Technische Universität Berlin
Information Technologies for Business Intelligence

Master Thesis

Concept Drift Adaptation in Large-scale Distributed Data Stream Processing

Anastasiia Basha

Matriculation #: 0376691

Supervisor: Prof. Dr. Volker Markl
Advisors: Marcela Charfuelan, Nikolaas Steenbergen

31/07/2016

Erklärung (Declaration of Academic Honesty)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

I hereby declare to have written this thesis on my own and without forbidden help of others, using only the listed resources.

Datum	Anastasiia Basha

Contents

List of Figures

List of Listings

List of Algorithms

List of Acronyms

1. English Abstract

Many modern applications need to process vast amounts of data over short or long periods of time. Traditional machine learning techniques, where model is created and trained once and later used for new samples, are not applicable in this setting anymore. Consequently, the new concept arised – *Online Learning*. The model is continuously updated with training samples as they arrive. An important aspect of online learning is that it takes the time dimension of data into account. And the data characteristics might change over time. This phenomena is called *Concept Drift* [29]. The ability to adapt to such a change is a very important requirement for an online learning algorithm.

The need for processing continuous, potentially infinite sequences of data sparked the development of multiple distributed stream processing frameworks. This thesis explores the opportunity to use one of such systems, Apache Flink, for implementation of scalable adaptive online learning algorithms. In particular, this thesis concentrates on the scalable implementation of the bagging ensemble algorithm. As the algorithm has no implicit ability to handle concept drift we explore different options of integrating the concept drift detection into it. We also analyse different techniques to make the algorithm resistant to false drift detections.

This thesis also explores different evaluation techniques for the online learning algorithms, as the traditional methods used in offline learning, like cross validation, are not applicable in the case of infinite data streams.

2. Deutscher Abstract

Viele moderne Softwareapplikationen müssen in der Lage sein immense Datenmengen über kurze oder längere Zeiträume zu verarbeiten. Traditionelle Techniken im Gebiet des maschinellem Lernen, bei denen ein Modell erstellt, einmal trainiert und anschließend für die Interpretation neuer Daten gebraucht wird, sind in diesem Szenario nicht mehr ohne weiteres anwendbar. Dementsprechend entstand das neue Konzept des *Online Learning*. Hierbei wird das Modell kontinuierlich mit den neuen Trainingsdatenpunkten aktualisiert, sobald diese eingetroffen sind. Ein wichtiger Aspekt des *Online Learning* ist, dass es die zeitliche Dimension der Daten miteinbezieht. Die Datencharakteristiken sind nicht zwangsläufig stabil und können sich über Zeit ändern. Dieses Phänomen ist bekannt als Concept Drift [29]. Die Fähigkeit zur Adaption an sich ändernden Datencharakteristiken über Zeit ist eine wichtige Voraussetzung für die Effektivität eines *Online Learning* Algorithmus.

Die Notwendigkeit der Verarbeitung, kontinuierlicher, möglicherweise unendlicher Datensequenzen löste die Entwicklung von verschiedenen verteilten Streamverarbeitungs Frameworks aus. Diese Arbeit Untersucht die Implementierung eines skalierbaren adaptiven online learning algorithmus in einem dieser Systeme, Apache Flink. Diese Arbeit konzentriert sich auf die skalierbare Implementierung des *Bagging Ensamble* Algorithmus.

Dieser beinhaltet keine implizite Möglichkeit concept drifts zu verarbeiten, wir untersuchen verschieden Optionen zur Integration eines concept drift Mechanismus. Zusätzlich analysieren wir verschieden Techniken um den Algorithmus robuster gegenüber falschen Drift detektionen zu machen.

Diese Arbeit Untersucht außerdem verschiedene Evaluationstechniken für online Algorithmen, weil traditionelle Methoden die im offline learning verwendet werden, zum Beispiel Kreuzvalidierung, im Falle unendlicher Datenströme nicht ohne weiteres anwendbar sind.

3. Introduction

We live in the age when the speed and amounts of data produced are enormous. According to a recent IDC report [59] the data generated in 2014 is estimated to be 4.4 zettabytes (trillion gigabytes) and this figure is growing exponentially. The report predicts that by 2020 it will reach 44 zettabytes. In order to extract useful information from this big amount of data new efficient and scalable algorithms are necessary. The use of traditional machine learning strategies where data is stored, the learning model is built offline and used on new unseen data, is limited by the inability of modern systems to process these amounts of data in main memory. This sparked the development of multiple frameworks for distributed processing of large data, like Apache Hadoop[1] or Apache Spark[2].

But in many cases data comes in form of potentially infinite streams. The models need to be updated continuously and be ready to be used at any moment for making time-critical decisions. This triggered the establishment of a whole new concept – *Online Learning*. In online learning the model is trained on new samples as they arrive but it can be used on new unseen data at any point of time. The areas where online learning gained its ultimate popularity are Fraud and Spam Detection, Adaptive Recommender Systems, Time Series Analysis and Sensor Data Analysis.

One important characteristic of online learning is the temporal dimension. Over time the data distribution might change and online learning systems need to be able to adapt to these changes. This phenomena is called *Concept Drift* [29]. One example of such change might be the change in the interests of a person when doing online shopping. The adaptive recommender system needs to be able to make valid recommendations based on the new interests of the person.

There are many frameworks capable of processing data streams. They include Apache Storm[3], Google Cloud Dataflow[4] and Apache Samza[5]. However the systems capable of performing scalable machine learning on big data streams are at the stage of early development. One of such systems is Apache SAMOA [3]. Algorithms implemented in SAMOA can run on top of different distributed stream processing engines, including Apache Flink.

Apache Flink [2] is a scalable stream processing engine which provides fault tolerance, exactly-once message delivery guarantees, high-throughput and low-latency data process-

[1]Apache Hadoop: http://hadoop.apache.org/
[2]Apache Spark: http://spark.apache.org/
[3]Apache Storm: http://storm.apache.org/
[4]Google Cloud Dataflow: https://cloud.google.com/dataflow/
[5]Apache Samza: http://samza.apache.org/

ing and strong consistency guarantees in presence of stateful computations [4]. These properties together with the set of highly expressive data transformations available in Streaming API make Flink a promising candidate system for implementation of scalable machine learning algorithms.

3.1. Motivation

Apache Flink is a relatively new and promising open source tool in the world of the large scale distributed systems processing. As such, its intended machine learning library is for the moment limited, in particular for the streaming API. Although Apache SAMOA provides many different algorithms which can run on top of Flink, we believe that native implementation of *ML* algorithms would provide better performance and can benefit from exploitation of multiple optimizations specific to Apache Flink.

In this thesis we explore the possibility of implementing scalable adaptive classification algorithms in Apache Flink. As part of the work, we define the abstractions for implementation of different *ML* pipelines in Apache Flink. As the ability to adapt to concept drift is one of the most important characteristics of a streaming algorithm, we aim at studying available drift detection and adaptation techniques and identifying which ones are the most suitable for implementation in Apache Flink.

In order to evaluate our implementation we study and implement the evaluation technique for distributed online learning algorithms. In particular, our main concerns are adaptability of the algorithm, latency of data processing and the throughput of the system.

3.2. Contributions

The contributions of this thesis are:

1. We defined the abstractions for implementation of scalable streaming algorithms in Apache Flink (Section 6.3) and proposed a generic learning model which can be used with arbitrary classification algorithm (Section 7.1).

2. We proposed a concept drift adaptation approach (see Algorithm 8) which makes the model more resistant to false drift detections.

3. We implemented a generic adaptive bagging algorithm in Apache Flink, which can be used with arbitrary base learner classifier and change detector. For our evaluation we implemented a Naive Bayes classifier and the *ADWIN* change detector (Section

7.2). The algorithm proved to perform better in terms of classification accuracy and execution time than the adaptive algorithms available in Apache SAMOA (Section 8.6).

4. We explored the evaluation techniques which can be used to measure the accuracy of distributed algorithms and implemented k-fold bagging evaluation technique in Apache Flink (Section 8.2). We implemented and compared several approaches to measure accuracy of the algorithm in presence of concept drifts (Section 8.1).

3.3. Thesis Outline

In Section 4 we present a definition of online learning and requirements for large-scale online predictive methods. Furthermore, we provide a definition and types of *Concept Drift* and requirements for adaptive algorithms. Section concludes with an overview of typical applications which require the introduction of concept drift detection and adaptation techniques.

Section 5 provides a detailed classification of existing change detection and adaptation algorithms. It also presents an overview and detailed description of several single-model and ensemble algorithms. Section concludes with a discussion of further problems which may influence the efficiency of concept drift adaptation.

Section 6 provides an overview of Apache Flink Streaming engine and concepts which are required for understanding of implementation details presented in the following sections. Furthermore, the section discusses the components of adaptive online learning pipeline in Flink.

Section 7 introduces the implementation details of the two algorithms we chose. The section discusses and justifies the design choices we made during implementation.

In Section 8 we provide the results of our in-depth evaluation of the implemented methods. We describe an evaluation method and its implementation. Furthermore, the section is divided into four parts: the first part presents the evaluation results on artificially generated datasets, the second part provides the results of evaluation on some real datasets, third part presents the evaluation of both algorithms in terms of latency and throughput. Finally, we compare the performance of our algorithm with several algorithms available in streaming machine learning platform Apache SAMOA [3].

Section 9 presents an overview of the thesis results and future work.

4. Background

This chapter presents the background of the thesis. Section 4.1 elaborates on the concept of *Online Learning* and the requirements which arise when learning from data streams. Section 4.2 introduces the definition and types of *Concept Drift*. Learning in presence of concept drift entails several other constrains. These constrains are detailed in the Section 4.2.3. The section concludes with examples of typical applications which require explicit or implicit handling of concept drift.

4.1. Online Learning

Traditional machine learning algorithms operate in offline setting. That is when we have all the data from the start and use this data to train our model. In most cases these algorithms are of iterative nature, so each sample is used for training more than once. The process results in a model that can be used for prediction on new unseen data items [28]. In the era of *Big Data* a need emerged to process huge amounts of data over short or long periods of time without storing it. This makes the previously proposed approaches infeasible.

The requirements for a large-scale predictive methods are presented in [26]:

1. The algorithm should be trained continuously on blocks of data or separate samples, rather than require all of the data from the beginning.

2. The algorithm should use each sample for training only once.

3. The size of the model should be independent from the size of the data and the model should require an approximately constant amount of memory.

4. The algorithm should be ready for use at any point of training process.

But depending on available computational resources some of these conditions may be relaxed. For example, there are algorithms with partial memory which store a window of fixed or varying number of samples [28]. Section 5.1 presents the different types of algorithms in more detail.

Another important aspect of online learning is temporal locality. Data streams tend to evolve over time and models built with old data may loose accuracy as the distribution from which the samples are drawn changes. So, another important criteria for an online leaning algorithm is ability to handle evolution of the data over time [6].

Examples of applications which can make use of online learning range from *Time Series and Sensor Data Analysis*, *Fraud and Spam Detection* to *Sentiment Analysis* and *Adaptive Recommender Systems* [53, 46, 43, 39]

4.2. Concept Drift

This section introduces a formal definition and existing types of *Concept Drift*, criteria for evaluation of the ability of the algorithm to adapt to concept drift and the typical applications which require an adaptation mechanism.

4.2.1. Definition

Learning algorithms operate in a dynamic environment prone to unexpected changes and these algorithms are expected to adapt to these changes quickly by incorporating new data. This unexpected change in the underlying data distribution is commonly referred to as *Concept Drift* [28]. Although concept drift is an issue of many learning algorithms like classification, regression and clustering, this work without loss of generality is focusing on handling concept drift in classification setting.

Formally a classification task is defined as follows. The system is given a set of learning examples in form of a pair (X, y), where $X \in \Re^n$ is a set of input features and $y \in \Re^1$ is a target variable [28].

According to the Bayesian Decision Theory [25] a classification problem can be defined in probabilistic terms. The *a priori probability* $P(c_i)$ $\forall c_i \in C$ (where C is the set of all possible classes) is the probability that the next sample which comes for classification is of class c_i. This probability reflects our prior knowledge about how likely are samples of one class compared to others. The *class-conditional probability density function* $p(X|c_i)$ $\forall c_i \in C$ reflects the probability distribution of X when the class is known to be c_i. The final classification decision is based on the *posterior probability*

$$P(c_i|X) = \frac{p(X|c_i)P(c_i)}{p(X)} \tag{1}$$

Over time the concept drift may happen. The concept drift is said to happen between time points t_0 and t_1 if

$$\exists X : p_{t0}(X, c) \neq p_{t1}(X, c) \tag{2}$$

where p_{t_i} is the joint distribution at the time t_i:

$$p(X, c_i) = P(c_i|X)p(X) = p(X|c_i)P(c_i) \tag{3}$$

So, the concept drift can take place in case one of the following change [34]:

- a priori probability $P(c_i)$
- class-conditional probability $p(X|c_i)$
- and consequently, posterior probability $P(c_i|X)$

Based on the type of change in the data distribution two types of concept drift are distinguished [28]:

- **Real Concept Drift** is defined as the change in posterior probabilities $P(c_i|X)$.
- **Virtual Concept Drift** happens when the distribution of incoming data $p(X)$ changes with no effect on posterior probabilities.

Although both types can affect the accuracy of the classifier in different ways, it is the real concept drift that requires a substantial change in the underlying model in order to be adapted to. This is the reason why research in this area is mainly concentrated on handling the real concept drift.

4.2.2. Types of Concept Drift

Based on the way and the speed the old concept is substituted by the new one several types of concept drift can be distinguished. The different types of concept drift are presented in Figure 1 [28]. It is a one-dimensional example which depicts the change of the mean of a variable.

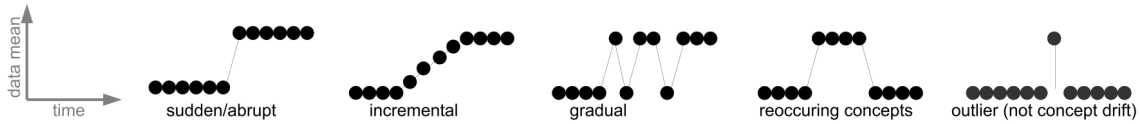


Figure 1: Types of Concept Drift [28]

The *abrupt* change is characterized by a sudden switch from one concept to another. The example of this is the sudden change in stock market prices and the behavior of the people

due to some political, economic or social event. The *incremental* change refers to gradual but consistent switch of the concept. The *gradual* concept drift entails some fluctuation between the old and the new concepts before the final switch. An example of such a drift might be the change in the music taste of the user. Another phenomena is the *reoccurring* concept. For example, depending on the season of the year several products are in higher demand. Finally, there also can be some noise, or outliers, in the data, which do not require any change of the model. Ideally the learning method should be able to adapt to any of the aforementioned concept drift types. In practice, however, methods generally concentrate on one or two of them.

4.2.3. Learning in Presence of Concept Drift

Apart from the requirements presented in Section 4.1 several additional specifications are imposed on the online learning methods in the presence of concept drift. These are also the criteria for evaluation of the ability of the algorithm to handle concept drift:

- **Delay.** Reflects how fast the method can detect/adapt to the concept drift.
- **Resistance to noise.** Characterizes the ability of the method to distinguish the noise in the data from the real concept drift.
- **Cost of adaptation.** Defines whether the algorithm needs to recompute the model from scratch after detecting the concept drift, or the localized re-computation is enough [21].

4.2.4. Typical Applications

According to [28] applications requiring the adaptation to concept drift can be grouped into four categories:

- **Monitoring and control.** In this scenario the data is typically presented in form of the time series and the task is to detect the anomalies. Examples include activities on the web, computer networks, telecommunication and financial transactions.
- **Management and strategic planning.** This category includes predictive analytics tasks, such as estimating the creditworthiness, stock prises behavior or electricity demand.

- **Personal assistance and information.** This category encloses tasks like recommender system, personal news or mail categorization.

- **Ubiquitous environment applications.** This category comprises systems which interact with changing environments, for example, sensor data analysis.

Different types of applications may lay different priorities on the requirements presented in Section 4.2.3. Some, like financial transactions monitoring task may put a delay as the highest priority as they would need to detect the fraud as soon as possible. Others, like ubiquitous environment applications, require a great resistance to noise to avoid false reaction of the system.

5. Concept Drift Handling Methods

This section presents an overview of existing concept drift handling methods. Section 5.1 presents an elaborate classification of methods. In Section 5.2 we detail several algorithms which use one learner instance. Section 5.3 presents a couple of ensemble methods which exploit several instances of the learner. Finally, Section 5.4 discusses the issues which may greatly influence the adaptation to concept drift and present some possible solutions.

5.1. Taxonomy of methods

This section presents an in depth classification of existing change detection/adaptation methods. According to [28] and [38] change adaptation methods are generally classified with respect to *memory* they require, *type of forgetting mechanism* they offer, *type of information* they use for detecting/adapting to drift, *model management* (number of base learners).

5.1.1. Type of Indicator

This categorization reflects the type of information that is used in order to detect the concept drift and adapt a model to it [38]. The range of such indicators is wide, the most common include:

- **Classification accuracy** is by far the most common type of indicator. The drop in accuracy or the change in the average distance between the classification errors may signify the change in the underlying concept. The methods based on classification accuracy must be cautious of the drop in accuracy due to the noisy data.
- **Time stamp** can be used as a change indicator as well. For example, it can be used as an input variable to the classification tree and when there is a slit on this variable – the concept drift is signaled.
- **Model complexity** change for some classifier models can signal the concept drift. The example of this can be the rapid increase in the amount of rules in the rule-based classifier of the number of support vectors in SVM.

This list does not cover all the existing methods. Many of them use the combinations of different measures as well as propose various custom concept drift indicators.

5.1.2. Model Dependency

This categorization stems from the fact that several concept drift detectors can be used with any existing base learner while others are tightly coupled with a specific algorithm. For example methods which use the accuracy as a concept drift indicator are generally *model-independent* while those which use the time stamp as an input variable to the tree can only be used with classification trees, consequently are *model-dependent*.

5.1.3. Processing Mode

By processing mode the algorithms can be divided into *single example* and *window-based*. *Single example* algorithms process samples one by one in the order they arrive. Each sample is used to update the model and then discarded. Consequently, these algorithms do not have an access to the previous samples. *Window-based* algorithms are alternatively called *batch-based*. They process data in windows of fixed or variable size. Algorithm may opt to adapt the size of the window based on the stability of the system. When the concept drift is detected the window size is decreased in order to contain only the samples from the new concept, whereas when the concept is stable the window size is increased to improve the model by incorporating more samples.

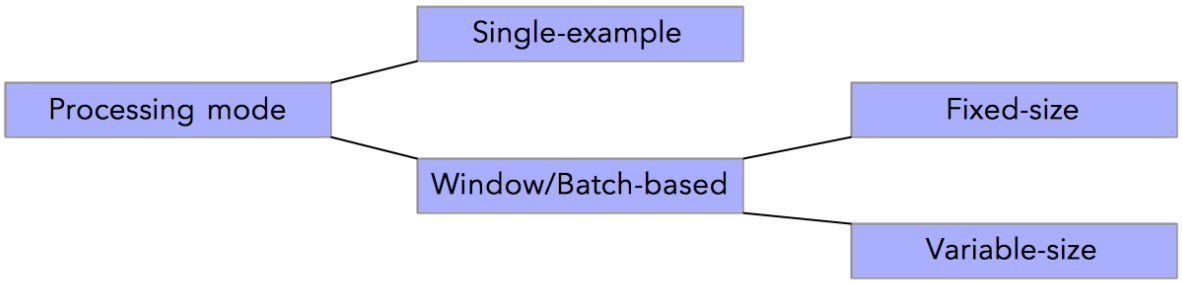


Figure 2: Processing Mode

5.1.4. Adaptation Mode

There are two distinct ways to adapt to the concept drift – detect it *explicitly* and rebuild/enhance the underlying model or use some forgetting heuristics regardless of

whether the drift actually happened or not [38]. This second way is called *implicit detection*. The drawback of implicit detection is its high delay in concept drift adaptation. As the old concept is being forgotten with a constant speed it also cannot build highly accurate models when the concept is stable for long periods. The *explicit detection* algorithms can be further divided by the way they adapt the model after the change is detected. The easiest way is to *replace* the model *completely* upon the detection of the drift [28]. This category includes also the algorithms which work in so-called *warning-alarm mode*. The algorithm has two levels/thresholds – after the first one is reached it is signaling the concept drift *warning*. At this point algorithm starts to build the new model but still uses the old one for classification. When the second level, *alarm*, is reached – the model is replaced with a new one. More sophisticated way to adapt to the concept drift is to enhance the model to incorporate the new knowledge without fully rebuilding it. There are also methods which use the combination of aforementioned methods. These methods are in the *hybrid* category.

