Chapter 1

Streaming Data Mining with Massive Online Analytics (MOA)

Albert Bifet

LTCI, Télécom ParisTech
Université Paris-Saclay, France
albert.bifet@telecom-paristech.fr

Jesse Read

LIX, École Polytechnique
Université Paris-Saclay, France
jesse.read@polytechnique.edu

Geoff Holmes* and Bernhard Pfahringer†

Computer Science Department
University of Waikato, New Zealand
**geoff@waikato.ac.nz*
†*bernhard@waikato.ac.nz*

Fast Big Data is being produced at high-velocity in real-time. To effectively deal with this type of streaming data produced in real time, we need to be able to adapt to changes on the distribution of the data being produced, and we need to do it using the minimum amount of time and memory. The Internet of Things (IoT) is a good example and motivation of this type of streaming data produced in real time.

Massive **O**nline **A**nalytics (MOA) is a software environment for implementing algorithms and running experiments for online learning from evolving data streams. MOA is designed to deal with the challenging problem of scaling up the implementation of state of the art algorithms to real world dataset sizes. MOA includes classification and clustering methods. It contains collection of offline and online methods as well as tools for evaluation. MOA supports bi-directional interaction with WEKA, the Waikato Environment for Knowledge Analysis, and is released under the GNU GPL license.

1. Introduction

Nowadays, data is generated at an increasing rate from sensor applications, measurements in network monitoring and traffic management, log records or click-streams in web exploring, manufacturing processes, call detail records, email, blogging, twitter posts and others. In fact, all data generated can be considered as streaming data or as a snapshot of streaming data, since it is obtained from an interval of time.

In the data stream model, data arrive at high speed, and an algorithm must process them under very strict constraints of space and time. MOA is an open-source framework for dealing with massive, potentially infinite, evolving data streams.

A data stream environment has different requirements from the traditional batch learning setting. The most significant are the following:

Requirement 1 Process an example at a time, and inspect it only once (at most)
Requirement 2 Use a limited amount of memory
Requirement 3 Work in a limited amount of time
Requirement 4 Be ready to predict at any time.

Figure 1 illustrates the typical use of a data stream classification algorithm, and how the requirements fit in a repeating cycle:

(1) The algorithm is passed the next available example from the stream (Requirement 1).
(2) The algorithm processes the example, updating its data structures. It does so without exceeding the memory bounds set on it (requirement 2), and as quickly as possible (Requirement 3).
(3) The algorithm is ready to accept the next example. On request it is able to predict the class of unseen examples (Requirement 4).

As data stream mining is a relatively new field, evaluation practices are not nearly as well researched and established as they are in the traditional batch setting. The majority of experimental evaluations use less than one million training examples. In the context of data streams this is disappointing, because to be truly useful at data stream classification the algorithms need to be capable of handling very large (potentially infinite) streams of examples. Demonstrating systems only on small amounts of data does not build a convincing case for capacity to solve more demanding data stream applications [1].

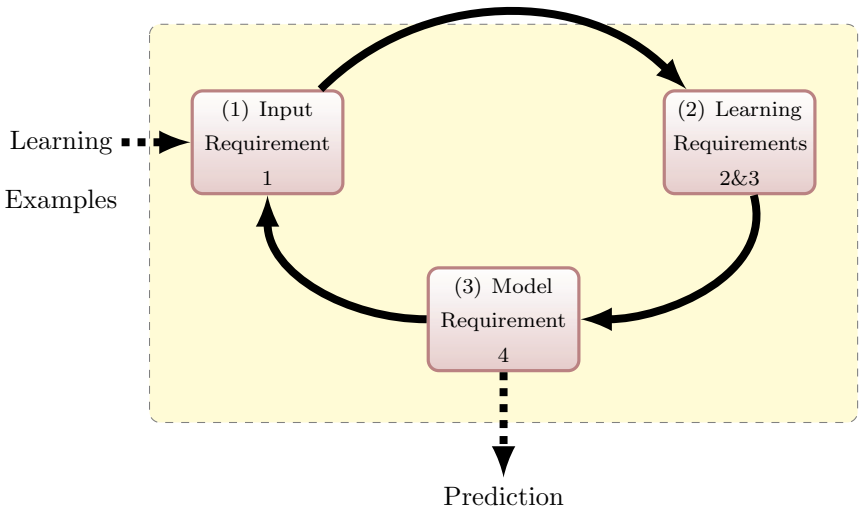

Fig. 1. The data stream classification cycle.

MOA permits evaluation of data stream learning algorithms on large streams, in the order of tens of millions of examples where possible, and under explicit memory limits. Any less than this does not actually test algorithms in a realistically challenging setting.

Other Machine Learning frameworks as Weka, RapidMiner or scikit-learn were not designed for data stream mining. In those frameworks, data is considered static, and can be stored in main memory. MOA was designed specifically for the data stream setting, with the challenging capabilities of adapting to changes and processing data without storing it.

2. Concept Drift

Dealing with data streams evolving over time, our models need to adapt to changes on the data. To do that, we need to know when it is the best moment to adapt them. This is why it is important to detect changes, in a fast and accurate way, so that we can update or replace our current models when it is more appropriate.

A *change detector* or *drift detector* is an algorithm that having as input a stream of instances, outputs an alarm if it detects change on the distribution of the data, and optionally a prediction of the next instance to come. In general, the input to this algorithm is a sequence $x_1, x_2, \ldots, x_t, \ldots$ of data

items whose distribution varies over time in an unknown way. The outputs of the algorithm are, at each time step, an estimation of some important parameters of the input distribution, and a signal alarm indicating that distribution change has recently occurred.

There are many different algorithms to detect change in streams. We start looking at the classical ones used in statistical quality control, and then we look at more recent ones such as ADWIN.

2.1. *The CUSUM Test*

The cumulative sum (CUSUM algorithm), which was first proposed in 1954 [2], is a change detection algorithm that gives an alarm when the mean of the input data is significantly different from zero. The CUSUM input ϵ_t can be any filter residual, for instance the prediction error from a Kalman filter.

