Department of Computer Science
The University of Auckland
New Zealand

Learning From Data Streams With Changes in Volatility

Ruolin Jia

February 2017

Supervisors:

Yun Sing Koh

Gill Dobbie

Abstract

Data streams arrive fast, and they are unbounded in size. Thus, reducing time and memory overheads are two of the most important concerns when mining a data stream. A data stream's concept may evolve over time, which is known as the concept drift. Concept drifts will affect the prediction quality of the current learning model and are required to be handled. However, in most cases, there is a trade-off between the prediction quality and cost efficiency for a model. In this thesis, we present a novel framework known as the Volatility-Adaptive Classifier System (VACS). The system contains an adaptive classifier and a non-adaptive classifier. The former can maintain a higher prediction quality but takes more overheads, and the latter is cheap in overheads but its prediction quality may be susceptible to concept drifts. VACS automatically applies the adaptive classifier when the concept drifts are frequent, and switches to the non-adaptive classifier when drifts are infrequent. As a result, VACS can maintain a relatively low computational cost while still maintaining a high enough overall prediction quality. To the best of our knowledge, this is the first data stream mining framework that applies different learners to reduce the learning overheads.

Acknowledgements

I would like to acknowledge and thank my supervisors Yun Sing Koh and Gill Dobbie. I could only complete this dissertation with their kind guidances, patient helps, and eruditions on defining and solving research problems. My thank also goes to my family for their strong support for my education. My special gratitude goes to nature itself for its beautiful complexity such that I can hold the curiosity as a crucial motivation to solve scientific research problems.

Contents

1 Introduction

Data stream is a sequence of unlimited data arriving in real time. The properties of data streams add challenges and concerns for designing data stream learning algorithms. Algorithms need to be efficient regarding time and memory, and also maintains a good prediction quality.

Our research is to design a framework called Volatility Adaptive Classifiers System (VACS). VACS had lower computational cost than any adaptive learner while maintaining a similar prediction quality in a stream with volatility changes.

In this chapter, we present a brief overview of the research problem, objectives, and the resulting system.

A data stream is a special data model that can represent various types of data generated daily. For example, electricity usage records produced by a power station, online tweets generated in a region, transactions recorded in a stock market can all be presented by data streams. Such data are generated in order, have enormous size, and arrive without limits every day. The task of data stream mining is to find valuable information from these unbounded streams of data. Data stream mining requires specially designed learning algorithms which are different from those used in traditional data mining. These requirements raise challenges for developing a data stream learning algorithm. Specifically, instances in a stream can arrive very fast, allowing only limited time for the algorithm to learn its underlying concepts. A data stream also has unbounded size, which means that the learner has to approximate the information from a stream with limited memory. Essentially, a data stream can evolve over time. It means that underlying concepts in a stream can change over time so the model with high prediction accuracy at one time may lose its prediction ability at a later time. This phenomenon is known as concept drifts. To maintain the long-term quality of a learning model, stream learning algorithms are expected to take detection or adaptation actions to overcome these concept drifts. In summary, these properties of a data stream require the data stream learning algorithm to be efficient in time and memory cost and maintain high model quality by overcoming concept drifts.

1.1 Problem & Motivation

The frequency of concept drifts is not constant in a stream [17]. For example, the data collected by a sensor has infrequent concept drifts most of the time. If there is an anomalous event happening, it will affect the sensor, and the number of concept drifts will increase in this anomalous period. A learning model can overcome a concept drift by adjusting its model structure to generalise the new concept and maintain a high prediction quality. This technique is known as the model adaptation. An algorithm that can perform the model adaptation is called the adaptive learner, and the algorithm that does not have such a function is known as the non-adaptive learner. Model adaptations come with a large computational cost. Thus, there is a trade-off between the model quality and the learning efficiency. It is reasonable to apply the adaptive learner when the frequency of concept drifts is high to get the high prediction quality. When the frequency of concept drifts becomes low, it is not necessary to use the adaptive learner with a large cost.

Memory and time are two main concerns when mining a stream. Currently, there is no data stream learning technique considering to use data stream volatility to adjust the model adaptation behaviour to reduce the computational cost. We are addressing this problem by creating a new learning framework containing both adaptive and non-adaptive learners.

1.2 Objectives

Huang et al. [16] implemented a drift detection method showing that steam volatility can be used reliably to optimise stream mining tasks. In this research project, we aim to explore another potential value of the volatility information to optimise the data stream mining task: to apply different learners on volatility-changing streams to reduce the overall time and memory cost of mining data streams while maintaining the high enough prediction quality.

In this thesis, we have two major objectives:

- To develop a data stream classification framework that can automatically choose the most suitable classifier between adaptive and non-adaptive classifiers in real time when mining a volatility-changing stream. An adaptive classifier with larger training overhead will be used when the stream volatility is relatively high, and the non-adaptive classifier with smaller training overhead will be used when the volatility is relatively low.
- To show the framework can work as expected on various streams with volatility changes including both synthetic datasets and real-world datasets.

1.3 Contributions

To meet our objectives, we will make the following contributions:

- Developing Volatility Adaptive Classifiers System (VACS). VACS is able to choose between the adaptive classifier and the non-adaptive classifier given different levels of stream volatility
- Developing the volatility-changing data stream generator which can generate synthetic datasets with volatility changes.
- Evaluating the performance of VACS by comparing with VFDT-ADWIN and HAT-ADWIN (HAT-ADWIN and VFDT-ADWIN are discussed in Chapter 2) on both synthetic and real-world datasets.

1.4 Overview of Technique

VACS is the framework that we developed to address our research problem.

VACS is composed of a pair of classifiers with distinctive properties. One classifier is an adaptive classifier. It is expected to smoothly adapt to concept changes and produces better prediction quality when drifts appear, but its high prediction quality comes with the large computational cost. The other classifier is a non-adaptive classifier. It has smaller time and memory cost but may perform worse than its peer. When learning from a data stream, VACS is expected to take less computational cost compared with applying an adaptive classifier and maintains decent prediction quality by automatically switching between these two classifiers. VACS uses stream volatility [17], which represents the frequency of detected concept drifts in a stream, as the criterion to switch between classifiers. In particular, when the volatility is high, VACS applies the adaptive learner to maintain a better prediction quality in the period with frequent concept drifts. When the volatility is low, it is deemed to be unnecessary to spend considerable computational cost to be prepared for infrequent concept drifts, so it switches to the non-adaptive classifier to reduce the cost. As a result, VACS is expected to have a high enough prediction accuracy with relatively low overheads.

1.5 Disseration Structure

In this introduction chapter, we have discussed our research problem, motivation and main objectives. We have also briefly discussed our contributions and the resulting technique.

There are other 6 chapters in the thesis followed by this introduction chapter.

In Chapter 2, we review the definition of data stream with more details. We also discussed some well-known data stream algorithms for drift detections and classifications.

In Chapter 3, we illustrate how VACS works. VACS is composed of several modules, and we discuss each module through this chapter.

In Chapter 4, we discuss the experimental design. We specify what indicators will be measured in our experiments. We also discuss the implementation for our synthetic data generators used in our experiments.

In Chapter 5, we present the results of various experiments with VACS through tables, plots and analysis. Experiments are run on both synthetic and real-world data. We compare the performance among VACS, HAT-ADWIN, and VFDT-ADWIN. We also investigate how different parameters influence VACS.

In Chapter 6, we discuss the application of VACS after summarising our experimental results. We specify different scenario where VACS is suggested to be used.

In Chapter 7, we conclude our research and discuss several limitations of VACS. We propose some reasonable future works to address those limitations.

2 Literature Review And Preliminaries

Data streams have various distinctive properties compared with the static data stored in classical databases. Thus, data stream mining is different from the traditional data mining in many aspects. Traditional learning algorithms cannot easily mine streaming data. So it requires researchers to develop new techniques to handle this special field of data mining.

In this chapter, we review literature in the field of data stream mining on which our research is based. We will discuss what a data stream is and challenges of handling data streams. We will also discuss some well-known data stream classification algorithms. Lastly, we discuss concept drifts in data streams and methods for handling concept drifts.

2.1 Data Stream Mining Overview

A data stream is a high-speed continuous stream of data in real time. Data streams can be encountered in many contexts. In the business scenario, streaming data can be the 20 million sales transactions collected by WalMart in one day. In the scientific research scenario, earth sensing satellite and astronomical observatories can deliver gigabytes of data each day [9]. Mining knowledge from those data delivers both economic and scientific research benefits. However, the special nature of data stream's make it hard to process the data using traditional learning algorithms. In this section, we will review the definition of data streams, challenges of mining data streams, and solutions to solve these challenges.

In a formal definition, a stream S is an unbounded bag of pairs (s, γ), in which s is a tuple, and γ is its related time-stamp. $\gamma \in \Gamma$ where Γ is the *time domain* [2].