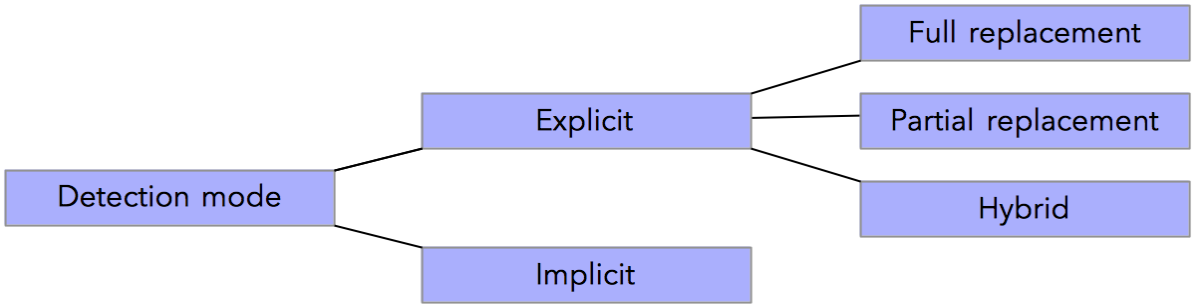


Figure 3: Adaptation Mode

5.1.5. Model Management

In terms of model management all the learning techniques can be divided into *single-model* and *ensemble* approaches. Ensemble model uses the combination of several ensemble members to make a final prediction. These models have gained extreme popularity lately as the ensemble of classifiers can often outperform a single classifier in the presence of concept drift [54]. The ensemble approaches can be further divided into the following groups [37]:

- **Dynamic combiners.** Given a set of learners trained on different subsets of data the final decision is the combination of the outputs of these learners. The adaptation to concept drift is done by changing the rule used to combine the decisions of the

individual learners. For example, each learner gets a weight that corresponds to its accuracy on recent data.

- **Updated training data.** This group of algorithms uses the new data to update all ensemble members. In order to keep the ensemble diverse, different ways to choose the data distribution between members exist. They do not have an explicit mechanism to track the concept drift and are generally used with external change detectors.

- **Structural changes to the ensemble.** This group comprises the methods which adapt to the drift in the data by making changes to the structure of the ensemble. For example, a new member is built on each new chunk of data and old members are discarded when they loose their accuracy.

The list in not exhaustive and these methods can also be combined to achieve better performance.

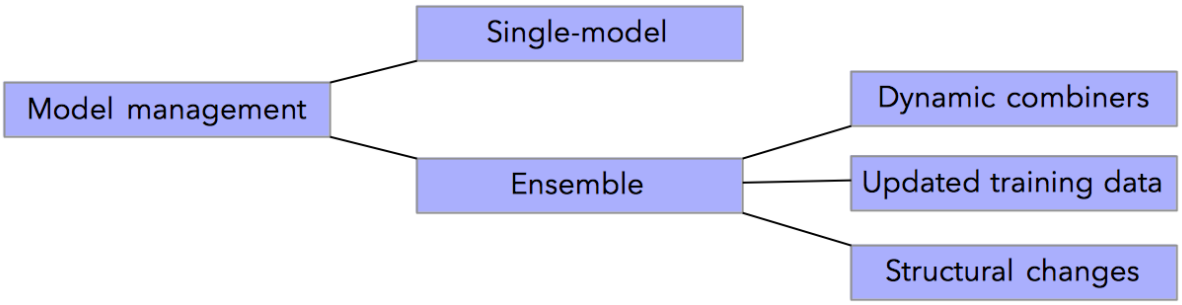


Figure 4: Model management

5.2. Single Model Methods

This section presents the methods which use one instance of the learner.

5.2.1. Classifier-free methods

This section presents several concept drift detection methods which can be used with any classifier.

ADWIN

Adaptive Windowing (ADWIN) method [15] maintains the sample window of maximal size N. Whenever the change is detected it shrinks the window to include the samples only from the new concept and when the concept is stable the window grows automatically. The input to the algorithm is a confidence value $\delta \in (0, 1)$ and (possibly infinite) sequence of values $x_1, x_2, ..., x_t, ...$ These values can be, for example, the current accuracy measurements of the classifier after each sample. For every t the value x_t is drawn from some distribution D_t independently. We denote μ_t as the expected value when x_t is drawn from D_t. The value μ_t is unknown to us at every point of time t.

ADWIN keeps a sliding window W over the newest n samples. Let $\hat{\mu}_W$ denote the observed average of the samples in the window W. The main idea of the algorithm is that when two subwindows of W have a significant difference in averages we can assume that the expected values are different and samples of these two subwindows are drawn from different distributions. The older subwindow is then dropped. Algorithm 1 presents the detailed description of *ADWIN* [15].

Let $n = n_0 + n_1$ denote the length of the window W and n_i the length of the subwindow W_i. The value of $\hat{\mu}_{W_i}$ denotes the average of the subwindow W_i. We define:

$$m = \frac{1}{\frac{1}{n_0} + \frac{1}{n_1}} \tag{4}$$

$$\delta' = \frac{\delta}{n} \tag{5}$$

The value of the threshold ϵ_{cut} is computed as follows:

$$\epsilon_{cut} = \sqrt{\frac{2}{m} \cdot {\sigma_W}^2 \cdot \ln \frac{2}{\delta'}} + \frac{2}{3m} \ln \frac{2}{\delta'} \tag{6}$$

The main limitation of *ADWIN* is memory and processing time requirements. Due to a big amount of possible split points of the window W it is very costly to check the condition for all subwindows. The authors proposed a modification of this method which uses a variant of the exponential histogram [22] for compression and requires only $O(\log W)$ memory and update time. *ADWIN* is also slow to detect the gradual concept drift due to the fact that it gives the same weight to all the samples in the window. In the case of slow change it would be beneficial for the algorithm to give a larger weight to more recent samples [14].

Algorithm 1 Adaptive Windowing

```
Initialize Window W
for all t > 0 do
    W ← W ∪ {x_t}                                  ▷ Add x_t to the head of W
    repeat
        Drop elements from the tail of W
    until |μ̂_{W_0} − μ̂_{W_1}| ≥ ε_cut holds for every split of W into W = W_0 · W_1
    output μ̂_W
```

DDM

Drift Detection Method (DDM) [30] is based on *Statistical Process Control (SPC)*, a technique to monitor and control the quality of the product during a continuous manufacturing [28]. The input to the algorithm is a sequence of pairs (x_i, y_i) where x_i is a training sample and y_i is a boolean value which indicates whether the classifier did a correct prediction on sample x_i. Let p_i denote the probability of $false$ prediction in Bernoulli trials and s_i as its standard deviation. The standard deviation is calculated as follows:

$$s_i = \sqrt{\frac{p_i\left(1-p_i\right)}{i}} \tag{7}$$

DDM exploits the fact that for a sufficiently large number ($n > 30$) of samples the Binomial distribution can be closely approximated by a Normal distribution with the same mean and variance.

The algorithm manages two registers p_{min} and s_{min}. For every new sample p_i and s_i are calculated and algorithm checks whether $p_i + s_i < p_{min} + s_{min}$. In this case p_{min} and s_{min} are updated with new values. Algorithm works in a *warning-alarm* way as explained in Section 5.1.4. The condition checked at every step is:

$$p_i + s_i \geq p_{min} + \alpha \cdot s_{min} \tag{8}$$

Value of α is controlling the confidence level for drift. The authors propose $\alpha = 2$ for *warning* level and $\alpha = 3$ for *alarm* level. This corresponds to 95% and 99% confidence for warning and alarm respectively.

The main advantage of *DDM* is that it has $O\left(1\right)$ memory and processing time complexity in case we start to build a new model at a warning level point. The drawback is that the algorithm requires at least 30 time steps after the drift is detected [56]. Also, the

approach is very efficient with detecting abrupt changes but has difficulties with gradual drifts. If the change is slow it can pass without triggering the alarm level [9].

EDDM

Early Drift Detection Method (EDDM) was proposed as an enhancement of *DDM* to improve the performance in presence of gradual drifts [9]. The main idea of the algorithm is to take into account not only the number of the errors but also the distance between two consecutive errors. It is based on the fact that when the concept is stable the distance between the errors increases but when the concept drift is taking place – it decreases notably.

Let p'_i be the average distance between two errors and s'_i its standard deviation. We track two values p'_{max} and s'_{max} which reflect the point where the distance between two errors reached its maximum. Whenever we encounter $(p'_i, s'_i) : p'_i + 2s'_i > p'_{max} + 2s'_{max}$ we update these values. The condition checked on each step is:

$$\frac{p'_i + 2s'_i}{p'_{max} + 2s'_{max}} < \alpha \tag{9}$$

As with *DDM* the value of α will regulate the extent of the drift and the smaller it is – the bigger confidence of the drift is. The authors propose to choose the values between 0.95 and 0.90.

The drawback of the method is similar to the *DDM* – it searches for concept drift only when a minimum of 30 errors have happened. It works better in presence of gradual concept drift than *DDM* but is more sensitive to noise [9].

ECCD

Exponentially Weighted Moving Average for Concept Drift Detection (ECCD) method uses *Exponentially Weighted Moving Average (EWMA)* chart to monitor the misclassification rate of a streaming classifier [48]. The input to the algorithm is a sequence of boolean values $x_1, x_2, ..., x_t, ...$, where $x_i = 0$ when the predicted label was correct and $x_i = 1$ if it was incorrect. Let μ_t be the mean of the stream at time t, μ_0 the mean of the stream before concept drift and μ_1 – after it. We know that before the change $\mu_t = \mu_0$. More specifically, the values x_i will fluctuate around the value μ_0. The *EWMA* estimator of μ_t is defined as follows:

$$\begin{aligned} Z_0 &= \mu_0 \\ Z_t &= (1 - \lambda)\, Z_{t-1} + \lambda x_t \end{aligned} \tag{10}$$

It is a recent estimate of μ_t which with the help of configurable parameter λ progressively downweights the older data. According to Bernoulli distribution the pre-change standard deviation of *EWMA* estimator will be defined as [58]:

$$\sigma_{Z_t}^2 = \sqrt{p_0 \left(1 - p_0\right) \frac{\lambda}{2 - \lambda} \left(1 - \left(1 - \lambda\right)^{2t}\right)} \tag{11}$$

where p_0 is a probability of misclassifying a point before the drift happens. As this parameter is unknown at the beginning we need to estimate it. The estimator $\hat{p}_{0,t}$ is defined as follows:

$$\hat{p}_{0,t} = \frac{1}{t} \sum_{i=1}^{t} x_i = \frac{t-1}{t} \hat{p}_{0,t-1} + \frac{1}{t} x_t \tag{12}$$

Estimator Z_t gives more weight to recent samples than $\hat{p}_{0,t}$ which means that it is more sensitive to change. When the drift occurs Z_t is supposed to converge to new value μ_1 faster. Consequently, when the difference between these two estimators becomes significant it means that the drift takes place. So, we signal the change when:

$$Z_t > \hat{p}_{0,t} + L\sigma_{Z_t} \tag{13}$$

The control limit parameter L allows to achieve the constant rate of false positive change detections. This parameter depends on the estimate $\hat{p}_{0,t}$ at a given point of time t. Authors use regression techniques to arrive at the polynomial approximations for the computation of L. Algorithm 2 presents the complete method [48].

Algorithm 2 Exponentially Weighted Moving Average for Concept Drift Detection

1: Choose value for λ
2: $Z_0 = 0$, $\hat{p}_{0,0} = 0$
3: **for all** x_t **do**
4: $\quad \hat{p}_{0,t} = \frac{t}{t+1} \hat{p}_{0,t-1} + \frac{1}{t+1} x_t$
5: $\quad \hat{\sigma}_{x_t} = \hat{p}_{0,t} \left(1 - \hat{p}_{0,t}\right)$
6: $\quad \hat{\sigma}_{Z_t} = \sqrt{\frac{\lambda}{2-\lambda} \left(1 - \left(1 - \lambda\right)^{2t}\right) \hat{\sigma}_{x_t}}$
7: $\quad$ Compute value of L_t based on current value of $\hat{p}_{0,t}$
8: $\quad Z_t = \left(1 - \lambda\right) Z_{t-1} + \lambda x_t$
9: $\quad$ **if** $Z_t > \hat{p}_{0,t} + L_t \hat{\sigma}_{Z_t}$ **then**
10: $\quad\quad$ Signal Concept Drift

Algorithms, like *ADWIN*, *DDM*, *EDDM* and *ECCD* are called drift detectors. The way they can be used in distributed setting is creating a separate instance of drift detector for each instance of learner. Another way would be to have a global instance of drift detector, but this option would cause a potential bottleneck in the distributed system.

Paired Learners

The method proposed in [8] adopts a strategy of combining two learners – stable and reactive ones. The stable learner is useful during the periods of time when the concept does not change and by incorporating more data samples is very accurate. The reactive learner is responsible for drift detection and is much more accurate during the periods of concept change. By combining these two learners authors claim to have achieved the accuracy comparable or better than that of many ensemble approaches.

The stable learner S is trained with all the samples starting from the last concept drift. The reactive learner R_w is trained on the window of w recent samples. The stable learner is used for classification while the reactive one servers as an indicator of drift. The performance of both learners is tracked and when the reactive learner starts to outperform the stable learner it might be an indication of concept drift, as it means that the older samples are harming the performance of the stable learner.

In order to identify the point where it is beneficial to substitute the stable learner with the reactive one the simple accuracy indicators over the last w samples is not enough. In order to avoid the situations where the reactive learner was outperforming the stable one at some point of time but lost its dominance later (this can happen, for example, when the reactive learner has learned from the noisy data) authors proposed another method to track the performance of the learners. They use the circular buffer C of w bits. When the reactive learner classifies a sample correctly while the stable one not, a new bit in C is set to 1. In other cases – the bit is unset. When the proportion of bits in the buffer surpasses the threshold θ the stable concept is substituted with a reactive one and the buffer is reset. Algorithm 3 introduces the Paired Learners.

The experiments showed that the method works comparably good or better in comparison with several ensemble methods. It is an achievement taking into account that it uses only two learners while the ensemble methods utilize up to 50. The main drawback of the algorithm is the necessity to keep the window of samples. The parameter w, the size of the window, has a great influence on how the algorithm performs in presence of abrupt or gradual drift. Also, if the algorithm is unable to "un-learn" the given sample, the new model will have to be built on the last w samples after the arrival of each new sample.

Algorithm 3 Paired Learners

Input:
w: window size
θ: threshold for substitution of stable learner
Initialize stable learner S, reactive learner R_w, circular buffer C
for all (x_t, y_t) **do**
    $y_S = S.classify\,(x_t)$
    $y_R = R_w.classify\,(x_t)$
    if $y_S \neq y_t \wedge y_R = y_t$ **then**
        $C.setBit\,()$
    else
        $C.unsetBit\,()$
    if $\theta < C.proportionOfSetBits\,()$ **then**
        $S = R_w$
        $C.unsetAllBits\,()$
    $S.train\,(x_t, y_t)$
    $R_w.train\,(x_t, y_t)$

5.2.2. Classifier-specific methods

This section introduces several algorithms which use a specific implicit drift adaptation mechanisms.

Hoeffding Trees

Very Fast Decision Tree (VFDT) [24] is a decision tree induction algorithm that was designed to handle high speed data streams using constant memory and time per example. Although the basic algorithm assumes that distribution of samples does not change over time, it is a basis for a great amount drift adaptation algorithms. These *VFDT* enhancements are examined later in this section. The appealing feature of the algorithm is that it does not store the samples and at the same time guarantees that with a sufficient number of samples the output will be asymptotically identical to that of a conventional learner. The algorithm is based on the idea that a small sample size is ofter sufficient to choose the best candidate attribute for a split. The optimal size of the sample within a predefined precision is defined by Hoeffding bound. Hoeffding bound states that with confidence $1-\delta$ after n independent observations, the true mean of real-valued variable r with range R will not differ from the observed mean by more than:

$$\epsilon = \sqrt{\frac{R^2 \ln\left(\frac{1}{\delta}\right)}{2n}} \tag{14}$$

The algorithm works as follows [29]. When the new sample arrives the tree is traversed from the root to the leaf corresponding to this sample. Each leaf holds some statistics which is recomputed when a new sample arrives. Let $H(\cdot)$ be the information gain function for the attribute. Lets assume that after observing n examples in the leaf, x_a is an attribute with the highest $H(\cdot)$ and x_b is one with the second-highest $H(\cdot)$. Let $\Delta\bar{H} = \bar{H}(x_a) - \bar{H}(x_b)$. If $\Delta\bar{H} > \epsilon$ holds then according to Hoeffding bound with a confidence of $1 - \delta$ attribute x_a is the best attribute to split on. The leaf is transformed into a decision node. In order to avoid evaluating these conditions on every step the algorithm adopts a strategy to compute $H(\cdot)$ only when the minimum of k samples arrived where k is a user-defined parameter.

In order to avoid a case when two attributes exhibit an equal values of $H(\cdot)$ for a very long time because they are equally good for splitting another constant τ was introduced. So when we reach the condition $\Delta\bar{H} < \epsilon < \tau$ the split is done on the currently best attribute.

Concept-adapting Very Fast Decision Tree (CVFDT) [32] is proposed as an enhancement to *VFDT*. The basic idea of *CVFDT* is the same but the algorithm manages to adapt to the change by building an alternative subtree whenever the old one becomes questionable and replaces it when the new subtree becomes more accurate. In particular, it keeps a window w over the most recent examples. After every n new examples the algorithm determines again the best candidate at every decision point. If on any node there is a new best candidate the algorithm suspects that there was a concept drift and begins to build an alternative subtree starting from this node while still keeping and updating the old one. It uses the samples in the window as a validation set and when a new subtree performs on average better than the old one – the old one is pruned. If after the a maximum number of validations the old one performs on average better – the new subtree is pruned. If the amount of alternative subtrees reaches the maximum number – the algorithm prunes the lower-performing ones [11].

A good point about this algorithm is that it does not require to rebuild the model from scratch whenever there is a change in the data but rather the necessary changes are done on the subtrees. But as an implicit change adaptation algorithm it is slow to react to the concept drift. Another drawback of the algorithm is the need to define several parameters, such as n – the number of samples the algorithm needs to do another check for the existence of better candidates. This implicitly means that we do not expect the

change to happen more often than every n samples which is an unrealistic assumption [16].

Hoeffding Option Tree (HOT) [47] is based on a very similar idea as *CVFDT* except that it does not keep the window of samples but adopts another strategy to prune the low-performing subtrees, or *options*.

