POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione

Master of Science in Computer Science and Engineering

A Comparative Study on Streaming Machine Learning Algorithms for Binary Classification Under Concept Drift and Class Imbalance

Supervisor: Prof. Emanuele Della Valle

Co-supervisor: Dr. Alessio Bernardo

Master Graduation Thesis
Enrico Voltan
928681

Academic Year 2019/2020

Abstract

Nowadays, data coming from electronic devices, such as smartphones, credit cards, televisions, and cars, surround us. The ability to analyze all the data generated in real-time is the future challenge and the starting point to understand where to innovate. The research for new solutions opened a new branch of machine learning called Streaming Machine Learning (SML). This new approach focuses on data streams, sequences of data, possibly infinite, arriving in sequential order, once at a time. The two difficulties appearing in the real-world are concept drift and class imbalance. The former refers to the changes in the characteristics of the data, while the latter refers to an unequal distribution between the classes. Focusing on the binary classification task, I studied and implemented various state-of-art algorithms able to deal with both the concept drift and class imbalance problems. The result is an easy-to-reuse benchmarking environment that we exploited to conduct a wide experimental campaign. I tested the algorithm on artificial and real data streams with different imbalance levels and various kinds of concept drift. I collected empirical evidence that rebalancing data streams significantly improves the performances during different concept drift types.

Sommario

Al giorno d'oggi, dati provenienti da dispositivi elettronici come smartphones, carte di credito, televisioni e automobili, ci circondano. La capacità di analizzare tutti questi dati in tempo reale è la sfida del futuro e il punto di partenza per capire dove innovare. La ricerca di nuove soluzioni ha aperto un nuovo ramo di apprendimento automatico chiamato "Streaming Machine Learning" (SML). Questo nuovo metodo analizza sequenze di dati, chiamate "data streams", potenzialmente infinite, che arrivano in ordine temporale. Le due maggiori difficoltà che appaiono in situazioni reali in questo campo sono il "concept drift" e lo sbilanciamento di classe. La prima si riferisce a possibili cambiamenti nelle caratteristiche dei dati, mentre la seconda a disparità di rappresentazione tra le classi di dati. Concentrandomi sulla classificazione binaria, ho studiato e implementato gli algoritmi dello stato dell'arte che affrontano entrambi i problemi. Il risultato è un ambiente di valutazione che ho usato per condurre una estesa campagna sperimentale. Ho testato gli algoritmi usando "data streams", sia artificiali sia reali, aventi diversi livelli di sbilanciameto di classe e vari tipi di "concept drift". Le prove sperimentali raccolte dimostrano che affrontare il problema dello sbilanciamento di classe migliora significativamente le prestazioni durante i diversi tipi di "concept drift".

Acknowledgements

I wish to express my gratitude to Prof. Emanuele Della Valle for giving me the opportunity of working on such an interesting research. I am grateful for his availability and support.
I wish to extend my special thanks to Dr. Alessio Bernardo for his supervision and guidance during all the phases of the research and for overseeing the writing of my thesis.
I wish to express my profound gratitude to my parents for all the sacrifices they have always made for me. I am truly thankful to them for always encouraging me to be the best version of myself.

Contents

1 Introduction **1**
1.1 Big Data . . . 1
1.2 Streaming Machine Learning . . . 2
1.3 Contributions of the thesis . . . 4
1.4 Document Structure . . . 5

2 State of the Art **7**
2.1 Traditional Machine Learning . . . 7
2.2 Streaming Machine Learning . . . 8
2.2.1 Hoeffding Tree . . . 9
2.3 Concept Drift . . . 11
2.3.1 Speed of Drift . . . 13
2.3.2 Severity of Drift . . . 14
2.3.3 Concept Drift detection . . . 14
2.4 Class Imbalance . . . 16
2.4.1 Minority class distributions . . . 17
2.4.2 Offline Algorithms to deal with Class Imbalance . . . 17
2.5 Evaluation Methodology . . . 18
2.5.1 Error Estimation . . . 18
2.5.2 Performance Measures . . . 19

3 Problem Setting **23**
3.1 SML Algorithms to deal with Class Imbalance and Concept Drift . 23
3.1.1 C-SMOTE . . . 23
3.1.2 RebalanceStream . . . 24
3.1.3 Online Bagging . . . 26
3.1.4 Ensemble of Online Sequential Extreme Learning Machine . 29
3.2 Problem Statement . . . 32
3.2.1 Benchmarking environment . . . 32
3.2.2 Replication Study . . . 34
3.2.3 Data streams . . . 34
3.2.4 Performances evaluation . . . 35
3.2.5 Problem Recap . . . 35

4 Problem Solving **37**
4.1 Technologies adopted . . . 37
4.1.1 Implementing and running the algorithms . . . 37
4.1.2 Tracking memory requirements . . . 39
4.1.3 Visualizing the results . . . 39
4.2 Data streams . . . 39
4.2.1 SEA and SINE1 . . . 40
4.2.2 Cluster generator . . . 41
4.2.3 Real Datasets . . . 46

5 Implementation experience **49**
5.1 Cluster drift generation . . . 49
5.2 Algorithms implementation . . . 50
5.2.1 Native OOB and UOB . . . 51
5.2.2 Improved OOB and UOB . . . 52
5.2.3 Ensembles of the improved versions . . . 54
5.3 Benchmarking setup . . . 57
5.4 Experiments configuration . . . 58

6 Results **61**
6.1 P(X|y) drift . . . 61
6.2 P(y|X) drift . . . 64
6.3 P(y) drift . . . 68
6.4 Resources requirements . . . 71
6.5 Real datasets . . . 72

7 Conclusions and Future Work **77**

Appendix A **79**

Bibliography **95**

List of Figures

2.1 An example of decision tree structure. 9
2.2 An example of decision tree classifier 10
2.3 An example of the three sources of concept drift. 12
2.4 Possible temporal characteristics of a drift, src: [1]. 13
2.5 Confusion Matrix. 20

3.1 The structure of a Single hidden-Layer Feed-forward Network, src: [2]. 30

4.1 Data stream classification cycle, src: [3]. 38
4.2 The illustrated sequence of the appearing minority drift. 44
4.3 The illustrated sequence of the disappearing minority drift. 44
4.4 The illustrated sequence of the minority share drift. 44
4.5 The illustrated sequence of the cluster movement drift. 45
4.6 The illustrated sequence of the cluster jitter drift. 45
4.7 The illustrated sequence of the appearing cluster drift. 45
4.8 The illustrated sequence of the splitting cluster drift. 46
4.9 The illustrated sequence of the shape shift drift. 46
4.10 The illustrated sequence of the borderline drift. 46
4.11 The distribution of the class probability during the real data streams. 48

5.1 Original stream generation sequence. 50
5.2 Custom stream generation sequence. 50
5.3 The UML class diagram of the Online bagging algorithms. 51
5.4 The experiments workflow. 57

6.1 Mean Recall and Fscore with streams having a P(X|y) drift. 62
6.2 Recall and Gmean during streams having a P(X|y) drift and high imbalance ratios. 64
6.3 Mean Recall with streams having a P(y|X) drift. 65
6.4 Recall and Fscore during streams having a P(y|X) drift and Imbalance ratio *1:9*. 66

6.5 K temporal and Gmean during SEA and SINE1 streams having a P(y|X) drift. 67
6.6 Recall during streams having a P(y) drift with imbalance ratio *1:9*. . . 69
6.7 Gmean during streams having a P(y) drift 70
6.8 Time and Memory requirements of each algorithm. 71
6.9 Mean Time and Memory requirements of the Online Bagging based algorithms. 72
6.10 Recall average with real datasets. 74
6.11 Gmean during real datasets. 75

List of Tables

2.1 Main differences between batch and stream learning, src: [4]. 9

4.1 The characteristics of the artificial data streams used in the experiments. 42

6.1 Average statistics with real datasets. 73

List of Listings

5.1 The update of the class sizes method 52
5.2 The update of the class recalls method. 52
5.3 The class imbalance detection. 53
5.4 The native OOB lambda computing 53
5.5 The native UOB lambda computing 53
5.6 The training procedure 53
5.7 The Improved OOB methods 54
5.8 The Improved UOB methods 54
5.9 The updating procedure of the smoothed recall 55
5.10 The prediction procedure of the WEOB1 ensemble 56
5.11 The prediction procedure of the WEOB2 ensemble 56

List of Algorithms

1 Pseudocode of Hoeffding Tree. 11
2 Pseudocode of C-SMOTE 24
3 Pseudocode of RebalanceStream 25
4 Pseudocode of the RebalanceStream new learners training 26
5 Pseudocode of Online Bagging methods 28
6 Pseudocode of Improved Online Bagging methods 29
7 Pseudocode of ESOS-ELM 33

Chapter 1

Introduction

Nowadays, we are generating data through the devices we interact with. This big amount of information, called *BigData*, could be analyzed and exploited to make our lives better.

1.1 Big Data

Data available are becoming "bigger" everyday. The definition of "bigger" has been characterized by Laney et al. [5] with three V and extended by Khan et al. [6] with seven V:

- Volume: In January 2021, there were 4,2 billion social media users worldwide and each user, in average, spend almost 7 hours online each day, of which 2 hours and 25 minutes using social media [7]. The size of the data generated continues to grow but not as much as our tools ability to process it;
- Velocity: Data move fast. For example, it takes seconds to process a credit card swipe, going through a fraudulent transaction detection system, payment system, bookkeeping, and so on;
- Variety: Data appear in various unstructured forms like social media updates, photos, videos, sensors, voice recordings and so, extracting information from them is not straightforward as in traditional databases;
- Veracity: Data are not always accurate and trustworthy. They need to be analyzed also in this dimension to extract correct information;
- Validity: Data can be valid for a specific application and then invalid for another application. Validity measures the correctness of data with respect to the intended usage;

- Volatility: Data have a retention period after which they are not useful or the cost of storing them exceeds their value; and
- Value: Data have value as long as they lead to better decisions. This is the only motivation for processing these large datasets. It's a trade-off between costs and benefits.

Traditionally, data are stored in a database and they are analyzed all together. They are split in two sets, one used for training the model and one used for testing it. Every time new data are collected, the model is recomputed. When Volume or Velocity become too big, this process becomes expensive and sometimes unfeasible. Searching for new solutions that can deal with all these dimensions will lead to building more useful models increasing the relevance of this science. Streaming Machine Learning (SML) aims at doing this with a completely new approach.

1.2 Streaming Machine Learning

The research for new solutions has opened a new branch of Machine Learning called Streaming Machine Learning (SML). This new approach is based on data streams, sequences of data arriving once at a time. The goal is to get information in real-time without the need to come back to already seen data and using a limited amount of resources such as memory and time. Two big challenges in this new field are the concept drift phenomenon [8] and the class imbalance [9]. Data streams evolve over time and the distributions from which the data are obtained can change. Models built on old data can become inconsistent. This is called concept drift and can happen in various forms and speeds. The second challenge concerns data streams where there is an unequal distribution between the classes. Since the instances contained in the minority class(es) rarely occur, the patterns for classifying these classes tend to be rare, undiscovered, or ignored i.e.fraudulent transactions or spam emails. These kind of information are rare but important and models need to take them into consideration. The range of applications of Streaming analysis is wide, and the problems of concept drift and class imbalance are always present. Some examples are:

- Internet of Things: every day, new sensors are placed in industries to monitor processes [10], in houses to improve security [11] or in cities to monitor the mobility of people [12].
  - Concept Drift: sensors' goals could be the detection of anomalies in finished products. Defects can change frequency and location in the product's surface based on which industrial machine is damaged.

  - Class Imbalance: following the same example, anomalies are usually rare events that need to be captured correctly to maintain high quality standards.

- Social Media: users continuously produce data about their interactions through photos, likes, comments, tags. These data are exploited to recommend new and interesting content [13].
  - Concept Drift: people's interests change with time and recommendation engines need to adapt to it in order to maintain good performances.
  - Class Imbalance: recommendation engines are usually based on categories but content categories have different audience sizes. This will be a problem if the number of interactions is not artificially rebalanced.

- Health Care: hospitals collect a large amount of real-time data about the patients that need to be fast and accurately processed [14]. In this type of environment, being able to process both static data and real-time data can be game-changing.
  - Concept Drift: a patient's vital sign could change at any moment without any warnings and the algorithm for the classification of the general health status needs to change accordingly.
  - Class Imbalance: hospitals monitor patient's vital signs and the machine's predictions about a particular disease need to be re-balanced with respect to the frequencies and dangerousness of these diseases.

- Epidemics and disasters: using real-time data to analyze this kind of time-sensitive information can help with disaster control and prevention [15].
  - Concept Drift: during the Covid-19 epidemics, the number of infections changes rapidly. In Italy, for example, there where three major spreading periods that differ on both the location of the most affected areas and the number of infections.
  - Class Imbalance: in order to make an accurate prediction, the features of positives, which come from a minority class, need to be re-balanced with respect to the negatives which come from the majority class.

1.3 Contributions of the thesis

Algorithms proposed to deal with concept drift have been tested on balanced streams with various types of drifts, while algorithms proposed to deal with class imbalance have been tested on stable streams or streams where the only change was on the imbalance ratio. The contributions of my thesis can be summarized as follows:

- Implementation of the state-of-the-art algorithms for classification with imbalanced data streams and concept drift, increasing the suite of algorithms offered by MOA [3], an open-source software library to run machine learning experiments on data streams;
- Upgrade of a synthetic data streams generator [16] implemented in MOA [3] adding the possibility to generate concept drifts of various types and speeds and imbalance ratios;
- The building of a benchmarking environment for algorithm comparison which allows to reproduce my experiment comparing SML algorithms implemented in MOA using the developed data stream generator; and
- Comparison of the state-of-the-art algorithms detailing their strengths and weaknesses, proposing new solutions to enhance them.

1.4 Document Structure

The thesis is structed as follows:

- **Chapter 1: Introduction** - It introduces the streaming approach to machine learning. It explains which are the main challenges to face.
- **Chapter 2: State Of Art** - It explains the main characteristics of streaming machine learning. It introduces the concept drift and class imbalance problems. In the end, it details the model evaluation methodologies.
- **Chapter 3: Problem Setting** - It details the existing solutions and It explains why there was a need for a benchmarking environment.
- **Chapter 4: Problem Solving** - It describes my approach to the problem. It details the technologies adopted and the characteristics of the data streams used in the experiments.
- **Chapter 5: Implementation Experience** - It describes my upgrade of the data stream generator and how I structured the implementation of the state-of-the-art algorithms. Finally, it presents the building blocks of the benchmark environment.
- **Chapter 6: Results** - It presents the results using detailed tables and plots. The performances of the algorithms are analyzed from different angles using various measures.
- **Chapter 7: Conclusions and Future Work** - It discusses the conclusions based on the experiments and outlines some directions for future improvements of this work.
- **Appendix A: IEBench** - It presents the paper submitted to the ECML PAKDD 2021 conference which summarizes this work.

Chapter 2

State of the Art

In this chapter, I present the SML approach detailing its characteristics and its challenges. Section 2.1 describes the traditional approach to machine learning, while Section 2.2 explains what changed w.r.t. the Streaming approach. Section 2.3 explains concept drift, the first challenge of this field, and it describes the already existing solutions. Section 2.4 explains the challenge of class imbalance and the existing approaches in traditional settings. Finally, Section 2.5 describes the evaluation methodologies detailing the techniques to test the models and to measures their performances.

2.1 Traditional Machine Learning

Traditional Machine Learning algorithms build models to extract information from a dataset composed by already collected data. The workflow of a Machine learning application [17] is composed by phases listed below.

- Data Collection: data are collected through hardware sensors, software applications, or user surveys and stored in a database.
- Data Pre-processing: data are processed and made suitable for the analysis. They are transformed in a multidimensional format in which every field is a different measure. The final result is a structured dataset ready to be processed by the algorithms.
- Model Construction: during this phase, models are built using only a subset of the data called *training set*. In order to choose the most suited algorithm and its parameters, techniques like cross-validation [18] are adopted.

- Testing phase: during this phase, the chosen model is fitted on the whole *training set* and evaluated on the remaining set of data called *testing set*. The result is a final score about the model performances with unseen data.

- Prediction phase: finally, the model is used to make predictions on new and unlabeled data.

The main issues of this approach appear when new labeled data arrive. Every time this happens, the whole model needs to be retrained in order to use the new information. When data arrive continuously, it becomes prohibitive with respect to the available resources to store and analyze them. The streaming learning approach aims to solve this problem.

2.2 Streaming Machine Learning

Online or Streaming Machine Learning (SML) algorithms learn from a continuous and possibly infinite data flow. The main advantage of SML models is that they do not need to be retrained every time new samples arrive but they can learn incrementally. These streaming algorithms have new time and resource constraints:

- Process one data sample at a time;
- Use a limited amount of time to process each data sample;
- Use a limited amount of memory;
- The update of the model with new data sample must be carried out without making intensive use of the already considered data;
- Be ready to make new predictions at any time; and
- Adapt to temporal changes.

The streaming setting differs from the traditional one for numerous reasons. It is not possible to split the dataset into different sets but new approaches need to be implemented to train and test the models. In Streaming scenarios, new specific challenges arise. For example, if the task is to detect fraudulent transactions, when we label one as a fraud it cannot be executed and it will not be possible to know what it really was. Often labels can be missing or delayed. I list the main differences in Table 2.1. In this thesis, I will focus on the most widely used binary classification task where the model needs to assign the correct class label to every data sample. The experiments are run in a clean way following this workflow:

- A new sample (X, y) arrives with both its features and label;
- The model f makes a prediction $\hat{y} = f(X)$;
- The pair (X, y) is used to train the model f and the pair $(\hat{y}, y)$ is used to update the statistics about the classifier performance.

Table 2.1: Main differences between batch and stream learning, src: [4].

Batch learning	Stream learning
Offline	Real Time
Slow data generation	Fast data generation
Persistent data	Transient data
Process entire data	Process samples of data
Constant availability	Limited availability
Fixed size	Unbound size
Random access	Sequential access
Known data characteristics	Unpredictable data characteristics

2.2.1 Hoeffding Tree

In traditional machine learning, a widely used category of classifiers is decision trees. They are models composed of nodes, branches, and leaves as shown in

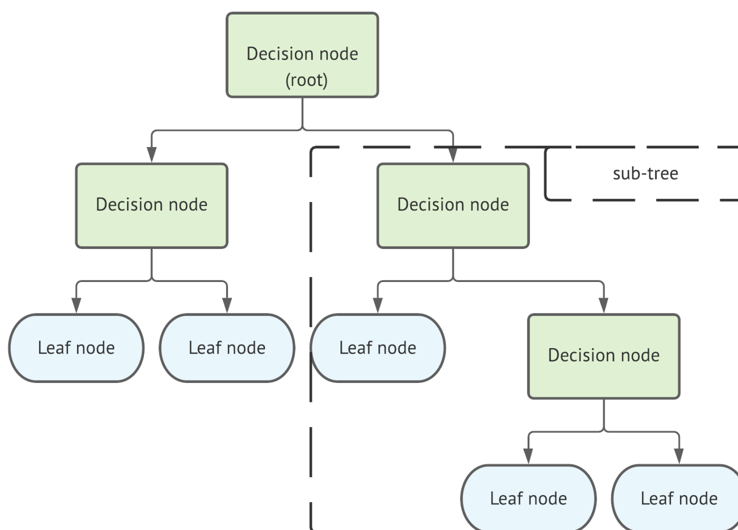


Figure 2.1: An example of decision tree structure.

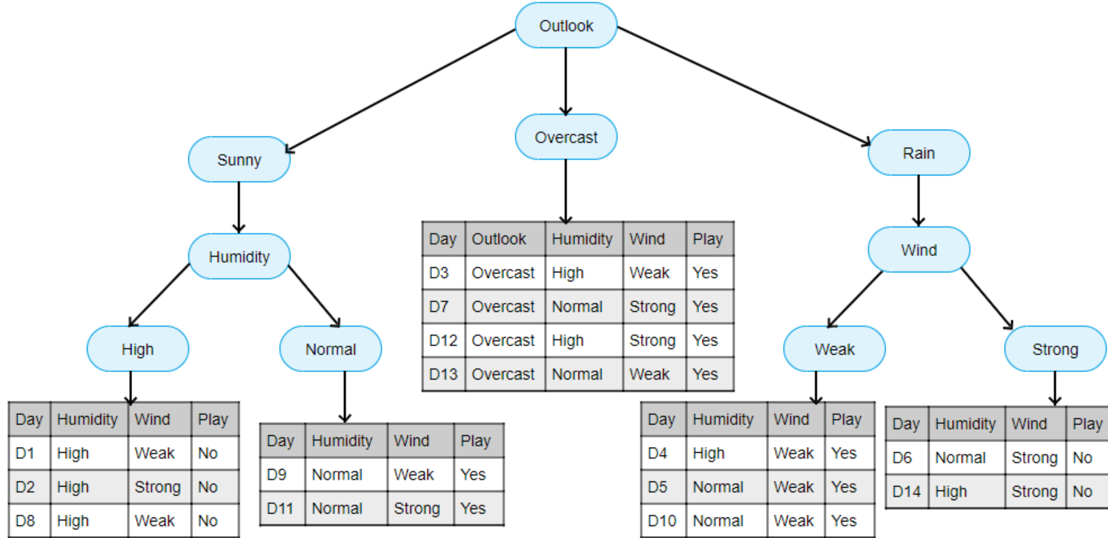


Figure 2.2: An example of decision tree classifier, src[1].

Figure 2.1. Given a training set in input composed of samples with n features and a label, the decision tree algorithm splits the set into subsets based on an attribute value. In this way, it creates a node of the tree. This process is repeated recursively on each subset until the data in the subset in a node have all the same label value, an example of this procedure is shown in Figure 2.2.

The Hoeffding Tree [19] is a SML model that builds a tree incrementally. Starting with a root node, once enough data samples have passed through that node, a test is chosen and the corresponding leaves created. This is done recursively for every node. A node is split when it has processed enough sample that the Hoeffding bound proposed by Hoeffding in [20] has been exceeded. This bound states that, with probability 1-δ, with n independent observations, the true mean of a random variable with range R (in case of classification it is $log(c)$ where c is the number of classes for a discrete variable) differs from the empirical mean by not more than the ϵ value calculated as in Equation 2.1.

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

This bound is used to compare the Gini index (introduced by Gini in [21]) of the attributes in a node. The Gini index of a set S measures the error rate of random classifier that assigns classes to instances according to their prior frequencies,

Algorithm 1 Pseudocode of Hoeffding Tree.

```
1: function HoeffdingTree(S, HT, δ)
2:     while hasNext(S) do
3:         X, y ← next(S)
4:         leaf ← sortToLeaf(X, HT)
5:         updateCounts(leaf, X, y)
6:         if data samples at leaf belong to more than one class then
7:             i0, i1 ← attributesWithBestGini(leaf)
8:             if Gini(leaf, i0) - Gini(leaf, i1) > sqrt(R^2 ln(1/δ) / 2n) then
9:                 leaf ← newNode()
10:                for each value j in i0 do
11:                    addLeaf(leaf, i0, j)
```

calculated as in Equation 2.2.

$$Gini(S) = 1 - \sum_{j=1}^{n} p_j^2 \tag{2.2}$$

where n is the number of classes and p_j is the frequency of data samples belonging to class j. The Gini index of an attribute is the difference between the Gini indexes before and after the split. If the difference of the two best indexes is greater than the bound, new leaves are generated. Each leaf stores the counts n_{ijk} of the samples processed with attribute i having value j and class k. The pseudocode of this algorithm is shown in Algorithm 1. S refers to the data stream, HT refers to a single leaf (root) and δ is the bound parameter.

The more interesting characteristics of this algorithm are the constant time learning and the fact that, with enough data samples, it converges to the tree that would be produced by an offline learner.

2.3 Concept Drift

Learning from data streams means being able to accurately model the underlying data distribution but also to be flexible and adapt to changes as fast as possible. The data streams have a temporal nature and therefore their characteristics and distribution can change over time. This characteristic of a data stream is named concept drift [8]. The drift can occur with different speeds, sizes, and severities. As described in [22], a concept drift between time t_0 and time t_1 can be defined by Equation 2.3.

$$\exists\, X : p_{t_0}(X|\, y) \neq p_{t_1}(X|\, y) \tag{2.3}$$

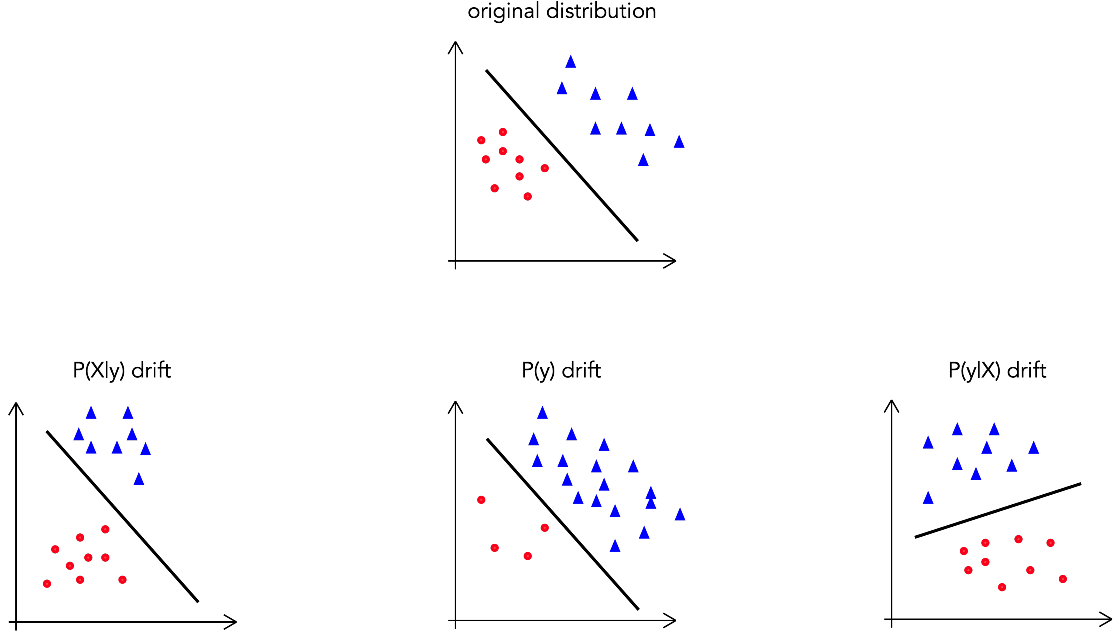


Figure 2.3: An example of the three sources of concept drift.

Where p_{t_0} and p_{t_1} are the joint distribution, at time t_0 and t_1 respectively, between the set of input variables X and the target variable y. Bayes theorem dissects $P(X|\, y)$ into different terms, each one can be a cause of change. The theorem is stated mathematically by Equation 2.4.

$$P(y|\, X) = \frac{P(X|\, y)P(y)}{P(X)} \tag{2.4}$$

Each one of the four probability can change, the resulting drifts are detailed in the following list.

- $P_t(X) \neq P_{t+1}(X)$: in this case, it is possible to see changes in the overall distribution of data and it could also mean that the decision boundary is shifting. Being an independent change from the class labels, it is insufficient to define a concept drift.

- $P_t(X|\, y) \neq P_{t+1}(X|\, y)$: in this case, the probability of seeing a data sample X is changing but its label y is not. It shows that we are seeing new data samples from the same environment and the drift does not affect the decision boundary. An example of his particular drift, known as *virtual concept drift*, is shown in Figure 2.3.