The CUSUM test is as follows:

$$g_0 = 0$$

$$g_t = \max\,(0, g_{t-1} + \epsilon_t - \upsilon)$$

$$\text{if } g_t > h \text{ then alarm and } g_t = 0$$

The CUSUM test is memoryless, and its accuracy depends on the choice of parameters υ and h.

2.2. *The Page Hinckley Test*

The CUSUM test is a stopping rule. Other stopping rules exist. For example, the Page Hinckley test, also presented in [2]. The Page Hinckley Test is as follows, when the signal is increasing:

$$g_0 = 0, \qquad g_t = g_{t-1} + (\epsilon_t - \upsilon)$$

$$G_t = \min(g_t)$$

$$\text{if } g_t - G_t > h \text{ then alarm and } g_t = 0$$

In the case that the signal is decreasing, instead of $G_t = \min(g_t)$, we should use $G_t = \max(g_t)$ and $G_t - g_t > h$ as the stopping rule. As the CUSUM test, the Page Hinckley test is memoryless, and its accuracy depends on the choice of parameters υ and h.

2.3. *Drift Detection Method*

The drift detection method (DDM) proposed by Gama *et al.* [3] controls the number of errors produced by the learning model during prediction. It compares the statistics of two windows: the first contains all the data, and the second contains only the data from the beginning until the number of errors increases. Their method doesn't store these windows in memory. It keeps only statistics and a window of recent errors.

The number of errors in a sample of n examples is modelled by a binomial distribution. For each point t in the sequence that is being sampled, the error rate is the probability of misclassifying (p_t), with standard deviation given by $s_t = \sqrt{p_t(1-p_t)/t}$. They assume that the error rate of the learning algorithm (p_t) will decrease while the number of examples increases if the distribution of the examples is stationary. A significant increase in the error of the algorithm, suggests that the class distribution is changing and, hence, the actual decision model is considered to be inappropriate. Thus, they store the values of p_t and s_t when $p_t + s_t$ reaches its minimum value during the process (obtaining p_{pmin} and s_{min}). And it checks when the following conditions trigger:

- $p_t + s_t \geq p_{min} + 2 \cdot s_{min}$ for the warning level. Beyond this level, the examples are stored in anticipation of a possible change of context.
- $p_t + s_t \geq p_{min} + 3 \cdot s_{min}$ for the drift level. Beyond this level the concept drift is considered to be real, the model induced by the learning method is reset and a new model is learnt using the examples stored since the warning level triggered. The values for p_{min} and s_{min} are reset too.

This approach demonstrates good behavior detecting abrupt changes and gradual changes when the gradual change is not very slow, but it has difficulties when the change is slowly gradual. In that case, the examples will be stored for a long time, the drift level can take too much time to trigger and the examples in memory can be excessive.

2.4. *ADWIN*

`ADWIN` (`AD`aptive sliding `WIN`dow) [4] is a change detector and estimation algorithm. It solves, in a well-specified way, the problem of tracking the average of a stream of bits or real-valued numbers. `ADWIN` keeps a variable-length window of recently seen items, with the property that the window has the maximal length statistically consistent with the hypothesis "there has been no change in the average value inside the window".

More precisely, an older fragment of the window is dropped if and only if there is enough evidence that its average value differs from that of the rest of the window. This has two consequences: one, change is reliably detected whenever the window shrinks; and two, at any time the average over the existing window can be used as a reliable estimate of the current average in the stream (barring a very small or recent change that is not yet statistically significant).

The inputs to `ADWIN` are a confidence value $\delta \in (0, 1)$ and a (possibly infinite) sequence of real values $x_1, x_2, x_3, \ldots, x_t, \ldots$ The value of x_t is available only at time t. Each x_t is generated according to some distribution D_t, independently for every t. We denote with μ_t the expected value of x_t when it is drawn according to D_t. We assume that x_t is always in $[0, 1]$; rescaling deals with cases where $a \leq x_t \leq b$. No further assumption is being made about the distribution D_t; in particular, μ_t is unknown for all t.

`ADWIN` is parameter- and assumption-free in the sense that it automatically detects and adapts to the current rate of change. Its only parameter is a confidence bound δ, indicating how confident we want to be in the algorithm's output, inherent to all algorithms dealing with random processes.

It is important to note that `ADWIN` does not maintain the window explicitly, but compresses it using a variant of the exponential histogram technique [5]. This means that it keeps a window of length W using only $O(\log W)$ memory and $O(\log W)$ processing time per item, rather than the $O(W)$ one expects from a naïve implementation.

3. Classification

Classification is one of the most widely used data mining techniques. In very general terms, given a list of groups (often called classes), classification seeks to predict to which group a new instance may belong. The outcome of classification is typically either the identification of a single group or the production of a probability distribution of likelihood of membership of each group. A spam filter is a good example, where we want to predict if new emails are considered spam or not. Twitter sentiment analysis is another example, where we want to predict if the sentiment of a new incoming tweet is positive or negative.

More formally, the classification problem can be formulated as follows: given a set of instances of the form (x, y), where $x = x_1, \ldots, x_k$ is a vector of attribute values, and y is a discrete class from a set of n_C different

classes, the classifier builds a model $y = f(x)$ to predict the classes y of future examples. For example, x could be a tweet and y the polarity of its sentiment; or x could be an email message, and y the decision whether it is spam or not.