Gama [10] summarised the following features of a data stream:

1. Data are unlimited in size.

2. Data continuously flow at high-speed over time.

3. Data distribution may evolve over time.

4. Instances in the data stream may be dependent on each other.

5. Instances in the data stream have time order.

Given these features, it is either difficult or impossible to mine data streams with traditional machine learning algorithms. Traditional machine learning algorithms developed in the last few decades are mostly batch learning algorithms [10]. Batch learning algorithms need to scan all available data at once, and in most cases, they may need more than one scan over all data. For example, decision tree [23], which is a popular classification algorithm, needs to scan all data in the database several times to construct a tree that performs reliable prediction tasks.

Challenges appear in the context of data streams. In this context, data is unbounded in size and difficult to store in a database. It makes scanning the whole database practically impossible. Moreover, continuity of the stream indicates that arriving data does not necessarily have an end, such that an algorithm can never have the perfect confidence of seeing all data. Additionally, traditional algorithms such as decision tree and regression algorithms assume the data comes from one distribution, but a data stream may evolve over time and its underlying distribution shifts during its evolution. As a consequence, traditional machine learning algorithms may fail to learn in the data stream scenario, so it is necessary for researchers to grow a new branch in the research field of knowledge discovery and data mining.

Based on the nature of data streams, when designing data stream learning algorithms, we need to consider three main requirements.

Firstly, because data streams arrive at high speed, on the one hand, algorithms are required to be fast, for example, have linear time complexity. In particular, the speed of building a prediction model should be higher than the data arrival rate [1]. On the other hand, algorithms can only scan the data once. One example of algorithms implementing these requirements is VFDT (Very Fast Decision Tree) presented in [8]. The algorithm is a special decision tree which needs only one scan over the data to build a classification tree efficiently.

Secondly, algorithms should be able to mine unlimited data with limited memory. Therefore, algorithms should use approximation techniques such as statistical sampling, aggregation functions, or data synopsis to approximate the unbounded data stream. One popular implementation of a stream approximation technique is reservoir sampling [25]. It maintains a fixed-size data buffer known as a reservoir. When a new instance arrives, an old instance in the reservoir is randomly picked and removed to create space for the new instance.

Thirdly, algorithms need to handle the evolution of data. Algorithms should either continuously adapt to concept changes or be notified when a change has occurred and respond correspondingly. The VFDT algorithm mentioned earlier does not satisfy this requirement, because it cannot adapt its tree structure if incoming data has a distribution change. Hulten et al. built the CVFDT (Concept-adapting Very Fast Decision Tree) algorithm [18] based on the original VFDT. Specifically, CVFDT constantly adapts its model consistent with a sliding window representing the most recent distribution of the data. Later on, Bifet et al. [5] proposed another adaptive decision tree known as HAT (Hoeffding Adaptive Tree), which combines the drift detection technique with the adaptive tree and produces better prediction accuracy than CVFDT. HAT is discussed in the later sections in detail.

Analysis

We have presented an overview of the definition, challenges, and corresponding requirements when designing data stream mining algorithm. We introduced existing algorithms addressing those requirements. We also contrasted the differences between traditional algorithms and stream algorithms.

Beyond the above three requirements which have been widely discussed and implemented in various algorithms, we identify an additional requirement that can be considered to improve mining performance when designing the algorithm. Consider two properties of data streams. The first property is concept evolution, where recently collected data has the most valuable information. Secondly, in many stream mining contexts, we have only one chance to inspect each instance in a real-time stream. By considering these two properties, we contribute a new requirement: stream algorithms need to extract as much information as possible by scanning a limited size of data. One example of addressing this requirement is stream volatility detection, which is presented by Huang et al. [17]. Stream volatility is the rate of concept changes in a stream, which is considered as additional information exploited from the stream. Such information has been used for optimising the performance of the drift detector [16]. In a similar way, our research detects volatility as additional information to choose appropriate learners for the current mining tasks.

2.2 Very Fast Decision Tree (VFDT)

In this section, we particularly explore the idea and implementation of the Very Fast Decision Tree (VFDT) algorithm. We briefly discuss the traditional decision tree algorithm and explain VFDT in detail in comparison with the traditional decision tree. While comparing between these two algorithms, we also discuss the motivation for VFDT to incorporate its differences from the original decision tree.

2.2.1 Traditional Decision Tree

Decision tree [23] is a classification model in a tree structure. Each inner node in the tree is a decision node containing an attribute. Each leaf is a classification node containing a class label. When performing the classification task, a data instance is passed from the root. Each decision node (including the root) chooses the next child branch to pass the instance based on the instance's attribute. Once the instance reaches a leaf, the class label in the leaf is selected as the prediction class.

Training a decision tree is the process of growing a single-node tree into a complex tree by recursively splitting each leaf into internal nodes. Nodes are split on data attributes. To split a node by an attribute, the node chooses the attribute splitting the dataset with the largest information gain compared with other attributes. Training a decision tree in the data stream scenario can be difficult [10]. This is because before splitting each node, it needs to scan all instances to decide whether to perform the split. In Section 2.1, we explained that it is not possible for the algorithm to scan all data in a data stream. As a consequence, it is necessary to adjust the traditional decision tree in the new context.

2.2.2 Very Fast Decision Tree (VFDT) Algorithm and Hoeffding Bound

VFDT is a decision tree algorithm that is specifically designed for mining data streams and was originally presented in [8].

The intuition of VFDT is to split an inner node using confident enough instances rather than all instances in a database. When new instances are input in the VFDT, each leaf in the VFDT calculates information gain for each possible splitting attribute using previously seen instances. Information gains on the leaf are updated if the leaf receives a new instance. Once the differences between highest information gain and second highest information gain are greater than a confidence threshold ϵ in a leaf, this node is split on the highest information gain's attribute.

VFDT uses Hoeffding Bound [15] to determine the threshold ϵ of the split decision. Hoeffding Bound has an attractive property as it is independent of the distribution of data.

By applying Hoeffding Bound, ϵ is calculated by:

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

In this formula, R is the range of the input random variable. For information gain, this can be obtained by $R = \log 2(Number\ of\ Classes)$ [10]. n is the number of instances seen. δ is a parameter smaller than 1.

Let $\Delta\bar{G}$ be the observed difference between the highest information gain obtained by splitting the data on attribute X_a and second highest information gain obtained by splitting the data on attribute X_b. Let ΔG be the same difference in the population. The algorithm splits a leaf on X_a if $\Delta\bar{G} > \epsilon$.

Domingos et al. [8] showed that Hoeffding Bound guarantees that a split is correct with probability $1 - \epsilon$ if $\Delta\bar{G} > \epsilon$. If $\Delta\bar{G} > \epsilon$, a split is made.

Algorithm 1 shows the pseudo-code of VFDT. The sufficient statistic n_{ijk} is the count of instances which have attribute i equal to j with class label k.

Algorithm 1: `VFDT`: The Hoeffding Tree Algorithm.

```
input  : $S$: Stream of examples
         $X$: Set of nominal Attributes
         $Y : Y = \{y_1, \ldots, y_k\}$ Set of class labels
         $H(.)$: Split evaluation function
         $N_m$: Minimum number of examples
         $\delta$: is 1 minus the desired probability
         of choosing the correct attribute at any node.
         $\tau$: Constant to solve ties.
output: $HT$: is a Decision Tree
1  begin
2      Let $HT \leftarrow$ Empty Leaf (Root);
3      foreach example $(x, y_k) \in S$ do
4          Traverse the tree $HT$ from the root till a leaf $l$;
5          if $y_k$ is unknown then
6              Classify the example with the majority class in the leaf $l$;
7          else
8              Update sufficient statistics $n_{ijk}$;
9              if Number of examples in $l > N_{\min}$ then
10                 Compute $H_l(X_i)$ for all the attributes;
11                 Let $X_a$ be the attribute with highest $H_l$;
12                 Let $X_b$ be the attribute with second highest $H_l$;
13                 Compute $\epsilon = \sqrt{\frac{R^2 \ln(2/\delta)}{2n}}$ (Hoeffding Bound);
14                 if $(H(X_a) - H(X_b) > \epsilon)$ then
15                     Replace $l$ with a splitting test based on attribute $X_a$;
16                     Add a new empty leaf for each branch of the split;
17                 else
18                     if $\epsilon < \tau$ then
19                         Replace $l$ with a splitting test based on attribute $X_a$;
20                         Add a new empty leaf for each branch of the split;
21     end for
22 end
```

2.2.3 Review Analysis

In this section, we haven shown both the traditional decision tree algorithm and the VFDT algorithm. By comparing traditional decision tree and VFDT, we can identify the main difference between these two is that instead of scanning all data to decide a potential split on the leaf to grow a decision tree, VFDT trains a decision tree by making the split decision when enough samples have been seen. This is motivated by the unlimited size of data streams, as it is not possible for the learning model to see all instances in the stream. VFDT provides inspiration to design stream mining algorithms based on existing traditional learning algorithms by incorporating a statistical approach to making estimations. The statistical approach can help to make reasonable decisions based on the limited size of samples from the population formed by the unlimited size of data. In the case of VFDT, it uses Hoeffding Bound to make reasonable splitting decisions on an unlimited stream.