An enhancement of *CVFDT* algorithm was proposed in [16]. The authors use *Hoeffding Window Tree (HWT)* in combination with such drift detectors as *ADWIN* and *EWMA*. *HWT* is a decision tree based on a keeping sliding window of the last instances on the stream. The main differences between *CVFDT* and *HWT-ADWIN* are that the alternative subtrees are created as soon as the change is detected and there is no need to wait for another n samples to arrive and that the old trees are replaced as soon as there is evidence that the new tree is more accurate instead of waiting for another fixed amount of samples. This allows the algorithm to react and adapt to changes more quickly.

Prototype-based Learning

The different approach to concept drift handling was proposed in [50]. *SyncStream* is a prototype-based classification model, which maintains a set prototypes of the data in a custom structure called the PTree. The authors also proposed two variants of external concept drift detectors. One is based on *Principal Component Analysis (PCA)* and another on statistical analysis.

The *PCA*-based drift detector as well as *ADWIN* splits the data points corresponding to label l into two subsets with respect to the time of their arrival, D_t and D_{t+1}. The algorithm then computes the angle between the major principal components of these two sets. If the angle exceeds the certain threshold ϵ the concept drift is alarmed. Another external concept drift detector proposed is based on statistical analysis. It is similar to *ADWIN* but uses the extended Brunner and Munzel's generalized Wilcoxon test statistic [20, 44] to compare the differences between the class distributions of two data sets.

The PTree consists of two levels. The first level stores the set of $maxP$ prototypes which capture the current concept and the second level holds a set of $maxC$ older concepts also represented by the set of prototypes. In order to make a prediction on newly arrived sample x the first level is used, the prototypes of current concept P. Let $y \in P$ be the prototype closest to x. The label of this prototype will be a classifier's prediction. Also, on each step the representativeness of the prototype is updated as follows:

$$Rep(y) = Rep(y) + Sign(x.trueLabel, y.label) \tag{15}$$

Algorithm 4 SyncStream

```
1: Input: maxC, maxP, D
2: while NoError do
3:     for all x ∈ D do                                   ▷ for each sample
4:         y = NN(PTree, x)                               ▷ find nearest prototype
5:         x.label = y.label
6:         if x.trueLabel == x.label then
7:             y.Rep++                                    ▷ increase representativeness
8:         else
9:             y.Rep−−                                    ▷ decrease representativeness
10:        PTree.insert(x)
11:        if PTree.prototypeSize == maxP then
12:            S_n = PTree.getNegativePrototypes()
13:            PTree.remove(S_n)  ▷ remove the prototype of negative representativeness
14:            S_u = PTree.getUnchangedPrototypes()
15:            P = ConstrainedSync(S_u)                   ▷ summarize
16:            PTree.remove(S_u)
17:            PTree.insert(P)
18:        if PTree.conceptSize == maxC then
19:            PTree.removeOldestConcept()
20:        if Concept Drift then
21:            P_s = Sample(PTree.prototypes)
22:            C_s = ConstrainedSync(P_s)
23:            PTree.prototypes.clear()
```

where the initial value of $Rep(y)$ is 1 and $Sign(x, y)$ is the sign function which equals 1 if $x = y$ and -1 otherwise. Obviously in order to be able to find the closest prototype there should be a concept of *distance* between the samples. The point x is then added to the prototype level. As the amount of prototypes that is kept on prototype level of the tree is limited to $maxP$, authors proposed to use a data summarization technique by adapting synchronization-inspired clustering [18, 51]. As the method has to be applied to the supervised learning problem the authors modified the clustering method to take into account the label data. Now, when the number of the prototypes on the prototype level exceeds $maxP$, the algorithms performs the data summarization.

When the concept drift is detected by the external detector, the current prototypes on the prototype level are further summarized and then saved to the concept level. If the amount of concepts on the concept level exceeds $maxC$ the oldest concept is deleted. Algorithm 4 describes the *SyncStream* in more detail.

The authors claim that the method can handle both the abrupt and gradual concept drift. The experiments show that it is also resistant to noise in the data. The drawbacks are the processing time and memory requirements of the algorithm. By configuring parameters $maxC$ and $maxP$ the maximum size of the PTree can be controlled. But the higher $maxP$ value is – the longer processing time is required, because each incoming sample needs to be compared to all the prototypes on the prototype level. Also, the concepts saved on the concept level of the algorithm can be used to manage the reoccurring concepts.

The authors do not discuss the potential of this algorithm to be used in distributed setting. But taking into account the operations performed on *PTree*, like nearest neighbor search and data points summarization, we assume that it would require to store the tree in some sort of shared memory, which could be accessed by each worker. This could be a potential bottleneck.

SVM-based method

The method proposed in [35] utilizes *Support Vector Machine (SVM)* to detect concept drift. The algorithm uses a window of training examples. The size of the window adjusts to the speed and amount of concept drift. The key idea of the method is to select the size which minimizes the generalization error on new examples. To estimate the generalization error the method uses a special form of $\xi\alpha$-estimates [33] which is an efficient method to evaluate the performance of *SVM*.

$\xi\alpha$-estimators are based on the leave-one-out error estimation. Having a training set $S = ((x_1, y_1), ..., (x_n, y_n))$, the first element (x_1, y_1) is removed. This set is used to train a classifier. The classifier is then tested on the held out example (x_1, y_1). If the classifier gives an incorrect prediction it is said to produce a leave-one-out error. The process is repeated for all the samples in the training set. The estimate of the generalization error is the number of leave-one-out errors divided by number of samples in the training set n. As the generalization error is very expensive to compute, the method uses $\xi\alpha$-estimator to evaluate the upper bound on the number of leave-one-out errors instead of computing it brute force.

$\xi\alpha$-estimators are computed from the two arguments, $\vec{\xi}$ – the vector of training losses at the solution of the primal *SVM* training problem and $\vec{\alpha}$ – the solution of the dual *SVM* problem. Both vectors are available after the training of *SVM*.

The window adjustment algorithm works as follows. The window of samples is divided time-wise in batches of fixed size m. At batch t, the most recent batch, the method tries various window sizes (see Equation 16) by training a *SVM* for each resulting window and computing the $\xi\alpha$-estimates bases on the result of training. The method chooses the window size which minimizes the $\xi\alpha$-estimate. Algorithm summarizes the method [35].

$$\begin{aligned} &\vec{z}_{(t,1)}, ..., \vec{z}_{(t,m)} \\ &\vec{z}_{(t-1,1)}, ..., \vec{z}_{(t-1,m)}, \vec{z}_{(t,1)}, ..., \vec{z}_{(t,m)} \\ &\vec{z}_{(t-2,1)}, ..., \vec{z}_{(t-2,m)}, \vec{z}_{(t-1,1)}, ..., \vec{z}_{(t-1,m)}, \vec{z}_{(t,1)}, ..., \vec{z}_{(t,m)} \end{aligned} \tag{16}$$

The main drawback of the method is the expense of computation. With each new batch of data the method builds the *SVM* classifier k times, where k is the current amount of data batches in the window. Another drawback is the fixed size of the batch, m. This puts a strong assumption that the drift cannot occur more often than every m samples which is not realistic and causes the delay in drift detection.

The authors do not mention is the algorithm can be effectively used in distributed environment. One possible way this could be done is to parallelize the computation of $\xi\alpha$-estimators. This would thought require each parallel worker to have an access to each batch of the data. This could be achieved by either some kind of shared storage or by broadcasting the data to each parallel worker.

Algorithm 5 *SVM*-based Concept Drift Detection Method

1: **for all** $h \in \{0, ..., t-1\}$ **do** ▷ for each batch h
2: train *SVM* on examples $\vec{z}_{(t-h,1)}, ..., \vec{z}_{(t,m)}$
3: compute $\xi\alpha$-estimate on examples $\vec{z}_{(t,1)}, ..., \vec{z}_{(t,m)}$
4: choose the window size which minimizes $\xi\alpha$-estimate

5.3. Ensemble Methods

This section presents an overview of ensemble methods. These methods use a set of classifiers to make a final prediction on the sample. The classifiers are either trained on different subsets of the data or with different parameters and are said to often outperform the single-classifier approaches [54].

SEA

The *Streaming Ensemble Algorithm (SEA)* [52] was one of the first ensemble algorithms to handle concept drift in the streaming data with classifier ensembles [28]. According to classification in Section 5.1.5 this algorithm is adapting to concept drift with the help of structural changes to the ensemble. The algorithm works in batch-mode. The ensemble has a fixed size. The ensemble member C_t is built on each new chunk of data. This member is kept until the next chunk of data arrives. When a new chunk of data arrives the performance of all the old members of the ensemble is evaluated on this chunk. If

C_{t-1}, the member built on the previous step, outperforms one of the existing ensemble members – that member is substituted by C_{t-1}. The final classification decision is done by simple majority voting.

The main concern of the algorithm is to introduce the right quality measure for ensemble member. The classifiers diversity plays a very significant role in the ensemble performance. That is why the accuracy of a classifier as a quality measure would not work because it would produce a very homogeneous ensemble. Instead, authors proposed another method to determine which classifiers should be favored. They retain the classifiers which correctly classified the points on which the ensemble was nearly undefined. This also prevents the susceptibility to noise which would be present were the authors to favor the classifiers which make a correct prediction on the points incorrectly classified by the majority of the members in the ensemble. More specifically, the authors define the following percentages used to compute the the quality measure for the ensemble member T:

P_1 = percentage of top vote-getter
P_2 = percentage of second-highest vote-getter
P_C = percentage for the correct class
P_T = percentage for the prediction of the new tree T

If both the ensemble and the member T correctly classified the sample, the quality of T is increased by $1 - |P_1 - P_2|$. That is if the member T made a correct prediction when the vote was close, it gets a bigger increase in its quality measure. If the member T was correct but the whole ensemble made a wrong prediction its quality measure is increased by $1 - |P_1 - P_C|$. Finally, if the prediction of T was incorrect its quality measure is decreased by $1 - |P_C - P_T|$.

The method adjusts to concept drift by pruning the old members of the ensemble when they loose their accuracy. There is a trade-off between the ensemble's fast reaction to concept drift and the the noise sensitivity which depends on the block size. If the block size is small the ensemble is more reactive to the drift but it is also more prone to learn the noise in the data. On the other hand, when the block size is big the ensemble will require more data to substitute the outdated members of the ensemble. Another parameter that influences the ability of the ensemble to react to the change fast is the size of the ensemble. The bigger the ensemble is – the slower the change adaptation will be. On the other hand, during the stable data distribution periods the bigger ensemble is likely to perform better. The algorithm is also computationally expensive, as each chunk of data is used to evaluate the performance of each member in the ensemble.

With the presence of multiple learners which can run in parallel the method stands out as a good candidate for implementation in the distributed setting. The method has two

points of synchronization – the process of choosing a candidate member for eviction and the voting point.

AWE

The method proposed in [54] is similar to *SEA*. The *Accuracy Weighted Ensemble (AWE)* algorithm also learns a new classifier for each new data chunk and substitutes the low-performing members with the high-performing ones. The main difference of the method is in the way the final classification decision of the ensemble is derived. The method also uses a different indicator of the member's efficiency.

More specifically, each member C_i has a weigh associated with it which is reversely proportional to the expected error of C_i on the new chunk of data S_t. Let us assume that S_t consists of m samples in form of (x, y). The classification error of the classifier C_i with respect to chunk S_t is $1 - f_y^i(x)$, where $f_y^i(x)$ is a probability given by C_i that x has a label y. The mean square error of C_i is computed as follows:

$$MSE_i = \frac{1}{m} \sum_{j=1}^{m} \left(1 - f_{y_j}^i(x_j)\right)^2 \tag{17}$$

Let Y be the set of all possible values of label y. The MSE_i of the classifier which predicts randomly based on the class distributions is computed as follows:

$$MSE_r = \sum_{i=1}^{|Y|} p(y_i)(1 - p(y_i))^2 \tag{18}$$

The final weight that is assigned to the classifier is the difference between the MSE_i of this classifier and the MSE of random classifier:

$$w_i = MSE_r - MSE_i \tag{19}$$

The final classification decision is computed as weighted majority of the ensemble. At each step the classifier with the lowest weight is removed from the ensemble.

It is worth mentioning that both *SEA* and *AWE* are able to handle reoccurring concepts as they can utilize the members trained on old data. The defining factor is the chunk size though.

DWM

Dynamic Weighted Majority (DWM) method copes with the concept drift by dynamically adding and removing members from the ensemble in response to fluctuations in the accuracy of the ensemble and its individual members [36]. The algorithm maintains a set of experts each of which is assigned a weight w_i. These weights reflect the performance of each individual member and are used to produce a final weighted prediction.

Algorithm 6 Dynamic Weighted Majority

Input:
β: factor for decreasing weights, $0 \leq \beta \leq 1$
θ: threshold for deleting experts
p: period between member removal, creation and weight update

1: $m = 1$ ▷ number of members in ensemble
2: $e_m = createNewMember()$ ▷ create first member
3: $w_m = 1$
4: **for all** (x_t, y_t) **do** ▷ for each training sample
5: **for all** e_i **do** ▷ for each ensemble member
6: $\lambda = e_i.classify(x_t)$
7: **if** $\lambda \neq y_t$ **and** $t \ mod \ p = 0$ **then**
8: $w_i = \beta w_i$
9: $\sigma_\lambda = \sigma_\lambda + w_i$ ▷ sum of weighted predictions
10: $\Lambda = argmax_i \sigma_i$ ▷ make weighed prediction
11: **if** $t \ mod \ p = 0$ **then**
12: $w = normalizeWeights(w)$ ▷ normalize the vector of weights
13: $\{e, w\} = removeMembers(\{e, w\}, \theta)$ ▷ remove low-performing members
14: **if** $\Lambda \neq y_t$ **then**
15: $m = m + 1$
16: $e_m = createNewMember()$
17: $w_m = 1$
18: train all members with (x_t, y_t)

The algorithm starts with one learner. Each new training sample is used to train every classifier in the ensemble. The new learner is added when the ensemble classifies a sample incorrectly. In order to prevent the algorithm from creating too many members when the concept drift happens or when there is a noise in the data, the authors introduce a parameter p which defines how often *DWM* creates and removes the members. Each time the individual member misclassifies the sample its weight is decreased using the multiplicative factor β. The weights of the members is then normalized in order to prevent the new members to dominate the predictions. When the weight of a members drops beyond parameter θ, it is removed.

In order for an ensemble algorithm to be effective it needs to introduce some diversity to its members. *DWM* does this with the help of varying the data each member is trained on as each member starts at different time. *DWM* copes with concept drift by substituting the members with low performance. The problem of this algorithm is the unlimited number of members which depends only on the underlying data. This does not allow to predict the memory usage of the algorithm. Another issue is the parameter p which as in many algorithms controls a trade-off between fast adaptation to the drift and resistance to noise.

ASHT

Adaptive-Size Hoeffding Tree (ASHT) is an ensemble of Hoeffding Trees [17]. The method is based on the intuition that smaller trees adapt to the changes more quickly than the big ones, while big ones perform better during the stable periods. The algorithm keeps an ensemble of Hoeffding Trees of different sizes, which guarantees a good diversity of the ensemble. The size of the *nth* member of the ensemble is equal to twice the size of the *(n-1)th*. When the number of split nodes in the tree surpasses the maximum allowed value – it deletes some nodes to reduce its size. Also, each tree is assigned a weight which is proportional to the inverse of the square of its error. It monitors the error with an *EWMA* chart.

The algorithm keeps an ensemble of highly diverse members which has a good impact on its accuracy and concept drift adaptation ability. Experiments show that it outperforms the Hoeffding Option Tree on the number of datasets. The drawback of the algorithm is the high memory and processing time requirements.

Ensembles of Restricted Hoeffding Trees

The method proposed in [13] builds an ensemble of Hoeffding Trees each of which is limited to a small subset of attributes. The idea of the method is to enumerate all possible attribute subsets of a given size k. Each tree models the interactions between the attributes in its subset. As the results of the trees are not of equal importance the method uses the predictions of each tree to train the set of perceptron classifiers. The perceptron is built for each label value using Hoeffding trees' class probability estimates as the input data.

The perceptrons are built using stochastic gradient descent. The weights of the perceptron are updated after each training sample. An important aspect of stochastic gradient descent is the learning rate. The traditional approach to setting the learning rate is to decrease it as the amount of samples increases. The authors use the following equation to set the learning rate, where m is the number of attributes and n is the number of training samples we have seen so far:

$$\alpha = \frac{2}{2 + m + n} \tag{20}$$

The problem with this learning rate is that is assumes that the training data is identically distributed. As the input data of the perceptron in this case is the prediction of the classification tree, this is not the case. First, the accuracy of the tree tends to improve over the time and second, it might as well deteriorate as the concept drift happens. In these two cases we would like the learning rate to increase again in order to capture the change. The authors tackle this problem by adding an external concept drift detector to monitor the accuracy of each individual classifier and reset the learning rate when the drift is detected. It is done by setting the value of n to zero. The very same explicit drift detectors are used to reset the trees after the concept drift happens.

The approach is interesting in that it results in an ensemble of great diversity. The problem of the algorithm is its computational complexity and huge memory requirements. Given m attributes the algorithm generates all possible subsets of size k. This results in $\binom{m}{k}$ subsets. Consequently, the method is not applicable for high-dimensional problems. However, the value of $k = 2$ proved to be very practical in many experiments. This is due to the fact that many practical classification problems often appear to exhibit only very low-dimensional interactions [13]. More interestingly, for artificial datasets the large values of k (like $m - 2$) turned out to be beneficial, while for real ones - small values of k(like 2) worked better. As $\binom{m}{k} == \binom{m}{m-k}$ the method will produce the same amount of trees for both cases. But even in this case the drawback of the method is that although some trees prove to be useless in final decision and converge to a very low weight, they still have to be maintained by the algorithm.

DDD

In [40, 41, 42] the authors present a thorough study on the influence of ensemble diversity on tackling various types of drift. The method they present makes use of this study by combining several ensembles with different levels of diversity to tackle various types of concept drift and noise in the data. In their experiments they use the online bagging technique introduced originally in [45].

The online version of bagging aims at simulating the batch version of bagging [19] on potentially infinite data streams. The method proved to be very effective in improving generalization performance in comparison with single-learner models [10]. The original method constructs an ensemble of m learners each trained on the subset consisting of n samples drawn from the original training dataset of size n at random, but with replacement [19]. The number of times each individual sample will be used for the training of the particular learner, K, follows the binomial distribution. In the online version of

bagging authors exploit the fact that this binomial distribution can be approximated by the $Poisson\,(\lambda)$ distribution with $\lambda = 1$, given the infinite size of training dataset:

$$K \sim \frac{\exp\,(-1)}{k!} \tag{21}$$

The authors came to the conclusion that by varying parameter λ of the $Poisson\,(\lambda)$ distribution they can vary the diversity of the ensemble. The lower the value of λ is, the smaller values it will produce resulting in ensemble members being trained on very distinct subsets on data. By varying the diversity of the ensemble in this way they have studied how diversity influences the adaptation of the ensemble to different types of concept drift. The concept drift is classified by severity (high and low) and speed (high and low, corresponding to abrupt and gradual drift). Initially two ensembles are being trained – of low and high diversity. An external concept drift detector is used to detect the point of change. After the change two new ensembles are created but the two old ones continue to be trained. Thus, the study compares four strategies of concept drift adaptation.

