frequent patterns.* PhD thesis, Universitat Politécnica de Catalunya, 2009.

[12] Albert Bifet and Eibe Frank. Sentiment knowledge discovery in twitter streaming data. In *Proceedings of the 13th Discovery Science International Conference*, volume 6332 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2010.

[13] Albert Bifet and Ricard Gavaldà. Kalman filters and adaptive windows for learning in data streams. In *Proceedings of the 9th Discovery Science International Conference*, volume 4265 of *Lecture Notes in Computer Science*, pages 29–40. Springer, 2006.

[14] Albert Bifet and Ricard Gavaldà. Learning from time-changing data with adaptive windowing. In *Proceedings of the 7th SIAM International Conference on Data Mining*. SIAM, 2007.

[15] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Moa: Massive online analysis. *Journal of Machine Learning Research*, 11:1601–1604, 2010.

[16] Albert Bifet, Geoffrey Holmes, and Bernhard Pfahringer. Leveraging bagging for evolving data streams. In *Proceedings of the 2010 European Conference on Machine Learning and Knowledge Discovery in Databases*, volume 6321 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2010.

[17] Albert Bifet, Geoffrey Holmes, Bernhard Pfahringer, and Ricard Gavaldà. Improving adaptive bagging methods for evolving data streams. In *Proceedings of the 1st Asian Conference on Machine Learning*, volume 5828 of *Lecture Notes in Computer Science*, pages 23–37. Springer, 2009.

[18] Albert Bifet, Geoffrey Holmes, Bernhard Pfahringer, and Ricard Gavaldà. Detecting sentiment change in twitter streaming data. In *Proceedings of the 2nd Workshop on Applications of Pattern Analysis*, volume 17 of *JMLR Proceedings*, pages 5–11. JMLR, 2011.

[19] Albert Bifet, Geoffrey Holmes, Bernhard Pfahringer, Richard Kirkby, and Ricard Gavaldà. New ensemble methods for evolving data streams. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 139–148. ACM, 2009.

[20] Albert Bifet and Richard Kirkby. Data stream mining: a practical approach. Technical report, The University of Waikato, August 2009.

[21] Albert Bifet and Richard Kirkby. *Massive Online Analysis*, August 2009.

[22] Albert Bifet, Jesse Read, Indre Zliobaite, Bernhard Pfahringer, and Geoff Holmes. Pitfalls in benchmarking data stream classification and how to avoid them. In *Proceedings of the 2013 European Conference on Machine Learning and Knowledge Discovery in Databases*, volume 8188 of *Lecture Notes in Computer Science*, pages 465–479. Springer, 2013.

[23] Daniel Billsus and Michael J. Pazzani. User modeling for adaptive news access. *User Model. User-Adapt. Interact.*, 10(2-3):147–180, 2000.

[24] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.

[25] Remco R. Bouckaert. Efficient AUC learning curve calculation. In *Proceedings of the 19th Australian Joint Conference on Artificial Intelligence*, volume 4304 of *Lecture Notes in Computer Science*, pages 181–191. Springer, 2006.

[26] Remco R. Bouckaert. Voting massive collections of bayesian network classifiers for data streams. In *Proceedings of the 19th Australian Joint Conference on Artificial Intelligence*, volume 4304 of *Lecture Notes in Computer Science*, pages 243–252. Springer, 2006.

[27] Max Bramer. *Principles of Data Mining.* Springer, 2007.

[28] Leo Breiman, Jerome H. Friedman, Charles J. Stone, and R. A. Olshen. *Classification and Regression Trees.* Wadsworth Statistics/Probability. Wadsworth, 1984.

[29] Dariusz Brzezinski. Mining data streams with concept drift. Master's thesis, Poznan University of Technology, Poznań, Poland, 2010.

[30] Dariusz Brzezinski and Jerzy Stefanowski. Accuracy updated ensemble for data streams with concept drift. In *Proceedings of the 6th International Conference on Hybrid Artificial Intelligence Systems*, volume 6679 of *Lecture Notes in Computer Science*, pages 155–163. Springer, 2011.

[31] Dariusz Brzezinski and Jerzy Stefanowski. Classifiers for concept-drifting data streams: Evaluating things that really matter. In *Proceedings of the 1st International Workshop on Real-World Challenges for Data Stream Mining*, pages 10–14. Otto-von-Guericke University Magdeburg, 2013.