2.3 Concept Drifts

A data stream may evolve over time. The causes of change can be various. One possible source of concept drift is a change in natural properties in the observing object [10]. For example, suppose we are monitoring an electricity network in an area, a natural disaster suddenly happens and causes serious damage to the local power station. A second source is the hidden variables [14]. Hidden variables are those variables that we either ignore or unable to observe. A change of these variables may also cause concept drift in a data stream.

Formally, Gama [10] defines a concept drift as the change in the joint probability $P(\vec{x}, y)$ where

$$P(\vec{x}, y) = P(y|\vec{x}) \times P(\vec{x})$$

where $\vec{x}$ is the attributes vector and y is the dependent variable in the data stream.

Concept drifts can be classified as *real* and *virtual* [12]. *Real concept drifts* occur when the underlying concepts of a stream change. For example, in the case of spam email classification in which the outputs are either spam or not spam. A real concept drifts occur when a spammer finds a new writing style to compose spam which is not seen before. *Virtual concept drifts* refer to the change where the underlying concepts do not change, but only the distribution of instances change. In the example of spam classification, a virtual concept drift can happen when the number of spam emails suddenly increases, but the writing style of spam is not changed. It is important to distinguish between these two types of drifts and treat them differently. From the predictive perspective, an occurrence of real concept drift often requires model adaptation to guarantee high prediction accuracy [12].

Concept drifts reduce the prediction accuracy of the current learning model. Thus, dealing with concept drifts is not trivial. Gama [10] describes a data stream as a sequence $\{S_1, S_2, \ldots, S_n\}$ in which each S_i is a sub-stream with a different distribution. The task of a drift detection algorithm is to detect the transition point between each sub-stream.

Drift detection can be a difficult task. Gama [10] identifies two elements of data streams that can make drift detection a challenging task.

1. Two consecutive sub-streams S_i and S_{i+1} may have a transition phase where some instances of both distributions appear mixed. So one sub-stream can appear as noise in the other consecutive sub-stream.

2. There is random noise in one distribution. It can be hard to differentiate the noise from the actual concept drifts.

Therefore, a capable drift detector should combine the robustness to random noise and the sensitiveness to the actual concept drift.

In the following review section, we will review three classes of methods to handle concept drifts: *Data Management*, *Drift Detection*, *Adaptation*, and *model adaptation*. We will also compare different approaches' advantages and disadvantages.

2.3.1 Data Management

In general, data management methods use the most relevant instances to maintain the prediction model. Out-dated instances learned by the prediction model are forgotten. Gama categorised data management methods into two classes: *Full Memory* and *Partial Memory*.

Full Memory is a category of data management methods. It stores all instances in memory and assigns larger weights to the recent instances than the old instances. The weight for each instance is expressed as a fading factor α and $0 < \alpha < 1$. α decays over times such that the old instances become less important in the prediction model. The functions of decay can be various. Koychev [20] proposed a linear decay method, in which the rate of decay is constant over time. Klinkenberg et al. [19] developed an exponential function for decay. In this method, $\omega(x)$ is the weight of the instance x received i time steps ago and $\omega(x) = exp(-\lambda i)$.

Partial Memory is the data management method that stores only recent instances to maintain the prediction model. Many implementations in this category use sliding windows to store recent data instances. The simple sliding window is a fixed size window in which a user specifies a size l for the window, and only the most recent l instances are used for maintaining the prediction model. A more complex sliding window has an adaptive size. An adaptive window method normally works with other drift detectors, and it shrinks when it detects a drift and increases in size in the period without any drifts. Partial memory methods have benefits over full memory methods because they use less memory, but their sliding windows may fail to capture all useful information from the stream.

2.3.2 Drift Detection

Drift detection methods signal an alarm if they detect an occurrence of any drift. The prediction model responds accordingly, once it receives the alarm.

The Cumulative Sum Algorithm (CUSUM)

CUSUM is a change detection algorithm proposed by Page [22]. The intuition of CUSUM is that it signals a drift alarm if the instances in the data stream are significantly different from zero.

The CUSUM algorithm can be expressed by the following formulas:

$$
\begin{aligned}
&S_0 = 0 \\
&S_t = max(0, S_{t-1} + (x_t - v)
\end{aligned}
$$

where x_t is the instance received at time t. The user specifies two parameters: v and λ. λ is the threshold value. CUSUM signals a drift alarm at time t if $S_t > \lambda$. v is a sensitivity parameter. A larger v will bring faster detection, but more false positive alarms may appear. CUSUM's advantage is that it is simple to implement, and it is memoryless. CUSUM only maintains a cumulative value in the memory. However, it requires two input parameters, thus requiring users to find the best parameter choice.

Page-Hinkley Test (PH Test)

PH Test is a variant of CUSUM [22]. PH test computes a PH value $PH_T = m_T - M_T$. It signals a drift alarm if $PH_T > \lambda$ where λ is a threshold parameter. m_T is a cumulative variable. This value represents the cumulated difference between the current observed values and the mean of a data stream. m_T can be calculated by the following formulas:

$$
\begin{aligned}
\bar{x}_T &= \frac{1}{T}\sum_{t=1}^{T} x_t \\
m_T &= \sum_{t=1}^{T}(x_t - \bar{x}_T - \delta)
\end{aligned}
$$

In the above formulas, δ is another parameter that represents the allowed magnitude of changes.

M_T is the minimum value among all $m_1, \ldots, m_T$.

Compared with CUSUM, PH test is more efficient and effective at detecting abrupt changes in the average of a Gaussian signal [21].

Drift Detection Method (DDM)

DDM is a binary stream drift detection algorithm proposed by Gama et al. [11]. DDM is used along with a prediction model. The correctness of prediction can be treated as a binary stream. For example, we use 0 to denote *Correct Prediction* and 1 to denote *Wrong Prediction*. As a result, when we train a model with a data stream, we can obtain a binary stream representing its prediction performance. A significantly increasing error in the binary streams shows a drift in the training data. DDM uses this idea to detect drifts. It models classification errors as a binomial distribution. Let p_i be probability of wrong predictions at instance i. We can obtain its standard deviation as:

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

DDM has a two-phase alarm: warning alarm and drift alarm. When it signals the warning alarm, it creates a window to store all incoming instances since this alarm. When it signals a drift alarm after the warning, the old prediction model is dropped, and the new model is created using instances in the window.

The algorithm updates the minimal p_i and s_i. DDM signals a warning alarm when $p_i + s_i > p_{min} + \alpha s_{min}$. It signals the drift alarm when $p_i + s_i > p_{min} + \beta s_{min}$ where $\alpha < \beta$. In [11], the authors set α as 2 and β as 3 to approximate 95% and 99% confidence intervals around the mean.

DDM is very effective on abrupt drifts, but if the drift is relatively gradual, DDM may pass the drift without an alarm.

Early Drift Detection Method (EDDM)

EDDM is a variant of DDM proposed by Baena-Garcia et al. [3]. DDM takes the binary stream as input, where as EDDM takes distances between each point as input. Let p_i be the average distance between two error point seen at time i, and s_i be its standard deviation. We also store the maximal p and s, denoted as p_{max} and s_{max}. We set warning threshold α and drift threshold β. We then can compute two conditions by:

$$\frac{p_i + 2s_i}{p_{max} + s_{max}} < \alpha \tag{2.1}$$

$$\frac{p_i + 3s_i}{p_{max} + s_{max}} < \beta \tag{2.2}$$

Like DDM, if the equation 2.1 is not satisfied, the algorithm enters the warning state and starts to store incoming instances. Then if condition 2.2 is not satisfied, it rebuilds the learning model with the stored instances.

EDDM can handle gradual drift better than DDM while maintaining its performance for abrupt drift. However, the drawback of EDDM is that it is sensitive to random noise.

Adaptive Windowing (ADWIN)

Bifet et al. [4] proposed a window-based algorithm called ADWIN to detect changes.

ADWIN maintains a buffer window W. The input of the algorithm is a stream of integers. For each new instance, it inserts the instance to the head of the window and drops instances from the window tail. Every time it drops an item, it splits the W into two parts in all possible ways and computes the difference between $\hat{\mu_{W_0}}$ and $\hat{\mu_{W_1}}$ which are the means of two sub-windows. If the difference is greater than the threshold ϵ_{cut} computed in any possible split, it means there are significant differences in the stream, and the algorithm signals that there is a drift. Algorithm 12 shows the pseudo-code of ADWIN.