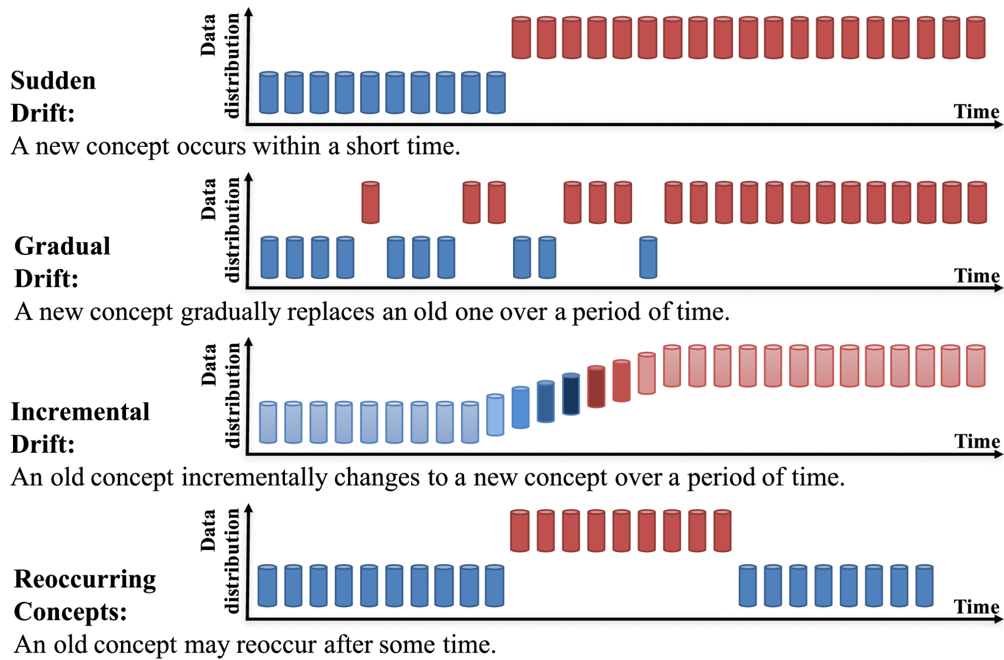


Figure 2.4: Possible temporal characteristics of a drift, src: [1].

- $P_t(y) \neq P_{t+1}(y)$: in this case, the probability of seeing any data example from a particular class y is changing. This will cause the class ratio to change and possibly the switch between minority and majority class. It can affect the performances of the algorithms due to a change in the class imbalance status but It does not necessary shift the decision boundary.
- $P_t(y|\ X) \neq P_{t+1}(y|\ X)$: in this case, the probability of a data example X of belonging to a particular class y is changing. This drift will cause the decision boundary to shift, and, as a consequence, it will lead algorithm's performances to deteriorate. An example of this particular drift, known as *real concept drift*, is shown in Figure 2.3.

2.3.1 Speed of Drift

Concept drifts are also classified for their speed of change [1]. As shown in Figure 2.4, we can classify them as:

- *Sudden*: it occurs when the data distribution changes completely in a few steps to a significantly new one;
- *Gradual probabilistic*: it occurs when the change in the data distribution takes

some samples to complete and, during the shift, the samples come from both the distributions;

- *Gradual continuous or Incremental*: it occurs when the data distribution changes unnoticeable step by step but the accumulated change becomes significant over time;
- *Recurring or Periodic*: it occurs when an already seen data distribution comes back later in time. It can be a great advantage for the algorithm to notice it and adapt accordingly; and
- *Noise*: is a non-informative fluctuation that should not be mistaken for a real drift and it should be filtered in order not to affect the algorithm's performances.

2.3.2 Severity of Drift

Drifts can also differ for their severity [23], meaning the amount of change caused by it. They can be distinguished as *Local* or *Global*.

- *Local*: it occurs when only some areas of the instance space are affected by the drift. Such drifts are more difficult to catch due to the scarcity of data representing the drift.
- *Global*: it occurs when changes affect all or the major part of the instance space. This drifts have a strong effect on the algorithm's performance thus they are easier to catch.

2.3.3 Concept Drift detection

Learning algorithms for classification need to be trained to correctly classify each arriving sample as a function of its feature. Every time this function change, the learners need to detect it and adapt as fast as possible. Drift detection methods can be categorized as either statistical-based, window-based, or ensemble-based [4].

The following list details them.

- **Statistical-based** approach detect concept drifts through monitoring model statistics like the prediction accuracy, correlations between features, or the frequency of class labels. The most famous drift detectors using a statistical method are *CUSUM Test* [24], *Page-Hinckley Test* [25], *Ddm* [26] and *LFR* [27].

- **Window-based** methods, rather than monitoring arriving instances from a stream individually, use one or more sliding windows to monitor statistics. Larger windows can compute the performances with higher accuracy, but they can miss small concept drift within themselves, while smaller windows have a faster concept drift detection. A fundamental problem when designing this type of algorithms is the sizing of the window.
- **Ensemble-based** learners are a group of simple classifiers, called *base learners*, that combine the predictions in order to achieve greater performance. The loss of predictive performance of base classifiers can be exploited to deal with concept drift. An example of such algorithms is Dynamic Weighted Majority (DWM) [28] which updates the weight of the base learners' predictions after each sample, based on their correctness. The algorithm deal with concept drifts adding a new base learner to the ensemble when it makes a wrong prediction and removing base learners having the weight under a threshold.

2.3.3.1 Adaptive Sliding Windows

The Adaptive Sliding Window (ADWIN) [29] algorithm is a drift detector that avoids the sizing problem of window-based algorithms. ADWIN keeps the recently seen data samples in a variable-length window. Every time new data are collected, It computes the average of different "large enough" slices of the window. When an older slice's average differs from the one of the rest of the window of more than a threshold, ADWIN drops the old slice and a concept drift is detected. Therefore, the window has the property that it has the maximal length statistically consistent with the hypothesis "There is no change with the average value inside the window". The only hyper-parameter of this algorithm is the statistical confidence bound for the drift δ. The algorithm compares the averages of two sub-windows W_0 and W_1 with the test presented in Equation 2.5 and decides whether they are likely to come from the same distribution.

$$Test : |\mu_{w_0} - \mu_{w_1}| > \epsilon_{cut} \tag{2.5}$$

The parameter ϵ_{cut} is computed as explained in Equation 2.6.

$$\begin{aligned} m &= \frac{1}{1/n_0 + 1/n1} \\ \epsilon_{cut} &= \sqrt{\frac{1}{2m} * \ln \frac{4n}{\delta}} \end{aligned} \tag{2.6}$$

Computing this test for every "large enough" sub-window can become computationally expensive. A new version of this algorithm called ADWIN2 was proposed

in the same study [29]. The main idea was to compress the window in buckets using a variant of the exponential histogram technique presented in [30]. The memory requirements are reduced to $O(\log W)$ and the worst-case time to process a new data sample is $O(\log^2 W)$.

2.3.3.2 Hoeffding Adaptive Tree

An effective application of the ADWIN technique was proposed to make the Hoeffding Tree able to deal with concept drifts because making new leaves is not enough in order to deal with this challenge.

The new algorithm is called Hoeffding Adaptive Tree [31] and it uses ADWIN2 as a change detector and error estimation. An ADWIN drift-detector is assigned to every node. Each time a change is detected, an alternate tree is created starting from the detector's node. As soon as there is evidence that an existing alternate tree is more accurate, it replaces the sub-tree having as root the starting node.

2.4 Class Imbalance

In real-world scenarios [9], data samples coming from different classes have different frequencies and the ratio of class probabilities can be significantly skewed. The challenge is to make the learner focus its attention on the less represented class which is more difficult to learn but usually more important. This condition is referred to as the class imbalance problem. A dataset is said to be imbalanced when one of the classes, named minority class, is heavily under-represented in comparison to the other, named majority class. This problem is present also in multi-class scenarios in which there are three or more classes. In such cases it is possible to reduce the multi-class classification problem to multiple binary ones.

The following list details the most famous approaches.

- **One vs one**: presented by Hastie et. al. [32], it consist in training a learner for each pair of classes. The number of binary problems will be $K(K-1)/2$ and a new instance is labeled with the class receiving the majority of the votes.

- **One vs all**: as described by Joshi et. al. [33], a binary problem is associated to each class, seen as the positive class, while all the other classes will form the negative class. The number of binary problems will be K and a new instance is labeled with the class having the higher value of the corresponding predictor.

- **Error Correcting Output Codes**: presented by Dietterich et. al. [34] consist in associating a binary code of lenght L to each class. L binary classifiers are trained and the prediction will be a binary code. The prediction is the class having the code most similar to the predicted one.

In this work, I focus on the binary classification problem.

2.4.1 Minority class distributions

Measures to define the level of imbalance are the percentage of minority class data samples and the size ratio between classes. Class imbalance ratio is not the only main difficulty related to class imbalance. The drop in classification performance is also related to another important factor i.e. the minority class distribution in the feature space. This concerns how the minority class samples are located in the instance space with respect to the majority class ones. The study [35] propose 4 different categories of minority class data samples:

- *Safe*: If it is located in regions mostly populated by samples belonging to the same class;
- *Borderline*: If it is placed in the boundary regions between classes, where the samples from both classes overlap;
- *Rare*: If it is placed in regions populated by data samples of a different class; and
- *Outlier*: If it is distant from the corresponding class clusters.

2.4.2 Offline Algorithms to deal with Class Imbalance

The algorithms proposed to solve the imbalance problem in traditional settings are the starting points for the proposed solutions on imbalanced data streams. These solutions can be classified into the three different categories listed below.

- **Data-level methods**: this family of approaches aims at solving the problem by filtering training data to rebalance the class distributions. This can be achieved by oversampling the minority class samples, undersampling the majority class samples, or combining both. The simplest and most popular resampling techniques are random oversampling and random undersampling, where data samples are randomly chosen to be added or removed until a balanced dataset is obtained. There are also more elaborated resampling techniques. The most famous oversampling one is SMOTE [36]. This algorithm consist in generating new minority class samples starting from the

similarities between original ones in the feature space. In order to generate a new synthetic sample the algorithm randomly select a minority class sample and one of its k nearest neighbors. The new sample is placed on the line that connect the selected data points. An important undersampling technique is One-sided selection (OSS) [37], which has two phases: during the first one, it consider all the minority class samples and adds one majority class sample at a time, only if they would be misclassified by a 1-NN until there are not any misclassifications. The second phase consists of removing all the data samples that lie in the border of the two classes. Resampling techniques have been widely tested on real-world applications which have proved their effectiveness. They do not depend on the classifiers thus are more versatile than algorithm-level methods.

- **Algorithm-level methods**: these methods are applied during the algorithm training and they aim at solving the class imbalance problem by penalizing more a prediction error on a minority class data sample. This family of algorithms includes cost-sensitive learning, where the cost of making a wrong prediction on a minority class sample is weighted with the class ratio, and threshold methods, which consist on tuning the threshold used to map probabilities to class labels. For example, in binary classification, instead of the standard 0.5, the algorithm could just decrease the threshold for labeling samples as belonging to the minority class.
- **Ensemble methods**: they become the most used approaches to handling class imbalance. Ensembles are groups of base learners that need to be trained with different datasets in order to have different "points of view" for the prediction. This can be done with techniques such as Bagging [38] or Boosting [39]. Ensembles can be easily integrated with different resampling techniques in order to emphasize the minority class.

2.5 Evaluation Methodology

One of the fundamental tasks in machine learning processes is the model evaluation since it decide which model is the more appropriate to solve the problem at hand. In particular, the evaluation process is composed, firstly, by an error estimation and then by an evaluation of the performance measures.

2.5.1 Error Estimation

In traditional machine learning, with a static training set, usually, the performances of the models are evaluated with a procedure known as cross-validation [18].

However, this split cannot be performed on a data stream where data are unbounded and evolving. The following list explains the main approaches for model evaluation on data streams [40].

- **Holdout:** with this approach, a testing set is created periodically to test the performances of the model. Holdout is useful when data are abundant and testing on all the data would be computationally expensive. Two parameters need to be set: i) the size of the window for testing purposes and ii) the frequency of testing. This evaluation method is fast but it is not suggested to use when concept drift is expected.

- **Interleaved test-then-train:** with this approach, each data sample is firstly used to test the model and then it is used for training. The model performance is being continuously evaluated and all the data samples are used to train the model. This method is more precise than Holdout but more computationally expensive.

- **Prequential:** this approach is similar to the previous one but the recent samples are considered more important, using a sliding window or a decaying factor whose sizes are parameters of the evaluation.

2.5.2 Performance Measures

In binary classification, the instances belong to two different classes which are called, for simplicity, "positive" and "negative". In order to evaluate the performances of a model, the predictions are reported in a table called confusion matrix presented in Figure 2.5. After every evaluation, the instances predicted as positive are labeled with "P" and the instances predicted as negative are labeled with "N". The correct predictions are labeled as "T" and the wrong ones are labeled as "F". The counting of each label combination forms a confusion matrix, where:

- **True Positives (TP)**: samples correctly classified as positives. They belong to the positive class;

- **False Positives (FP)**: samples wrongfully classified as positives. They belong to the negative class;

- **True Negatives (TN)**: samples correctly classified as negatives. They belong to the negative class; and

- **False Negatives (FN)**: samples wrongfully classified as negatives. They belong to the positive class.

TRUE CLASS

	POSITIVE	NEGATIVE
PREDICTED CLASS POSITIVE	TP	FP
PREDICTED CLASS NEGATIVE	FN	TN

Figure 2.5: Confusion Matrix.

Different measures can be computed using this table. *Accuracy* is one of the most used metric in classification problems and it is computed as $\frac{TP+TN}{TP+TN+FP+FN}$.

In data streams, data is evolving and the number of instances per class is changing. *Accuracy* does not take into account this important factor and therefore is not well suited for stream classifiers. Furthermore it does not take into account the class distribution of the dataset [41]. For example, if the instances belonging to the positive class are only 5% of the dataset, a classifier predicting everything as negative will reach a 95% accuracy, but it will miss all the positive instances. Useful measures for data stream classification that take the class imbalance problem into account are *Recall*, *Fscore*, *Gmean* and *Kappa* statistics.

2.5.2.1 Recall and Fscore

To better evaluate the models that focus on class imbalance, it's not possible to base our evaluation metric only on the *accuracy*, but we need measures that focus on each class separately. These measures are the *Recall*, meaning the fraction of instances of a particular class correctly predicted, and the *Precision*, meaning the fraction of predictions of a particular class that are correct.

$$Positive\ Recall\ or\ Sensitivity = \frac{TP}{TP+FN}$$

$$Negative\ Recall\ or\ Specificity = \frac{TN}{TN+FP}$$

$$Positive\ Precision = \frac{TP}{TP+FP}$$

$$Negative\,Precision = \frac{TN}{TN + FN}$$

The combination of these two metrics, *Precision* and *Recall*, is called *Fscore* and it is defined by Equation 2.7 as the harmonic mean of the two.

$$F_\beta = (1 + \beta^2) \times \frac{Recall \times Precision}{Recall + (\beta^2 \times Precision)} \tag{2.7}$$

The parameter β is used to decide if one of the two measures is more important than the other. When $\beta = 1$, this measure is named $F_1 score$ and it weights equivalently the two measures. It is defined by Equation 2.8.

$$F_1 = 2 \times \frac{Recall \times Precision}{Recall + Precision} \tag{2.8}$$

2.5.2.2 Gmean

A measure that takes into account the performances of the model on both the two classes was presented by Kubat et. al. [42]. It is called *Gmean* and it measures the balance between classification performances on the majority and minority classes. It is defined by Equation 2.9 as the geometric mean of the recall on each class.

$$Gmean = \sqrt{Sensitivity \times Specificity} \tag{2.9}$$

2.5.2.3 Kappa statistics

The original Kappa statistic, also called Cohen's Kappa [43], is based on the accuracy score, but it re-scales it with respect to the accuracy that would have happened through random predictions. It is defined by Equation 2.10.

$$\kappa = \frac{p_0 - p_r}{1 - p_r} \tag{2.10}$$

p_0 is the prequential accuracy of the classifier, and p_r is the probability to make a correct prediction of a random classifier, one that assigns to each class the same number of samples. In case of very imbalanced class distributions the Kappa M statistic κ_m, proposed by Bifet [44] can perform better and it is defined as in Equation 2.11:

$$\kappa_m = \frac{p_0 - p_m}{1 - p_m} \tag{2.11}$$

where p_m is the probability to make a correct prediction of a majority class classifier, one that assigns to the majority class all the new samples. This type of measure fails on evaluating change detectors with data streams that have temporal dependence on the class distributions. Another version of the kappa statistics

has been evaluated in order to exclude any inaccuracy in the previous metrics and take this type of drift into account, the Kappa-Temporal statistics [45]. This metric considers the presence of temporal dependencies in data streams and it is computed as in Equation 2.12.

$$\kappa_{temp} = \frac{p_0 - p_{nc}}{1 - p_{nc}} \tag{2.12}$$

where p_{nc} is the probability to make a correct prediction of a no-change classifier, one that assigns to the samples the class label observed on the sample before.

The κ_{temp} and κ_m measure different aspects of the performance and they can be used jointly.

Chapter 3

Problem Setting

This chapter presents the problem that this thesis aims to solve. In particular, Section 3.1 details the existing solutions to the class imbalance problem in streaming scenarios, presenting each algorithm's pseudocode. Section 3.2 describes why there is a need for a benchmarking environment for the class imbalance and concept drift solutions.

3.1 SML Algorithms to deal with Class Imbalance and Concept Drift

The algorithms proposed to solve the imbalance problem in online scenario refer to the respective offline scenario solutions. The studied approaches are Data-level, based on SMOTE [36], and Ensembles, based on the online bagging technique [46]. In the following sections I present the pseudocode of the algorithms.

3.1.1 C-SMOTE

C-SMOTE [47] is an online version of SMOTE [36] where the minority samples are collected in a window managed by ADWIN [29]. The algorithm keeps 4 counters: S_0 and S_1 count the arriving samples of each class, while $S_{\underline{0}}$ and $S_{\underline{1}}$ count the generated synthetic samples of each class. To rebalance the data stream, the algorithm uses two windows, one called W which keeps the samples related to the current concept and one called W_{label} which keeps the corresponding labels. Every time a new sample arrives, the class ratio is checked and, if it is less than a certain threshold t, an online SMOTE version is applied until the minority sample ratio is greater than the threshold. In order to generate the synthetic samples, the real ones are selected randomly and only once for each rebalance phase. A

Algorithm 2 Pseudocode of C-SMOTE

```
1: function C-SMOTE(minSizeMinority, learner, t, S)
2:     W, W_label ← ∅
3:     S_0, S_1, S_0̲, S_1̲ ← 0
4:     S_gen̲ ← ∅
5:     adwin ← ∅
6:     while hasNext(S) do
7:         X, y ← next(S)
8:         train(X, y, learner)
9:         W_label ← add(y)
10:        W ← add(X, y)
11:        updateCounters(y, S_0, S_1)
12:        adwin ← add(y)
13:        if getChange(adwin) then
14:            resizeWindows(adwin, W, W_label, S_0, S_1, S_0̲, S_1̲, S_gen̲)
15:        W_min, S_min, S_min̲ ← selectMinorityClass(W, W_label, S_0, S_1, S_0̲, S_1̲)
16:        W_maj, S_maj, S_maj̲ ← selectMajorityClass(W, W_label, S_0, S_1, S_0̲, S_1̲)
17:        if checkMinSize(minSizeMinority, S_min) then
18:            imbalanceRatio ← ratio(S_min, S_maj, S_min̲, S_maj̲)
19:            while t > imbalanceRatio do
20:                X̂, ŷ ← newSample(W_min, S_gen̲)
21:                S_min̲ ← S_min̲ + 1
22:                train(X̂, ŷ, learner)
23:                imbalanceRatio ← ratio(S_min, S_maj, S_min̲, S_maj̲)
```

sample will be selected more than once only when the amount of rebalance needed is greater than the size of the minority window. The count of artificially generated samples is done using an array called S_{gen} which, for each sample in W, counts how many times it is used to introduce synthetic samples. When ADWIN detects a change, the two windows are resized and all the counters updated. The complete pseudocode is presented in Algorithm 2. l refer to the base learner, t refers to the class ratio to achieve and $minSizeMinority$ refers to the minimum number of minority samples in the window to start the rebalancing procedure.

3.1.2 RebalanceStream

RebalanceStream is another data-level solution based on SMOTE [48]. This algorithm uses ADWIN and multiple models to deal with class imbalance.
It starts with a single base learner. Incoming data are collected in a batch and

Algorithm 3 Pseudocode of RebalanceStream

```
1: function RebalanceStream(S, learner)
2:     adwin, confusionMatrix ← ∅
3:     batch, resetBatch ← ∅
4:     while hasNext(S) do
5:         X, y ← next(S)
6:         eval ← prequentialEvaluation(learner, X, y)
7:         confusionMatrix ← add(eval)
8:         adwin ← add(y)
9:         train(X, y, learner)
10:        batch ← add(X, y)
11:        if getWarning(adwin) then
12:            w ← true
13:        if w == true then
14:            resetBatch ← add(X, y)
15:        if getChange(adwin) then
16:            kStatLearner ← k-statistics based on confusionMatrix
17:            kStatBal, lBal ← trainLearner(batch, True)
18:            kStatReset, lReset ← trainLearner(resetBatch, False)
19:            kStatResetBal, lResetBal ← trainLearner(resetBatch, True)
20:            max ← max(kStatLearner,kStatBal,kStatReset,kStatResetBal)
21:            learner ← model having max
22:            confusionMatrix ← confusionMatrix of model having max
23:            Reset other models, batch and resetBatch
24:            w ← false
```

the corresponding class labels are collected in a window managed by ADWIN. When it detects a drift warning, the algorithm starts collecting samples in a new batch called reset-Batch. When ADWIN confirms the change, three new learners are trained in parallel: i) one only with the reset-Batch, ii) one with the reset-Batch balanced with SMOTE, and iii) one with the original Batch rebalanced with SMOTE. The one with the better k-statistic is chosen to be the new learner and will replace the active model. Then, the other models and both the Batch and the reset-Batch are resetted. The pseudocode of the main algorithm is presented in Algorithm 3. The procedure to train the three temporary learners is detailed in Algorithm 4.

Algorithm 4 Pseudocode of the RebalanceStream new learners training

```
1: function TRAINLEARNER(batch, rebalance)
2:     newLearner ← newClassifier()
3:     confusionMatrix ← ∅
4:     if rebalance then
5:         batch ← SMOTE(batch)
6:     for each (X, y) ∈ batch do
7:         eval ← prequentialEvaluation(newLearner, X, y)
8:         confusionMatrix ← add(eval)
9:         train(X, y, newLearner)
10:    kStat ← k-statistics based on confusionMatrix
11:    return kStat,newLearner
```

3.1.3 Online Bagging

Bagging [38] is one of the most used machine learning technique to build ensembles. It consists on training each base learner with a different dataset with the same size as the original one, but created by sampling with replacement from the original training set.

In offline Bagging, each base model is trained with a data set containing K copies of each of the original training data sample where K is extracted from a binomial distribution as detailed in Equation 3.1.

$$P(K = k) = \binom{N}{k}(\frac{1}{N})^k(1 - \frac{1}{N})^{N-k} \tag{3.1}$$

An online version has been proposed in [46] and It is based on the fact that a data stream is an infinite size dataset.

Applying the limit as $N \to \infty$ to the above binomial distribution will result in a Poisson distribution with lambda equal to 1 as detailed in Equation 3.2.

$$\begin{aligned} P(K = k) = \lim_{N\to\infty} \frac{N!}{k!(N-k)!}(\frac{1}{N})^k(1 - \frac{1}{N})^{N-k} \\ \lim_{N\to\infty} \frac{1}{k!}\frac{N!}{(N-k)!}(\frac{1}{N})^k = \frac{1}{k!} \\ \lim_{N\to\infty}(1 - \frac{1}{N})^N = e^{-1} \\ \lim_{N\to\infty}(1 - \frac{1}{N})^{-k} = 1 \\ P(K = k) = \frac{e^{-1}}{k!} \end{aligned} \tag{3.2}$$

In Online Bagging each base learner is trained with each data sample k times, with k drawn from a $Poisson(1)$ distribution.

3.1.3.1 Native OOB and UOB

Wang et. al. [49] proposed a solution to the problem of class imbalance based on online bagging. The idea is to make an ensemble of base learners where, for each one of them, the classes are balanced. This is achieved by adapting the lambda of the Poisson distribution based on each sample's class.

The class balancing can be done in two ways: undersampling the majority class or oversampling the minority class. In the original Oversampling Online Bagging (OOB) and Undersampling Online Bagging (UOB) there are two important time decaying variables for each class: w_k which denotes the size percentage of class k and R_k which denotes the accuracy of the model on class k.

When two classes have the w_k difference greater than a threshold δ_1 ($0 < \delta_1 < 1$) and the $R(k)$ difference greater than a threshold δ_2 ($0 < \delta_2 < 1$), the small class is labeled as the minority and the large class is labeled as majority. After comparing all the classes, the unlabeled ones are treated as normal. This procedure leads to three label sets: minority class set Y_{min}, majority class set Y_{maj} and normal class set Y_{nor}.

OOB will update each learner one time if the sample is from a majority class, otherwise, the number of updates will be chosen from a Poisson distribution with lambda $\lambda = 1/w_k$.

UOB instead will update each learner 1 time if the sample is from a minority class, otherwise, the number of updates will be chosen from a Poisson distribution with lambda $\lambda = 1 - w_k$.

The pseudocode is presented in Algorithm 5, the input variable *Over* is *True* to denote the oversampling version and it is *False* to denote the undersampling version.

The main advantages of these Online Bagging based algorithms are:

1. Since resampling is algorithm-independent, it allows any type of online classifier to be used;
2. Time-decayed class size used in OOB and UOB dynamically estimates imbalance status without storing old data or using windows;
3. Being ensembles of classifiers, they are expected to be more accurate than a single classifier.

Algorithm 5 Pseudocode of Online Bagging methods

```
1: function OnlineBagging(S,Ensemble,Over)
2:     Y_min, Y_maj, Y_nor ← ∅
3:     Sizes ← ∅
4:     Recalls ← ∅
5:     while hasNext(S) do
6:         X, y ← next(S)
7:         eval ← prequentialEvaluation(Ensemble, X, y)
8:         Recalls ← updateRecalls(Recalls, eval, y)
9:         Sizes ← updateSizes(Sizes, y)
10:        Y_min, Y_maj, Y_nor ← updateClassSets(Sizes, Recalls)
11:        w_k ← Sizes[y]
12:        if y ∈ Y_min and Over then
13:            λ ← 1/w_k
14:        else
15:            if y ∈ Y_maj and not Over then
16:                λ ← 1 − w_k
17:            else
18:                λ ← 1
19:        for each learner in Ensemble do
20:            K ← Poisson(λ)
21:            train(X, y, learner) K times
```

3.1.3.2 Improved Online Bagging

These algorithms have been proposed by a more recent study [50] of OOB and UOB focusing on the two class problem. The original online bagging algorithms compute lambda with a formula that takes into account only the class size of the class the current data sample belongs to. This can be a problem in the case of $P(y)$ drift when the classes become balanced and λ will not be equal to 1 for both classes. Also, w_k depends on the number of classes and so will λ, this makes λ greater when there are more classes but with the same imbalance rate.

The paper proposes a new method to set λ based on the size ratio of the two classes. The pseudocode is presented in Algorithm 6. Considering only two classes, and naming w_{maj} the size of the majority class and w_{min} the size of the minority class, λ will be set as follow:

- The improved OOB will set it to w_{maj}/w_{min} for the minority class and 1 for the majority class;

Algorithm 6 Pseudocode of Improved Online Bagging methods

```
1: function ImprovedOnlineBagging(S,Ensemble,Over)
2:     Sizes ← ∅
3:     while hasNext(S) do
4:         X, y ← next(S)
5:         Sizes ← updateSizes(Sizes, y)
6:         w_k ← Sizes[y]
7:         if Over then
8:             w_max ← max(Sizes)
9:             λ ← w_max/w_k
10:        else
11:            w_min ← min(Sizes)
12:            λ ← w_min/w_k
13:        for each learner in Ensemble do
14:            K ← Poisson(λ)
15:            train(X, y, learner) K times
```

- The improved UOB will set it to w_{min}/w_{maj} for the majority class and 1 for the minority class.

The same paper which proposed the improved versions also presented two ensemble strategies to combine the strength of OOB and UOB. These ensembles need a new parameter, called Smoothed Recall. It is a moving average of the recall of each class to smooth out its short-term fluctuations. To weight the predictions of the OOB and UOB, their Gmean values are computed using their Smoothed Recalls. WEOB1 uses the normalized Gmean values of OOB and UOB as their weights to compute a weighted sum of their predictions, while WEOB2 compares the Gmean values and uses only the prediction of the model with the higher one.

