Mining Data Streams with Concept Drift

Thesis · September 2010

DOI: 10.13140/RG.2.1.4634.6086

CITATIONS
30

READS
259

1 author:

Dariusz Brzezinski
Poznan University of Technology
30 PUBLICATIONS **477** CITATIONS

SEE PROFILE

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

Master's thesis

MINING DATA STREAMS WITH CONCEPT DRIFT

Dariusz Brzeziński

Supervisor
Jerzy Stefanowski, PhD Dr Habil.

Poznań, 2010

Tutaj przychodzi karta pracy dyplomowej;
oryginał wstawiamy do wersji dla archiwum PP, w pozostałych kopiach wstawiamy ksero.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Thesis structure . . . 2
- 1.3 Acknowledgments . . . 2

2 Mining data streams **3**
- 2.1 Data streams . . . 3
- 2.2 Concept drift . . . 5
- 2.3 Data stream mining . . . 6
  - 2.3.1 Online learning . . . 7
  - 2.3.2 Forgetting mechanisms . . . 7
  - 2.3.3 Taxonomy of methods . . . 8
- 2.4 Applications . . . 9
  - 2.4.1 Monitoring systems . . . 9
  - 2.4.2 Personal assistance . . . 10
  - 2.4.3 Decision support . . . 10
  - 2.4.4 Artificial intelligence . . . 11

3 Single classifier approaches **13**
- 3.1 Traditional learners . . . 13
- 3.2 Windowing techniques . . . 14
  - 3.2.1 Weighted windows . . . 15
  - 3.2.2 FISH . . . 15
  - 3.2.3 ADWIN . . . 17
- 3.3 Drift detectors . . . 18
  - 3.3.1 DDM . . . 18
  - 3.3.2 EDDM . . . 20
- 3.4 Hoeffding trees . . . 20

4 Ensemble approaches **23**
- 4.1 Ensemble strategies for changing environments . . . 23
- 4.2 Streaming Ensemble Algorithm . . . 25
- 4.3 Accuracy Weighted Ensemble . . . 26
- 4.4 Hoeffding option trees and ASHT Bagging . . . 28
- 4.5 Accuracy Diversified Ensemble . . . 29

5 MOA framework **33**
- 5.1 Stream generation and management . . . 33

5.2 Classification methods . . . 34
5.3 Evaluation procedures . . . 35
5.3.1 Holdout . . . 35
5.3.2 Interleaved Test-Then-Train . . . 35
5.3.3 Data Chunks . . . 36

6 Experimental evaluation **37**
6.1 Algorithms . . . 37
6.2 Data sets . . . 38
6.3 Experimental environment . . . 39
6.4 Results . . . 39
6.4.1 Time analysis . . . 39
6.4.2 Memory usage . . . 41
6.4.3 Classification accuracy . . . 42
6.5 Remarks . . . 45

7 Conclusions **47**

A Implementation details **49**
A.1 MOA Installation . . . 49
A.2 Attribute filtering . . . 50
A.3 Data chunk evaluation . . . 50
A.4 Accuracy Weighted Ensemble . . . 51
A.5 Accuracy Diversified Ensemble . . . 51

B Additional figures **53**

Bibliography **71**

Streszczenie **77**

Chapter 1

Introduction

1.1 Motivation

In today's information society, computer users are used to gathering and sharing data anytime and anywhere. This concerns applications such as social networks, banking, telecommunication, health care, research, and entertainment, among others. As a result, a huge amount of data related to all human activity is gathered for storage and processing purposes. These data sets may contain interesting and useful knowledge represented by hidden patterns, but due to the volume of the gathered data it is impossible to manually extract that knowledge. That is why *data mining* and *knowledge discovery* methods have been proposed to automatically acquire interesting, non-trivial, previously unknown and ultimately understandable patterns from very large data sets [26, 14]. Typical data mining tasks include association mining, classification, and clustering, which all have been perfected for over two decades.

A recent report [35] estimated that the digital universe in 2007 was 281 billion gigabytes large and it is forecast that it will reach 5 times that size until 2011. The same report states that by 2011 half of the produced data will not have a permanent home. This is partially due to a new class of emerging applications - applications in which data is generated at very high rates in the form of transient *data streams*. Data streams can be viewed as a sequence of relational tuples (e.g., call records, web page visits, sensor readings) that arrive continuously at time-varying, possibly unbound streams. Due to their speed and size it is impossible to store them permanently [45]. Data stream application domains include network monitoring, security, telecommunication data management, web applications, and sensor networks. The introduction of this new class of applications has opened an interesting line of research problems including novel approaches to knowledge discovery called *data stream mining*.

Current research in data mining is mainly devoted to static environments, where patterns hidden in data are fixed and each data tuple can be accessed more than once. The most popular data mining task is classification, defined as generalizing a known structure to apply it to new data [26]. Traditional classification techniques give great results in static environments however, they fail to successfully process data streams because of two factors: their overwhelming volume and their distinctive feature - *concept drift*. Concept drift is a term used to describe changes in the learned structure that occur over time. These changes mainly involve substitutions of one classification task with another, but also include steady trends and minor fluctuations of the underlying probability distributions [54]. For most traditional classifiers the occurrence of concept drift leads to a drastic drop in classification accuracy. That is why recently, new classification algorithms dedicated to data streams have been proposed.

The recognition of concept drift in data streams has led to sliding-window approaches that model a forgetting process, which allows to limit the number of processed data and to react to changes. Different approaches to mining data streams with concept drift include instance selection methods, drift detection, ensemble classifiers, option trees and using Hoeffding boundaries to estimate classifier performance.

Recently, a framework called *Massive Online Analysis* (MOA) for implementing algorithms and running experiments on evolving data streams has been developed [12, 11]. It includes a collection of offline and online data stream mining methods as well as tools for their evaluation. MOA is a new environment that can facilitate and consequently accelerate the development of new time-evolving stream classifiers.

The aim of this thesis is to review and compare single classifier and ensemble approaches to data stream mining. We test time and memory costs, as well as classification accuracy, of representative algorithms from both approaches. The experimental comparison of one of the algorithms, called Accuracy Weighted Ensemble, with other selected classifiers has, to our knowledge, not been previously done. Additionally, we propose and evaluate a new algorithm called Accuracy Diversified Ensemble, which selects, weights, and updates ensemble members according to the current stream distribution. For our experiments we use the Massive Online Analysis environment and extend it by attribute filtering and data chunk evaluation procedures. We also verify the framework's capability to become the first commonly used software environment for research on learning from evolving data streams.

1.2 Thesis structure

The structure of the thesis is as follows. Chapter 2 presents the basics of data stream mining. In particular, definitions of data streams, concept drift as well as types of stream learners and their applications are shown. Chapter 3 gives a deeper insight into single classifier approaches to data stream mining, presenting windowing techniques and Hoeffding trees. Ensemble approaches to classification in data streams, including the Streaming Ensemble Algorithm, Hoeffding Option Tree, and our Accuracy Diversified Ensemble, are presented in Chapter 4. The Massive Online Analysis framework, which was used for evaluation purposes in this thesis, is presented in Chapter 5. Chapter 6 describes experimental results and compares single classifier and ensemble algorithms for mining concept-drifting data streams. Finally, Chapter 7 concludes the thesis with a discussion on the completed work and possible lines of further investigations.

1.3 Acknowledgments

The author would like to thank all the people who contributed to this study. He is grateful to his supervisor, prof. Jerzy Stefanowski, for his inspiration, motivation, and the care with which he reviewed this work. The author is also greatly indebted to many other teachers of the Institute of Computing Science of Poznań University of Technology who got him interested in machine learning and data mining. Finally, the author wishes to thank his family: his parents, for their love, unconditional support and encouragement to pursue his interests, and his sister, for sharing her experience of dissertation writing and giving invaluable advice.

Chapter 2

Mining data streams

Before describing and evaluating different approaches to mining streams with concept drift, we present the basics of data streams. First, in Section 2.1 we focus on the main characteristics of the data stream model and how it differs from traditional data. Next, in Section 2.2, we define and categorize concept drift and its causes. Section 2.3 discusses the differences between data stream mining and classic data mining. It also presents a taxonomy of adaptive classification techniques. Finally, Section 2.4 describes the main applications of data stream mining techniques. As this thesis concentrates on classification techniques, we will use the term *data stream learning* as a synonym for data stream mining.

2.1 Data streams

A *data stream* is an ordered sequence of instances that arrive at a rate that does not permit to permanently store them in memory. Data streams are potentially unbounded in size making them impossible to process by most data mining approaches.

The main characteristics of the data stream model imply the following constraints [5]:

1. It is impossible to store all the data from the data stream. Only small summaries of data streams can be computed and stored, and the rest of the information is thrown away.
2. The arrival speed of data stream tuples forces each particular element to be processed essentially in real time, and then discarded.
3. The distribution generating the items 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 data stream summarization techniques. Constraint 3, is more important in some applications than in others. Many of the first data stream mining approaches ignored this characteristic and formed the group of *static data stream learning* algorithms. Other researches considered constraint 3 as a key feature and devoted their work to *evolving data stream learning*. Both of these approaches to stream data mining will be discussed in Section 2.3.

Most of data stream analysis, querying, classification, and clustering applications require some sort of summarization techniques to satisfy the earlier mentioned constraints. Summarization techniques are used for producing approximate answers from large data sets usually by means of data reduction and synopsis construction. This can be done by selecting only a subset of incoming data or by using sketching, load shedding, and aggregation techniques. In the next paragraphs, we present the basic methods used to reduce data stream size and speed for analysis purposes.

Sampling. Random sampling is probably the first developed and most common technique used to decrease data size whilst still capturing its essential characteristics. It is perhaps the easiest form of summarization in a data stream and other synopses can be built from a sample itself [2]. To obtain an unbiased sample of data we need to know the data set's size. Because in the data stream model the length of the stream is unknown and at times even unbounded, the sampling strategy needs to be modified. The simplest approach, involving sampling instances at periodic time intervals, provides a good way to "slow down" a data stream, but can involve massive information loss in streams with fluctuating data rates. A better solution is the reservoir sampling algorithm proposed by Vitter [77], which analyzes the data set in one pass and removes instances from a sample with a given probability rather than periodically selecting them. Chaudhuri, Motwani and Narasayya extended the reservoir algorithm to the case of weighted sampling [18]. Other approaches include sampling for decision tree classification and k-means clustering proposed by Domingos et al. [21], sampling for clustering data streams [39], and sampling in the sliding window model studied by Babcock, Datar, and Motwani [2].

Sketching. Sketching involves building a statistical summary of a data stream using a small amount of memory. It was introduced by Alon, Matias, and Szegedy [1] and consists of *frequency moments*, which can be defined as follows: "Let $\boldsymbol{S} = (x_1, ..., x_N)$ be a sequence of elements where each x_i belongs to the domain $D = \{1, ...d\}$. Let the multiplicity $m_i = |\{j : x_j = i\}|$ denote the number of occurrences of value i in the sequence $\boldsymbol{S}$. For $k \geq 0$, the *kth frequency moment* F_k of $\boldsymbol{S}$ is defined as $F_k = \sum_{i=1}^{d} m_i^k$; further, we define $F_\infty^* = max_i m_i$" [2]. The frequency moments capture the statistics of the data streams distribution in linear space. F_0 is the number of distinct values in sequence $\boldsymbol{S}$, F_1 is the length of the sequence, F_2 is the self-join size, and F_∞^* is the most frequent item's multiplicity. Several approaches to estimating different frequency moments have been proposed [2]. Sketching techniques are very convenient for summarization in distributed data stream systems where computation is performed over multiple streams. The biggest shortcoming of using frequency moments is their accuracy [45].

Histograms. Histograms are summary structures capable of aggregating the distribution of values in a dataset. They are used in tasks such as query size estimation, approximate query answering, and data mining. The most common types of histograms for data streams are: V-optimal histograms, equal-width histograms, end-biased histograms. V-optimal histograms approximate the distribution of a set of values $v_1, ..., v_2$ by a piecewise constant function $\widehat{v}(i)$, so as to minimize the sum of squared error $\sum_i (v_i - \widehat{v}(i))^2$. Ways of computing V-optimal histograms have been discussed in [47, 40, 36]. Equal-width histograms aggregate data distributions to instance counts for equally wide data ranges. They partition the domain into buckets such that the number of v_i values falling into each bucket is uniform across all buckets. Equal-width histograms are less difficult to compute and update than V-optimal histograms, but are also less accurate [37]. End-biased histograms contain exact counts of items that occur with frequency above a threshold, but approximate the other counts by a uniform distribution. End-biased histograms are often used by *iceberg queries* - queries to find aggregate values above a specified threshold. Such queries are common in information retrieval, data mining, and data warehousing [2, 25].

Wavelets. Wavelets are used as a technique for approximating data with a given probability. Wavelet coefficients are projections of a given signal (set of data values) onto an orthogonal set of basis vectors. There are many types of basis vectors, but due to their ease of computation, the most commonly used are Haar wavelets. They have the desirable property that the signal

reconstructed from the top few wavelet coefficients best approximates the original signal in terms of the L2 norm. Research in computing the top wavelet coefficients in the data stream model is discussed in [36, 62].

Sliding Window. Sliding windows provide a way of limiting the analyzed data stream tuples to the most recent instances. This technique is deterministic, as it does not involve any random selections and prevents stale data from influencing statistics. It is used to approximate data stream query answers, maintain recent statistics [20], V-optimal histograms [38], compute iceberg queries [60], among other applications [45]. We will give a deeper insight into using sliding windows in data stream mining in Section 3.2.

2.2 Concept drift

As mentioned in the previous section, the distribution generating the items of a data stream can change over time. These changes, depending on the research area, are referred to as *temporal evolution*, *covariate shift*, *non stationarity*, or *concept drift*. Concept drift is an unforseen substitution of one data source S_1 (with an underlying probability distribution Π_{S_1}), with another source S_2 (with distribution Π_{S_2}). The most popular example to present the problem of concept drift is that of detecting and filtering out spam e-mail. The distinction between unwanted and legitimate e-mails is user-specific and evolves with time. As concept drift is assumed to be unpredictable, periodic seasonality is usually not considered as a concept drift problem. As an exception, if seasonality is not known with certainty, it might be regarded as a concept drift problem. The core assumption, when dealing with the concept drift problem, is uncertainty about the future - we assume that the source of the target instance is not known with certainty. It can be assumed, estimated, or predicted, but there is no certainty [85].

Kelly et al. [50] presented three ways in which concept drift may occur:

- prior probabilities of classes, $P(c_1), ..., P(c_k)$ may change over time,
- class-conditional probability distributions, $P(X|c_i)$, $i = 1, ..., k$ might change,
- posterior probabilities $P(c_i|X)$, $i = 1, ..., k$ might change.

It is worth noting that the distributions $P(X|c_i)$ might change in such a way that the class membership is not affected (e.g. symmetric movement to opposite directions). This is one of the reasons why this type of changed is often referred to as virtual drift and change in $P(c_i|X)$ is referred to as real drift. From a practical point of view, the distinction between virtual and real drifts is of little importance, and we will not make that distinction in this thesis.

Figure 2.1 shows six basic types of changes that may occur in a single variable along time. The first plot (Sudden) shows abrupt changes that instantly and irreversibly change the variables class assignment. Real life examples of such changes include season change in sales. The next two plots (Incremental and Gradual) illustrate changes that happen slowly over time. Incremental drift occurs when variables slowly change their values over time, and gradual drift occurs when the change involves the class distribution of variables. Some researchers do not distinguish these two types of drift and use the terms gradual and incremental as synonyms. A typical example of incremental drift is price growth due to inflation, whilst gradual changes are exemplified by slowly changing definitions of spam or user-interesting news feeds. The left-bottom plot (Recurring) represents changes that are only temporary and are reverted after some time. This type of change is regarded by some researchers as *local drift* [75]. It happens when several data generating sources are expected to switch over time and reappear at irregular time intervals. This drift is not certainly

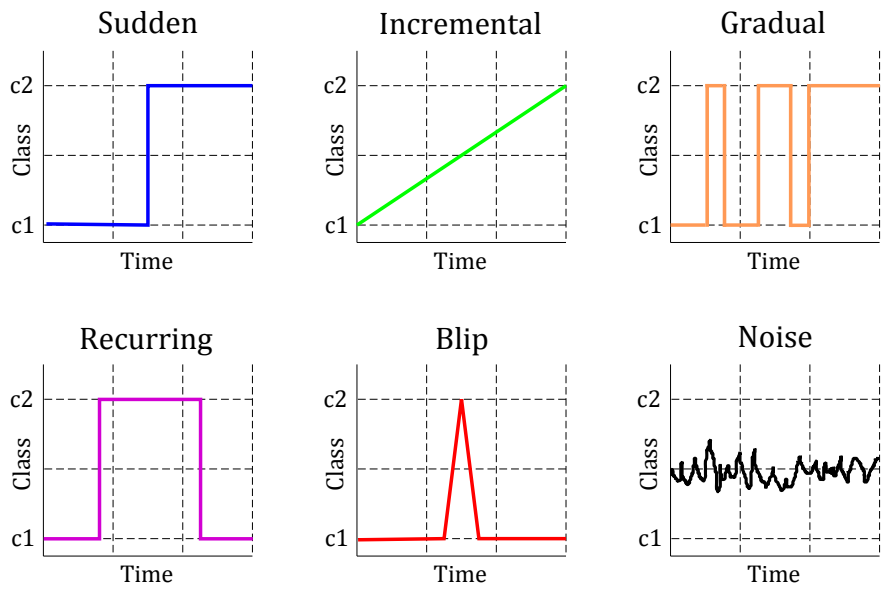

Figure 2.1: Types of changes in streaming data. Apart from Noise and Blip, all the presented changes are treated as concept drift and require model adaptation.

periodic, it is not clear when the source might reappear, that is the main difference from the seasonality concept used in statistics [85]. The fifth plot (Blip) represents a "rare event", which could be regarded as an outlier in a static distribution. In streaming data, detected blips should be ignored as the change they represent is random. Examples of blips include anomalies in landfill gas emission, fraudulent card transactions and network intrusion [55]. The last plot in Figure 2.1 represents random changes, which should be filtered out. Noise should not be considered as concept drift as it is an insignificant fluctuation that is not connected with any change in the source distribution.

It is important to note that the presented types of drift are not exhaustive and that in real life situations concept drift is a complex combination of many types of drift. If a data stream of length t has just two data generating sources S_1 and S_2, 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 the assumption about the change types is absolutely needed for designing adaptivity strategies.

2.3 Data stream mining

After discussing the characteristics of data streams and drifting concepts we see that data stream learning must differ from traditional data mining techniques. Table 2.1 presents a comparison of traditional and stream data mining environments.

Table 2.1: Traditional and stream data mining comparison [29].

	Traditional	**Stream**
No. of passes	Multiple	Single
Processing time	Unlimited	Restricted
Memory usage	Unlimited	Restricted
Type of result	Accurate	Approximate
Concept	Static	Evolving
Distributed	No	Yes

Data stream classifiers must be capable of learning data sequentially rather than in batch mode and they must react to the changing environment. The first of these two requirements is fulfilled by using *online learners*, the second by implementing a *forgetting method*.

2.3.1 Online learning

Online learning, also termed incremental learning, focuses on processing incoming examples sequentially in such a way that the trained classifier is as accurate, or nearly as accurate, as a classifier trained on the whole data set at once. During classifier training, the true label for each example is known immediately or recovered at some stage later. Knowing the true label of an example, the classifier is updated, minimally if possible, to accommodate the new training point. Traditionally, online learners are assumed to work, like most data mining algorithms, in static environments. No forgetting of the learned knowledge is envisaged.

A well designed online classifier should have the following qualities [27, 73, 54, 11]:

1. **Incremental:** The algorithm should read blocks of data at a time, rather than require all of it at the beginning.
2. **Single pass:** The algorithm should make only one pass through the data.
3. **Limited time and memory:** Each example should be processed in a (small) constant time regardless of the number of examples processed in the past and should require an approximately constant amount of memory.
4. **Any-time learning:** If stopped at time t, before its conclusion, the algorithm should provide the best possible answer. Ideally, the trained classifier should be equivalent to a classifier trained on the batch data up to time t.

A learner with the above qualities can accurately classify large streams of data without the need of rebuilding the classifier from scratch after millions of examples. Such learners can be constructed by *scaling up* traditional machine learning algorithms. This is done either by wrapping a traditional learner to maximize the reuse of existing schemes, or by creating new methods tailored to the data stream setting. Examples of both approaches will be discussed in detail in Chapters 3 and 4. As mentioned earlier, online learners do not have any forgetting mechanism by default and need to be equipped with one to be suitable for data mining streams with concept drift.

2.3.2 Forgetting mechanisms

A data stream classifier should be able to react to the changing concept by forgetting outdated data, while learning new class descriptions. The main problem is how to select the data range to remember. The simplest solution is forgetting training objects at a constant rate and using only a window of the latest examples to train the classifier. This approach, based on fixed parameters, is caught in a tradeoff between stability and flexibility. If the window is small, the system will be very responsive and will react quickly to changes, but the accuracy of the classifier might be low due to insufficient training data in the window. Alternatively, a large window may lead to a sluggish, but stable and well trained classifier [54].

A different approach involves ageing at a variable rate. This may mean changing window size when a concept drift is detected or using decay functions to differentiate the impact of data points. This approach tries to dynamically modify the window parameters to find the best balance between accuracy and flexibility. It is usually better suited for environments with sudden drift, where the change is easier to detect.

Another approach to forgetting outdated and potentially harmful data is selecting examples according to their class distribution. This is especially useful in situations where older data points may be more relevant than more recent points. This approach, considered in adaptive nearest neighbors models, uses weights that decay with time, but can be modified depending on the neighborhood of the most recent examples.

There is no best generic solution when it comes to implementing a forgetting mechanism for a data stream learner. Depending on the environment, the expected types of drift, different approaches may work better. For rather static data streams with gradual concept drift, static windows should give the best accuracy. Sudden changes suggest the use of system with data ageing at a variable rate, while recurring context should be best handled by density-based forgetting systems. Examples of windowing techniques are discussed in more detail in Section 3.2.

2.3.3 Taxonomy of methods

Based on "how" and "when" the algorithms learn and forget, Žliobaitė [84] proposed a taxonomy of the main groups of adaptive classifiers. The taxonomy is graphically presented in Figure 2.2.

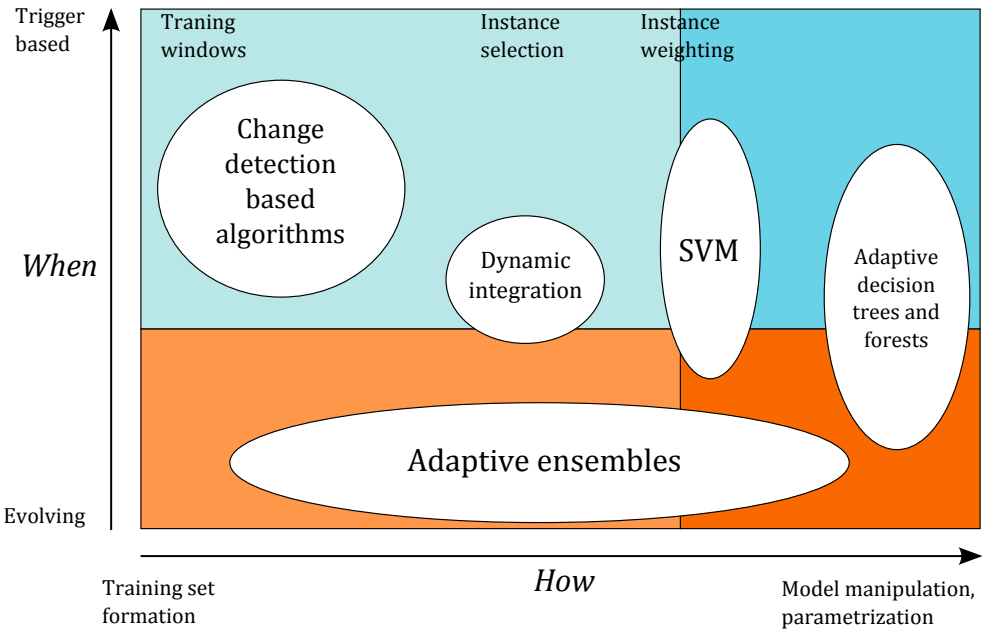

Figure 2.2: A taxonomy of adaptive supervised learning techniques [84].

The "when" dimension ranges from gradually evolving to trigger based learners. Trigger based means that there is a signal (usually a drift detector) which indicates a need for model update. Such methods work well in data streams with sudden drift as the signal can be used to rebuild the whole model instead of just updating it. On the other hand, evolving methods update the model gradually and usually react to changes without any drift detection mechanism. By manipulating ensemble weights or substituting models they try to adapt to the changing environment without rebuilding the whole model.

The "how" dimension groups learners based on how they adapt. The adaptation mechanisms mainly involve example selection or parametrization of the base learner. Basic methods of adjusting models to changing concepts were presented in Section 2.3.2. Detailed descriptions of approaches presented in Figure 2.2 will be discussed with algorithm pseudo-codes in Chapters 3 and 4.

2.4 Applications

In this section we discuss some of the main data stream application domains in which concept drift plays an important role. Along with the characteristics of each domain we present real life problems and discuss the sources of drift in the context of these problems. A more thorough discussion on concept drift application domains can be found in [85].