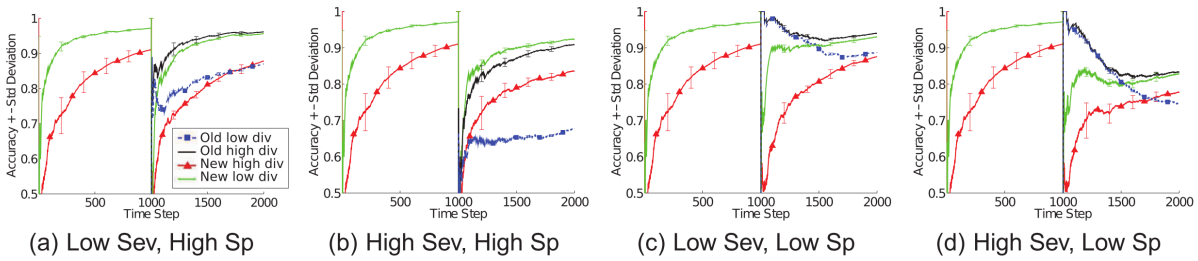


Figure 5: Diversity influence in presence of abrupt and gradual concept drift [41]

The results are shown on the Figure 5 [41]. The drift happens at time point 1000. Before the drift the low diversity ensemble manages to capture the concept faster and achieves better accuracy. In case of low severity drift(Figure 5(a)) the old high diversity ensemble manages to learn the new concept faster than the others. The old data still provides some useful insight here. But in case of high severity(Figure 5(b)) old data is harmful and the new low diversity ensemble manages to learn the new concept faster than the old ones.

The situation looks different in case of gradual drift. In case of both low and high severity(Figure 5(c, d)) shortly after the drift the old ensembles perform better. It can be explained by the fact that the concept very gradually diverts from the one they have learned. Longer after the drift old high and the new low diversity ensembles achieve the best performance.

Taking this study into account the authors proposed a new method *Diversity for Dealing with Drifts (DDD)* [41]. Algorithm 7 provides the detailed description. *DDD* uses external drift detector and operates in two modes: before and after the drift. Before the drift two ensembles are trained: low and high diversity ones. The low diversity ensemble is used for classification and drift detection. When the drift is detected the new low and high diversity ensembles are created. The old ensembles are still maintained though. The old high diversity ensemble starts to learn in low diversity mode in order to capture the new concept faster. The final classification in this mode is a weighted majority of three ensembles: old low and high diversity and new low diversity ones. The new high diversity ensemble is not considered because it will likely be slow in capturing the new concept. The weights are proportional to the accuracies obtained since the point of drift. The accuracies of new low diversity ensemble and old high diversity trained in low diversity mode are constantly monitored in order to determine which of the two manages to capture the new concept faster. One of the two is finally chosen and the algorithm switches to stable mode again.

Algorithm 7 Diversity for Dealing with Drifts

Input:
W: multiplier weight for the old low diversity ensemble
p_l: parameter for learning in low diversity mode
p_h: parameter for learning in high diversity mode

$mode = beforeDrift$
Initialize h_{nl} ▷ new low diversity
Initialize h_{nh} ▷ new high diversity
$acc_{ol} = acc_{oh} = acc_{nl} = acc_{nh} = 0$ ▷ accuracies
$std_{ol} = std_{oh} = std_{nl} = std_{nh} = 0$ ▷ standard deviations
for all x_t **do**
    if mode == before drift **then**
        $prediction = h_{nl}(x_t)$
    else
        $sum_{acc} = acc_{nl} + acc_{ol} * W + acc_{oh}$
        $w_{nl} = acc_{nl}/sum_{acc}$
        $w_{ol} = acc_{ol} * W/sum_{acc}$
        $w_{oh} = acc_{oh}/sum_{acc}$
        $prediction = weightedMajority(h_{nl}(x_t), h_{ol}(x_t), h_{oh}(x_t), w_{nl}, w_{ol}, w_{oh})$
        $update(acc_{nl}, std_{nl}, acc_{ol}, std_{ol}, acc_{oh}, std_{oh}, x_t)$
    $drift = detectDrift(h_{nl}, x_t)$

17: **if** $drift == true$ **then**
18: **if** mode == before drift **or** (mode == after drift **and** $acc_{nl} > acc_{oh}$) **then**
19: $h_{ol} = h_{nl}$
20: **else**
21: $h_{ol} = h_{oh}$
22: $h_{oh} = h_{nh}$
23: initialize h_{nl}
24: initialize h_{nh}
25: $acc_{ol} = acc_{oh} = acc_{nl} = acc_{nh} = 0$
26: $std_{ol} = std_{oh} = std_{nl} = std_{nh} = 0$
27: mode = after drift
28: **if** mode == after drift **then**
29: **if** $acc_{nl} > acc_{oh}$ and $acc_{nl} > acc_{ol}$ **then**
30: mode = before drift
31: **else**
32: **if** $acc_{oh} - std_{oh} > acc_{nl} + std_{nl}$ **and** $acc_{oh} - std_{oh} > acc_{ol} + std_{ol}$ **then**
33: $h_{nl} = h_{oh}$
34: $acc_{nl} = acc_{oh}$
35: mode = before drift
36: $learn\,(h_{nl}, x_t, p_l)$
37: $learn\,(h_{nh}, x_t, p_h)$
38: **if** $mode == afterDrift$ **then**
39: $learn\,(h_{ol}, x_t, p_l)$
40: $learn\,(h_{oh}, x_t, p_l)$
41: **Output:** *prediction*

The algorithm proves to be able to handle both abrupt and gradual change and is very noise resistant due to the fact that the old ensemble is not discarded after the drift is detected but depending on the accuracy either restored or substituted by highly accurate new one. The main drawback of the algorithm is the high complexity and memory requirement. Moreover, during the concept drift periods it requires twice as much resources as in the stable one.

5.4. Further Problems with Concept Drift Handling

This section details some issues which may affect the drift adaptation and presents several methods to handle these issues.

5.4.1. Class Imbalance Problem

Class imbalance problem is characterized by unequal number of training samples of different classes. Such an imbalanced class distribution is often seen in data stream problems, like fraud or intrusion detection in computer networks or medical diagnosis. Detecting concept drift in imbalanced data streams is a more difficult task than in balanced ones. When the drift involves the minority class the recall of it suffers a significant drop while the overall accuracy does not reflect this. This is why the overall accuracy can be an insufficient drift indicator if it affects the minority class [56].

DDM-OCI

The authors propose a new method called *Drift Detection Method for Online Class Imbalance (DDM-OCI)* which is based on existing *DDM* method [56] described in Section 5.2.1. The main difference of the method is that it tracks the recall in the minority class instead of the overall accuracy. The significant drop in the decayed recall of minority class may indicate the drift in this class. For detecting which class is currently the minority one, the class imbalance detector is proposed in [57].

For the class c_k the current recall R_k is defined by n_k^+/n_k, where n_k^+ denotes the number of correctly classified examples with true label c_k and n_k is the total number of examples of true label c_k received so far. At time stamp t the recall will be updated by $R_k^{(t)} = \eta' R_k^{(t-1)} + (1-\eta')\,[x \leftarrow c_k]$ if the sample x is of class c_k. The time decay factor $\eta' \in (0 < \eta' < 1)$ is supposed to emphasize the performance in current moment. The value of $[x \leftarrow c_k]$ is equal to 1 if sample x is correctly classified and 0 otherwise.

The method proves to detect the concept drift in minority class faster than *DDM*, but is also more prone to false alarms. Also, the drift in minority class can as well happen without affecting the minority class recall. Such drifts will not be detected by *DDM-OCI* [55].

LFR

The *Linear Four Rates (LFR)* algorithm maintains the probability matrix CP where $CP\,[1,1]$, $CP\,[0,0]$, $CP\,[1,0]$, $CP\,[0,1]$ are the underlying percentages of true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN) respectively. It tracks four characteristic rates (True Positive Rate, True Negative Rate, Positive Predicted Value, Negative Predicted Value) of the classifier. The rates are computed as follows: $P_{tpr} = TP/\,(TP+FN)$, $P_{tnr} = TN/\,(TN+FP)$, $P_{ppv} = TP/\,(FP+TP)$, $P_{npv} = TN/\,(TN+FN)$.

The algorithm uses modified rates $R_*^{(t)}$ as the test statistics for $P_*^{(t)}$. The rate $R_*^{(t)}$ is computed as $R_*^{(t)} = \eta_* R_*^{(t-1)} + (1 - \eta_*)\, 1_{\{y_t = \hat{y}_t\}}$ where η_* is the time decay factor and $1_{\{y_t = \hat{y}_t\}}$ is equal to 1 if $y_t = \hat{y}_t$ and 0 otherwise. The algorithm works in warning-alarm fashion and requires two parameters: warning significance level (δ_*) and detection significance level (ϵ_*) to be set for each rate.

In order to derive a reliable running confidence interval for $R_*^{(t)}$ the authors introduce a method to obtain a reasonable empirical distribution function using Monte Carlo simulation for given factor η_*, rate P_* and number of observed values N_*.

The method proved to be less prone to false alarms than *DDM-OCI* and in many cases is able to detect the drift earlier. The original method is applicable only to the binary classification problems. Although by denoting the classes as minority and non-minority ones the method can be perfectly applicable to multi-class classification problems.

6. Machine Learning in Apache Flink Streaming

This section introduces the basic concepts about streaming dataflow engine Apache Flink [2] which are important for explanation of the scalable algorithms implementation. Section 6.1 presents basic concepts of Apache Flink: data stream, transformation and stream partitioning. Section 6.2 introduces another important concept for implementation of Machine Learning algorithms – state in Flink. Finally, Section 6.3 introduces the building blocks for our Machine Learning pipelines.

6.1. Stream Transformations and Partitioning

The streaming pipeline in Apache Flink Streaming consists of multiple consequent data transformations expressed by a flow of operators. Each operator may have multiple subtasks (instances) which are processing samples in parallel. The number of subtasks is defined by the parallelism of the operator. The stream produced by one of the subtasks of the arbitrary operator then is either forwarded to the subtask of the next operator or some kind of repartitioning takes place [2].

The main transformations in Apache Flink Streaming are :

Map/FlatMap

The transformation takes one element as an input and produces one(*Map*) or multiple(*FlatMap*) elements as an output. The types of input and output elements might differ.

KeyBy

This transformation is used to partition the stream into several disjoint streams based on the given key.

Window

Window transformation is used to collect the samples into batches according to some characteristic, like time, count or some custom condition.

Window Apply Function

The Window function is used to process the samples in the batch collected by the Window transformation.

It is important to mention that due to the parallel processing of the samples by the subtasks of the operator the order in which the samples are produced by these subtasks might not correspond to the order in which they came to the system. Flink introduces the mechanism to reorder the samples based on the injection or event time but this entails semi-batch data processing. The reason is that in order to reorder the data according to event time it needs to be collected into windows first.

Another important note about Flink is lack of explicit communication mechanism between operators and subtasks of one operator. Although it is an obvious design decision for Apache Flink, it makes the implementation of Machine Learning streaming pipelines quite difficult.

6.2. State in Apache Flink

Currently Flink supports two types of state. The *non-partitioned* state is inherent in each instance(subtask) of operator. It means that this state is based solely on the input that particular instance receives. The second type is *partitioned* state. This kind of state is based on the particular partitioning of the input stream. For example, if a stream was partitioned by the key, then there will be a separate state based on the samples of each key.

6.3. Machine Learning Pipelines: Building Blocks

This section introduces the main components which are combined in order to build a Machine Learning system.

6.3.1. Generic Learning Model

The general assumption we make about the learning process is presented in Figure 6. We assume that our data source produces two types of items: labeled and unlabeled. The labeled samples need to be integrated into the model. The unlabeled samples are to be classified. The output consists only of unlabeled samples classified by the algorithm.

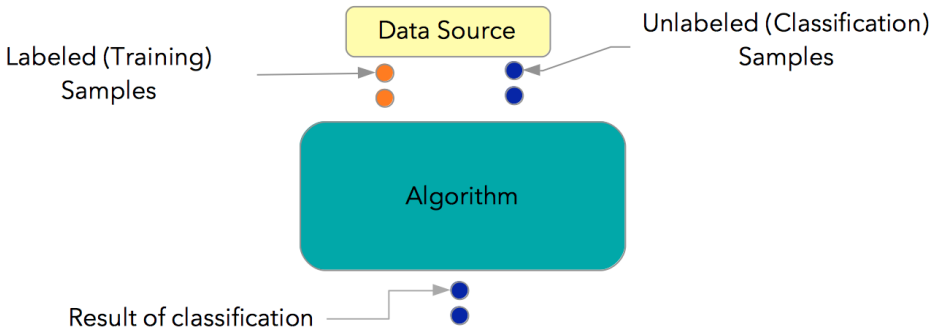


Figure 6: Generic Learning Model

6.3.2. Base Classifier

The main component of the Machine Learning system is the base classifier. It is the component which can access samples for both learning and classification. The training samples are incorporated into existing model which is then used to classify the samples without the label. The classifier itself can be the implementation of an arbitrary online algorithm.

As the base classifier we used our implementation of Online Naive Bayes algorithm.

6.3.3. Change Detector

This component is responsible for detecting concept drift in the flow of training samples and notifying the system which in turn has to accommodate the model to the new concept. The change detector accepts the pair <*real label, predicted label*> and can be an implementation of an arbitrary change detection algorithm.

6.3.4. Performance Tracker

This component handles the same input as the change detector and is used to track the performance of the learning system. The decrease in the performance of the algorithm can signalize the concept drift.

7. Implementation Details

This section introduces the implementation of two algorithms in Apache Flink Streaming. Section 7.1 presents the implementation of the algorithm which we used as a baseline for evaluation. Section 7.2 introduces the details of implementation of the bagging algorithm in Apache Flink.

7.1. Baseline Algorithm

Considering the absence of a globally shared state in Apache Flink Streaming the base approach to achieve scalability is to produce multiple instances of the same learning model and use them for classification. Figure 7 depicts the dataflow for this algorithm. In order to build multiple instances of the same learner we use the same training data for each of the learners.

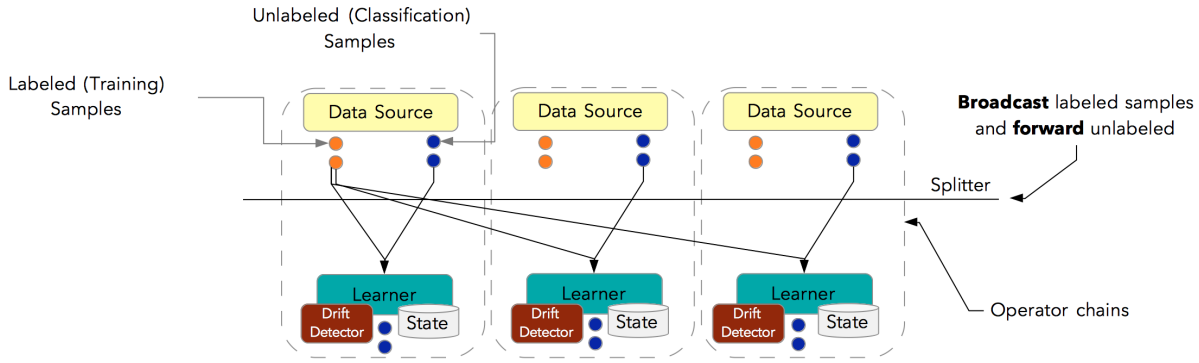


Figure 7: Baseline Algorithm: each learner is trained on the same data

We implemented this approach by broadcasting the training samples to each partition. The unlabeled samples, on the other hand, are distributed among the learners for classification. As the input of each learner is a non-partitioned stream, each learner holds its own non-partitioned state.

Each learner is equipped with its own change detector. The easiest way to adapt to the concept drift would be to reset the model whenever the drift is detected. But this approach results in the huge loss of accuracy during the recovery periods. Basically, whenever the model is reset, it needs some number of new training samples in order to recover. But if the model has to classify the samples during this period, the accuracy will drop significantly. Moreover, the noisy data may result in false drift detections. So, there

is a need for a method which is more stable during recovery periods and resistant to false concept drift detections.

Algorithm 8 Reactive Algorithm

```
1: Input:
2: θ: threshold for substitution of stable learner by reactive one
3: w: threshold for dropping the reactive learner
4: Initialize stable learner S, concept drift detector DT, reactive learner R
5: state = STABLE
6: c_R = 0                          ▷ The counter for substitution of stable learner
7: c_warn = 0                       ▷ The counter for dropping the reactive learner
8: for all (x_t, y_t) do
9:     y_S = S.classify(x_t)
10:    if state = WARNING then
11:        c_warn = c_warn + 1
12:        y_R = R.classify(x_t)
13:        if y_S ≠ y_t ∧ y_R = y_t then
14:            c_R = c_R + 1
15:            if c_R = θ then          ▷ Reactive learner outperforms the stable one
16:                S = R
17:                state = STABLE
18:                c_R = 0
19:                c_warn = 0
20:            else if c_warn = w then  ▷ Stable learner outperforms the reactive one
21:                R.reset()
22:                state = STABLE
23:                c_R = 0
24:                c_warn = 0
25:    DT.input(y_S, y_t)
26:    if DT.driftDetected then
27:        state = WARNING
28:    S.train(x_t, y_t)
29:    if state = WARNING then
30:        R.train(x_t, y_t)
```

One of such methods is the Paired Learners method described in Section 5.2.1. But, as was already mentioned, the disadvantage of this method is that it needs to either work with a classifier that can "un-learn" the samples or a reactive model has to be rebuilt after each incoming sample. This would induce a considerable slowdown of our learner. We introduced a modification of the algorithm which takes an explicit concept drift detector into account. Algorithm 8 introduces the details.

Whenever the concept drift is detected we do not reset the model but instead start building the new model. The old stable model is still used for classification and the new training samples are integrated into both of the models: stable and reactive. Starting from this point we track the performance of the reactive model and if it outperforms the stable model, we substitute it with the reactive one. This helps to avoid rapid decrease in performance whenever the drift is detected. The algorithm has two parameters: θ – the number of times the reactive model has to outperform the stable one in order to replace it and w – the maximum window length during which we will track the performance of both models. In this way we will also avoid reseting the model on the false drift detection as the reactive model built on new samples will not outperform the old one unless there is a real drift.

7.2. Bagging Algorithm

The ensemble algorithm we chose for implementation is online bagging described in Section 5.3. The reasons for this choice are mainly connected with the concepts of Apache Flink Streaming engine. Specifically, Flink pipelines samples as soon as they arrive, which means that an algorithm with the single-sample processing mode can benefit more from Flink streaming. As data is not collected in batches, but pipelined immediately, each training sample will be integrated into a model as soon as it is received. The second reason is the constant number of learners in the algorithm. This will induce the constant memory usage. Moreover, each learner receives nearly the same load of labeled and unlabeled input samples which results in an equal load per learner. Another important reason is the minimal communication between the learners imposed by the algorithm. As was mentioned, Flink does not provide an explicit way of communication between operators and subtasks of these operators. By minimizing the need for such communication the algorithm will benefit in terms of data latency. Final considerations have to do with the generic nature of the algorithm: it can work with arbitrary base classifier and change detector. The amount of samples each learner is trained with, thus the ensemble diversity, can be regulated with the Poisson λ parameter.

Figure 8 presents the data-flow diagram of the algorithm. The first stage for each sample produced by data source is *Data Distributor*. In case of parallel *Data Source* and equal parallelism of *Data Source* and *Data Distributor* operators the implementation can benefit from operator chaining. This allows to pipeline all the samples produced by specific *Data Source* to a *Data Distributor* on the same node without the need for stream repartitioning and network communication. The *Data Distributor* is implemented as a *FlatMap* transformation. For the labeled samples the *Data Distributor* uses the Poisson distribution to compute which of the learners will receive this sample for training and the weight of this sample for each particular learner. Unlabeled samples will need to be sent to each

learner as each of them needs to vote on it. This operator produces the records in format *<learnerID, item, weight>*, where *learnerID* is used as a key for stream partitioning at the next step and *weight* is assigned according to Poisson distribution. More details on the implementation can be found in the Appendix A.1.