Evaluation is one of the most fundamental tasks in the stream data mining process, since it helps to decide what techniques are more appropriate to use for a specific data stream mining problem. The main challenge is to know when a method is outperforming another method only by chance, or if there is a statistical significance to that claim. Some of the methodologies applied are the same as in the case of non-dynamic data, where all data can be stored in memory. However, mining evolving data streams has new challenges and uses new evaluation methodologies. One thing worth noting before we continue is that almost all of the discoveries made in data mining and particularly classification assume that data is IID (Independent, Identically, Distributed). Thus a stationary distribution is randomly producing data, in no particular order and the underlying distribution generating the data is not changing. In a dynamic-data environment no part of IID remains valid. It is often the case, for example, that for certain time-periods the labels or classes of instances are correlated, intrusion detection has a majority of periods containing instance class labels designated no-intrusion and then shorter much less frequent periods of intrusion. This is another aspect of data stream mining that would benefit from further research.

For evolving data streams, the main difference with traditional data mining evaluation, is in how to perform the error estimation. Resources are limited and cross-validation may be too expensive.

The evaluation procedure of a learning algorithm determines which examples are used for training the algorithm, and which are used for testing the model output by the algorithm.

In traditional batch learning the problem of limited data is overcome by analyzing and averaging multiple models produced with different random arrangements of training and test data. In the stream setting the problem of (effectively) unlimited data poses different challenges.

When considering what procedure to use in the non-distributed data stream setting, one of the unique concerns is how to build a picture of accuracy over time. Two main approaches arise:

- **Holdout**: when data is so abundant, that it is possible to have test sets periodically, then we can measure the performance on these holdout sets. There is a training data stream that is used to train the learner

continuously, and small test data sets that are used to compute the performance periodically.

- **Interleaved Test-Then-Train or Prequential**: when data is not abundant, and there are no test sets, then each individual example can be used to test the model *before* it is used for training, and from this the accuracy can be incrementally updated. The model is always being tested on examples it has not seen.

Holdout evaluation gives a more accurate estimation of the accuracy of the classifier on more recent data. However, it requires recent test data that it is difficult to obtain for real datasets. There is also the issue of ensuring coverage of important change events, if the holdout is during a less volatile period of change then it might give an over-estimate of classifier performance. Gama *et al.* [6] propose to use a forgetting mechanism for estimating holdout accuracy using prequential accuracy: a sliding window of size w with the most recent observations, or fading factors that weigh observations using a decay factor α. The output of the two mechanisms is very similar (every window of size w_0 may be approximated by some decay factor α_0).

In a distributed data stream setting, we have classifiers that can be trained at the same time. The approaches in this setting are the following [7]:

- **k-fold distributed split-validation**: when there is abundance of data and k classifiers. Each time a new instance arrive, it is decided with probability $1/k$ if it will be used for testing. If it is used for testing, it is used by all the classifiers. If not, then it is used for training and assigned to only one classifier. Doing that, each classifier sees different instances, and they are tested using the same data.
- **5×2 distributed cross-validation**: when data is less abundant, and we want to use only 10 classifiers. We have 5 groups of 2 classifiers, and for each group, each time a new instance arrive, it is decided with probability 1/2 which of the two classifiers is used to test; the other classifier of the group is used to train. All instances are used to test or to train, and there is no overlapping between test instances and train instances.
- **k-fold distributed cross-validation**: when data is scarce and we have k classifiers. Each time a new instance arrive, it is used for testing in one classifier selected randomly, and trained using the others. This is the equivalent evaluation to k-fold distributed cross-validation.

The *decision tree* is a very popular data mining technique since it is very easy to interpret and visualize the model it builds. It consists of a tree structure, where each internal node corresponds to an attribute that splits into a branch for each attribute value, and leaves correspond to classification predictors, usually majority class classifiers. Figure 2 shows an example.

Contains "Money"	**Domain type**	**Has attach.**	**Time received**	**spam**
yes	com	yes	night	yes
yes	edu	no	night	yes
no	com	yes	night	yes
no	edu	no	day	no
no	com	no	day	no
yes	cat	no	day	yes

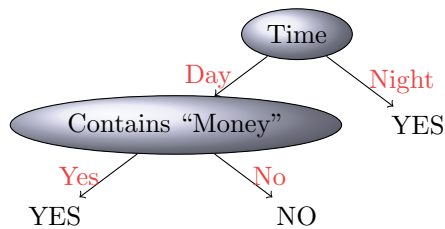

Fig. 2. A data set that describes e-mail features for deciding if it is spam, and a decision tree build using this data set.

Decision tree accuracy performance can be improved, using other classifiers at the leaves, such as Naive Bayes, or using ensembles of classifiers, as we will see later on.

The basic way to build a tree is the following, starting by creating a root node at the beginning $node = root$, and then doing the following:

(1) Assign A as the best decision attribute for $node$.
(2) For each value of A, create new descendant of $node$.
(3) Sort training instances to leaf nodes.
(4) If training instances are perfectly classified, then STOP, else iterate over new leaf nodes.

Two common measures are used to select the best decision attribute:

- Information Gain: computed as the decrement in entropy

$$\text{Information Gain} = \text{Entropy(before Split)} - \text{Entropy(after split)}$$

  where Entropy is a measure of the uncertainty associated with a random variable defined as $\text{Entropy} = -\sum^{c} p_i \cdot \log p_i$.
- Gini impurity Gain: computed using the Gini impurity measure instead of the entropy

$$\text{Gini Index} = \sum^{c} p_i(1-p_i) = 1 - \sum^{c} p_i^2$$

  The Gini index is a measure of the statistical dispersion associated with a random variable.

3.1. *The Hoeffding Tree*

In the data stream setting, where we can not store all the data, the main problem of building a decision tree is the need of reusing the examples to compute the best splitting attributes. Hulten and Domingos [8] proposed the Hoeffding Tree or VFDT, a very fast decision tree for streaming data, where instead of reusing instances, we wait for new instances to arrive. The most interesting feature of the Hoeffding tree is that it builds an identical tree with a traditional one, with high probability if the number of instances is large enough, and that it has theoretical guarantees about that.