[32] Dariusz Brzezinski and Jerzy Stefanowski. Combining block-based and online methods in learning ensembles from concept drifting data streams. *Inf. Sci.*, 265:50–67, 2014.

[33] Dariusz Brzezinski and Jerzy Stefanowski. Prequential AUC for classifier evaluation and drift detection in evolving data streams. In *Proceedings of the 3rd International Workshop on New Frontiers in Mining Complex Patterns*, 2014.

[34] Dariusz Brzezinski and Jerzy Stefanowski. Reacting to different types of concept drift: The accuracy updated ensemble algorithm. *IEEE Trans. Neural Netw. Learn. Syst.*, 25:81–94, 2014.

[35] Amit Chakrabarti, Graham Cormode, and Andrew McGregor. A near-optimal algorithm for computing the entropy of a stream. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 328–335. SIAM, 2007.

[36] Darryl Charles, Aphra Kerr, Moira McAlister, Michael McNeill, Julian Kücklich, Michaela M. Black, Adrian Moore, and Karl Stringer. Player-centred game design: Adaptive digital games. In *Proceedings of the 2005 Digital Games Research Conference*, 2005.

[37] Edith Cohen and Martin J. Strauss. Maintaining time-decaying stream aggregates. *J. Algorithms*, 59(1):19–36, 2006.

[38] Lior Cohen, Gil Avrahami, Mark Last, and Abraham Kandel. Info-fuzzy algorithms for mining dynamic data streams. *Appl. Soft Comput.*, 8(4):1283–1294, 2008.

[39] Magdalena Deckert. Batch weighted ensemble for mining data streams with concept drift. In *Proceedings of the 20th ISMIS International Symposium on Foundations of Intelligent Systems*, volume 6804 of *Lecture Notes in Computer Science*, pages 290–299. Springer, 2011.

[40] Magdalena Deckert. Incremental rule-based learners for handling concept drift: An overview. *Foundations of Computing and Decision Sciences*, 38(1):35–65, 2013.

[41] Magdalena Deckert and Jerzy Stefanowski. Comparing block ensembles for data streams with concept drift. In Mykola Pechenizkiy and Marek Wojciechowski, editors, *Workshop Proceedings of the 16th East European ADBIS Conference*, volume 185 of *Advances in Intelligent Systems and Computing*, pages 69–78. Springer, 2012.

[42] Magdalena Deckert and Jerzy Stefanowski. RILL: algorithm for learning rules from streaming data with concept drift. In *Proceedings of the 21st ISMIS International Symposium on Foundations of Intelligent Systems*, volume 8502 of *Lecture Notes in Computer Science*, pages 20–29. Springer, 2014.

[43] Sarah Jane Delany, Padraig Cunningham, and Alexey Tsymbal. A comparison of ensemble and case-base maintenance techniques for handling concept drift in spam filtering. In *Proceedings of the 19th International Florida Artificial Intelligence Research Society Conference*, pages 340–345. AAAI Press, 2006.

[44] Sarah Jane Delany, Padraig Cunningham, Alexey Tsymbal, and Lorcan Coyle. A case-based technique for tracking concept drift in spam filtering. *Knowl.-Based Syst.*, 18(4-5):187–195, 2005.

[45] Janez Demsar. Statistical comparisons of classifiers over multiple data sets. *J. Machine Learning Research*, 7:1–30, 2006.

[46] Thomas G. Dietterich. Ensemble methods in machine learning. In *Proceedings of the 1st International Workshop on Multiple Classifier Systems*, pages 1–15. Springer, 2000.

[47] Yi Ding and Xue Li. Time weight collaborative filtering. In *Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management*, pages 485–492. ACM, 2005.

[48] Gregory Ditzler and Robi Polikar. Incremental learning of concept drift from streaming imbalanced data. *IEEE Trans. Knowl. Data Eng.*, 25(10):2283–2301, 2013.

[49] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 71–80. ACM, 2000.

[50] Steve Donoho. Early detection of insider trading in option markets. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 420–429. ACM, 2004.

[51] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. Whiley, 2001.

[52] Ryan Elwell and Robi Polikar. Incremental learning of concept drift in nonstationary environments. *IEEE Trans. Neural Netw.*, 22(10):1517–1531, Oct. 2011.