2.4.1 Monitoring systems

Monitoring systems are characterized by large data streams which need to be analyzed in real time. Classes in these systems are usually highly imbalanced, which makes the task even more complicated. The typical task of a monitoring system is to distinguish unwanted situations from "normal behavior". This includes recognizing adversary actions and alerting before critical system states.

Network security. The detection of unwanted computer access, also called *intrusion detection*, is one of the typical monitoring problems. Intrusion detection systems filter incoming network traffic in search of suspicious behavior. The source of concept drift in this application is mainly connected with the attacker. Adversary actions taken by the intruder evolve with time, to surpass the also evolving security systems. The technological progress is another source of concept drift, as modern systems providing more functionalities often provide more possibilities of attack [56, 61, 68].

Telecommunications. Intrusion and fraud detectors are also an important part of telecommunication systems. The goal is to prevent fraud and stop adversaries from accessing private data. Once again, the source of concept drift is the change in adversary behavior as well as change in the behavior of legitimate users [63, 43].

Finance. Streams of financial transactions can be monitored to alert for possible credit card and Internet banking frauds. Stock markets also use data stream mining techniques to prevent insider trading. In both cases the data labeling might be imprecise due to unnoticed frauds and misinterpreted legitimate transactions. Like in the previous examples, the source of concept drift is the evolving user behavior. This is especially challenging with insider trading where the adversary possesses non-public information about a company and tries to distribute his transactions in a non-trivial way [22].

Transportation. Data stream mining techniques can be employed to monitor and forecast traffic states and public transportation travel time. This information can be useful for scheduling and traffic planning as well as dynamic reacting to traffic jams. Traffic patterns can change seasonally as well as permanently, thus the systems have to be able to handle concept drift. Human driver factors can also be significant to concept drift [64].

Industrial monitoring. There are several emerging applications in the area of sensor monitoring where a large numbers of sensors are distributed in the physical world and generate streams of data that need to be combined, monitored, and analyzed [2]. 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 systems context [29, 76].

2.4.2 Personal assistance

Data stream mining techniques are also used to personalize and organize the flow of information. This can include individual assistance for personal use, customer profiling, and recommendation systems. The costs of mistakes are relatively low compared to other applications, and the class labels are mostly "soft". A mistake made by a recommendation system is surely less important than a mistake made by an intrusion detector. Moreover, a recommendation systems user himself does not know for sure, which of the two given movies he likes more. Personal assistance systems do not have to react in real time, but are usually affected by more than one source of drift.

News feeds. Most individual assistance applications are related to textual data. They aim at classifying news feeds and categorizing articles. Drifting user interests can be a cause of reoccurring contexts in such systems. Also article topics and nomenclature may drift, causing the distribution of articles to change independently from the users interests. This is a typical example of virtual drift. There are also applications addressing web personalization and dynamics, which is again subject to drifting user interests. Here, mostly data logs are mined to profile user interests without his involvement [49].

Spam filtering. Spam filtering is a more complicated type of information filtering. In contrast to most personal assistance systems, it is open to adversary actions (spamming). Adversaries are adaptive and change spam content rapidly to overcome filters. The amount and types of illegitimate mail are subject to seasonality, but also drift irregularly over time. The definition of spam may also differ between users [49].

Recommendation systems. An application that is not strictly connected with data streams, but also involves concept drift is customer profiling and assistance in recommendation systems. One of the challenges faced by recommender systems is the sparsity of data. Most users rate/buy only a few products, but the recommendation task needs to be performed on the whole data set. The publicity of recommender systems research has increased rapidly with the NetFlix movie recommendation competition. The winners used temporal aspect as one of the keys to the problem. They noted three sources of drift in the competition task: the change of movie popularity over time, the drift of users' rating scale, and changes in user preferences [4].

Economics. Macroeconomic forecasts and financial time series are also subjects to data stream mining. The data in those applications is drifting primary due to a large number of factors that are not included in the model. The publicly known information about companies can form only a small part of attributes needed to properly model financial forecasts as a stationary problem. That is why the main source of drift is a hidden context.

2.4.3 Decision support

Decision support with concept drift includes diagnostics and evaluation of creditworthiness. The true answer whether a decision is correct is usually delayed in these systems. Decision support and diagnostic applications typically involve limited amount of data and are not required to be made in real time. The cost of mistakes in these systems is large, thus the main challenge is high accuracy.

Finance. Bankruptcy prediction or individual credit scoring is typically considered to be a stationary problem. However, just like in the earlier mentioned financial applications, there is drift due to hidden context. The decisions that need to be made by the system are based on fragmentary information. The need for different models for bankruptcy prediction under different economic conditions was acknowledged, but the need for models to be able to deal with non stationarity has been rarely researched [85].

Biomedical applications. Biomedical applications present an interesting field of concept drift research due to the adaptive nature of microorganisms. As microorganism mutate, their resistance to antibiotics changes. Patients treated with antibiotics when it is not necessary, can become "immune" to their action when really needed. Other medical applications include changes in disease progression, discovering emerging resistance and monitoring nonsomnical infections. Concept drift also occurs in biometric authentication. The classification drift in this application is usually caused by hidden context such as new light sources, image background, and rotation as well as physiological factors, for example growing beard. The adaptivity of the algorithms should be used with caution, due to potential adversary behavior [75].

2.4.4 Artificial intelligence

Learning in dynamic environments is a branch of machine learning and AI where concept drift plays an important role. Classification algorithms learn how to interact with the environment to achieve a given goal and since the environment is changing, the learners need to be adaptive to succeed in completing their task. *Ubiquitous Knowledge Discovery* (UKD), which deals with mobile distributed systems such as navigation systems, vehicle monitoring, household management systems, is also prone to concept drift.

Navigation systems. The winners of the 2005 DARPA Grand Challenge used online learning for road image classification. The main sources of concept drift were the changing road conditions. The designing of a soccer player robot brings similar challenges and sources of drift [74, 57].

Smart homes and virtual reality. Intelligent household appliances need to be adaptive to changing environment and user needs. Also virtual reality needs mechanisms to take concept drift into account. In computer games and flight simulators, the virtual reality should adapt to the skills of different users and prevent adversary actions like cheating [17].

Chapter 3

Single classifier approaches

In this chapter we discuss the most popular single classifiers used to classify streaming data. In Section 3.1 we present learners proposed for stationary classification tasks that can also be employed to classify data streams. In Section 3.2 we discuss the use of windows to model the forgetting process, necessary to react to concept drift. Drift detectors, wrapper methods allowing to rebuild classifiers only when necessary, are presented in Section 3.3. Finally, in Section 3.4 we discuss Very Fast Decision Trees (VFDT), an anytime system that builds decision trees using constant time and memory per example.

3.1 Traditional learners

Some of the popular classifiers proposed for stationary data mining fulfill both of the stream mining requirements - have the qualities of an online learner and a forgetting mechanism. Some methods that are only able to process data sequentially, but do not adapt, can be easily modified to react to change. In the following paragraphs we present four learners that fall into these groups: neural networks, Naive Bayes, nearest neighbor methods, and decision rules.

Neural networks. In traditional data mining applications, neural networks are trained using the epoch protocol. The entire set of examples is sequentially passed through the network a previously defined number of times (epochs) and updates neuron weights, usually according to the backpropagation algorithm. Presenting the same data multiple times allows the learner to better adjust to the presented concept and provide better classification accuracy.

By abandoning the epoch protocol, and presenting examples in a single pass, neural networks can easily work in data stream environments. Each example is seen only once and usually constant time is required to update neuron weights. Most 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. 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. A real world application using neural networks for data stream mining is given by Gama and Rodrigues [33].

Naive Bayes. This model is based on the Bayes' theorem and computes class-conditional probabilities for each new example. Bayesian methods learn incrementally by nature and require constant memory. Naive Bayes is a *lossless classifier*, meaning it "produces a classifier functionally equivalent to the corresponding classifier trained on the batch data" [54]. To add a forgetting

mechanism usually sliding windows are employed to "unlearn" the oldest examples. A single Naive Bayes model will generally not be as accurate as more complex models. Bayesian networks, which give better results, are also suited to the data stream setting, it is only necessary to dynamically learn their structure [13].

Nearest neighbor. Nearest neighbor classifiers, also called *instance-based learners* or *lazy learners*, provide an accurate 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, the reference set grows with each example increasing memory requirements and classification time. A more recent method from this family called IB3, 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.

Decision rules. Rule-based models can also be adjusted to data stream environments. Decision rule classifiers consist of rules - 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 many rules can affect a decision for a single example. These observations served as base for developing complex data stream mining systems like SCALLOP [28], FACIL [31] and FLORA [80]. These systems learn rules incrementally and employ dynamic windows to provide a forgetting mechanism.

3.2 Windowing techniques

The most popular approach to dealing with time changing data involves the use of *sliding windows*. Windows provide a way of limiting the amount of examples introduced to the learner, thus eliminating those data points that come from an old concept. The procedure of using sliding windows for mining data stream is presented in Algorithm 3.1. Because in this work we discuss algorithms that have the property of any-time learning and should be able to provide the best answer after each example, the pseudo-codes do not contain explicit return statements. We assume that the output classifier is available at any moment of the processing of the input stream.

Algorithm 3.1 The basic windowing algorithm

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

1: initialize window W;
2: **for all** examples $x_i \in \mathcal{S}$ **do**
3: $W \leftarrow W \cup \{x_i\}$;
4: if necessary remove outdated examples from W;
5: rebuild/update C using W;

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 - 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 thrown away. When using windows

of fixed size, the user is caught in a tradeoff. If he chooses a small window size the classifier will react quickly to changes, but may loose on accuracy in periods of stability, choosing a large size will result in increasing accuracy in periods of stability, but will fail to adapt to rapidly changing concepts. That is why dynamic ways of modeling the forgetting process have been proposed.

3.2.1 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 [19] analyzed the use of different decay functions for calculating data stream aggregates. Equations 3.1 through 3.3 present the proposed functions.

$$w_{exp}(t) = e^{-\lambda t}, \quad \lambda > 0 \tag{3.1}$$

$$w_{poly}(t) = \frac{1}{t^{\alpha}}, \quad \alpha > 0 \tag{3.2}$$

$$w_{chord}(t) = 1 - \frac{t}{|W|} \tag{3.3}$$

Equation 3.1 presents an exponential decay function, 3.2 a polynomial function, and 3.3 a chordal function. For each of the functions t represents the age of an example. A new example will have $t = 0$ whilst the last example that fits chronologically in a window will have $t = |W| - 1$. The use of decay functions allows to gradually weight the examples offering a compromise between large and small fixed windows. Algorithm 3.2 presents the process of obtaining a window with decaying weights.

Algorithm 3.2 Weighted windows

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

1: **for all** examples $x_i \in \mathcal{S}$ **do**
2: **if** $|W| = k$ **then**
3: remove the oldest example from W;
4: $W \leftarrow W \cup \{x_i\}$;
5: **for all** examples $x_j \in W$ **do**
6: calculate example's weight $w(x_j)$;

3.2.2 FISH

Žliobaitė [85] proposed a family of algorithms called FISH, that use time and space similarities between examples as a way of dynamically creating a window. To explain her approach, let us consider an illustrative example, which we present in Figure 3.1. 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 in space area, depicted with a red circle, the correct class changes as the optimal boundary rotates. The examples shows that similarity in an evolving environment depends on both time and space.

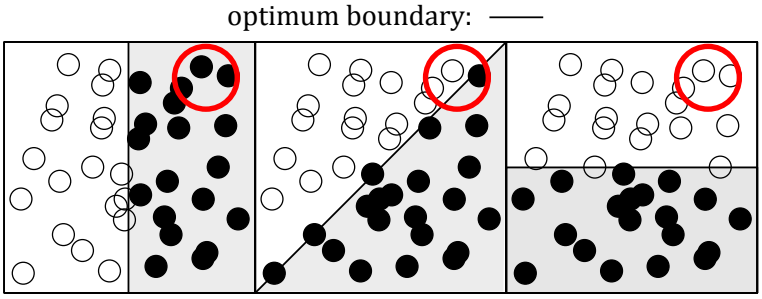

Figure 3.1: Rotating hyperplane example [82]: (left) initial source $S1$, (center) source $S2$ after 45° rotation, (right) source $S3$ after 90° rotation. Black and white dots represent the two classes.

The author proposed the selection of training examples based on a distance measure D_{ij} defined as follows:

$$D_{ij} = a_1 d_{ij}^{(s)} + a_2 d_{ij}^{(t)} \tag{3.4}$$

where $d^{(s)}$ indicates distance in attribute space, $d^{(t)}$ indicates distance in time, and $a1, a2$ are the weight coefficients. In order to manage the balance between the time and space distances, $d^{(s)}$ and $d^{(t)}$ need to be normalized. For two examples x_i, x_j the author proposes Eucledian distance ($d_{ij}^{(s)} = \sqrt{\sum_{k=1}^{p} |x_i^k - x_j^k|^2}$) for distance in space and the number of dividing examples ($d_{ij}^{(t)} = |i - j|$) for 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 FISH3 in Algorithm 3.3.

Algorithm 3.3 FISH3 [85]

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 $x_i \in \mathcal{S}$ **do**
2: **for** $\alpha = 0$; $\alpha \leq 1$; $\alpha \leftarrow \alpha + proportionStep$ **do**
3: $a_1 \leftarrow \alpha$;
4: $a_2 \leftarrow 1 - \alpha$;
5: **for all** remembered historical examples $x_j \in \{x_{i-b}, ..., x_{i-1}\}$ **do**
6: calculate distance D_{ij} using (3.4);
7: sort the distances from minimum to maximum;
8: **for** $s = k$; $s \leq b$; $s \leftarrow s + windowStep$ **do**
9: select s instances having the smallest distance D;
10: using cross-validation build a classifier C_s using the instances indexed $\{i_1, ..., i_s\}$ and test it on the k nearest neighbors indexed $\{i_1, ..., i_k\}$;
11: record the acquired testing error e_s;
12: find the minimum error classifier C_l, where $l = \arg\min_{l=k,...,b}(e_l)$;
13: $W \leftarrow$ instances indexed $i_1, ..., i_l$;

For each new example x_i 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 x_i 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 l, 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 l 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 [82] takes the time/space proportion as a parameter, while the first algorithm from the family, FISH1 [83], uses a fixed window of the nearest instances. To implement a variable sample size the FISH2 and FISH3 incorporate the principles from two windowing methods proposed by Klinkenberg et al. [52] and Tsymbal et al. [75].

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.

Experiments performed on 6 small and medium size data sets for 4 types of base classifiers (decision tree, Nearest Neighbor, Nearest Mean, and Parzen Window) showed that integration of similarity in time and feature space when selecting a training set improve generalization performance [85]. Additionally, FISH2 has been compared with, and outperformed, the windowing methods of Klinkenberg and Tsymbal. The FISH family of algorithms should be regarded as a generic extension to other classification techniques that can be employed when dynamic widowing is necessary.

3.2.3 ADWIN

Bifet [6, 7] proposed an adapting sliding window algorithm called ADWIN suitable for data streams with sudden drift. The algorithm keeps a sliding window W with 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_t remained constant in W with confidence δ"? The pseudo-code of ADWIN is listed in Algorithm 3.4.

Algorithm 3.4 Adaptive windowing algorithm [7]

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

1: initialize window W;
2: **for all** $x_i \in \mathcal{S}$ **do**
3: $W \leftarrow W \cup \{x_i\}$;
4: **repeat**
5: drop the oldest element from W;
6: **until** $|\hat{\mu_{W_0}} - \hat{\mu_{W_1}}| < \epsilon_{cut}$ holds for every split of W into $W = W_0 \cdot W_1$;

The key part of the algorithm lies in the definition of ϵ_{cut} and the test it is used for. The authors state that different statistical tests can be used for this purpose, but propose only one specific implementation. Let n denote the size of W, and n_0 and n_1 the sizes of W_0 and W_1 consequently, so that $n = n_0 + n_1$. 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:

$$\epsilon_{cut} = \sqrt{\frac{1}{2m} \cdot \frac{4}{\delta'}}, \tag{3.5}$$

where

$$m = \frac{1}{1/n_0 + 1/n_1}, \text{ and } \delta' = \frac{\delta}{n}.$$

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 is calculated using the Hoeffding bound, thus gives formal guarantees of the base classifiers performance. The phrase "holds for every split of W into $W = W_0 \cdot W_1$" means that we need to check all pairs of subwindows W_0 and W_1 created by splitting W into 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 [7]. The originally proposed ADWIN algorithms are also lossless learners, thus the window size W can grow infinitely if no drift occurs. This can be easily improved by adding a parameter that would limit the windows maximal size. In its original form, proposed by Bifet, ADWIN works only for 1-dimensional data, e.g., the running error. For this method to be used for n-dimensional raw data, a separate window should be maintained for each dimension. Such a modified model, although costly, reflects the fact that the importance of each feature may change at different pace.

3.3 Drift detectors

After windowing techniques, another group of algorithms allowing to adapt almost any learner to evolving data streams are *drift detectors*. Their task is to detect concept drift and alarm the 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 numeric sequences the first proposed tests where the Cumulated Sum (CUSUM) [67] and the Geometric Moving Average (GMA) [71]. 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 have been proposed. Below, we discuss two recently proposed tests designed for drifting data streams.

3.3.1 DDM

Gama et al. [30] 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. That is why 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 as given by Equation 3.6.

$$s_i = \sqrt{\frac{p_i(1-p_i)}{i}} \tag{3.6}$$

The authors use the fact, that for a sufficiently large number of examples ($n > 30$), the Binomial distribution is closely approximated by a Normal distribution with the same mean and variance. For each example in the data stream the error rate is tracked updating two registers: p_{min} and s_{min}. These values are used to calculate a *warning level* condition presented in Equation 3.7 and an *alarm level* condition presented in Equation 3.8. 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.

$$p_i + s_i \geq p_{min} + \alpha \cdot s_{min} \tag{3.7}$$

$$p_i + s_i \geq p_{min} + \beta \cdot s_{min} \tag{3.8}$$

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 3.5 shows the steps of the Drift Detection Method.

Algorithm 3.5 The Drift Detection Method [30]

Input: $\mathcal{S}$: a data stream of examples
C: classifier
Output: W: a window with examples selected to train classifier C

1: Initialize$(i, p_i, s_i, ps_{min}, p_{min}, s_{min})$;
2: $newDrift \leftarrow false$;
3: $W \leftarrow \emptyset$;
4: $W' \leftarrow \emptyset$;
5: **for all** examples $x_i \in \mathcal{S}$ **do**
6: **if** prediction $C(x_i)$ 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: compute s_i using (3.6);
11: $i \leftarrow i + 1$;
12: **if** $i > 30$ (approximated normal distribution) **then**
13: **if** $p_i + s_i \leq ps_{min}$ **then**
14: $p_{min} \leftarrow p_i$;
15: $s_{min} \leftarrow s_i$;
16: $ps_{min} \leftarrow p_i + s_i$;
17: **if** drift detected (3.8) **then**
18: Initialize$(i, p_i, s_i, ps_{min}, p_{min}, s_{min})$;
19: $W \leftarrow W'$;
20: $W' \leftarrow \emptyset$;
21: **else if** warning level reached (3.7) **then**
22: **if** $newDrift = true$ **then**
23: $W' \leftarrow \emptyset$;
24: $newDrift \leftarrow false$
25: $W' \leftarrow W' \cup \{x_i\}$
26: **else**
27: $newDrift \leftarrow true$;
28: $W \leftarrow W \cup \{x_i\}$;

Algorithm 3.6 DDM: Initialize()

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

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: $s_{min} \leftarrow \infty$;

DDM works best on data streams with sudden drift as gradually changing concepts can pass without triggering the alarm level. When no changes are detected, DDM works like a lossless learner constantly enlarging the window size which can lead to the memory limit being exceeded.

3.3.2 EDDM

Baena-García et al. [3] 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 errors and s'_i as its standard deviation. Using these values the new warning and alarm conditions are given by Equation 3.9 and 3.10.

$$\frac{p'_i + 2 \cdot s'_i}{p'_{max} + 2 \cdot s'_{max}} < \alpha \tag{3.9}$$

$$\frac{p'_i + 3 \cdot s'_i}{p'_{max} + 3 \cdot s'_{max}} < \beta \tag{3.10}$$

EDDM works better than DDM for slow gradual drift, but is more sensitive to noise. Another drawback of this method is that it considers the thresholds and searches for concept drift when a minimum of 30 errors have occurred. This is necessary to approximate the Binomial distribution by a Normal distribution, but can take a large amount of examples to happen.

3.4 Hoeffding trees

Decision trees were the first learners 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:

$$\epsilon = \sqrt{\frac{R^2 ln(1/\delta)}{2n}}. \tag{3.11}$$

Using this bound, Domingos and Hulten [21] proposed a classifier called Very Fast Decision Tree which we present in Algorithm 3.7.

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 [69, 16, 70] and differs almost only in the selection of the split attribute. Instead of selecting the best attribute (in terms of split evaluation function $G(\cdot)$) after viewing all the examples, it uses the Hoeffding bound to calculate the number of examples necessary to select the right split-node with probability $1 - \delta$.

Algorithm 3.7 The Hoeffding tree algorithm [21]

Input: $\mathcal{S}$: a data stream of examples
$\mathcal{X}$: a set of discrete attributes
$G(\cdot)$: a split evaluation function
δ: split confidence
Output: H_T: a Hoeffding decision tree

```
1: H_T ← a tree with a single leaf l_1 (the root);
2: X_1 ← X ∪ {X_0};
3: G_1(X_0) ← G obtained by predicting the most frequent class in S;
4: for all classes y_k ∈ Y do
5:    for all values x_ij of each attribute X_i ∈ X do
6:       n_ijk(l_1) ← 0;
7: for all examples (x̄, y) ∈ S do
8:    Sort (x̄, y) into a leaf l using H_T;
9:    for all attribute values x_ij ∈ x̄ such that X_i ∈ X_l do
10:      n_ijk(l) ← n_ijk(l) + 1;
11:   label l with the majority class among the examples seen so far at l;
12:   if the examples seen so far at l are not all of the same class then
13:      compute G_l(X_i) for each X_i ∈ X_l − {X_0} using the counts n_ijk(l);
14:      X_a ← the attribute with the highest G_l;
15:      X_b ← the attribute with the second-highest G_l;
16:      compute Hoeffding bound ε using (3.11);
17:      if G_l(X_a) − G_l(X_b) > ε and X_a ≠ X_0 then
18:         replace l by an internal node that splits on X_a;
19:         for all branches of the split do
20:            add a new leaf l_m;
21:            X_m ← X − {X_a};
22:            G_m(X_0) ← the G obtained by predicting the most frequent class at l_m;
23:            for all classes y_k ∈ Y and each value x_ij of each attribute X_i ∈ X_m − {X_0} do
24:               n_ijk(l_m) ← 0;
```

Many enhancements to the basic VFDT algorithm have been proposed. Domingos and Hulten [21] introduced a method of limiting memory usage. They proposed to eliminate the statistics held by the "least promising" leafs. 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 rate of error at l. To reduce memory usage even more, they also suggested the removal of statistics of the poorest attributes in each leaf.

The Hoeffding 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 [48] proposed the use of an alternative bound which requires less examples for each split node. They also proposed a way of dealing with numerical attributes, which VFDT originally does not support, called Numerical Interleave Pruning (NIP). NIP creates data structures similar to histograms for numerical attributes with many distinct values. With time, the number of bins in such histograms can be pruned allowing the memory usage to remain constant.

A different approach to dealing with numerical attributes was proposed by Gama et al. [32]. They 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 [44, 11, 32] to Hoeffding trees include the use of *grace periods*, *tie-breaking*, and *skewed split prevention*. 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. Still,

after each example leaf statistics are 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 τ, which is used in an additional condition $\epsilon < \tau$ in line 17 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" [11].

The originally proposed VFDT algorithm was designed for static 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. [44] in the paper "Mining Time-Changing Data Streams". 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 forgot by updating node statistics and necessary model changes are performed on subtrees rather than the whole classifier.

Different approaches to adding a forgetting mechanism to the Hoeffding Tree include using an Exponential Weight Moving Average (EWMA) or ADWIN as drift detectors [5]. The latter, gives performance guarantees concerning the obtained error rate and both mentioned methods are more accurate and less memory consuming than CVFDT. The price the EWMA and ADWIN tree extensions pay is the average time necessary to process a single example.

Hoeffding trees represent the current state-of-the-art in single classifier mining of data streams. They fulfill all the requirements of an online learner presented in Section 2.3 and provide good interpretability. Their performance has been compared with traditional decision trees, Naive Bayes, kNN, and ensemble methods [21, 10, 32, 44, 48]. They proved to be much faster and less memory consuming while handling extremely large datasets. The compared ensemble methods require much more time and memory, and the accuracy boost they offer was usually marginal compared to the used resources.

Chapter 4

Ensemble approaches

Classifier ensembles are a common way of boosting classification accuracy. Due to their modularity, they also provide a natural way of adapting to change by modifying ensemble members. In this chapter we discuss the use of ensemble classifiers to mine evolving data streams. In Section 4.1 main types of ensemble modification techniques are shown. Following sections describe three specific adaptive ensemble algorithms. Section 4.2 discusses the Streaming Ensemble Algorithm, Section 4.3 Accuracy Weighted Ensembles, and Section 4.4 Hoeffding Option Trees. Finally, in Section 4.5 we propose a new data stream classifier called Accuracy Diversified Ensemble.