The pseudo-code of VFDT is shown in Figure 3. The Hoeffding Tree is based on the Hoeffding bound. This inequality or bound justifies that a small sample can often be enough to choose an optimal splitting attribute. Suppose we make n independent observations of a random variable r with range R, where r is an attribute selection measure such as information gain or Gini impurity gain. The Hoeffding inequality states that with probability $1-\delta$, if the true mean r of r is at least $E[r]-\epsilon$, then

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

Using this fact, the Hoeffding tree algorithm, can determine, with high probability the smallest number n of examples needed at a node when selecting a splitting attribute.

The Hoeffding Tree maintains in each node the statistics needed for splitting attributes. For discrete attributes, this is the same information as needed for computing the Naive Bayes predictions: a 3-dimensional table

```
HoeffdingTree(Stream, δ)
1  ▷ Let HT be a tree with a single leaf(root)
2  ▷ Init counts n_ijk at root
3  for each example (x, y) in Stream
4        do HTGrow((x, y), HT, δ)

HTGrow((x, y), HT, δ)
1  ▷ Sort (x, y) to leaf l using HT
2  ▷ Update counts n_ijk at leaf l
3  if examples seen so far at l are not all of the same class
4     then
5          ▷ Compute G for each attribute
6          if G(Best Attr.)−G(2nd best) > sqrt(R² ln 1/δ / 2n)
7             then
8                  ▷ Split leaf on best attribute
9                  for each branch
10                      do ▷ Start new leaf and initialize counts
```

Fig. 3. The Hoeffding Tree algorithm.

that stores for each triple (x_i, v_j, c) a count $n_{i,j,c}$ of training instances with $x_i = v_j$, together with a 1-dimensional table for the counts of $C = c$. The memory needed depends on the number of leaves of the tree, but not on the size of the data stream.

A theoretically appealing feature of Hoeffding Trees not shared by other incremental decision tree learners is that it has sound guarantees of performance. Using the Hoeffding bound one can show that its output is asymptotically nearly identical to that of a non-incremental learner using infinitely many examples.

Domingos *et al.* [8] improved the Hoeffding Tree algorithm with an extended method called VFDT, with the following characteristics:

- Ties: when two attributes have similar split gain G, the improved method splits if the Hoeffding bound computed is lower than a certain threshold parameter τ.

$$G(\text{Best Attr.}) - G(\text{2nd best}) < \sqrt{\frac{R^2 \ln 1/\delta}{2n}} < \tau$$

- To speed up the process, instead of computing the best attributes to split every time a new instance arrives, it computes them every time a number n_{min} of instances has arrived.
- To reduce the memory used in the mining, it deactivates the least promising nodes that have lower $p_l \times e_l$ where
  - p_l is the probability to reach leaf l
  - e_l is the error in the node l
  - It is possible to initialize the method with an appropriate decision tree. Hoeffding Trees can grow slowly and performance can be poor initially so this extension provides an immediate boost to the learning curve.

A way to improve the classification performance of the Hoeffding Tree is to use Naive Bayes learners at the leaves instead of the majority class classifier. Gama *et al.* [9] were the first to use Naive Bayes in Hoeffding Tree leaves, replacing the majority class classifier. However, Holmes *et al.* [10] identified situations where the Naive Bayes method outperformed the standard Hoeffding tree initially but is eventually overtaken. To solve that, they proposed a hybrid adaptive method that generally outperforms the two original prediction methods for both simple and complex concepts.

The Hoeffding Adaptive Tree [11] is an extension of the Hoeffding Tree that uses ADWIN as a change detector, to adapt the tree structure of the decision tree to the changes in the distribution of the learning data. Users can use the Hoeffding Adaptive Tree easily without needing to set parameters that depend on the scale of the data change.

3.2. *Ensemble Methods*

Ensemble methods are combinations of several models whose individual predictions are combined in some manner (e.g., averaging or voting) to form a final prediction. When tackling non-stationary concepts, ensembles of classifiers have several advantages over single classifier methods: they are easy to scale and parallelize, they can adapt to change quickly by pruning under-performing parts of the ensemble, and they therefore usually also generate more accurate concept descriptions.

Bagging, boosting and stacking are traditional ensemble methods for non-streaming environments. Usually ensemble methods outperform single classifiers at the cost of more time and memory resources.

Bagging is one of the simplest ensemble methods to implement. Non-streaming bagging [12] builds a set of M base models, training each model with a bootstrap sample of size N created by drawing random samples with replacement from the original training set. Each base model's training set contains each of the original training examples K times where $P(K = k)$ follows a binomial distribution:

$$P(K = k) = \binom{n}{k} p^k (1-p)^{n-k} = \binom{n}{k} \frac{1}{n}^k \left(1 - \frac{1}{n}\right)^{n-k}$$

This binomial distribution for large values of n tends to a Poisson(1) distribution, where Poisson(1)$= \exp(-1)/k!$. Using this fact, Oza and Russell [13, 14] proposed *Online Bagging*, an online method that instead of sampling with replacement, gives each example a weight according to Poisson(1). Figure 4 shows the pseudocode of this ensemble method.

ONLINE BAGGING(M)

Input: M - number of classifiers in the ensemble

1 Initialize base models h_m for all $m \in \{1, 2, ..., M\}$
2 **for** each example (x, y) in Stream
3 **do for** $m = 1, 2, ..., M$
4 **do** Set $w = Poisson(1)$
5 Update h_m with the current example with weight w

6 **anytime output:**
7 **return** hypothesis: $h_{fin}(x) = \arg\max_{y \in Y} \sum_{t=1}^{T} I(h_t(x) = y)$

Fig. 4. Oza and Russell's *Online Bagging* for M models.