Unlike the other algorithms discussed earlier, threshold ϵ_{cut} is not set by the user but computed using the Hoeffding Bound theorem [15] in real time. Bifet et al. [4] constructed the following formula to compute ϵ_{cut} by :

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

$$\delta' = \delta/n$$

$$m = \frac{1}{1/n_0 + 1/n_1} \text{ (harmonic mean of } n_0 \text{ and } n_1\text{)}$$

where δ is a bound parameter given by the user and $0 < \delta < 1$. n_0 and n_1 are sizes of two split sub-windows. Bifet et al. [4] proved two theorems in their paper:

1. (False positive rate bound). If μ_t remains constant within W , the probability that ADWIN shrinks the window at time t is smaller or equal to δ.

2. (False negative rate bound). If some partition of W is in two parts W_0W_1 where W_1 contains the most recent items, and we have $|\mu_{W_0} - \mu_{W_1}| > 2\epsilon_{cut}$, then ADWIN shrinks W to W_1 or shorter with probability $1 - \delta$.

ADWIN can be used with machine learning algorithms. Like DDM, we output a 0 when the prediction model makes a correct prediction and a 1 when it makes a wrong prediction. As a consequence, a stream of binary numbers is generated. We input the stream into ADWIN, and it can decide if there are changes in the distribution of the binary stream. A change in the binary stream signals that the learning algorithm has experienced a concept change in the data stream, and relevant adaptations (e.g., rebuild the model) should be taken.

There are two main benefits of using ADWIN. Firstly, ADWIN requires the user to set only one parameter δ. Secondly, ADWIN provides a proven guarantee of its performance on the false positive rate and false negative rate given the parameter δ. The drawbacks of

Algorithm 2: `ADWIN`: Adaptive Windowing

```
   input : $S$: Stream
1  begin
2      Instantiate window $W$;
3      foreach instance $x_t \in S$ do
4          $W \leftarrow W \cup \{x_t\}$;
5          repeat
6              Drop elements from the tail of $W$
7          until $|\hat{\mu_{W_0}} - \hat{\mu_{W_1}}| < \epsilon_{cut}$ holds for every split of $W = W_0 \cdot W_1$;
8          Output $\hat{\mu_W}$;
9          if $W$ has shrunk then
10             Signal the drift;
11     end for
12 end
```

ADWIN are its larger time and memory complexity compared with the other algorithms discussed earlier. ADWIN's complexity analysis can be viewed in [4].

ADWIN is the drift detector that will be applied in this research project.

VFDT With ADWIN Drift Detector (VFDT-ADWIN)

We discuss a usage example of the drift detector on a learning algorithm. VFDT-ADWIN is an example of applying VFDT in the stream with a drift detector. VFDT-ADWIN has an ADWIN drift detector on top of VFDT. If the VFDT correctly classifies an instance, it inputs 0 into the drift detector, and if the classification is incorrect, it inputs 1. As a result, the drift detector reads a sequence of binaries denoting the prediction errors. Once the drift detector signals a drift in the error stream and the error is increasing, the drift detector sends a signal to the VFDT. At this point, VFDT prunes the whole tree and grows a new tree. The algorithm's pseudocode can be seen in Algorithm 3. In line 12 to 15, we make the drift detector signal a rebuilding command if the error is increasing. The tree is not rebuilt if the drift detector finds a drift in the errors stream and the error is decreasing.

Algorithm 3: `VFDT-ADWIN`: VFDT with the ADWIN drift detector

```
input : $S$: Stream of examples
begin
    Start VFDT;
    Start ADWIN;
    foreach example $(x, y_k) \in S$ do
        Let VFDT classify $(x, y_k)$;
        Let $Y$ be the predicted class of $(x, y_k)$ ;
        if $Y == y_k$ then
            input 0 into ADWIN;
        else
            input 1 into ADWIN;
        if ADWIN detects a drift then
            Let $a$ = ADWIN's entries' mean before the drift ;
            Let $b$ = ADWIN's entries' mean after the drift ;
            if $a < b$ then
                Rebuild VFDT;
    end for
end
```

2.3.3 Model Adaptation

Model adaptation methods adapt the learner to the new concept when drift appears in the stream. For example, adaptive decision tree algorithm can rebuild part of the decision tree to adapt to the new concept.

CVFDT

One implementation of an adaptive algorithm is CVFDT [18]. CVFDT is a variant of VFDT shown in Algorithm 1. The main difference is that CVFDT maintains a sliding window storing recent instances learned by the decision tree. For each new instance added on the head of the window, the oldest instance from the window's tails is dropped. The CVFDT adjusts the attribute-class statistics n_{ijk} on each node by decrementing the old instance count and incrementing the new instance count. During this learning and forgetting process, CVFDT grows an alternative sub-tree at a node if it finds the confidence that the splitting test on the node is inappropriate, and switches the sub-tree rooted at this node to the alternative sub-tree once the alternative sub-tree produces more accurate predictions than the old one.

Hoeffding Adaptive Tree using ADWIN (HAT-ADWIN)

Hoeffding Adaptive Tree using ADWIN (HAT-ADWIN) is proposed by Bifet et al. [5]. This algorithm combines the drift detector ADWIN with a VFDT decision tree. HAT-

ADWIN installs ADWIN drift detector on each node. The ADWIN drift detector monitors the attribute-class statistics on its host node. If a change in the attribute-class statistics at that node is detected, it starts to grow an alternative tree rooted at that node. Each incoming instance is sorted through this node are also sorted into the alternative tree. HAT-ADWIN constantly compares the prediction accuracies between the node's original subtree and the alternative sub-tree, once the alternative sub-tree has a higher accuracy, it drops the original subtree and places it into the alternative tree. Algorithms 4 and 5 show the pseudo-code of HAT-ADWIN. Note that in Algorithm 5 line 6, Bifet et al. [5] use a slightly different Hoeffding bound than VFDT. It actually uses δ/n to replace the original δ in VFDT's Hoeffding bound. This change intends to handle the fact that many tests are performed, and authors want all of them to be simultaneously correct with probability $1-\delta$. This process is the *Bonferroni correction* discussed in [8].

Bifet et al. [5] proposed two other HAT variants. Instead of installing the ADWIN drift detector at nodes, HAT can also place a linear incremental estimator and Exponential Weight Moving Average (EWMA) estimator on each node that take the input of attribute-class statistics. The statistics on each node is replaced by those estimators' values.

Algorithm 4: HAT-ADWIN: Hoeffding Adaptive Tree using ADWIN

```
1  Let $HT$ be a tree with a single leaf (root);
2  Init estimators $ADWIN$ at root;
3  foreach $(x, y)$ in Stream do
4      HATTreeGrow $((x, y), HT, \delta)$;
5  end for
```

Algorithm 5: HATTreeGrow

```
1  Sort $(x, y)$ to leaf $l$ using $HT$;
2  Update statistics and drift detectors $ADWIN$ at nodes traversed in the sort;
3  if this node has an alternate tree $T_{alt}$ then
4      HATTreeGrow$((x, y), T_{alt}, \delta)$
5  Compute G for each attribute;
6  if $H(Best\ Attr.)-H(2nd\ best) > \sqrt{\frac{R^2 \ln(2n/\delta)}{2n}}$ then
7      Split leaf on best attribute;
8      foreach new branch do
9          Start new leaf and initialise estimators
10     end for
11 if one change detector has detected change then
12     Create an alternate subtree if there is none
13 if existing alternate tree is more accurate then
14     replace current node with alternate tree
```

Analysis

HAT is better than CVFDT because it can produce equal or even better prediction accuracy than CVFDT, and it does not require the user to specify the sliding window length. However, HAT-ADWIN has longer training time than CVFDT.

2.3.4 Recurring Concepts

Concept drifts are often encountered in real-world data streams. However, in these real-world cases, most concepts between each drift are recurrent [14]. This means that after experiencing a concept drift, the arriving concept may have been previously seen. To handle this case, one can store a pool of models to represent several concepts in a stream. When a new concept appears, it checks whether the new concept has been seen previously in an existing model. If the concept has been seen in an existing model, it uses that model to learn and predict on the incoming stream. Consequently, we do not need to experience the significant accuracy drop and re-train a new model when a concept drift occurs. Thus, the benefit of using such methods is to maintain the high prediction quality in streams with recurring concepts.

The Recurring Concept Drift (RCD) framework [13] is an approach for handling recurring concepts. RCD system maintains a pool of classifiers with a drift detector. Each classifier has a buffer containing the sample of instances trained by the classifier. The sample of each classifier represents its learned concept. One classifier is the active classifier learning from the stream. When a drift is detected, RCD uses the statistical approach to compare recently seen instances and all existing buffers of samples in the classifiers' pool. If any classifier's buffer is similar to the recently seen instances, it decides that this existing classifier has already learned the current concept, it then uses that classifier as the active classifier and drops the existing one. If no classifier's buffer is similar to the recent instances, it adds the currently active classifier to the pool.