3.1.4 Ensemble of Online Sequential Extreme Learning Machine

This very interesting algorithm has been proposed in [51] and it is a bagging ensemble of OS-ELM [52] networks plus a WELM [53] which act as long-term memory to deal with recurrent concepts. In this section, I will use small letters (eg. g) to denote scalar values (eg. $\boldsymbol{w}$), bold letters to denote arrays, and bold capital letters to denote matrices (eg. $\boldsymbol{H}$).

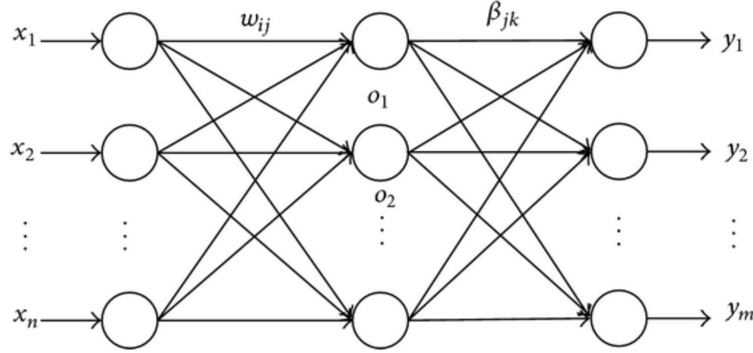


Figure 3.1: The structure of a Single hidden-Layer Feed-forward Network, src: [2].

3.1.4.1 ELM

Extreme learning machine (ELM) [54] is a single-step learning algorithm for single hidden-layer feed-forward network (SLFN). An example of these networks is shown in Figure 3.1. A standard SLNF with L hidden nodes, N samples $(\boldsymbol{x}_i, \boldsymbol{y}_i)$, and activation function $g(\boldsymbol{x})$ can be modeled as in Equation 3.3.

$$\begin{gathered}\sum_{i=1}^{L} \beta_i g_i(\boldsymbol{w}_i \cdot \boldsymbol{x}_j + b_i) = \boldsymbol{y}_j \\ j = 1, ..., N\end{gathered} \tag{3.3}$$

In order to train the SLFN the goal is to find the values $\boldsymbol{w_i}$, b_i, β_i which minimize the cost function defined by Equation 3.4.

$$E = \sum_{N}^{j=1} (\sum_{L}^{i=1} \beta_i g(\boldsymbol{w}_i \cdot \boldsymbol{x}_j + b_i) - \boldsymbol{y}_j)^2 \tag{3.4}$$

The input weights $\boldsymbol{w}$ and biases b connecting input layer to the hidden layer (hidden node parameters) are assigned randomly, computing the activation matrix $\boldsymbol{H}$ with size $(N \times L)$.

Then, a solution can be computed with a Moore-Penrose generalized matrix inversion as in Equation 3.5

$$\begin{gathered}\boldsymbol{H}^{\dagger} = (\boldsymbol{H}^T \boldsymbol{H})^{-1} \boldsymbol{H}^T \\ \boldsymbol{\beta} = \boldsymbol{H}^{\dagger} \boldsymbol{T}\end{gathered} \tag{3.5}$$

The solution has the property of being a least-square solution of the linear system $\boldsymbol{H\beta} = \boldsymbol{T}$, more specifically it is the unique least square solution with the smallest norm.

3.1.4.2 OS-ELM

The online sequential version of this algorithm, called OS-ELM, has been proposed in [52] and it updates the ELM with data chunks. The algorithm works following these phases:

1. Initialization phase: the learner need to be initialized with a chunk of data, assigning random input weight, bias and computing the matrix $\boldsymbol{H_0}$ and the output weights $\boldsymbol{\beta_0}$ as a standard ELM. The matrices $\boldsymbol{P_0} = (\boldsymbol{H_0^T H_0})^{-1}$ and $\boldsymbol{T_0} = \left[\boldsymbol{y_1}, ..., \boldsymbol{y_{N_0}}\right]$ are initialized. The chunk counter k is set to 0;

2. Sequential learning phase: when a new chunk arrives,the counter k is incremented and the matrices $\boldsymbol{H_k}$ and $\boldsymbol{T_k}$ are computed.
   The output weights $\boldsymbol{\beta_k}$ are updated using Equation 3.6.

$$\begin{aligned} \boldsymbol{P_k} &= \boldsymbol{P_{k-1}} - \boldsymbol{P_{k-1}H_k^T}(\boldsymbol{I} + \boldsymbol{H_k P_{k-1} H_k^T})^{-1}\boldsymbol{H_k P_{k-1}} \\ \boldsymbol{\beta_k} &= \boldsymbol{\beta_{k-1}} + \boldsymbol{P_k H_k^T}(\boldsymbol{T_k} - \boldsymbol{H\beta_{k-1}}) \end{aligned} \tag{3.6}$$

Weighted extreme learning machine (WELM) has been proposed along with its online version WOS-ELM [53] to make the ELM address the class imbalance with a cost-sensitive approach. They correct the $\boldsymbol{\beta}$ computation using a matrix $\boldsymbol{W} = \boldsymbol{W_-} + \boldsymbol{W_+}$ where $\boldsymbol{W_-} = diag(1/m^-, ..., 1/m^-, 0, ..., 0)$ and $\boldsymbol{W_+} = diag(0, ..., 0, 1/m^+, ..., 1/m^+)$, m^- and m^+ are the sizes of the negative and positive class. The samples are ordered with respect to their class, first all the negative class samples and then the positive class ones. This matrix is applied in the WOS-ELM equations as shown in Equation 3.7.

$$\begin{aligned} \boldsymbol{P_0} &= (\boldsymbol{H_0^T W_0 H_0})^{-1} \\ \boldsymbol{\beta_0} &= \boldsymbol{P_0 H_0^T W_0 T_0} \\ \boldsymbol{P_k} &= \boldsymbol{P_{k-1}} - \boldsymbol{P_{k-1}H_k^T}(\boldsymbol{W_k} + \boldsymbol{H_k P_{k-1} H_k^T})^{-1}\boldsymbol{H_k P_{k-1}} \\ \boldsymbol{\beta_k} &= \boldsymbol{\beta_{k-1}} + \boldsymbol{P_k H_k^T W_k}(\boldsymbol{T_k} - \boldsymbol{H\beta_{k-1}}) \end{aligned} \tag{3.7}$$

The main drawback of WOS-ELM is that it is suited only for stationary data streams and it cannot handle concept drift learning.

3.1.4.3 ESOS-ELM

Ensemble of Subset Online Sequential Extreme Learning Machine (ESOS-ELM) [51], has been proposed as an OS-ELM based ensemble able to handle both the class imbalance and the concept-drift problems. The idea is to keep two sorted lists of the classifiers, one ordered with respect to the number of minority data samples processed and one ordered with respect to the number of majority data samples processed. When a new chunk of data is ready for the training, the imbalance ratio m is computed, the first m classifiers of the first list will process all the minority class data samples, while the top m classifiers of the second list will process $1/m$ of the majority class data samples each. Every learner will have a balanced training chunk. In cases in which m is greater than the number of learners in the ensemble, only the misclassified majority data samples will be kept. In order to handle concept drift, the Dynamic Weighted Majority technique is used. When the ensemble misclassifies a sample, a new OS-ELM learner is added and it will be initialized using the WELM initialization. A long-term buffer, called ELM-Store, is used to save past concepts. When a concept drift is detected from the Gmean of the ensemble falling under a threshold θ, a WELM is trained and stored. When a stored classifier performs better than the main ensemble it is introduced with weight 1. This store has a limited resource requirement because the matrices to store are only dependent on the number of neurons and the matrix inversion is done only one time when the drift is detected, also the number of WELMs to store is proportional to the number of drifts detected and not to the number of samples processed. The pseudocode of the algorithm using batch size equal to one is presented in Algorithm 7, M refers to the ensemble size, p refers to the number of time step between new classifiers addition, θ refers to the threshold for change detection and n_0^- and n_0^+ are the class sizes in the initialization set.

3.2 Problem Statement

After studying the problems of class imbalance and concept drift in data streams and researching for solutions, I felt the lack of a well-defined environment to compare the proposed algorithms. The main difficulties to replicate the experiments concerned the data streams generation and the evaluation procedure.

3.2.1 Benchmarking environment

The studies presenting the algorithms did not report a common well-defined benchmarking of their experiments. According to [55], a benchmark needs to measure all the important features, using broadly accepted and easy understandable metrics.

Algorithm 7 Pseudocode of ESOS-ELM

```
1: function ESOSELM(M, p, θ, β_u, n_0^+, n_0^-, S)
2:     ir_0 ← floor(n_0^- / n_0^+), c ← floor(n_0^- / M)
3:     a ← 1, b ← c, d ← 1, e ← c, count ← 0
4:     for each u in 1...M do
5:         OS-ELM ← Initialize using β_u = (x_a^-, ..., x_b^-, x_d^+, ..., x_e^+)
6:         Ensemble ← add(OS-ELM)
7:         w_u ← 1
8:         a ← a + c, b ← b + c
9:         count ← count + 1
10:        if count = ir_0 then
11:            d ← d + c, e ← e + c, count ← 0
12:    initialize ELM-store
13:    while hasNext(S) do
14:        Index_p ← sortAscend(Ensemble, +)      ▷ sort with respect to positive class
15:        Index_n ← sortAscend(Ensemble, −)      ▷ sort with respect to negative class
16:        X, y_i ← next(S)
17:        O ← Classify(Ensemble, X)
18:        if ChangeDetectionTest(θ) then                          ▷ Change detected
19:            train(ELM-Store, WELM)
20:            initialize ELM-store
21:        else
22:            ELM-store ← add(X, y_i)
23:        if y_i = "+" then      ▷ minority class sample processed by ir n. of classifiers
24:            for each u in 1, ..., ir with index_p order do
25:                o ← Classify_u(X)
26:                WeightUpdate(o, y_i, p)
27:                train(X, y, OS-ELM_u)
28:        else                   ▷ majority class sample processed by single classifier
29:            u ← index_n(0)
30:            o ← Classify(OS-ELM_u, X)
31:            WeightUpdate(o, y_i, p)
32:            train(X, y, OS-ELM_u)
33:        if O ≠ y_i and i mod p = 0 then
34:            OS-ELM ← initialize OS-ELM with Equation 3.7
35:            Ensemble ← add(OS-ELM)
36:            M ← M + 1
37:            w_{M−1} ← 1
```

All systems need to be fairly compared. Moreover, the experiments needs to be cost-effective and easy repeatable.

Important measures when comparing algorithms' performances are the resource requirements, both time and memory. The algorithms were not tested from this point of view on any study. A problem I faced with MOA was the retrieval of the exact memory requirements of the experiments. MOA is developed in java thus it executes in a Java Virtual Machine. This makes the single process' resources hidden. In order to solve this problem I run each experiment in a different Docker[1] container as detailed in Section 4.1.2.

I present the developed environment used to run the experiments in Section 5.3.

3.2.2 Replication Study

Most of the algorithm implementations were not directly available in MOA. The online bagging based algorithms still needed an implementation in this framework. The implementation is detailed in Section 5.2. C-SMOTE and RebalanceStream were already available in MOA[2], while ESOS-ELM was implemented in an independent repository[3]. Furthermore, a problem faced to replicate the experiments was understanding all the details and all the settings used during the tests. The configuration settings are not all entirely described in the papers. I define all the parameters used during the experiments in Section 5.4.

3.2.3 Data streams

A problem that I met when I tried to replicate the experiments exactly as described was that both artificial and real data streams were not well defined and difficult to reproduce or retrieve. Each study uses different data streams as detailed in the following list.

- *Native online bagging* [49]: The only data stream used to test the algorithms was composed of randomly extracted samples from a real dataset manually setting the imbalance ratios. The dataset was the one used in the 2009 fault detection competition of the PHM society [56] which is not available anymore from the official website. Moreover, the data streams' lenght was relatively short, limited to 1000 steps, with the drift happening at the $500th$ step. The drifts on all these data streams were only of the $P(y)$ kind.
- *Improved online bagging* [50]: These algorithms have been tested with artificial data streams generated from a multivariate Gaussian distribution. The

[1]https://www.docker.com/

[2]https://github.com/Waikato/moa

[3]https://github.com/dabrze/imbalanced-stream-generator

parameters of this distribution are not specified and the data streams are not available. Here too, the data streams' lenght is 1000 steps, with the drift happening at the $500th$ step. These algorithms have also been tested with two real datasets, one was the same unavailable dataset of the previous study [49]. The drifts on all these data streams were only of the $P(y)$ kind.

- *RebalanceStream* [48]: This algorithm was tested on artificial data streams created with only one generator. It was not tested on any real data stream. Also here, only $P(y)$ drifts has been considered.
- *C-SMOTE* [47]: This study tested both C-SMOTE and RebalanceStream on numerous real data streams but It did not defined the drifts happening on these streams.
- *ESOS-ELM* [51]: This study was the most detailed from a data stream point of view. It was the only one not restricted to $P(y)$ drifts.

In Section 4.2, I define a precise way to generate each one of the data streams used for testing the algorithms in order to make it possible to exactly reproduce the experiments.

3.2.4 Performances evaluation

The studies regarding the class imbalance problem lack a standard way of evaluating the performances. Both the online bagging studies [49] [50] reported the Gmean evaluation only at specific time-steps; while the ESOS-ELM study [51] reported evaluations based only on the Gmean measure with a lack of class-specific measures. This study is the only one using a Holdout evaluation and not a prequential evaluation. The testing sets are extracted with different strategies for each data stream. Furthermore, no study reported the resource requirements to run the experiments. In Section 5.4, I describe the evaluators' settings used in the experiments and in Chapter 6 I compare the results using both Gmean and class-specific. In Section 6.4 I report the different resource requirements of each algorithm.

3.2.5 Problem Recap

To sum up, the problems faced during the progress of the thesis are:

1. Implement all the algorithms with the MOA framework;
2. Retrieve all the settings used on the studies;

3. Define a set of data streams which is complete w.r.t concept drifts and imbalance levels. It need to be composed of both real and artificial data streams;

4. Define an evaluation strategy in order to fairly evaluate the algorithms during the artificial and real data streams.

Chapter 4

Problem Solving

This chapter presents the proposed approaches to the problem detailed in the previous Chapter. In particular, Section 4.1 presents the technologies I used to conduct my research, while Section 4.2 details the characteristics of the data streams used during the experiments

4.1 Technologies adopted

The technologies I choose to build the environment and run the experiments were different for each phase. I used MOA [3] to implement and test the algorithms, Docker and InfluxDB to collect the information about the memory requirements and Tableau to build an effective visualization.

4.1.1 Implementing and running the algorithms

I implemented the algorithms and run the experiments with MOA[1]. Massive Online Analysis is an open-source software library developed by the University of Waikato. It implements numerous SML techniques divided for the respective tasks (Classification, Regression, Clustering, Recommender systems, Pattern mining).

It is written in Java which gives it large portability and compatibility with a lot of well-supported libraries. The experiments are called *Tasks*. They are defined by a string and they can be set and run through a Graphical User Interface (GUI) or a Command Line Interface (CLI). The MOA workflow starts with choosing a data stream, which can be generated or read from a file, a learner, for example, a classifier, and an evaluation method.

[1] `https://moa.cms.waikato.ac.nz/`

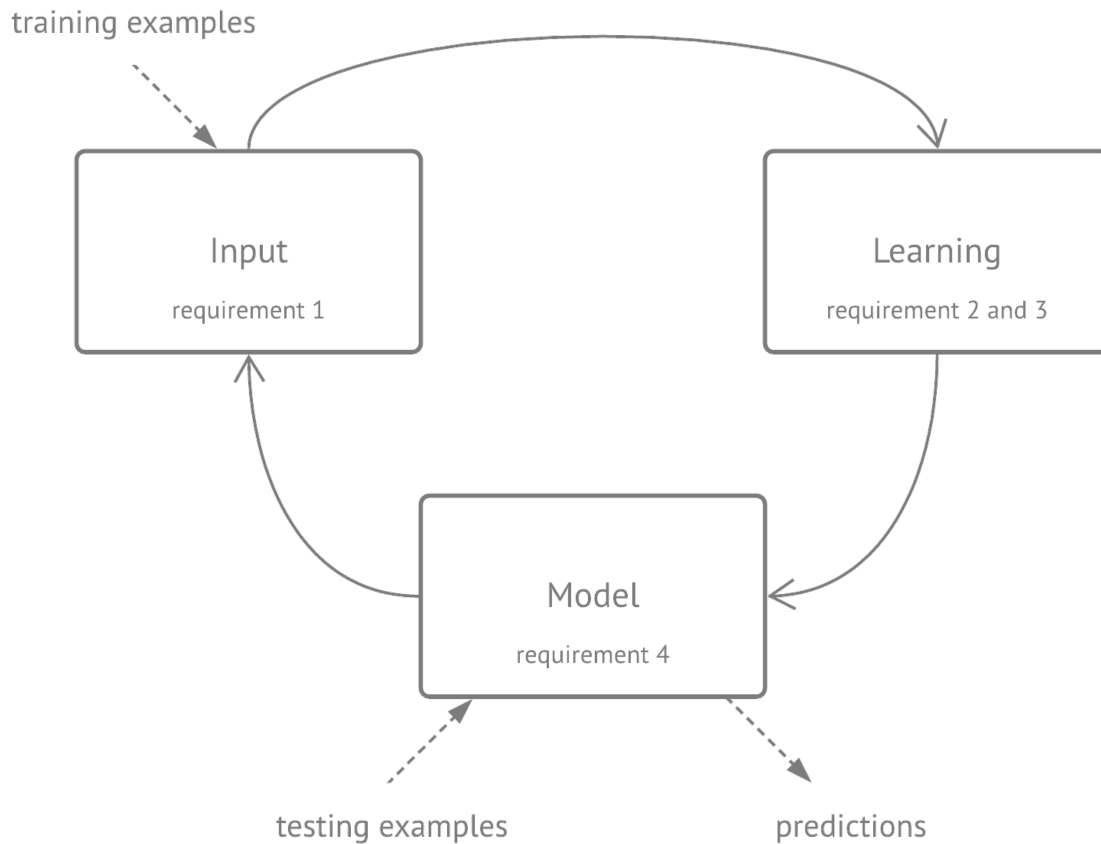


Figure 4.1: Data stream classification cycle, src: [3].

The learning cycle follows three fundamental steps which are illustrated in Figure 4.1:

- *Input*: the next example of the stream is passed to the algorithm, one at a time (requirement 1);
- *Learning*: the algorithm processes the new example and updates the model as fast as possible (requirement 2) and with a limited amount of memory (requirement 3); and
- *Model*: the algorithm can supply a model which is ready to predict the class of the next unseen example (requirement 4).

4.1.2 Tracking memory requirements

A problem faced with MOA was the retrieval of the memory requirements of the running algorithms because MOA executes in a Java Virtual Machine. It loads, verifies and executes the code and manages the memory of any java process making them indistinguishable. The solution I adopted was running each experiment in a different docker container and monitor the memory requirements with InfluxDB. A Docker container[2] is a unit of software that incubates a process using a predefined system configuration. It allows to packages up code and all its dependencies so that the application can runs quickly and reliably from one computing environment to another. InfluxDB[3] is an open-source data store for time-stamped data, running docker with InfluxDB allows monitoring the resource requirements of each container thus the ones of each experiment.

4.1.3 Visualizing the results

In order to build an effective visualization of the results and compare the algorithms, I used Tableau[4]. It is a data analytic and visualization tool widely used in the industry today. It has a graphical drag and drop interface where it easy to build different types of plots aggregating data in different dimensions.

4.2 Data streams

I tested the algorithms with different drifts, class imbalance levels, and data distributions but always with two classes. For convenience, in all data streams, the *minority* class is always the class *1*, while the *majority* one is the class *0*.

The artificial data streams generated with the SEA [57] and SINE1 [26] generators have the three different types of drift mentioned in Section 2.3 with gradual and sudden speeds and four imbalance ratios.

Nine data streams, each with a different drift, have been generated with a cluster generator. It generates the minority class distribution with a cluster shape and the drifts consist on making the clusters appear, move, change the shape and split. These data streams have been generated with incremental, sudden and periodic speeds with four imbalance ratios. Every artificial data stream has $100,000$ instances. The gradual, incremental, and periodic drifts start at the time step $45,000$ and takes $10,000$ time steps to complete, while the sudden drift happens at time step $50,000$.

[2] `https://www.docker.com/`
[3] `https://www.influxdata.com/`
[4] `https://public.tableau.com/`

The last tests have been performed on the real data streams PAKDD'09 [58], KDDCup'99 [59] and Elec [26].

4.2.1 SEA and SINE1

In order to reproduce each of the different types of concept drifts shown in Section 2.3, I choose two widely used artificial data generators: SINE1 [26] and SEA [57]. SINE1 instances are composed of two attributes (x_1, x_2) uniformly distributed in $[0, 1]$. The class is determined by $x_2 - \sin x_1 \leqslant \theta$, where θ is a threshold value. SEA instances, instead, are composed by three attributes (x_1, x_2, x_3) uniformly distributed in $[0, 10]$. Only two of them are used to compute the label, while the third one is just noise. The equation to determine the class label is $x_1 + x_2 \leqslant \theta$, where θ is a threshold value. The concept drifts generated are explained in the followings list.

- $p(y)$ *Concept Drift*: these streams involve only a the $p(y)$ type of concept drift (see Figure 2.3), without $p(X|y)$ and $p(y|X)$ changes. Data streams generated by SINE1 have a severe class imbalance change, in which the imbalance ratio of the first half is reversed on the second half. Data streams generated by SEA have a less severe change, in which the data streams are balanced during the first half and become imbalanced during the latter half. In the gradual drifting cases, $p(y)$ is changed linearly during the concept transition period (time step $45,000$ to time step $55,000$).

- $p(X|y)$ *Concept Drift*: these streams focus on the $p(X|y)$ type of concept drift (see Figure 2.3), without $p(y)$ and $p(y|X)$ changes. The data stream is constantly imbalanced. The concept drift in each data stream is defined by a change on a constraint on the x_1 parameter for the negative class (0) instances. During the first half of the stream the probability is $p(x_1 < n) = 0.9$ while during the second half, it is $p(x_1 < n) = 0.1$. In the gradual drifting cases, it is changed linearly during the concept transition period.

- $p(y|X)$ *Concept Drift*: these streams focus on the $p(y|X)$ type of concept drift (see Figure 2.3), without $p(y)$ and $p(X|y)$ changes. The data stream is constantly imbalanced. Data streams generated by SINE1 have a concept swap, while data streams generated by SEA have a concept drift due to a θ value change making it is less severe than the change in SINE1 because some of the data samples from the old concept are still valid under the new concept after the threshold moves completely.

Sixteen data streams have been generated for every concept drift. Eight of them have been generated with SEA and eight with SINE1. Each one has a different

combination of imbalance ratio, *1:9, 2:8, 3:7,* or *4:6*, and speed, *sudden* or *gradual*. I also classified the distribution of the minority class in each dataset into *safe*, *borderline*, *rare* and *outlier*, following [35]. This study proposes to assess the type of a sample by analysing its local neighbourhood. For each minority samples, its five nearest neighbours samples has been analyzed. The proportion of neighbours from the same class against neighbours from the opposite class can range from *5:0* (all neighbours are from the same class as the analysed data sample) to *0:5* (all neighbours belong to the opposite class). Depending on this proportion, the labels are assigned to the examples in the following way:

- *5:0* or *4:1*: the sample is labelled as *safe*;
- *3:2* or *2:3*: the sample is labelled as *borderline*;
- *1:4*: the sample is labelled as *rare*; and
- *0:5*: the sample is labelled as *outlier*.

All the datasets characteristics are detailed in Table 4.1.

4.2.2 Cluster generator

This stream generator was originally implemented by Dariusz Brzeziński [5] and it is composed by an imbalance generator that manages how the data are distributed in the feature space and a drift generator which manages how to perform the drifts from one distribution to the next one. The main characteristic of the imbalance generator is the data distributions of the two classes. The *majority* class data points spread over all the feature space except in the areas where the *minority* class clusters are located. These clusters can vary in number, shape, and size. After deciding the location of the centroid of each cluster, the minority samples are randomly extracted from four regions of the feature space:

- *Safe zone*: this is the region near the centroid of the cluster where there is not any majority class example;
- *Borderline zone*: this is a zone defined by a radius bigger than the one of the safe zone and there can be examples from both classes;
- *Outlier zone*: a point resides in this zone if it is at least far from the center of every cluster by double of each the borderline radius;
- *Rare zone*: this is a zone composed of minority class examples distant from the core of the minority class that form small groups of two-three samples.

[5] `https://github.com/dabrze/imbalanced-stream-generator`

Table 4.1: The characteristics of the artificial data streams used in the experiments.