4.1 Ensemble strategies for changing environments

Ensemble algorithms are sets of single classifiers (components) whose decisions are aggregated by a voting rule. The combined decision of many single classifiers is usually more accurate than that given by a single component. Studies show that to obtain this accuracy boost, it is necessary to diversify ensemble members from each other. Components can differ from each other by the data they have been trained on, the attributes they use, or the base learner they have been created from. For a new example, class predictions are usually established by member voting. A generic ensemble training scheme is presented in Algorithm 4.1.

Algorithm 4.1 Generic ensemble training algorithm [51]

Input: $\mathcal{S}$: a 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: assign a weight to each example in $\mathcal{S}$ to create weight distribution D_m;
4: build/update C_i with $\mathcal{S}$ modified by weight distribution D_m;

Ensemble training is a costly process. It requires at least k times more processing than the training of a single classifier, plus member example selection and weight assignment usually make the process even longer. In massive data streams, single classifier models can perform better because there might not be time for running and updating an ensemble. On the other hand, if time is not of primary importance, but very high accuracy is required, an ensemble would be the natural solution.

Kuncheva [54] proposes to group ensemble strategies for changing environments as follows:

- *Dynamic combiners (horse racing)* - individual classifiers (experts) are trained in advance and the forgetting process is modeled by changing the expert combination rule.
- *Updated training data* - the experts in the ensemble are created incrementally by incoming examples. The combination rule may or may not change in the process.
- *Updating the ensemble members* - ensemble members are update online or retrained with blocks of data.
- *Structural changes of the ensemble* - periodically or when change is detected, ensemble members are reevaluated and the worst classifiers are updated or replaced with a classifier trained on the most recent examples.
- *Adding new features* - as the importance of features evolves with time, the attributes used by team members are changed without redesigning the ensemble structure.

We will discuss horse racing, updating members, and structural changes in more detail.

Horse racing. The horse racing approach owes its name to an example that is used to explain this method. In a series of horse races, a person (ensemble classifier) wants to predict the outcome of each race (example). The person has k experts (ensemble members) he can trust or ignore. For each race the person remembers an expert's decision and updates his trustworthiness. With each new race the set of experts the person listens to is different as he only chooses the most trustworthy advisors.

The most famous representatives of this group include the Weighted Majority algorithm [59], Hedge, Winnow [58], and Mixture of experts [46]. The horse racing approach may not be always appropriate for evolving data streams. This is because, in this approach, the individual classifiers are not retrained at any stage. Thus, after some time, the ensemble may be left without any experts adequate to the current concept because the available members have been trained on outdated data. In batch learning the experts are trained on the whole (finite) set of examples, so there is no danger of lack of expertise.

Updated training data for online ensembles. In this approach the task is to differentiate incrementally built ensemble members. This may involve sampling [65], filtering training examples for consecutive members so they specialize in different concept cases [15], and using data chunks to train the experts [34]. The proposed methods following this approach are usually variants of the corresponding batch methods for stationary environments. When faced with changing environments, we have to introduce a forgetting mechanism. One possible solution would be to use a window of past examples, preferably of variable size.

A different representative of this approach is the online bagging algorithm proposed by Oza [66]. In this method the experts are incremental learners that combine their decision using a simple 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 Poison distribution. Depending on whether the base classifier is lossless or not, this method might need a separate forgetting mechanism.

Changing the ensemble structure. Changing the ensemble structure usually involves removing a classifier to replace it with a newer one. The problem lies in the way the dropped experts should be chosen. The simplest strategy removes the oldest classifier in the ensemble and trains a new classifier to take its place. Wang et al. [78] propose a more sophisticated method which

evaluates all classifiers using the most recent chunk of data as the testing set. Street and Kim [73], on the other hand, consider a "quality score" for replacing a classifier based on its merit to the ensemble, not only on the basis of its individual accuracy. Both listed examples of ensemble structure changing algorithms will be discussed in detail in Sections 4.2 and 4.3.

Structure changing can be costly due to the process of selecting the weakest ensemble components. Nevertheless, a good selection of learners boosts classification accuracy, and sometimes even offers mathematical assurances of the new ensemble's performance.

4.2 Streaming Ensemble Algorithm

Street and Kim [73] proposed an ensemble method called Streaming Ensemble Algorithm (SEA) that changes its structure to react to changes. They propose a heuristic replacement strategy of the "weakest" expert based on two factors: accuracy and diversity. Accuracy is 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 the source of success of such ensemble methods like bagging or boosting in static environments. The pseudo-code for SEA is listed in Algorithm 4.2.

Algorithm 4.2 The Streaming Ensemble Algorithm [73]

Input: $\mathcal{S}$: a data stream of labeled examples
d: size of data chunk x_i
$Q(\cdot)$: a classifier quality measure
Output: $\mathcal{E}$: an ensemble of classifiers

1: **for all** data chunks $x_i \in \mathcal{S}$ **do**
2: build classifier C_i using x_i;
3: evaluate classifier C_{i-1} on x_i;
4: evaluate all classifiers E_j in ensemble $\mathcal{E}$ on x_i;
5: **if** $\mathcal{E}$ not full **then**
6: $\mathcal{E} \leftarrow \mathcal{E} \cup \{C_{i-1}\}$;
7: **else if** $\exists j : Q(C_{i-1}) > Q(E_j)$ **then**
8: replace member E_j with C_{i-1};

The algorithm processes the incoming stream in data chunks. The size of those chunks is an important parameter because it is responsible for the trade-off between accuracy and flexibility discussed in Section 3.2. Each data chunk is used to train a new classifier, which is later compared with ensemble members. If any ensemble member is "weaker" than the candidate classifier it is dropped and the new classifier takes its place. To evaluate the classifiers Street and Kim propose using the classification accuracy obtained on the most recent data chunk. They assign weights to components according to their accuracy and additionally diversify the candidate classifiers weight as follows:

- if both C_{i-1} and $\mathcal{E}$ are correct, then the weight of C_{i-1} is increased by $1 - |P_1 - P_2|$;
- if C_{i-1} is correct and $\mathcal{E}$ in incorrect, then the weight of C_{i-1} is increased by $1 - |P_1 - P_{correct}|$;
- both C_{i-1} is incorrect, then the weight of C_i is decreased by $1 - |P_C - P_{C_{i-1}}|$,

where $P1$ and $P2$ denote the two highest percentages of votes gained by the decision classes, $P_{correct}$ the percentage of votes of the correct decision class, and $P_{C_{i-1}}$ the percentage of votes gained by the class predicted by the new candidate classifier.

In the paper introducing SEA, the authors used *C4.5* decision trees as base classifiers and compared the ensemble's accuracy with single pruned and unpruned decision trees. SEA performed

almost as well as a pruned tree on static data sets and much better on data sets 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.

4.3 Accuracy Weighted Ensemble

A similar way of restructuring an ensemble was proposed by Wang et al. [78]. In their algorithm, called Accuracy Weighted Ensemble (AWE), they train a new classifier C' on each incoming data chunk and use that chunk to evaluate all the existing ensemble members to select the best component classifiers. Wang et al. stated and proved that for an ensemble $\mathcal{E}_k$ built from the k most recent data chunks and a single classifier G_k also built from k most recent chunks, the following theorem stands:

Theorem 4.3.1 *$\mathcal{E}_k$ 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.*

The proposed weighting provides a forgetting mechanism that is capable of handling reoccurring concept equally well to sudden drifts and periods of stability. To explain their solution, the authors of the algorithm discuss an illustrative example that presents the importance of accurate weighting of ensemble members.

Let us assume a stream of 2-dimensional data partitioned into sequential chunks based on their arrival time. Let x_i be the data that came in between time t_i and t_{i+1}. Figure 4.1 shows the distribution of the data and the optimum decision boundary during each time interval. Because the distributions in the data chunks differ, there is a problem in determining the chunks that should remain influential to accurately classify incoming data.

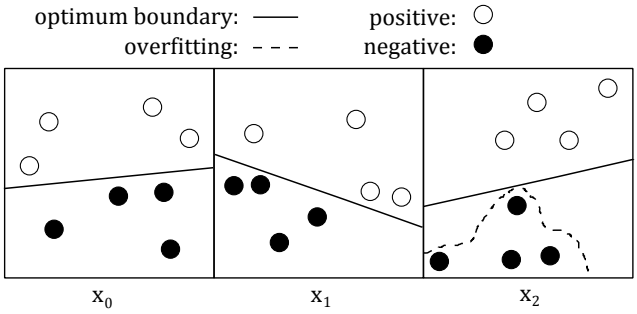

Figure 4.1: Data distribution.

Figure 4.2 shows the possible chunk sets that can be selected. The best set consists of chunks x_0 and x_2, which have similar class distribution. This shows that decisions based on example class distribution 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.

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

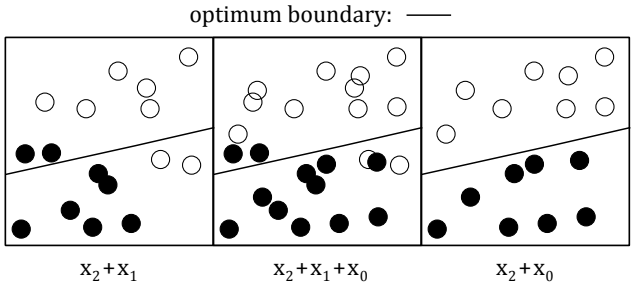

Figure 4.2: Training set selection.

According to Theorem 4.3.1, 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 chunk x_i, as shown in Equations 4.1-4.3.

$$MSE_i = \frac{1}{|x_i|} \sum_{(x,c) \in x_i} (1 - f_c^i(x))^2, \tag{4.1}$$

$$MSE_r = \sum_c p(c)(1 - p(c))^2, \tag{4.2}$$

$$w_i = MSE_r - MSE_i, \tag{4.3}$$

Function $f_c^i(x)$ denotes the probability given by classifier C_i that x is an instance of class c. This is an interesting feature of the weight calculation, as most weighting functions use only the components prediction rather than the probability of all possible classes. It is also important to note, that for the candidate classifier denoted as C', the error rate is calculated using cross validation on the current chunk to avoid overfitting. Previous ensemble members are evaluated on all the examples of the most recent data chunk. The value of MSE_r is the mean square error of a randomly predicting classifier and it is used to zero the weights of models that do not contain any useful knowledge about the data.

The complete pseudo-code for the Accuracy Weighted ensemble is listed in Algorithm 4.3.

Algorithm 4.3 Accuracy Weighted Ensemble [78]

Input: $\mathcal{S}$: a data stream of examples
d: size of data chunk x_i
k: the total number of classifiers
$\mathcal{C}$: a set of previously trained classifiers (optional)
Output: $\mathcal{E}$: a set of k classifiers with updated weights

1: **for all** data chunks $x_i \in \mathcal{S}$ **do**
2: train classifier C' on x_i;
3: compute error rate of C' via cross validation on $\mathcal{S}$;
4: derive weight w' for C' using (4.3);
5: **for all** classifiers $C_i \in \mathcal{C}$ **do**
6: apply C_i on x_i to derive MSE_i;
7: compute w_i based on (4.3);
8: $\mathcal{E} \leftarrow k$ of the top weighted classifiers in $\mathcal{C} \cup \{C'\}$
9: $\mathcal{C} \leftarrow \mathcal{C} \cup \{C'\}$

For the first k data chunks the algorithm outputs a set of all available classifiers, but when processing further chunks it selects only the k best components to form an ensemble. Wang et al.

discussed that for a large data stream it is impossible to remember all the experts created during the ensembles 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 a new parameter that would limit the number of classifiers available for selection.

The AWE algorithm works well on data streams with reoccurring concepts as well as different types of drift. As with SEA it is crucial to properly define the data chunk size as it determines the ensembles flexibility. It is also worth noticing, that AWE will improve its performance gradually over time and is best suited for large data streams.

4.4 Hoeffding option trees and ASHT Bagging

Recently, two interesting ensemble methods which use Hoeffding trees have been proposed. Kirkby [51] proposed an Option Tree similar to that of Kohavi and Kunz [53] 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 pseudo-code for the Hoeffding Option Tree is listed in Algorithm 4.4.

The algorithm works similarly to the Hoeffding Tree listed in Algorithm 3.7. The differences show from line 18 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 23) 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.

For evolving data streams the author proposes to use a windowing technique that stores an estimation of the current error at each leaf [11]. Hoeffding Option Trees offer a good compromise between accurate, but time and memory expensive, traditional ensemble methods and fast, but less accurate, single classifiers.

A different ensemble method designed strictly for Hoeffding trees was proposed by Bifet et al. [10, 9]. Adaptive-Size Hoeffding Tree Bagging (ASHT Bagging) diversifies ensemble members by using trees of different sizes. As the authors state: "The intuition behind this method is as follows: smaller trees adapt more quickly to changes, and larger trees do better during periods with no or little change, simply because they were built on more data" [10].

Apart from diversifying ensemble members, ASHT Bagging provides a forgetting mechanism. Each tree in the ensemble has a maximum size s. After a node splits, if the number of split nodes of the ASHT tree is higher than the maximum value, the tree needs to reduce its size. This can be done by either deleting the oldest node along with its children, or by deleting all the tree nodes and restarting its growth. The authors propose the maximal size of the n-th member to be twice the maximal size of the $(n-1)$-th tree. The suggested size of the first tree is 2. The weighting of a component classifier in the ensemble is proportional to the inverse of the square of its error, monitored by a exponential weighted moving average window.

Algorithm 4.4 Hoeffding option tree [51]

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

```
1: H_OT ← a tree with a single leaf l_1 (the root);
2: for all examples x_i ∈ S do
3:   Sort x_i into a leaves/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 ← 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 x_i;
10:        X_a ← the attribute with the highest G_l;
11:        X_b ← the attribute with the second-highest G_l;
12:        compute Hoeffding bound ε using (3.11);
13:        if X_a ≠ X_∅ and (G_l(X_a) − G_l(X_b) > ε or ε < τ) 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:      else
18:        if optionCount_l < k then
19:          compute G_l() for existing splits and (non-used) attributes;
20:          s ← existing child split with highest G_l
21:          X_s ← (non-used) attribute with highest G_l
22:          compute Hoeffding bound (3.11) using δ' instead of δ;
23:          if G_l(X_s) − G_l(s) > ε then
24:            add an additional child option to l that splits on X_s;
25:            for all branches of the split do
26:              add a new option leaf with initialized sufficient statistics;
27:        else
28:          remove attribute statistics stored at l;
```

When compared with Hoeffding Option Trees, ASHT Bagging proves to be more accurate on most data sets, but is extremely time and memory expensive [10]. For data intensive streams Option Trees or single classifier methods are a better choice. On the other hand, if both runtime and memory consumption are less of a concern, then variants of bagging usually produce excellent accuracies.

4.5 Accuracy Diversified Ensemble

In Section 4.3 we discussed the construction of the Accuracy Weighted Ensemble. Based on its weighting mechanism, we propose a new algorithm called Accuracy Diversified Ensemble (ADE), which not only selects but also updates components according to the current distribution.

Wang et al. designed AWE to use traditional batch classifiers as base learners. Because of this, they have to create ensemble members from single chunks and later only adjust component weights according to the current distribution. This makes the data chunk size a crucial parameter for AWE's performance. We propose to use online learners as components, so as to update existing

members rather than just adjusting their weights. This modification allows to decrease the data chunk size without the risk of creating less accurate classifiers.

Another drawback of AWE is its weight function. Because the algorithm is designed to perform well on cost-sensitive data, the MSE_r threshold in Equation 4.3 cuts-off "risky" classifiers. In rapidly changing environments (like the Electricity data set presented in Section 6.2) this threshold can "mute" all ensemble members causing no class to be predicted. To avoid this, in ADE we propose a simpler weight function presented in Equation 4.4.

$$w_i = \frac{1}{(MSE_i + \epsilon)} \tag{4.4}$$

MSE_i is calculated just like in Equation 4.1 and the ϵ component is a very small constant value, which allows weight calculation in rare situations when $MSE_i = 0$. Our slightly modified weight function prevents the unwanted "muting" in sudden concept drift situations.

As mentioned earlier we want to update base learners according to the current distribution. To introduce diversity, we do this only to selected classifiers. First of all, we only consider current ensemble members - the k top weighted classifiers. Other stored classifiers are regarded as not accurate enough to be corresponding to the current distribution. Additionally, we use MSE_r as a threshold, similarly to the way it was used in Equation 4.3 (line 12 of ADE psuedo-code). "Risky" classifiers can enter the ensemble, but will not be updated.

The diversity introduced by updating only selected components could be marginal in periods of stability. When no concept drift occurs, the classifiers trained on more examples are more accurate. The most accurate classifiers are added to the ensemble and updated with new examples. Eventually, after many data chunks of stability, the ensemble can consist of almost identical members, as they were all trained mostly on the same examples. That is why we employ online bagging, as described in Section 4.1, for updating ensemble members. This way updating examples are incrementally sampled reducing the risk of creating an ensemble of identical components.

The full pseudo-code of the Accuracy Diversified Ensemble is listed in Algorithm 4.5. The key modifications, apart from base learner and weight changes, start in line 11.

Algorithm 4.5 Accuracy Diversified Ensemble

Input: $\mathcal{S}$: a data stream of examples
d: size of data chunk x_i
k: the total ensemble members
Output: $\mathcal{E}$: a set of k online classifiers with updated weights

1: $\mathcal{C} \leftarrow \emptyset$
2: **for all** data chunks $x_i \in \mathcal{S}$ **do**
3: train classifier C' on x_i;
4: compute error rate of C' via cross validation on $\mathcal{S}$;
5: derive weight w' for C' using (4.4);
6: **for all** classifiers $C_i \in \mathcal{C}$ **do**
7: apply C_i on x_i to derive MSE_i;
8: compute weight w_i based on (4.4);
9: $\mathcal{E} \leftarrow k$ of the top weighted classifiers in $\mathcal{C} \cup \{C'\}$
10: $\mathcal{C} \leftarrow \mathcal{C} \cup \{C'\}$
11: **for all** classifiers $C_e \in \mathcal{E}$ **do**
12: **if** $w_e > \frac{1}{MSE_r}$ and $C_e \neq C'$ **then**
13: update classifier C_e with x_i using Oza online bagging;

Compared to existing ensemble methods the Accuracy Diversified Ensemble provides a new learning strategy. ADE differs from AWE in weight definition, the use of online base classifiers,

bagging, and updating components with incoming examples. Ensemble members are weighted, can be removed, and are not always updated, unlike in the online bagging approach proposed by Oza. Compared to ASHT and HOT, we do not limit base classifier size, do not use any windows, and update members only if they are accurate enough according to the current distribution.

The main concept of our approach is that only components closely related to the current distribution should be updated, and when done so, they need to be additionally diversified.

In Chapter 6 we compare accuracy, time, and memory performance of the Accuracy Diversified Ensemble with four classifiers: a windowed decision tree, a Hoeffding Tree with a drift detector, the Accuracy Weighted Ensemble, and the Hoeffding Option Tree. We check if time and memory requirements remain constant after changing AWE's batch members to incremental learners. We also verify if bagging is really a good way of boosting accuracy. To do this, we perform experiments on two versions of ADE, one with and one without bagging, and compare average results.

Chapter 5

MOA framework

Massive Online Analysis (MOA) is a software environment for implementing algorithms and running experiments for online learning [8, 12, 11]. It is implemented in Java and contains a collection of data stream generators, online learning algorithms, and evaluation procedures.

In this chapter, we discuss the components of MOA and our contributions to the framework. Section 5.1 describes stream generation, drift and noise addition, as well as discusses the performance of an attribute selection filter we implemented. Section 5.2 lists the available classification methods and describes the process of implementing a custom classifier. Finally, in Section 5.3, we present the predefined evaluation methods in MOA and our proposition to data stream evaluation - Data Chunks. Details concerning the code and execution of the implemented features can be found in Appendix A.

5.1 Stream generation and management

Work in MOA is divided into *tasks*. Main tasks in MOA include classifier training, learner evaluation, stream file generation, and stream speed measurement. Tasks can be executed from a graphical user interface (GUI) as well as from the command line. The main application window of the GUI is presented in Figure 5.1. The user interface allows to run many tasks concurrently, controlling their progress and presenting partial results.

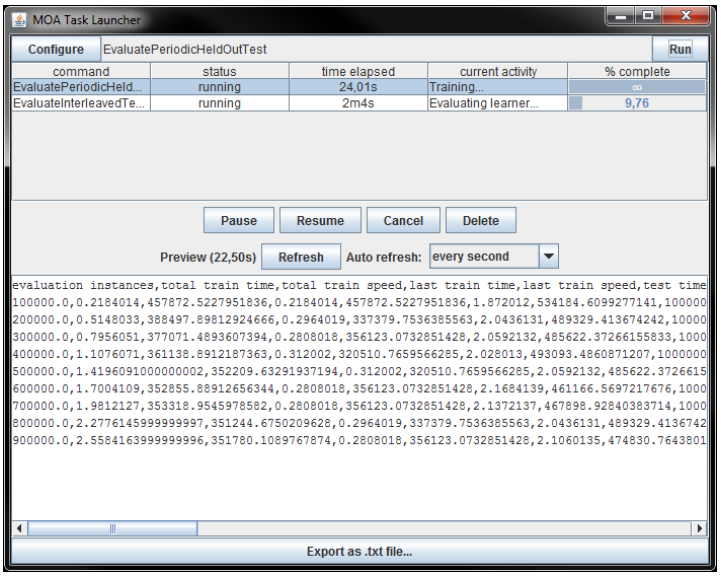

Figure 5.1: MOA main window.

MOA is capable of reading ARFF files, which are commonly used in machine learning thanks to the popularity of the WEKA project [79, 41]. It also allows to create data streams on the fly using generators, by joining several streams, or by filtering streams. The data stream generators available in MOA are: Random Trees [21], SEA [73], STAGGER [72], Rotating Hyperplane [78, 23, 24], Random RBF, LED [32], Waveform [32], and Function [48].

The framework also provides an interesting feature that allows to add concept drift to stationary data streams. MOA uses the sigmoid function to model a concept drift event as a weighted combination of two pure distributions that characterize the target concepts before and after the drift. The user can define the concept before and after the drift, the moment of the drift, and its width [12].

In its data stream processing workflow, MOA has a filtering stage. In its current release, the framework only allows to add noise to the stream. That is why we decided to implement a new filtering feature. Some real data sets for evaluating online learners like Spam Corpus or Donation (described in detail in Section 6.2) have too many attributes. We implemented a static attribute selection filter that allows to select the features that will be passed to the learner. It would be easy to implement dynamic attribute selection by performing batch attribute selection on a sample of the stream. Unfortunately, the process of filtering each example in MOA can be very time expensive. The cost of filtering attributes is so high, because it is done after the instance has been created. This means that to cut an example with 20,000 attributes to 15,000 attributes, first 20,000 attributes are read and used to create an instance, and later all those attributes are read and filtered to create a new instance with 15,000 attributes. If the framework introduced a new stage, prior to example creation, the attribute filtering process could be much more efficient.

5.2 Classification methods

The main classification methods implemented in MOA include: Naive Bayes, Hoeffding Tree, Hoeffding Option Tree, Bagging, Boosting, Bagging using ADWIN, Bagging using Adaptive-Size Hoeffding Trees, and the Weighted Majority Algorithm. The framework also allows to use a classifier from WEKA and combine any learner with a drift detector. The configuration window for an example classifier is presented in Figure 5.2.

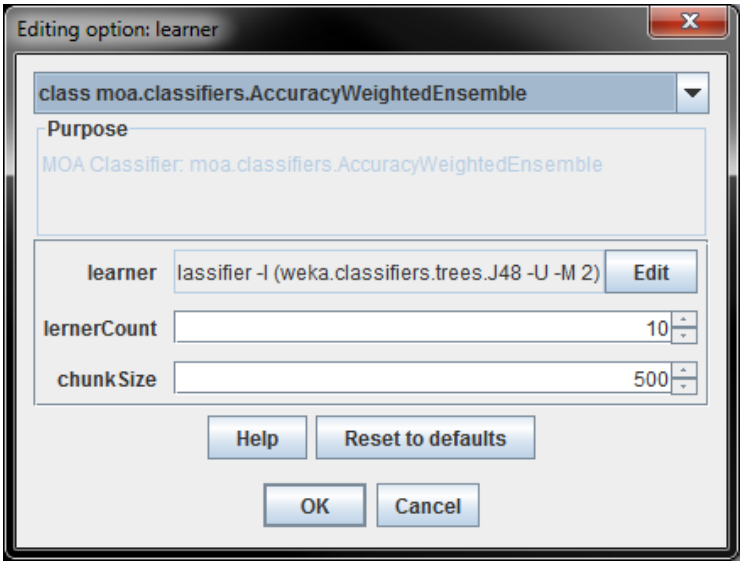

Figure 5.2: Classifier settings window.

MOA is designed to work in modular fashion allowing users to implement their own tasks and add them to the framework without much effort. Creating a new classifier requires only to extend an abstract class called `moa.classifiers.AbstractClassifier` and implement the desired algorithm. The setting windows in the GUI are created dynamically. Using this feature of MOA, we implemented the AWE and ADE algorithms described in Sections 4.3 and 4.5. The newly implemented classifiers are compared with other algorithms in Chapter 6.

5.3 Evaluation procedures

Currently MOA comes with two evaluation methods called Holdout and Interleaved Test-Then-Train. We propose a third method that evaluates classifiers using chunks of data. All three methods are described in the following subsections.

5.3.1 Holdout

In batch learning the most common classifier evaluation method is cross-validation. Unfortunately, cross-validation becomes costly for large data sets and in these cases it is often accepted to measure performance on a single holdout set. In batch learning this is done by dividing examples between train and test sets before classifier training. In "*Data Stream mining: A Practical Approach*" [11], the authors argue that data stream learning can be viewed as a large-scale case of batch learning, and that the holdout procedure is appropriate for these environments.