RCD system has similar aspects to our proposed system (VACS) because they both use more than one learner to mine data streams. However, these two methods are motivated by different purposes. We will compare and contrast between RCD and our proposed system (VACS) in Chapter 6.

2.3.5 Data Stream Volatility

Data stream volatility is a new technique presented by Huang et al. [17]. Volatility describes the frequency of detected drifts in a stream. A stream is considered to have high volatility if its changes are detected more frequently, and have low volatility if the changes are detected infrequently.

The change of the volatility in a stream is known as volatility shift. Huang et al. [17] presented an algorithm to detect the volatility shift in a data stream. The method looks at the relative fluctuation of volatility levels in a stream with respect to the stream itself. The input of the volatility shift detector is a sequence of real values $\{p_1, \ldots, p_t\}$ representing the distance intervals between each drift point in the stream. When a new drift is detected, the algorithm can compute a new interval value by subtracting the position of previous drift points from the position of the newly detected drift. The volatility shift detector maintains a buffer and a reservoir shown in Figure 2.1. The buffer is a sliding window storing the most recent interval values, and the reservoir is a data structure implementing the reservoir sampling [25]. After each new interval value is generated, the algorithm stores the value into the buffer. If the buffer is full, it slides and removes one item from the window's tail. The removed item is inserted into the reservoir. If the reservoir is full, it randomly picks an item from the reservoir and removes it to create the space for the new item. In this design, the buffer represents the recent volatility level of the stream, and the reservoir represents the overall volatility level of the stream. Once the buffer and the reservoir are significantly different, the algorithm can conclude that there is a volatility shift detected in the stream. The significant difference is determined by relative variance calculated by $\frac{\sigma_B}{\sigma_R}$ where σ_B is the variance of samples in the buffer and σ_R is the variance of samples in the reservoir. Once $\frac{\sigma_B}{\sigma_R} > 1.0 + \beta$ or $\frac{\sigma_B}{\sigma_R} < 1.0 - \beta$ where β is a confidence threshold and $0 < \beta < 1$, it alarms that there is a significant difference between the reservoir and the buffer.

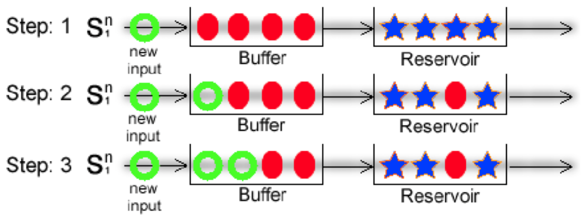

Figure 2.1: Volatility Detector Example (cited from [17])

Volatility is a novel concept in the data stream mining field. It provides a new dimension of information of the stream. Such information can be used to optimise the performance of data stream mining algorithms. One application of volatility is to predict the future drift positions in a stream in order to improve the performance of the drift detector. After introducing the concept of data stream volatility, Huang et al. [16]

presented a method to predict the expected probability of future drifts] using volatility information. The predicted probability is used to adjust the ADWIN drift detector in real time. When the expected probability of experiencing a drift is high, ADWIN is tuned to be more sensitive. When that probability is low, ADWIN is adjusted to be less sensitive. As a result, it reduces the false alarm rate of the ADWIN drift detector.

Volatility is a fundamental concept in our research. We explore the potential of using stream volatility to select learners in order to save the overall time and memory cost. Details are covered in later chapters.

2.4 Adaptive And Non-Adaptive Learners

In this section, we discuss two categories of data stream learning algorithms: adaptive learner and non-adaptive learner.

VFDT is considered to be a non-adaptive algorithm. Non-adaptive algorithms do not have the ability to adapt their model to new concepts in an evolving data stream. Instead, with the help of the drift detector such as ADWIN, a non-adaptive algorithm can react to concept drift by rebuilding its model. However, rebuilding the whole model might not be necessary, because when concept drift occurs, the old model can still guarantee some prediction accuracy because not all attributes in the stream may experience concept drifts. For example, if the stream has 10 attributes and only 3 attributes evolve, most nodes of the old decision tree before the concept drift can still be reused after the drift. Therefore, the significant drawback of rebuilding a model is losing information learned from the old stream which may still be useful in the future, and the rebuilding may result in a dramatic drop in the prediction accuracy.

In contrast, HAT-ADWIN and CVFDT are adaptive algorithms. Adaptive algorithms are able to partially update their models to fit the new concept in an evolving stream. One disadvantage of using adaptive algorithms is that these algorithms have larger overhead than non-adaptive algorithms in terms of time and memory. This is because adaptive algorithms need additional computation and storage to perform model adaptation. For example, HAT-ADWIN needs to update several drift detectors with the additional time cost. Also, HAT-ADWIN can have alternative trees which consume more memory. We show these overhead differences through experiments in later sections. Adaptive learners can be characterised into *Blind Methods* and *Informed Methods* [10]. Blind methods continuously adapt to the current concept and forget the out-dated instances without being aware of the concept drift. Blind methods can be implemented with weighted examples or sliding windows. Informed methods come with a drift detector such as ADWIN. It only takes the model adaptation when a concept drift is detected.

Later, our proposed framework, known as VACS, combines advantages of both types of learners, so we can maintain a high prediction accuracy while consuming relatively low cost.

3 Methodology

In this chapter, we propose the Volatility Adaptive Classifier System (VACS). The intuition of VACS is to apply different classifiers in a stream with changing volatility. It aims to reduce learning overheads while maintaining high enough prediction quality. VACS is composed of several modules: Volatility Measurement Window, Double Reservoirs Classifier Selector and two component learners. We will explain each module in more detail in this chapter.

3.1 Key Assumptions

We make some key assumptions in this research.

Assumption on users' expectations

Whether a system can expectably perform its mining task depends on different cases. Normally there is a trade-off relationship between model quality and its maintenance cost (training time and memory usage). One reasonable decision might be maintaining enough prediction accuracy. Once the model quality is good enough, we stop improving the model but focus on minimising the cost as much as we can. Based on this, we make the first key assumption on users' expectations: the user expects to use adaptive learners when drifts occur more frequently to maintain a high enough overall prediction accuracy. The user expects to use non-adaptive learners when drifts are less frequent to save time and memory usage.

Assumption on deciding high and low volatilities

To determine a volatility level to be "high"" or "low" can also be based on different criteria. One example might be setting a threshold. Whenever the number of drifts occurs in a fixed amount of time is greater than the threshold, then we treat the current volatility as high. Otherwise, we treat it as low. This approach may not be optimal because it is hard for users to set a proper threshold, and a fixed threshold value may not always be appropriate in evolving streams. The second approach is to evaluate relative volatility difference in the stream: when the volatility level is on the higher tier among previous volatility levels in the current stream, we treat it as high. Otherwise, we treat it as low. This approach is better than the first one, because it does not need the user to decide a threshold, and the volatility threshold adapts with the stream. We make our second key assumption based on the second criterion: we assume that the volatility in a stream can be classified into high and low, and a volatility level is considered to be high by users if it is relatively high compared with previously measured volatility levels. Otherwise, the volatility is low.

Assumptions on classifiers

We assume some properties on adaptive classifiers (or high volatility classifier) and non-adaptive classifiers (or low volatility classifier). In our discussions in later chapters, we assume these properties hold true in most cases.

The adaptive classifier:

- produces equal or better prediction accuracy than non-adaptive classifier in the currently mined stream.
- has relative large memory and time overhead.

The non-adaptive classifier:

- produces equal or worse prediction accuracy than the adaptive classifier in the currently mined stream.
- has relatively small memory and time overhead.

We assume that HAT-ADWIN has desirable properties of the adaptive classifier given its discussion in [5].

VFDT with the ADWIN drift detector (VFDT-ADWIN) is assumed to have the non-adaptive classifier's properties. These properties are evident in our experiments.

3.2 Volatility Adaptive Classifier System Overview

We present Volatility Adaptive Classifier System (VACS) in this research. VACS is composed of various modules. In this section, we present an overview of VACS before discussing its details. Abstractly, the problem solved by VACS can be formulated as following:

Given A stream.

Given A classifier option for the high volatility period.

Given A classifier option for the low volatility period.

Output Install the appropriate classifier and learn from the stream.

Output Predict the class label of instances with unknown class.

Figure 3.1 presents an overview structure of VACS.

