Big Data Streaming Using Adaptive Machine Learning And Mining Algorithms

Samson Immanuel and Dr. A.P. Haran

Assistant Professor, Karunya University, School of Electrical Sciences, Department of ECE, Samson Immanuel, Box.641114. Coimbatore. Tamil Nadu, India

Dean Mechanical and Aerospace, Park College of Engineering and Technology, Dr. A.P. Haran, Coimbatore, Tamil Nadu. India

ABSTRACT

Big data stream model, data arrive at high speed, and the algorithms that must process them have very strict constraints of space and time. In this we propose and illustrate a framework for developing algorithms that can adaptively learn from big data streams that change over time. Our methods are based on using change detectors and estimator modules at the right places. The propose an modified adaptive sliding window algorithm MADWIN for detecting change and keeping updated statistics from a Big data stream, and use it as a black-box in place or counters or accumulators in algorithms initially not designed for drifting data. Since MADWIN has rigorous performance guarantees, the cost of hardware has declined relative to the cost of energy, the energy efficiency and environmental impact of computing systems and programs are receiving increased attention. Since MADWIN has rigorous performance guarantees, this opens the possibility of extending such guarantees to the resulting learning algorithm. The main advantage of the methods is that they require no guess about how fast or how often the stream will change; typically have several user-defined parameters to this effect.

INTRODUCTION

In today’s information society, extraction of knowledge (Joo Gama, 2003) is becoming a very important task for many people. We live in an age of knowledge revolution. This knowledge revolution is based in an economic change from adding value by producing things which is, ultimately limited, to adding value by creating and using knowledge which can grow indefinitely. To deal with these huge amounts of data in a responsible way, green computing is becoming a necessity. Green computing (Albert Bifet, 2005) is the study and practice of using computing resources efficiently. A main approach to green computing is based on algorithmic efficiency.

Big data mining techniques that we will use come essentially from Machine Learning (Albert Bifet, 2009). In particular, we will use the traditional distinction between supervised and unsupervised learning. In supervised methods data instances are labeled with a “correct answer” and in unsupervised methods they are unlabeled. Clusters are typical examples of unsupervised methods. Classification is the distribution of a set of instances of examples into groups or classes according to some common relations or affinities.

The Big Data Stream model represents input data that arrives at high speed (Charu C, 2006). This data is so massive that we may not be able to store all of what we see, and we don’t have too much time to process it. It requires that at a time t in a data stream with domain N, this three performance measures: the per-item processing time, storage and the computing time to be simultaneously preferably log (N,t).

Big Data Stream Mining:

The main challenge is that ‘Big data-intensive’ mining is constrained by limited resources of time, memory, and sample size. Data mining has traditionally been performed over static datasets, where data mining algorithms can afford to read the input data several times.

The following constraints apply in the Big Data Stream model:

1. The amount of data that has arrived and will arrive in the future is extremely large; in fact, the sequence is potentially infinite. Thus, it is impossible to store it all. Only a small summary can be computed and stored, and
the rest of the information is thrown away. Even if the information could be all stored, it would be unfeasible to go over it for further processing.
2. The speed of arrival is large, so that each particular element has to be processed essentially in real time, and then discarded.
3. The distribution generating the items can change over time. Thus, data from the past may become irrelevant (or even harmful) for the current summary.

An interesting approach to mining big data streams is to use a sliding window to analyze them (Brain Babcock, 2003). This technique is able to deal with concept drift. The main idea is instead of using all data seen so far, use only recent data. We can use a window of size W to store recent data, and deleting the oldest item when inserting the newer one. An element arriving at time t expires at time t + W.

In the case of trees, only labeled tree mining methods are considered in the literature. There are four broad kinds of subtrees: bottom-up subtrees, top-down subtrees, induced subtrees, and embedded subtrees. Bottom-up subtree mining is the simplest from the subtree mining point of view.

Algorithms for embedded labeled frequent trees include:

- Rooted Ordered Trees
  - TreeMiner: This algorithm, developed by (Zaki Mohammed J. Zaki, 2002), uses vertical representations for support counting, and follows the combined depth-first/breadth traversal idea to discover all embedded ordered subtrees.
- Rooted Unordered Trees
  - SLEUTH: This method, also by Zaki, extends Tree Miner (Mohammed Javeed Zaki, 2005) to the unordered case using two different methods for generating canonical candidates: the class-based extension and the canonical extension.