The MOA implementation of holdout evaluates the model periodically, at a constant user-defined interval, e.g., after every one million training examples. Testing the model too often is undesired, as it may drastically slowdown the evaluation process without providing any significant information about the tested classifier's performance.

For data streams without concept drift, a static holdout set should accurately evaluate a classifier. The only constraints concerning the test set are that it should be independent from the training set and sufficiently large, relatively to the target concepts complexity. For evolving data streams, it is necessary to dynamically populate the testing set with previously unused data. This can be done by periodically using a set of examples for testing before training.

Currently MOA provides only holdout evaluation with a static testing set. Although it is not stated anywhere in the documentation [12], the held out set is created by taking the n first examples from the stream, where n is the size of the testing set. In our opinion, this implementation cannot be used to evaluate drifting data streams, as it tests the accuracy for only the first occurring concepts.

5.3.2 Interleaved Test-Then-Train

An alternative approach to evaluating data stream algorithms involves testing the model with each incoming example and later using that example for training. This technique does not need separate memory for a test set and makes maximum use of the available data. When interleaving testing with training classifier performance can be examined with the most detailed possible resolution - for each example. That last mentioned property can be a problem, as storing statistics for each example in large data streams could be time and memory inefficient. For this reason, MOA allows to reduce the storage requirements of the results by recording statistics only at periodic intervals defined by the user.

The Test-Then-Train approach can be used equally well for static and evolving data streams. The disadvantage of this technique, compared to using a held out set, is that it is practically

impossible to correctly measure training and testing times. This method also gives obscured accuracy results, because the classifiers will make more mistakes at the beginning of the learning process.

In their report about data stream mining [11], Kirkby and Bifet compared average accuracy plots for both described evaluation techniques and opted for holdout. We believe that the holdout procedure, at least in the way it is implemented in MOA, is inappropriate for evolving data streams. We propose a compromise between testing after each example and creating a static held out set - data chunks.

5.3.3 Data Chunks

Evaluating with data chunks works similarly to the Test-Then-Train method with the difference that it uses data chunks instead of single examples. The procedure reads incoming examples without processing them, until they form a data chunk of size s. Each new data chunk is first used to test the existing model, then it updates the model, and finally it is disposed to preserve memory. Figure 5.3 gives an illustration of the data chunk evaluation method.

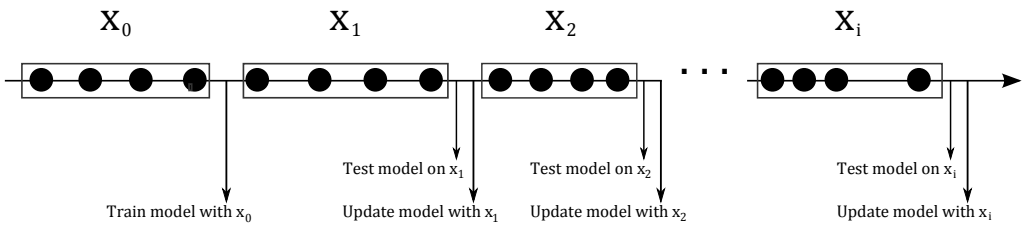

Figure 5.3: Data chunk evaluation method.

This approach allows to measure training and testing times and reduces the accuracy obscuring effect. It is suitable for static and evolving streams and provides a natural method of reducing result storage requirements. In this approach the estimate of accuracy is still more pessimistic than the one calculate via a static holdout procedure, but the ability to evaluate classifiers on drifting data streams makes it much better choice.

Chapter 6

Experimental evaluation

This chapter discusses the experiments carried out to compare selected data stream algorithms from Chapters 3 and 4. Section 6.1 lists the algorithms chosen for the comparative experiments along with their parameter settings. Section 6.2 gives insight into the main characteristics of each data set. Section 6.3 briefly describes the experimental environment. Finally, in Sections 6.4 and 6.5 we present and discuss experimental results.

6.1 Algorithms

One of the main goals of this thesis is to compare selected single classifier and ensemble methods in data stream environments. For our experiments, we chose two single classifier and four ensemble approaches, including two versions of our proposed algorithm. From single classifier methods, we chose a Hoeffding Tree with a DDM drift detector (HT+DDM) and a tree with a static window (Tree+Win). From ensemble methods, we chose the Accuracy Weighted Ensemble (AWE), the Hoeffding Option Tree (HOT), and the Accuracy Diversified Ensemble with (ADEBag) and without (ADE) bagging.

To make the comparison more significant, we tried to set the same parameter values for all the algorithms. For ensemble methods we set the number of component classifiers to 15: AWE, ADE, and ADEBag have 15 trees, HOT has 15 options. We also set the static window size to $15 \times chunkSize$ to make the number of examples seen by the windowed classifier similar to that seen by AWE and ADE. The parameters of the Hoeffding Tree with the drift detector are the same as those of the option tree, and the base classifiers (also Hoeffding Trees) of the ensembles. All these trees have Naive Bayes leaves, split confidence $\delta = 0.01$, and grace period $\gamma_r = 100$, $\gamma_a = 200$ for real and artificial data sets consequently. The secondary split confidence for HOT is set to $\delta' = 0.5$. The windowed tree is not a Hoeffding tree, but a traditional decision tree with Naive Bayes leaves.

All the algorithms are evaluated using the data chunk evaluation method described in Section 5.3.3. The data chunk size is equal $d_r = 500$ and $d_a = 1000$, for real and artificial data sets consequently. AWE uses each data chunk to build a new classifier and test those previously created, while ADE does the same with halves of the chunks. ADE can use halves of the chunks because it can incrementally update components, while a component of AWE is built on a chunk and is never updated. Data chunk sizes also defined the result sampling frequency. All the result plots consist of sampling points joined by lines, where a sampling point represents an average value (accuracy, memory or time) of 500 or 1000 examples.

6.2 Data sets

For the purposes of research into data stream classification there is a shortage of suitable and publicly available real-world benchmark data sets. Most of the common benchmarks for machine learning algorithms are not suitable for evaluating data stream classification. They contain too few examples and do not contain any type of concept drift.

To demonstrate their systems, several researchers have used private real world data that cannot be reproduced by others. Examples of this include the web trace from the University of Washington used to evaluate VFDT [21], and the credit card fraud data used by Wang et al. [78, 23, 24].

That is why, it has become a common practice by researchers to publish results based also on synthetic data sets. The authors of data stream mining algorithms have constructed unique data generation schemes for the purpose of evaluating their models. Some of the more popular generators are: Waveform, RBF, SEA, LED, Hyperplane and Random generated trees.

In our experiments we use four real and four synthetic data sets with concept drift, all of which are publicly available. A short description of each data set is given below.

Electricity market data (Elec). Electricity is a data set first described by Harries [42]. It consists of energy prices from the electricity market in the Australian state of New South Wales. These prices were affected by market demand, supply, season, weather and time of day and evolve seasonally while also showing sensitivity to short-term events. From the original data set we selected only a time period with no missing values comprised of 27,552 instances described by 7 features. Decision class values "up" and "down" indicate the change of the price and are moderately balanced (the default accuracy is 57.60%).

Ozone level detection (Ozone). Ozone is a streaming problem concerning local ozone peak prediction, that is based on eight hours measurement [81, 82]. The data set consists of 2,534 entries and is highly unbalanced (2% or 5% positives depending on the criteria of "ozone days"). The true model behind the data is stochastic as a function of measurable factors and evolves gradually over time. Another difficulty in mining this data set, is that many of the 72 features collected for each instance are irrelevant.

Spam Assassin corpus (Spam). Spam is a series of entries of real-world spam and legitimate emails chronologically ordered according to their date and time of arrival [49]. The data set consists of 9,324 instances and initially 40,000 features selected from the *Spam Assassin* (http://spamassassin.apache.org/) data collection. The data is unbalanced (20% of spam, 80% of legitimate emails) and represents gradual concept drift.

Donation data (Don). Donation is a data set used for The Second International Knowledge Discovery and Data Mining Tools Competition. It was also used by Wei Fan [23] to evaluate his *Systematic data selection technique*. The data represents a regression problem where the goal is to estimate the return from a direct mailing in order to maximize donation profits. In this data set there are almost 200,000 instances described by 479 features. The data contains examples of sudden concept drift.

LED Generator (Led). LED is popular artificial data set that originates from the CART book [16]. It consists of a stream of 24 binary attributes, 17 of which are irrelevant, that define the digit displayed on a seven-segment LED display. The data set is known to have an optimal

Bayes classification rate of 74%. We use this generator to acquire 1,000,000 examples with sudden and gradual concept drift.

Waveform (Wave). Waveform is an artificial data set used by Gama et al. [32]. It consists of a stream with three decision classes where the instances are described 40 attributes. It is known that the optimal Bayes error for this data set is 14%. This generator is used to acquire 1,000,000 instances with gradual concept drift.

Hyperplane (Hyp). Hyperplane is a popular data set generator used in many experiments [78, 23, 24, 82]. It is mainly used to generate streams with gradual concept drift by rotating the decision boundary for each concept. We set the generator to create 5,000,000 instances described by 10 features and 2 decision class values. We also add 10% of noise to the concepts to randomly differentiate the instances.

SEA generator (Sea). SEA was proposed by Street and Kim [73]. The data set has two decision classes with sudden concept drift and is created by generating 60,000 random points in a three-dimensional feature space. All three features have values between 0 and 10. The generator then divides those points into four blocks with different concepts and assigns classes according to a linear function. For the experiments in this thesis, we generate a data stream of 20,000,000 instances with 10% of noise.

6.3 Experimental environment

All the tested algorithms were implemented in Java as part of the MOA framework. We implemented the AWE and ADE algorithms, and the Data Chunk evaluation procedure. All the other algorithms were already a part of MOA. The experiments took place on a machine equipped with an Intel Pentium Core 2 Duo P9300 @ 2.26 GHz processor and 3.00 GB of RAM. Each algorithm was tested on 8 data sets, described in the previous section, using the Data Chunk evaluation procedure.

6.4 Results

According to the main characteristics of data streams described in Chapter 2, we divide the results into three groups. First, in Section 6.4.1, we analyze the algorithms' time performance. We compare train and test times, and verify if the constant processing time requirement is met. In Section 6.4.2, we discuss the algorithms' memory usage over time. Finally, in Section 6.4.3 we compare the classification accuracies achieved by all the algorithms.

The most common way of displaying results in data stream mining papers is a graphical plot, typically with the number of training examples on the x-axis [11]. During our experiments, we generated 4 plots for each data set: train time, test time, memory usage, and accuracy. The most interesting figures will be discussed in the following sections along with tabular summaries. All the generated plots are given in Appendix B.

6.4.1 Time analysis

Tables 6.1 and 6.2 present average chunk train and test times for each data set. We can see that single classifiers process data streams much faster than ensemble methods. This is quite natural as simpler learners usually require less time for training and testing.

Table 6.1: Average test and train times in ms for data chunks in real data sets.

	Elec		Ozone		Spam		Don	
	Train	Test	Train	Test	Train	Test	Train	Test
HOT	101.11	6.64	62.40	**1.00**	13160.98	1430.62	2168.54	17.19
AWE	56.91	13.29	179.40	27.30	-	-	3290.93	1074.98
HT+DDM	**4.04**	**0.58**	19.50	7.80	**7100.80**	**2921.81**	53.17	**5.55**
Tree+Win	20.51	1.44	**15.60**	7.80	-	-	**1.55**	17.02
ADE	75.11	15.31	241.80	54.60	-	-	3292.64	1086.82
ADEBag	72.22	15.31	237.90	54.60	-	-	3351.61	1090.17

Table 6.2: Average test and train times in ms for data chunks in artificial data sets.

	Led		Wave		Hyp		Sea	
	Train	Test	Train	Test	Train	Test	Train	Test
HOT	563.68	**9.93**	2573.86	51.27	5170.84	14.18	4876.41	6.44
AWE	751.40	181.85	558.21	159.19	41.13	**4.96**	29.44	5.61
HT+DDM	**22.44**	12.07	140.60	**8.07**	1998.79	7.95	2125.06	**2.40**
Tree+Win	23.08	200.19	**25.44**	18.41	**28.24**	8.45	**25.63**	2.66
ADE	803.93	185.39	636.59	162.06	240.22	49.39	90.05	18.84
ADEBag	798.01	184.64	646.62	164.53	251.89	53.05	95.68	19.67

An interesting observation, that concerns not only time experiments, is that due to the large number of attributes in the Spam data set (20,000 attributes) AWE, ADE, ADEBag, and Tree+Win were unable to process that stream. To explain this, let us notice that the Naive Bayes tree that was windowed is not an incremental learner and builds complex models more quickly than Hoeffding trees. That is why it used more memory than the Java heap size limit. Similarly, ensemble methods build 15 models instead of 1 and also fail to process the Spam data set with the available memory.

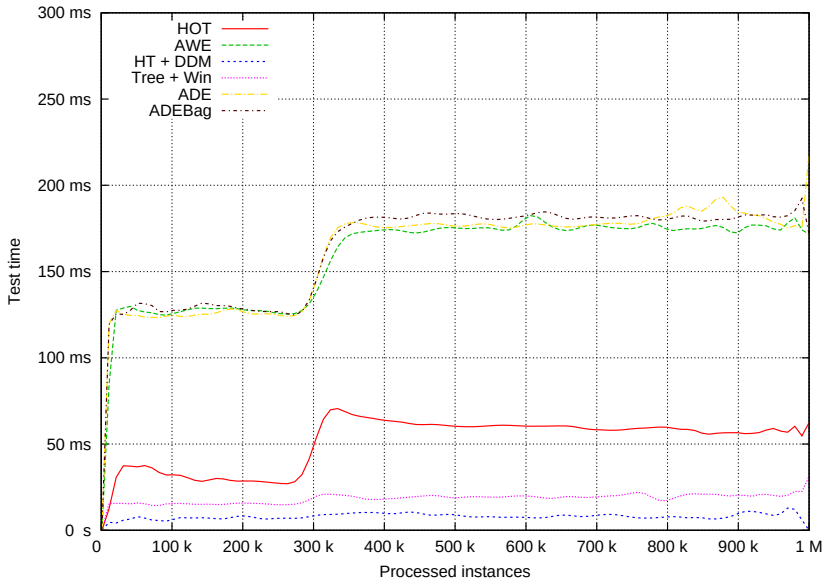

Figure 6.1: Chunk test time on the Waveform data set. Constant testing time for all the algorithms and a visible example of concept drift.

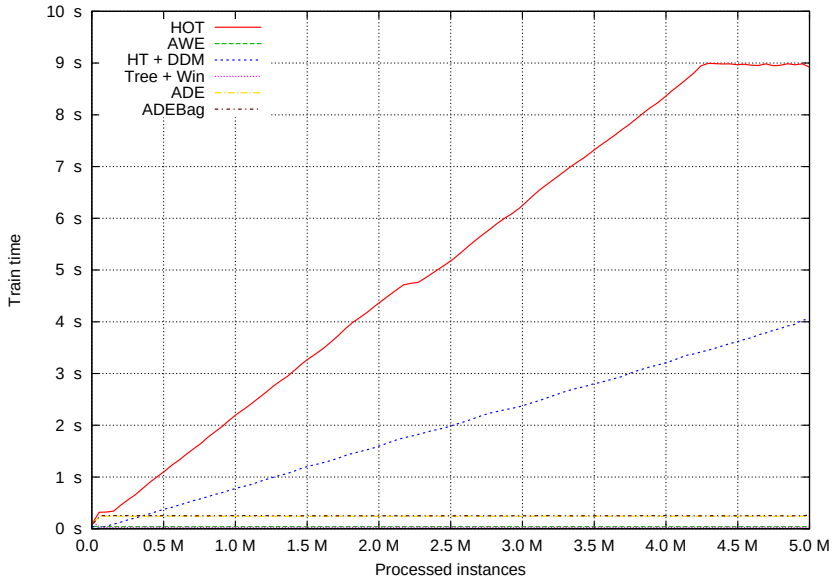

Figure 6.2: Chunk train time on the Waveform data set. An example of linear training time growth for HOT and HT+DDM.

Looking at test time plots, like the one presented in Figure 6.1, we can notice that testing times remain close to constant through out the whole processing of the data stream. This observation is true for all data sets (see plots B.9-B.16). On the other hand, classifier training is not constant for all algorithms (see plots B.1-B.8). HOT shows clear linear growth of training time when no sudden drift occurs. HT+DDM keeps low training time for smaller data sets, but also requires linearly more time for large data sets. An example of the linear growth of training time for HOT and HT+DDM can be seen in Figure 6.2.

The growth of training time for HOT and HT+DDM is due to the fact that we did not restrict maximum model memory. We did so to test the algorithms in an environment optimal for their performance. Hoeffding trees, like most decision trees, become more complex as they see more examples. In periods of stability HOT and HT+DDM will successively grow bigger trees, thus consuming more memory. The windowed tree, AWE, and ADE are built from a limited number of data chunks and have an architectural memory limit. Theoretically ADE and ADEBag components could become more complex in periods of stability, but as these experiments show this is practically impossible.

Single classifiers seem to be the best choice when processing time is of crucial importance. For large data sets, AWE, ADE and ADEBag come close second, as HOT and HT+DDM clearly lose on this criterion, gradually requiring more and more processing time.

6.4.2 Memory usage

According to Table 6.3, HT+DDM used the least memory for small data sets, but consumed more resources than AWE, ADE, and ADEBag for the three largest data streams. Sadly, due to bad wrapping between MOA and WEKA objects, the framework outputted incorrect model size values for the windowed tree. We believe, that its size should remain close to constant after a window of examples. As this approach builds only one classifier, it is very probable that the windowed tree would be the most memory effective approach for large data streams.

Table 6.3: Average trained model size for all data sets measured in MB.

	Elec	Ozone	Spam	Don	Led	Wave	Hyp	Sea
HOT	0.41	0.08	29.78	13.97	2.76	12.27	18.49	24.45
AWE	0.23	0.28	-	5.52	0.58	**0.33**	**0.17**	**0.18**
HT+DDM	**0.02**	**0.03**	**23.69**	**0.66**	**0.06**	1.17	14.05	21.54
Tree+Win	-	-	-	-	-	-	-	-
ADE	0.36	0.36	-	5.64	0.88	0.92	0.86	0.46
ADEBag	0.30	0.36	-	5.64	0.95	1.00	0.88	0.48

An interesting fact can be noticed when analyzing Figures 6.3 and 6.4. For small data sets (Elec, Ozone, Spam, Don, Wave) the Hoeffding Option Tree clearly requires linearly more memory with each processed data chunk. Interestingly, for the largest data sets (Hyp, Sea) HOT reaches a point where the model's size remains constant. We did not limit memory usage for these experiments, so this must mean that the model's ability to evolve has reached its limit. After expanding all 15 options for each node the tree might not be able grow any more.

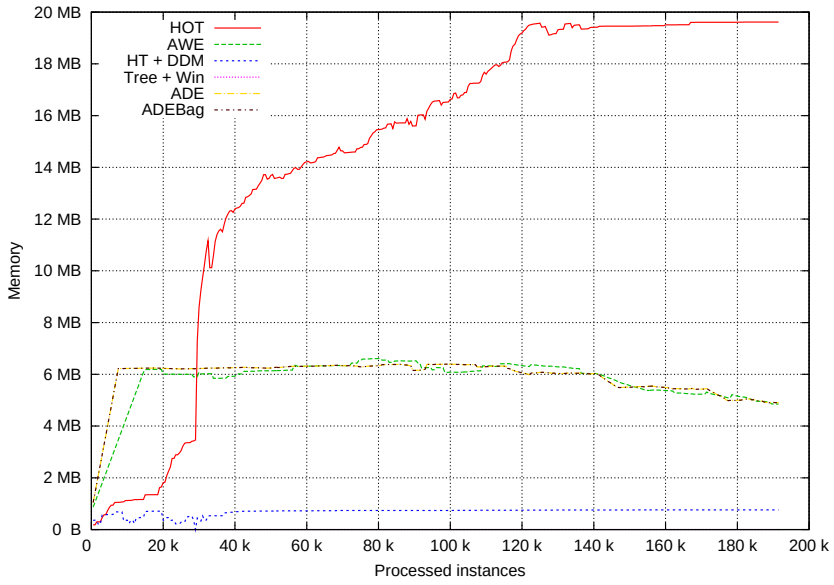

Figure 6.3: Memory usage on the Donation data set. Linear growth of HOT and relatively large memory usage of ensemble methods compared to HT+DDM.

The analysis of all memory plots (Figures B.17-B.24) shows that memory requirements are similar to training time requirements. HT+DDM and HOT need much more memory for larger data sets than Tree+Win, AWE, ADE, and ADEBag, which processed the data streams using constant memory.

6.4.3 Classification accuracy

Table 6.4 presents average classification accuracies obtained by the tested algorithms on all the data sets. Average accuracy is a good measure for evaluating overall performance, but in evolving environments, the classifiers reaction to change is of crucial importance. That is why we analyze more thoroughly two plots presenting the reaction of algorithms to gradual and sudden concept drift.

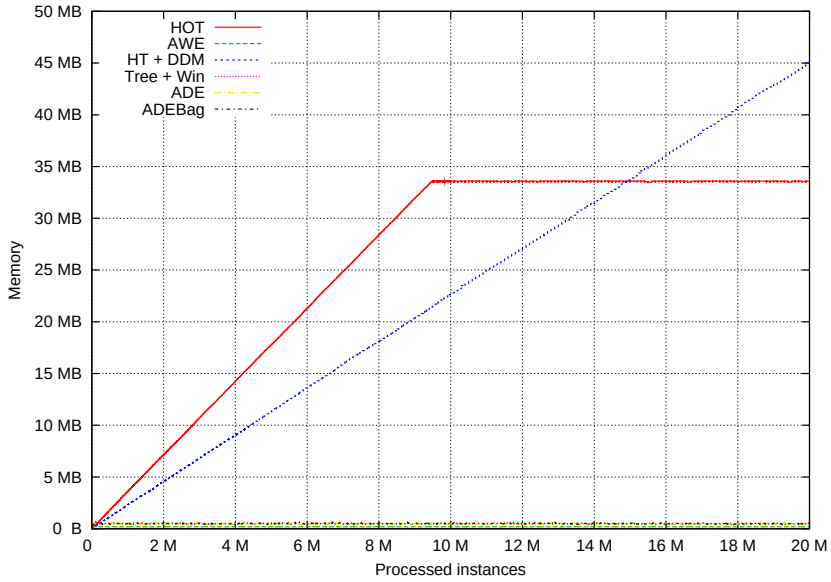

Figure 6.4: Memory usage on the SEA data set. An example of linear memory growth for HOT and HT+DDM, and a case of HOT reaching its option limit.

Table 6.4: Average accuracy for all data sets in percent.

	Elec	Ozone	Spam	Don	Led	Wave	Hyp	Sea
HOT	74.37	91.60	**75.25**	94.35	70.68	**82.57**	**85.07**	**89.81**
AWE	71.22	67.59	-	94.35	71.16	79.63	70.38	78.52
HT+DDM	70.04	84.29	67.03	**94.35**	69.93	81.27	84.40	89.54
Tree+Win	66.32	**91.60**	-	89.18	67.76	58.60	72.27	83.02
ADE	**74.92**	76.56	-	94.34	71.41	82.26	84.72	88.61
ADEBag	74.25	76.56	-	94.34	**71.42**	82.50	84.98	88.75

Figure 6.5 shows the accuracy plot for the Waveform data set, where gradual drift occurs around the 300,000th example. Most of the tested classifiers react to the change with a short drop in accuracy, which is later corrected after adjusting to the new concept. The only approach that fails to successfully cope with gradual change is the windowed tree. Because its forgetting process is static, it has no chance of unlearning outdated examples quickly, and since the drift is spread in time, its decrease in accuracy is longterm.

More complex concept drift was introduced in the generation of the LED data set. We joined two gradually evolving LED data sets with a sudden change. After half million examples we replaced one data source with another. The algorithms' reactions to this type of change are presented Figure 6.6.

We see that all classifiers become less accurate after the sudden drift. Once again the windowed tree suffers the most, but the accuracy drop is not as drastic as it was for the Waveform data set. For this, more complex, concept drift other algorithms also have problems with adjusting to change. ADE and ADEBag seem to cope best with this situation. HOT, which performed well before the drift, falls down even below the level of AWE.

In periods of stability, HOT and HT+DDM grow accurate but complex structures, which are later difficult to rebuild. AWE, ADE, and ADEBag are modular, allow quick substitution of components, and therefore quick reaction to sudden drifts.

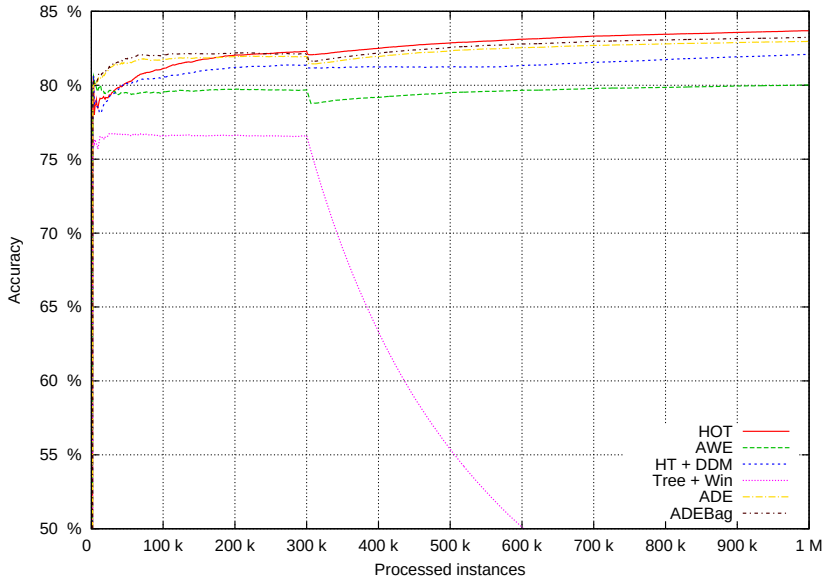

Figure 6.5: Accuracy on the Waveform data set.