[53] Wei Fan. Systematic data selection to mine concept-drifting data streams. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 128–137. ACM, 2004.

[54] Wei Fan, Yi an Huang, Haixun Wang, and Philip S. Yu. Active mining of data streams. In *Proceedings of the 4th SIAM International Conference on Data Mining*. SIAM, 2004.

[55] Tom Fawcett. Using rule sets to maximize ROC performance. In *Proceedings of the 1st IEEE International Conference on Data Mining*, pages 131–138, 2001.

[56] Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996.

[57] Francisco J. Ferrer-Troyano, Jesús S. Aguilar-Ruiz, and José Cristóbal Riquelme Santos. Discovering decision rules from numerical data streams. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 649–653. ACM, 2004.

[58] Peter A. Flach. ROC analysis. In Claude Sammut and Geoffrey I. Webb, editors, *Encyclopedia of Machine Learning*, pages 869–875. Springer, 2010.

[59] John H. Flavell. Piaget's legacy. *Psychological Science*, 7(4):200–203, 1996.

[60] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[61] Mohamed M. Gaber and João Gama. State of the art in data streams mining. Tutorial during the 8th European Conference on Machine Learning, 2007.

[62] João Gama. *Knowledge Discovery from Data Streams*. Chapman and Hall/CRC, 2010.

[63] João Gama and Mohamed M. Gaber. *Learning from Data Streams: Processing Techniques in Sensor Networks*. New generation computing. Springer, 2007.

[64] João Gama and Pedro Medas. Learning decision trees from dynamic data streams. *J. UCS*, 11(8):1353–1366, 2005.

[65] João Gama, Pedro Medas, Gladys Castillo, and Pedro P. Rodrigues. Learning with drift detection. In *Proceedings of the 17th Brazilian Symposium on Artificial Intelligence*, volume 3171 of *Lecture Notes in Computer Science*, page 286–295. Springer, 2004.

[66] João Gama, Pedro Medas, and Ricardo Rocha. Forest trees for on-line data. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 632–636. ACM, 2004.

[67] João Gama, Ricardo Rocha, and Pedro Medas. Accurate decision trees for mining high-speed data streams. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 523–528. ACM, 2003.

[68] João Gama and Pedro P. Rodrigues. Stream-based electricity load forecast. In *Proceedings of the 11th European Conference on Principles and Practice of Knowledge Discovery in Databases*, volume 4702 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2007.

[69] João Gama, Raquel Sebastião, and Pedro P. Rodrigues. On evaluating stream learning algorithms. *Machine Learning*, 90(3):317–346, 2013.

[70] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM Computing Surveys*, 46(4), 2014.

[71] Jing Gao, Wei Fan, Jiawei Han, and Philip 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*, pages 3–14. SIAM, 2007.

[72] Valerio Grossi and Franco Turini. Stream mining: a novel architecture for ensemble-based classification. *Knowl. Inf. Syst.*, 30(2):247–281, 2012.

[73] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, 2009.

[74] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[75] Michael Harries. Splice-2 comparative evaluation: Electricity pricing. Technical report, The University of South Wales, 1999.

[76] Haibo He and Yunqian Ma. *Imbalanced Learning: Foundations, Algorithms, and Applications.* Wiley-IEEE Press, 1st edition, 2013.

[77] Constantinos S. Hilas. Designing an expert system for fraud detection in private telecommunications networks. *Expert Syst. Appl.*, 36(9):11559–11569, 2009.

[78] Thomas Ryan Hoens and Nitesh V. Chawla. Learning in non-stationary environments with class imbalance. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 168–176. ACM, 2012.

[79] Jin Huang and Charles X. Ling. Using AUC and accuracy in evaluating learning algorithms. *IEEE Trans. Knowl. Data Eng.*, 17(3):299–310, 2005.

[80] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 97–106. ACM, 2001.

[81] Elena Ikonomovska, Suzana Loskovska, and Dejan Gjorgjevik. A survey of stream data mining. In *Proceedings of the 8th ETAI National Conference with International Participation*, 2007.

[82] Nathalie Japkowicz and Mohak Shah. *Evaluating Learning Algorithms: A Classification Perspective.* Cambridge University Press, 2011.

[83] Ruoming Jin and Gagan Agrawal. Efficient decision tree construction on streaming data. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 571–576. ACM, 2003.