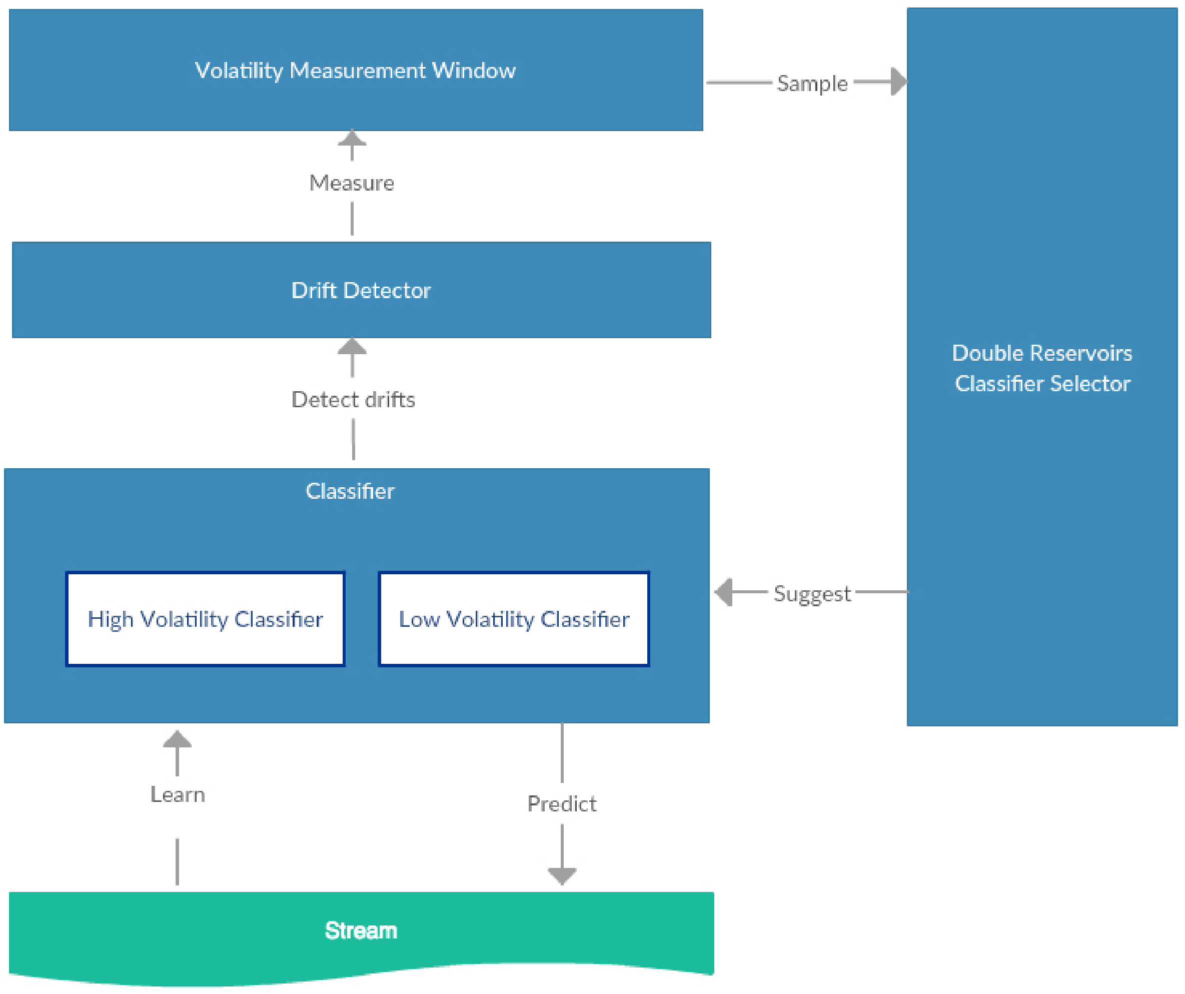
Volatility Measurement Window
Sample
Measure
Drift Detector
Detect drifts
Double Reservoirs Classifier Selector
Classifier
High Volatility Classifier
Low Volatility Classifier
Suggest
Learn
Predict
Stream

Figure 3.1: VACS Overview Structure

VACS learns a stream by using two classifiers. It switches between two classifiers given different levels of current stream volatility. In particular, we use VFDT-ADWIN as the non-adaptive classifier option used in the low volatility period, and it takes HAT-ADWIN as the adaptive classifier option used in the high volatility period. One task of VACS is to measure the volatility level of the stream and use the measurement as an input to decide which classifier is the appropriate choice. To measure the volatility of a stream, We developed a sliding volatility measurement window discussed in Section 3.3. Measurements of volatility level are sampled by the double reservoirs classifier selector (discussed in Section 3.4) which approximates the previous high and low volatility information separately in a stream. VACS decides whether to switch to the other classifier by querying for suggestions from the Double Reservoirs Classifier Selector.

3.3 Volatility Measurement Window

One task of VACS is to measure the volatility level of a stream such that our system can suggest the classifier option based on different volatility levels. Huang et al. [17] calculate volatility level by calculating interval lengths between each drift point in a fixed pool. In other words, their method measures the time differences among a fixed number of drifts in which small differences denote high volatility while large differences denote low. Their method is appropriate to calculate relative volatility shift in a stream. But it introduces a measurement delay in stream volatility because it needs to wait for the next drift in order to calculate the new volatility level. The delay problem can be severe if the volatility drops from high to low, because of the time difference between the next drift and the recent one increases, it needs to wait a long period for the next drift to appear in order to update the measurement. Our system is susceptible to delays because the system must make suggestions of classifier option based on the current volatility level.

We develop a new method to measure the level of volatility which calculates the level of volatility using a sliding window with fixed size T. Let γ be the number of drifts detected in the most recent T instances. Then γ can denote the level of the current volatility if T is fixed when learning from a stream. This new method mitigates the delay problem, because in the case when the level of volatility decreases, it does not need to wait until the next drift occurs, instead, it continually removes early drift counts from the sliding window, such that we can measure a volatility drop without seeing the next drift. We suggest a method of choosing T. See Section 5.3.4 for more details.

3.4 Double Reservoirs Classifier Selector

Double Reservoirs Classifier Selector (DRCS) is a module in VACS. The main task of the Double Reservoirs Classifier Selector (DRCS) is to sample and approximate both high and low levels of volatility measured in a stream while the learner is learning from the stream. We classify the volatility in a stream into "high" and "low" as two classes. We do not want to lose information about high volatility periods of a stream when sampling the low volatility. Similarly, we do not want to lose information about low volatility periods when sampling the high volatility. A single reservoir will not satisfy this requirement because the reservoir sampling removes a random element when inserting a new element. DRCS separately samples the volatility levels from low and high periods in a stream by using two independent reservoirs. Based on its approximated information about high and low volatility levels of a stream, it can suggest VACS to switch to the proper classifier.

DRCS has two functions: sampling and suggesting. The sampling function is called by VACS constantly. The sampling function of DRCS maintains two reservoirs named

High Reservoir and *Low Reservoir*. Those two reservoirs sample each input γ value using the reservoir sampling method [25]. The first γ is inserted into the low reservoir for initialisation. After initiation, when a new measured stream volatility γ arrives, it compares γ with the mean of the elements in the two reservoirs. If the value is lower than the mean, it stores this value into the *Low Reservoir*. If the value is greater than the mean, it stores the value into the *High Reservoir*. We specify a means' difference threshold λ. If the difference between two reservoirs' means is greater than λ, the reservoir's state is set to be active. VACS can only suggest to switch a classifier when DRCS is active. This setting can prevent two undesirable behaviours. Firstly, it prevents VACS from switching classifiers when there are rare volatility changes in the stream. Secondly, if there are volatility changes in the stream but these changes only appear in a later period, it prevents VACS from switching classifiers at the early stage in which a changing volatility has not been measured yet. Intuitively, λ is used to indicate the size of the volatility change that matters to the user, if there are volatility differences greater than this threshold in a stream, we can treat the stream as a volatility-changing stream and activate our system, otherwise, we use the single learner to handle the stream. λ is also related to the volatility measurement value T. A larger T value can result in larger measured numbers of drifts in the window. Thus, larger volatility values can be obtained. So λ should increase with T. However, if λ is overly large, VACS may never be activated. In our experiments, we choose 15 as our λ values with a $T = 300,000$. Through experiments, we found it works well in most of our experiments.

The algorithm pseudo-code is shown in Algorithm 6.

The second function of DRCS is suggesting the algorithm option. This function is called when VACS queries DRCS. When VACS queries DRCS, it inputs the current stream volatility level γ. DRCS compares the input volatility level with the mean of the elements in two reservoirs and suggests what classifiers should be used. DRCS can suggest two possible classifiers, namely high volatility classifier (the adaptive learner) and low volatility classifier (the non-adaptive learner). The pseudo-code is shown in Algorithm 7.