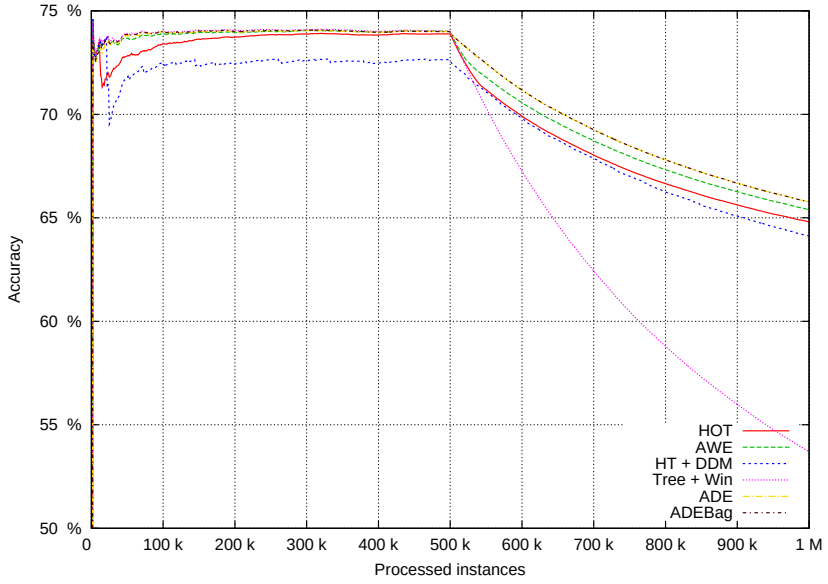

Figure 6.6: Accuracy on the LED data set.

It is hard to select one most accurate classifier from the compared algorithms. For large data streams with little and mostly gradual concept drift, HOT gives best results with ADEBag and HT+DDM being close second. For the smallest data set, the windowed tree works best. For the most rapidly changing streams (Elec, LED), the ADE approaches proved to be the most flexible. We clearly see that depending on time and memory requirements, as well as predicted types of drift, different approaches work better.

6.5 Remarks

Recently, Bifet et al. [10] showed a comparison of many online learning algorithms, including HOT and HT+DDM. Their results, concerning these two classifiers, our similar to ours: HT+DDM is faster and much more memory efficient than HOT. The only difference is that on the majority of their data sets HT+DDM classified better, while our experiments proved HOT to be more accurate. This difference may be dependent on the data streams they used for evaluation. During their tests, Bifet et al. generated two streams of 10 million and two of 1 million data points, whereas we tested a stream up to 20 million examples.

Our experiments, like most data stream classifier evaluations [10, 44, 30, 78], measure memory requirements of algorithms rather than examining their performance in environments with insufficient RAM. Additional tests need to be performed in order to see if HOT is equally accurate when it has only the memory used by ADE or HT+DDM.

The comparison of AWE with HOT and HT+DDM was, to our knowledge, never done before. Although AWE is more flexible when handling sudden drift presented in LED, HOT and HT+DDM outperform the weighted ensemble on all the other data sets. AWE was previously compared with a decision tree built on a window of examples [78] with results similar to ours. In most cases, weighting chunks of examples according to distribution improves classifier accuracy.

The Accuracy Diversified Ensemble is our methodological contribution and therefore was not previously compared with any other algorithm. ADE was more accurate than AWE on all but one data set whilst still requiring constant processing time and memory. We find it to be a promising compromise between accurate but time and memory costly HOT and HT+DDM, and the slightly lighter but much less accurate AWE and simple windowing technique. Further experiments, especially simulating limited memory environments, need to be performed to fully confirm our approach's usefulness.

Discussing the design of the Accuracy Diversified Ensemble, we were not sure if bagging was really the best choice to additionally diversify ensemble members. We decided to test our approach in two versions - with and without online sampling. Table 6.5 compares the average performance of both methods on all data sets.

Table 6.5: Average accuracy, time, and memory performance for ADE and ADEBag.

	Chunk train	Chunk test	Memory	Accuracy
ADE	-	-	-	+0.11%
ADEBag	+3.07%	+5.59%	+4.90%	-

We see that the ADE version with bagging requires more time and memory. This is a small overhead caused by the sampling process. What is surprising, bagging does not improve accuracy on an average. ADEBag had better accuracy on 4 data streams, was equally good three times, and lost only on the Electricity data set. The fact that ADE was much better on that last mentioned data set made the result in Table 6.5 unfavorable for the version with bagging.

For the majority of data sets bagging did not change or improved the accuracy of the ADE very slightly. We believe that adding more diversity ADE components could make the classifier more accurate, but different methods than bagging (e.g. the use of different base classifiers or boosting) have to be explored.

Chapter 7

Conclusions

In this thesis we addressed the problem of mining time evolving data streams. We defined the main characteristics of data streams and discussed different types of changes that occur in streaming data. During our discussion we focused on non-random class definition changes called concept drift. We reviewed existing single classifier and ensemble approaches to mining data streams with concept drift. Our analysis led to the development of a new algorithm called Accuracy Diversified Ensemble, which is based on our critique of the earlier developed Accuracy Weighted Ensemble.

Moreover, one of the aims of this thesis was the evaluation of the Massive Online Analysis framework as a software environment for research on learning from evolving data streams. MOA is a relatively young project and still needs some work to become a reliable data stream development tool. MOA's modular structure and dynamic graphical interface creation similar to WEKA are good decisions. But even with its close relation to WEKA, MOA sometimes has trouble with communicating with its relative. Not being able to correctly determine model size when using a WEKA classifier is one of the cases. A different limitation is the inability of a MOA wrapper in WEKA to handle class attributes other than nominal.

The authors of MOA wrote a manual [12] and a technical report [11], which document the frameworks usage and theoretical foundations. On the other hand, MOA's source code is not documented. This makes code re-usage more difficult and does not facilitate the frameworks future development. A different technical remark concerns the result generation in MOA. When evaluating a classifier in MOA, results appear incrementally as the stream is being processed. If at one point of the data stream the classifier or framework crashes, all results are lost. This is an implementation decision, as results acquired prior to the crash could be easily saved. In our opinion, outputting partial results with additional error information would be a better solution.

MOA has had only one stable release to date. Nevertheless, it contains 8 of the most popular data stream generators, 11 classifiers, 3 evaluation methods and many other functions implemented in a single package. MOA is surely the most comprehensive benchmark environment dedicated solely to data stream mining. Despite its drawbacks, there is a chance that with the effort of its authors it will become a commonly known framework for mining concept drifting streams, like WEKA is for traditional data mining.

We extended the MOA framework by implementing our Accuracy Diversified Ensemble, the Accuracy Weighted Ensemble, and a data chunk evaluation method to experimentally compare selected single and ensemble data stream classifiers. We found that the classifier proposed by us requires constant time and memory, and gives comparably good or better accuracy than more resource expensive methods. Additionally, during the evaluation of our algorithm, we verified the

usefulness of bagging as a way of additionally diversifying our ensemble. We found that bagging provided time and memory overhead without improving accuracy.

Our experimental results can be partially compared with previous publications. Findings concerning the windowed classifier, Hoeffding Option Tree and Hoeffding Tree with a drift detector, are similar to those stated by Bifet et al. [10]. The main difference in results is that in our setting the Hoeffding Option Tree was practically always more accurate than the single Hoeffding Tree with a drift detector, while the tests of Bifet et al. showed a slight domination of the single classifier. Our comparison of the Accuracy Weighted Ensemble and the Accuracy Diversified Ensemble with different algorithms provides new, previously unpublished results.

The analysis of algorithms reviewed in this thesis shows that mining of data streams with concept drift is growing into a new branch of knowledge discovery, with its own unique research problems. The need for online processing with time and memory constraints forces researchers to focus on resource usage while designing accurate classifiers. Additionally, concept drift introduces the requirement for a forgetting mechanism that dynamically removes outdated data. The MOA framework proposes a way of unifying the implementation and evaluation of algorithms that tackle these problems. This shows that data stream mining is becoming a mature field of study aiming to meet the challenges of the data stream phenomenon in real life applications.

As future work, we plan to carry out additional experiments to analyze the relationship between data chunk size and the performance of bagging in the Accuracy Diversified Ensemble. Furthermore, since bagging did not prove to provide additional accuracy to our method, we also plan to explore different ways of diversifying ensemble members like the use of heterogeneous base learners or boosting. Finally, in future experiments we plan to compare a larger number of algorithms to take a broader look at the performance of the most recent stream mining techniques.

Appendix A

Implementation details

In this appendix we describe the installation process and the most important implementation details of the software attached to this thesis. Section A.1 lists software requirements and presents the installation process. Section A.2 discusses the implementation of the Attribute Filter, Section A.3 the implementation of the Data Chunk Evaluation procedure, Section A.4 the implementation of the Accuracy Weighted Ensemble algorithm. Finally, in Section A.5 we shortly lists the parameters of our algorithm - the Accuracy Diversified Ensemble.

A.1 MOA Installation

MOA is written in Java and requires at least JRE 5 for running and Java 5 SDK for development. Due to Java's portability, MOA can be run on Windows, Mac and Unix/Linux systems. The framework requires the following files:

- moa.jar (http://sourceforge.net/projects/moa-datastream/)
- weka.jar (http://sourceforge.net/projects/weka/)
- sizeofag.jar (http://www.jroller.com/resources/m/maxim/sizeofag.jar)

An extended version of MOA is provided with this thesis. To run a MOA task from the command line on Windows, use the following pattern:

```
java.exe -cp <moaFolder>\moa.jar -javaagent:<sizeofagFolder>\sizeofag.jar
moa.DoTask <taskName> <taskParameters>
```

For example:

```
java.exe -cp "C:\moa.jar" -javaagent:"C:\sizeofag.jar" moa.DoTask
EvaluateInterleavedChunks -l AccuracyWeightedEnsemble -i 1000000
-s generators.WaveformGenerator
```

To run the graphical interface:

```
java -cp <moaFolder>\moa.jar -javaagent:<sizeofagFolder>\sizeofag.jar
moa.gui.TaskLauncher
```

The MOA distribution comes with two run files (moa.sh and moa.bat), which ease the execution of the graphical interface. More details concerning the use of the graphical interface and command line can be found in the Masive Online Analysis Manual [12].

A.2 Attribute filtering

The Attribute Filter allows to select the attributes of each example that will be passed to the learner. For streams with many features the filtering process can significantly slow down stream processing, but allows to decrease model size. The attributes to be removed are specified by indexes of the unwanted attributes. The user can list single attributes separated by commas (1,4,12) or define ranges of attributes (5-8,13-41).

The filter is implemented in the file `RemoveAttributesFilter.java` placed in the `moa.streams.filters` package of the MOA framework. `RemoveAttributesFilter` extends the `AbstractStreamFilter` class and can be only used with the Filtered-Stream method.

Parameters:

- -a: Indexes of attributes to be removed (-1 = no filtering)

Example usage:

```
   java.exe -cp moa.jar -javaagent:sizeofag.jar moa.DoTask LearnModel
-s (FilteredStream -s (ArffFileStream -f (ozone.arff))
-f (RemoveAttributesFilter -a 1))
```

A.3 Data chunk evaluation

The Data Chunk Evaluation procedure evaluates a classifier on a stream by testing then training with consecutive data chunks. In the implementation, accuracy, time, and memory are updated with each example in the data chunk and later averaged. Similarly, in the training phase, the classifier is trained incrementally (when possible), but the created model is available for further processing after the whole data chunk rather than after each example. Thus, the sampling interval should be equal or greater than the data chunk size.

The evaluation method is implemented in the file `EvaluateInterleavedChunks.java` placed in the `moa.tasks` package of the MOA framework. `EvaluateInterleavedChunks` extends the `MainTask` class and is available directly from the task selection combo box in the graphical interface.

Parameters:

- -l : Classifier to train
- -s : Stream to learn from
- -e : Classification performance evaluation method
- -i : Maximum number of instances to test/train on (-1 = no limit)
- -c : Number of instances in a data chunk
- -t : Maximum number of seconds to test/train for (-1 = no limit)
- -f : How many instances between samples of the learning performance
- -b : Maximum byte size of model (-1 = no limit)
- -q : How many instances between memory bound checks
- -d : File to append intermediate csv results to

- -O : File to save the final result of the task to

Example usage:

```
java.exe -cp moa.jar -javaagent:sizeofag.jar moa.DoTask
EvaluateInterleavedChunks -l HoeffdingTreeNB -s generators.WaveformGenerator
-i 1000000 -c 1000
```

A.4 Accuracy Weighted Ensemble

The Accuracy Weighted Ensemble was implemented according to the pseudo-code listed in Algorithm 4.3. Instance based pruning and other enhancements for cost-sensitive applications [78] were not implemented. The number of folds parameter is used only for candidate classifiers built from the most recent data chunk. Previously built classifiers are tested on the entire, most recent data chunk. The train and test sets for cross-validation are created by the `trainCV()` and `testCV()` methods implemented in WEKA.

The classifier is implemented in the file `AccuracyWeightedEnsemble.java` placed in the `moa.classifiers` package of the MOA framework. `AccuracyWeightedEnsemble` extends the `AbstractClassifier` class and is available from the learner selection dialog in the graphical interface.

Parameters:

- -l : Member classifier type
- -n : Maximum number of classifier in an ensemble
- -r : Maximum number of classifiers to store and choose from when creating an ensemble
- -c : Chunk size used for member creation and evaluation
- -f : Number of cross-validation folds for candidate classifier testing

Example usage:

```
java.exe -cp moa.jar -javaagent:sizeofag.jar moa.DoTask
EvaluateInterleavedChunks -l (AccuracyWeightedEnsemble -n 20 -c 1000)
-s generators.WaveformGenerator -i 1000000 -c 1000
```

A.5 Accuracy Diversified Ensemble

The Accuracy Diversified Ensemble was implemented according to the pseudo-code listed in Algorithm 4.5 with an option that determines whether or not to perform bagging.

The classifier is implemented in the file `AccuracyDiversifiedEnsemble.java` placed in the `moa.classifiers` package of the MOA framework. `AccuracyDiversifiedEnsemble` extends the `AccuracyWeightedEnsemble` class and shares all its parameters.

Parameters:

- Same parameters as AccuracyWeightedEnsemble
- -b : If set, no bagging is performed
- -a : If set, a bagged example is always presented to the classifier (always add 1 to the sampled Poison distribution)

Example usage:

```
   java.exe -cp moa.jar -javaagent:sizeofag.jar moa.DoTask
EvaluateInterleavedChunks -l (AccuracyDiversifiedEnsemble -a -n 20 -c 1000)
-s generators.WaveformGenerator -i 1000000 -c 1000
```

Appendix B

Additional figures

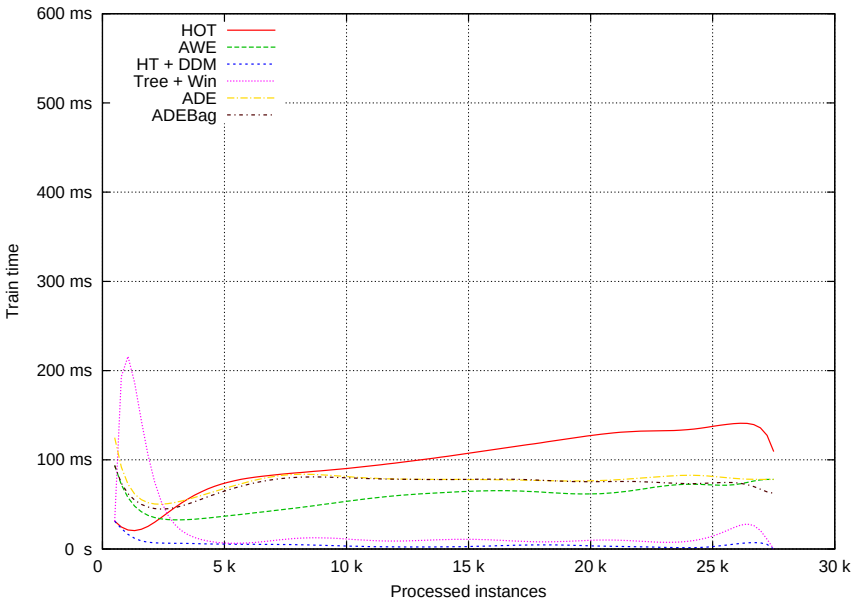

Figure B.1: Chunk train time on the Electricity data set.

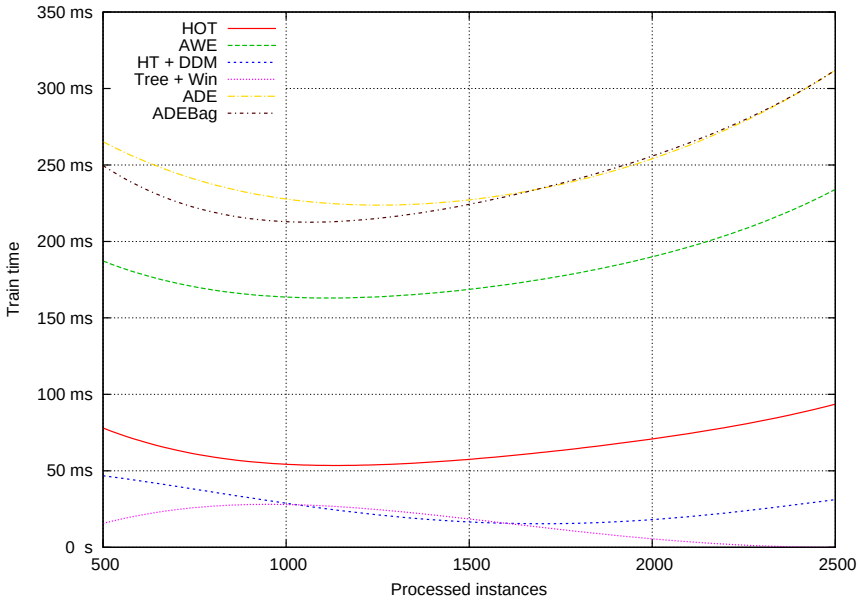

Figure B.2: Chunk train time on the Ozone data set.

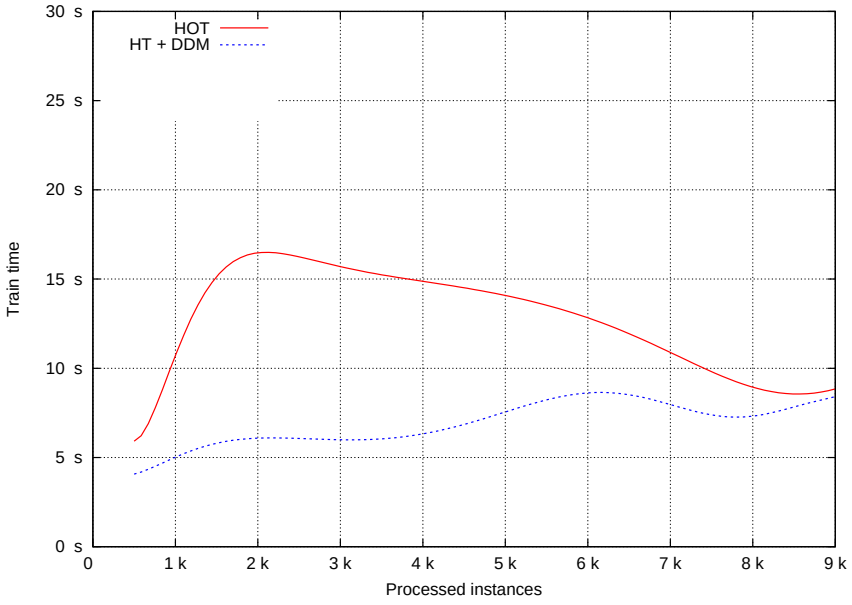

Figure B.3: Chunk train time on the Spam data set.

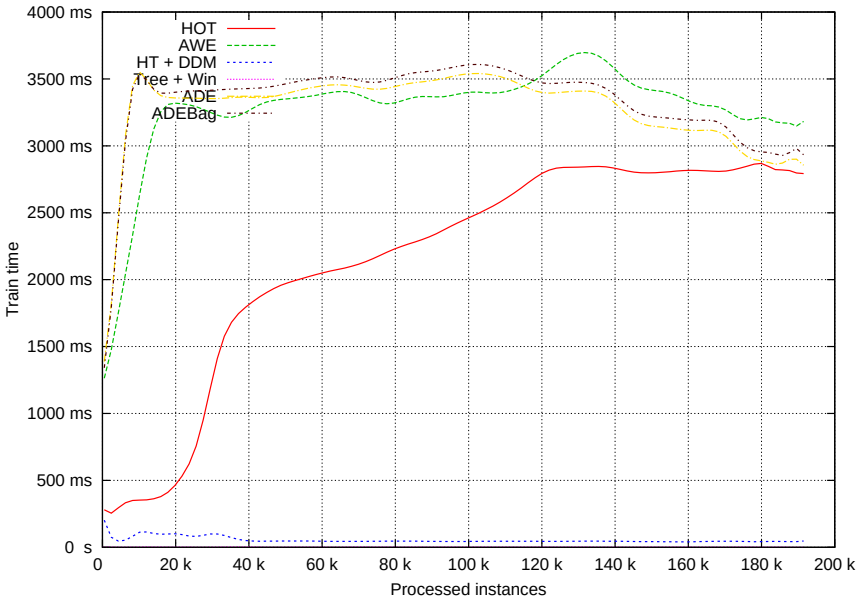

Figure B.4: Chunk train time on the Donation data set.

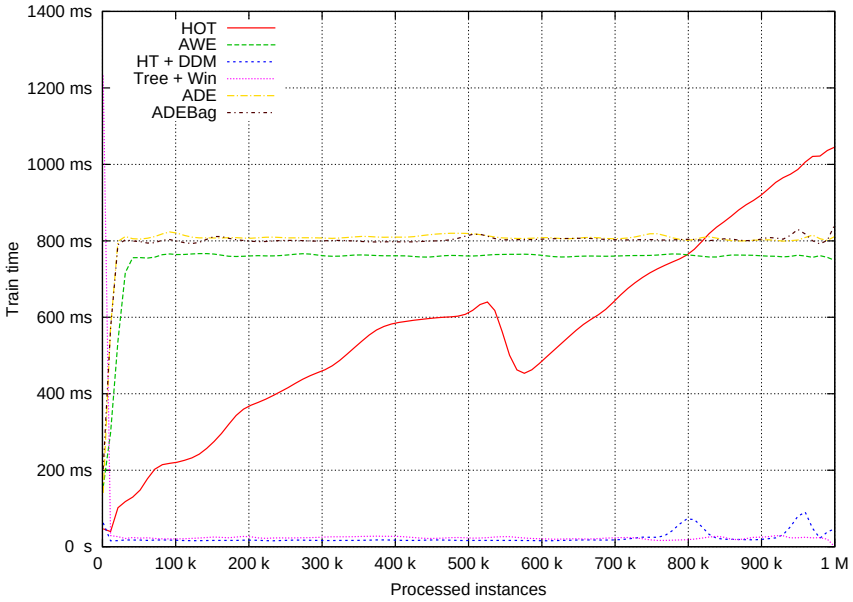

Figure B.5: Chunk train time on the LED data set.

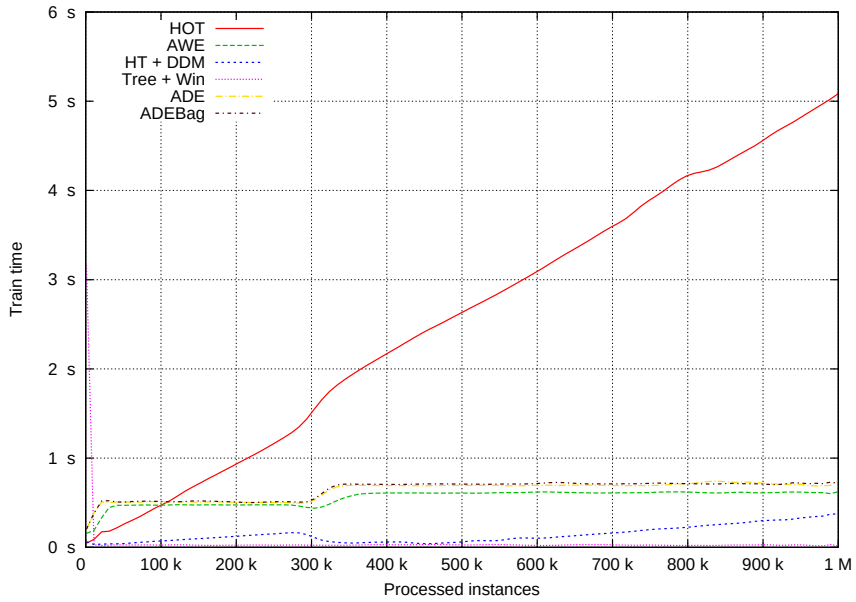

Figure B.6: Chunk train time on the Waveform data set.