[84] Matthew T. Karnick, Metin Ahiskali, Michael Muhlbaier, and Robi Polikar. Learning concept drift in nonstationary environments using an ensemble of classifiers based approach. In *Proceedings of the 2008 International Joint Conference on Neural Networks*, pages 3455–3462. IEEE, 2008.

[85] Ioannis Katakis, Grigorios Tsoumakas, Evangelos Banos, Nick Bassiliades, and Ioannis P. Vlahavas. An adaptive personalized news dissemination system. *J. Intell. Inf. Syst.*, 32(2):191–212, 2009.

[86] Mark G. Kelly, David J. Hand, and Niall M. Adams. The impact of changing populations on classifier performance. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 367–371. ACM, 1999.

[87] Richard Kirkby. *Improving Hoeffding Trees.* PhD thesis, Department of Computer Science, University of Waikato, 2007.

[88] Ralf Klinkenberg and Thorsten Joachims. Detecting concept drift with support vector machines. In *Proceedings of the 17th International Conference on Machine Learning*, pages 487–494. Morgan Kaufmann, 2000.

[89] Milosz Kmieciak and Jerzy Stefanowski. Handling sudden concept drift in Enron message data streams. *Control and Cybernetics*, 40(3):667–695, 2011.

[90] Ron Kohavi and Clayton Kunz. Option decision trees with majority votes. In *Proceedings of the 14th International Conference on Machine Learning*, pages 161–169. Morgan Kaufmann, 1997.

[91] J. Zico Kolter and Marcus A. Maloof. Dynamic weighted majority: An ensemble method for drifting concepts. *J. Machine Learning Research*, 8:2755–2790, Dec. 2007.

[92] Yehuda Koren. Collaborative filtering with temporal dynamics. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 447–456. ACM, 2009.

[93] Yehuda Koren. Collaborative filtering with temporal dynamics. *Commun. ACM*, 53(4):89–97, 2010.

[94] Petr Kosina and João Gama. Handling time changing data with adaptive very fast decision rules. In *Proceedings of the 2012 European Conference on Machine Learning and Knowledge Discovery in Databases*, volume 7523 of *Lecture Notes in Computer Science*, pages 827–842. Springer, 2012.

[95] Petr Kosina and João Gama. Very fast decision rules for multi-class problems. In *Proceedings of the 2012 ACM Symposium on Applied Computing*, pages 795–800. ACM, 2012.

[96] Petr Kosina and João Gama. Very fast decision rules for classification in data streams. *Data Min. Knowl. Discov.*, 29(1):168–202, 2015.

[97] Petr Kosina, João Gama, and Raquel Sebastião. Drift severity metric. In *Proceedings of the 19th European Conference on Artificial Intelligence*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 1119–1120. IOS Press, 2010.

[98] Georg Krempl, Indrė Žliobaitė, Dariusz Brzezinski, Eyke Hüllermeier, Mark Last, Vincent Lemaire, Tino Noack, Ammar Shaker, Sonja Sievi, Myra Spiliopoulou, and Jerzy Stefanowski. Open challenges for data stream mining research. *SIGKDD Explorations*, 16(1):1–10, 2014.

[99] Ludmila I. Kuncheva. Classifier ensembles for changing environments. In *Proceedings of the 5th International Workshop on Multiple Classifier Systems*, volume 3077 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2004.

[100] Ludmila I. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms.* Wiley-Interscience, 2004.

[101] Ludmila I. Kuncheva. Classifier ensembles for detecting concept change in streaming data: Overview and perspectives. In *Proceedings of the 2nd SUEMA Workshop*, pages 5–10, 2008.

[102] Terran Lane and Carla E. Brodley. Temporal sequence learning and data reduction for anomaly detection. *ACM Trans. Inf. Syst. Secur.*, 2(3):295–331, 1999.

[103] Mark Last. Online classification of nonstationary data streams. *Intell. Data Anal.*, 6(2):129–147, 2002.

[104] Andreas D. Lattner, Andrea Miene, Ubbo Visser, and Otthein Herzog. Sequential pattern mining for situation and behavior prediction in simulated robotic soccer. In *Proceedings of RoboCup 2005: Robot Soccer World Cup IX*, volume 4020 of *Lecture Notes in Computer Science*, pages 118–129. Springer, 2005.