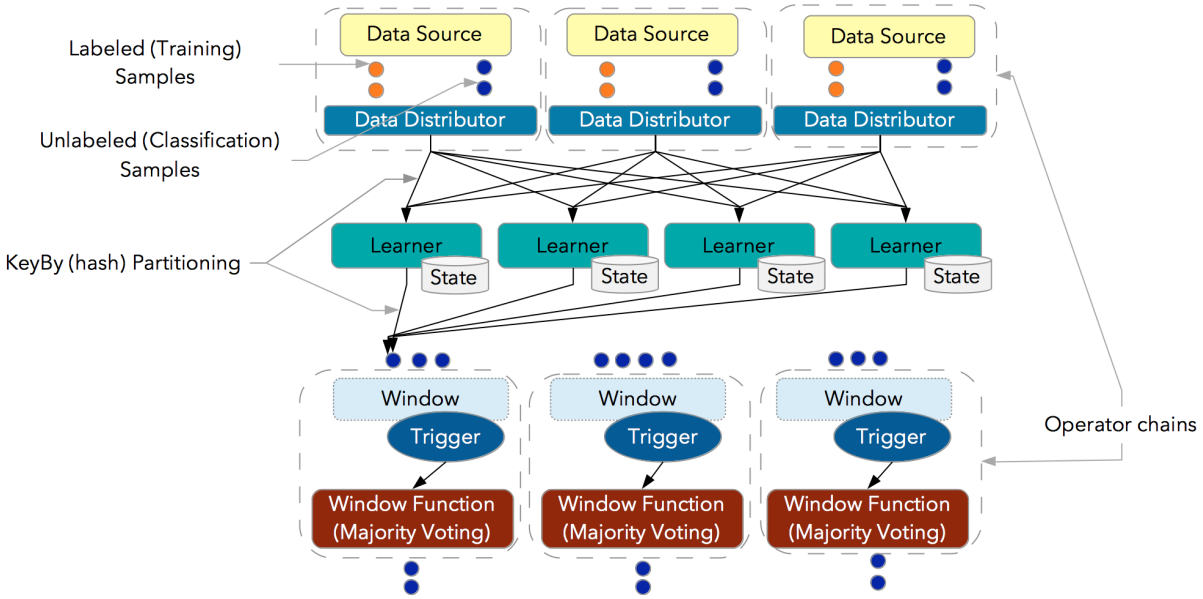


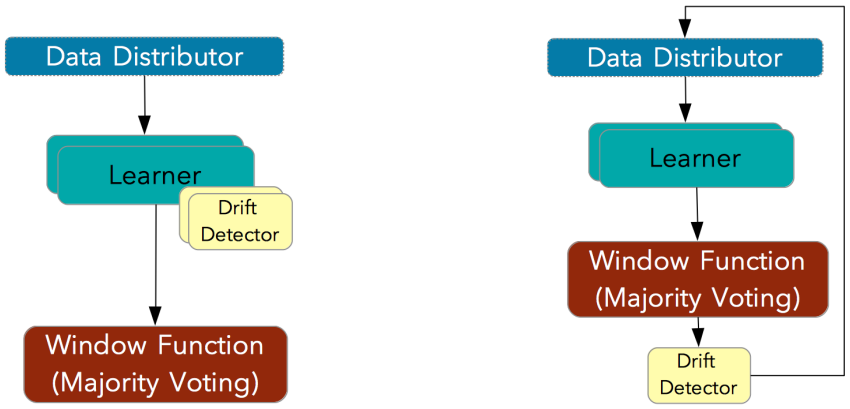
Figure 8: Bagging Algorithm: each learner is trained on different subset of data

Flink uses hash partitioning by the given key in order to deliver the samples assigned to each learner. The learner is also implemented as a *FlatMap* transformation. Each learner has its partitioned state where the model is kept. Labeled samples are integrated into the model, unlabeled are classified and propagated to the output. Each learner outputs the records in the format *<itemID, item, label>*, where the *itemID* is a unique id assigned to each item and label is the classification result. The stream is repartitioned one more time, this time using *itemID* as a key. This is necessary to enable voting on the final label of the sample. For this we need to collect all votes in one operator. The learner code can be found in Appendix A.2.

The voting takes place in *WindowFunction*. The current implementation performs simple majority voting and outputs the final label for each sample. The code of this *WindowFunction* can be found in Appendix A.3.

As the algorithm is not implicitly adaptive it needs the explicit change detector in order to handle concept drift in the stream. The complicated question for the current implementation is how to integrate the explicit change detector.

The two main components of the algorithm, the learner and the change detector need some type of communication, as whenever the concept drift is detected every learner needs to be notified.



(a) Local Drift Detector (b) Global Drift Detector

Figure 9: Bagging Algorithm Implementation Options

To enable this in Apache Flink we developed two options how to enable the communication between these two components depicted in Figure 9. The first one is to unify these components into one operator in order for them to share the state. The second one is to use Flink iterations in order to deliver the concept drift signal up the pipeline to the learner. In the case of iteration feedback we need to introduce the changes to the way labeled instances are treated. Each learner needs to classify not only the unlabeled instances but also labeled ones and after the final decision on the labeled instance is made, the result must be sent to the drift detector. The detector tracks the accuracy of the algorithm and in case of the drift sends the message up the iteration loop to the *Data Distributor*. The *Data Distributor* then notifies the learners. The learner in this case must be implemented as a *CoFlatMap* function to handle two different types of input – the samples and the messages from detector. The option with the local detector for each of the learners does not require the change in the introduced data flow.

A potential disadvantage of the global drift detector is the dependence of the delay in signal delivery on the network speed and the load of the nodes. No guarantees can be given on the average drift adaptation time. Moreover, the fact that all the samples need to be classified and voted on can potentially harm the throughput.

On the other hand, the fact that the local drift detector handles exclusively the labeled samples received by the particular learner makes it less stable and prone to false drift

detections. Moreover, when one of the learners detects the drift there is no way for it to notify the others. And unless more than 50% of the learners adapt a model to the new concept the global vote will be dominated by the old concept.

Considering these advantages and disadvantages the version we chose for implementation and evaluation is the Local Drift Detector. In order to make the system more stable to the false drift detections we use the reactive algorithm introduced in Algorithm 8 as the base learner.

8. Evaluation

This section introduces the performance evaluation of the implemented algorithms. Section 8.1 presents the overview of evaluation techniques for streaming problems. Section 8.2 presents the method we used for evaluation – k-fold bagging and details of its implementation. Section 8.3 presents the evaluation on artificial datasets. Artificial datasets are good for evaluation of concept drift adaptation algorithms as they allow us to control the points where the drifts occur. This section investigates how different configuration parameters of both algorithms influence their performance. Section 8.4 presents the results of evaluation on some real datasets which are widely used for evaluation of concept drift detection and adaptation techniques. Section 8.5 compares the implemented algorithms in terms of their throughput and latency. Finally, Section 8.6 compares our bagging algorithm with several algorithms available in streaming machine learning framework Apache SAMOA [3].

8.1. Evaluating Accuracy of Streaming Algorithms

As stream learning algorithms are being actively studied now there is also a need in the effective evaluation of these algorithms. The most popular methods of evaluating online algorithms are predictive sequential evaluation and holdout an independent test set [27].

The holdout approach applies the current model to the test set at regular time intervals. The test set consists of the samples preserved especially for the evaluation task. The predictive sequential (*prequential*) [23] approach uses each incoming sample to first evaluate the current model and afterwards uses the sample to train (or update) the model. The prequential evaluator is pessimistic, this means that in the same conditions it will produce the higher error [27]. This can be explained by the fact that at each point of time the model is trained on the smaller or equal amount of samples than in the holdout set approach.

The basic prequential approach evaluates the performance of the algorithm over all the stream. This results in the inability of this algorithm to reflect the current performance in case of concept drift. As the model recovers from it by incorporating new samples, the prequential error still includes the low performance estimates during the concept drift period. This leads us to the conclusion that in order for this approach to reflect the current performance of the algorithm it needs to take the temporal aspect into account.

There are two forgetting strategies for prequential algorithm presented in [27]. The first approach keeps the fixed-size window over the newest samples to evaluate the performance

on. The second approach uses the fading factor α to downweight the old samples. These two approaches are similar in that each value of the fading factor corresponds to some value of the window size. The prequential error with the forgetting mechanism converges fast to the holdout error [29].

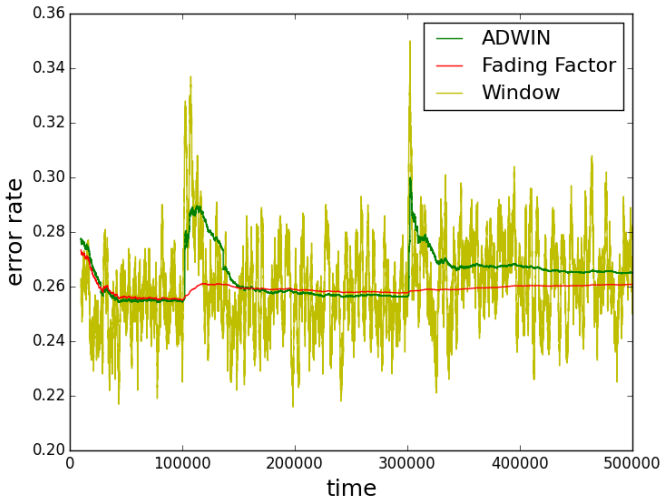


Figure 10: Comparison of Performance Estimation Strategies

The key difficulty of these two algorithms is selecting a proper value of the fading factor parameter or the window size. The small window size will result in a fair reflection of the accuracy during concept drift and fast adaptability of the performance evaluator. While the bigger window size will produce lower variance estimations during the stable phases [27].

In order to avoid this trade-off the authors of [12] propose using *ADWIN* algorithm described in Section 5.2.1 as the way to choose the appropriate size for the window. The method is parameter-free and adapts automatically to the current rate of change. This allows to use it as a fair estimation of the current error rate average in the stream. The comparison between the three discussed evaluation strategies is depicted in Figure 10. The dataset used for this evaluation is artificially generated with two drifts at time points 100000 and 300000. The window size used for this case was 1000, which is obviously too small as the variance of estimations during stable periods makes it hard to see when the real drifts happened. The fading factor α used is 0.975, which, apparently, is too high as the graph fails to depict the areas of concept drift. The *ADWIN* performance tracker shows the best result and does not require any tuning of parameters.

8.2. K-fold Bagging Validation

The issue of model validation becomes particularly tricky for streaming algorithms. The traditional techniques, like cross-validation, become infeasible in streaming setting due to the unlimited amount of data. The authors in [12] propose a new strategy for model validation in streaming context.

The method runs k instances of the same classifier. Each new sample is used in some of the instances as a training sample, while in others as a testing sample. The authors discuss three different strategies to determine the percentage of learners which receive the sample for testing and for training:

- send the sample to one learner for training and to others for testing;
- send the sample to one learner for testing and to others for learning;
- use Poisson(1) distribution for each sample to compute the percentage. This option will result in around two thirds of the learners receiving the sample for learning and the remaining 1/3 for testing;

The first approach trains each learner on the disjoint data streams. It might potentially underutilize the available data [12]. In contrast, the second approach will result in the learners trained on almost identical datasets. This makes the the third approach a perfect compromise.

8.2.1. Implementation in Apache Flink Streaming

We implemented the k-fold bagging evaluation technique in Apache Flink. The implementation is based on the data flow presented on the Figure 8.

For each incoming sample the *Data Distributor* will use the Poisson(1) distribution to compute which of the learners will receive this sample for training and which ones for testing. Each learner is equipped with its *Performance Tracker*. Taking into account the results we received when comparing different performance trackers (see Section 8.1) we chose the *ADWIN* tracker. When the learner receives the sample for training it trains the model with it. With each incoming test sample the learners classify the sample, update their performance and emit the result produced by performance tracker in the format *<itemID, performance>*.

The stream is be partitioned by *itemID*. The results for each particular item are collected by the trigger and the Window Function computes the average performance over all learners. The trigger used to collect the performance measurements from the learners needs to know the exact number of learners the sample has been sent to for testing. For this to work we implemented the custom trigger which receives this information with each sample and waits for the correct number of samples before triggering.

Apache Flink poses a different challenge when trying to implement any evaluation technique. When the parallelism of the operators in the data flow is greater than one, the order of resulting samples may not correspond to the order they came in. And in the case when we are interested in temporal aspect of the accuracy we need to have the temporal order of accuracy measurements. For this purpose we added the final operator to the data flow – the *CountWindowAll* operator. The parallelism of this iterator is 1. It receives the samples and assigns the timestamp to them in the order they were received at this operator. This allows us to plot the accuracy over time.

This evaluation technique was implemented for both baseline reactive algorithm and bagging with local change detector. These two algorithms are compared in the following sections.

8.3. Evaluation on Artificial Datasets

The benefit of evaluating the algorithms on artificially generated data streams is that we have control over the occurrence of concept drift, the type of the drift and the amount of noise in the data stream. In this work we used the artificial data generators provided in Apache SAMOA [3] framework. The data generators and parameters will be specified for each particular experiment.

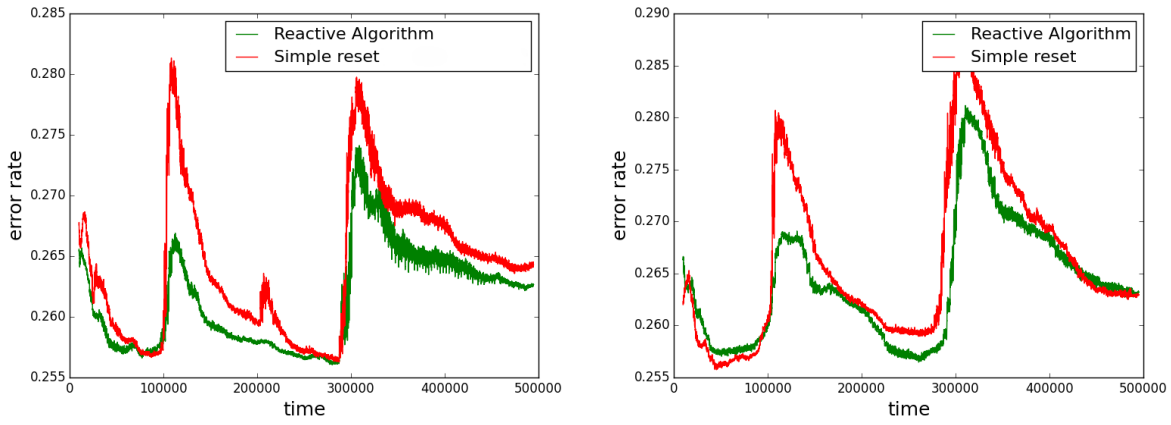
8.3.1. Baseline: Comparison of Reactive and Simple Reset Version

In Section 7.1 we proposed two options for the base learner algorithm: simple reset on drift detection and reactive algorithm. The first just resets the model on drift detection, whereas the second one replaces the stable model with the reactive one if outperformed. This section compares the performance of these two versions.

The data used for this experiment was produced with stream generator for SEA concepts functions proposed in [52]. The generator produces a three-dimensional feature vector for a two class decision problem. The three real-valued attributes are in range $[0, 10]$. Only the first two attributes are relevant for prediction. The class decision boundary is defined

as $x_1 + x_2 \leq \theta$, where x_1 and x_2 are the first two dimensions of the feature vector and θ is the decision boundary. To simulate the change of concept the value of θ is changed over time. We generate two drifts. The first function uses value $\theta = 8$, second $\theta = 9$, third $\theta = 7$. Note, that second drift is more severe than the first one.

Each experiment is performed for two types of drift – abrupt and gradual. We generated data with two consequent drifts – one at the point 100000 (samples) and another one – at 300000. Note, that in this work we refer to the id (serial number) of the sample as its time stamp. The point of the drift is in the center of the drift. Specifically, when the drift is at the point 100000, and the width of the drift is 20000, it means that it starts at a sample with id 90000 and ends at the one with id 110000. The noise percent in the data is 25%. The noise is generated by changing the label of the sample to the opposite one in the randomly chosen 25% of the samples . The width of the abrupt drift is 4 and the width of the gradual drift is 50000.



(a) Two abrupt drifts

(b) Two gradual drifts

Figure 11: Baseline Algorithm with simple reset base learner vs. with reactive base learner

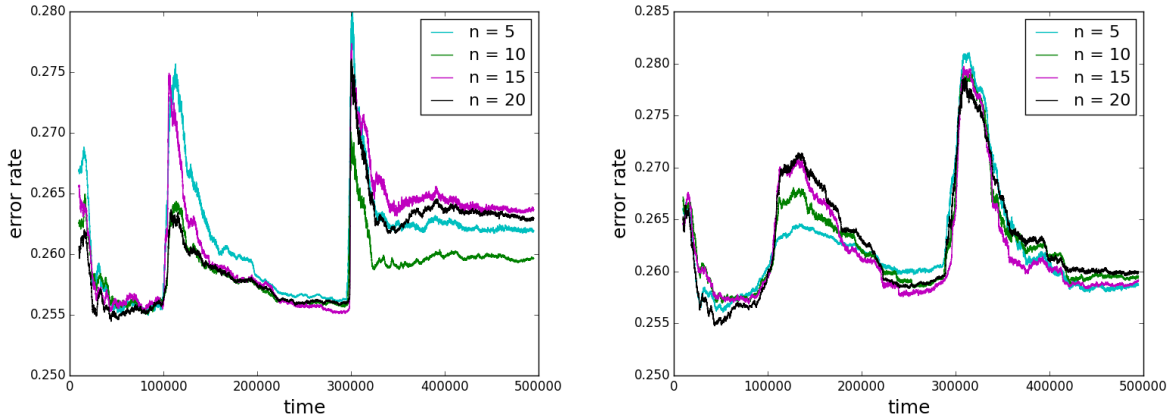
Figure 11 presents the result of evaluation of two base learners. The change detector used in this example is *ADWIN*. The reactive algorithm performs better in this setting in terms of classification error rate. The reason is that whenever the real concept drift takes place the drift detector tends to alarm drift detection multiple times which results in great instability if we reset the model each time. As can be seen in Figure 11a at the point 200000, a false drift detection caused the spike in the error rate. This was not the case for the algorithm with reactive base learner.

In the following sections we use the Reactive Algorithm as the base learner for both Baseline and Bagging Algorithms, as it clearly outperforms the plain reset algorithm.

8.3.2. Bagging Algorithm: Number of Learners

In this experiment we explore how the error rate of the bagging algorithm expained in Section 7.2 depends on the number of learners in the ensemble.

The setting (data source and types of drift) for the experiment is the same as in Section 8.3.1. The base learner we use for bagging algorithm is Reactive Learner with parameters $\theta = 10$ and $w = 200$. These values are optimal for this setting and were chosen after several experiments with this dataset.



(a) Two abrupt drifts

(b) Two gradual drifts

Figure 12: Bagging. Different number of learners

The result is presented on Figure 12. As we can see with the increase in the amount of learners the accuracy increases, but at the same time the algorithm becomes less capable to adapt to the concept drift. Before the (time)point of the first concept drift in both cases the algorithm with 20 learners was the most accurate. But in 3 out of 4 drifts in this experiment(both abrupt drifts and first gradual) the algorithm with 10 learners achieved higher accuracy after the drift than the one with 15 and 20 learners. In the remaining case (first abrupt drift, Figure 12a) it achieved the same accuracy as the one with 20, but better than the one with 15. During the first gradual drift (Figure 12b) the algorithm with 5 learners performed best in terms of classification error but failed to adapt as well as the others to the drift. This could be the reason why it performed worse during the second gradual drift.