Example 1. Let D be a dataset of 4 instances : A, B, C, D. Imagine that we have 5 classifiers, and we run a non-streaming bagging, performing sampling with replacement. The inputs for each classifier will be the following:

Classifier 1: B, A, C, B
Classifier 2: D, B, A, D
Classifier 3: B, A, C, B
Classifier 4: B, C, B, B
Classifier 5: D, C, A, C

And this is equivalent to the following sorted inputs:

Classifier 1: A, B, B, C: A(1) B(2) C(1) D(0)
Classifier 2: A, B, D, D: A(1) B(1) C(0) D(2)
Classifier 3: A, B, B, C: A(1) B(2) C(1) D(0)
Classifier 4: B, B, B, C: A(0) B(3) C(1) D(0)
Classifier 5: A, C, C, D: A(1) B(0) C(2) D(1)

So, to perform bagging in a data streaming setting, we just need to assign each new instance that arrives a weight of Poisson(1).

When data is evolving over time, it is important that models adapt to the changes in the stream and evolve over time. `ADWIN` bagging [15] is the online bagging method of Oza and Russell with the addition of the `ADWIN` algorithm as a change detector and as an estimator for the weights of the boosting method. When a change is detected, the worst classifier of the ensemble of classifiers is removed and a new classifier is added to the ensemble.

A more powerful adaptive bagging exists that extends `ADWIN` bagging, called *leveraging bagging* [16]. It leverages the performance of bagging, with two randomization improvements: increasing resampling and using output detection codes. Figure 5 shows the pseudocode of this method.

LEVERAGING BAGGING(M)

Input: M - number of classifiers in the ensemble

1 Initialize base models h_m for all $m \in \{1, 2, ..., M\}$
2 Compute for each classifier m and class y a binary output code matrix $\mu_m(y)$
3 **for** each example (x, y) in Stream
4 **do for** $m = 1, 2, ..., M$
5 **do** Set $w = Poisson(\lambda)$
6 Update h_m with the current example with weight w and binary mapped class $\mu_m(y)$
7 **if** `ADWIN` detects change in error of one of the classifiers
8 **then** Replace classifier with higher error with a new one

9 **anytime output:**
10 **return** hypothesis: $h_{fin}(x) = \arg\max_{y \in Y} \sum_{t=1}^{T} I(h_t(x) = \mu_t(y))$

Fig. 5. *Leveraging Bagging* for M models.

Resampling with replacement is done in Online Bagging using Poisson(1). There are other sampling mechanisms:

- Lee and Clyde [17] uses the Gamma distribution (Gamma(1,1)) to obtain a Bayesian version of Bagging. Note that Gamma(1,1) is equal to Exp(1).
- Bulhman and Yu [18] propose subagging, using resampling without replacement.

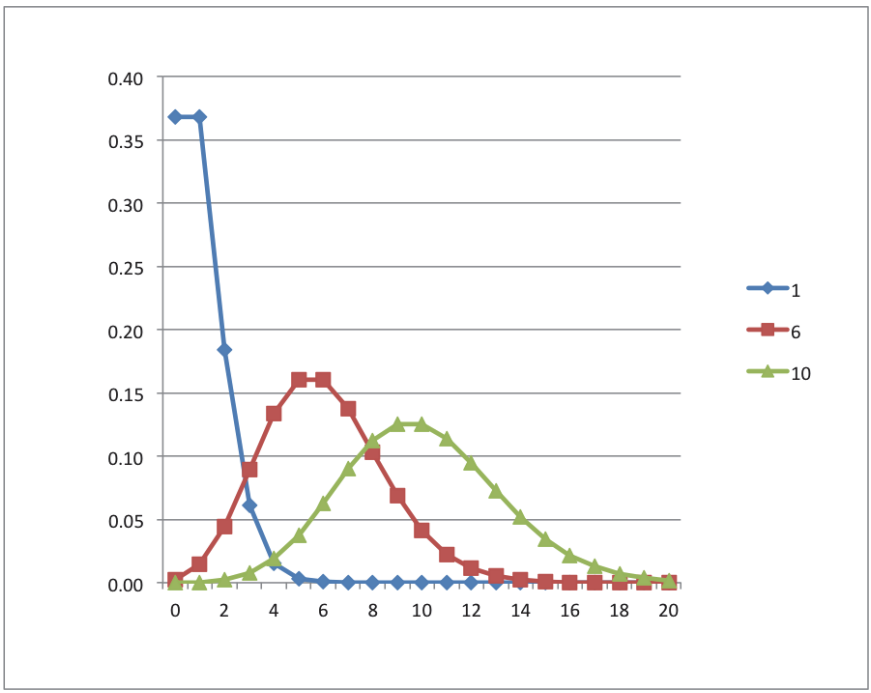

Fig. 6. Poisson distribution.

Leveraging bagging increases the weights of this resampling using a larger value λ to compute the value of the Poisson distribution. The Poisson distribution is used to model the number of events occurring within a given time interval.

Figure 6 shows the probability function mass of the distribution of Poisson for several values of λ. The mean and variance of a Poisson distribution is λ. For $\lambda = 1$ we see that 37% of the values are zero, 37% are one, and 26%

are values greater than one. Using a weight of Poisson(1) we are taking out 37% of the examples, and repeating 26% of the examples, in a similar way to non streaming bagging. For $\lambda = 6$ we see that 0.25% of the values are zero, 45% are lower than six, 16% are six, and 39% are values greater than six. Using a value of $\lambda > 1$ for Poisson(λ) we are increasing the diversity of the weights and modifying the input space of the classifiers inside the ensemble. However, the optimal value of λ may be different for each dataset.

A second improvement is to add randomization at the output of the ensemble using output codes. Dietterich and Bakiri [19] introduced a method based on error-correcting output codes, which handles multiclass problems using only a binary classifier. The classes assigned to each example are modified to create a new binary classification of the data induced by a mapping from the set of classes to {0,1}. A variation of this method by Schapire [20] presented a form of boosting using output codes.

In leveraging bagging, we assign to each class a binary string of length n and then build an ensemble of n binary classifiers. Each of the classifiers learns one bit for each position in this binary string. When a new instance arrives, we assign x to the class whose binary code is closest. We can view an error-correcting code as a form of voting in which a number of incorrect votes can be corrected.