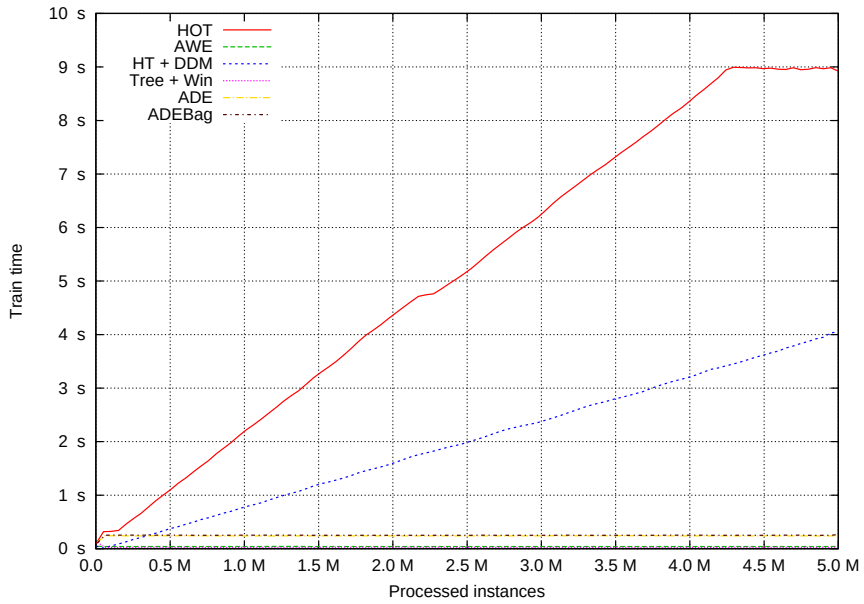

Figure B.7: Chunk train time on the Hyperplane data set.

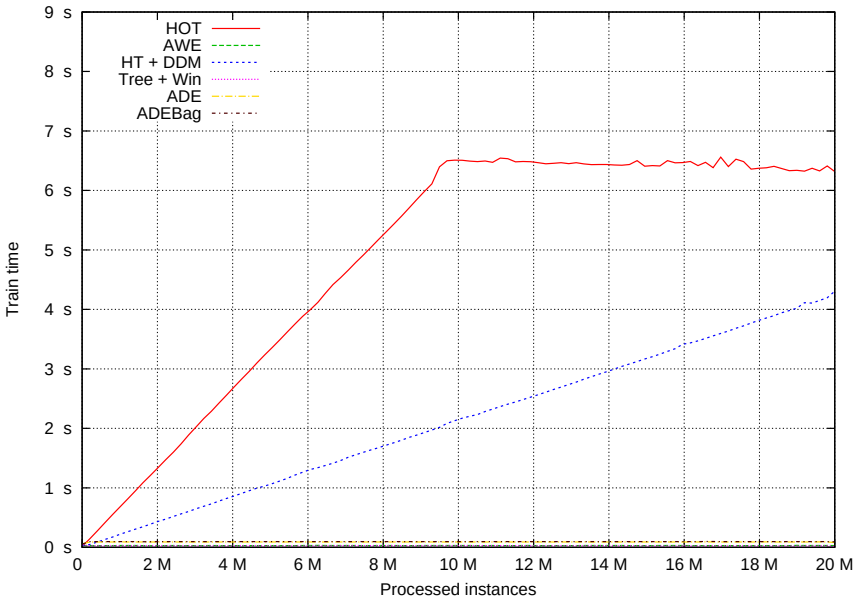

Figure B.8: Chunk train time on the SEA data set.

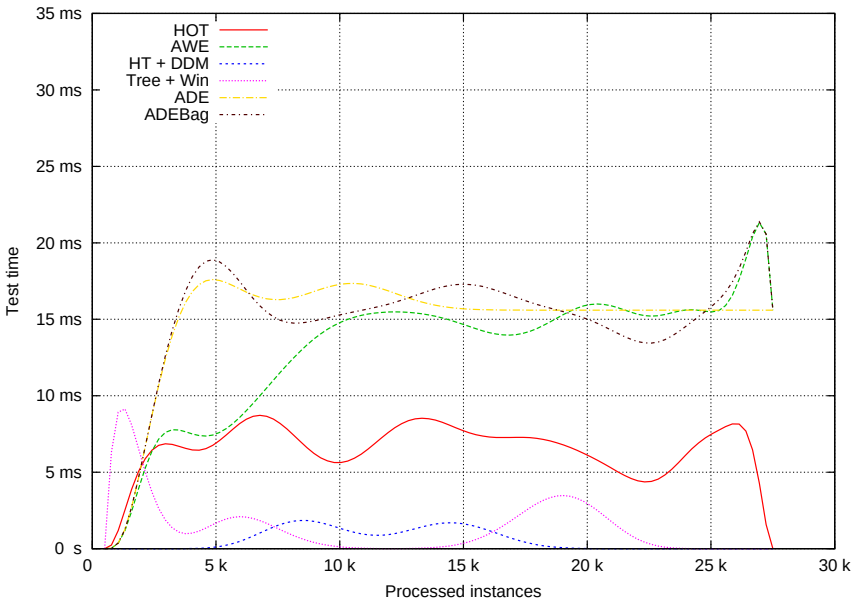

Figure B.9: Chunk test time on the Electricity data set.

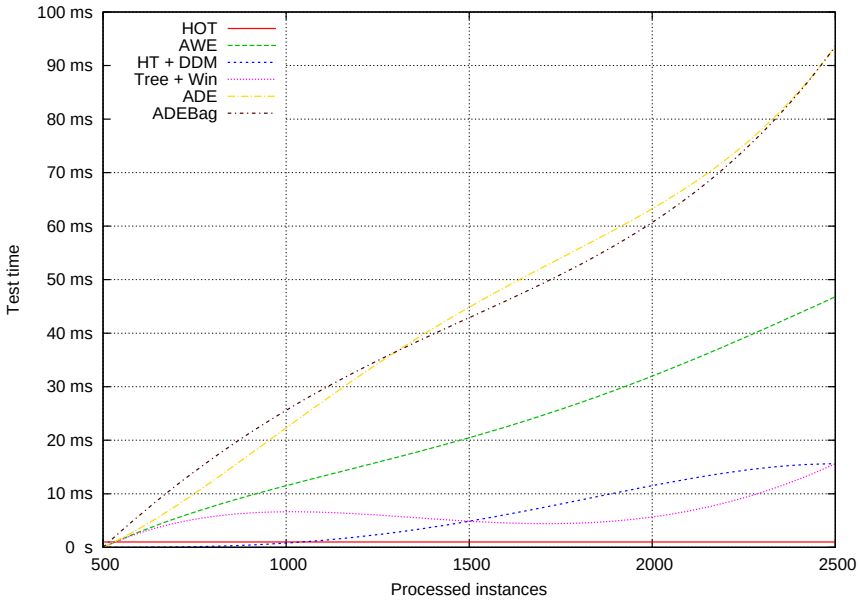

Figure B.10: Chunk test time on the Ozone data set.

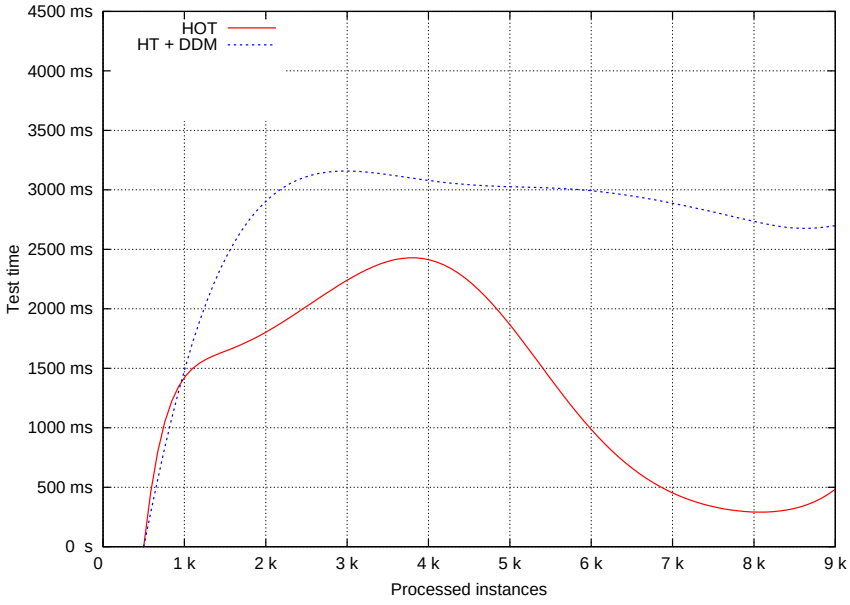

Figure B.11: Chunk test time on the Spam data set.

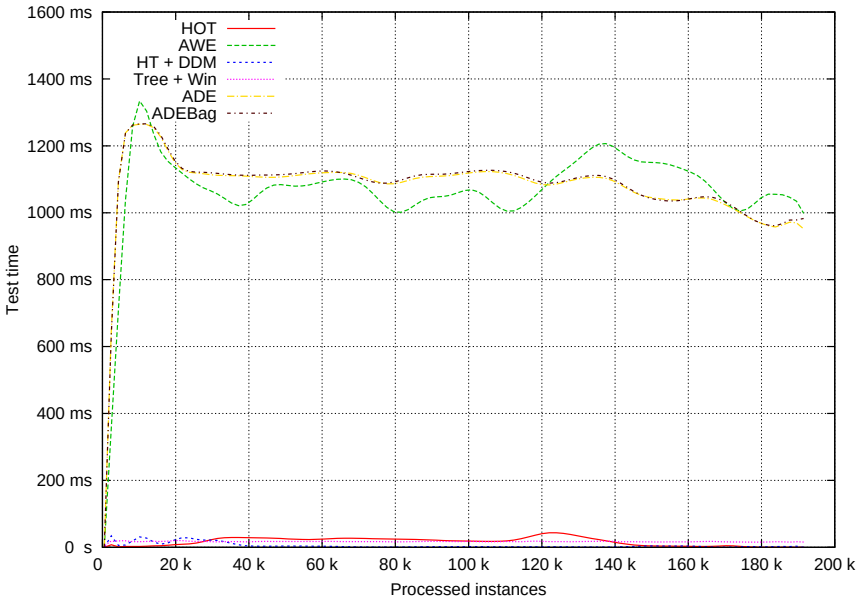

Figure B.12: Chunk test time on the Donation data set.

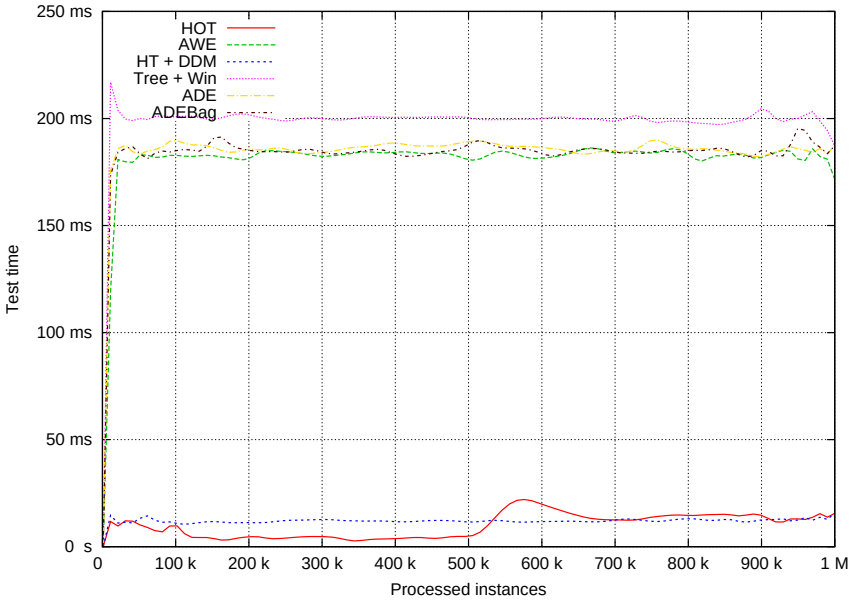

Figure B.13: Chunk test time on the LED data set.

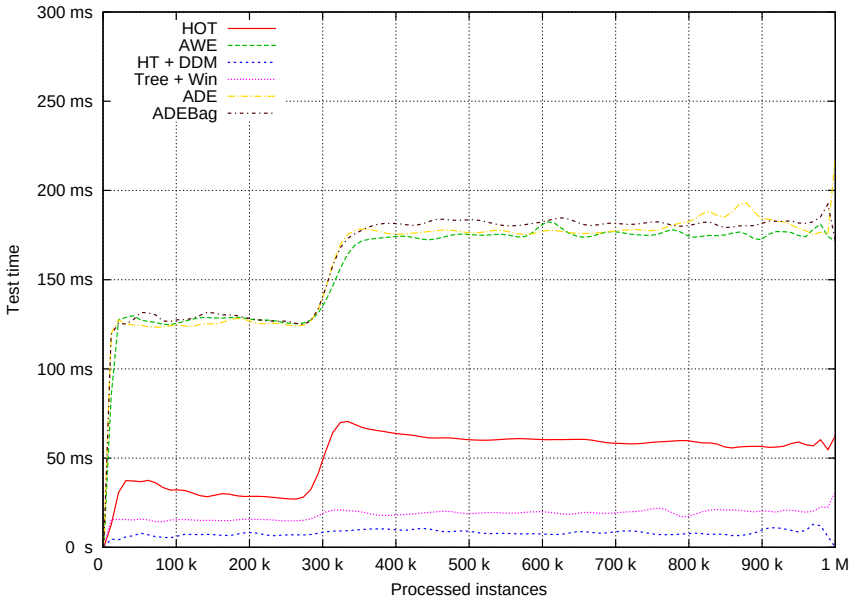

Figure B.14: Chunk test time on the Waveform data set.

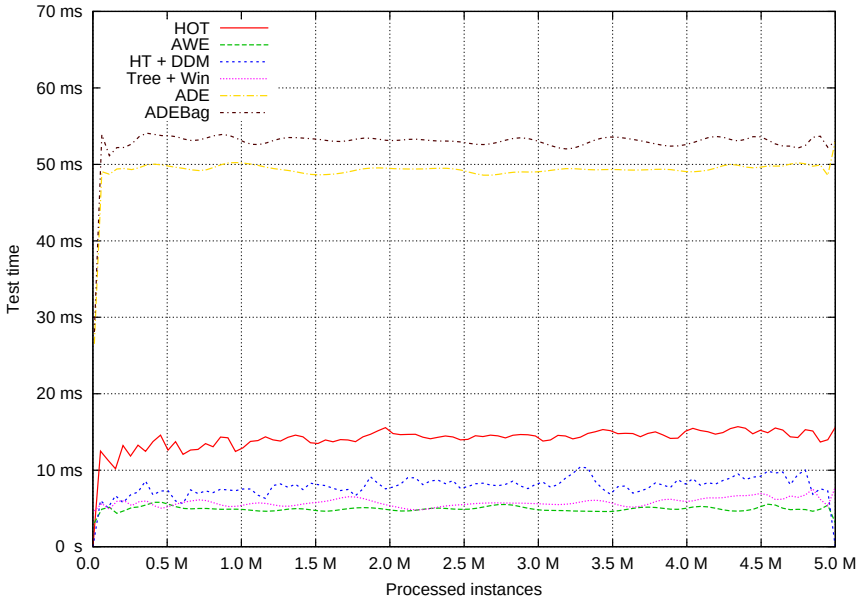

Figure B.15: Chunk test time on the Hyperplane data set.

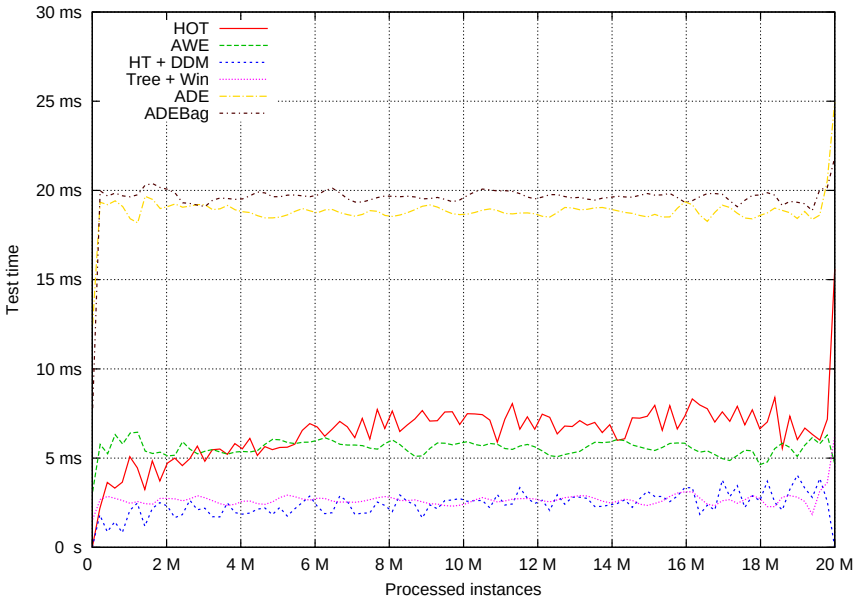

Figure B.16: Chunk test time on the SEA data set.

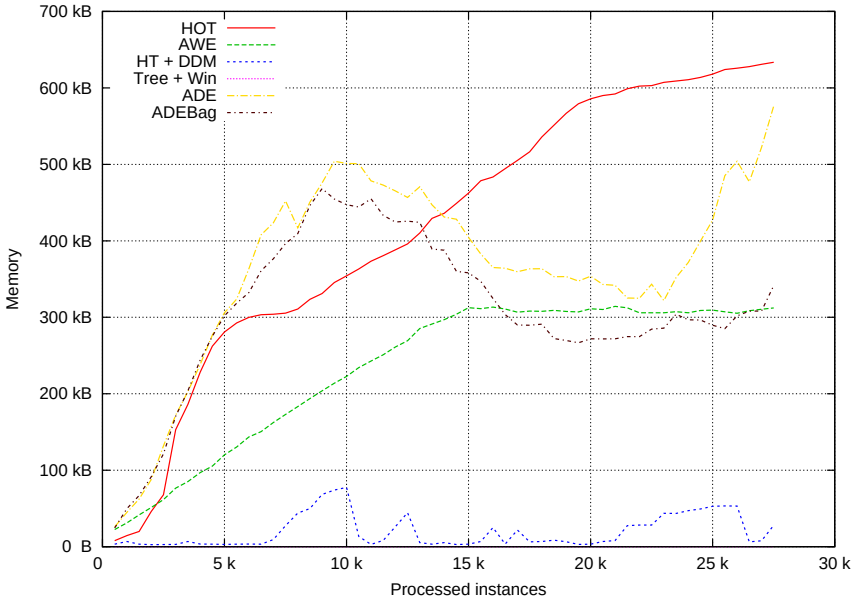

Figure B.17: Memory usage on the Electricity data set.

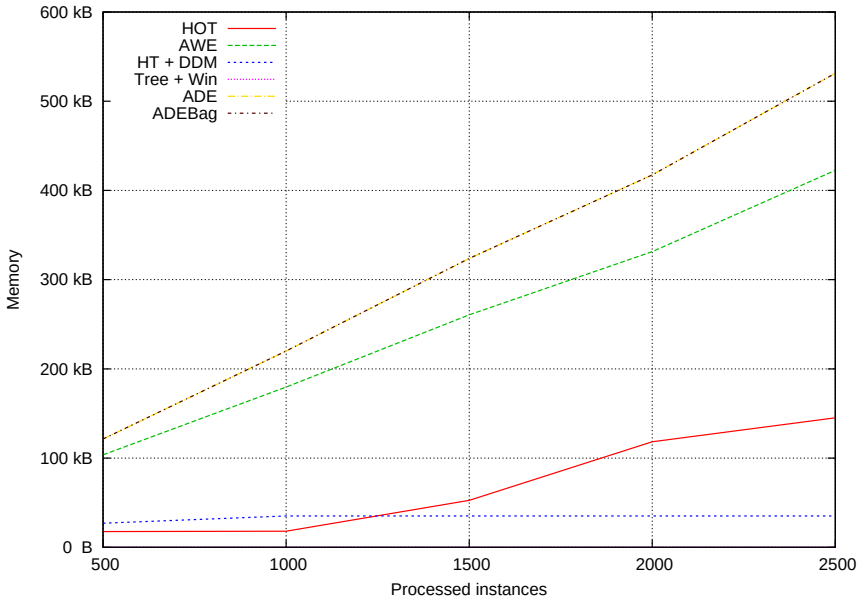

Figure B.18: Memory usage on the Ozone data set.

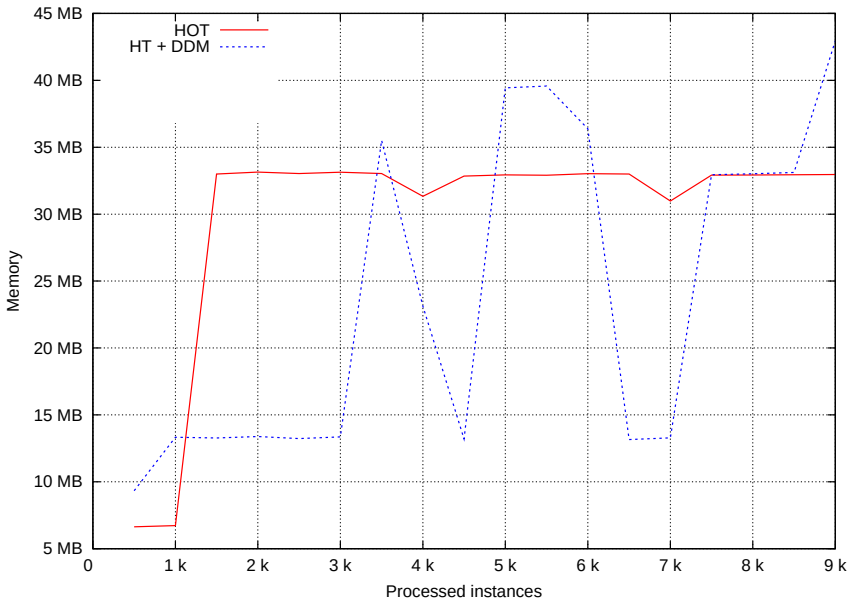

Figure B.19: Memory usage on the Spam data set.

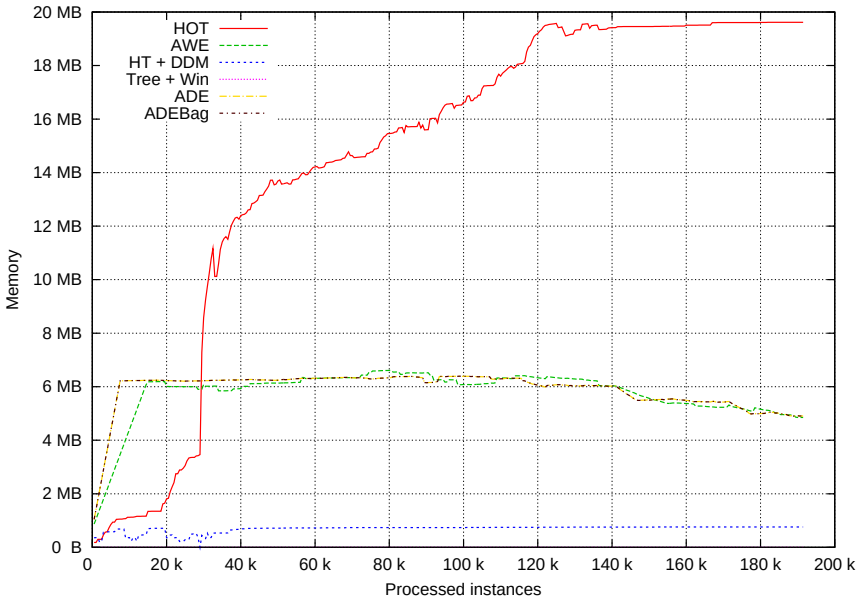

Figure B.20: Memory usage on the Donation data set.

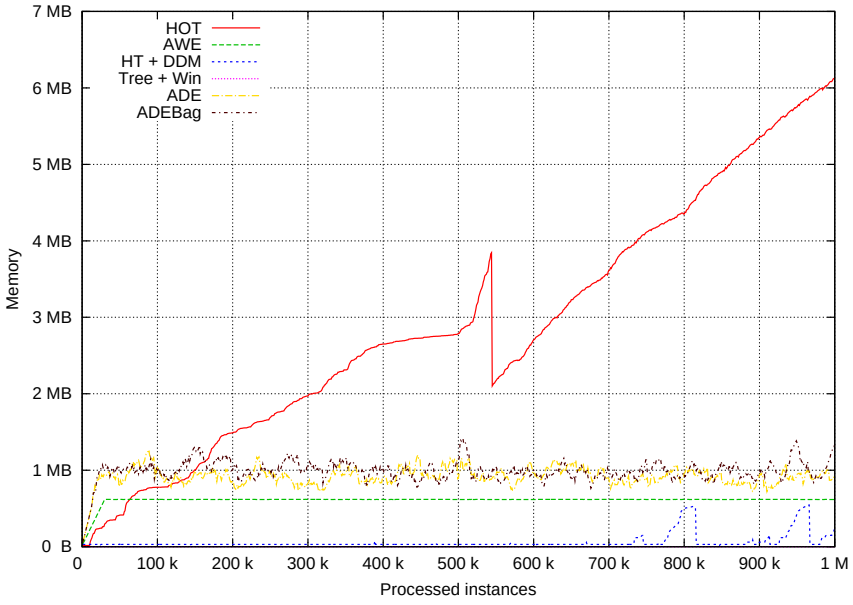

Figure B.21: Memory usage on the LED data set.