As the algorithm achieved the best performance with n = 10, we use this number of learners for the following experiments with this dataset.

8.3.3. Bagging Algorithm: Poisson λ Parameter

This experiment investigates the dependence between the accuracy of our bagging algorithm and the Poisson distribution λ parameter. In the case of bagging this parameter defines which learners will receive a sample for training and assigns the weight to each sample. Figure 13 depicts the Poisson robability mass function for three parameters: $\lambda = 1$, $\lambda = 2$ and $\lambda = 3$. For each (discrete) value of k it shows the probability of this value to be produced by the corresponding function. For example, for $\lambda = 1$ the probability to produce $k = 0$ is higher than for $\lambda = 2$. This means that the higher values of this parameter are likely to produce higher weights for our samples.

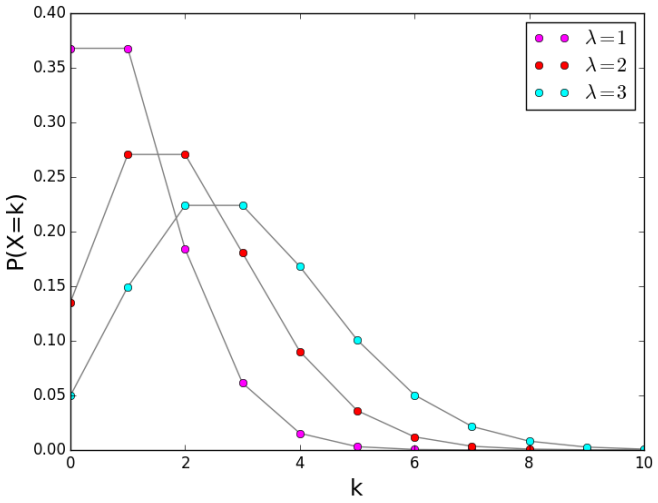
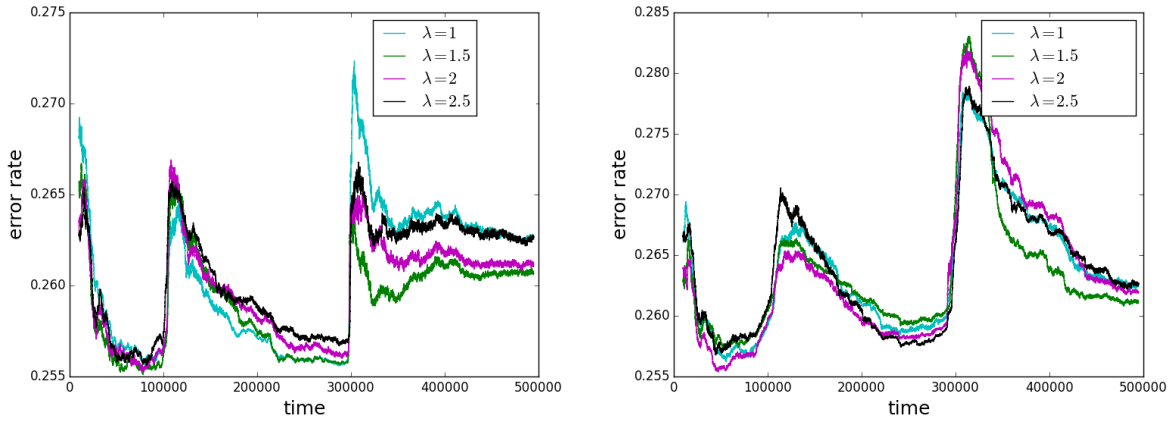


Figure 13: Poisson Probability Mass Function

The data generator we use for this experiment is the same as in the Section 8.3.1. The number of learners is 10. We use the reactive algorithm with parameters $\theta = 10$ and $w = 200$ for the base learner.

Figure 14 presents the result of the experiment. The experiment shows that the increase in the value of λ makes the ensemble less capable to adapt to the drift quickly. This is due to the fact that amount of samples each member in the ensemble was trained on is higher with a higher value of λ. Moreover, with the increase in λ we tend to send the samples to more base learners thus increasing the amount of data sent through network. Due to this fact it might be better to choose the smaller value of λ.



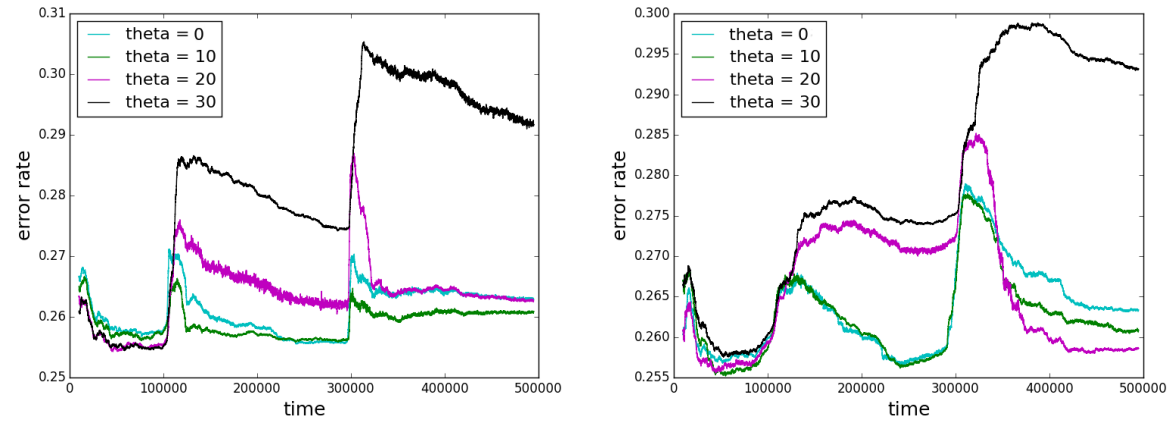
(a) Two abrupt drifts (b) Two gradual drifts

Figure 14: Bagging. Different values of λ parameter

8.3.4. Bagging Algorithm: Parameter θ in the Reactive Algorithm

This experiment examines the effect of the parameter θ in Algorithm 8 on the accuracy of the bagging algorithm. The input data for the experiment is the same as in the previous sections. The number of learners is 10 and Poisson parameter $\lambda = 1.5$.

As described in Section 7.1 parameter θ defines how many times the new reactive model has to outperform the stable one in order to substitute it.



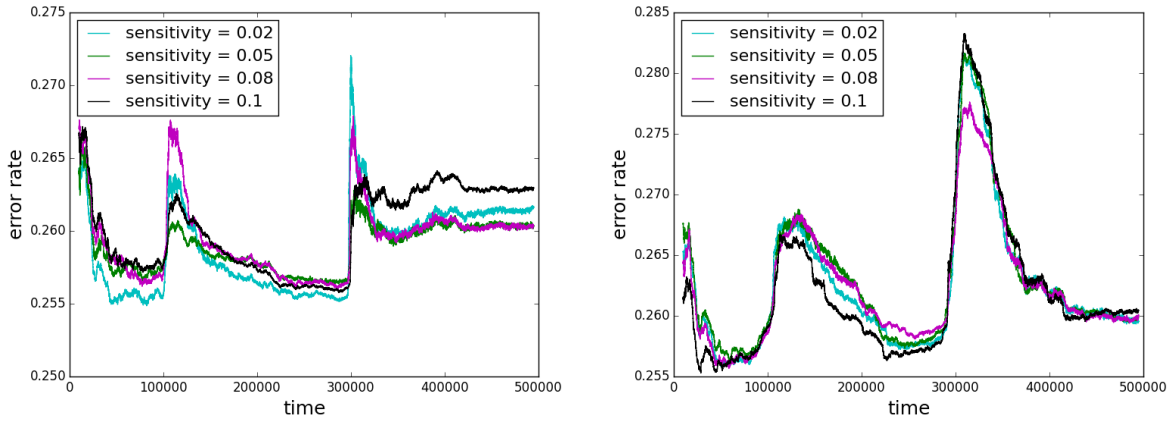
(a) Two abrupt drifts (b) Two gradual drifts

Figure 15: Bagging. Different values of θ parameter in reactive algorithm

The result of the experiment is presented in Figure 15. The result suggests, that the use of the reactive algorithm instead of the plain model reset ($\theta = 0$) makes the performance more stable in periods of drift adaptation, as was already mentioned in Section 8.3.1. However, if the value of the parameter θ is too high, it results in an inability of the algorithm to adapt to drift as it is for $\theta = 30$. With $\theta = 20$ algorithm does not manage to adapt to first abrupt drift (Figure 15a) as good as the options with $\theta = 10$ and $\theta = 0$. In case of gradual drift (15b) both $\theta = 20$ and $\theta = 30$ options are not able to detect the first (less severe) drift. The $\theta = 20$ does however detect the second gradual drift. The algorithm with $\theta = 10$ manages to adapt to all drifts better than the one with $\theta = 0$.

8.3.5. Change Detector Sensitivity

This experiment examines the influence of the change detector sensitivity on the performance of both baseline and bagging algorithms. We use the *ADWIN* change detection algorithm. The parameter which defines the sensitivity of this detector is δ. The higher the value of this parameter – the more sensitive the detector is. The potential problem of highly sensitive change detectors are false alarms. With the introduction of reactive algorithm (see Section 7.1) we hope to prevent the reset of the model on false drift detection.



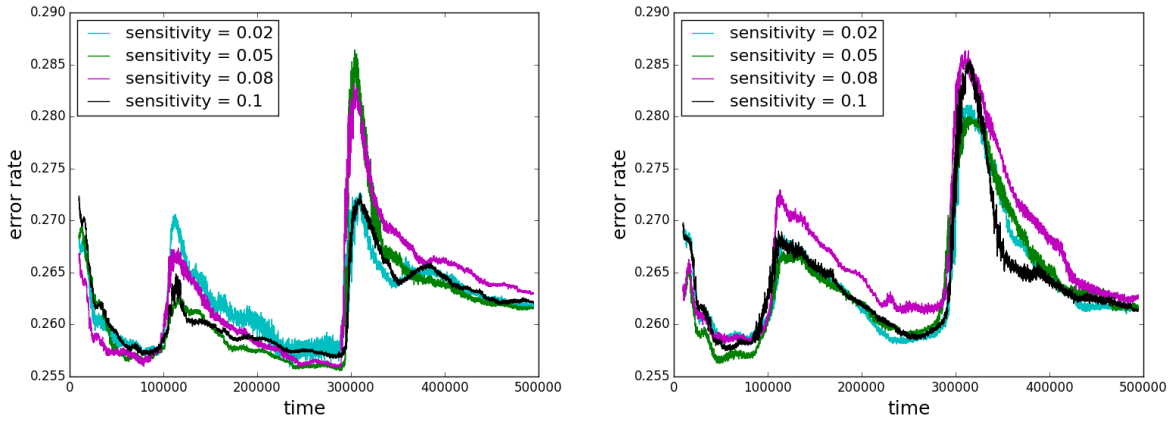
(a) Two abrupt drifts

(b) Two gradual drifts

Figure 16: Bagging. Different values of sensitivity parameter δ in change detector

Figure 16 depicts the influence of the sensitivity parameter on the accuracy of the bagging algorithm. Figure 17 presents the same for the baseline algorithm. As can be seen in the Figure 16a for the bagging algorithm in case of abrupt drifts the lower sensitivity change detector (cases with $\delta = 0.02$ and $\delta = 0.05$) allowed the algorithm to adapt faster to the drift. This was the opposite for the baseline algorithm – the change detector with

the higher sensitivity ($\delta = 0.1$) gave better performance in case of abrupt drift. For the gradual drift with both algorithms (Figure 16b and 17b) the higher sensitivity detector allowed the algorithm to detect the first drift (less severe) better, but proved to perform the worst in case of second (high severity) drift. This can be explained by the fact, that when the drift is more severe the highly sensitive drift detector tends to signalize it multiple times. The reactive algorithm aims at improving the stability in these cases. The result of this experiment shows that the reactive base learner indeed makes the algorithm stable to false alarms, as the value of detector sensitivity actually has very little effect on the performance.



(a) Two abrupt drifts

(b) Two gradual drifts

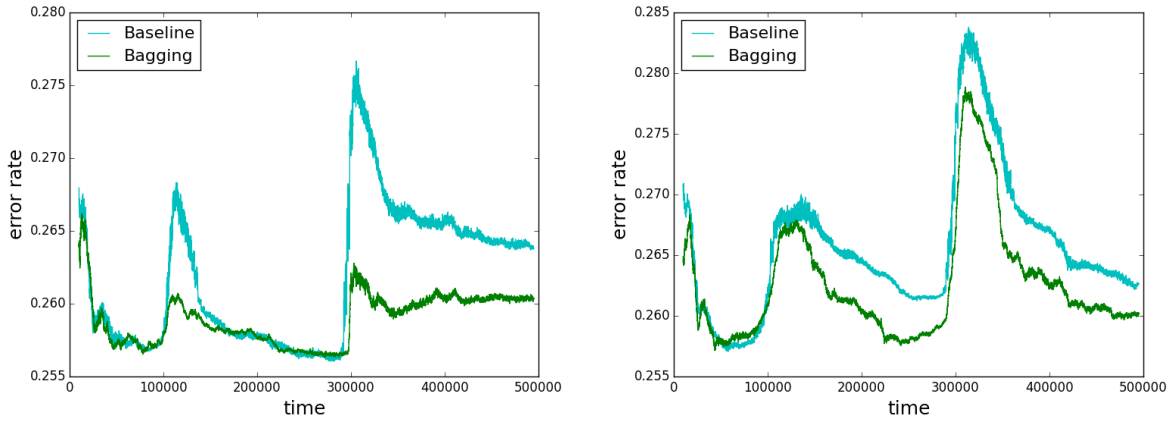
Figure 17: Baseline. Different values of sensitivity parameter δ in change detector

Figure 18 compares the performance of bagging and baseline algorithm with sensitivity values $\delta = 0.05$ for bagging and $\delta = 0.02$ for baseline. We can observe, that the bagging algorithm performs slightly better in this case.

8.3.6. Comparison of Baseline and Bagging Algorithms

This section presents the comparison between the baseline and bagging approaches on different artificially generated datasets.

Figure 19 presents the evaluation on the Agrawal dataset [7]. The data records are characterized by 9 attributes (of both categorical and numerical types) and belong to one of two classes Group A and Group B). The attributes are: salary, commission, age, education level, etc. The drift is simulated by changing the rule which defines the class of the record. There are 10 rules, each of them is defined over different sets of attributes. The examples of the rules are:



(a) Two abrupt drifts

(b) Two gradual drifts

Figure 18: Comparison. Bagging sensitivity = 0.05. Baseline sensitivity = 0.02

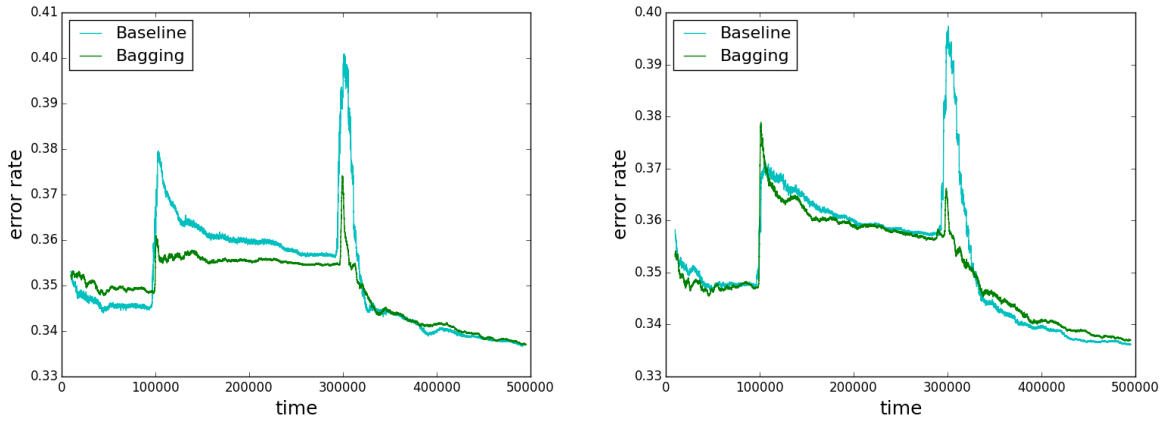
Group A: $((age < 40) \vee (age \geq 60))$ Group B: otherwise	or	Group A: $((age < 40) \wedge (50K \leq salary \leq 100K)) \vee$ $((40 \leq age < 60) \wedge (75K \leq salary \leq 125K)) \vee$ $((age \geq 60) \wedge (25K \leq salary \leq 75K))$ Group B: otherwise

Both abrupt and gradual drifts happen at points 100000 and 300000. The width of abrupt drift is 4 and the width of gradual drift is 50000.

The plot shows that bagging algorithm is capable of learning the new concept faster than the baseline algorithm in all cases. By faster we mean, that the period of higher error rate in points of drift is shorter for bagging algorithm. Also, the plot shows that the base learner can learn some rules better than the others (for example, during the period of first rule, before the point 100000, the error rate was lower than during the second rule, after the first drift).

Figure 20 shows the performance of both algorithms on STAGGER dataset [49]. The dataset has three independent attributes – size, color and shape. There are three different concepts:

- Class A if $size = small \wedge color = red$, Class B - otherwise
- Class A if $color = green \vee shape = circular$, Class B - otherwise
- Class A if $size = medium \vee size = large$, Class B - otherwise

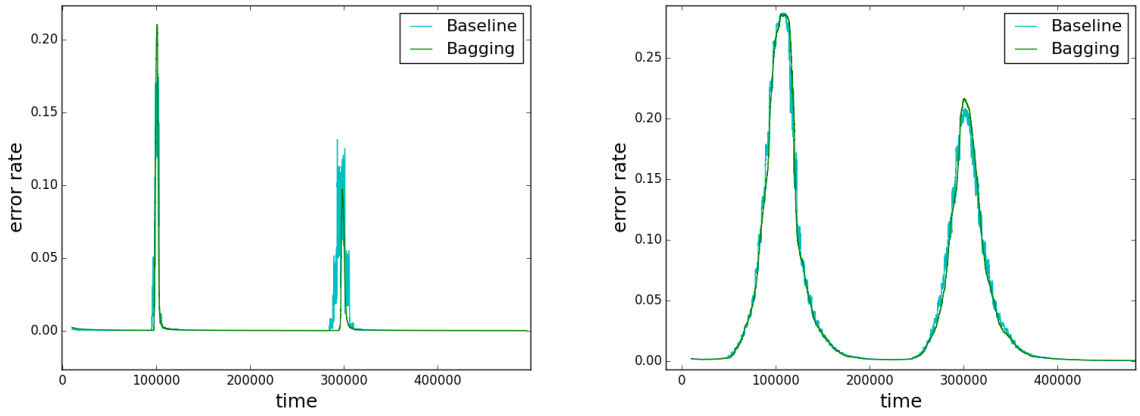


(a) Two abrupt drifts (b) Two gradual drifts

Figure 19: Comparison on Agrawal Dataset

Drifts happen at points 100000 and 300000. The width of abrupt drift is 4 and the width of gradual drift is 50000.

Both algorithms managed to learn the concepts very well and were able to adapt to drift quickly.