[105] Yan-Nei Law and Carlo Zaniolo. An adaptive nearest neighbor classification algorithm for data streams. In *Proceedings of the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases*, volume 3721 of *Lecture Notes in Computer Science*, pages 108–120. Springer, 2005.

[106] Mihai Lazarescu, Svetha Venkatesh, and Hung Hai Bui. Using multiple windows to track concept drift. *Intell. Data Anal.*, 8(1):29–59, 2004.

[107] Daniel Leite, Pyramo Costa Jr., and Fernando Gomide. Evolving granular neural network for semi-supervised data stream classification. In *Proceedings of the 2010 International Joint Conference on Neural Networks*, pages 1–8. IEEE, 2010.

[108] Daniel Leite, Pyramo Costa Jr., and Fernando Gomide. Evolving granular neural networks from fuzzy data streams. *Neural Networks*, 38:1–16, 2013.

[109] Pei-Pei Li, Xuegang Hu, and Xindong Wu. Mining concept-drifting data streams with multiple semi-random decision trees. In *Proceedings of the 4th International Conference on Advanced Data Mining and Applications*, volume 5139 of *Lecture Notes in Computer Science*, pages 733–740. Springer, 2008.

[110] Ryan Lichtenwalter and Nitesh V. Chawla. Adaptive methods for classification in arbitrarily imbalanced and drifting data streams. In *Proceedings of the PAKDD 2009 International Workshops on New Frontiers in Applied Data Mining*, volume 5669 of *Lecture Notes in Computer Science*, pages 53–75. Springer, 2009.

[111] Patrick Lindstrom, Sarah Jane Delany, and Brian Mac Namee. Handling concept drift in a text data stream constrained by high labelling cost. In *Proceedings of the 23rd International Florida Artificial Intelligence Research Society Conference*. AAAI Press, 2010.

[112] Nick Littlestone and Manfred K. Warmuth. The weighted majority algorithm. *Inf. Comput.*, 108(2):212–261, 1994.

[113] Konrad Lorincz, David J. Malan, Thaddeus R. F. Fulford Jones, Alan Nawoj, Antony Clavel, Victor Shnayder, Geoffrey Mainland, Matt Welsh, and Steve Moulton. Sensor networks for emergency response: Challenges and opportunities. *IEEE Pervasive Comput.*, 3(4):16–23, 2004.

[114] F. Lotte, M. Congedo, A. Laccuyer, F. Lamarche, and B. Arnaldi. A review of classification algorithms for eeg-based brain-computer interfaces. *J. Neural Eng.*, 4(2):R1, 2007.

[115] Oded Maimon and Lior Rokach, editors. *Data Mining and Knowledge Discovery Handbook, 2nd ed.* Springer, 2010.

[116] Markos Markou and Sameer Singh. Novelty detection: a review - part 1: statistical approaches. *Signal Process.*, 83(12):2481–2497, 2003.

[117] Frank J. Massey. The Kolmogorov-Smirnov test for goodness of fit. *J. Am. Stat. Assoc.*, 46(253):68–78, 1951.

[118] Mohammad M. Masud, Qing Chen, Latifur Khan, Charu C. Aggarwal, Jing Gao, Jiawei Han, Ashok N. Srivastava, and Nikunj C. Oza. Classification and adaptive novel class detection of feature-evolving data streams. *IEEE Trans. Knowl. Data Eng.*, 25(7):1484–1497, 2013.

[119] Mohammad M. Masud, Qing Chen, Latifur Khan, Charu C. Aggarwal, Jing Gao, Jiawei Han, and Bhavani M. Thuraisingham. Addressing concept-evolution in concept-drifting data streams. In *Proceedings of the 10th IEEE International Conference on Data Mining*, pages 929–934. IEEE Computer Society, 2010.

[120] Mohammad M. Masud, Jing Gao, Latifur Khan, Jiawei Han, and Bhavani M. Thuraisingham. A practical approach to classify evolving data streams: Training with limited amount of labeled data. In *Proceedings of the 8th IEEE International Conference on Data Mining*, pages 929–934. IEEE Computer Society, 2008.

[121] Mohammad M. Masud, Jing Gao, Latifur Khan, Jiawei Han, and Bhavani M. Thuraisingham. A multi-partition multi-chunk ensemble technique to classify concept-drifting data streams. In *Proceedings of the 13th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, volume 5476 of *Lecture Notes in Computer Science*, pages 363–375. Springer, 2009.