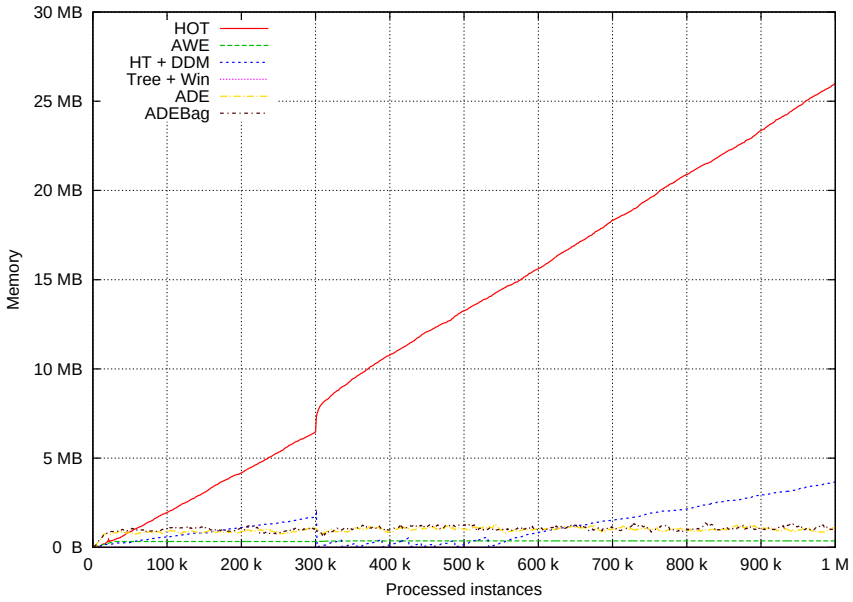

Figure B.22: Memory usage on the Waveform data set.

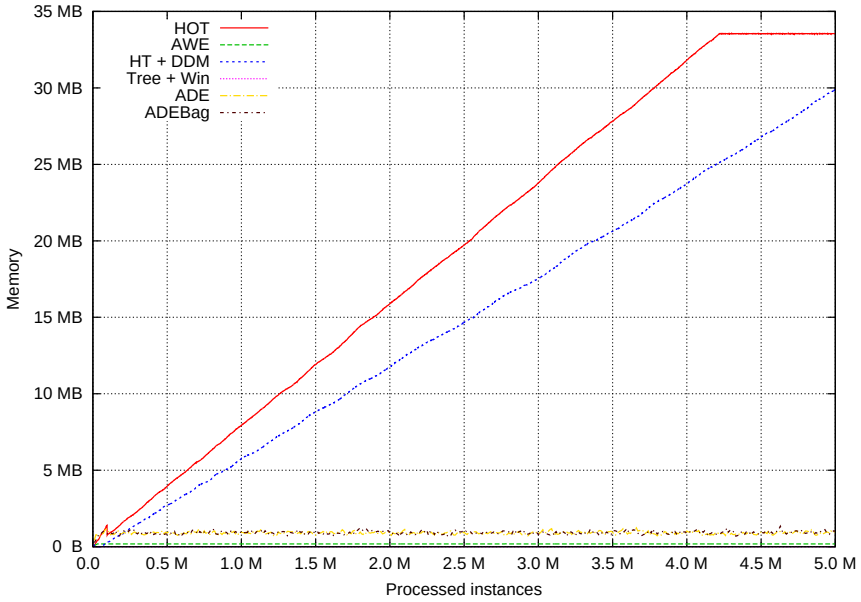

Figure B.23: Memory usage on the Hyperplane data set.

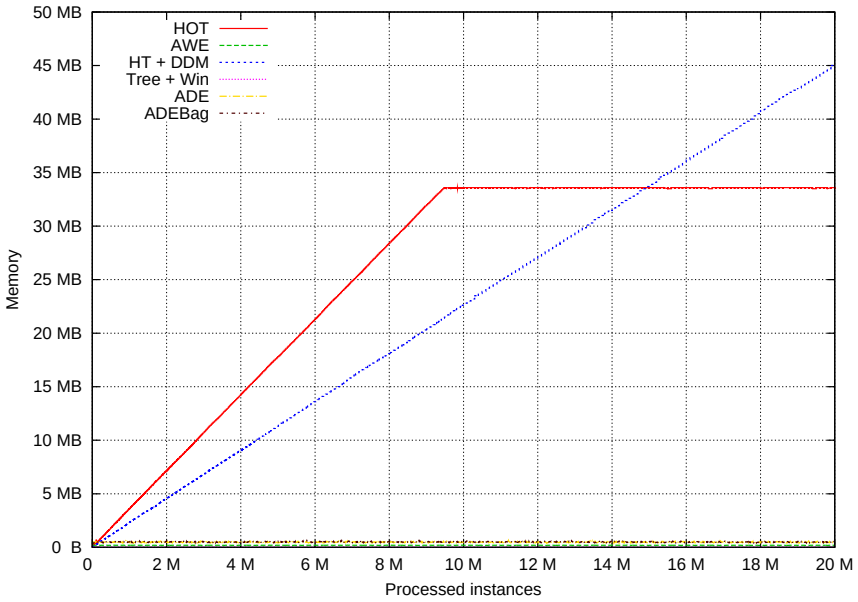

Figure B.24: Memory usage on the SEA data set.

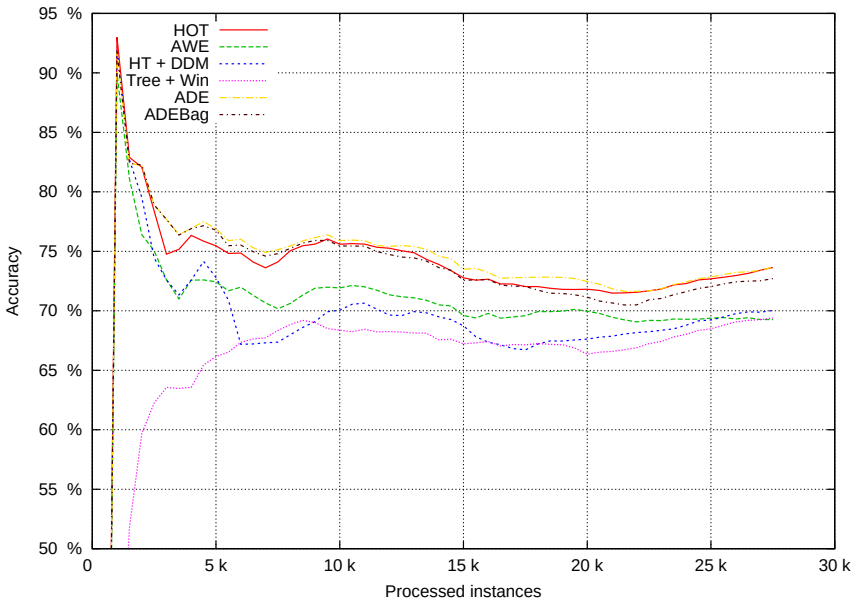

Figure B.25: Accuracy on the Electricity data set.

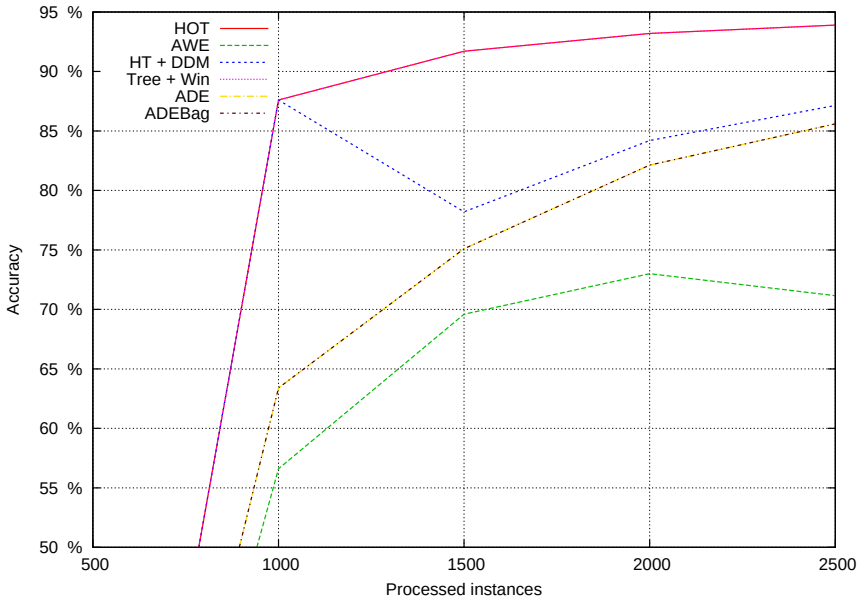

Figure B.26: Accuracy on the Ozone data set.

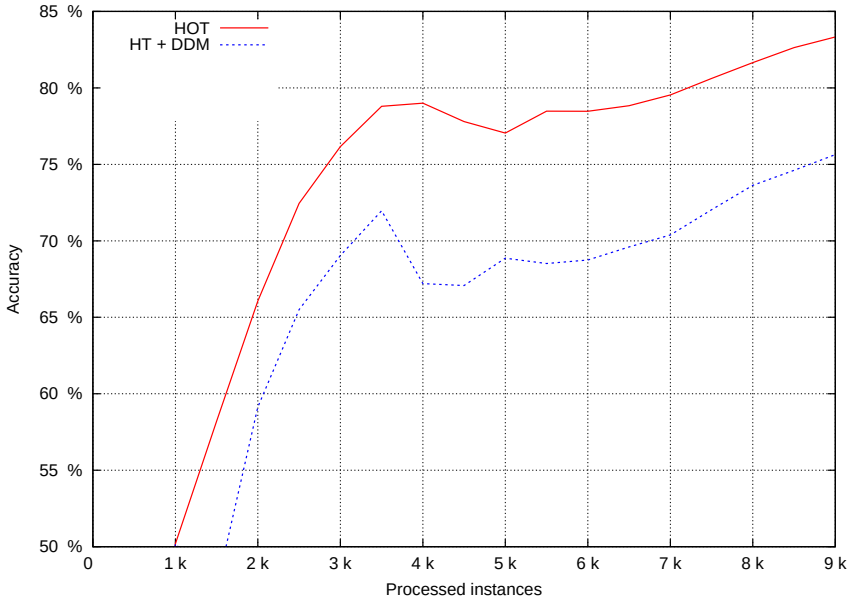

Figure B.27: Accuracy on the Spam data set.

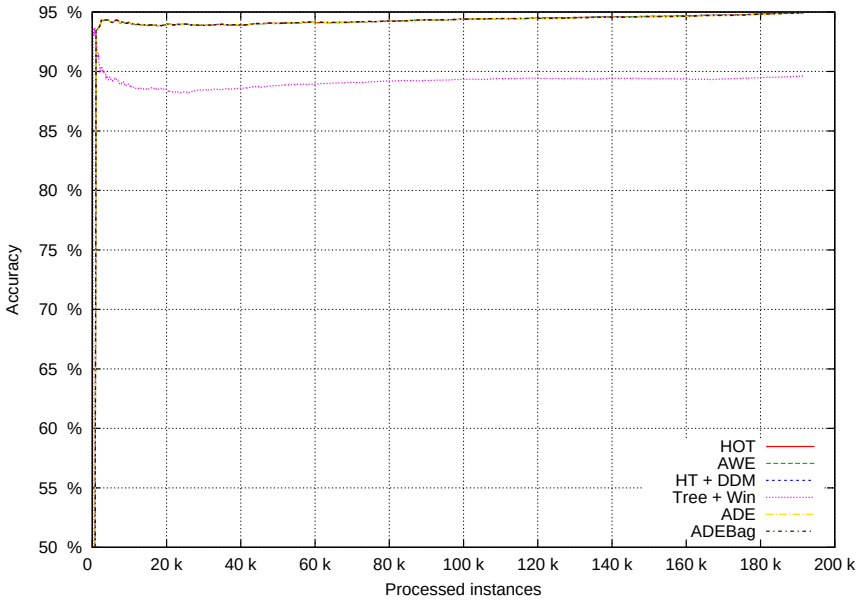

Figure B.28: Accuracy on the Donation data set.

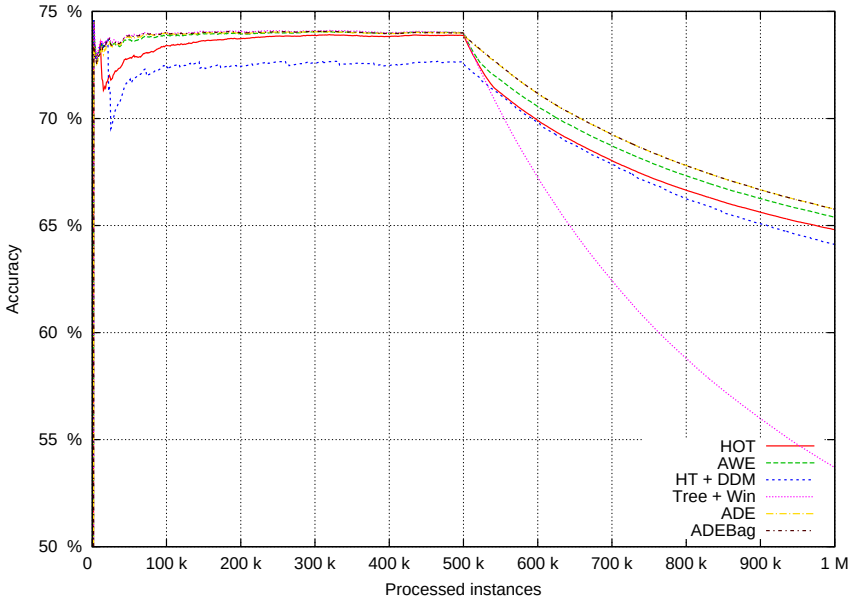

Figure B.29: Accuracy on the LED data set.

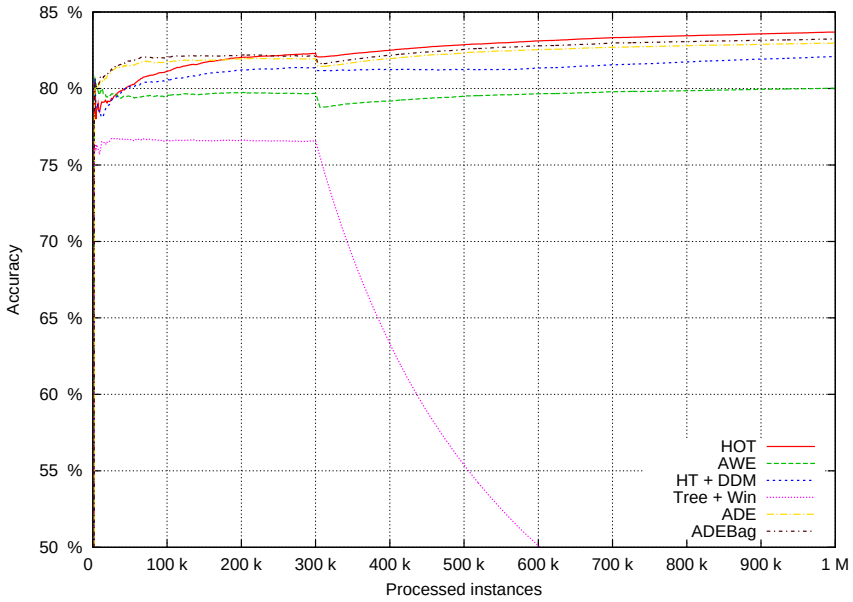

Figure B.30: Accuracy on the Waveform data set.

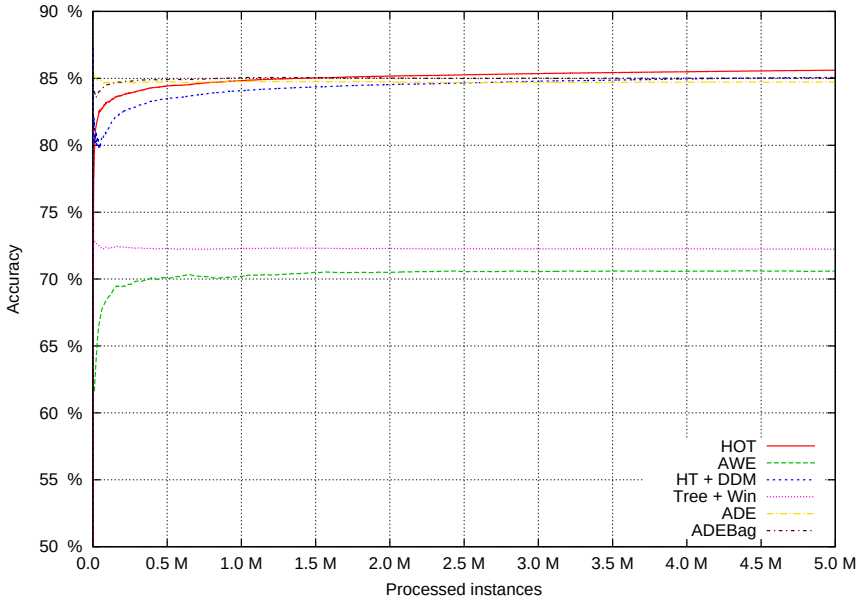

Figure B.31: Accuracy on the Hyperplane data set.

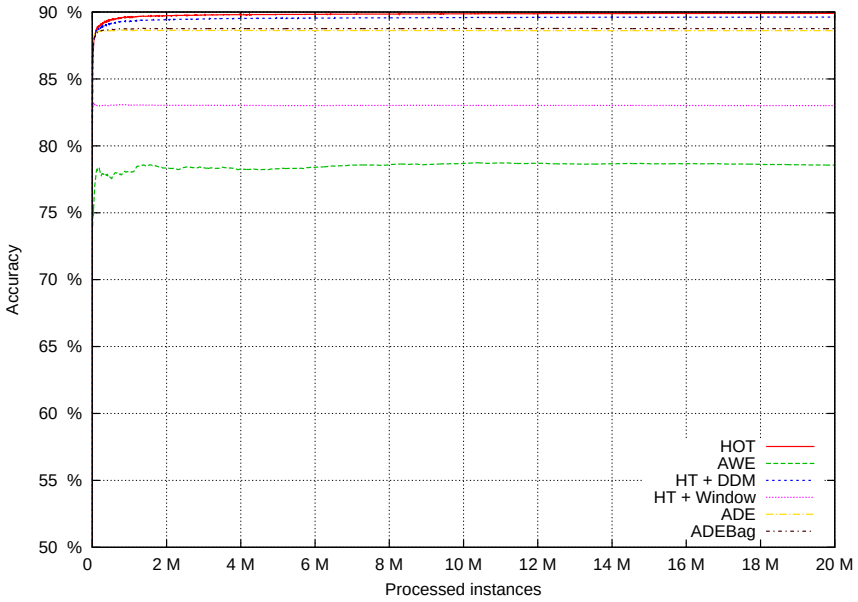

Figure B.32: Accuracy on the SEA data set.

Bibliography

[1] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[2] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In Lucian Popa, editor, *PODS*, pages 1–16. ACM, 2002.

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

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

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

[6] Albert Bifet and Ricard Gavaldà. Kalman filters and adaptive windows for learning in data streams. In Ljupco Todorovski, Nada Lavrac, and Klaus P. Jantke, editors, *Discovery Science*, volume 4265 of *Lecture Notes in Computer Science*, pages 29–40. Springer, 2006.

[7] Albert Bifet and Ricard Gavaldà. Learning from time-changing data with adaptive windowing. In *SDM*. SIAM, 2007.

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

[9] Albert Bifet, Geoffrey Holmes, Bernhard Pfahringer, and Ricard Gavaldà. Improving adaptive bagging methods for evolving data streams. In Zhi-Hua Zhou and Takashi Washio, editors, *ACML*, volume 5828 of *Lecture Notes in Computer Science*, pages 23–37. Springer, 2009.

[10] Albert Bifet, Geoffrey Holmes, Bernhard Pfahringer, Richard Kirkby, and Ricard Gavaldà. New ensemble methods for evolving data streams. In John F. Elder IV, Françoise Fogelman-Soulié, Peter A. Flach, and Mohammed Javeed Zaki, editors, *KDD*, pages 139–148. ACM, 2009.

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

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

[13] Remco R. Bouckaert. Voting massive collections of bayesian network classifiers for data streams. In Abdul Sattar and Byeong Ho Kang, editors, *Australian Conference on Artificial Intelligence*, volume 4304 of *Lecture Notes in Computer Science*, pages 243–252. Springer, 2006.

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

[15] Leo Breiman. Pasting small votes for classification in large databases and on-line. *Machine Learning*, 36(1-2):85–103, 1999.

[16] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[17] 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 *DIGRA Conf.*, 2005.

[18] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. On random sampling over joins. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD Conference*, pages 263–274. ACM Press, 1999.

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

[20] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows (extended abstract). In *SODA*, pages 635–644, 2002.

[21] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *KDD*, pages 71–80, 2000.

[22] Steve Donoho. Early detection of insider trading in option markets. In Won Kim, Ron Kohavi, Johannes Gehrke, and William DuMouchel, editors, *KDD*, pages 420–429. ACM, 2004.

[23] Wei Fan. Systematic data selection to mine concept-drifting data streams. In Won Kim, Ron Kohavi, Johannes Gehrke, and William DuMouchel, editors, *KDD*, pages 128–137. ACM, 2004.

[24] Wei Fan, Yi an Huang, Haixun Wang, and Philip S. Yu. Active mining of data streams. In Michael W. Berry, Umeshwar Dayal, Chandrika Kamath, and David B. Skillicorn, editors, *SDM*. SIAM, 2004.

[25] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. Computing iceberg queries efficiently. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB*, pages 299–310. Morgan Kaufmann, 1998.

[26] Usama M. Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery: An overview. In *Advances in Knowledge Discovery and Data Mining*, pages 1–34. American Association for Artificial Intelligence, 1996.

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

[28] Francisco J. Ferrer-Troyano, Jesús S. Aguilar-Ruiz, and José Cristóbal Riquelme Santos. Discovering decision rules from numerical data streams. In Hisham Haddad, Andrea Omicini, Roger L. Wainwright, and Lorie M. Liebrock, editors, *SAC*, pages 649–653. ACM, 2004.

[29] Mohamed Medhat Gaber and João Gama. State of the art in data streams mining. ECML, 2007.

[30] J. Gama, P. Medas, G. Castillo, and P. Rodrigues. Learning with drift detection. In *SBIA Brazilian Symposium on Artificial Intelligence*, page 286–295, 2004.

[31] João Gama and Mohamed Medhat Gaber, editors. *Learning from Data Streams: Processing techniques in Sensor Networks*. Springer, 2007.

[32] João Gama, Ricardo Rocha, and Pedro Medas. Accurate decision trees for mining high-speed data streams. In Lise Getoor, Ted E. Senator, Pedro Domingos, and Christos Faloutsos, editors, *KDD*, pages 523–528. ACM, 2003.

[33] João Gama and Pedro Pereira Rodrigues. Stream-based electricity load forecast. In Joost N. Kok, Jacek Koronacki, Ramon López de Mántaras, Stan Matwin, Dunja Mladenic, and Andrzej Skowron, editors, *PKDD*, volume 4702 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2007.

[34] Venkatesh Ganti, Johannes Gehrke, and Raghu Ramakrishnan. Mining data streams under block evolution. *SIGKDD Explorations*, 3(2):1–10, 2002.

[35] John F. Gantz, David Reinsel, Christopeher Chute, Wolfgang Schlichting, Stephen Minton, Anna Toncheva, and Alex Manfrediz. The expanding digital universe: An updated forecast of worldwide information growth through 2011. Technical report, IDC Information and Data, 2008.

[36] Anna C. Gilbert, Sudipto Guha, Piotr Indyk, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *STOC*, pages 389–398, 2002.

[37] Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD Conference*, pages 58–66, 2001.

[38] Sudipto Guha and Nick Koudas. Approximating a data stream for querying and estimation: Algorithms and performance evaluation. In *ICDE*, pages 567–. IEEE Computer Society, 2002.

[39] Sudipto Guha, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams. In *FOCS*, pages 359–366, 2000.

[40] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Cure: An efficient clustering algorithm for large databases. *Inf. Syst.*, 26(1):35–58, 2001.

[41] 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.

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

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

[44] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *KDD*, pages 97–106, 2001.

[45] Elena Ikonomovska, Suzana Loskovska, and Dejan Gjorgjevik. A survey of stream data mining, 2005.

[46] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3:79–87, 1991.

[47] H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, and Torsten Suel. Optimal histograms with quality guarantees. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB*, pages 275–286. Morgan Kaufmann, 1998.

[48] Ruoming Jin and Gagan Agrawal. Efficient decision tree construction on streaming data. In Lise Getoor, Ted E. Senator, Pedro Domingos, and Christos Faloutsos, editors, *KDD*, pages 571–576. ACM, 2003.

[49] 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.

[50] Mark G. Kelly, David J. Hand, and Niall M. Adams. The impact of changing populations on classifier performance. In *KDD*, pages 367–371, 1999.

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

[52] Ralf Klinkenberg and Thorsten Joachims. Detecting concept drift with support vector machines. In Pat Langley, editor, *ICML*, pages 487–494. Morgan Kaufmann, 2000.