Type	Data	Speed	Min Class Type		Class +1		Class 0	
			Before CD	After CD	Old Concept	New Concept	Old Concept	New Concept
$p(y)$	SINE1	Sudden	Safe	Safe	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 \geq 0$	$x_2 - \sin x_1 \geq 0$
	SINE1_g	Gradual	Safe	Safe	$p(y) = 0.1$	$p(y) = 0.9$	$p(y) = 0.9$	$p(y) = 0.1$
	SINE1	Sudden	Safe	Safe	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 \geq 0$	$x_2 - \sin x_1 \geq 0$
	SINE1_g	Gradual	Safe	Safe	$p(y) = 0.2$	$p(y) = 0.8$	$p(y) = 0.8$	$p(y) = 0.2$
	SINE1	Sudden	Safe	Safe	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 \geq 0$	$x_2 - \sin x_1 \geq 0$
	SINE1_g	Gradual	Safe	Safe	$p(y) = 0.3$	$p(y) = 0.7$	$p(y) = 0.7$	$p(y) = 0.3$
	SINE1	Sudden	Safe	Safe	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 \geq 0$	$x_2 - \sin x_1 \geq 0$
	SINE1_g	Gradual	Safe	Safe	$p(y) = 0.4$	$p(y) = 0.6$	$p(y) = 0.6$	$p(y) = 0.4$
	SEA	Sudden	Safe	Safe	$x_1 + x_2 \leq 7$	$x_1 + x_2 \leq 7$	$x_1 + x_2 > 7$	$x_1 + x_2 > 7$
	SEA_g	Gradual	Safe	Safe	$p(y) = 0.5$	$p(y) = 0.1$	$p(y) = 0.5$	$p(y) = 0.9$
	SEA	Sudden	Safe	Safe	$x_1 + x_2 \leq 7$	$x_1 + x_2 \leq 7$	$x_1 + x_2 > 7$	$x_1 + x_2 > 7$
	SEA_g	Gradual	Safe	Safe	$p(y) = 0.5$	$p(y) = 0.2$	$p(y) = 0.5$	$p(y) = 0.8$
	SEA	Sudden	Safe	Safe	$x_1 + x_2 \leq 7$	$x_1 + x_2 \leq 7$	$x_1 + x_2 > 7$	$x_1 + x_2 > 7$
	SEA_g	Gradual	Safe	Safe	$p(y) = 0.5$	$p(y) = 0.3$	$p(y) = 0.5$	$p(y) = 0.7$
	SEA	Sudden	Safe	Safe	$x_1 + x_2 \leq 7$	$x_1 + x_2 \leq 7$	$x_1 + x_2 > 7$	$x_1 + x_2 > 7$
	SEA_g	Gradual	Safe	Safe	$p(y) = 0.5$	$p(y) = 0.4$	$p(y) = 0.5$	$p(y) = 0.6$
$p(X\|y)$	SINE1	Sudden	Safe	Safe	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 \geq 0$	$x_2 - \sin x_1 \geq 0$
	SINE1_g	Gradual	Safe	Safe	$p(y) = 0.1$	$p(y) = 0.1$	$p(x_1 < 0.5) = 0.9$	$p(x_1 < 0.5) = 0.1$
	SINE1	Sudden	Safe	Safe	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 \geq 0$	$x_2 - \sin x_1 \geq 0$
	SINE1_g	Gradual	Safe	Safe	$p(y) = 0.2$	$p(y) = 0.2$	$p(x_1 < 0.5) = 0.9$	$p(x_1 < 0.5) = 0.1$
	SINE1	Sudden	Safe	Safe	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 \geq 0$	$x_2 - \sin x_1 \geq 0$
	SINE1_g	Gradual	Safe	Safe	$p(y) = 0.3$	$p(y) = 0.3$	$p(x_1 < 0.5) = 0.9$	$p(x_1 < 0.5) = 0.1$
	SINE1	Sudden	Safe	Safe	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 \geq 0$	$x_2 - \sin x_1 \geq 0$
	SINE1_g	Gradual	Safe	Safe	$p(y) = 0.4$	$p(y) = 0.4$	$p(x_1 < 0.5) = 0.9$	$p(x_1 < 0.5) = 0.1$
	SEA	Sudden	Borderline	Safe	$x_1 + x_2 \leq 7$	$x_1 + x_2 \leq 7$	$x_1 + x_2 > 7$	$x_1 + x_2 > 7$
	SEA_g	Gradual	Borderline	Safe	$p(y) = 0.1$	$p(y) = 0.1$	$p(x_1 < 5) = 0.9$	$p(x_1 < 5) = 0.1$
	SEA	Sudden	Borderline	Borderline	$x_1 + x_2 \leq 7$	$x_1 + x_2 \leq 7$	$x_1 + x_2 > 7$	$x_1 + x_2 > 7$
	SEA_g	Gradual	Borderline	Safe	$p(y) = 0.2$	$p(y) = 0.2$	$p(x_1 < 5) = 0.9$	$p(x_1 < 5) = 0.1$
	SEA	Sudden	Safe	Safe	$x_1 + x_2 \leq 7$	$x_1 + x_2 \leq 7$	$x_1 + x_2 > 7$	$x_1 + x_2 > 7$
	SEA_g	Gradual	Safe	Safe	$p(y) = 0.3$	$p(y) = 0.3$	$p(x_1 < 5) = 0.9$	$p(x_1 < 5) = 0.1$
	SEA	Sudden	Safe	Safe	$x_1 + x_2 \leq 7$	$x_1 + x_2 \leq 7$	$x_1 + x_2 > 7$	$x_1 + x_2 > 7$
	SEA_g	Gradual	Safe	Safe	$p(y) = 0.4$	$p(y) = 0.4$	$p(x_1 < 5) = 0.9$	$p(x_1 < 5) = 0.1$
$p(y\|X)$	SINE1	Sudden	Safe	Safe	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 \geq 0$	$x_2 - \sin x_1 \geq 0$	$x_2 - \sin x_1 < 0$
	SINE1_g	Gradual	Safe	Safe	$p(y) = 0.1$	$p(y) = 0.1$	$p(y) = 0.9$	$p(y) = 0.9$
	SINE1	Sudden	Safe	Safe	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 \geq 0$	$x_2 - \sin x_1 \geq 0$	$x_2 - \sin x_1 < 0$
	SINE1_g	Gradual	Safe	Safe	$p(y) = 0.2$	$p(y) = 0.2$	$p(y) = 0.8$	$p(y) = 0.8$
	SINE1	Sudden	Safe	Safe	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 \geq 0$	$x_2 - \sin x_1 \geq 0$	$x_2 - \sin x_1 < 0$
	SINE1_g	Gradual	Safe	Safe	$p(y) = 0.3$	$p(y) = 0.3$	$p(y) = 0.7$	$p(y) = 0.7$
	SINE1	Sudden	Safe	Safe	$x_2 - \sin x_1 < 0$	$x_2 - \sin x_1 \geq 0$	$x_2 - \sin x_1 \geq 0$	$x_2 - \sin x_1 < 0$
	SINE1_g	Gradual	Safe	Safe	$p(y) = 0.4$	$p(y) = 0.4$	$p(y) = 0.6$	$p(y) = 0.6$
	SEA	Sudden	Safe	Borderline	$x_1 + x_2 \leq 7$	$x_1 + x_2 \leq 13$	$x_1 + x_2 > 7$	$x_1 + x_2 > 13$
	SEA_g	Gradual	Safe	Borderline	$p(y) = 0.1$	$p(y) = 0.1$	$p(y) = 0.9$	$p(y) = 0.9$
	SEA	Sudden	Safe	Borderline	$x_1 + x_2 \leq 7$	$x_1 + x_2 \leq 13$	$x_1 + x_2 > 7$	$x_1 + x_2 > 13$
	SEA_g	Gradual	Safe	Borderline	$p(y) = 0.2$	$p(y) = 0.2$	$p(y) = 0.8$	$p(y) = 0.8$
	SEA	Sudden	Safe	Safe	$x_1 + x_2 \leq 7$	$x_1 + x_2 \leq 13$	$x_1 + x_2 > 7$	$x_1 + x_2 > 13$
	SEA_g	Gradual	Borderline	Borderline	$p(y) = 0.3$	$p(y) = 0.3$	$p(y) = 0.7$	$p(y) = 0.7$
	SEA	Sudden	Safe	Safe	$x_1 + x_2 \leq 7$	$x_1 + x_2 \leq 13$	$x_1 + x_2 > 7$	$x_1 + x_2 > 13$
	SEA_g	Gradual	Safe	Safe	$p(y) = 0.4$	$p(y) = 0.4$	$p(y) = 0.6$	$p(y) = 0.6$

The parameters of this generator are:

- *Number of attributes*: the number of features for each data point;
- *Number of clusters*: the number of clusters where the minority class data points can be located;
- *Positive share*: the number between 0 and 1 which specify the ratio of positive class examples with respect to the total number;
- *Safe ratio*: probability weight of a minority sample to be from the "safe" zone;
- *Borderline ratio*: probability weight of a minority sample to be from the "borderline" zone;
- *Outlier ratio*: probability weight of a minority sample to be from the "outlier" zone;
- *Rare ratio*: probability weight of a minority sample to be rare. These points are generated near an outlier, forming small groups of minority samples outside the clusters;
- *Uniform or normal distribution*: the minority samples can have these two types of distribution inside the clusters; and
- *standard deviations*: Number of standard deviations fitting in clusters with a normal distribution.

The drift generator can generate nine different types of drift. Experiments have been performed with each one of them with four imbalance rates (*1:9, 2:8, 3:7, 4:6*) and three different drift speeds: *sudden* with the drift happening at the 50,000th time step, *incremental* starting at 45,000th time step and ending at the 55,000th time step, *recurrent* starting at the 45,000th time step, going to the next distribution until the 50,000th time step and coming back at the original one at the 55,000th time step. An illustrative sequence that explains what the each type of drift consist on is shown in the following list.

- *Appearing minority*: this drift, presented in Figure 4.2, starts with zero positive instances. the minority clusters start appearing as it progress to its end, increasing both the clusters' radius and the minority samples probability.

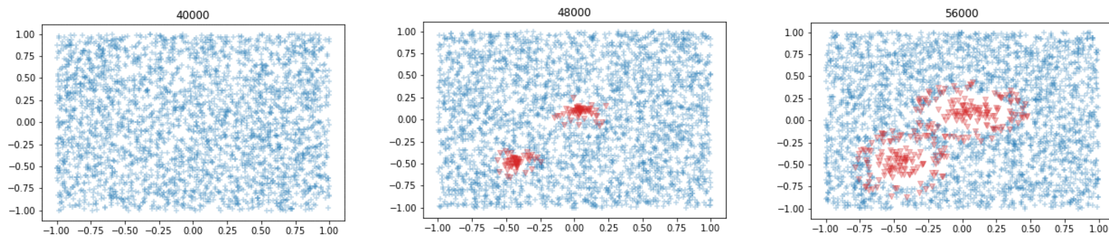


Figure 4.2: The illustrated sequence of the appearing minority drift.

- *Disappearing minority*: this drift, presented in Figure 4.3, consists on making the minority clusters radius and the minority samples probability decrease to zero as the drift progress to its end.

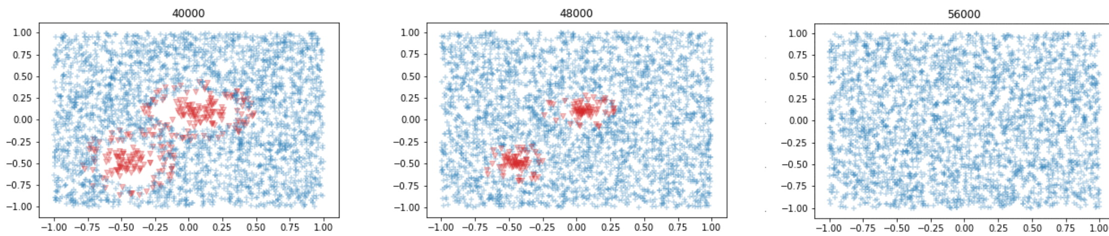


Figure 4.3: The illustrated sequence of the disappearing minority drift.

- *Minority share*: this drift, presented in Figure 4.4, is similar to the appearing minority because the minority appears as the drift progress but the clusters are already there.

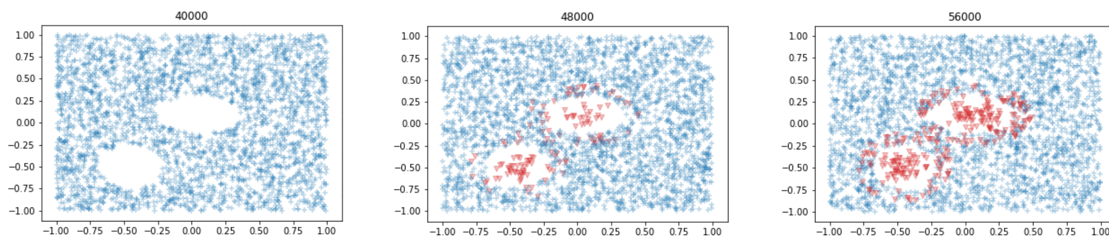


Figure 4.4: The illustrated sequence of the minority share drift.

- *Cluster movement*: this drift, presented in Figure 4.5, ends with the minority clusters in a different random position. The movement progress follows the line from the original to the target position.

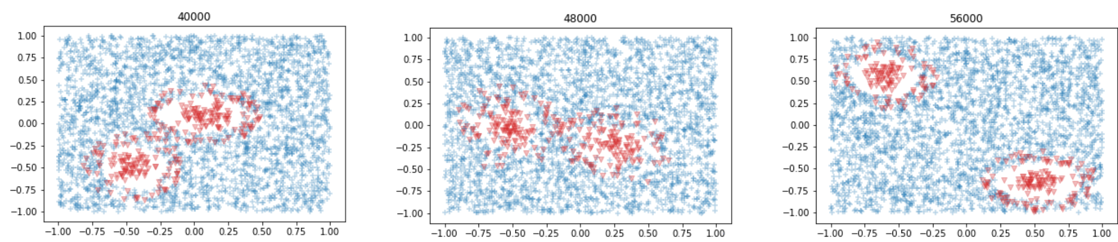


Figure 4.5: The illustrated sequence of the cluster movement drift.

- *Cluster jitter*: this drift, presented in Figure 4.6, consists on making the minority clusters center move randomly, giving the clusters a shaking effect.

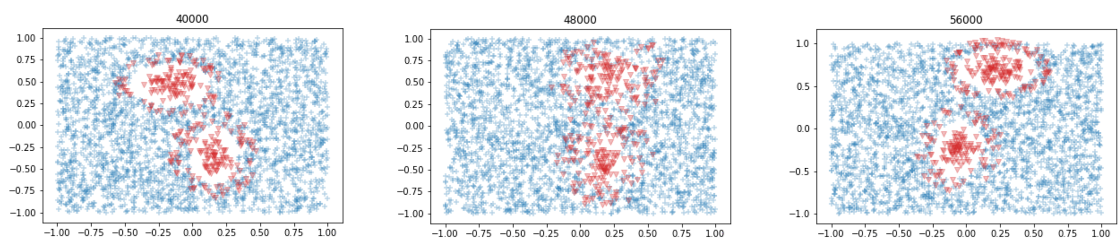


Figure 4.6: The illustrated sequence of the cluster jitter drift.

- *Appearing cluster*: This drift, presented in Figure 4.7, consists on making a new cluster appear without any modification to the one already there.

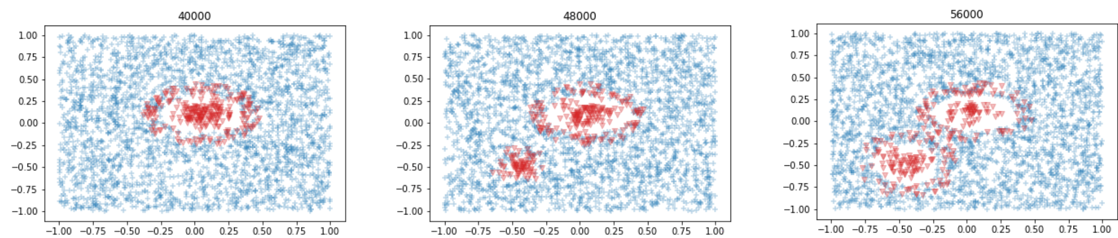


Figure 4.7: The illustrated sequence of the appearing cluster drift.

- *Splitting cluster*: this drift, presented in Figure 4.8, consists on making the minority cluster split in two which will move to two different target position.

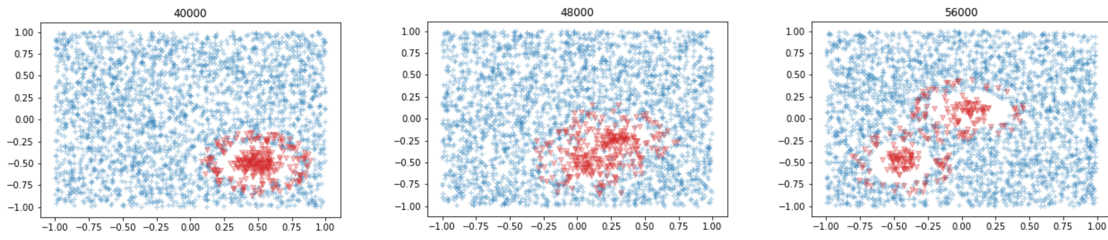


Figure 4.8: The illustrated sequence of the splitting cluster drift.

- *Shape Shift*: this drift, presented in Figure 4.9, consists on making the minority clusters change shape, increasing the radiuses and producing longer shapes.

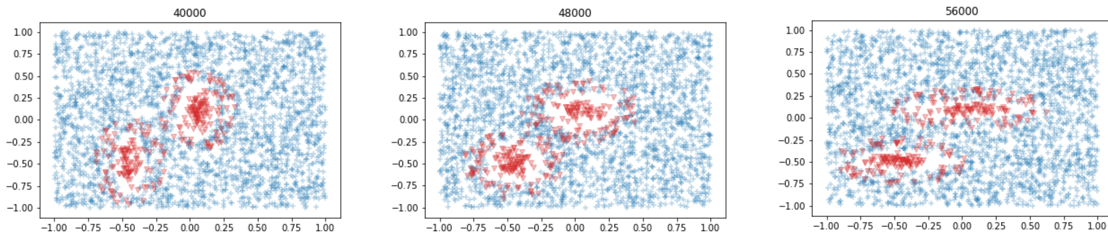


Figure 4.9: The illustrated sequence of the shape shift drift.

- *Borderline shift*: this drift, presented in Figure 4.10, consists on making the borderline radius of the minority clusters grow through the center making the safe radius shrink.

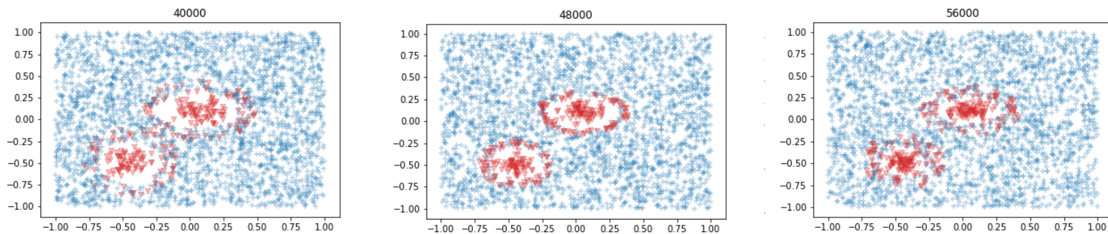


Figure 4.10: The illustrated sequence of the borderline drift.

4.2.3 Real Datasets

I tested the algorithms on the following real datasets in order to confirm the results achieved on the artificial data streams. In all these datasets, the positive class is

the minority one as it was in the artificial ones. Their class distribution is shown in Figure 4.11.

- **PAKDD:** The 13th Pacific-Asia Knowledge Discovery and Data Mining Conference (PAKDD 2009) [58] presented a competition focused on the problem of credit risk assessment.
  The models need to be robust against performance degradation caused by gradual market changes along a few years of business operation. This dataset contains 50000 instances composed by twenty seven features, thirteen categorical and fourteen numerical. The percentage of positive class instances is 20%.

- **Electricity:** Electricity [26] is another widely used dataset for imbalanced classification. This data were collected from the Australian New South Wales Electricity Market. In this market, prices are not fixed and are affected by the demand and supply of the market. They are set every five minutes. The class label identifies the change of the price relative to a moving average of the last 24 hours. This dataset contains 45,312 instances dated from 7 May 1996 to 5 December 1998. Each instance is composed of eight numerical features and the percentage of positive class instances is 42%.

- **KDDCup:** The KDDCup [59] is the dataset used for The Third International Knowledge Discovery and Data Mining Tools Competition. The competition task was to build a classifier able to distinguish intrusions from normal connections. This database contains a wide variety of intrusions simulated in a military network environment and it contains 494,021 instances with twenty seven features, seven categorical and twenty numerical. The percentage of positive class instances is 20%.

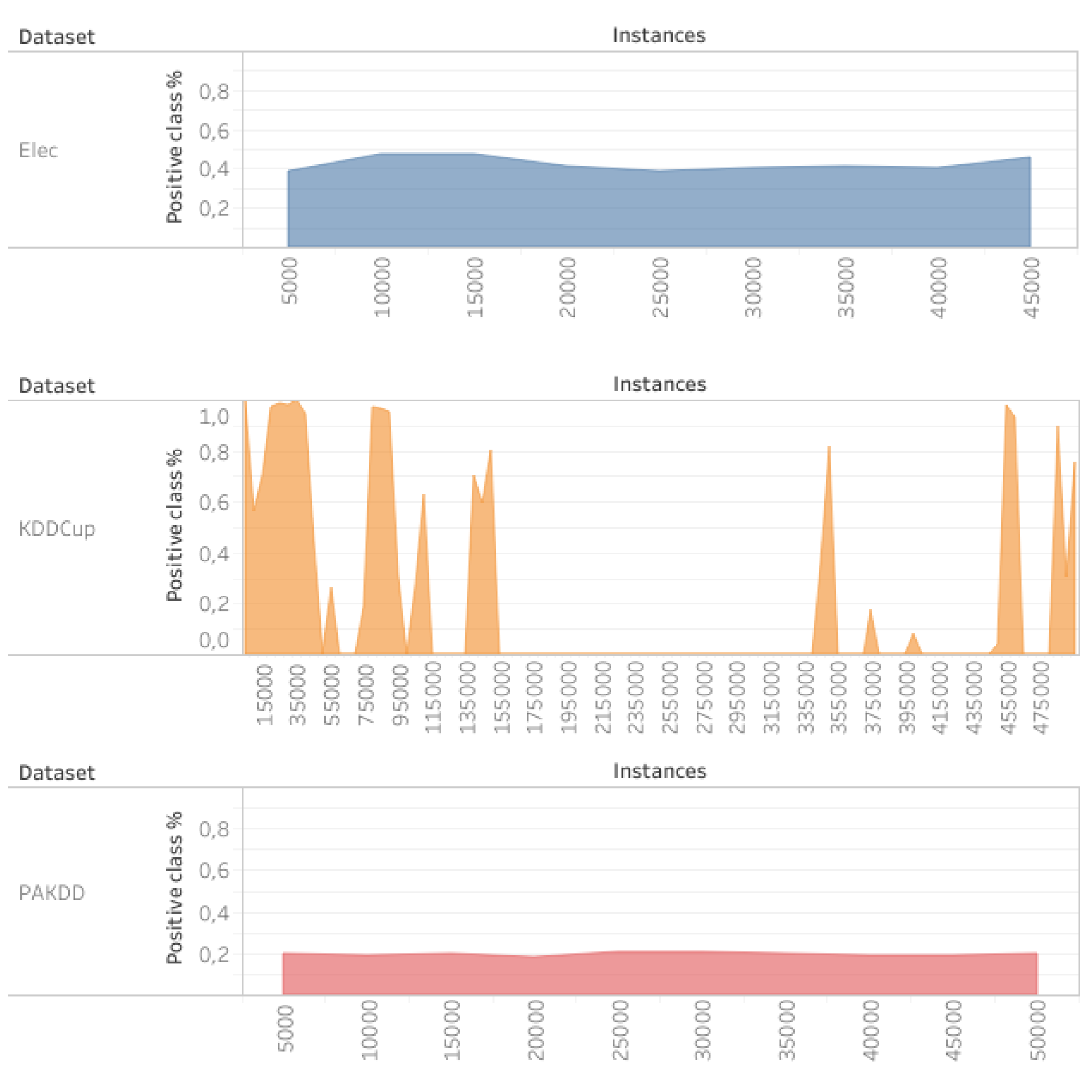


Figure 4.11: The distribution of the class probability during the real data streams.

Chapter 5

Implementation experience

This chapter presents the structure of my work towards the experiment execution. Section 5.1 describes my upgrade at the data stream generator, Section 5.2 describes my implementation of the online bagging based algorithms, while Section 5.3 describes the building blocks of the benchmarking environment. Finally, Section 5.4 details the algorithm parameter and the evaluation configuration of the experiments.

5.1 Cluster drift generation

The original implementation of the cluster generator makes the drift start from the *start instance* and end at the *end instance*. A variable called *progress* is incremented in order to keep track of how much drift has been already made. When the data stream reaches the $45,000th$ time step, the *base distribution* enters the "drift phase" in which it is modified as the drift requires, starting from a progress of 0 and ending at a progress of 1.

This implementation was not exactly as I planned to perform the drifts. For example, an incremental appearing minority drift starting at the $45,000th$ time step and ending at the $55,000th$ was starting with the base distribution with the minority class clusters present and only at the $45,000th$ time step the progress was set to zero making the minority clusters disappearing and reappearing gradually until the $55,000th$ time step when the drift was completed and the distribution was back to the base one. In order to solve this problem, I extended the drift class adding two variables: *Realstart* and *Realend*. These variables indicate when to start increasing the *progress* counter. The original *start instance* and *end instance* are set at the start and end of the data stream respectively. In this way the drift starts when the stream begins but with progress 0 until the *Realstart*

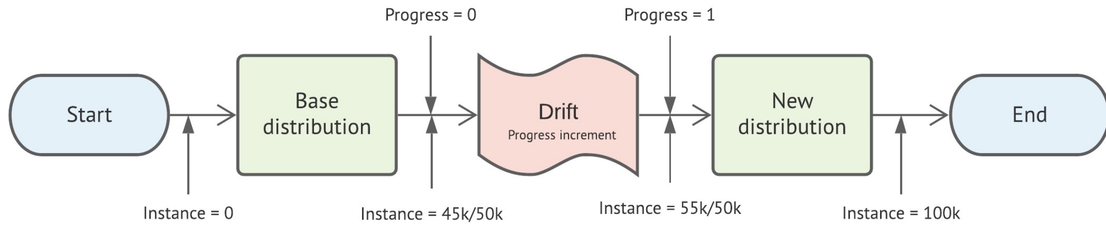


Figure 5.1: Original stream generation sequence.

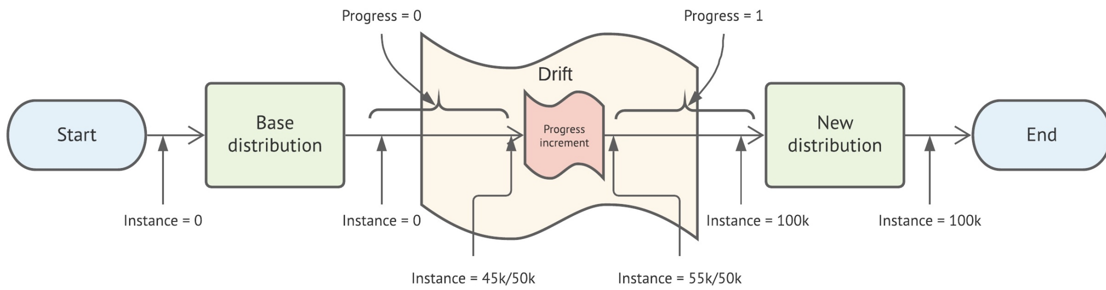


Figure 5.2: Custom stream generation sequence.

instance is reached. The difference between these two drift generation strategies are illustrated in Figure 5.1 and Figure 5.2.

5.2 Algorithms implementation

I integrated the ESOS-ELM [51] algorithm available in this repository[1] and I implemented the algorithms based on Online Bagging. The C-SMOTE [47] and RebalanceStream [48] algorithms were already available in MOA[2].

The implementation of the Online Bagging algorithms follows the paper structure. As shown in Figure 5.3, they are extensions of the OzaBag algorithm which is the online version of a simple bagging algorithm where each instance is trained by the classifiers with a weight extracted from a Poisson with lambda equal to 1. Firstly, I implemented the native OOB and UOB with the second being an extension of the first. Then I implemented the improved versions and the ensembles.

[1]`https://github.com/dabrze/imbalanced-stream-generator`

[2]`https://github.com/Waikato/moa`

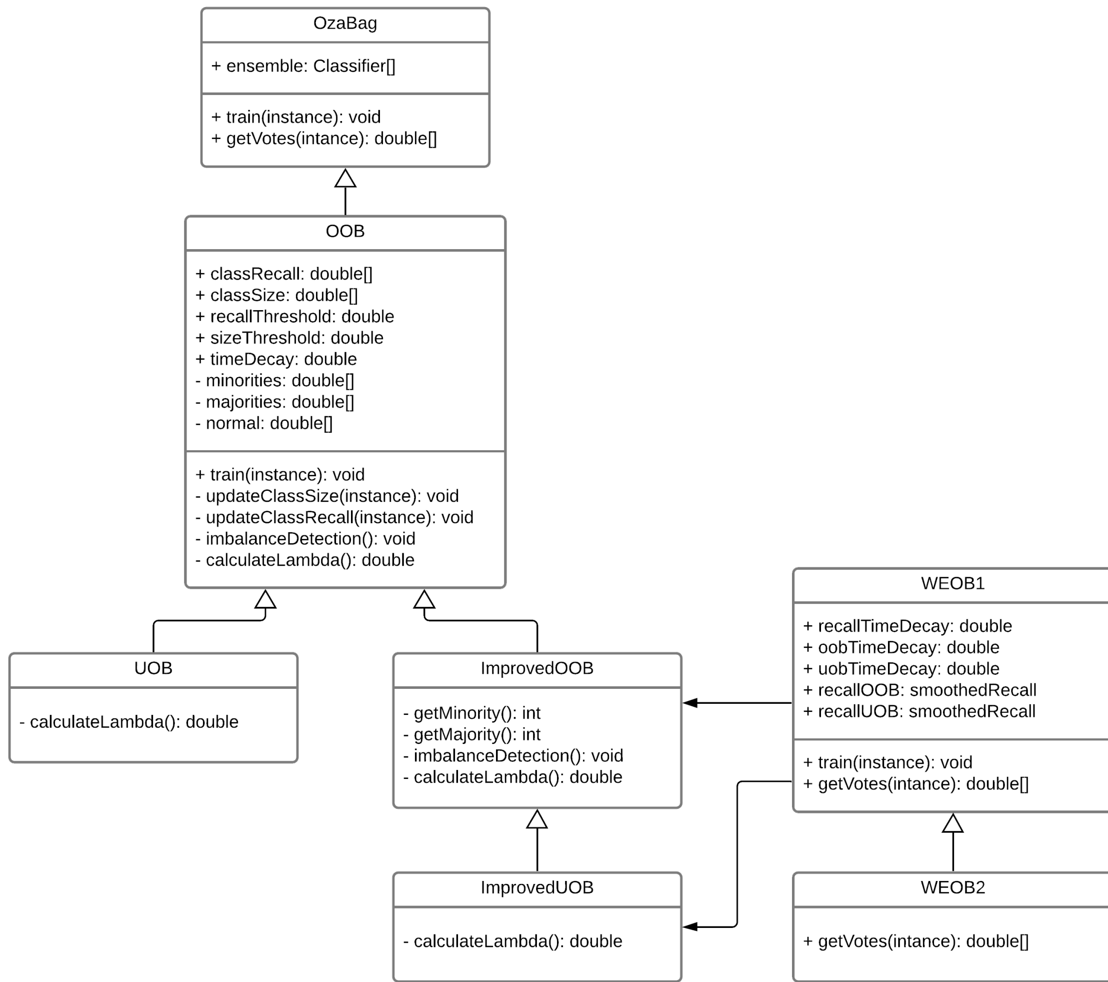


Figure 5.3: The UML class diagram of the Online bagging algorithms.