[122] Pawel Matuszyk, Georg Krempl, and Myra Spiliopoulou. Correcting the usage of the hoeffding inequality in stream mining. In *Proceedings of the 12th International Symposium on Advances in Intelligent Data Analysis*, volume 8207 of *Lecture Notes in Computer Science*, pages 298–309. Springer, 2013.

[123] Oleksiy Mazhelis and Seppo Puuronen. Comparing classifier combining techniques for mobile-masquerader detection. In *Proceedings of the the 2nd International Conference on Availability, Reliability and Security*, pages 465–472. IEEE Computer Society, 2007.

[124] João Mendes-Moreira, Carlos Soares, Alípio Mário Jorge, and Jorge Freire de Sousa. The effect of varying parameters and focusing on bus travel time prediction. In *Proceedings of the 13th Pacific-Asia Conference on Advances in Knowledge Discovery*

and Data Mining, volume 5476 of *Lecture Notes in Computer Science*, pages 689–696. Springer, 2009.

[125] Leandro L. Minku, Allan P. White, and Xin Yao. The impact of diversity on online ensemble learning in the presence of concept drift. *IEEE Trans. Knowl. Data Eng.*, 22(5):730–742, May 2010.

[126] Leandro L. Minku and Xin Yao. DDD: A new ensemble approach for dealing with concept drift. *IEEE Trans. Knowl. Data Eng.*, 24(4):619–633, Apr. 2012.

[127] Tom M. Mitchell. *Machine learning.* McGraw Hill series in computer science. McGraw-Hill, 1997.

[128] Vahid Moosavi and Ludger Hovestadt. Modeling urban traffic dynamics in coexistence with urban data streams. In *Proceedings of the 2nd ACM SIGKDD International Workshop on Urban Computing.* ACM, 2013.

[129] H. Mouss, D. Mouss, N. Mouss, and L. Sefouhi. Test of page-hinkley, an approach for fault detection in an agro-alimentary production system. In *Proceedings of the 5th Asian Control Conference*, volume 2, pages 815–818, 2004.

[130] Kyosuke Nishida, Koichiro Yamauchi, and Takashi Omori. ACE: Adaptive classifiers-ensemble system for concept-drifting environments. In *Proceedings of the 6th International Workshop on Multiple Classifier Systems*, volume 3541 of *Lecture Notes in Computer Science*, pages 176–185. Springer, 2005.

[131] Nikunj C. Oza. *Online Ensemble Learning.* PhD thesis, The University of California, Berkeley, CA, Sep 2001.

[132] Nikunj C. Oza and Stuart J. Russell. Experimental comparisons of online and batch versions of bagging and boosting. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 359–364, 2001.

[133] E. S. Page. Continuous inspection schemes. *Biometrika*, 41(1/2):100–115, 1954.

[134] PAKDD 2009 data mining competition. URL http://sede.neurotech.com.br:443/PAKDD2009/.

[135] Animesh Patcha and Jung-Min Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51(12):3448–3470, 2007.

[136] Mykola Pechenizkiy, Jorn Bakker, Indre Zliobaite, Andriy Ivannikov, and Tommi Kärkkäinen. Online mass flow prediction in CFB boilers with explicit detection of sudden concept drift. *SIGKDD Explorations*, 11(2):109–116, 2009.

[137] R. Pelossof, M. Jones, I. Vovsha, and C. Rudin. Online coordinate boosting. In *Proceedings of the 12th IEEE International Conference on Computer Vision*, pages 1354–1361. IEEE Computer Society, 2009.

[138] Bernhard Pfahringer, Geoffrey Holmes, and Richard Kirkby. New options for hoeffding trees. In *Proceedings of the 20th Australian Joint Conference on Artificial Intelligence*, volume 4830 of *Lecture Notes in Computer Science*, pages 90–99. Springer, 2007.

[139] Michael J. Procopio, Jane Mulligan, and Gregory Z. Grudic. Learning terrain segmentation with classifier ensembles for autonomous robot navigation in unstructured environments. *J. Field Robotics*, 26(2):145–175, 2009.

[140] Foster J. Provost and Pedro Domingos. Tree induction for probability-based ranking. *Machine Learning*, 52(3):199–215, 2003.