Frequent subtree mining can be found in (Yun Chi, 2001). Arimura and Uno proposed CLOATT considering closed mining in attribute trees, which is a subclass of labeled ordered trees and can also be regarded as a fragment of description logic with functional roles only. These attribute trees are defined using a relaxed tree inclusion. Termier et al. considered the frequent closed tree discovery problem for a class of trees with the same constraint as attribute trees. Labeled trees are trees in which each vertex is given a unique label. Unlabeled trees are trees in which each vertex has no label, or there is a unique label for all vertices.

Frequent Pattern Mining:

Patterns are graphs, composed by a labeled set of nodes (vertices) and a labeled set of edges. The number of nodes in a pattern is called its size. Examples of patterns are item sets, sequences, and trees (Mohammed Javeed Zaki, 2005). The (infinite) set of all patterns will be denoted with T, but actually all our developments will proceed in some finite subset of T which will act as our universe of discourse. The input to our big data mining process, now is a given finite dataset D of transactions, where each transaction s 2 D consists of a transaction identifier, t_id, and a pattern. Tids are supposed to run sequentially from 1 to the size of D. From that dataset, our universe of discourse U is the set of all patterns that appear as subpattern of some pattern in D.

The algorithms on unlabeled trees can be found in Gabriel (Valiente, 2002). Unlabeled trees are trees in which each vertex has no label, or there is a unique label for all vertices. A comprehensive introduction to the algorithms on unlabeled trees can be found.

Algorithms for Big data mining with change:

In this section we review some of the data mining methods that deal with big data streams and concept drift. There are many algorithms in the literature that address this problem.

FLORA: Widmer and Kubat:

FLORA is a supervised incremental learning system that takes as input a big data stream of positive and negative example of a target concept that changes over time. The original FLORA algorithm uses a fixed moving window approach to process the big data. The concept definitions are stored into three description sets:

- ADES description based on positive examples
- NDES descriptions based on negative examples
- PDES concept descriptions based on both positive and negative examples

The algorithm was further improved to allow previously (FLORA1), (FLORA2) extracted knowledge to help deal with recurring concepts (FLORA3) and to allow it to handle noisy data (FLORA4).

Support Vector Machines: Klinkenberg:

Klinkenberg and Joachims presented a method to handle concept drift with support vector machines. A proper introduction to SVM, their method maintains a window on the training data with an appropriate size
without using a complicated parameterization. The key idea is to automatically adjust the window size so that the estimated generalization error on new examples is minimized.

This reflects the assumption that the most recent examples are most similar to the new examples in batch \( t + 1 \). The window size minimizing the estimate of the error rate is selected by the algorithm and used to train a classifier for the current batch.

The window adaptation algorithm is shown in figure 3.1.

**SVMWINDOWSIZE:**

1. For \( h \in \{0; \ldots ; t - 1\} \)
2. Do train SVM on examples \( z(t-h;1); \ldots ; z(t;m) \)
3. Compute \( \hat{\mathcal{E}} \) estimate on examples \( z(t-h;1); \ldots ; z(t;m) \)
4. Return \( \hat{\mathcal{E}} \) which minimizes \( \hat{\mathcal{E}} \) estimate.

Fig.1: Window size adaption algorithm.

**OLIN: Last**

Last in [Las02] describes an online classification system that uses the infofuzzy network (IFN). The system called OLIN (On Line Information Network) gets a continuous stream of non-stationary data and builds a network based on a sliding window of the latest examples.

The system dynamically adapts the size of the training window and the frequency of model reconstruction to the current rate of concept drift.

OLIN uses the statistical significance of the difference between the training and the validation accuracy of the current model as an indicator of concept stability.

OLIN adjusts dynamically the number of examples between model reconstructions by using the following heuristic: keep the current model for more examples if the concept appears to be stable and reduce drastically the size of the validation window, if a concept drift is detected.

OLIN generates a new model for every new sliding window. This approach ensures accurate and relevant models over time and therefore an increase in the classification accuracy. However, the OLIN algorithm has a major drawback, which is the high cost of generating new models. OLIN does not take into account the costs involved in replacing the existing model with a new one.

**CVFDT: Domingos**

It uses analytical techniques to choose the splitting criteria, and the information gain to estimate the merit of each possible splitting-test. For multi-class problems, the algorithm builds a binary tree for each possible pair of classes leading to a forest-of-trees. During the training phase the algorithm maintains a short term memory. Given a data stream, a limited number of the most recent examples are maintained in a data structure that supports constant time insertion and deletion. When a test is installed, a leaf is transformed into a decision node with two descendant leaves. The sufficient statistics of the leaf are initialized with the examples in the short term memory that will fall at that leaf.