Leveraging bagging uses random output codes instead of deterministic codes. In standard ensemble methods, all classifiers try to predict the same function. However, using output codes each classifier will predict a different function. This may reduce the effects of correlations between the classifiers, and increase diversity of the ensemble.

Random output codes are implemented in the following way: we choose for each classifier m and class c a binary value $\mu_m(c)$ in a uniform, independent, and random way. We ensure that exactly half of the classes are mapped to 0. The output of the classifier for an example is the class which has more votes of its binary mapping classes. Table 1 shows an example for an ensemble of 6 classifiers in a classification task of 3 classes.

Leveraging bagging is an extension of `ADWIN` bagging and uses the same strategy to deal with concept drift. Algorithm 5 shows the pseudo-code for Leveraging Bagging. First it builds a matrix with the values of μ for each classifier and class. For each new instance that arrives, it gives it a random weight of $Poisson(k)$. It trains the classifier with this weight, and when a change is detected, the worst classifier of the ensemble of classifiers is removed and a new classifier is added to the ensemble. To predict the class of an example, it computes for each class c the sum of the votes for $\mu(c)$ of

Table 1. Example matrix of random output codes for 3 classes and 6 classifiers.

	Class 1	Class 2	Class 3
Classifier 1	0	0	1
Classifier 2	0	1	1
Classifier 3	1	0	0
Classifier 4	1	1	0
Classifier 5	1	0	1
Classifier 6	0	1	0

all the ensemble classifiers, and outputs as a prediction the class with the most votes.

3.3. *Classification in MOA*

MOA contains stream generators, classifiers and evaluation methods. Figure 7 shows the MOA graphical user interface. However, a command line interface is also available.

Considering data streams as data generated from pure distributions, MOA models a concept drift event as a weighted combination of two pure distributions that characterizes the target concepts before and after the drift. Within the framework, it is possible to define the probability that instances of the stream belong to the new concept after the drift. It uses the sigmoid function, as an elegant and practical solution [15, 21].

MOA contains the data generators most commonly found in the literature. MOA streams can be built using generators, reading ARFF files, joining several streams, or filtering streams. They allow for the simulation of a potentially infinite sequence of data. The following generators are currently available: Random Tree Generator, SEA Concepts Generator, STAGGER Concepts Generator, Rotating Hyperplane, Random RBF Generator, LED Generator, Waveform Generator, and Function Generator.

MOA contains several classifier methods such as: Naive Bayes, Decision Stump, Hoeffding Tree, Hoeffding Adaptive Tree [11], Hoeffding Option Tree [22], Bagging, Boosting, Bagging using `ADWIN`, Bagging using Adaptive-Size Hoeffding Trees [15] and Leveraging Bagging [16].

For example, a non-trivial example of the EvaluateInterleavedTestThenTrain task creating a comma separated values file, training the HoeffdingTree classifier on the WaveformGenerator data, training and testing on a total of 100 million examples, and testing every one million examples, is encapsulated by the following command line:

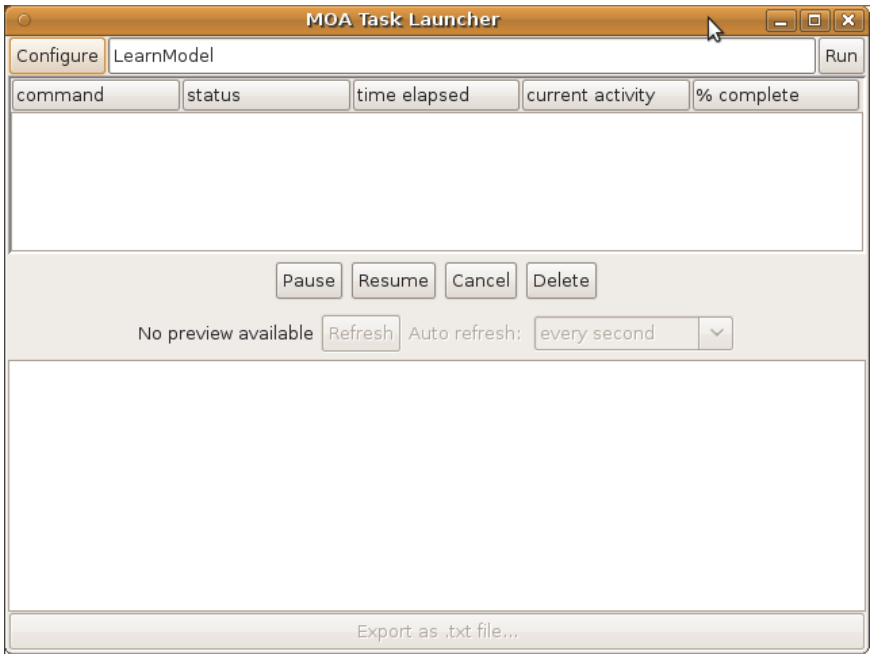

Fig. 7. MOA graphical user interface.

```
java -cp.:moa.jar:weka.jar -javaagent:sizeofag.jar moa.DoTask\
  "EvaluateInterleavedTestThenTrain -l HoeffdingTree \
  -s generators.WaveformGenerator \
  -i 100000000 -f 1000000" > htresult.csv
```

MOA is easy to use and extend. A simple approach to writing a new classifier is to extend `moa.classifiers.AbstractClassifier`, which will take care of certain details to ease the task.

4. Clustering

Clustering is an *unsupervised* learning task that mines unlabeled data. It is useful, when we have unlabeled data, and we want to find relevant groups in the data. Clustering consists in the distribution of a set of instances into non-known groups according to some common relations or affinities. The main difference with classification is that the groups are not-known before starting the learning process. There are many examples of clustering:

market segmentation of customers, or finding social network communities are two examples.

We can see clustering as an optimization problem, where we want to optimize a cost function. Some clustering methods needs the k parameter to know the quantity of clusters to find in the data, and other methods does not have any restriction in the number of clusters to find in the data.