5.2.1 Native OOB and UOB

As described in the corresponding study [49], the first step of the training procedure is to group the class in the majority, minority and normal sets. This is done by keeping track of the size and recall of each class with a decaying factor. A decay factor is useful to deal with concept drift in order to give more weight to the recent statistics w.r.t what happened far back in the stream. The oversampling or undersampling is activated only if the difference between recalls is greater than a threshold. This avoids useless corrections that could deteriorate the performances. The implementation code of the updating methods is shown below in Listing 5.1 and 5.2.

Listing 5.1: The update of the class sizes method

```
protected void updateClassSize(Instance inst) {
    // Set the same weight for all the classes at the beginning of the stream
    if (this.classSize == null) {
        classSize = new double[inst.numClasses()];
        Arrays.fill(classSize, 1d / classSize.length);
    }
    // update the class size with the decaying factor theta for all the classes
    for (int i=0; i<classSize.length; ++i) {
        classSize[i] = theta.getValue() * classSize[i] + (1d - theta.getValue()) * ((int) inst.classValue()
             == i ? 1d:0d);
    }
}
```

Listing 5.2: The update of the class recalls method.

```
protected void updateClassRecall(Instance inst){
    // Start with the same recall for all the classes at the beginning of the stream
    if (this.classRecall == null) {
        classRecall = new double[inst.numClasses()];
        Arrays.fill(classRecall, 1d);
    }
    int classk = (int)inst.classValue();
    // update the recall of the instance class
    classRecall[classk] = theta.getValue() * classRecall[classk] + (1d - theta.getValue()) * (
         correctlyClassifies(inst) ? 1d:0d);
}
```

The method invoked by MOA when a new instance is ready for the classifier training is the *trainOnInstanceImpl* method. This method starts with updating the class sizes and recalls with the new instance class. Then, it assigns each class to a set as detailed in Listing 5.3. Finally, It computes the corresponding lambda.

The OOB lambda is computed by the method shown in Listing 5.4. It will return a value greater than 1 if the instance's class belongs to the minority set and 1 otherwise. The class corresponding to the UOB algorithm overrides only the method relative to lambda computation. The lambda of the UOB will be less than 1 in case of the instance's class belonging to the majority set and 1 otherwise. It is computed by the method shown in Listing 5.5.

For each classifier in the ensemble, it weights the instance with a Poisson extraction with the computed lambda and passes it to the classifier for the training. The complete code of the training procedure is shown in Listing 5.6.

5.2.2 Improved OOB and UOB

The implementation of the improved versions of the two algorithms will be a class extension of the OOB class. This will require overriding only the lambda computing method. These algorithms do no require the classes to be grouped with respect to the three sets, majority, minority and normal but lambda is computed only with respect to the bigger or smaller class.

Listing 5.3: The class imbalance detection.

```
protected void imbalanceDetection(){
    //compare the recall and size of the classes to assign them to the correct group
    for (int i=0; i<classSize.length-1; i++)
        for (int j=i; j<classSize.length; j++){
            // if the difference of the class sizes and recalls is above the corresponding thresholds assign
                them to the minorities and majorities groups
            if(classSize[j] - classSize[i] > sizethreshold.getValue() && classRecall[j] - classRecall[i] >
                recallthreshold.getValue()){
                minorities.add(i);
                majorities.add(j);
            }
            else if(classSize[i] - classSize[j] > sizethreshold.getValue() && classRecall[i] - classRecall[j]
                 > recallthreshold.getValue()){
                minorities.add(j);
                majorities.add(i);
            }
        }
    for (int i=0; i<classSize.length; i++) {
        // fill the normal group with the classes in both or none of the other groups
        if (minorities.contains(i))
            majorities.remove(i);
        if (!minorities.contains(i) && !majorities.contains(i))
            normal.add(i);
    }
}
```

Listing 5.4: The native OOB lambda computing

```
protected double calculatePoissonLambda(Instance inst) {
    // increase the lambda if the class is in the minorities group
    if (minorities.contains((int)inst.classValue()))
        return 1/classSize[(int)inst.classValue()];
    return 1d;
}
```

Listing 5.5: The native UOB lambda computing

```
public double calculatePoissonLambda(Instance inst) {
    // decrease the lambda if the class is in the majorities group
    if (majorities.contains((int)inst.classValue()))
        return 1 - classSize[(int)inst.classValue()];
    return 1d;
}
```

Listing 5.6: The training procedure

```
public void trainOnInstanceImpl(Instance inst) {
    // update the class sizes and recalls
    updateClassSize(inst);
    updateClassRecall(inst);
    imbalanceDetection();
    //compute the lambda for the poisson extraction
    double lambda = calculatePoissonLambda(inst);
    for (moa.classifiers.Classifier classifier : this.ensemble) {
        //extract the instance weight
        int k = MiscUtils.poisson(lambda, random_obj);
        if (k > 0) {
            Instance weightedInst = inst.copy();
            weightedInst.setWeight(inst.weight() * k);
            classifier.trainOnInstance(weightedInst);
        }
    }
}
```

Listing 5.7: The Improved OOB methods

```
// find the index of the class with the bigger size
protected int getMajorityClass() {
    int indexMaj = 0;
    for (int i=1; i<classSize.length; ++i) {
        if (classSize[i] > classSize[indexMaj]) {
            indexMaj = i;
        }
    }
    return indexMaj;
}
protected double calculatePoissonLambda(Instance inst) {
    int majClass = getMajorityClass();
    return classSize[majClass] / classSize[(int) inst.classValue()];
}
```

Listing 5.8: The Improved UOB methods

```
// find the index of the class with the smaller size
protected int getMinorityClass() {
    int indexMin = 0;
    for (int i=1; i<classSize.length; ++i) {
        if (classSize[i] <= classSize[indexMin]) {
            indexMin = i;
        }
    }
    return indexMin;
}
public double calculatePoissonLambda(Instance inst) {
    int minClass = getMinorityClass();
    return classSize[minClass] / classSize[(int) inst.classValue()];
}
```

The methods to compute the index of those classes will be invoked by the lambda computing ones. Their code is shown in Listing 5.7 and 5.8.

5.2.3 Ensembles of the improved versions

Two ensembles of the improved OOB and UOB have been proposed in their same paper, they are called *WEOB1* and *WEOB2*. These ensembles weigh the predictions of the two algorithms with their Gmean computed with a moving average of the recall called *Smoothed Recall*. This new measure explained in the paper is a moving average of the class recall, the goal is to smooth out the short-term fluctuations of the original measures. It is implemented with a sliding window with constant updating computing time and memory allocation proportional to the size of the window, which is a new parameter with a default value equal to 1000. The code is shown in Listing 5.9

The WEOB1 prediction is a weighted sum of the OOB and UOB prediction, the weights are the normalized Gmean values of the corresponding algorithms. The code to get the class prediction is shown in Listing 5.10.

WEOB2 simply uses the Gmean values to choose which prediction to use based on which one of the two algorithms has the greater one. The code to get the class prediction is shown in Listing 5.11.

Listing 5.9: The updating procedure of the smoothed recall

```
// This method is called during the training of the ensembles and it update the class recalls of one
     algorithm
public void insertPrediction(int classValue, boolean pred){
    double r;
    //array initialization
    if( this.windowPos == -1){
        r = pred ? 1d:0d;
        for(int c = 0; c< recalls.length; c++)
            if( c == classValue) {
                recalls[c][0] = r;
                smoothedRecalls[c] = r;
            }
            else {
                recalls[c][0] = 0;
                smoothedRecalls[c] = 0;
            }
        windowPos = 0;
        return;
    }
    r = this.theta * recalls[classValue][windowPos] + (1d - this.theta) * (pred ? 1d:0d);
    //newWP -> cursor for the position of the new instance on the window
    int newWP = windowPos + 1;
    if(newWP >= recalls[classValue].length)
        newWP = 0;
    // remove the oldest recalls if the window is full
    for(int c = 0; c< smoothedRecalls.length; c++) {
        //smoothedRecall become the sum of the saved recall
        smoothedRecalls[c] = smoothedRecalls[c] * windowSize;
        smoothedRecalls[c] = smoothedRecalls[c] - recalls[c][newWP];
    }
    // increase the size of the window if its not at the maximum
    this.windowSize = Math.min(windowSize + 1, recalls[classValue].length);
    //add the new recalls to the window
    for(int c = 0; c < smoothedRecalls.length; c++) {
        if (c == classValue) {
            recalls[c][newWP] = r;
        } else {
            recalls[c][newWP] = recalls[c][windowPos];
        }
        smoothedRecalls[c] = (smoothedRecalls[c] + recalls[c][newWP]) / windowSize;
    }
    //update the cursor window position
    windowPos = newWP;
}
```

Listing 5.10: The prediction procedure of the WEOB1 ensemble

```
1  public double[] getVotesForInstance(Instance inst) {
2      double[] oobVotes = oob.getVotesForInstance(inst);
3      double[] uobVotes = uob.getVotesForInstance(inst);
4      double[] finalVotes = new double[oobVotes.length];
5      //initialize the smoothed recalls of UOB and OOB
6      if(classRecallUOB == null){
7          oob.randomSeedOption.setValue(this.randomSeedOption.getValue());
8          uob.randomSeedOption.setValue(this.randomSeedOption.getValue());
9          classRecallOOB = new SmoothedRecall(inst.numClasses(),recalltheta.getValue(),
               SmoothedRecallWindowSizeOption.getValue());
10         classRecallUOB = new SmoothedRecall(inst.numClasses(),recalltheta.getValue(),
               SmoothedRecallWindowSizeOption.getValue());
11     }
12     // compute the corresponding gmeans
13     double uobGmean = classRecallUOB.getGmean();
14     double oobGmean = classRecallOOB.getGmean();
15     // normalize the gmeans values
16     double alphaO = oobGmean / (oobGmean + uobGmean);
17     double alphaU = uobGmean / (oobGmean + uobGmean);
18     // sum the votes of the OOB and UOB weighted with the normalized gmeans
19     for(int i = 0; i < finalVotes.length; i++){
20         try {
21             finalVotes[i] = alphaO * oobVotes[i] + alphaU * uobVotes[i];
22         }catch (IndexOutOfBoundsException e){
23             finalVotes[i] = 0;
24         }
25     }
26     return finalVotes;
27 }
```

Listing 5.11: The prediction procedure of the WEOB2 ensemble

```
1  public double[] getVotesForInstance(Instance inst) {
2      double[] oobVotes = oob.getVotesForInstance(inst);
3      double[] uobVotes = uob.getVotesForInstance(inst);
4      if(classRecallUOB == null){
5          oob.randomSeedOption.setValue(this.randomSeedOption.getValue());
6          uob.randomSeedOption.setValue(this.randomSeedOption.getValue());
7          classRecallOOB = new SmoothedRecall(inst.numClasses(),recalltheta.getValue(),
               SmoothedRecallWindowSizeOption.getValue());
8          classRecallUOB = new SmoothedRecall(inst.numClasses(),recalltheta.getValue(),
               SmoothedRecallWindowSizeOption.getValue());
9      }
10     double uobGmean = classRecallUOB.getGmean();
11     double oobGmean = classRecallOOB.getGmean();
12     if (oobGmean>uobGmean){
13         return oobVotes;
14     }
15     return uobVotes;
16 }
```

5.3 Benchmarking setup

I built a Benchmarking environment to automate the process of running the experiments and collecting the results. The step sequence is represented in Figure 5.4.

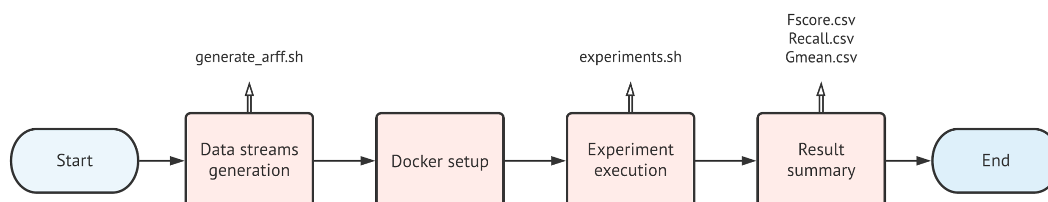


Figure 5.4: The experiments workflow.

It is written in python and each step is done sequentially without needing any manual operation. Various configurations can be set just by updating the variables in a *Config* file, it is possible to set different algorithms and data streams.

The benchmarking environment is public and can be found in this repository[3]. It is composed of the phases detailed in the following list.

- *Data streams generation*: the experiments start with the generation of the data streams. I saved the data streams into file *.arff* in order for their generation not to affect the time and memory statistic of the algorithms. A bash file will be created and executed in order to interface with the MOAs *CLI* and run the tasks. The data streams generated with the cluster generator have the following parameters:
  - Number of attributes: 2;
  - Number of clusters: 2;
  - positive share: the imbalance ratio of the experiment;
  - safe ratio: 0.5;
  - borderline ratio: 0.5;
  - outlier ratio: 0;
  - rare ratio: 0;
  - uniform or normal distribution: normal distribution;
  - standard deviations: 3.

[3] `https://github.com/08volt/moa-replicationstudy/releases/tag/v1.0.0`

- *Docker setup*: as explained in Section 4.1.2, the experiments are run inside a docker container. A docker image needs to be configured with the description of the library and software needed. It's composed by a *Dockerfile* containing the java version and the relative path to the working directory, and by a *Dockerfile.yml* describing the services and their configuration, in my case *influxDB*.
- *Experiment execution*: during this phase, a bash file is created. It will run the tests sequentially, each in a different docker container. Each experiment is run 10 times and the results of each one of them are saved in a different *.csv* file which will contain all the output statistics from MOA.
- *Result summary*: during this phase, a file for each of the statistics selected in the *Config* file is created. These files contain the results of all the experiments at each evaluation step. During the experiments, the evaluation has been performed every $5,000$ instances. With these summaries, a visualization and comparison of the performances will be straightforward.

5.4 Experiments configuration

All the data streams and datasets has been tested with the same algorithm configuration. All the algorithms, except *ESOS-ELM*, has been configured with the Hoeffding Adaptive Tree as base learner. All the ensemble algorithms are composed by ten base learners and have the other parameter configurations set the default. All the parameters are detailed in the following list:

- *Online Bagging algorithm*:
  - θ class size: 0.9
  - recall threshold: 0.4
  - class size threshold: 0.6
- *Online Bagging ensembles*:
  - θ recall: 0.9
  - SmoothedRecall window size: 1000
- *C-SMOTE*:
  - k-neighbours: 5
  - threshold: 0.5
  - minWindowSize: 100

- *RebalanceStream*:
  - maxInstanceLimitBatch = -1 (no limit)
  - minInstanceLimitBatch = -1 (no limit)
- *ESOS-ELM*:
  - *OS-ELM*:
    * neurons: 100
    * initialBatchSize: 1000
    * BatchSize: 1000
    * usePseudoInverse: True
    * epsilon: 0.001
  - *WELM*:
    * neurons: 100
    * initialBatchSize: 1000
    * usePseudoInverse: True
    * epsilon: 0.001

Artificial data streams have been tested using a prequential evaluation with a fixed-length window evaluator. We set the window size to the stream's length before the concept drift: 50,000 for abrupt drifts and 45,000 in the other two cases. This allows restarting measuring when the drift happens without the influence of the statistics before the drift. Real datasets instead have been tested with a prequential evaluation with a fading factor evaluator setting the fading parameter at 0.995. The complete experiment strings are available in this public repository[4].

[4] `https://github.com/08volt/moa-replicationstudy/releases/tag/v1.0.0`

Chapter 6

Results

In this chapter, I show all the results obtained and I compare the performances of the various algorithms. I proceed by analyzing one type of drift at a time. In particular, Section 6.1 presents the results on the data streams with a virtual drift, Section 6.2 presents the results on the data streams with real drifts and Section 6.3 presents the results on the data streams with imbalance ratio drifts. Section 6.4 compares the resource requirements of the algorithms during the experiments. Finally, Section 6.5 compares the performances on the real datasets.

I used OzaBag, which is the online version of a simple bagging algorithm, as a baseline for the Online Bagging techniques. Hoeffding Adaptive Tree has been used as a baseline for C-SMOTE and RebalanceStream. The plot's legend is always presented in decreasing order of Gmean and the point size is proportional to the sum of the recalls' standard deviation.

6.1 P(X|y) drift

P(X|y) drift makes the examples probability distribution on the instance space change but the decision boundary does not shift. The artificial data streams with this kind of drift are only the ones generated with the SEA and SINE1 generators. In both cases the shift concerns the majority class samples. The majority class distribution become more dense in a smaller area near the decision boundary as detailed in Table 4.1. I analyzed the performances of the algorithms w.r.t. a high imbalance ratio (*2:8* and *1:9*) and a low imbalance ratio (*4:6* and *3:7*).
Figure 6.1 compares the Recall and Fscore results of each class. I summarize the results for each algorithm in the following list.

- *native UOB and OOB*: these are the best performing algorithms. UOB achieved the best recall of the minority class with both high and low im-

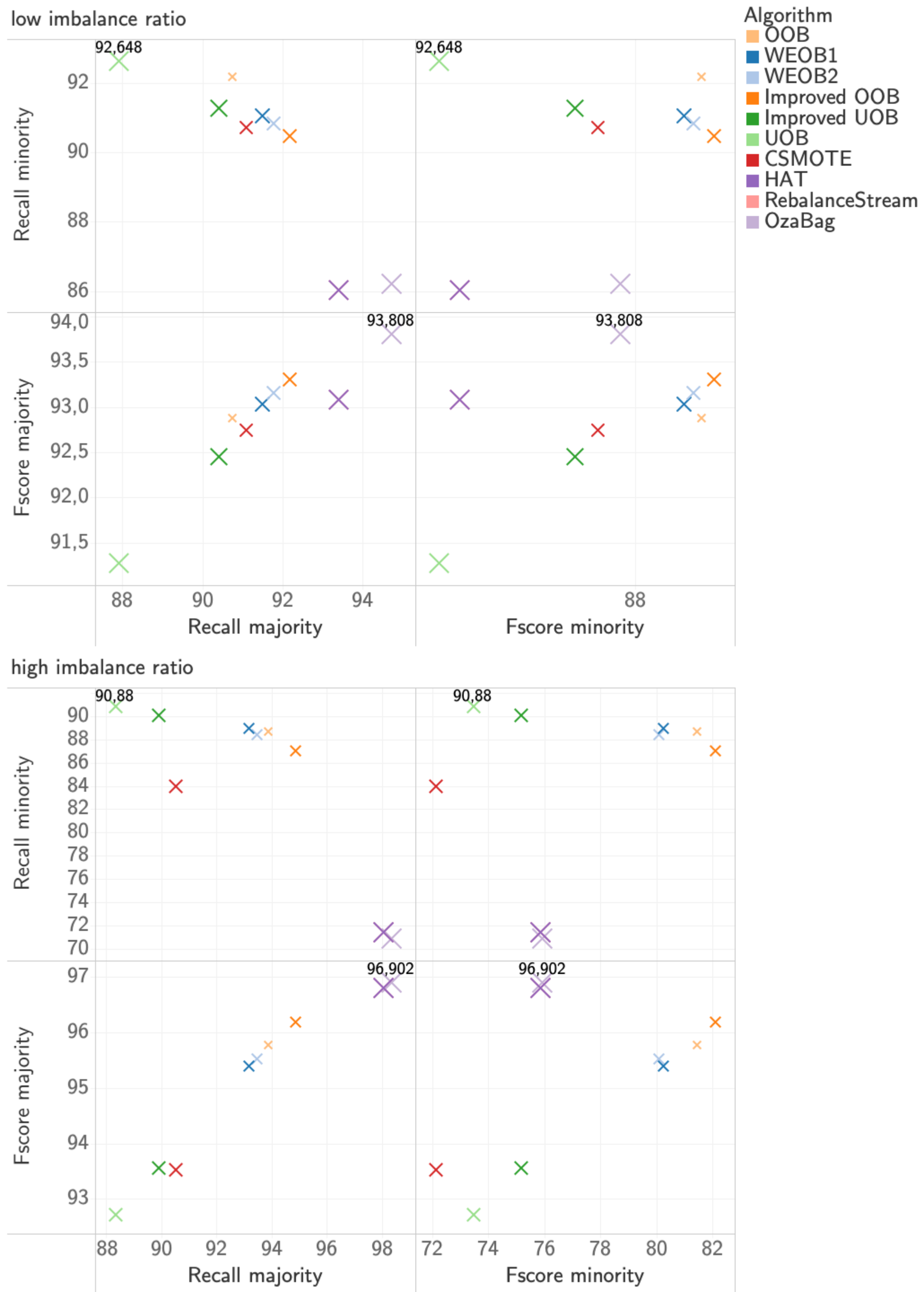


Figure 6.1: Mean Recall and Fscore with streams having a P(X|y) drift.

balance ratios, significantly improving the performances of OzaBag in the first case. The downside is that it loses a significant amount of recall of the majority class. OOB mitigates this, achieving a great recall of the minority class without losing too much of the one of the majority class, thus achieving the best Gmean. An analysis of the Fscore measures allows noticing that, with both low and high imbalance ratios, the Fscore of minority class of OOB is greater than the one of OzaBag while the one of UOB is lower. This means that OOB increases the recall of the minority class keeping a good precision while UOB achieves the highest recall along with the worst precision.

- *Improved UOB and OOB*: in both cases, these algorithms increase the recall of the majority class losing on the one of the minority class w.r.t. the corresponding native versions. The improved UOB is able to improve the Gmean of its native version but this does not happen with OOB. Interesting to notice is that both the minority class and majority class Fscore increases. This could lead to better general performances in multi-class settings. The performances w.r.t. the Gmean measure are similar to each other, but Improved OOB with better recall of the majority class and Improved UOB with better recall of the minority class.

- *Ensembles*: the two ensembles of the improved versions have similar performances: WEOB1 performs better on the recall of the minority class while WEOB2 performs better on the recall of of majority class. Their performances are always a mediation between the two base algorithms.

- *C-SMOTE and RebalanceStream*: C-SMOTE is able to increase significantly the Hoeffding Adaptive Tree performances on the minority class, losing recall of the majority class but overall performing better during all the data streams as shown by the Gmean measure in Figure 6.2. The right plots on Figure 6.1 show that as the imbalance ratio increase this algorithm loses a lot of precision of the minority class w.r.t. its baseline. RebalanceStream, on the other hand, is not able to detect any drift and its performances are exactly the same as the Hoeffding adaptive tree.

- *ESOS-ELM*: this algorithm needs several instances to pass before being able to have accurate predictions. This influences the averaging plots like the ones shown in Figure 6.1, thus its performances are shown only in the plots in Figure 6.2 where it is possible to see this behavior. The plots focus on high imbalance ratios where the algorithm performances reach the ones of C-SMOTE, having a mean recall of the minority class of 86.51 and a mean recall of the majority class of 95.29 during the second half of the stream.

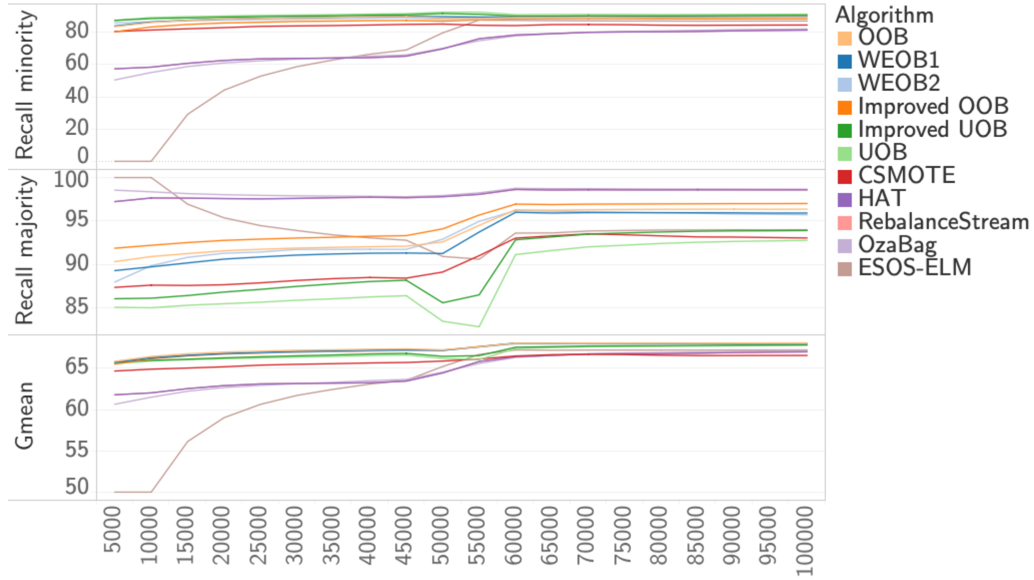


Figure 6.2: Recall and Gmean during streams having a P(X|y) drift and high imbalance ratios.

6.2 P(y|X) drift

P(y|X) drift, also called *Real* drift, is the one that makes the decision boundary shift. The artificial data streams with this kind of drift are:

- Borderline shift;
- Shape shift;
- jitter;
- Cluster movement;
- Appearing cluster;
- Splitting clusters;
- SEA P(y|X); and
- SINE1 P(y|X).

This type of drift is the one that more affects the performance of the algorithms thus it is really important to know how the performances are influenced by the class rebalance.

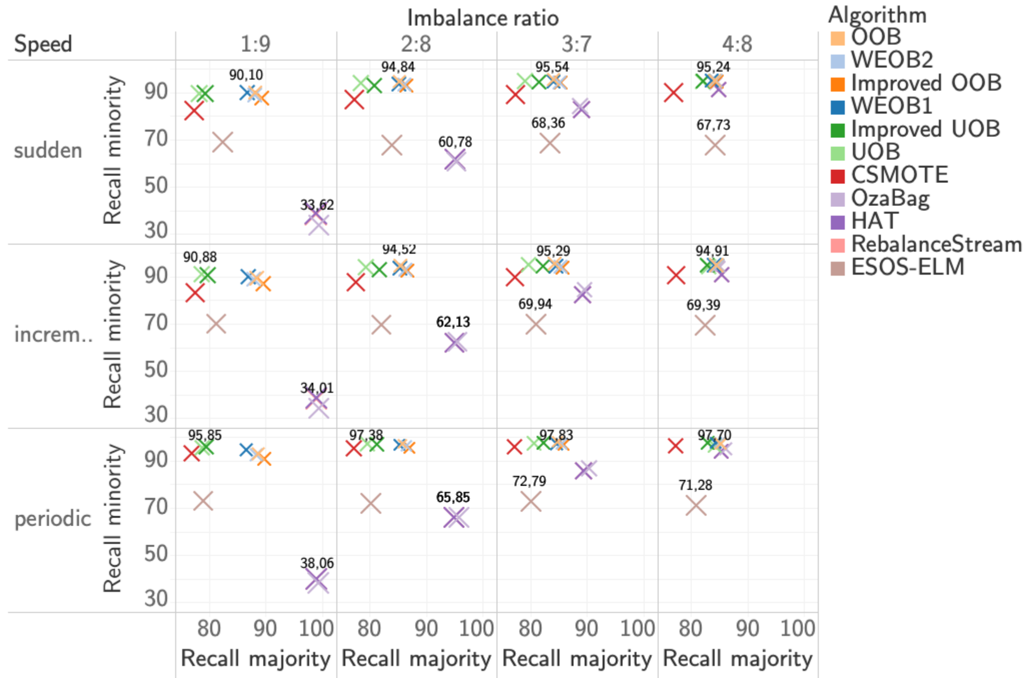


Figure 6.3: Mean Recall with streams having a P(y|X) drift.