**CVFDT((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 Add, Remove and Forget Examples
5. CVFDTGROW\((x; y); HT, δ\)
6. CHECKSPLITVALIDITY\((HT; n, δ)\)
7. CVFDTGROW\((x; y); HT, _\)
8. Sort \((x; y)\) to leaf \( l \) using \( HT \)
9. Update counts \( n_{ijk} \) at leaf \( l \) and nodes traversed in the sort
10. If examples seen so far at \( l \) are not all of the same class
11. Then Compute G for each attribute
12. If \( G(\text{Best Attr.}) - G(2\text{nd best}) > \sqrt{\frac{R^2 \ln \frac{1}{δ}}{2n}} \)
13. Then Split leaf on best attribute
14. For each branch
15. Do Start new leaf and initialize counts
16. Create alternate subtree
17. CHECKSPLITVALIDITY\((HT; n, δ)\)
for each node $l$ in HT that it is not a leaf
  2 do for each tree $T_{alt} \in ALT(l)$
  3 do CHECKSPLITVALIDITY($T_{alt}; n, \delta$)
  4 if exists a new promising attributes at node $l$
  5 do Start an alternate subtree

The CVFDT algorithm

AWIN:

The starting point of our work is the following observation: In the big data stream mining literature, most algorithms incorporate one or more of the following ingredients: windows to remember recent examples; methods for detecting distribution change in the input.

![Diagram](image)

Our claim is that by basing mining algorithms on well-designed, well-capsulated modules for these tasks, one can often get more generic and more efficient solutions than by using ad-hoc techniques as required.

MADBDS:

Most approaches for predicting and detecting change in streams of data can be discussed in the general framework: The system consists of three modules: a Memory module, an Estimator Module, and a Change Detector or Alarm Generator module. These three modules interact as shown in Figure.

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, estimation of some important parameters of the input distribution, and a signal alarm indicating that distribution change has recently occurred.

We consider a specific, but very frequent case, of this setting: that in which all the $x_t$ are real values. The desired estimation is usually the expected value of the current $x_t$, and less often another distribution statistic, such as the variance. The only assumption on the distribution is that each $x_t$ is drawn independently from each other.

Memory is the component where the algorithm stores all the sample data or summary that considers relevant at current time, that is, that presumably shows the current data distribution.

The Estimator component is an algorithm that estimates the desired statistics on the input data, which may change over time. The algorithm may or may not use the data contained in the Memory. The simplest Estimator algorithm for the expected is the linear estimator, which simply returns the average of the data items contained in the Memory. Other examples of run-time efficient estimators are Auto-Regressive, Auto Regressive Moving Average, and Kalman filters.

Table 3.1: We classify these predictors in four classes, depending on whether Change Detector and Memory modules exist:
**Experiment Results:**

Massive Online Analysis (MOA) is a framework for online learning from continuous data streams. The data stream evaluation framework and most of the classification algorithms evaluated in this thesis were implemented in the Java programming language extending the MOA framework.

MOA includes a collection of offline and online methods as well as tools for evaluation. In particular, it implements boosting, bagging, and Hoeffding Trees, all with and without Naïve Bayes classifiers at the leaves.

MOA is related to WEKA, the Waikato Environment for Knowledge Analysis Geoffrey Holmes (2007), which is an award-winning open-source workbench containing implementations of a wide range of batch machine learning methods. WEKA is also written in Java. The main benefits of Java are portability.

Our first version of ADWIN0 is computationally expensive, because it checks exhaustively all “large enough” subwindows of the current window for possible cuts. Furthermore, the contents of the window is kept explicitly, with the corresponding memory cost as the window grows. To reduce these costs we present a new version MADWIN using ideas developed in data stream algorithmic to find a good cut point quickly.