The k-means clustering method is one of the most used methods in clustering, due to its simplicity. The k-means algorithm starts selecting k centroids in a random way. After that there are two main steps: first assign to each instance the nearest point, and second, recompute the cluster centroids using these new assignments. This is done in a iterative way, until a stopping criterion is accomplished, mainly based in the sum of the distance to the centroids. k-means is not a streaming method as it requires to do several passes over the data.

Streaming methods for clustering have two phases, an on-line and an off-line phase. In the on-line, a set of micro-clusters is computed and updated in a very fast way, and in the off-line phase, a classical batch clustering method as k-means is performed using the micro-clusters computed in the on-line phase. The on-line phase is doing only one pass over the data, and the off-line phase is doing several passes, but not over all the data, only over the set of micro-clusters, usually a small set of less than 200 points.

4.1. *Clustering in MOA*

MOA contains also an experimental framework for clustering data streams, so that it will be easy for researchers to run experimental data stream benchmarks. The features of MOA for stream clustering are:

- data generators for evolving data streams (including events such as novelty, merge, etc. [23]),
- an extensible set of stream clustering algorithms,
- evaluation measures for stream clustering,
- visualization tools for analyzing results and comparing different settings.

For stream clustering we added new data generators that support the simulation of cluster evolution events such as merging or disappearing of clusters [23].

Fig. 8. Visualization tab of the clustering MOA graphical user interface.

MOA contains several stream clustering algorithms such as the following ones:

- StreamKM++ [24]: It computes a small weighted sample of the data stream and it uses the k-means++ algorithm as a randomized seeding technique to choose the first values for the clusters. To compute the small sample, it employs coreset constructions using a coreset tree for speed up.
- CluStream [25]: It maintains statistical information about the data using micro-clusters. These micro-clusters are temporal extensions of cluster feature vectors. The micro-clusters are stored at snapshots in time following a pyramidal pattern. This pattern allows to recall summary statistics from different time horizons.
- ClusTree [26]: It is a parameter free algorithm automatically adapting to the speed of the stream and it is capable of detecting concept drift,

novelty, and outliers in the stream. It uses a compact and self-adaptive index structure for maintaining stream summaries.

- Den-Stream [27]: It uses dense micro-clusters (named core-micro-cluster) to summarize clusters. To maintain and distinguish the potential clusters and outliers, this method presents core-micro-cluster and outlier micro-cluster structures.
- D-Stream [28]: This method maps each input data record into a grid and it computes the grid density. The grids are clustered based on the density. This algorithm adopts a density decaying technique to capture the dynamic changes of a data stream.
- CobWeb [29]. One of the first incremental methods for clustering data. It uses a classification tree. Each node in a classification tree represents a class (concept) and is labeled by a probabilistic concept that summarizes the attribute-value distributions of objects classified under the node.

MOA contains measures for analyzing the performance of the clustering models generated. It contains measures commonly used in the literature as well as novel evaluation measures to compare and evaluate both online and offline components. The available measures evaluate both the correct assignment of examples [30] and the compactness of the resulting clustering. The visualization component (cf. Figures 8 and 9) allows to visualize the stream as well as the clustering results, choose dimensions for multi dimensional settings, and compare experiments with different settings in parallel.

Beside providing an evaluation framework, the second key objective is the extensibility of the benchmark suite regarding the set of implemented algorithms as well as the available data feeds and evaluation measures.

Figure 8 shows a screenshot of our visualization tab. For this screenshot two different settings of the CluStream algorithm [25] were compared on the same stream setting (including merge/split events every 50000 examples) and five measures were chosen for online evaluation (CMD, F1, Precision, Recall and SSQ). The upper part of the GUI offers options to pause and resume the stream, adjust the visualization speed, choose the dimensions for x and y as well as the components to be displayed (points, micro- and macro clustering and ground truth). The lower part of the GUI displays the measured values for both settings as numbers (left side, including mean values) and the currently selected measure as a plot over the arrived examples (right, F1 measure in this example). For the given setting one can see a clear drop in the performance after the split event at roughly

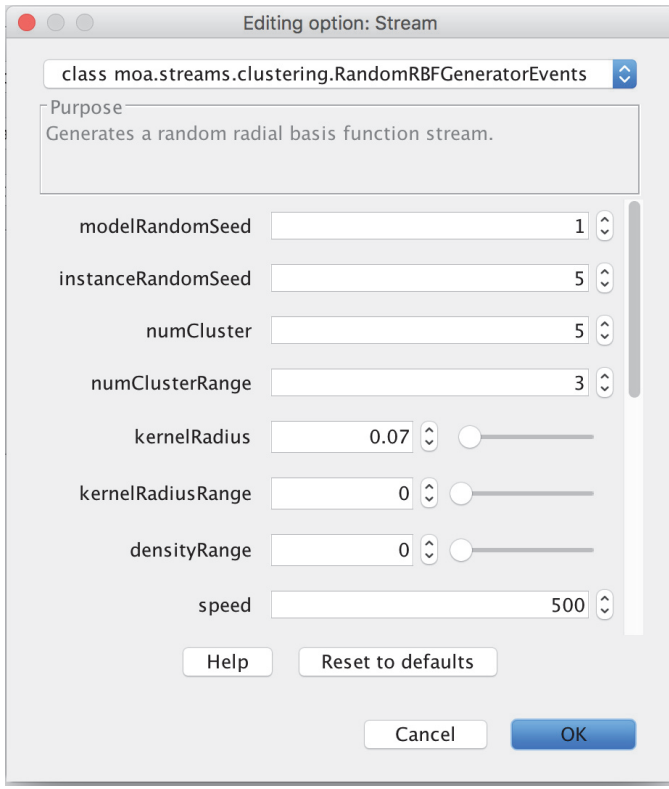

Fig. 9. Option dialog for the RBF data generator (by storing and loading settings benchmark streaming data sets can be shared for repeatability and comparison).