The plots in Figure 6.3 show that, the more the imbalance ratio is high, the more the characteristics of the algorithms come to light. The speed of the drift does not affect much the overall performance. Therefore a more detailed plot of the performances of the algorithm with data streams having imbalance ratio *1:9* is shown in Figure 6.4.

Interesting is the difference of performance between SEA and SINE1 which have a low and high severity respectively. Their K temporal statistic and Gmean are shown in Figure 6.5.

I summarize the results for each algorithm in the following list.

- *native UOB and OOB*: differently from the previous category of drifts, UOB has a better recall of the minority class than OOB only when the imbalance ratio is *1:9*. Again, OOB is the algorithm with the best Gmean having a high recall of the minority class and the second higher recall of the majority after its improved version. The plots in Figure 6.4 show that even when the imbalance ratio is high and both the mean and final step recall of the minority class of the UOB are the best ones, during the drift the best performing algorithm is OOB. This behavior can be noticed also in Figure 6.5 thanks to the Kappa temporal statistic. The plots in Figure 6.5 regarding the SINE

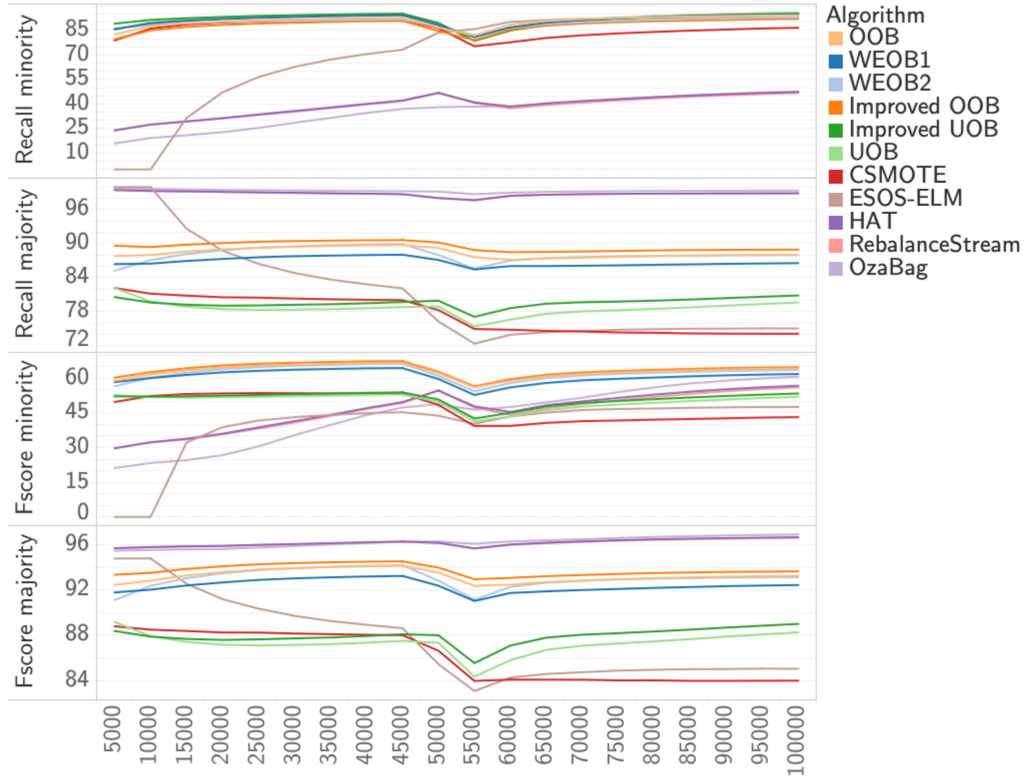


Figure 6.4: Recall and Fscore during streams having a P(y|X) drift and Imbalance ratio *1:9*.

stream highlight how the OzaBag performances increase with oversampling while decrease with undersampling during such a severe drift. Oversampling increases the importance of some of the new examples, thus making the HAT learners react to the drift faster. On the other hand, undersampling "filter" majority class samples, thus slowing the drift detection and worsening the algorithm performance w.r.t not applying any resampling. Figure 6.5 shows the importance of resampling with drifts having low severity as SEA. The Gmean of the OzaBag drops because the old concept boundary is difficult to forget if the wrongly classified example are a few (minority class) and the majority class examples continue to be correctly classified.

- *Improved UOB and OOB*: the improved version of the algorithms increase the recall of the majority class losing a bit on the one of the minority class w.r.t. the corresponding native versions. Again, the improved UOB is able to improve the Gmean of its native version but this does not happen with OOB.

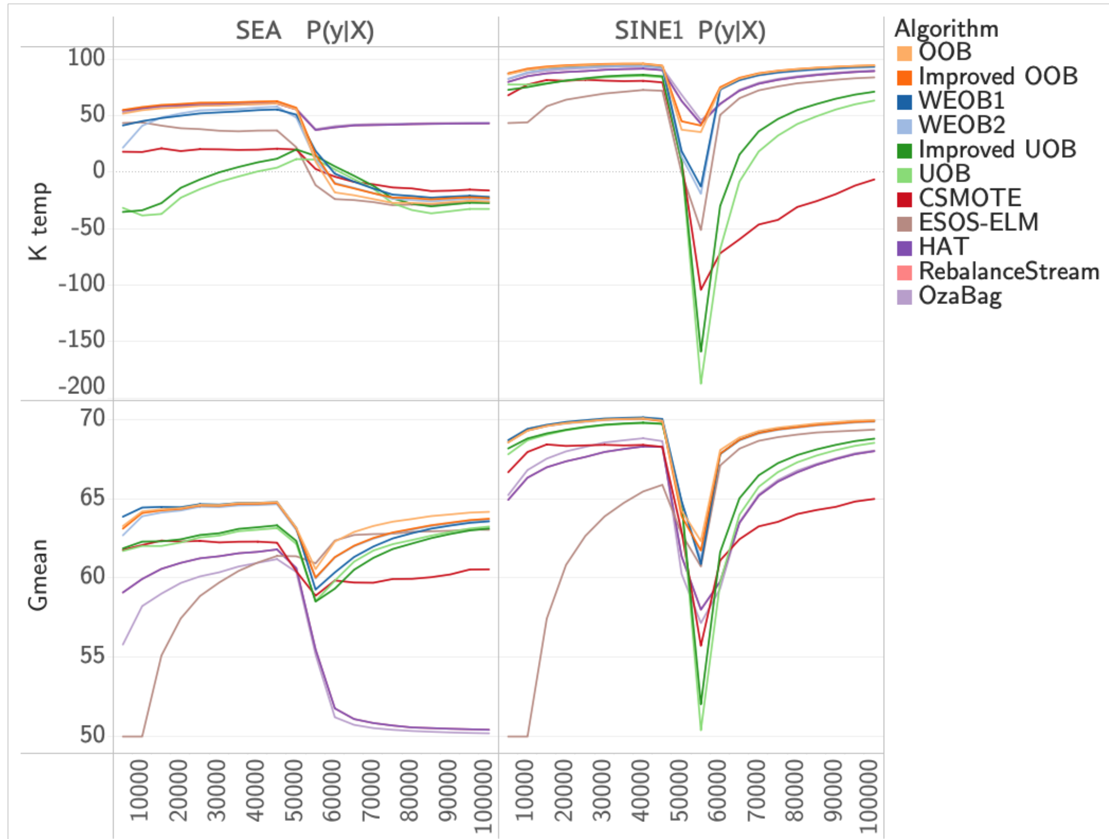


Figure 6.5: K temporal and Gmean during SEA and SINE1 streams having a P(y|X) drift.

- *Ensembles*: the ensembles of the improved versions have similar performances, the WEOB1 performing better on the recall of the minority class and the WEOB2 performing better on the recall of the majority class. In the plots in Figure 6.4 can be seen that with a high imbalance ratio their performances are always near the best performing algorithm during both the recalls thus achieving a better Gmean than both the base algorithms.
- *C-SMOTE and RebalanceStream*: Figure 6.3 shows that C-SMOTE starts to increase significantly the Hoeffding Adaptive Tree performances on the minority class only from a *2:8* imbalance ratio. Figure 6.4 highlights that it is the worst performing with respect to the majority class. A major problem of this algorithm is shown by its K temporal statistic in figure 6.5, when the drift is severe like in SINE1 streams, where the class concepts swap, it cannot recover its accuracy and its Kappa temporal statistic remains under the level of a no-change classifier. The ADWIN drift detector only looks at the class

label of the seen samples thus is aware only of changes in the imbalance ratio and it is not emptied when a P(y|X) happens. This makes the algorithm generate synthetic samples with the old concept. RebalanceStream, on the other hand, is not able to detect any drift and its performances are exactly the same as the Hoeffding adaptive tree.

- *ESOS-ELM*: as shown in 6.4 and 6.5 this algorithm needs several instances to pass before being able to have accurate predictions. The recall plots in Figure 6.4 show that when the imbalance is *1:9* it is the algorithm with the best recall of the minority class just after the drift but it has a significant worsening of the recall of the majority class.

6.3 P(y) drift

The P(y) drift affects the imbalance ratio. Each data stream with this kind of drift has different level of change. In the list below, I details the different ratios. I separate the imbalance ratio before and after the drift with *"−"* and the positive and negative class probability with ":". Moreover, *"m"* identifies the minority class ratio of the stream and the *"M"* the majority one:

- Disappearing minority: *m:M - 0:1*;
- Appearing minority: *0:1 - m:M*;
- Minority share: *0:1 - m:M*;
- SEA P(y): *0.5:0.5 - m:M*; and
- SINE P(y): *m:M - M:m.*

I present the plots of the average Gmean measure to give a general view of the algorithms' performances in Figure 6.7. I show the plots of each class recall in Figure 6.7 and Figure 6.6, I identify as "minority" the first class appearing as minority class on the data streams.

- *Online Bagging*: Figure 6.7 highlights that, with this kind of drift, the rebalancing algorithms improve the OzaBag performances after the drift only with an imbalance ratio of *2:8*. Figure 6.6 shows that, with streams that start with a zero probability for the minority class, the rebalancing phase becomes important for a fast minority class recall when the samples start appearing. In these cases, which are *appearing-minority* and *minority-share* the improved version of UOB works really well outperforming the other algorithms on the

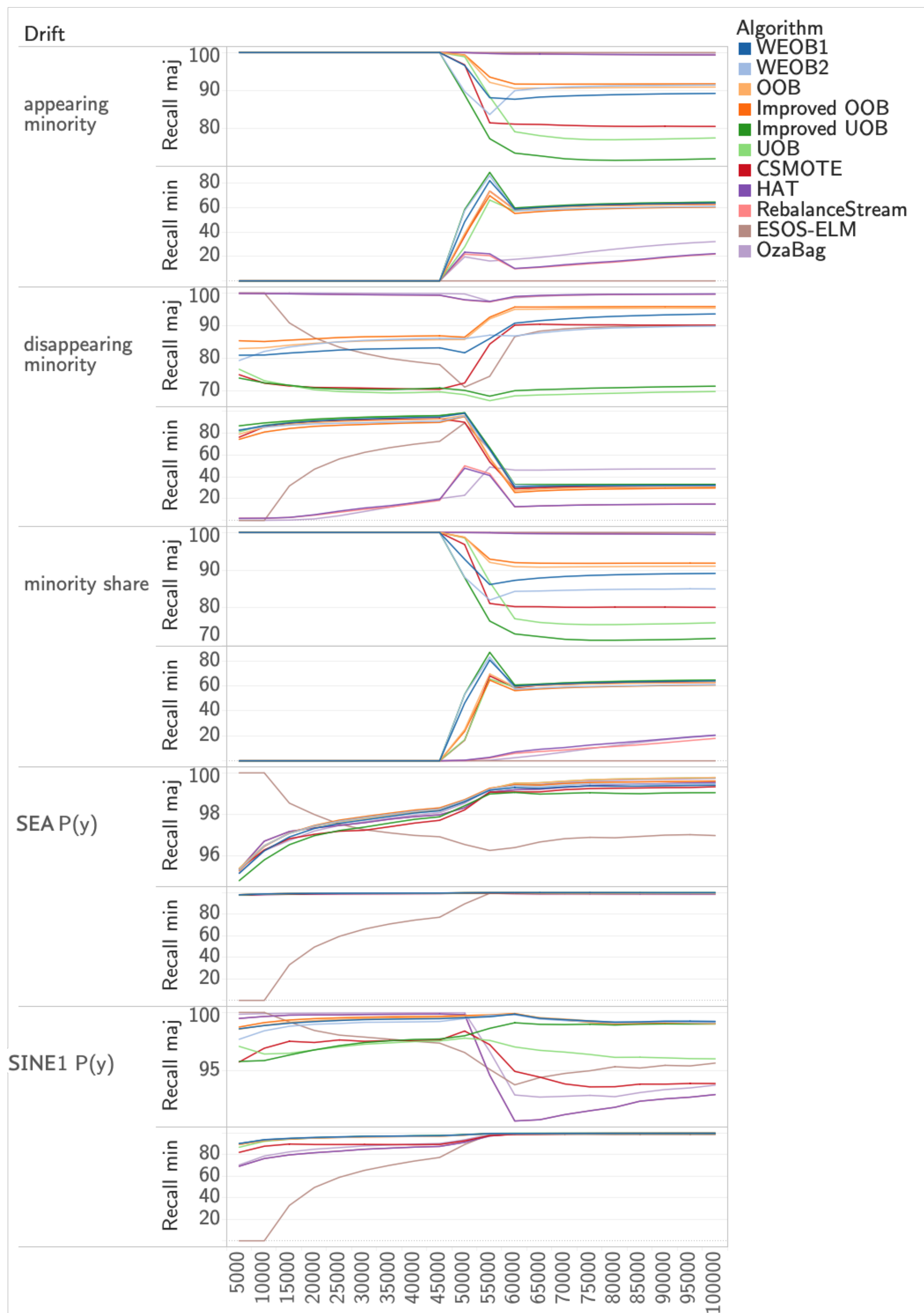


Figure 6.6: Recall during streams having a P(y) drift with imbalance ratio *1:9*.

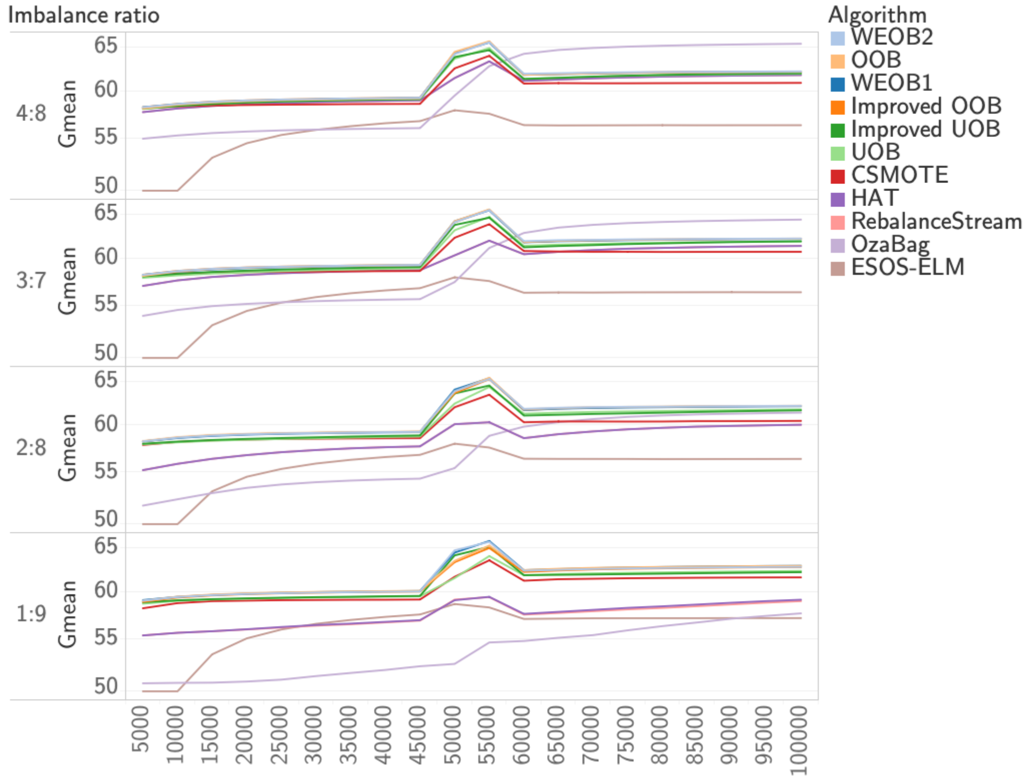


Figure 6.7: Gmean during streams having a P(y) drift

recall of the minority class after the drift. As already seen before, OOB and its improved version improve the recall of the minority class w.r.t. OzaBag without losing too much recall of the majority class. The ensembles achieve the best performances having a fast recall of the minority class after the drift, close to the one of the Improved UOB, and keeping a recall of the majority class similar to the oversampling algorithms thus achieving the best Gmean.

- *C-SMOTE and RebalanceStream*: C-SMOTE always improves the performances of the Hoeffding Adaptive Tree on the minority class. During the clusters drifts it achieve the recall of the minority class of the undersampling algorithms but with a better recall of the majority class. RebalanceStream on the other hand does not seem to improve the Hoeffding Adaptive Tree performances.
- *ESOS-ELM*: as shown in Figure 6.7, this algorithm needs time to adjust its predictions but it is always the worst-performing. More importantly, when the stream starts without the minority class, this algorithm is not able to

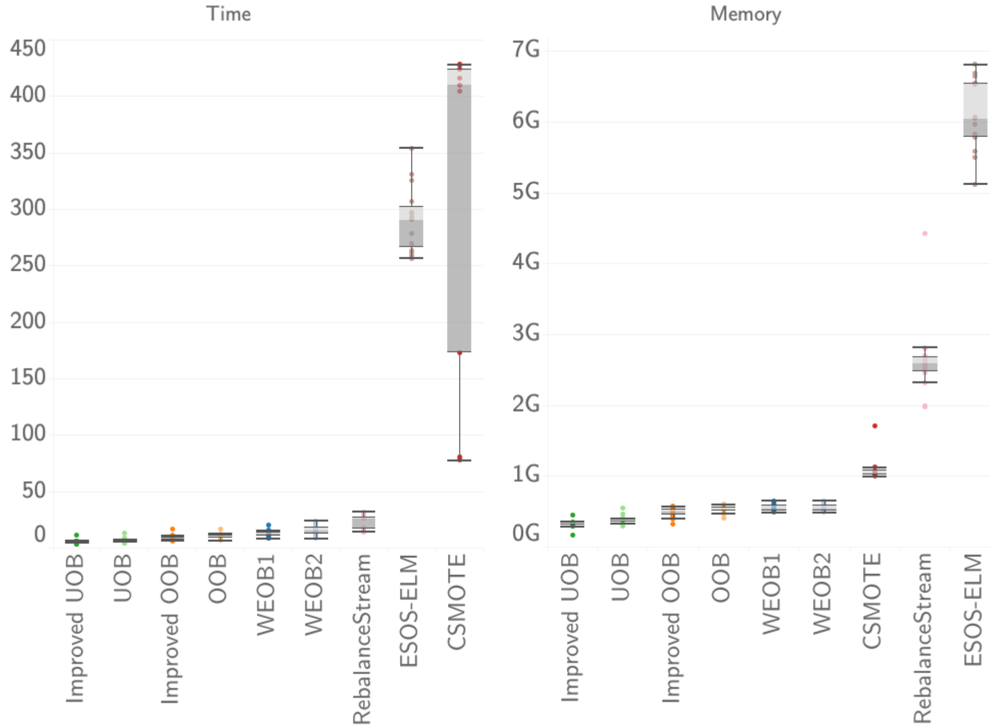


Figure 6.8: Time and Memory requirements of each algorithm.

identify any of its samples when they start appearing.

6.4 Resources requirements

Each algorithm needs a different amount of resources. Being able to estimate the required amount is useful when they are limited. Figure 6.8 shows both the mean time, measured in seconds, and memory, measured in bytes, of the each experiments, grouped by algorithm.

The online bagging algorithms outperform the others from the resource point of view. They do not require saving any sample or label, other than time-decaying variables. The greater resources are required by the ensemble algorithms which need the Smoothing Recalls windows but their size is not proportional with the seen samples thus never becoming too big. ESOS-ELM and C-SMOTE are the worst-performing from a timing point of view. The slow part of C-SMOTE is the K-NN search when applying the online smote: the time to generate a new sample is proportional to the window size, this makes the algorithm stuck when this size becomes too big. A possible improvement can be limiting the maximum

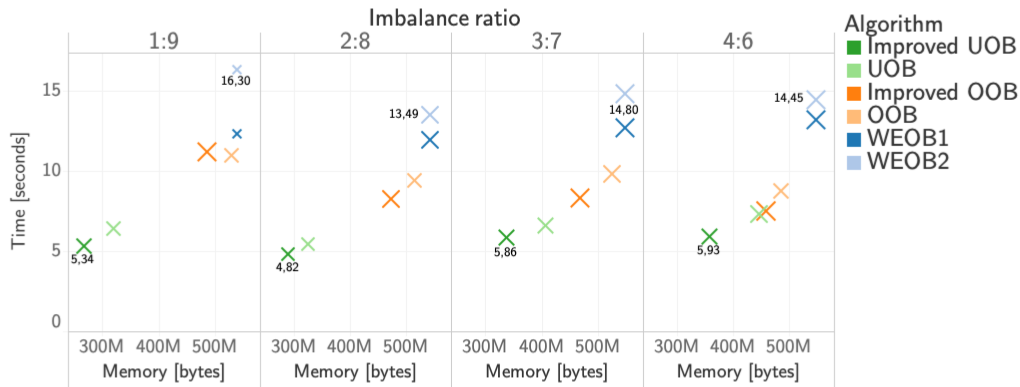


Figure 6.9: Mean Time and Memory requirements of the Online Bagging based algorithms.

number of samples on the window making it more efficient also during the concept drifts. ESOS-ELM requires completely different mathematical operations than an Hoeffding Tree thus it is not comparable. The other plot shows a memory problem of RebalanceStream which requires keeping a batch with all the seen samples since the last drift.

In order to better compare the online bagging based algorithms I show a plot restricted to them in Figure 6.9. The higher is the imbalance ratio the more the undersampling algorithms improve the time and memory requirements with respect to the others. Improved UOB is the one requiring less time and less memory. During streams with an imbalance ratio of *1:9* it requires half of the time needed by the oversampling algorithms. Both the improved versions achieve better time and memory consumption than the corresponding native versions.

6.5 Real datasets

As explained in Section 4.2.3, in order to confirm the obtained results, I tested the algorithm with three real datasets.

Figure 6.11 and Figure 6.10 show different algorithm performances with every dataset. I details the results for each detaset in the following list.

- The *Electricity* dataset has a constant and low imbalance ratio. In this case, the undersampling algorithms do not improve the recall of the minority class but only worst the one of the majority class. RebalanceStream behaves exactly like a standard Hoeffding Adaptive Tree, which is better than what C-SMOTE does, worsening both the recalls. The best performing is the native OOB which has the best Gmean and both the best recalls if we do not

Table 6.1: Average statistics with real datasets.

Dataset	Algorithm	min Recall	maj Recall	Gmean	min Fscore	maj Fscore
Electricity	OzaBag	$82,5 \pm 8,9$	$91,3 \pm 3,5$	$65,9 \pm 1,8$	$84,9 \pm 6,9$	$89,5 \pm 2,7$
	OOB	**$88,6 \pm 5,9$**	$91,4 \pm 3,2$	**$67,1 \pm 1,3$**	**$88,5 \pm 4,7$**	**$91,4 \pm 2,5$**
	UOB	$82,5 \pm 8,4$	$90,5 \pm 3,8$	$65,7 \pm 1,7$	$84,6 \pm 6,1$	$89,0 \pm 2,8$
	Improved OOB	$87,1 \pm 6,9$	$90,8 \pm 3,6$	$66,7 \pm 1,5$	$87,3 \pm 5,3$	$90,6 \pm 2,7$
	Improved UOB	$80,9 \pm 9,1$	$87,4 \pm 6,3$	$64,8 \pm 1,6$	$82,1 \pm 6,4$	$86,8 \pm 2,6$
	WEOB1	$84,7 \pm 7,6$	$90,1 \pm 3,9$	$66,1 \pm 1,5$	$85,7 \pm 5,7$	$89,6 \pm 2,4$
	WEOB2	$86,8 \pm 6,9$	$90,4 \pm 4,5$	$66,5 \pm 1,7$	$87,1 \pm 5,7$	$90,4 \pm 3,3$
	HoeffdingAdaptiveTree	$79,4 \pm 7,4$	$85,5 \pm 5,7$	$64,2 \pm 2,1$	$80,1 \pm 7,1$	$84,7 \pm 4,8$
	C-SMOTE	$77,4 \pm 10,4$	$85,3 \pm 7,4$	$63,7 \pm 2,5$	$78,4 \pm 8,8$	$83,8 \pm 5,6$
	RebalanceStream	$79,4 \pm 7,1$	$85,5 \pm 5,4$	$64,2 \pm 2,0$	$80,1 \pm 6,7$	$84,7 \pm 4,6$
	ESOS-ELM	$1,4 \pm 1,5$	**$99,3 \pm 0,7$**	$54,3 \pm 8,2$	$2,6 \pm 2,7$	$71,5,8 \pm 8,4$
PAKDD'90	OzaBag	$0,2 \pm 0,3$	**$99,9 \pm 0,2$**	$50,0 \pm 0,0$	$0,4 \pm 0,6$	$89,2 \pm 1,4$
	OOB	$58,9 \pm 4,6$	$61,2 \pm 5,1$	$54,8 \pm 0,9$	$37,0 \pm 2,4$	$71,5 \pm 3,4$
	UOB	**$83,0 \pm 3,8$**	$34,2 \pm 5,3$	$54,1 \pm 0,7$	$36,4 \pm 2,4$	$49,2 \pm 5,3$
	Improved OOB	$39,8 \pm 6,1$	$76,8 \pm 6,3$	$54,0 \pm 1,1$	$33,6 \pm 2,8$	$80,1 \pm 3,0$
	Improved UOB	$70,0 \pm 7,9$	$50,1 \pm 9,7$	$54,8 \pm 0,9$	**$37,2 \pm 2,2$**	$63,0 \pm 8,4$
	WEOB1	$58,9 \pm 8,0$	$62,1 \pm 7,7$	$55,0 \pm 0,8$	$36,9 \pm 2,1$	**$71,8 \pm 5,2$**
	WEOB2	$62,8 \pm 7,5$	$55,8 \pm 8,6$	$54,4 \pm 1,3$	$36,7 \pm 2,7$	$67,5 \pm 5,8$
	HoeffdingAdaptiveTree	$0,5 \pm 0,5$	$99,8 \pm 0,3$	$50,1 \pm 0,1$	$1,0 \pm 1,0$	$89,2 \pm 1,4$
	C-SMOTE	$75,1 \pm 8,4$	$39,2 \pm 13,2$	$53,4 \pm 2,1$	$34,8 \pm 2,9$	$52,6 \pm 12,4$
	RebalanceStream	$0,5 \pm 0,5$	$99,8 \pm 0,3$	$50,1 \pm 0,1$	$1,0 \pm 1,0$	$89,2 \pm 1,4$
	ESOS-ELM	$45,1 \pm 23,1$	$62,5 \pm 19,3$	**$56,0 \pm 7,4$**	$25,9 \pm 13,2$	$69,7 \pm 10,5$
KDDCup'99	OzaBag	$99,8 \pm 0,2$	$96,9 \pm 12,0$	$69,6 \pm 7,2$	**$96,2 \pm 8,7$**	$97,7 \pm 10,8$
	OOB	$99,8 \pm 0,3$	$97,1 \pm 11,0$	$69,8 \pm 6,0$	$94,3 \pm 15,6$	$97,8 \pm 9,9$
	UOB	$99,5 \pm 1,2$	$94,3 \pm 18,5$	$69,2 \pm 6,7$	$95,0 \pm 13,3$	$94,0 \pm 18,0$
	Improved OOB	**$99,9 \pm 0,2$**	$97,4 \pm 8,9$	$70,2 \pm 1,7$	$93,9 \pm 14,6$	**$98,1 \pm 7,7$**
	Improved UOB	$96,0 \pm 2,6$	$98,4 \pm 10,1$	$69,2 \pm 7,0$	$89,6 \pm 21,2$	$92,1 \pm 19,2$
	WEOB1	$99,1 \pm 0,9$	$98,9 \pm 3,5$	$70,3 \pm 0,7$	$91,1 \pm 22,6$	$95,6 \pm 13,4$
	WEOB2	$99,3 \pm 1,5$	$97,7 \pm 9,7$	$70,0 \pm 3,6$	$95,1 \pm 11,2$	$96,0 \pm 12,9$
	HoeffdingAdaptiveTree	$99,4 \pm 0,5$	$97,1 \pm 8,9$	$70,1 \pm 1,6$	$94,2 \pm 4,6$	$96,9 \pm 9,6$
	C-SMOTE	$98,6 \pm 1,2$	$97,0 \pm 10,1$	$69,9 \pm 1,8$	$93,5 \pm 7,3$	$96,8 \pm 10,2$
	RebalanceStream	$99,4 \pm 0,5$	$97,0 \pm 9,1$	$70,1 \pm 1,7$	$94,1 \pm 4,6$	$96,8 \pm 9,8$
	ESOS-ELM	$0,0 \pm 0,0$	**$100,0 \pm 0,0$**	**$70,7 \pm 0,0$**	$0,0 \pm 0,0$	$83,1 \pm 35,5$

consider ESOS-ELM which predicts everything as belonging to the majority class. Table 6.1 also clearly shows how OOB is the best algorithm with this dataset achieving also the best Fscores.