[53] Ron Kohavi and Clayton Kunz. Option decision trees with majority votes. In Douglas H. Fisher, editor, *ICML*, pages 161–169. Morgan Kaufmann, 1997.

[54] Ludmila I. Kuncheva. Classifier ensembles for changing environments. In Fabio Roli, Josef Kittler, and Terry Windeatt, editors, *Multiple Classifier Systems*, volume 3077 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2004.

[55] Ludmila I. Kuncheva. Classifier ensembles for detecting concept change in streaming data: Overview and perspectives. In *2nd Workshop SUEMA 2008 (ECAI 2008)*, pages 5–10, 2008.

[56] 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.

[57] Andreas D. Lattner, Andrea Miene, Ubbo Visser, and Otthein Herzog. Sequential pattern mining for situation and behavior prediction in simulated robotic soccer. In Ansgar Bredenfeld, Adam Jacoff, Itsuki Noda, and Yasutake Takahashi, editors, *RoboCup*, volume 4020 of *Lecture Notes in Computer Science*, pages 118–129. Springer, 2005.

[58] Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2(4):285–318, 1987.

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

[60] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *VLDB*, pages 346–357. Morgan Kaufmann, 2002.

[61] 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 Thanaruk Theeramunkong, Boonserm Kijsirikul, Nick Cercone, and Tu Bao Ho, editors, *PAKDD*, volume 5476 of *Lecture Notes in Computer Science*, pages 363–375. Springer, 2009.

[62] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Dynamic maintenance of wavelet-based histograms. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB*, pages 101–110. Morgan Kaufmann, 2000.

[63] Oleksiy Mazhelis and Seppo Puuronen. Comparing classifier combining techniques for mobile-masquerader detection. In *ARES*, pages 465–472. IEEE Computer Society, 2007.

[64] 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 Thanaruk Theeramunkong, Boonserm Kijsirikul, Nick Cercone, and Tu Bao Ho, editors, *PAKDD*, volume 5476 of *Lecture Notes in Computer Science*, pages 689–696. Springer, 2009.

[65] Nikunj C. Oza. Online ensemble learning. In *AAAI/IAAI*, page 1109. AAAI Press / The MIT Press, 2000.

[66] Nikunj C. Oza. *Online Ensemble Learning*. PhD thesis, The University of California, Berkeley, CA, Sep 2001.

[67] E. S. Page. Continuous inspection schemes. *Biometrika*, 41(1/2):100–115, 1954.

[68] 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.

[69] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[70] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[71] S. W. Roberts. Control chart tests based on geometric moving averages. *Technometrics*, 42(1):97–101, 2000.

[72] Jeffrey C. Schlimmer and Richard H. Granger. Incremental learning from noisy data. *Machine Learning*, 1(3):317–354, 1986.

[73] W. Nick Street and YongSeog Kim. A streaming ensemble algorithm (sea) for large-scale classification. In *KDD*, pages 377–382, 2001.

[74] 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. *Journal of Field Robotics*, 23:661–692, 2006.

[75] Alexey Tsymbal, Mykola Pechenizkiy, Padraig Cunningham, and Seppo Puuronen. Dynamic integration of classifiers for handling concept drift. *Information Fusion*, 9(1):56–68, 2008.

[76] 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.

[77] Jeffrey Scott Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[78] Haixun Wang, Wei Fan, Philip S. Yu, and Jiawei Han. Mining concept-drifting data streams using ensemble classifiers. In Lise Getoor, Ted E. Senator, Pedro Domingos, and Christos Faloutsos, editors, *KDD*, pages 226–235. ACM, 2003.

[79] Weka Machine Learning Project. Weka. URL http://www.cs.waikato.ac.nz/~ml/weka.

[80] Gerhard Widmer and M. Kubat. Learning in the presence of concept drift and hidden contexts. In *Machine Learning*, pages 69–101, 1996.

[81] Kun Zhang, Wei Fan, Xiaojing Yuan, Ian Davidson, and Xiangshang Li. Forecasting skewed biased stochastic ozone days: Analyses and solutions. In *ICDM*, pages 753–764. IEEE, IEEE Computer Society, 2006.

[82] Indrė Žliobaitė. Combining time and space similarity for small size learning under concept drift. In Jan Rauch, Zbigniew W. Ras, Petr Berka, and Tapio Elomaa, editors, *ISMIS*, volume 5722 of *Lecture Notes in Computer Science*, pages 412–421. Springer, 2009.

[83] Indrė Žliobaitė. Instance selection method (fish) for classifier training under concept drift. Technical report, Vilnius University, Faculty of Mathematics and Informatic, 2009.

[84] Indrė Žliobaitė. Learning under concept drift: an overview. Technical report, Vilnius University, Faculty of Mathematics and Informatic, 2009.

[85] Indrė Žliobaitė. *Adaptive training set formation*. PhD thesis, Vilnius University, 2010.

Streszczenie

W dobie społeczeństwa informacyjnego, użytkownicy komputerów przyzwyczajeni są do gromadzenia i współdzielenia danych niemal w dowolnym miejscu i czasie. Portale społecznościowe, bankowość elektroniczna, usługi telekomunikacyjne, czy współdzielenie filmów oraz muzyki to tylko niektóre ze zjawisk powodujących gwałtowny wzrost liczby przechowywanych i przetwarzanych danych. Raport sprzed dwóch lat [35] szacował, że rozmiar świata danych elektronicznych wyniósł w 2007 roku 281 miliardów gigabajtów i że rozmiar ten wzrośnie pięciokrotnie do roku 2011. Ten sam raport zakłada, że do końca 2010 roku połowa z wytwarzanych danych nie będzie trwale zapisywana na żadnych nośnikach. Powodem tego są po części nowe rodzaje zastosowań informatyki, w których przetwarzane informacje przyjmują postać *strumieni danych*.

Strumień danych może być postrzegany jako sekwencja elementów (np. billingów rozmów telefonicznych, odwiedzin strony internetowej, odczytów z czujników), które napływają w sposób ciągły w zmiennych interwałach czasu. Strumienie, jak inne duże wolumeny danych, mogą być przedmiotem *eksploracji danych* i *odkrywania wiedzy*, czyli procesu poszukiwania nowych, nietrywialnych i potencjalnie użytecznych wzorców z danych [26, 14]. Eksploracja strumieni danych przedstawia nowe wyzwania w stosunku do tradycyjnie pojętej eksploracji danych. W rozważanym problemie rozmiar przetwarzanych danych jest uznawany za zbyt duży do trwałego przechowywania, a tempo napływania nowych elementów nie jest znane. Stąd potrzeba rozwijania nowych, dedykowanych podejść do eksploracji strumieni.

Niniejsza praca podejmuje tematykę eksploracji strumieni danych ze zmienną definicją klas. Jako jeden z podstawowych celów przyjęto dokonanie przeglądu i eksperymentalnego porównania istniejących metod klasyfikacji strumieni danych. Zaproponowano również nowy algorytm oparty na krytyce istniejącego rozwiązania i skonfrontowano jego trafność, wymagania czasowe oraz pamięciowe z innymi metodami. Jako dodatkowe cele przyjęto implementacyjne rozszerzenie i wykorzystanie do eksperymentów środowiska MOA, które jest nowym projektem i nie zostało jeszcze szerzej opisane przez osoby inne niż przez autorów. Postanowiono zbadać przydatność MOA do tworzenia i porównywania nowych rozwiązań z dziedziny eksploracji strumieni danych.

Ze względu na charakterystykę strumieni danych, strumieniowe algorytmy eksploracji muszą przetwarzać pojawiające się elementy przyrostowo, przetwarzając każdy przykład tylko raz, przy ograniczonym czasie, ograniczonej pamięci, dostosowując się równocześnie do zmian w źródle strumienia. Źródło strumienia jest tutaj rozumiane jako definicja klas decyzyjnych pozwalająca generować nowe przykłady. Typowe zmiany w źródle strumienia to zaniknięcie klasy, pojawienie się nowej, stopniowa zmiana definicji, gwałtowna zmiana definicji, oraz czasowo zanikająca klasa. O ile ograniczenia czasowe i pamięciowe bywały już wcześniej implementowane w propozycjach algorytmów eksploracji danych, o tyle zmiany definicji klas w czasie są cechą charakterystyczną strumieni i nie były rozważane w tradycyjnych metodach eksploracji.

Przegląd najpopularniejszych metod eksploracji strumieni danych można podzielić na dwie grupy: pojedyncze klasyfikatory oraz klasyfikatory złożone. Grupę pojedynczych klasyfikatorów

otwierają algorytmy znane z tradycyjnej eksploracji danych jak metoda najbliższych sąsiadów, sztuczne sieci neuronowe [33], metody bayesowskie [13], czy reguły decyzyjne [28, 31, 80]. Wszystkie wymienione metody, po drobnych modyfikacjach, mogą być wykorzystane do klasyfikowania strumieni. Tradycyjne metody eksploracji są rozwijane od wielu lat i istnieje wiele gotowych implementacji, a nawet całych środowisk do ich testowania. Wadą tych podejść jest to, że dają one z reguły gorsze rezultaty niż metody wyspecjalizowane w przetwarzaniu strumieni [5].

Do specjalistycznych metod klasyfikacji strumieni należy między innymi technika okien przesuwnych pozwalająca ograniczyć pamięć algorytmów do najnowszych przykładów. Dwa najciekawsze algorytmy z tej grupy to FISH [83, 82, 85] i ADWIN [6, 7]. Innym sposobem na „zapominanie"przestarzałych przykładów w strumieniach danych jest przebudowa klasyfikatora po wykryciu zmiany. Do tego celu wykorzystywane są detektory zmian jak DDM [30] czy EDDM [3], które obserwując rozkład błędów popełnianych przez klasyfikator starają się określić moment wystąpienia zmiany w źródle strumienia.

Najbardziej znanym pojedynczym klasyfikatorem zaprojektowanym do przetwarzania strumieni danych jest algorytm VFDT (Very Fast Decision Tree) [21]. Klasyfikator ten jest modyfikacją drzewa decyzyjnego, która pozwala przyrostowo budować klasyfikator gwarantując równocześnie trafność klasyfikacji bliską drzewu budowanemu wsadowo, na całym zbiorze danych. Gwarancja trafności jest zapewniana przez wykorzystanie granicy Hoefffdinga jako mechanizmu określającego liczbę przykładów potrzebnych do stworzenia najlepszego, z zadanym przez użytkownika prawdopodobieństwem, rozgałęzienia. Z racji wykorzystania owego mechanizmu, algorytm VFDT jak i wszystkie jego modyfikacje są nazywane drzewami Hoeffdinga.

Klasyfikatory złożone to zbiory pojedynczych klasyfikatorów, zwanych klasyfikatorami bazowymi, które wspólnie (z reguły przez głosowanie) przewidują klasę decyzyjną. Modularna budowa klasyfikatorów złożonych sprawia, że mają one wiele cech przydatnych w przetwarzaniu strumieni danych. Ważenie głosów klasyfikatorów bazowych pozwala dynamicznie reagować na zmiany. Budowanie nowych klasyfikatorów bazowych z nadchodzących przykładów pozwala stopniowo polepszać działanie klasyfikatora bez przebudowywania wcześniej nauczonych fragmentów. Główną wadą klasyfikatorów złożonych jest ich czas działania. Przetwarzają one dane z reguły o wiele dłużej niż pojedyncze klasyfikatory, lecz oferują często znacznie lepszą trafność klasyfikacji.

Do jednych z pierwszych zaproponowanych strumieniowych klasyfikatorów złożonych należą algorytmy Streaming Ensemble Algorithm (SEA) [73] i Accuracy Weighted Ensemble (AWE) [78]. Obie metody przetwarzają strumień paczkami (*ang. data chunks*), z których budują nowe klasyfikatory bazowe. Następnie, jeśli to korzystne, nowo-zbudowany klasyfikator bazowy zastępuje inny, „słabszy"klasyfikator bazowy. O „sile"klasyfikatora decyduje nadana mu waga, która jest inaczej obliczana dla obu algorytmów. SEA zlicza błędy popełniane przez każdy klasyfikator bazowy, a następnie sprawdza zgodność głosów między nimi by określić wagi. Sprawdzając zgodność predykcji elementów składowych, SEA premiuje klasyfikatory bazowe, które specjalizują się w innych przykładach niż pozostali głosujący. Algorytm AWE jako wagę przypisuje oszacowany na najnowszej paczce przykładów błąd średniokwadratowy danego klasyfikatora bazowego. Jeśli element składowy A ma najmniejszy błąd średniokwadratowy na ostatniej paczce, to AWE zakłada, że nadchodzące przykłady będą podobne do tych z ostatniej paczki i nadaje najwyższą wagę elementowi składowemu A.

Zupełnie innym typem klasyfikatora złożonego jest niedawno zaproponowany algorytm Hoeffding Option Tree (HOT) [51]. Algorytm jest zainspirowany wcześniejszą pracą Kunza oraz Kohaviego [53] i modyfikuje ich pomysły, by dopasować klasyfikator do świata strumieni danych. HOT można sobie wyobrazić jako skompresowaną postać kliku drzew Hoeffdinga połączonych jednym korzeniem. Jest to uproszczony opis, ale przedstawia główne zalety tego podejścia: gwarancję

trafności zapewnianą prze granicę Hoeffdinga, złożenie decyzji kilku klasyfikatorów bazowych i małą zajętość pamięciową. Innym klasyfikatorem wykorzystującym drzewa Hoeffdinga jest ASHT Bagging [10, 9]. W tym algorytmie drzewa Hoeffdinga są składowymi klasyfikatora złożonego, a wagi są im nadawane zgodnie z liczbą popełnianych błędów. Cechą szczególną tej metody jest wprowadzenie ograniczeń na kolejne drzewa składowe - każde drzewo ma określony maksymalny rozmiar, dwa razy większy od poprzedniego drzewa. Po przekroczeniu swojego rozmiaru drzewo jest budowane od początku z nowych przykładów. Różne rozmiary drzew działają jak różne rodzaje pamięci. Drzewa większe pamiętają lepiej starsze przykłady, a drzewa małe specjalizują się w tych najnowszych.

Analiza wymienionych algorytmów doprowadziła do zaproponowania w tej pracy nowego klasyfikatora złożonego o nazwie Accuracy Diversified Ensemble (ADE). ADE opiera się głównie na krytyce algorytmu AWE - wprowadza do niego nowe elementy zachowując ideę ważenia elementów składowych według błędu średniokwadratowego. Klasyfikatorami bazowymi w ADE są drzewa Hoeffdinga, a nie jak w AWE zwykłe drzewa decyzyjne lub inne tradycyjne klasyfikatory. Zmiana ta pozwala na douczanie wcześniej stworzonych składowych, a tym samym na zmniejszenie rozmiaru paczek przykładów bez niebezpieczeństwa budowania mniej trafnych drzew. Klasyfikatory bazowe są tak jak w oryginalnym algorytmie ważone według błędu średniokwadratowego, lecz sama funkcja obliczająca wagę została zmodyfikowana. Zniesiono próg błędu pozwalający przypisać wagę, by uniknąć zaobserwowanego w eksperymentach dla AWE zjawiska zerowania wszystkich wag przy nagłych zmianach w strumieniu. By dodatkowo premiować składowe trafnie klasyfikujące przykłady z ostatniej paczki, zaproponowano douczać klasyfikatory bazowe zbudowane na poprzednich paczkach, tylko jeśli ich obecna trafność przekracza próg wyznaczany na podstawie rozkładu przykładów w ostatniej paczce.

Proponowany algorytm może za pomocą jednej paczki douczyć więcej niż jeden klasyfikator bazowy. By uniknąć teoretycznego niebezpieczeństwa upodobniania się drzew składowych, postanowiono wykorzystać zaproponowany przez Ozę algorytm Online Bagging [66] do dodatkowej dywersyfikacji klasyfikatorów bazowych. By się upewnić, że takie dynamiczne urozmaicanie składowych jest konieczne, w eksperymentach porównano dwie wersje ADE - z i bez baggingu.

Jednym z celów tej pracy było implementacyjne rozszerzenie, wykorzystanie do testów i ocena środowiska MOA. MOA (*ang. Massive Online Analysis*) to środowisko do implementowania i przeprowadzania eksperymentów na algorytmach klasyfikujących strumienie danych [8, 12, 11]. Zaimplementowane jest w języku Java i zawiera zbiór generatorów strumieni, algorytmów eksploracji danych i metod oceny algorytmów. Praca w MOA podzielona jest na zadania (*ang. task*). Do podstawowych zadań oferowanych przez środowisko należą: uczenie klasyfikatora, ocena klasyfikatora, generowanie strumienia do pliku, pomiar prędkości strumienia. Wszystkie zadania mogą być wykonywane z poziomu interfejsu graficznego lub konsoli. Ciekawą opcją jest możliwość wykonywania kilku zadań równolegle z poziomu interfejsu graficznego.

Strumienie danych w MOA mogą być generowane, odczytywane z plików ARFF, łączone i filtrowane. Ważną opcją jest również możliwość wprowadzania dynamicznych zmian definicji klas w czasie. MOA zawiera najpopularniejsze generatory strumieni, w tym: Random Trees [21], SEA [73], STAGGER [72], Rotating Hyperplane [78, 23, 24], Random RBF, LED [32] i Waveform [32]. W ramach rozszerzania środowiska zaimplementowano w tej pracy metodę usuwania wybranych atrybutów ze strumienia. Zauważono, że etap filtrowania, oryginalnie zaimplementowany w MOA, znacznie spowalnia przetwarzanie strumieni i wymaga optymalizacji.

W MOA zaimplementowano szereg popularnych klasyfikatorów strumieniowych, w tym: Naiwny Bayes, drzewa Hoeffdinga, HOT, Bagging Ozy, Boosting, okno przesuwne ADWIN, ASHT Bagging i Weighted Majority Algorithm. Środowisko pozwala również na wykorzystanie klasyfika-

tora z platformy WEKA w połączeniu z detektorem zmian (DDM lub EDDM). W ramach tej pracy zaimplementowano i dodano do MOA algorytmy AWE i ADE. Podobnie jak w środowisku WEKA, napisanie własnego klasyfikatora w MOA wymaga jedynie zaimplementowania klasy dziedziczącej z klasy `AbstractClassifier`, a interfejs graficzny dla takiej klasy jest tworzony dynamicznie.

Ostatnim rozszerzeniem jakie w tej pracy wprowadzono do MOA jest nowa metoda oceny klasyfikatorów. Oryginalnie w MOA istnieją dwie metody szacowania skuteczności klasyfikatorów: *Holdout* i *Interleaved Test-Then-Train*. Pierwsza z nich wykorzystuje n początkowych przykładów jako testowe i co pewien interwał czasowy sprawdza trafność klasyfikatora uczonego na pozostałych elementach strumienia. Druga metoda wykorzystuje każdy nowy przykład najpierw do testowania, a następnie do douczania klasyfikatora. Wadą metody Holdout jest statyczność zbioru testującego, która może skutkować złą oceną dla zmieniających się strumieni. Z kolei metoda Interleaved Test-Then-Train wykonuje testy bardzo często zaniżając trafność klasyfikacji i uniemożliwiając osobny pomiar czasu dla uczenia i testowania. Postanowiono zaimplementować metodę oceny będącą kompromisem pomiędzy tymi wcześniej wymienionymi - *metodę oceny paczkami*. Metoda oceny paczkami (*ang. Data Chunk Evaluation*) grupuje przykłady w paczki, które najpierw wykorzystywane są do testowania a później do uczenia. Tworzenie paczek pozwala obliczać czas uczenia i testowania grup przykładów oraz niweluje problem zaniżania trafności klasyfikacji występujący w metodzie Interleaved Test-Then-Train. Metoda ta może również być wykorzystywana do oceny działania klasyfikatorów na strumieniach zmiennych w czasie, gdyż zbiór nie jest statyczny tak jak w metodzie Holdout. Wszystkie testy wykonane w ramach tej pracy wykorzystywały do porównywania klasyfikatorów metodę oceny paczkami.

Do eksperymentalnego porównania algorytmów wybrano dwa klasyfikatory pojedyncze (drzewo Hoeffdinga z detektorem DDM i drzewo decyzyjne z oknem przesuwnym), dwa klasyfikatory złożone (AWE i HOT) i dwie wersje zaproponowanego w tej pracy algorytmu ADE (z i bez baggingu). Działanie każdego z algorytmów przetestowano na czterech rzeczywistych i czterech sztucznie wygenerowanych zbiorach danych. Ogólnodostępne rzeczywiste zbiory danych są małe w porównaniu ze strumieniami narzucającymi wymagania pamięciowe i czasowe. Dlatego właśnie wykorzystano cztery generatory do stworzenia sztucznych strumieni mających po 1, 5 i 20 milionów przykładów. Dla wszystkich zbiorów porównywano czas uczenia, testowania, zajętość pamięciową i trafność klasyfikacji. Wszystkie wyniki zaprezentowano w postaci przebiegów czasowych w dodatku B.

Wyniki pokazują, że pojedyncze klasyfikatory przetwarzają strumienie szybciej niż złożone klasyfikatory. Jest to spodziewany wynik, gdyż prostsze klasyfikatory z reguły działają szybciej niż bardziej złożone. Dla pięciu najmniejszych zbiorów zużycie pamięci było najmniejsze dla drzewa Hoeffdinga, a dla większych strumieni najlepiej spisywały się algorytmy AWE i ADE. Klasyfikatory HOT i drzewo z detektorem zmian wykazywały wprost proporcjonalny do liczby przetworzonych przykładów wzrost wymagań pamięciowych. Przykład różnych wyników dla różnych rozmiarów strumieni pokazuje jak ważny jest zakres eksperymentów na algorytmach strumieniowych i jak potrzebne są sztuczne generatory danych. Różnice w trafności klasyfikacji były stosunkowo niewielkie pomiędzy algorytmami HOT, ADE i drzewem Hoeffdinga. Klasyfikatory AWE i drzewo z oknem przesuwnym wypadły o wiele gorzej. Widać, że zaproponowany algorytm ADE przy niewielkim wzroście czasu przetwarzania i zajętości pamięciowej znacznie poprawił swoją trafność klasyfikacji w stosunku do pierwowzoru. Ponadto, ADE ma stałe wymagania pamięciowe i czasowe, w przeciwieństwie do HOT i drzewa Hoeffdinga, których wymagania potrafiły rosnąć z czasem.

Jak wspomniano wcześniej, podczas testów porównaliśmy dwie wersje algorytmu ADE. Okazało się, że dodanie baggingu do procesu douczania klasyfikatorów bazowych nie podniosło znacząco trafności klasyfikacji. Ponadto, bagging wniósł dodatkowe, choć również niewielkie, narzuty czasowe i pamięciowe. Wydaje się, że ADE już bez baggingu radzi sobie dobrze z dywersyfikacją

klasyfikatorów bazowych i jeśli można osiągnąć jakieś korzyści z różnicowania drzew składowych w algorytmie to wydaje się, że nie przez bagging.

Wyniki eksperymentów przeprowadzonych w ramach tej pracy można częściowo porównać z publikacjami innych autorów. Porównanie wyników czasowych, pamięciowych i trafności klasyfikacji dla algorytmów HOT, drzewa Hoeffdinga i metody przesuwnych okien wypada podobnie jak w artykule napisanym przez Bifeta et al. [10]. Główna różnica polega na tym, że w testach w tej pracy HOT uzyskiwał zwykle lepszą trafność klasyfikacji niż pojedyncze drzewo Hoeffdniga z detektorem zmian podczas, gdy u Bifeta było odwrotnie. Porównanie AWE i ADE z pozostałymi algorytmami w tej pracy nie było nigdy wcześniej przeprowadzane i przedstawia nowe, wcześniej niepublikowane rezultaty.

Przegląd algorytmów dokonany w tej pracy pokazuje, że eksploracja strumieni danych ze zmienną definicją klas kształtuje się jako nowa gałąź odkrywania wiedzy ze swoimi własnymi problemami badawczymi. Ograniczenia czasowe i pamięciowe nałożone na algorytmy sprawiają, że trafność klasyfikacji nie może być traktowana jako najważniejsze kryterium oceny klasyfikatora. Ponadto, zmienność problemu decyzyjnego w czasie wymaga projektowania mechanizmów zapominania zwykle niewystępujących w tradycyjnych metodach eksploracji danych. Środowisko MOA jest propozycją zunifikowania sposobu implementacji i testowania algorytmów, które mierzą się z tymi problemami. To kolejny znak na to, że eksploracja strumieni danych staje się coraz dojrzalszą dziedziną informatyki dążącą do sprostania nadchodzącym wyzwaniom świata rzeczywistego.

W ramach dalszych badań, planowane jest przeprowadzenie dodatkowych eksperymentów sprawdzających czy rozmiar paczki przykładów wpływa na wyniki osiągane przez ADE z baggingiem. Ponadto, planowane jest zbadanie wpływu na trafność klasyfikacji ADE innych niż bagging metod dywersyfikacji składowych jak na przykład używanie różnych typów klasyfikatorów bazowych czy boosting. Dodatkowe eksperymenty będą okazją do poszerzenia testu o inne algorytmy eksploracji i dokonania tym samym pełniejszego przeglądu dostępnych metod.

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

Typeset using LaTeX in Computer Modern.

BibTeX:

```
@mastersthesis{BrzezMs2010,
    author  = {Dariusz Brzezi{\'n}ski},
    title   = {Mining data streams with concept drift},
    school  = {Poznan University of Technology},
    address = {Pozna{\'n}, Poland},
    year    = {2010}
}
```