160000 examples (event details are shown when choosing the corresponding vertical line in the plot). While this holds for both settings, the left configuration (red, CluStream with 100 micro clusters) is constantly outperformed by the right configuration (blue, CluStream with 20 micro clusters).

5. Conclusions

MOA is a classification and clustering system for massive data streams with the following characteristics:

- benchmark streaming data sets through stored, shared, and repeatable settings for the various data feeds and noise options, both synthetic and real,

- set of implemented algorithms for comparison to approaches from the literature,
- open source tool and framework for research and teaching similar to WEKA.

MOA is written in Java. The main benefits of Java are portability, where applications can be run on any platform with an appropriate Java virtual machine, and the strong and well-developed support libraries. Use of the language is widespread, and features such as the automatic garbage collection help to reduce programmer burden and error.

MOA can be found at:

`http://moa.cms.waikato.ac.nz/`

The website includes a tutorial, an API reference, a user manual, and a manual about mining data streams. Several examples of how the software can be used are available. The sources are publicly available and are released under the GNU GPL license.

The core team and the community developers of MOA plan to continue extending MOA by adding more classification methods, outlier detection, multi-label and multi-target learning, and frequent pattern mining methods.

References

[1] R. Kirkby. *Improving Hoeffding Trees*. PhD thesis, University of Waikato (November, 2007).

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

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

[4] A. Bifet and R. Gavaldà. Learning from time-changing data with adaptive windowing. In *SIAM International Conference on Data Mining* (2007).

[5] M. Datar, A. Gionis, P. Indyk, and R. Motwani, Maintaining stream statistics over sliding windows, *SIAM Journal on Computing*. **14**(1), 27–45 (2002).

[6] J. Gama, R. Sebastião, and P. P. Rodrigues. Issues in evaluation of stream learning algorithms. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 329–338 (2009).

[7] A. Bifet, G. D. F. Morales, J. Read, G. Holmes, and B. Pfahringer. Efficient online evaluation of big data stream classifiers. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10–13, 2015*, pp. 59–68 (2015).

[8] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 71–80 (2000).
[9] J. Gama and P. Medas, Learning decision trees from dynamic data streams, *Journal of Universal Computer Science*, **11**(8), 1353–1366 (2005).
[10] G. Holmes, R. Kirkby, and B. Pfahringer. Stress-testing hoeffding trees. In *PKDD*, pp. 495–502 (2005).
[11] A. Bifet and R. Gavaldà. Adaptive learning from evolving data streams. In *8th International Symposium on Intelligent Data Analysis*, pp. 249–260 (2009).
[12] L. Breiman, Bagging predictors, *Machine Learning*, **24**(2), 123–140 (1996). ISSN 0885-6125.
[13] N. C. Oza and S. J. Russell. Experimental comparisons of online and batch versions of bagging and boosting. In *KDD*, pp. 359–364 (2001).
[14] N. Oza and S. Russell. Online bagging and boosting. In *Artificial Intelligence and Statistics 2001*, pp. 105–112, Morgan Kaufmann (2001).
[15] A. Bifet, G. Holmes, B. Pfahringer, R. Kirkby, and R. Gavaldà. New ensemble methods for evolving data streams. In *15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2009).
[16] A. Bifet, G. Holmes, and B. Pfahringer. Leveraging bagging for evolving data streams. In *ECML/PKDD (1)*, pp. 135–150 (2010).
[17] H. K. H. Lee and M. A. Clyde, Lossless online bayesian bagging, *Journal of Machine Learning Research*, **5**, 143–151 (2004). ISSN 1532-4435.
[18] P. Bühlmann and B. Yu, Analyzing bagging, *Annals of Statistics* (2003).
[19] T. G. Dietterich and G. Bakiri, Solving multiclass learning problems via error-correcting output codes, *Journal of Artificial Intelligence Research (JAIR)*, **2**, 263–286 (1995).
[20] R. E. Schapire. Using output codes to boost multiclass learning problems. In *ICML '97: Proceedings of the Fourteenth International Conference on Machine Learning*, pp. 313–321, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1997). ISBN 1-55860-486-3.
[21] A. Bifet, G. Holmes, B. Pfahringer, and R. Gavaldà. Improving adaptive bagging methods for evolving data streams. In *First Asian Conference on Machine Learning, ACML 2009* (2009).
[22] B. Pfahringer, G. Holmes, and R. Kirkby. Handling numeric attributes in hoeffding trees. In *PAKDD Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pp. 296–307 (2008).
[23] M. Spiliopoulou, I. Ntoutsi, Y. Theodoridis, and R. Schult. MONIC: modeling and monitoring cluster transitions. In *ACM KDD*, pp. 706–711 (2006).
[24] M. R. Ackermann, C. Lammersen, M. Märtens, C. Raupach, C. Sohler, and K. Swierkot. StreamKM++: A clustering algorithm for data streams. In *SIAM ALENEX* (2010).
[25] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for clustering evolving data streams. In *VLDB*, pp. 81–92 (2003).

[26] P. Kranen, I. Assent, C. Baldauf, and T. Seidl. The ClusTree: indexing micro-clusters for anytime stream mining. *Knowledge and information systems*, **29**(2), 249–272 (2011).

[27] F. Cao, M. Ester, W. Qian, and A. Zhou. Density-based clustering over an evolving data stream with noise. In *SDM* (2006).

[28] L. Tu and Y. Chen, Stream data clustering based on grid density and attraction, *ACM Transactions on Knowledge Discovery from Data*, **3**(3), 1–27 (2009). ISSN 1556-4681. doi: http://doi.acm.org/10.1145/1552303.1552305.

[29] D. H. Fisher, Knowledge acquisition via incremental conceptual clustering, *Machine Learning*, **2**(2), 139–172 (1987). ISSN 0885-6125. doi: http://dx.doi.org/10.1023/A:1022852608280.

[30] J. Chen. Adapting the right measures for k-means clustering. In *ACM KDD*, pp. 877–884 (2009).