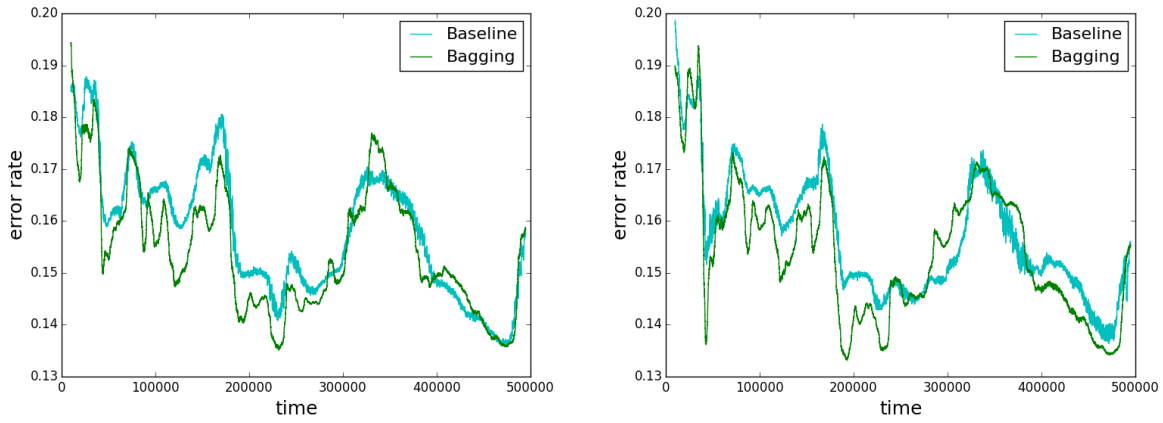
(a) Two abrupt drifts (b) Two gradual drifts

Figure 20: Comparison on STAGGER Dataset

Figure 21 presents the evaluation on Hyperplane dataset [32]. The dataset can be generated with any number of attributes and any number of those can have concept drift. The hyperplane is built by points that satisfy the equation

$$\sum_{i=1}^{d} w_i * x_i = w_0 \tag{22}$$

The points lying on different sides of the hyperplane will belong to different classes. The drift is simulated by changing the orientation and position of hyperplane. The magnitude of this change can be controlled. High magnitude setting produces more abrupt drifts, lower magnitude - gradual drifts. Each record in our generated dataset has 12 attributes, 8 of which are with concept drift, and 4 classes.



(a) Low magnitude drifting

(b) High magnitude drifting

Figure 21: Comparison on Hyperplane Dataset

As we can see in Figure 21 both algorithms manage to adapt to the rotation of the plane. The bagging algorithm in many cases manages to learn new concepts faster and achieves lower error rate.

8.4. Evaluation on Real Datasets

This section presents the evaluation on several real datasets which are widely used for evaluation of the adaptive algorithms. We choose two datasets - electricity price dataset and airline dataset. This type of evaluation allows us to see how our algorithms perform with real-life problems. However, no information about the presence of concept drifts or their position is available for these datasets.

8.4.1. Electricity Dataset

Electricity dataset [31] is a very popular benchmark dataset for performance evaluation of algorithms with concept drift adaptation. The dataset covers two years of electricity prices from New South Wales Electricity Market. The electricity price depends on the demand and supply of the market. As the consumption habits change, the price of electricity is subject to concept drift. The classification task is to predict whether the prices will go up or down. Dataset has 45,312 data points, each characterized by 6 attributes.

Figure 22 presents the performance of baseline and bagging algorithms in comparison with a classifier (Naive Bayes) with no concept drift adaptation.

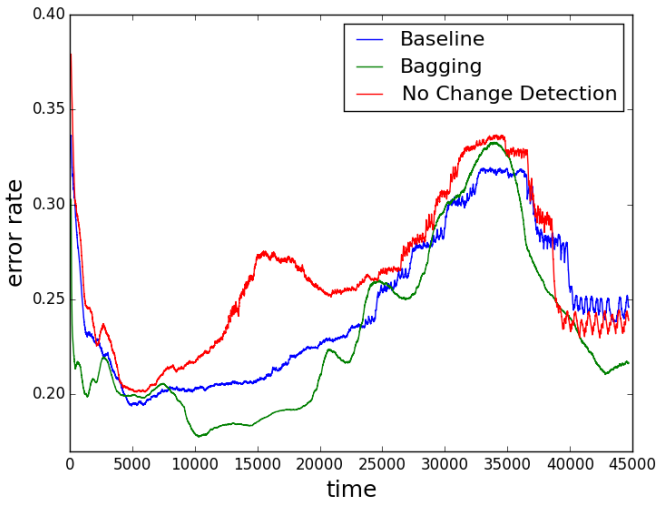


Figure 22: Performance on Electricity Dataset

The introduction of drift adaptation mechanism to an algorithm has a clear benefit, as both bagging and baseline algorithms perform better in terms of classification error rate. The bagging algorithm performs better that others on this dataset. A spike in the error rate level of algorithm with no change detection from point 10000 to point 20000 suggests that there might be a concept drift during this period. There is, however, no such spike in the error rate level of adaptive algorithms, both baseline and bagging, which means that both managed to adapt to the drift.

8.4.2. Airline Dataset

The airline dataset [1] consists of records describing the flight details for all the commercial flights within the USA. We used the normalized dataset available at [5]. The normalized version of dataset consists of seven attributes (*Airline*, *Flight*, *Airport From*, *Airport To*, *Day Of Week*, *Time*, *Length*). The records are sorted according to arrival/departure date. We suppose that the concept drifts could be caused, for example, by the change of season or the internal changes in the airlines (introduction of new vehicles, etc).

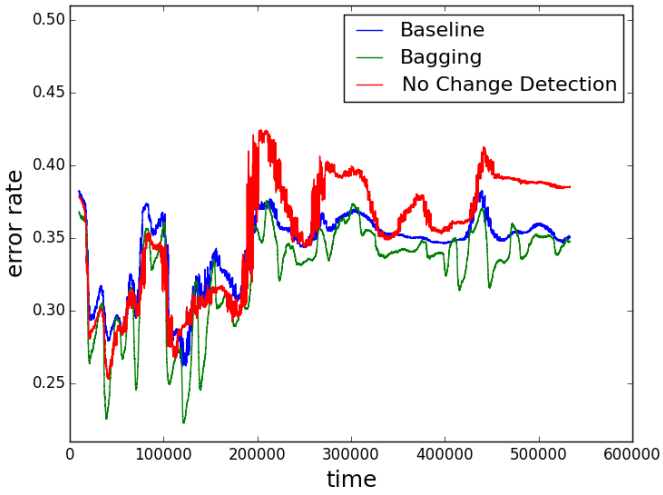


Figure 23: Performance on Airline Dataset

Figure 23 presents the results. The plot suggests that a drift happens around the point 200000, as before the performance of the algorithms with and without the drift adaptation was very similar. After that point there are several spikes in the error rate level in all the algorithms, the adaptive algorithms perform better, however. The bagging algorithm showed a better result in terms of error rate than the baseline.

8.5. Latency and Throughput Evaluation

In this section we present the comparison of our algorithms in terms of latency and throughput.

The setup for the experiment is the following. The cluster has one job manager node and seven task manager nodes. Each node has 3.72 GHz IBM POWER7 processor with 12

physical cores, 4 threads each. The physical memory on each task node is 48 GB, Flink managed memory for each task node is 20.4 GB.

The Flink configuration parameters which are crucial for evaluation of latency and throughput are the number of task slots per task manager, buffer timeout and checkpointing parameter. The number of task slots per task manager defines how many parallel operator instances can run on one task manager node. As the operator instances share the memory of the node this value has to be configured carefully. Flink manual[6] suggests to set this number to the number of physical cores on the task manager machine. Experimenting with different values we came to conclusion that in our case setting the value of 24, double of the number of actual cores, has no negative influence on the latency, while allows to achieve higher throughput.

Before one Flink operator sends the records to the next one in the data flow it can be configured to collect the records in the buffers. Specifically, it can send the buffer once it is full or after some timeout. The buffer timeout configuration parameter defines how much time one operator will be waiting for more records before sending the buffer to the next operator. A smaller value of this parameter will typically result in lower latencies but might have a negative effect on throughput [4]. As we aim to compare the performance of our algorithms, not the influence of Flink configuration on the performance, we set this parameter to 50 ms and do no further changes to it.

The checkpointing parameter allows to switch the Flink checkpoining mechanism on. Flink can periodically make state snapshots of a running stream topology and store them to the durable storage [4]. As we wanted to avoid these delays which are not directly related to the performance of our algorithm we switched the checkpoining mechanism off.

In the case of classification we will measure the latency as the time which it takes to classify the record. More specifically, we will measure the time between the record enters the first operator till it leaves the last one. The initial timestamp is set in the data source which creates the record. As we use the parallel Flink data source we expect that the record will be directly forwarded to the first parallel operator. We set the final timestamp in the Map we added to the end of the dataflow.

In order to measure the scalability of both algorithms we measure how the latency is influenced by the parallelism of the system. More specifically, we run our experiment starting with one task manager machine (parallelism of 24) up to 7 (parallelism of 168). We collect the statistics of per-sample latencies for time periods of more than 10 minutes.

The training data is generated by stream generator for SEA concepts functions [52], already mentioned in Section 8.3.1. Each parallel data source instance generates a sample

[6]https://ci.apache.org/projects/flink/flink-docs-release-0.8/config.html

every 10 milliseconds. The data source is configured to generate 50% of labeled samples, the rest are unlabeled.

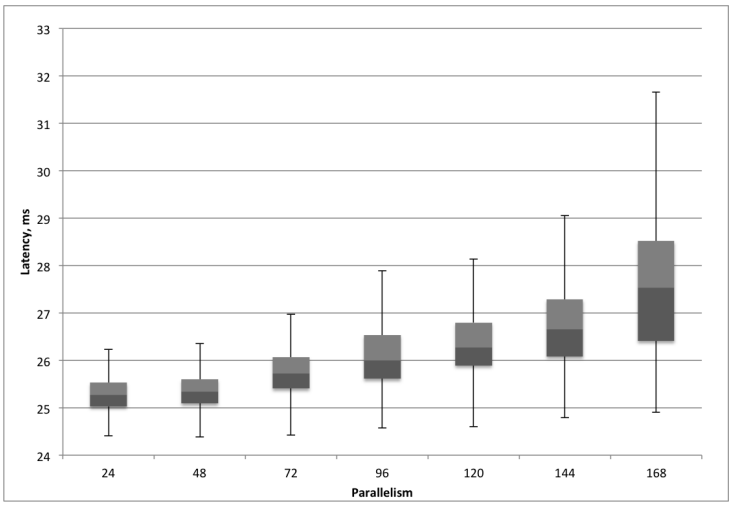


Figure 24: Latency. Baseline algorithm

Figure 24 presents the summary for the Baseline algorithm. The box plot presents the first (Q_1), second (median) and the third(Q_3) quartiles of the data points. It also shows the minimum and maximum achieved latency values. The plot excludes the outliers (points that are more than 3/2 times the interquartile range ($Q_3 - Q_1$) away from quartiles Q_1 and Q_3). We can see a very slight increase in the latency as the parallelism increases. This can be attributed to the fact that each data source has to broadcast the labeled samples to each learner. As the parallelism increases, the number of learners also increases and each data source has to broadcast the training samples to bigger amount of learners. The increase in the number of learners can potentially negatively reflect on the latency. But as the experiment shows this influence is negligible.

Figure 25 presents the results for the Bagging algorithm. As we can see the median of the latency of this algorithm is constant throughout all the parallelism values. This can be attributed to the fact that the operator state is stored on one machine and not distributed across multiple JVMs as in case of several task managers. The number of learners in this experiment is set to 10 and stays constant throughout all the experiments.

The throughput evaluation was performed on the above described cluster with the same configurations. The data source was now configured, however, to produce the records with the maximum possible speed as the aim of this experiment is to measure the maximum amount of records the algorithms can process in one unit of time. Also, as we are interested

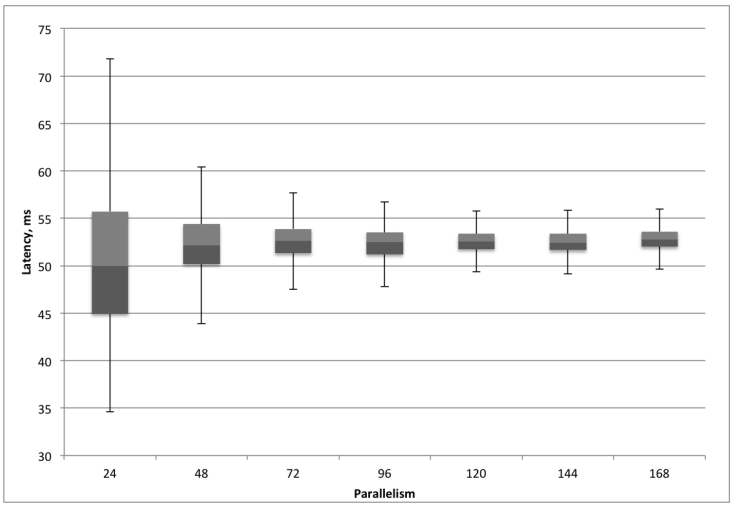


Figure 25: Latency. Bagging algorithm

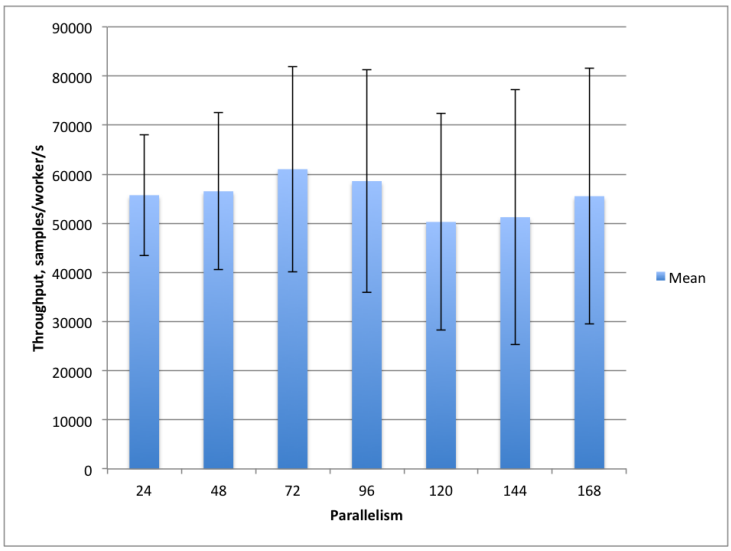


Figure 26: Throughput. Baseline algorithm

in the classification throughput we configured our data source to first train the model with 10000 samples and after that the model was used only for classification.

Figure 26 presents the result of evaluation for baseline algorithm. As expected, the throughput per task does not change throughout the different values of parallelism, which makes this algorithm linearly scalable. This can be explained by the fact that as the new

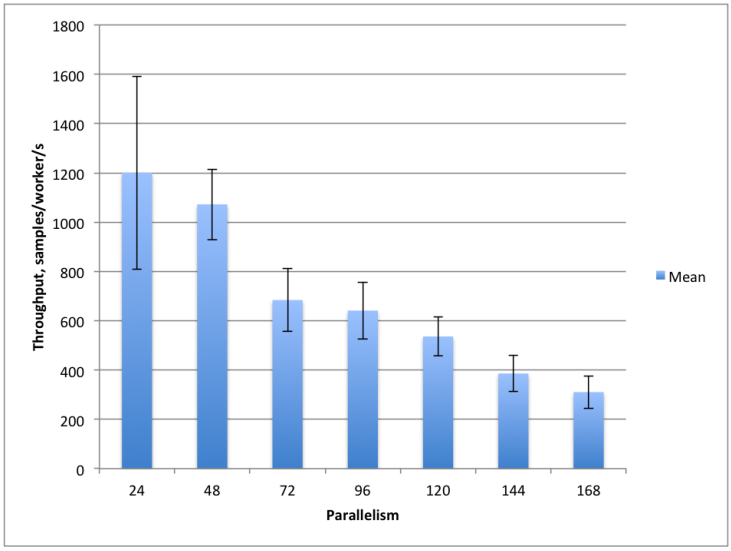


Figure 27: Throughput. Bagging algorithm

tasks are added, the new learners are created and each new learner can take the same amount of load.

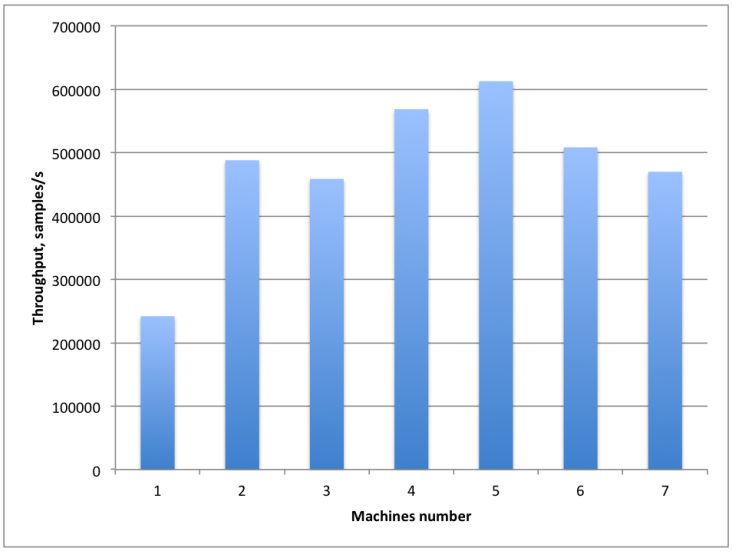


Figure 28: Total throughput. Bagging algorithm

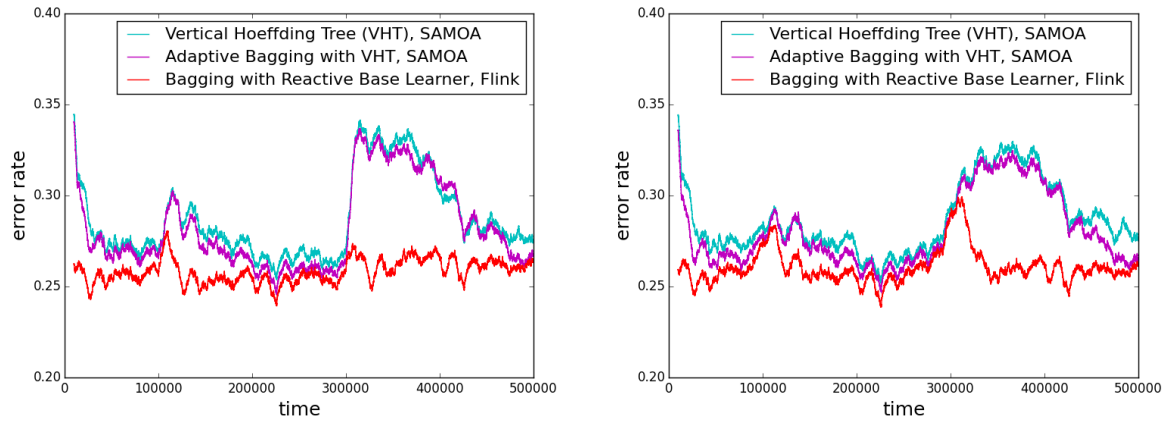
Figure 27 depicts the result of evaluation for bagging algorithm. This plot shows that the throughput per worker decreases as the amount of workers increases. This can be explained by the fact that throughout the evaluation we do not change the amount of

learners. The algorithm can still benefit in terms of throughput from adding more workers, as it means increasing the amount of data distributors and windows which handle majority voting, but, obviously, there is a limit of machines, after which it makes no more sense to add more. Figure 28 presents the total throughput of the system depending on the number of machines. The plot suggests that the optimal amount of machines for our case is 5, after which the total throughput of the system starts to decrease.