[141] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[142] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[143] S. W. Roberts. Control chart tests based on geometric moving averages. *Technometrics*, 42(1):97–101, 1956.

[144] Gordon J. Ross, Niall M. Adams, Dimitris K. Tasoulis, and David J. Hand. Exponentially weighted moving average charts for detecting concept drift. *Pattern Recognit. Lett.*, 33(2):191–198, 2012.

[145] Leszek Rutkowski, Maciej Jaworski, Lena Pietruczuk, and Piotr Duda. The CART decision tree for mining data streams. *Inf. Sci.*, 266:1–15, 2014.

[146] Leszek Rutkowski, Maciej Jaworski, Lena Pietruczuk, and Piotr Duda. Decision trees for mining data streams based on the gaussian approximation. *IEEE Trans. Knowl. Data Eng.*, 26(1):108–119, 2014.

[147] Leszek Rutkowski, Lena Pietruczuk, Piotr Duda, and Maciej Jaworski. Decision trees for mining data streams based on the McDiarmid's bound. *IEEE Trans. Knowl. Data Eng.*, 25(6):1272–1279, 2013.

[148] Marcos Salganicoff. Tolerating concept and sampling shift in lazy learning using prediction error context switching. *Artif. Intell. Rev.*, 11(1-5):133–155, 1997.

[149] Jeffrey C. Schlimmer and Richard H. Granger. Incremental learning from noisy data. *Machine Learning*, 1(3):317–354, 1986.

[150] Ammar Shaker and Eyke Hüllermeier. Recovery analysis for adaptive learning from non-stationary data streams. In *Proceedings of the 8th International Conference on Computer Recognition Systems*, volume 226 of *Advances in Intelligent Systems and Computing*, pages 289–298. Springer, 2013.

[151] Ammar Shaker and Eyke Hüllermeier. Recovery analysis for adaptive learning from non-stationary data streams: Experimental design and case study. *Neurocomputing*, 150:250–264, 2015.

[152] Jasmina Smailovic, Miha Grcar, Nada Lavrac, and Martin Znidarsic. Stream-based active learning for sentiment analysis in the financial domain. *Inf. Sci.*, 285:181–203, 2014.

[153] Myra Spiliopoulou and Georg Krempl. Tutorial on mining multiple threads of streaming data. In *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining*, PAKDD, 2013.

[154] Frank-Florian Steege and Horst-Michael Groß. Comparison of long-term adaptivity for neural networks. In *Proceedings of the 22nd International Conference on Artificial Neural Networks*, volume 7553 of *Lecture Notes in Computer Science*, pages 50–57. Springer, 2012.

[155] W. Nick Street and YongSeog Kim. A streaming ensemble algorithm (SEA) for large-scale classification. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 377–382, 2001.

[156] Robert Szewczyk, Eric Osterweil, Joseph Polastre, Michael Hamilton, Alan M. Mainwaring, and Deborah Estrin. Habitat monitoring with sensor networks. *Commun. ACM*, 47(6):34–40, 2004.

[157] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining.* Addison Wesley, May 2005.

[158] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Kenny Lau, Celia Oakley, Mark Palatucci, Vaughan Pratt, Pascal Stang, Sven Strohb, Cedric Dupont, Lars erik Jendrossek, Christian Koelen, Charles Markey, Carlo Rummel, Joe Van Niekerk, Eric Jensen, Gary Bradski, Bob Davies, Scott Ettinger, Adrian Kaehler, Ara Nefian, and Pamela Mahoney. The robot that won the DARPA grand challenge. *J. Field Rob.*, 23:661–692, 2006.

[159] Alexey Tsymbal. The problem of concept drift: definitions and related works. Technical report, Dept. Comput. Sci., Trinity College Dublin, 2004.

[160] Alexey Tsymbal, Mykola Pechenizkiy, Padraig Cunningham, and Seppo Puuronen. Dynamic integration of classifiers for handling concept drift. *Information Fusion*, 9(1):56–68, 2008.

[161] Ranga Raju Vatsavai, Olufemi A. Omitaomu, Joao Gama, Nitesh V. Chawla, Mohamed Medhat Gaber, and Auroop R. Ganguly. Knowledge discovery from sensor data (SensorKDD). *SIGKDD Explorations*, 10(2):68–73, 2008.