Algorithm 6: DRCS: Double Reservoir Classifier Selector Sampling

```
input : $\gamma$: The measured level of stream volatility
        $L$: The size of reservoirs
        $\lambda$: The means' difference threshold
begin
    Initialise the reservoir $LowReservoir$ with size $L$;
    Initialise the reservoir $HighReservoir$ with size $L$;
    Let $IsAcitve$ = False;
    foreach $\gamma$ do
        Let $s_{low}$ = Sum of elements in $LowReservoir$;
        Let $n_{low}$ = Number of elements in $LowReservoir$;
        Let $s_{high}$ = Sum of elements in $HighReservoir$;
        Let $n_{high}$ = Number of elements in $HighReservoir$;
        Let $n = n_{low} + n_{high}$;
        if $n = 0$ then
            Insert $\gamma$ into $LowReservoir$;
        else
            Let $\mu = \frac{s_{low}+s_{high}}{n}$;
            if $\gamma > \mu$ then
                Insert $\gamma$ into $HighReservoir$;
            else
                Insert $\gamma$ into $LowReservoir$;
            if $s_{high}/n_{high} - s_{low}/n_{low} > \lambda$ then
                Let $IsAcitve$ = True;
    end for
end
```

Algorithm 7: `DRCS`: Double Reservoir Classifier Selector Suggesting

```
input : $\gamma$: The measured level of stream volatility
begin
    if $n = 0$ or DRCS is inactive then
        Return null;
    else
        Let $s_{low}$ = Sum of elements in $LowReservoir$;
        Let $n_{low}$ = Number of elements in $LowReservoir$;
        Let $s_{high}$ = Sum of elements in $HighReservoir$;
        Let $n_{high}$ = Number of elements in $HighReservoir$;
        Let $\mu = \frac{s_{low}+s_{high}}{n}$;
        if $\gamma > \mu$ then
            Suggest the high volatility classifier;
        else
            Suggest the low volatility classifier;
end
```

3.5 Volatility Adaptive Classifiers System

In this section, we compose each module discussed in the previous sections into the complete Volatility Adaptive Classifiers System (VACS).

3.5.1 Algorithm

The algorithm firstly initiates the Double Reservoirs Classifier Selector (DRCS) and a drift detector. By default, we use ADWIN for the drift detector. It also initiates the volatility measurement window counting the number of recently detected drifts. We provide two classifiers for VACS satisfying our assumptions. One classifier is considered to be suitable in high volatility periods (adaptive classifier) while the other is deemed to be appropriate in low volatility periods (non-adaptive classifier). In VACS, only one classifier is active at any time to perform the learning task. The user decides which classifier should be active at the start when no volatility level has been detected. When the algorithm starts, it takes each arriving instance from the stream and classifies it with the active classifier. If the classifier correctly classifies the instance, we input 0 into the drift detector. Otherwise, we input 1 into the drift detector. The drift detector is modified such that it signals a drift only if the prediction error is increasing. Next, it updates the measurement γ (volatility level) which is the number of drifts detected in the sliding window. If the number of classified instances since the last volatility level measurement reaches a user-specified count τ, the algorithm measures and inputs γ into DRCS and then queries DRCS. The reason for adding an interval between two volatility level measurements is because it is not likely to measure the volatility level γ change if two measurements are close. If DRCS is active, it returns one classifier option from the two that is suggested to be used in the current level of volatility. If this classifier is not consistent with the one active in VACS, it switches the currently active classifier to the suggested one. Moreover, the algorithm re-initiates the current active classifier to its starting state. In the case of a decision tree, it resets the decision tree to a one-node tree without learning examples. The pseudo-code can be seen in Algorithm 8.

When the user wants to make a classification with an instance with the unknown class, VACS will use the currently active classifier to make the prediction.

Algorithm 8: VACS: Volatility Adaptive Classifiers System

```
input : $S$: Stream of examples
        $\tau$: Interval length between each volatility level measurement.
        $T$: Size of sliding window for measuring current volatility level.
        $HighVolClassifier$: The classifier used in high volatility period.
        $LowVolClassifier$: The classifier used in low volatility period.
        $StartingClassifier$: The classifier chosen at the start.
        $DT$: Drift Detector.
1  begin
2      Initiate drift detector $DT$;
3      Initiate Double Reservoir Classifier Selector $DRCS$;
4      Let $ActiveClassifier = StartingClassifier$;
5      Let $\gamma$ = Number of drifts detected when classifying the most $T$ instances;
6      foreach $example(x, y_k) \in S$ do
7          $ActiveClassifier$ classifies $(x, y_k)$;
8          if $ActiveClassifier$ correctly classified $(x, y_k)$ then
9              Let $e = 0$;
10         else
11             Let $e = 1$;
12         Input $e$ into $DT$;
13         Let $i$ = number of instances that has been classified;
14         if $i\%\tau = 0$ then
15             Input $\gamma$ into $DRCS$;
16             Query $DRCS$ to get the suggested classifier;
17             Let $SuggestedClassifier$ = Suggested Classifier provided by $DRCS$;
18             if $SuggestedClassifier$ is not null AND is not $CurrentClassifier$
                 then
19                 if $SuggestedClassifier$ is $HighVolClassifier$ then
20                     Set $ActiveClassifier = HighVolClassifier$;
21                 else
22                     Set $ActiveClassifier = LowVolClassifier$;
23                 Re-initiate $ActiveClassifier$;
24         Train $ActiveClassifier$ with $(x, y_k)$;
25     end for
26 end
```

3.5.2 Algorithm Parameters

In this subsection, we summarise all parameters required to be input by the user.

Size of reservoirs (L)

This parameter specifies the size of reservoirs used in the Double Reservoir Classifier Selector (DRCS). The size applies to both reservoirs, so two reservoirs have the same size.

Means' difference threshold (λ)

This parameter specifies the minimal change of volatility level that the user decides to be significant enough. The Double Reservoir Classifier Selector (DRCS) can only be active when the two reservoirs' means are greater than this parameter. Otherwise, DRCS does not provide suggestions when being queried.

Interval length between each volatility level measurement (τ)

This parameter defines the number of instances between each volatility level measurement. Also, it defines the number of instances between each input and query to DRCS.

Size of sliding window for measuring current volatility level (T)

Volatility level is computed as the number of drifts detected in recent classified instances in a sliding window. This value specifies the size of the sliding window.

Drift detector

VACS can take any drift detector that handles binary data. By default, we use ADWIN in our VACS.

High volatility classifier

It is the suggested classifier used during high volatility periods in the stream.

Low volatility classifier

It is the suggested classifier used during low volatility periods in the stream.

Starting classifier

It is the classifier used at the start of the stream when no volatility information has been measured. The user chooses one of the *High volatility classifier* and the *Low volatility classifier* as the starting classifier.

4
Experiments Design

In this chapter, we discuss experiment designs to evaluate VACS. We implemented VACS in the MOA framework [6]. The experiments compare VACS with VFDT-ADWIN and Hoeffding Adaptive Tree with ADWIN (HAT-ADWIN) with respect to prediction accuracy, total training time cost, and model memory cost. We also designed specific experiments to explore whether VACS is able to apply the appropriate classifier on volatility-changing streams effectively. We also discuss implementations of our synthetic data generators used in our experiments.

4.1 Implementation Environment

We implemented VACS in the Massive Online Analysis (MOA) Framework [6]. MOA is an open source JAVA framework for data stream mining. It includes a collection of stream learning algorithms along with tools for evaluating stream learning algorithms. MOA can be accessed from the MOA website[1].

We added VACS as a classifier class in MOA. We also modified some existing MOA classes including EvaluationPrequential class and RandomTreeGenerator class to serve our experiments.

Our sources developed in MOA are available at the Github page[2].

MOA's ADWIN algorithm is an optimised version of the original ADWIN, known as ADWIN2. This implementation uses a variation of the exponential histogram method [7] to approximate values in the sliding window. ADWIN2 is more time and memory efficient than the original ADWIN. We use MOA's ADWIN2 in our implementations where the drift detector is needed. More information about ADWIN2 implementation and experimentation can be accessed at [4] and the MOA Reference[3].

4.2 Testing Environment

Table 4.1 shows the testing environment of our experiments.

Table 4.1: Testing Environment

Operating System	Windows 7
Memory	8.00 GB
CPU	Intel i5-3570 @ 3.40GHz

[1]http://moa.cms.waikato.ac.nz/
[2]https://github.com/burnmg/volatility-adaptive-classifier
[3]http://www.cs.waikato.ac.nz/ abifet/MOA/API/

4.3 Evaluation Measurements

In our experiments, measure the performance of HAT-ADWIN, VFDT-ADWIN and VACS by evaluating total time cost, mean memory cost, maximal memory cost, the mean prediction accuracy and the mean prediction accuracy in the drifting period on each algorithm. We compare the measurement results of those algorithms and contrast the differences among them. Beyond those measurements, we also explore whether VACS can apply and switch between expected classifiers in volatility-changing streams.