8.6. Comparison with Algorithms in Apache SAMOA

Apache SAMOA [3] is a distributed streaming machine learning framework. Among the scalable classification algorithms implemented in Apache SAMOA are *Vertical Hoeffding Tree (VHT)*, Bagging, Adaptive Bagging and Boosting. *VHT* is a scalable verstion of *VFDT* algorithm explained in Section 5.2.2. Adaptive Bagging is a version of bagging where each learner is equipped with a concept drift detector. We compare the performance of our algorithm with the *VHT* algorithm and Adaptive Bagging algorithm that uses *VHT* as base learner and *ADWIN* as the change detector.

In order to be able to compare the performance of these algorithms we implemented the prequential evaluation (see Section 8.1) in Flink, as this technique is also implemented in SAMOA. As a performance tracker we used the Window Performance Tracker (see Section 8.1). The size of the window is 10000.



(a) Two abrupt drifts

(b) Two gradual drifts

Figure 29: Comparison with VHT and Adaptive Bagging algorithms. SEA concepts data generator

Throughout all the experiments in this section we use our implementation of bagging with 10 learners, *ADWIN* as the change detector and reactive algorithm (see Section 7.1) for base learner. Parameters for reactive learner are: $\theta = 10$ and $w = 200$. The Poisson parameter is $\lambda = 1.5$.

Figure 29 presents the comparison of the algorithms on the SEA concepts generator (see Section 8.3.1). We evaluate the performance with both abrupt and gradual drifts. The drifts happen at points 100000 and 300000. The first drift is less severe than the second one. The percent of the noise in the data is 25%. The width of the abrupt drift is 4, gradual – 50000.

Our implementation of bagging algorithm performs better in terms of classification error rate with both types of drift. Moreover, it adapts to the concept drift much faster. What is worth to note is that the non-adaptive *VHT* algorithm adapts to the drift almost as good as Adaptive Bagging algorithm.

Figure 30 presents the result of the evaluation on Hyperplane dataset (see 8.3.6). The parameters for data generation are the same as in the Section 8.3.6. We generate only low magnitude drifts in this experiment.

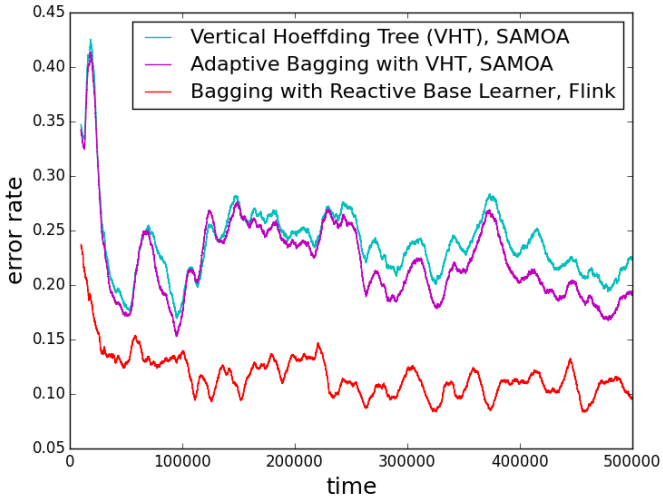


Figure 30: Comparison with VHT and Adaptive Bagging Algorithms. Hyperplane data generator

Our implementation of bagging algorithm performs better in terms of classification error.

Finally, Figures 31 and 32 present the evaluation on the real datasets described in Sections 8.4.1 and 8.4.2.

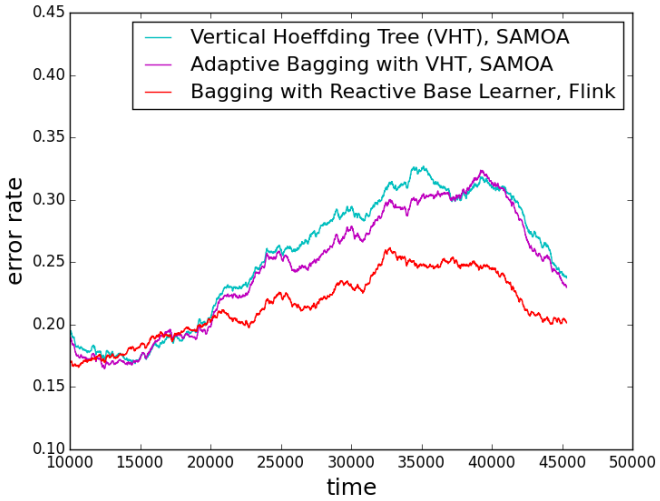


Figure 31: Comparison with VHT and Adaptive Bagging Algorithms. Electricity Dataset

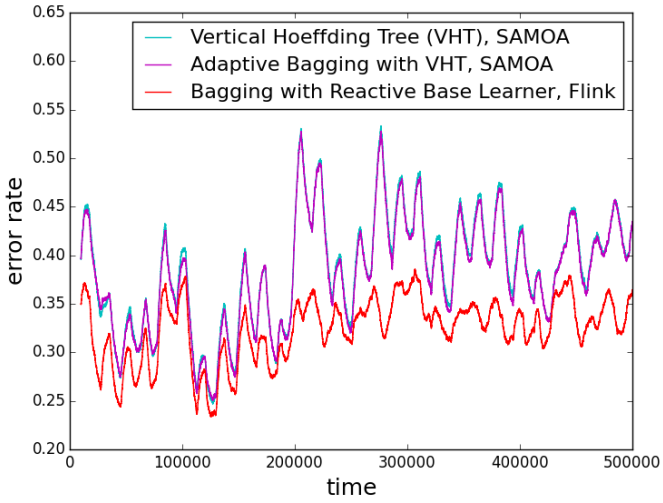


Figure 32: Comparison with VHT and Adaptive Bagging Algorithms. Airline Dataset

The results of the experiments suggest that our bagging algorithm achieves higher classification accuracy in presence of concept drift than both adaptive bagging and *VHT* algorithms. Note, however, that we tuned our algorithm, whereas in Apache SAMOA we used the default parameter values.

Apache SAMOA can use Flink as underlying stream engine. Consequently, we can compare the running times of our algorithm implemented natively in Apache Flink and the Apache SAMOA algorithms running on Flink. The cluster setup for the experiment is the same as in the Section 8.5. We used the parallelism of 168 (7 task manager machines). For a dataset of 500000 samples (618 MB) it took SAMOA implementation of *VHT* on average (from 4 runs) 113.25 s, Adaptive Bagging – 2670.75 s. Our implementation took on average 33.5 s. Note, that the SAMOA job deployed to Flink did not use all the available task slots.

9. Conclusion and Future Work

In this master thesis we explored different approaches to implement scalable adaptive online learning algorithms in the distributed streaming dataflow engine Apache Flink. Firstly, we developed an approach to distribute the arbitrary classification algorithm in Flink. Secondly, we implemented an ensemble algorithm.

The main objective of the thesis was to integrate the change adaptation mechanism into our implementation. We performed a comprehensive research into drift detection and adaptation algorithms. Several ways to integrate the change adaptation into learning algorithms in Flink were discussed. We implemented the one we found the most suitable taking into account Flink engine specifics. Finally, we proposed a drift adaptation algorithm which proved to be more resistant to false drift detections than simply building the new model on each drift detection.

In order to introduce reliable and illustrative evaluation results we explored and compared multiple evaluation techniques and implemented k-fold bagging evaluation. We performed comprehensive comparison of the two main classification algorithms that we implemented. Finally, we compared our adaptive bagging algorithm implementation with the algorithms available in Apache SAMOA and achieved higher classification accuracy and lower execution times. With this, we prove that the native implementation of machine learning algorithm in Apache Flink may achieve higher performance in terms of execution time and better scalability than Apache SAMOA algorithms running on Apache Flink engine.

Finally, we came to the conclusion that Apache Flink Streaming engine is a very promising framework for implementation of online machine learning algorithms. The limiting factor, however, is the absence of some concept of shared state between task instances.

As future work we are interested in exploring how the bagging algorithm could be optimized to deal with the class imbalance problem (see Section 5.4.1). Preliminary experiments showed that the ability of the algorithm to adapt to the drift in the minority class is much lower than when it happens in the majority class. The authors of [41] suggest that the parameter λ in the bagging algorithm can be used to perform undersampling for the majority class and oversampling for the minority class. However, oversampling may lead to overfitting and undersampling may result in the loss of important features in the data. We would like to investigate if this technique could be used for improving drift adaptation capabilities of the algorithm without loss in classification accuracy.

References

[1] Airline dataset. http://kt.ijs.si/elena_ikonomovska/data.html. [Online; accessed 02-Apr-2016].

[2] Apache Flink: Scalable batch and stream data processing. http://flink.apache.org/. [Online; accessed 22-Feb-2016].

[3] Apache SAMOA: Scalable advanced massive online analysis. https://samoa.incubator.apache.org/. [Online; accessed 22-Feb-2016].

[4] Data artisans: High-throughput, low-latency, and exactly-once stream processing with apache flink. http://data-artisans.com/blog/. [Online; accessed 25-Apr-2016].

[5] MOA massive online analysis | real time analytics for data streams | datasets. http://moa.cms.waikato.ac.nz/datasets/. [Online; accessed 02-Apr-2016].

[6] Charu C Aggarwal. *Data streams: models and algorithms*, volume 31. Springer Science & Business Media, 2007.

[7] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Database mining: A performance perspective. *IEEE transactions on knowledge and data engineering*, 5(6):914–925, 1993.

[8] Stephen H Bach and Marcus A Maloof. Paired learners for concept drift. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pages 23–32. IEEE, 2008.

[9] Manuel Baena-Garcıa, José del Campo-Ávila, Raúl Fidalgo, Albert Bifet, R Gavalda, and R Morales-Bueno. Early drift detection method. In *Fourth international workshop on knowledge discovery from data streams*, volume 6, pages 77–86, 2006.

[10] Eric Bauer and Ron Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine learning*, 36(1-2):105–139, 1999.

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

[12] Albert Bifet, Gianmarco de Francisci Morales, Jesse Read, Geoff Holmes, and Bernhard Pfahringer. Efficient online evaluation of big data stream classifiers. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery*

and Data Mining, KDD '15, pages 59–68, New York, NY, USA, 2015. ACM.

[13] Albert Bifet, Eibe Frank, Geoff Holmes, and Bernhard Pfahringer. Ensembles of restricted hoeffding trees. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 3(2):30, 2012.

[14] Albert Bifet and Ricard Gavalda. Kalman filters and adaptive windows for learning in data streams. In *Discovery science*, pages 29–40. Springer, 2006.

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

[16] Albert Bifet and Ricard Gavaldà. Adaptive learning from evolving data streams. In *Advances in Intelligent Data Analysis VIII*, pages 249–260. Springer, 2009.

[17] Albert Bifet, Geoff Holmes, Bernhard Pfahringer, Richard Kirkby, and Ricard Gavaldà. New ensemble methods for evolving data streams. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 139–148. ACM, 2009.

[18] Christian Böhm, Claudia Plant, Junming Shao, and Qinli Yang. Clustering by synchronization. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 583–592. ACM, 2010.

[19] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[20] Edgar Brunner and Ullrich Munzel. The nonparametric behrens-fisher problem: Asymptotic theory and a small-sample approximation. *Biometrical Journal*, 42(1):17–25, 2000.

[21] Ireneusz Czarnowski and Piotr Jędrzejowicz. Ensemble classifier for mining data streams. *Procedia Computer Science*, 35:397–406, 2014.

[22] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows: (extended abstract). In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 635–644, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[23] A Philip Dawid. Present position and potential developments: Some personal views: Statistical theory: The prequential approach. *Journal of the Royal Statistical Society. Series A (General)*, pages 278–292, 1984.

[24] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80. ACM, 2000.

[25] Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 2012.

[26] Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. American Association for Artificial Intelligence, Menlo Park, CA, USA, 1996.

[27] João Gama, Raquel Sebastião, and Pedro Pereira Rodrigues. On evaluating stream learning algorithms. *Mach. Learn.*, 90(3):317–346, March 2013.

[28] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM Comput. Surv.*, 46(4):44:1–44:37, March 2014.

[29] Joao Gama. *Knowledge discovery from Data Streams*. CRC Press, 2010.

[30] Joao Gama, Pedro Medas, Gladys Castillo, and Pedro Rodrigues. Learning with drift detection. In *Advances in artificial intelligence–SBIA 2004*, pages 286–295. Springer, 2004.

[31] Michael Harries and New South Wales. Splice-2 comparative evaluation: Electricity pricing. 1999.

[32] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 97–106. ACM, 2001.

[33] Thorsten Joachims. Making large scale svm learning practical. Technical report, Universität Dortmund, 1999.

[34] Mark G. Kelly, David J. Hand, and Niall M. Adams. The impact of changing populations on classifier performance. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '99, pages 367–371, New York, NY, USA, 1999. ACM.

[35] Ralf Klinkenberg and Thorsten Joachims. Detecting concept drift with support vector machines. In *ICML*, pages 487–494, 2000.

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

[37] Ludmila I Kuncheva. Classifier ensembles for changing environments. In *Multiple classifier systems*, pages 1–15. Springer, 2004.

[38] Ludmila I Kuncheva. Using control charts for detecting concept change in streaming data. *Bangor University*, 2009.

[39] Patrick Lindstrom, Sarah Jane Delany, and Brian Mac Namee. Handling concept drift in text data stream constrained by high labelling cost. 2010.

[40] Leandro L Minku, Allan P White, and Xin Yao. The impact of diversity on online ensemble learning in the presence of concept drift. *Knowledge and Data Engineering, IEEE Transactions on*, 22(5):730–742, 2010.

[41] Leandro L Minku and Xin Yao. Ddd: A new ensemble approach for dealing with concept drift. *Knowledge and Data Engineering, IEEE Transactions on*, 24(4):619–633, 2012.

[42] Leandro Lei Minku. *Online ensemble learning in the presence of concept drift*. PhD thesis, University of Birmingham, 2011.

[43] C Monteiro, R Bessa, V Miranda, A Botterud, J Wang, G Conzelmann, et al. Wind power forecasting: state-of-the-art 2009. Technical report, Argonne National Laboratory (ANL), 2009.

[44] Markus Neuhäuser and Graeme D Ruxton. Distribution-free two-sample comparisons in the case of heterogeneous variances. *Behavioral Ecology and Sociobiology*, 63(4):617–623, 2009.

[45] Nikunj C Oza. Online bagging and boosting. In *Systems, man and cybernetics, 2005 IEEE international conference on*, volume 3, pages 2340–2345. IEEE, 2005.

[46] M. Pechenizkiy, J. Bakker, I. Žliobaitė, A. Ivannikov, and T. Kärkkäinen. Online mass flow prediction in cfb boilers with explicit detection of sudden concept drift. *SIGKDD Explor. Newsl.*, 11(2):109–116, May 2010.

[47] Bernhard Pfahringer, Geoffrey Holmes, and Richard Kirkby. New options for hoeffding trees. In *AI 2007: Advances in Artificial Intelligence*, pages 90–99. Springer, 2007.

[48] Gordon J Ross, Niall M Adams, Dimitris K Tasoulis, and David J Hand. Exponentially weighted moving average charts for detecting concept drift. *Pattern Recognition Letters*, 33(2):191–198, 2012.

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

[50] Junming Shao, Zahra Ahmadi, and Stefan Kramer. Prototype-based learning on concept-drifting data streams. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 412–421. ACM, 2014.

[51] Junming Shao, Xiao He, Christian Bohm, Qinli Yang, and Claudia Plant. Synchronization-inspired partitioning and hierarchical clustering. *Knowledge and Data Engineering, IEEE Transactions on*, 25(4):893–905, 2013.

[52] W Nick Street and YongSeog Kim. A streaming ensemble algorithm (sea) for large-scale classification. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 377–382. ACM, 2001.

[53] S. Subramaniam, T. Palpanas, D. Papadopoulos, V. Kalogeraki, and D. Gunopulos. Online outlier detection in sensor data using non-parametric models. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 187–198. VLDB Endowment, 2006.

[54] Haixun Wang, Wei Fan, Philip S. Yu, and Jiawei Han. Mining concept-drifting data streams using ensemble classifiers. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 226–235, New York, NY, USA, 2003. ACM.

[55] Heng Wang and Zubin Abraham. Concept drift detection for streaming data. In *Neural Networks (IJCNN), 2015 International Joint Conference on*, pages 1–9. IEEE, 2015.

[56] Shuo Wang, Leandro L Minku, Diego Ghezzi, Daniele Caltabiano, Peter Tino, and Xin Yao. Concept drift detection for online class imbalance learning. In *Neural Networks (IJCNN), The 2013 International Joint Conference on*, pages 1–10. IEEE, 2013.

[57] Shuo Wang, Leandro L Minku, and Xin Yao. A learning framework for online class imbalance learning. In *Computational Intelligence and Ensemble Learning (CIEL), 2013 IEEE Symposium on*, pages 36–45. IEEE, 2013.

[58] Arthur B Yeh, Richard N Mcgrath, Mark A Sembower, and Qi Shen. Ewma control charts for monitoring high-yield processes based on non-transformed observations. *International Journal of Production Research*, 46(20):5679–5699, 2008.

[59] Matt Zwolenski, Lee Weatherill, et al. The digital universe: Rich data and the increasing value of the internet of things. *Australian Journal of Telecommunications and the Digital Economy*, 2(3):47, 2014.

Appendix

A. Bagging Algorithm Implementation Details

A.1. Data Distributor

```
class BaggingDataDistributor implements FlatMapFunction<Item, Tuple3<Integer, Item, Double>> {

  public void flatMap(Item item,
      Collector<Tuple3<Integer, Item, Double>> out) {

    for (int i = 0; i < ensembleLength; i++) {
      if (item.hasClassLabel()) {
        int weight = Poisson(theta);
        if (weight > 0) {
          out.collect(new Tuple3<Integer, Item, Double>(i, item, weight);
        }
      } else {
        out.collect(new Tuple3<Integer, Item, Double>(i, item, 0.0);
      }
    }
  }
}
```

Listing 1: Bagging: Data Distributor

A.2. Learner

```
public class BaggingLearner extends
      RichFlatMapFunction<Tuple3<Integer, Item, Double>,
                Tuple3<String, Item, Double>> {

  public void flatMap(Tuple3<Integer, Item, Double> tuple,
      Collector<Tuple3<String, Item, Double>> out) {

    if (item.hasClassLabel()) {
      model.train(item);
    } else {
      Double label = model.classify(item);
      out.collect(item.getId(), item, label);
    }
  }
}
```

Listing 2: Bagging: Learner

A.3. Majority Voting Window Function

```
public class BaggingMajorityVotingFunction implements
          WindowFunction<Tuple3<String, Item, Double>,
                 Tuple2<Item, Double> Tuple, GlobalWindow> {

  @Override
  public void apply(Tuple key, GlobalWindow window,
      Iterable<Tuple3<String, Item, Double>> values,
      Collector<Tuple2<Item, Double>> out) throws Exception {

    HashMap<Double, Long> voteCounter = new HashMap<Double, Long>();
    Item item = null;
    int count = 0;
    for (Tuple3<String, Item, Double> tuple : values) {
      item = tuple.f1;
      double label = tuple.f2;
      Long currentCount;
      if ((currentCount = voteCounter.get(label)) == null) {
        voteCounter.put(label, 1L);
      } else {
        voteCounter.put(label, currentCount + 1);
      }
    }

    double maxClass = -1;
    long maxCount = -1;
    for (Entry<Double, Long> entry : voteCounter.entrySet()) {
      if (entry.getValue() > maxCount) {
        maxCount = entry.getValue();
        maxClass = entry.getKey();
      }
    }
    out.collect(new Tuple2<Item, Double>(item, maxClass));
  }
}
```

Listing 3: Bagging: Majority Voting Window Function