- The *PAKDD* dataset has a more skewed class imbalance ratio and confirms that in such cases the UOB outperforms the others with respect to the recall of the minority class achieving a result of $83,0$. As shown in Figure 6.11 the best Gmean is achieved by WEOB1 which has a good trade-off between the recalls. C-SMOTE greatly improves the performances of a single Hoeffding Adaptive Tree which without any rebalancing achieve zero recall of the minority class.

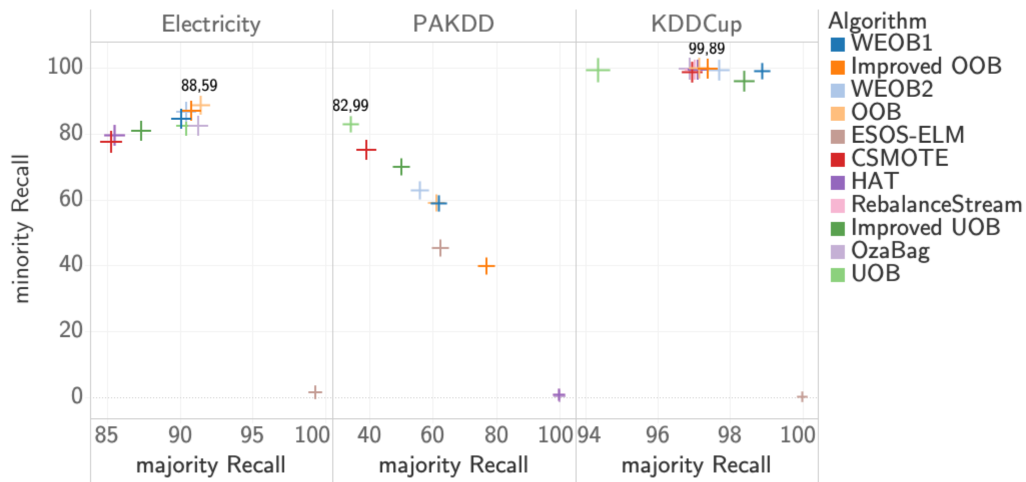


Figure 6.10: Recall average with real datasets.

- The *KDDCup* dataset is based on anomaly detection and it has high imbalance ratio changes. ESOS-ELM as with this kind of artificial data streams performs really bad and it is not able to detect any minority class example. Improved OOB reaches almost a perfect recall of the minority class with a mean value of 99.899, improving both the recalls w.r.t. OzaBag. The Native UOB on the other hand worsens both the recalls, the Improved version instead achieve a good recall of the majority class. The best Gmean is again achieved by WEOB1.

Figure 6.11: Gmean during real datasets.

Chapter 7

Conclusions and Future Work

This thesis aimed to select the best online learning algorithm able to deal with imbalanced data streams with concept drifts. I built a benchmarking environment to make the performed experiment easily repeatable. It is now easy to try other configurations and expand this study to new settings with the same environment.

The studies [49] and [50] on the online bagging techniques focused only on the P(y) drift where the undersampling is able to achieve greater performances. This thesis confirms their results but also shows that, with the other categories of drift, the undersampling algorithms are the ones that reduce the most the recall of the majority class. The algorithm that has proved to be the most solid was the original version of OOB. The two ensembles have always performed well. WEOB2 has been confirmed to be the best one during P(y) drifts as described in its paper [50] while WEOB1 has achieved great performances on both classes recall during P(y|X) drifts. Being the resource requirements really low, new ways to better combine the two algorithms could be studied. For example, use an adaptive window to compute the smoothed recalls in order to always choose the best performing algorithm in the current concept and recovering as fast as possible from every type of drift. As future work, an ensemble of the original version could be implemented in order to exploit the highest recall of the minority class of the UOB and the highest Gmean of the OOB.

Regarding the SMOTE-based algorithms, RebalanceStream wasn't able to detect almost any drift and thus its performances were identical to a simple Hoeffding Adaptive Tree except for its worst resource consumption. I think that an improvement could be obtained by using a different drift detection algorithm or using ADWIN to monitor the predictions' accuracy. Also keeping a batch of all the seen samples from the last drift can become too expensive and moreover it doesn't respect the memory constraint of the streaming algorithms. It can be useless since

it will be used when a new concept has started. I think that using only the resetBatch in order to rebalance the data stream could be enough because SMOTE will generate unique synthetic samples using only the most recent real ones, thus having a greater probability of them belonging to the new concept and using less time and memory to search for the k-nearest neighbors. The performance comparison, after the drift has been confirmed, can be executed between the base learner, a copy of the base learner trained with a balanced resetBatch and two new learners trained one only with the resetBatch and one with the reset batch balanced. This could achieve similar performances to the existing RebalanceStream algorithm but it will be for sure less expensive in terms of memory and time resources.

C-SMOTE, on the other hand, has been proven really useful when the imbalance ratio increases more than *2:8*, but it has shown difficulties with the P(y|X) drift. Another problem affecting C-SMOTE is its time requirement. New solutions like a limited window size or a faster synthetic sample generation could solve these two problems and make it a great algorithm challenging the online bagging ones.

ESOS-ELM has been the worst for what concerns the resource consumption and has not been able to exceed any other algorithm in any particular drift and has been proven slow to reach a good accuracy. The reason could be that it is really different from the others and it has a greater number of hyper-parameters to set. Moreover, it is the only one that couldn't be tested with the Hoeffding Adaptive Tree algorithm as base learner. I think that more studies and experiments need to be run with this algorithm.

As future work, it will be important to explore higher imbalance ratios on the data streams in order to identify the algorithms suited for more difficult but realistic scenarios. Both this study and the one that presented the online bagging algorithms used an Hoeffding Adaptive Tree as base learner. I think could be interesting to study how different base learners reacts to the various resampling techniques to deal with class imbalance. New algorithms and data streams can be easily integrated, enlarging the benchmark also to multi-class settings. Moreover, this benchmark can be improved running the experiments in parallel processes and extended to automate the hyper-parameter tuning process.

Appendix A

IEBench: Benchmarking Streaming Learners on Imbalanced Evolving Data Streams

Anonymous Author(s)

Anonymous

Abstract. Nowadays, data coming from electronic devices surround us, and the ability to analyze all of them in real-time is a big challenge and the starting point to understand where to innovate. A possible solution is Streaming Machine Learning (SML). This new approach focuses on data streams, unbounded sequences of data arriving in sequential order, even once at a time. The two challenges to face in the real-world are concept drift and class imbalance. The former refers to the changes in the characteristics of the data, while the latter refers to an unequal distribution between the classes. In this paper, we propose IEBench, an easy-to-use benchmarking environment for comparing streaming learners on artificial and real data streams. We implemented various state-of-art algorithms, and we conducted a comprehensive experimental campaign with IEBench. We evaluated the algorithms on artificial and real data streams with different imbalance levels and concept drift types. We collected empirical evidence of the role and impact of existing methods for rebalancing data streams in improving performances during different types of concept drift. IEBench eases the practitioners' task of testing existing algorithms on a new data stream and the scientists' one of developing new algorithms and systematically comparing them with the state-of-the-art.

Keywords: Streaming Machine Learning · Concept Drift · Class Imbalance · Benchmarking Environment

1 Introduction

The Streaming Machine Learning (SML) approach focuses on data streams, sequences of data arriving once at a time. Its goal is to i) get information in real-time without the need to come back to already seen data, and ii) use a limited amount of resources such as memory and time.

As identified by Krawczyk et al. [20], two significant challenges in this new field are the concept drift phenomenon [29] and the class imbalance [15]. Data streams evolve, and the distributions of the data can change. Models built on old data can become inconsistent. This phenomenon is called concept drift and can happen in various forms and speeds. The second challenge concerns classification tasks where data streams have an unequal distribution between the classes. Since the instances in the minority class(es) rarely occur, the patterns for classifying

these classes tend to be rare, undiscovered, or ignored. This kind of information is rare but essential, and models need to consider it.

The range of streaming analysis applications, where both the concept drift and class imbalance challenges appear, is broad. An example is the Internet of Things (IoT) field. Every day, new sensors are placed in houses to improve security [8], in cities to monitor the mobility of people [25], and in industries to watch processes [18]. As presented by this previous study, sensors' goal could detect anomalies in finished products. Defects can change frequency and location in the product's surface based on which industrial machine is damaged, causing concept drifts. Moreover, anomalies are usually rare events that require particular attention from the models to maintain high-quality standards. Another possible field is real-time healthcare. For example, Kathy Lee et al. [22] proposed a real-time tweet mining system to monitor the spread of flu and cancer, Hussain et al. [17] proposed a real-time monitoring system for stroke prognosis, and Hassan et al. [14] proposed a real-time monitoring system for predicting the health status of diabetes patients using wearable sensors. In general, a patient's vital signs could change at any moment without any warnings, and the algorithms for the classification of the general health status need to change accordingly. Besides, systems predicting multiple diseases need to be rebalanced with respect to these diseases' frequencies and dangerousness.

The majority of the SML state-of-art solutions focus on solving only the class imbalance problem in an online fashion, leaving their pipelined algorithm the task of handling the concept drift occurrence. However, the impacts that the rebalance phase has on the concept drift detection are still under investigation.

For these reasons, our research aims to provide empirical answers to the following research questions:

- RQ1 *Does rebalancing affect in a different way the performances of the algorithms for each concept drift type?*
- RQ2 *How do the different state-of-art rebalancing techniques affect the performances of the algorithms during evolving data streams?*
- RQ3 *How expensive in terms of time/memory consumed each rebalancing technique is?*

In particular, the contributions of this paper are:

- IEBench, a benchmarking environment for easily and effectively comparing SML algorithms performances;
- Empirical evidence that applying rebalancing techniques improves the algorithms' performances during each type of concept drift (answering RQ1);
- An extensive experimental campaign measuring the performances and resource requirements of the state-of-art algorithms able to deal with the concept drift and class imbalance (answering RQ2 and RQ3).

The remainder of this paper is organized as follows. Section 2 presents IEBench, a benchmarking environment for streaming algorithms. Section 3 presents the implemented and tested state-of-art algorithms. Section 4 describes

the artificial and real data streams used to test the algorithms. Section 5 describes the algorithms and evaluation settings used during the experiments. Section 6 presents the result obtained. Finally, Section 7 discusses the conclusions and outlines some directions for future improvements of this work.

2 Benchmarking environment

Applying streaming algorithms to real-world scenarios, to choose the one that better suits the problem at hand, there is the need to have a complete view of their performances. Benchmarking is a process of comparing algorithms using all the relevant measures and its development is composed by three phases: design, execution, and analysis. The design phase consists of choosing the data sources, the candidate algorithms, and the essential measures, ensuring the experiments' repeatability. Running the designed experiments in the appropriate computational environment and collecting all the results is the second phase. The last phase consists of exploring the gathered data and aggregating them to compare the algorithms under all the relevant aspects. Furthermore, a benchmark's economic costs need to be kept under the "worth of investment" threshold. As pointed out by Krawczyka et al. [20], there is the need for a framework to evaluate data stream classifier. We propose IEBench, a benchmarking environment which allows to automatically test multiple streaming learners on several streams.

The two most common real-world data streams challenges are concept drift and class imbalance. Studies regarding the state-of-art algorithms dealing with the class imbalance problem in data streams focus only on the class imbalance ratio changes. We used IEBench to study and compare how the online rebalancing techniques affect the algorithms' performances during the various types of concept drifts. We tested them using artificial data streams with thirty-one concept drifts that differ in type, speed, and severity with four imbalance ratios. Also, we used three real imbalanced and evolving data streams. We collected evidence that solving the imbalance problem can significantly improve the performances during the different concept drift types.

IEBench is developed to compare all the SML algorithms implemented in MOA and it is publicly available here[1].

3 State of Art algorithms

In recent years, different data-level and ensemble-based methods have appeared to handle class imbalance in data streams. The former refers to algorithms that manipulate the input data to rebalance the input distribution. In contrast, the latter refers to exploiting an ensemble of learners to rebalance the data distribution, assigning samples differently to each base learner.

[1] https://drive.google.com/drive/folders/1-zGny2FS7VO9vlcOJkSEDrxoiEr-cqQM?usp=sharing We intend to publish it on Github on paper acceptance.

Bagging [6] is an ensemble technique that trains multiple models with a data set containing K copies of each of the training data where K is drawn from a binomial distribution. Oza et al. [26] also proposed an online version that considers the unbounded nature of data streams. The innovation lies in the adoption of a bagging method that uses a Poisson(1) distribution. Theoretically, the Poisson distribution is a limiting case of the binomial distribution that arises when the number of trials increases indefinitely.

Wang et al. [30] proposed an ensemble solution to class imbalance based on online bagging. The idea is to make an ensemble of base learners where the classes are balanced for each of them. The rebalancing is achieved by adapting the lambda of the Poisson distribution based on each sample's class. Since the class balancing can be done undersampling the majority class or oversampling the minority class, Wang et al. proposed the Oversampling Online Bagging (OOB) and Undersampling Online Bagging (UOB) techniques. Both of them use two time-decaying variables for each class: w_k, which denotes the size percentage of class k, and R_k, which denotes the model's accuracy on class k. When two classes have the w_k difference greater than a threshold δ_1 $(0 < \delta_1 < 1)$ and the R_k difference greater than a threshold δ_2 $(0 < \delta_2 < 1)$, the small class is labeled as the minority and the large class is labeled as majority. After comparing all the classes, the unlabeled ones are treated as normal. This procedure leads to three label sets: minority, majority, and normal. When a sample belongs to the minority class, OOB updates each learner a number of times drawn from a Poisson distribution with $\lambda = 1/w_k$. At the same time, UOB uses $\lambda = 1 - w_k$ when a sample belongs to the majority class.

A more recent study proposed the Improved Oversampling Online Bagging (IOOB) and Improved Undersampling Online Bagging (IUOB) techniques [31]. They use a new method to set λ based on the two classes' size ratio. Considering only two classes, and naming w_{maj} the size of the majority class and w_{min} the size of the minority class, IOOB sets λ to w_{maj}/w_{min} for the minority class while IUOB sets λ to w_{min}/w_{maj} for the majority class.

Wang et al. [31] also presented two ensemble strategies to combine the strength of IOOB and IUOB. To weigh the two algorithms' predictions, the methods compute their Gmean values using a moving average of the recalls. WEOB1 uses the normalized Gmean values as weights to calculate a weighted sum of their predictions, while WEOB2 compares the Gmean values and uses only the model's prediction with the higher one. All these bagging-based strategies leave the concept drifts detection in the data distribution to the base learners.

C-SMOTE (CS) [2] is, instead, a data-level method. CS extends the oversampling technique SMOTE [9] and collects the class labels in a window managed by the ADWIN [3] drift detector. To rebalance the stream, CS uses two windows: one called W, which keeps the data samples, and one called W_{label}, which keeps the corresponding labels. Every time a new sample arrives, CS checks the class ratio and, if it is less than a certain threshold t, a continuous online SMOTE version is applied until the minority sample ratio is greater than the threshold. When ADWIN detects a change in the class imbalance ratio, CS resizes the two

windows. CS leaves the detection of the concept drifts in the data distribution to the base learner.

RebalanceStream (RS) [1] is another data-level solution based on SMOTE. RS starts with a single base learner. It collects incoming data in a batch and the corresponding class labels in a window managed by ADWIN. When it detects a drift warning in the imbalance ratio, the algorithm starts collecting samples in a new batch called reset-Batch. When the change is confirmed, RS trains three new learners in parallel: i) the first one only with the reset-Batch, ii) the second one with the reset-Batch balanced with SMOTE, and iii) the third one with the original Batch rebalanced with SMOTE. RS chooses the one with the best K-Statistic value [10] as the new learner, replaces the active model, drops the other models, and reset both the Batch and the reset-Batch. RS also leaves the detection of concept drifts in the data distribution to the base learner.

ESOS-ELM [24], has been proposed as an OS-ELM [23] based ensemble able to handle both the class imbalance and the concept drift problems. The OS-ELM is a sequential version of ELM which updates the model with data chunks. Extreme learning machine (ELM) [16] is a single-step learning algorithm for a single hidden-layer feed-forward network (SLFN). The ensemble's idea is to keep two sorted lists of the classifiers, one ordered w.r.t. the number of minority data samples processed, and one ordered w.r.t. the number of majority data samples processed. When a new chunk of data is ready for the training, the imbalance ratio m is computed, the first m classifiers of the first list will process all the minority class data samples, while the top m classifiers of the second list will process $1/m$ of the majority class data samples each. Every learner will have a balanced training chunk. To handle the concept drift, it uses the Dynamic Weighted Majority technique [19]. It adds a new OS-ELM learner initialized using the WELM initialization when the ensemble misclassifies a sample. A long-term buffer, called ELM-Store, is used to save past concepts. When it detects a concept drift from the Gmean of the ensemble falling under a threshold θ, it trains and stores a WELM. When a stored classifier performs better than the ensemble, it introduced back the stored classifier with weight 1.

4 Data streams

We tested the algorithms with artificial and real data streams with two classes and different drift types, class imbalance levels, and data distributions. As artificial data streams, we used the SEA [27], SINE1 [13], and Cluster [7] generators. We generated $100,000$ instances for each stream. In particular, the incremental and periodic drifts start at the time step $45,000$ and take $10,000$ time steps to complete, while the sudden drift happens at time step $50,000$. As real data streams, we tested the PAKDD [28], KDDCup [11] and Electricity [13] streams.
SEA and SINE1 To reproduce each of the different types of concept drifts [12], we choose two widely used artificial data generators: SINE1 [13] and SEA [27]. SINE1 instances are composed by two attributes (x_1, x_2) uniformly distributed in $[0, 1]$. The class is determined by $x_2 - \sin x_1 \leq \theta$, where θ is a threshold value.

SEA instances, instead, are composed by three attributes (x_1, x_2, x_3) uniformly distributed in $[0, 10]$. The label is a function of only two of them, while the third one is just noise. The equation to determine the class label is $x_1 + x_2 \leq \theta$, where θ is a threshold value. We consider the following types of concept drift.

$P(y)$ Concept Drift. The probability of seeing any data example from the class y is changing. It can affect the algorithms' performances due to a change in the class imbalance status, but it does not necessarily shift the decision boundary. Data streams generated by SINE1 have a severe class imbalance change, in which the imbalance ratio of the first half is reversed on the second half. Data streams generated by SEA have a less severe change, in which the data streams are balanced during the first half and become imbalanced during the latter half.

$P(X|y)$ Concept Drift. The probability of seeing a data sample X is changing but its label y is not. It shows that we see new data samples from the same environment, and the drift does not affect the decision boundary. The concept drift in each data stream is defined by a change on a constraint on the x_1 parameter for the negative class (0) instances. During the first half of the stream the probability is $p(x_1 < n) = 0.9$ while during the second half, it is $p(x_1 < n) = 0.1$. The data stream is constantly imbalanced.

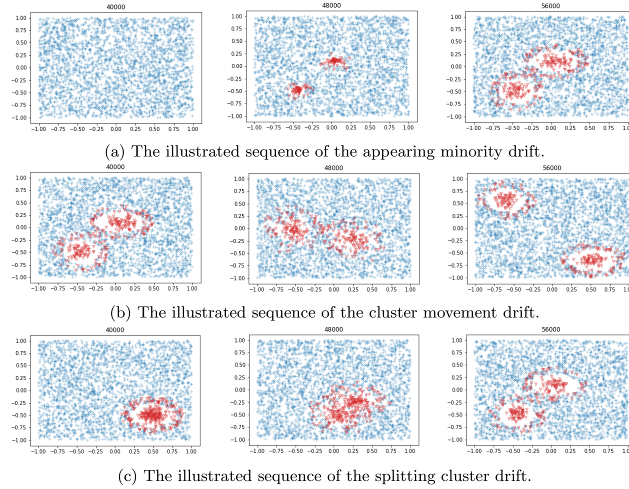
$P(y|X)$ Concept Drift. The probability of a data sample X to belong to a particular class y is changing. This drift will cause the decision boundary to shift, and, as a consequence, it will lead the algorithm performance to deteriorate. Data streams generated by SINE1 have a concept swap. In contrast, data streams generated by SEA have a concept drift due to a θ value change making it is less severe than the change in SINE1 because some of the examples from the old concept are still valid under the new concept.

We generated sixteen data streams for every concept drift: eight of them with SEA and eight with SINE1. Each one has a different combination of imbalance ratio, *1:9, 2:8, 3:7,* or *4:6*, and speed, *sudden* or *incremental*.

Cluster generator The study [20] indicate the need for more complex drifts and class distributions. This stream generator [7], originally implemented by Dariusz Brzeziński [2], meets these requirements. It consists of an imbalance generator that manages how the data is distributed in the feature space (two dimensional in the experiments) and a drift generator that manages how to perform the drifts from one distribution to the next. The main characteristic of the imbalance generator is the data distributions of the two classes. The *majority* class data points spread over all the feature space except in the areas where the *minority* class clusters are. These clusters can vary in number, shape, and size. There are two minority clusters with a normal distribution inside the cluster in the experiments. The generator places the minority samples with an equal probability in the "safe" and "borderline" regions. The "safe" one is near the centroid of the cluster where only minority samples are. The "borderline" one, which has radius bigger than the one of the safe zone, can include majority samples.

The drift generator can cause nine different types of drift. We performed experiments with each of them using four imbalance ratios (*1:9, 2:8, 3:7, 4:6*)

[2] https://github.com/dabrze/imbalanced-stream-generator



(a) The illustrated sequence of the appearing minority drift.

(b) The illustrated sequence of the cluster movement drift.

(c) The illustrated sequence of the splitting cluster drift.

Fig. 1: Examples of cluster drifts

and three different drift speeds. In the *sudden*, the drift happens at the 50,000th time step. In the *incremental*, it starts at the 45,000th and ends at the 55,000th . In the *periodic*, it starts at the 45,000th , goes to the next distribution until the 50,000th , and comes back at the original one at the 55,000th . We generated the following types of concept drift.

- *Appearing minority.* This drift, presented in Figure 1a, starts with zero positive instances. The minority clusters start appearing as the drift ends, increasing both the clusters' radius and the minority samples probability.
- *Minority share.* This drift is similar to the appearing minority because the minority appears as the drift progresses. The difference is that the clusters are already there but empty.
- *Disappearing minority.* This drift consists of making the minority clusters radius and the minority samples probability decrease to zero as the drift progress to its end. It is exactly the opposite of the appearing minority drift.
- *Appearing cluster.* This drift consists of making a new cluster appear without any modification to the one already present.
- *Cluster movement.* This drift, presented in Figure 1b, ends with the minority clusters in a different random position. The movement's progress follows the line from the original to the target position.

- *Cluster jitter.* This drift consists of making the minority clusters center move randomly, giving the clusters a shaking effect. The cluster will be in a different position at the end of the drift.
- *Splitting cluster.* This drift, presented in Figure 1c, starts with one minority cluster that will split in two while moving to two different target positions.
- *Shape Shift.* This drift consists of making the minority clusters change shape, increasing the radiuses, and producing wider shapes.
- *Borderline shift.* This drift consists of making the borderline radius of the minority clusters grow through the center, making the safe radius shrink.

Real data streams We tested the algorithms on the following real data streams to confirm the artificial data streams' results.

PAKDD [28] stream focuses on the problem of credit risk assessment. It has 50,000 instances composed by twenty-seven features, thirteen categorical and fourteen numerical. The percentage of positive class instances is 20%.

Electricity [13] stream is collected from the Australian New South Wales Electricity Market and is widely used for imbalanced classification. It has 45,312 instances dated from 7 May 1996 to 5 December 1998. Each instance consists of eight numerical features. The percentage of positive class instances is 42%.

KDDCup [11] stream task is to build a classifier able to distinguish intrusions from normal network connections. It has 494,021 instances with twenty-seven features, seven categorical and twenty numerical. The percentage of positive class instances is 20%.

5 Experimental Settings

To empirically compare the rebalancing techniques, we adopted the default configuration. The online bagging algorithms have the class size threshold set to 0.6 and the recall one to 0.4. We set the decaying factors of size and recall to 0.9. We use two windows having size 1000 to compute the IOOB and IUOB moving average recalls. CS has three parameters: we set the "k" of the k-nearest neighbors to 5, the class size threshold to 0.5, and the minimum size of the minority window to apply SMOTE to 100. RS has two parameters: the minimum and maximum size of the window. We set them to inactive. We tested ESOS-ELM using OS-ELMs and WELMs composed of 100 neurons, 1000 samples as batch size, and epsilon set to 0.001.

We tested all the data streams with the same algorithm configuration. Except for ESOS-ELM, we use the Hoeffding Adaptive Tree (HAT) [4] as a base learner able to deal with concept drift. All the ensemble algorithms use ten base learners. We use HAT and the standard online bagging OzaBag [26] with HAT as a base learner as baselines to compare the performance without any resampling.

To precisely evaluate the algorithms' performance during the whole stream, we tested the artificial data streams using a prequential evaluation with a fixed-length window evaluator. We set the window size to the stream's length before the concept drift: $50,000$ for abrupt drifts and $45,000$ in the other two cases.

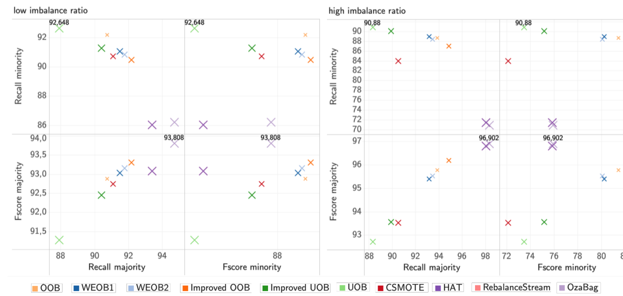


Fig. 2: Mean Recall and F_1-score with streams having a P(X|y) drift.

This allows restarting measuring when the drift happens without the influence of the statistics before the drift. Instead, we tested the real data streams using a prequential evaluation with a fading factor evaluator setting the fading to 0.995.