**MADWIN: Modified ADAPTIVE WINDOWING ALGORITHM**

1. Initialize W as an empty list of buckets
2. Initialize WIDTH, VARIANCE and TOTAL
3. for each t > 0
4. do SETINPUT(xt; W)
5. output \( \frac{\text{TOTAL}}{\text{WIDTH}} \) and ChangeAlarm
SETINPUT(item e, List W)
1. INSERTELEMENT(e, W)
2. repeat DELETEELEMENT(W)
3. until \( f(W) - f(W') < \text{cut holds} \)
4. for every split of W into W = W0 \( \cup \) W1
INSERTELEMENT(item e, List W)
1. create a new bucket b with content e and capacity 1
2. W = W\[ b \] (i.e., add e to the head of W)
3. update WIDTH, VARIANCE and TOTAL
4. COMPRESSBUCKETS(W)
DELETEELEMENT(List W)
1. remove a bucket from tail of ListW
2. update WIDTH, VARIANCE and TOTAL
3. ChangeAlarm = true
COMPRESSBUCKETS(List W)
1. Traverse the list of buckets in increasing order
2. do If there are more than Mbuckets of the same capacity
3. do merge buckets
4. COMPRESSBUCKETS(sublist of W not traversed)

In this section, we are going to consider the performance of ADWIN in a data stream environment. We are interested in:

- evolution of accuracy
- probability of false alarms probability of true detections for different rates of drift
- average delay time in detection

We construct the following experiments to test the performance of our algorithms:
1. Rate of false positives: we show that the ratio of false positives is as predicted by theory.
2. Accuracy: we compare the estimation accuracy of ADWIN to estimations obtained from fixed-size windows with or without flushing when change is detected. ADWIN often does better than the best fixed-size window.
3. Small probabilities: we show that when the input samples to estimators are generated from small probabilities, then ADWIN beats almost all fixed-size window estimators, with or without flushing.
4. Probability of true detections and average delay time for different rates of drift: we compare the number of true detections and average delay time with DDM, and we observe that ADWIN detects more changes than DDM, but its average delay time of detection sometimes is higher.
5. Accuracy on mining methods as Naïve Bayes and k-means Clustering.

In the first experiment, we investigate the rate of false positives of MADWIN. This is a very important measure, specially when there is a cost associated with a reported change. To do this, we feed MADWIN a data stream of 1000,000 bits, generated from a stationary Bernoulli distribution with parameter \( \mu \) and different confidence parameters \( \delta \).
Table 1.2: Parametic Table.

<table>
<thead>
<tr>
<th>µ</th>
<th>δ = 0.05</th>
<th>δ = 0.1</th>
<th>δ = 0.3</th>
<th>δ = 0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>.01</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>.1</td>
<td>0.0002</td>
<td>0.0004</td>
<td>0.0019</td>
<td>0.0026</td>
</tr>
<tr>
<td>.3</td>
<td>0.0025</td>
<td>0.0006</td>
<td>0.0036</td>
<td>0.0046</td>
</tr>
<tr>
<td>.5</td>
<td>0.0005</td>
<td>0.0008</td>
<td>0.0072</td>
<td>0.0092</td>
</tr>
<tr>
<td>.7</td>
<td>0.0075</td>
<td>0.0010</td>
<td>0.0144</td>
<td>0.0184</td>
</tr>
<tr>
<td>.9</td>
<td>0.0150</td>
<td>0.0012</td>
<td>0.0288</td>
<td>0.0368</td>
</tr>
</tbody>
</table>

In the second set of experiments, we want to compare MADWIN as an estimator with estimations obtained from fixed-size window, and with fixed size window which are flushed when change is detected. In the last case, we use a pair of windows (X, Y) of a fixed size W. Window X is used as a reference window that contains the first W elements of the stream that occurred after the last detected change. Window Y is a sliding window that contains the latest W items in the data stream. To detect change we check whether the difference of the averages of the two windows exceeds threshold $\varepsilon_{cut}$. If it does, we copy the content of window Y into reference window X, and empty the sliding window Y. This scheme is as in Stream Ensemble Algorithm (SEA) and we refer to it as “fixed-size windows with flushing”.

Accuracy on SEA Concepts dataset with drifts

**Conclusion:**

The methods are based on using change detectors and estimator modules at the right places; we choose implementations with theoretical guarantees in order to extend such guarantees to the resulting modified adaptive learning algorithm. We have proposed Modified adaptive sliding window algorithm (MADWIN) for detecting change and keeping updated statistics from a data stream, and use it as a black-box in place or counters or accumulators in algorithms initially not designed for drifting data. Since MADWIN has rigorous performance guarantees, this opens the possibility of extending such guarantees to the resulting learning algorithm. A main advantage of the methods is that they require no guess about how fast or how often the stream will change; other methods typically have several user-defined parameters to this effect.

**REFERENCES**