[162] Boyu Wang and Joelle Pineau. Online ensemble learning for imbalanced data streams. *CoRR*, abs/1310.8004, 2013.

[163] Haixun Wang, Wei Fan, Philip S. Yu, and Jiawei Han. Mining concept-drifting data streams using ensemble classifiers. In *Proceedings of the 9th ACM SIGKDD*

International Conference on Knowledge Discovery and Data Mining, pages 226–235. ACM, 2003.

[164] Hua Wang, Feiping Nie, Heng Huang, Jingwen Yan, Sungeun Kim, Shannon L. Risacher, Andrew J. Saykin, and Li Shen. High-order multi-task feature learning to identify longitudinal phenotypic markers for alzheimer's disease progression prediction. In *Proceedings of a meeting held at the 26th Annual Conference on Neural Information Processing Systems*, pages 1286–1294, 2012.

[165] Weka Machine Learning Project. URL http://www.cs.waikato.ac.nz/˜ml/weka.

[166] Gerhard Widmer and M. Kubat. Learning in the presence of concept drift and hidden contexts. In *Machine Learning*, pages 69–101, 1996.

[167] Gerhard Widmer and Miroslav Kubat. Effective learning in dynamic environments by explicit context tracking. In *Proceedings of the 1993 European Conference on Machine Learning*, volume 667 of *Lecture Notes in Computer Science*, pages 227–243. Springer, 1993.

[168] B. Widrow and M.A. Lehr. 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. *Proceedings of the IEEE*, 78(9):1415–1442, 1990.

[169] Shaomin Wu, Peter A. Flach, and Cèsar Ferri Ramirez. An improved model selection heuristic for AUC. In *Proceedings of the 8th European Conference on Machine Learning*, volume 4701 of *Lecture Notes in Computer Science*, pages 478–489. Springer, 2007.

[170] Yuang Yao, Lin Feng, and Feng Chen. Concept drift visualization. *Journal of Information and Computational Science*, 10(10):3021–3029, 2013.

[171] Shin-ichi Yoshida, Kohei Hatano, Eiji Takimoto, and Masayuki Takeda. Adaptive online prediction using weighted windows. *IEICE Transactions*, 94-D(10):1917–1923, 2011.

[172] Indrė Žliobaitė. Combining time and space similarity for small size learning under concept drift. In *Proceedings of the 18th ISMIS International Symposium on Foundations of Intelligent Systems*, volume 5722 of *Lecture Notes in Computer Science*, pages 412–421. Springer, 2009.

[173] Indrė Žliobaitė. Instance selection method (fish) for classifier training under concept drift. Technical report, Vilnius University, Faculty of Mathematics and Informatic, 2009.

[174] Indrė Žliobaitė. Learning under concept drift: an overview. Technical report, Vilnius University, Faculty of Mathematics and Informatic, 2009.

[175] Indrė Žliobaitė. *Adaptive training set formation.* PhD thesis, Vilnius University, 2010.

[176] Indrė Žliobaitė. Controlled permutations for testing adaptive learning models. *Knowl. Inf. Syst.*, pages 1–14, 2013.

[177] Indrė Žliobaitė, Albert Bifet, Bernhard Pfahringer, and Geoff Holmes. Active learning with evolving streaming data. In *Proceedings of the 2011 European Conference on Machine Learning and Knowledge Discovery in Databases*, volume 6913 of *Lecture Notes in Computer Science*, pages 597–612. Springer, 2011.

[178] Indrė Žliobaitė, Albert Bifet, Jesse Read, Bernhard Pfahringer, and Geoff Holmes. Evaluation methods and decision theory for classification of streaming data with temporal dependence. *Machine Learning*, 98:455–482, 2015.

[179] Indrė Žliobaitė, Marcin Budka, and Frederic Stahl. Towards cost-sensitive adaptation: When is it worth updating your predictive model? *Neurocomputing*, 150:240–249, 2015.

Poznan University of Technology
Faculty of Computing Science
Institute of Computing Science

Typeset using LaTeX in Latin Modern.

BibTeX:

```
@PHDTHESIS{BrzezPhd2015,
    author  = {Dariusz Brzezinski},
    title   = {Block-based and Online Ensembles for Concept-drifting Data Streams},
    school  = {Poznan University of Technology},
    address = {Poznan, Poland},
    year    = {2015}
}
```