To compare the algorithms performances we considered measures that take the class imbalance problem into account, which are *Recall* and F_1*-score* of each class, *Gmean* [21], and K_{temp} [32] statistics. For what concern the resources consumed, time is measured in *CPU seconds*, while memory in *MB*.

IEbench runs the experiments using MOA [5], an open-source software library developed in Java able to handle Massive Online Analysis[3]. Notably, MOA executes in a Java Virtual Machine, making the various processes indistinguishable. IEbench tracks each experiment's memory consumption running it inside a docker container and collecting metrics using Telegraf and InfluxDB[4].

6 Results Discussion

This section presents all the results obtained analyzing, for the artificial data streams, one type of drift at a time and their resource requirements. We compare the results over the real data streams separately. Note that the chart legend presents the algorithm in decreasing order of the average Gmean obtained.

P(X|y) Concept Drift The artificial data streams with this kind of drift are only those generated with the SEA and SINE1 generators. We analyzed the performances of the algorithms w.r.t. a high imbalance ratio (*2:8* and *1:9*) and a low imbalance ratio (*4:6* and *3:7*).

[3] https://moa.cms.waikato.ac.nz/
[4] https://www.influxdata.com/

Figure 2 compares the Recall and F_1 results of each class. OOB and UOB were the best performing algorithms. UOB achieved the best minority class recall with both high and low imbalance ratios, significantly improving the performances of OzaBag in the first case. The drawback was that it lost a significant amount of majority class recall. OOB mitigated that by achieving a better minority class recall without losing too much the majority class one, thus achieving the best Gmean. An analysis of the F_1 measures allows noticing that, with both low and high imbalance ratios, the minority class F_1 of OOB was greater than the OzaBag's one while the UOB's one was lower. Indeed, OOB increased the minority class recall keeping a good precision, while UOB achieved the highest recall along with the worst precision. In both cases, the corresponding improved versions (IOOB and IUOB) increased the majority class recall losing on the minority class one. Only IUOB was able to improve the Gmean w.r.t. its native version UOB. Interestingly, both the minority and majority classes F_1 increased, leading to better general performances in multi-class settings.

WEOB1 and WEOB2 had similar performances: the former performed better on the minority class recall, while the latter performed better on the majority class one.

CS significantly increased the HAT performances on the minority class, losing only on the majority class recall, but overall performing better during all the data streams, as shown by the Gmean measure. As the imbalance ratio increased, this algorithm lost a lot of minority class precision w.r.t. its baseline. RS, on the other hand, was not able to detect any drift and its performances are the same as the HAT.

ESOS-ELM needed to inspect several instances to give accurate predictions. Hence, its position in Figure 2 was bad, but, after the drift, it achieved a minority/majority class recall of 86.5/95.3.

P(y) Concept Drift Each data stream with this kind of drift has different level of change. Data streams generated through the cluster generator present a drift due to the minority class appearing or disappearing, while the data streams generated through the SEA and SINE1 generators present a drift due to an imbalance ratio change.

Figure 3 shows that, when the data stream started without any sample belonging to a class, as in the *appearing minority* and the *minority share* streams, there was a significant difference of minority recall between the rebalanced algorithm and their baselines. In these cases, IUOB performed well, outperforming the other algorithms on the minority class recall after the drift. As already seen before, OOB and IOOB improved the minority class recall w.r.t. OzaBag without losing too much majority class recall.

WEOB1 and WEOB2 achieved the best performances having one of the best minority recalls just after the drift, close to the one of the IUOB, and keeping the majority class recall similar to the oversampling algorithms ones, thus achieving the best Gmean.

CS constantly improved the performances of the HAT on the minority class. It reached the minority class recall of the undersampling algorithms during the

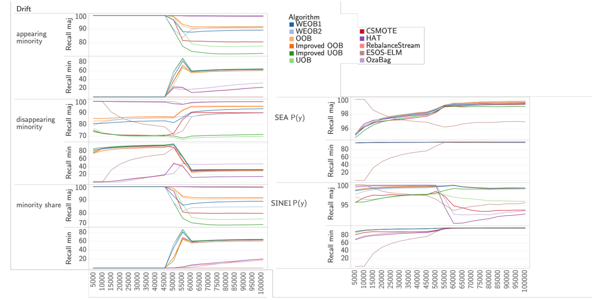


Fig. 3: Recall during streams having a P(y) drift with imbalance ratio *1:9*.

clusters drifts but with a better majority class recall. RS, on the other hand, did not seem to improve the HAT performances.

ESOS-ELM needed time to adjust its predictions. It was always the worst-performing. More importantly, if the stream started without the minority class, this algorithm could not identify any of its samples when they appear.

P(y|X) Concept Drift This kind of drift is the one that more affects the algorithms performances. Data streams with this type of drift are the corresponding SEA and SINE1 ones and all the ones with cluster movements and reshaping.

Figure 4 shows that, unlike the previous category of drifts, UOB had a better minority class recall than OOB only when the imbalance ratio was *1:9*. Again, OOB was the algorithm with the best Gmean having the highest minority class recall and the second higher majority class recall after IOOB. When the imbalance ratio was high, both the UOB minority class mean and final step recall were the best ones. However, during the drift, the best performing algorithm was OOB. Figure 5, which shows SINE1 stream results during such a severe drift, highlights this behavior thanks to the K_{temp}. Indeed, OzaBag's performance increased oversampling the minority class and decreased undersampling the majority class. Oversampling increased the importance of some of the new examples, making the HAT learners react to the drift faster. On the other hand, undersampling removed the majority-class samples, thus worsening the drift detection and the algorithm performances w.r.t not applying any resampling. Moreover, Figure 5 shows the importance of resampling also with drifts having low severity as SEA. The Gmean of the OzaBag dropped because the old concept boundary was difficult to forget if the wrongly classified samples were a few (minority class), and the majority class samples continued to be correctly classified.

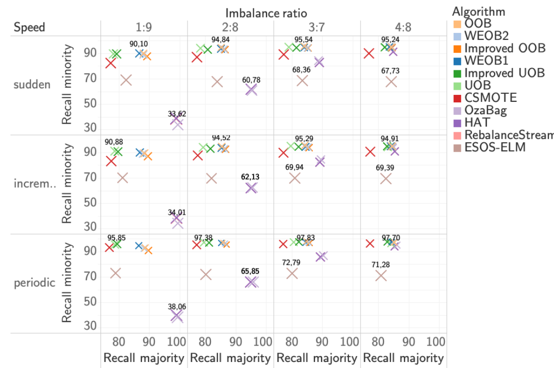


Fig. 4: Mean Recall with streams having a P(y|X) drift.

In Figure 4, IOOB and IUOB increased the majority class's recall losing a bit on the minority class one w.r.t. the corresponding native versions. Again, IUOB improved the UOB Gmean, but this did not happen with OOB.

WEOB1 and WEOB2 had similar performances. WEOB1 performed better on the minority class recall, while WEOB2 performed better on the majority class recall. Both classes' recalls were always near the best performing algorithm, achieving a Gmean better than both the baselines.

Figure 4 shows that CS significantly increased the HAT minority class performances only when the imbalance ratio was greater than *2:8*. Instead, Figure 5 shows a major problem of this algorithm in terms of K_{temp}. When the drift was severe, like in SINE1 streams, and the class concepts swapped, it cannot recover its accuracy, and its K_{temp} remained under the level of a no-change classifier. The ADWIN drift detector only looked at the seen samples' class label and did not detect P(y|X) drifts; thus, it generated synthetic samples with the old concept.

RS, on the other hand, was not able to detect any drift and its performances were the same as the HAT. As shown in Figure 5, ESOS-ELM needed to see several instances before having accurate predictions. When the imbalance ratio was *1:9*, it was the algorithm with the best minority class recall just after the drift. Still, it had a significant worsening of the recall of the majority class.

Real data streams Algorithm performances are different for each of the three real data streams tested, as shown by Figure 6.

The *Electricity* data stream has a constant imbalance ratio of *4:6*. In this case, the undersampling algorithms did not improve the minority class recall but only worst the majority class one. RS behaved exactly like a simple HAT,

Fig. 5: K_{temp} and Gmean during SEA and SINE1 streams having a P(y|X) drift and imbalance ratio *1:9*

which was better than what CS did, worsening both classes' recall. If we do not consider ESOS-ELM that predicted everything as belonging to the majority class, the best performing was OOB, which had the best Gmean, recall, and F_1 in both classes.

The *PAKDD* data stream results confirmed that, with high skewed imbalance ratios, UOB improved the minority class recall better than all the others, achieving a result of 83, 0. WEOB1, which has a good trade-off between the recall of both classes, achieved the best Gmean. CS significantly improved the performances of a single HAT which alone did not learn.

The *KDDCup* data stream, often used for anomaly detection evaluations, has high imbalance ratios that change frequently. ESOS-ELM behaved terribly, and it was not able to detect any minority class example. IOOB reached almost a perfect minority class recall with a mean value of 99.899, improving the recall of both classes w.r.t. OzaBag. UOB, on the other hand, worsened the recall of both classes, while IUOB achieved a good majority class recall. WEOB1, again, achieved the best Gmean.

Resource Requirements Each algorithm needed a different amount of resources. Table 1 presents the mean and variance of memory and time required by each algorithm during the experiments with the artificial data streams.

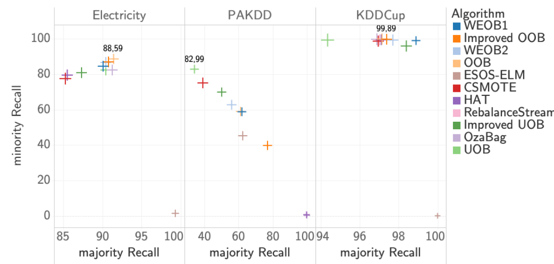


Fig. 6: Recall average with real data streams.

Table 1: Time and Memory requirements of each algorithm. The best ones are displayed in bold while the worst ones are displayed in italics.

	UOB	OOB	IUOB	IOOB	WEOB1	WEOB2	CS	RS	ESOS-ELM
Memory	374 ± 91	513 ± 74	**313 ± 81**	472 ± 89	545 ± 82	545 ± 82	1116 ± 265	2606 ± 639	*6014 ± 853*
Time	6 ± 6	10 ± 6	**5 ± 6**	9 ± 7	13 ± 6	15 ± 8	*315 ± 185*	23 ± 11	288 ± 32

The results clearly show that the online bagging algorithms outperformed the others from the resources point of view. One possible reason is that they save only the time-decaying variables, avoiding keeping any sample or label. WEOB1 and WEOB2 required the most significant resources among the online bagging algorithms since they need the windows. The windows size is not proportional to the seen samples, thus never becoming too big. ESOS-ELM and CS were the worst-performing from a time point of view. The slow part of CS is the K-NN search when applying the online SMOTE version: the time to generate a new sample is proportional to the window size; this makes the algorithm slow when the window size grows. RS has the worst memory consumption after the ESOS-ELM because it keeps a batch with all the seen samples since the last drift. ESOS-ELM requires completely different mathematical operations than HAT; thus, it is not comparable.

7 Conclusions

Addressing the class imbalance is crucial to building models able to solve real-world problems, particularly in presence of concept drifts. We developed IEBench, a benchmark that effectively compares SML algorithms performance on data streams. IEBench allowed us to conduct a comprehensive experimental campaign on imbalanced and evolving data streams. We implemented and tested nine state-of-art algorithms over a wide range of artificial and real data streams

measuring all the relevant statistics. We aggregated the results by concept drifts types, concept drifts speeds, and imbalance ratios to identify each algorithm's weaknesses and strengths.

The experimental campaign's outcomes showed that applying rebalancing techniques has improved the algorithm performances during all the concept drift types (RQ1). In particular, they provided a clear view of where to focus for each algorithm future improvements (RQ2). The online bagging-based algorithms have proven to be consistently the best in terms of performances and resources (RQ3), each in a different situation, opening the way to new ensemble ideas. CS has shown difficulties with the P(y|X) drifts alongside excessive time requirements (RQ3); future improvement should consider these results. RS needs a different drift detection technique to improve the performances of the base learner. Finally, ESOS-ELM is different from the others, and it has a high number of hyper-parameters to tune. This algorithm requires more studies and experiments.

As future work, we intend to use IEBench to explore higher imbalance ratios or more severe drifts on the data streams to identify the algorithms suited for more challenging scenarios. It is easy to integrate new algorithms and data streams, e.g., enlarging the benchmark to multi-class settings. Moreover, we plan to improve IEBench to run the experiments in parallel and extend it to automate the hyper-parameter tuning process.

References

1. Bernardo, A., Della Valle, E., Bifet, A.: Incremental rebalancing learning on evolving data streams. In: 2020 International Conference on Data Mining Workshops (ICDMW). pp. 844–850 (2020)
2. Bernardo, A., Gomes, H.M., Montiel, J., Pfahringer, B., Bifet, A., Della Valle, E.: C-smote: Continuous synthetic minority oversampling for evolving data streams. In: 2020 IEEE International Conference on Big Data (Big Data). pp. 483–492 (2020). https://doi.org/10.1109/BigData50022.2020.9377768
3. Bifet, A., Gavaldà, R.: Learning from time-changing data with adaptive windowing. In: SDM. pp. 443–448. SIAM (2007)
4. Bifet, A., Gavaldà, R.: Adaptive learning from evolving data streams. In: IDA. LNCS, vol. 5772, pp. 249–260. Springer (2009)
5. Bifet, A., Holmes, G., Kirkby, R., Pfahringer, B.: MOA: massive online analysis. J. Mach. Learn. Res. **11**, 1601–1604 (2010)
6. Breiman, L.: Bagging predictors. Mach. Learn. **24**(2), 123–140 (1996)
7. Brzezinski, D., Minku, L., Pewinski, T., Stefanowski, J., Szumaczuk, A.: The impact of data difficulty factors on classification of imbalanced and concept drifting data streams. Knowl. Inf. Syst. (2021), in press
8. Chan, C., Yu, E.W.M.: An abnormal sound detection and classification system for surveillance applications. In: EUSIPCO. pp. 1851–1855. IEEE (2010)
9. Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: SMOTE: synthetic minority over-sampling technique. J. Artif. Intell. Res. **16**, 321–357 (2002)
10. Cohen, J.: A coefficient of agreement for nominal scales. Educational and Psychological Measurement **20**(1), 37–46 (1960)

11. Dua, D., Graff, C.: UCI machine learning repository (2017), http://archive.ics.uci.edu/ml
12. Elwell, R., Polikar, R.: Incremental learning of concept drift in nonstationary environments. IEEE Trans. Neural Networks **22**(10), 1517–1531 (2011)
13. Gama, J., Medas, P., Castillo, G., Rodrigues, P.P.: Learning with drift detection. In: SBIA. LNCS, vol. 3171, pp. 286–295. Springer (2004)
14. Hassan, F., Shaheen, M., Sahal, R.: Real-time healthcare monitoring system using online machine learning and spark streaming. International Journal of Advanced Computer Science and Applications **11** (09 2020). https://doi.org/10.14569/IJACSA.2020.0110977
15. He, H., Garcia, E.A.: Learning from imbalanced data. IEEE Trans. Knowl. Data Eng. **21**(9), 1263–1284 (2009)
16. Huang, G., Siew, C.K.: Extreme learning machine: RBF network case. In: ICARCV. pp. 1029–1036. IEEE (2004)
17. Hussain, I., Park, S.J.: Healthsos: Real-time health monitoring system for stroke prognostics. IEEE Access **8**, 213574–213586 (2020)
18. Kammerer, K., Hoppenstedt, B., Pryss, R., Stökler, S., Allgaier, J., Reichert, M.: Anomaly detections for manufacturing systems based on sensor data - insights into two challenging real-world production settings. Sensors **19**(24), 5370 (2019)
19. Kolter, J.Z., Maloof, M.A.: Dynamic weighted majority: A new ensemble method for tracking concept drift. In: ICDM. pp. 123–130. IEEE Computer Society (2003)
20. Krawczyk, B., Minku, L.L., Gama, J., Stefanowski, J., Wozniak, M.: Ensemble learning for data stream analysis: A survey. Inf. Fusion **37**, 132–156 (2017)
21. Kubat, M., Holte, R.C., Matwin, S.: Machine learning for the detection of oil spills in satellite radar images. Mach. Learn. **30**(2-3), 195–215 (1998)
22. Lee, K., Agrawal, A., Choudhary, A.N.: Real-time disease surveillance using twitter data: demonstration on flu and cancer. In: KDD. pp. 1474–1477. ACM (2013)
23. Liang, N., Huang, G., Saratchandran, P., Sundararajan, N.: A fast and accurate online sequential learning algorithm for feedforward networks. IEEE Trans. Neural Networks **17**(6), 1411–1423 (2006)
24. Mirza, B., Lin, Z., Liu, N.: Ensemble of subset online sequential extreme learning machine for class imbalance and concept drift. Neurocomputing **149**, 316–329 (2015)
25. Moreira, J.: Travel Time Prediction for the Planning of Mass Transit Companies: a Machine Learning Approach. Ph.D. thesis, . (12 2008)
26. Oza, N.C., Russell, S.J.: Online bagging and boosting. In: AISTATS. Society for Artificial Intelligence and Statistics (2001)
27. Street, W.N., Kim, Y.: A streaming ensemble algorithm (SEA) for large-scale classification. In: KDD. pp. 377–382. ACM (2001)
28. Theeramunkong, T., Kijsirikul, B., Cercone, N., Ho, T.B. (eds.): PAKDD 2009, Bangkok, Thailand, April 27-30, Proceedings, LNCS, vol. 5476. Springer (2009)
29. Tsymbal, A.: The problem of concept drift: definitions and related work. Computer Science Department, Trinity College Dublin **106**(2), 58 (2004)
30. Wang, S., Minku, L.L., Yao, X.: A learning framework for online class imbalance learning. In: CIEL. pp. 36–45. IEEE (2013)
31. Wang, S., Minku, L.L., Yao, X.: Resampling-based ensemble methods for online class imbalance learning. IEEE Trans. Knowl. Data Eng. **27**(5), 1356–1368 (2015)
32. Zliobaite, I., Bifet, A., Read, J., Pfahringer, B., Holmes, G.: Evaluation methods and decision theory for classification of streaming data with temporal dependence. Mach. Learn. **98**(3), 455–482 (2015)

Bibliography

[1] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, João Gama, and Guangquan Zhang. Learning under concept drift: A review. *IEEE Trans. Knowl. Data Eng.*, 31(12):2346–2363, 2019.

[2] Hong Li, Lin Li, Zi Zhong, Yi Han, LiHong Hu, and Ying Lu. An accurate and efficient method to predict y-no bond homolysis bond dissociation energies. *Mathematical Problems in Engineering*, 2013, 08 2013.

[3] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. MOA: massive online analysis. *J. Mach. Learn. Res.*, 11:1601–1604, 2010.

[4] Scott Wares, John Isaacs, and Eyad Elyan. Data stream mining: methods and challenges for handling concept drift. *SN Applied Sciences*, 1, 10 2019.

[5] Doug Laney et al. 3d data management: Controlling data volume, velocity and variety. *META group research note*, 6(70):1, 2001.

[6] M. Khan, Muhammad Uddin, and Navarun Gupta. Seven v's of big data understanding big data to extract value. pages 1–5, 04 2014.

[7] Hootsuite & We Are Social. Digital 2021 global digital overview. *https://datareportal.com/reports/digital-2021-global-digital-overview*, 2021.

[8] Alexey Tsymbal. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin*, 106(2):58, 2004.

[9] Haibo He and Edwardo A. Garcia. Learning from imbalanced data. *IEEE Trans. Knowl. Data Eng.*, 21(9):1263–1284, 2009.

[10] Klaus Kammerer, Burkhard Hoppenstedt, Rüdiger Pryss, Steffen Stökler, Johannes Allgaier, and Manfred Reichert. Anomaly detections for manufacturing systems based on sensor data - insights into two challenging real-world production settings. *Sensors*, 19(24):5370, 2019.

[11] Cheung-Fat Chan and Eric W. M. Yu. An abnormal sound detection and classification system for surveillance applications. In *EUSIPCO*, pages 1851–1855. IEEE, 2010.

[12] João Moreira. *Travel Time Prediction for the Planning of Mass Transit Companies: a Machine Learning Approach.* PhD thesis, 12 2008.

[13] Shanle Ma, Xue Li, Yi Ding, and Maria E. Orlowska. A recommender system with interest-drifting. In *WISE*, volume 4831 of *Lecture Notes in Computer Science*, pages 633–642. Springer, 2007.

[14] Iqram Hussain and Se Jin Park. Healthsos: Real-time health monitoring system for stroke prognostics. *IEEE Access*, 8:213574–213586, 2020.

[15] Tommaso Alberti and Davide Faranda. On the uncertainty of real-time predictions of epidemic growths: A COVID-19 case study for china and italy. *Commun. Nonlinear Sci. Numer. Simul.*, 90:105372, 2020.

[16] Dariusz Brzezinski, Leandro Minku, Tomasz Pewinski, Jerzy Stefanowski, and Artur Szumaczuk. The impact of data difficulty factors on classification of imbalanced and concept drifting data streams. *Knowl. Inf. Syst.*, 2021. In press.

[17] Mowei Wang, Yong Cui, Xin Wang, Shihan Xiao, and Junchen Jiang. Machine learning for networking: Workflow, advances and opportunities. *IEEE Netw.*, 32(2):92–99, 2018.

[18] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI*, pages 1137–1145. Morgan Kaufmann, 1995.

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

[20] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[21] C. Gini and G. Ottaviani. *Memorie di metodologia statistica*. Number v. 1 in Memorie di metodologia statistica. E.V. Veschi, 1955.

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

[23] Imen Khamassi, M. Sayed Mouchaweh, Moez Hammami, and Khaled Ghédira. *A New Combination of Diversity Techniques in Ensemble Classifiers for Handling Complex Concept Drift: Methods and Applications*, pages 39–61. 01 2019.

[24] Albert Bifet. Classifier concept drift detection and the illusion of progress. In *ICAISC (2)*, volume 10246 of *Lecture Notes in Computer Science*, pages 715–725. Springer, 2017.

[25] E. S. Page. Continuos Inspection Schemes. *Biometrika*, 41(1-2):100–115, 06 1954.

[26] João Gama, Pedro Medas, Gladys Castillo, and Pedro Pereira Rodrigues. Learning with drift detection. In *SBIA*, volume 3171 of *Lecture Notes in Computer Science*, pages 286–295. Springer, 2004.

[27] Heng Wang and Zubin Abraham. Concept drift detection for streaming data. In *IJCNN*, pages 1–9. IEEE, 2015.

[28] Jeremy Z. Kolter and Marcus A. Maloof. Dynamic weighted majority: A new ensemble method for tracking concept drift. In *ICDM*, pages 123–130. IEEE Computer Society, 2003.

[29] Albert Bifet and Ricard Gavaldà. Learning from time-changing data with adaptive windowing. In *SDM*, pages 443–448. SIAM, 2007.

[30] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6):1794–1813, 2002.

[31] Albert Bifet and Ricard Gavaldà. Adaptive learning from evolving data streams. In *IDA*, volume 5772 of *Lecture Notes in Computer Science*, pages 249–260. Springer, 2009.

[32] Trevor Hastie and Robert Tibshirani. Classification by pairwise coupling. In *NIPS*, pages 507–513. The MIT Press, 1997.

[33] Bikash Joshi, Massih-Reza Amini, Ioannis Partalas, Liva Ralaivola, Nicolas Usunier, and Éric Gaussier. On binary reduction of large-scale multiclass classification problems. In *IDA*, volume 9385 of *Lecture Notes in Computer Science*, pages 132–144. Springer, 2015.

[34] Thomas G. Dietterich and Ghulum Bakiri. Solving multiclass learning problems via error-correcting output codes. *J. Artif. Intell. Res.*, 2:263–286, 1995.

[35] Krystyna Napierala and Jerzy Stefanowski. Identification of different types of minority class examples in imbalanced data. In *HAIS (2)*, volume 7209 of *Lecture Notes in Computer Science*, pages 139–150. Springer, 2012.

[36] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. SMOTE: synthetic minority over-sampling technique. *J. Artif. Intell. Res.*, 16:321–357, 2002.

[37] Miroslav Kubat and Stan Matwin. Addressing the curse of imbalanced training sets: One-sided selection. In *ICML*, pages 179–186. Morgan Kaufmann, 1997.

[38] Leo Breiman. Bagging predictors. *Mach. Learn.*, 24(2):123–140, 1996.

[39] Robert E. Schapire. The strength of weak learnability. *Mach. Learn.*, 5:197–227, 1990.

[40] Albert Bifet, Ricard Gavaldà, Geoff Holmes, and Bernhard Pfahringer. *Machine Learning for Data Streams with Practical Examples in MOA*. MIT Press, 2018.

[41] Gary M. Weiss. Mining with rarity: a unifying framework. *SIGKDD Explor.*, 6(1):7–19, 2004.

[42] Miroslav Kubat, Robert C. Holte, and Stan Matwin. Machine learning for the detection of oil spills in satellite radar images. *Mach. Learn.*, 30(2-3):195–215, 1998.

[43] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[44] Albert Bifet, Gianmarco De Francisci Morales, Jesse Read, Geoff Holmes, and Bernhard Pfahringer. Efficient online evaluation of big data stream classifiers. In *KDD*, pages 59–68. ACM, 2015.

[45] Indre Zliobaite, Albert Bifet, Jesse Read, Bernhard Pfahringer, and Geoff Holmes. Evaluation methods and decision theory for classification of streaming data with temporal dependence. *Mach. Learn.*, 98(3):455–482, 2015.

[46] Nikunj C. Oza and Stuart J. Russell. Online bagging and boosting. In *AISTATS*. Society for Artificial Intelligence and Statistics, 2001.

[47] A. Bernardo, H. M. Gomes, J. Montiel, B. Pfahringer, A. Bifet, and E. Della Valle. C-smote: Continuous synthetic minority oversampling for evolving data streams. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 483–492, 2020.

[48] A. Bernardo, E. della Valle, and A. Bifet. Incremental rebalancing learning on evolving data streams. In *2020 International Conference on Data Mining Workshops (ICDMW)*, pages 844–850, 2020.

[49] Shuo Wang, Leandro L. Minku, and Xin Yao. A learning framework for online class imbalance learning. In *CIEL*, pages 36–45. IEEE, 2013.

[50] Shuo Wang, Leandro L. Minku, and Xin Yao. Resampling-based ensemble methods for online class imbalance learning. *IEEE Trans. Knowl. Data Eng.*, 27(5):1356–1368, 2015.

[51] Bilal Mirza, Zhiping Lin, and Nan Liu. Ensemble of subset online sequential extreme learning machine for class imbalance and concept drift. *Neurocomputing*, 149:316–329, 2015.

[52] Nan-Ying Liang, Guang-Bin Huang, Paramasivan Saratchandran, and Narasimhan Sundararajan. A fast and accurate online sequential learning algorithm for feedforward networks. *IEEE Trans. Neural Networks*, 17(6):1411–1423, 2006.

[53] Bilal Mirza, Zhiping Lin, and Kar-Ann Toh. Weighted online sequential extreme learning machine for class imbalance learning. *Neural Process. Lett.*, 38(3):465–486, 2013.

[54] Guang-Bin Huang and Chee Kheong Siew. Extreme learning machine: RBF network case. In *ICARCV*, pages 1029–1036. IEEE, 2004.

[55] Wei Dai and Daniel Berleant. Benchmarking contemporary deep learning hardware and frameworks: A survey of qualitative metrics. In *CogMI*, pages 148–155. IEEE, 2019.

[56] The Prognostics and Health Management Society (PHM Society). Phm challenge competition data set. In *[Online]: http://www.phmsociety.org/references/datasets*, 2009.

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

[58] Thanaruk Theeramunkong, Boonserm Kijsirikul, Nick Cercone, and Tu Bao Ho, editors. *Advances in Knowledge Discovery and Data Mining, 13th Pacific-Asia Conference, PAKDD 2009, Bangkok, Thailand, April 27-30, 2009, Proceedings*, volume 5476 of *Lecture Notes in Computer Science*. Springer, 2009.

[59] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. http://archive.ics.uci.edu/ml.