Total Training Time Usage

We measure the total training time usage of an algorithm by accumulating its time taken for training each instance in a data stream. This measurement shows the training speed of the algorithm. As discussed in Chapter 2, speed is an important concern when applying learning algorithms in the context of data streams, because a data stream can flow at high-speed and the algorithm is expected to process the stream at an equivalent pace.

Average & Maximal Memory Usage

Memory usage is the bytes of memory an algorithm needs to occupy during its usage. We measure two types of memory usage: the average memory usage and the maximal memory usage.

When testing a classifier with a stream dataset, we measure the memory usage of the classifier after it is trained with every 1000 instances until all instances in the dataset are trained. As a result, we get a sequence of memory usage measurements for each stream. Then we obtain the average memory usage by taking the mean value of the sequence, and we obtain the maximal memory usage by taking the maximal value in the sequence.

The average memory usage shows the overall memory usage of an algorithm when it is trained with the dataset. Additionally, maximal memory usage is also a critical measurement because it shows how much free memory the machine should at least assign in order to guarantee the algorithm's availability. A large maximal memory usage of an algorithm may warn the user to prepare the machine with the large available memory in order to stably run the algorithm.

Average Prediction Accuracy

The prediction accuracy is a percentage showing how accurately the algorithm can classify an instance with unknown classes. We use MOA's built-in evaluation method known as Evaluate Prequential. This approach uses the same dataset for both training and testing. For each instance in the stream dataset, it inputs the classifier with the instance and checks whether it is correctly classified. After the checking, the instance is trained in the classifier. Therefore, this approach guarantees that the testing instance is not seen and learned by the classifier. We measure the prediction accuracy as the percentage of correctly classified instances in a sliding window with size 1000. For each 1000 instances in a stream dataset, we take one measurement. As a result, we obtain a sequence of correct classification percentages. We then take the mean value of this sequence as the average prediction accuracy of the tested algorithm on a dataset.

Average Prediction Accuracy in Drifting Periods

The performance of learning algorithms when experiencing drift is highly related to its reliability. Therefore, we measure another accuracy known as the Average Prediction Accuracy in Drifting Period.

Average Prediction Accuracy in Drifting Periods is similar to the average prediction accuracy, but this accuracy is only measured from the classifications of instances near the drift position in a stream. Specifically, it only measures the prediction accuracy of 20,000 instances after each drift point. Like mean prediction accuracy, we obtain a sequence of percentage values, and we take the mean value of those percentage values. This measurement is crucial because the stream learning algorithms' performances can be easily affected by concept drifts, and this measurement can tell us whether one learning algorithm can be more viable than others when drifts occur.

Percentage of Instances Classified by Expected Classifier (PICEC)

Percentage of Instances Classified by the Expected Classifier (PICEC) is a special measurement designed only for VACS. Our synthetic datasets' volatility varies between low and high levels of volatility constantly. VACS has two classifier options: high volatility classifier (adaptive) and low volatility classifier (non-adaptive). We obtain PICEC using the following calculation: we count the number of instances from the high volatility period in a stream classified by the high volatility classifier, and the number of instances from the low volatility period in a stream classified by low volatility classifier. We divide the sum of these two numbers by the total number of instances in the streaming dataset. The result is the PICEC.

$$h = count(instanceInHighVolPeriod \cap instanceClassifierByHighVolClassifier)$$
$$l = count(instanceInLowVolPeriod \cap instanceClassifierByLowVolClassifier)$$
$$PICEC = \frac{h + l}{count(allIntances)}$$

4.4 Synthetic Data Generator

In this section, we discuss synthetic data generators used in our experiments.

Implementations of generators covered in this section are available on our Github page[4].

4.4.1 Mutating Random Tree Generator

We developed a new data generator based on the random tree data generator. The aim of this generator is to simulate drifting streams where not all attributes may experience concept drifts. The adaptive learner (HAT-ADWIN) is expected to have a better prediction accuracy than the non-adaptive learner (VFDT-ADWIN) in such data streams because the adaptive learner only needs to adapt part of its model (sub-trees) where drifts are detected.

Random tree generator is a streaming data generator originally proposed by Domingos et al. [8]. This generator is implemented in the MOA framework [6].

Random tree generator randomly builds a decision tree model with user-specified parameters including the number of attributes and maximal depth of the tree. It then randomly generates data instances without any class labels. The algorithm then lets the decision tree model classify these instances, and assigns the model predicted class label as the instance's actual class.

When a drift occurs, not all attributes' distributions in the data stream may change. The number of attributes experiencing drift is not predictable. In order to simulate a drift in which distributions of unknown numbers of attributes change, we present the mutating random tree generator. What makes it different from the original random tree generator is that it can mutate a sub-tree in the tree model by recursion if we want to create a drift in the synthetic data stream. We introduce a value called mutating probability p. The recursive mutation process works like this: we firstly input a node from the decision tree, the node has probability p to rebuild the sub-tree rooted at this node. If the sub-tree is rebuilt, the algorithm ends and outputs the mutated decision tree. If the node is not rebuilt, it randomly picks a child and recursively applies the same process to this child with the mutating probability p. If the chosen child is a leaf, the algorithm ends and outputs the decision tree without mutating.

To add a mutation to a tree model, we apply this recursive process on the root of the tree. The output decision tree is nondeterministic. The first possible outcome is that the output tree may not be changed compared with the input tree if the algorithm ends because a leaf is chosen as the input node of the recursive step. The second possible outcome is that the tree is totally rebuilt if the root is chosen as the rebuilding node. The

[4]https://github.com/burnmg/volatility-adaptive-classifier

last possible outcome is the case in the middle of the above two: a partially rebuilt tree is returned as the outcome. As a result, we are able to synthesise a data stream with nondeterministic numbers of drifting attributes.

4.4.2 SEA Generator

The second synthetic data generator used in our experiments is the SEA generator. The SEA generator was originally introduced in [24] and is a built-in generator in the MOA framework [6].

The SEA generator generates data instances with three attributes. Only the first two attributes are relevant to its class label. The class label of each instance is determined by $f_1 + f_2 \leq \theta$ where θ is a threshold value. f_1, f_2 are the first two relevant attributes. There are four default thresholds available in MOA: 7, 8, 9, and 9.5.

To create a drift, we randomly pick a different threshold and generate incoming instances using the new threshold.

4.4.3 Multiple Concept Drifts Stream Generator

The MOA framework [6] has an existing concept drift stream generator adding drifts to any stream. Its generator creates a concept drift event as a weighted combination of two streams with different distributions. The generator takes two streams, say A and B. The new data instance can either be drawn from A or B which is determined by their weights. The stream with the larger weight has a larger probability of being used in the next draw. As time goes on, the weight of A is decreasing while the weight of B is increasing, such that the generator produces a stream with a distribution gradually evolving from A to B.

The weights of A and B can be calculated by the following formulas:

$$Weight_A(t) = e^{-4(t-t_0)/W}/(1 + e^{-4(t-t_0)/W})$$
$$Weight_B(t) = 1/(1 + e^{-4(t-t_0)/W})$$

The formulas are based on the sigmoid function. t is the position of the next generated instance. t_0 is the position of change. W is the length of change. Note that if W is set to a small value, this generator can also generate an abrupt change without interleaved evolution between two streams A and B.

This generator can produce a stream with one concept drift. We compose multiple such concept drift stream generators and develop a multiple concept drift stream generator which is able to produce multiple drifts in a stream. The general idea is that we let the generator keep two active streams A and B, and we draw instances from those two streams like the original concept drift stream generator. The difference is that when the drifting stream has mostly evolved from A to B, we replace B with A and assign a new stream to B. We repeat this process and a stream with multiple concept drifts is obtained.

In particular, the input to the multiple concept drifts generator is a list of streams, say $\{S_1, \ldots, S_n\}$, and we want to output a stream with $n-1$ drifts by combining these streams. We also specify the total length of the expected generated stream, and we calculate a list of drift positions $\{p_1, \ldots, p_{n-1}\}$ with equal distances from each other. To start the multiple concept drifts stream generator, let stream A be S_1 and let B be S_2 in the single concept drift generator. Let p_1 be the position of drift between S_1 and S_2, and we input p_1 as the drift point in the single concept drift stream generator. Let t be the position of the next generated instance. We then run this single concept drift stream generator. When the t is closer to p_2 than p_1, we assign S_2 to A and assign S_3 to B in this single concept drift stream generator, and let p_2 be the drift position. We then continue this incremental process until all streams in $\{S_1, \ldots, S_n\}$ are applied.

The multiple concept drifts stream generator is used with other stream generators. For example, we can use a mutating random tree generator in the multiple concept drift stream generator. In this case, $\{S_1, \ldots, S_n\}$ are streams generated by the mutating random tree generator, each S_{i+1} is obtained by mutating the previous stream S_i.

4.4.4 Volatility-Changing Stream Generator

We develop a volatility-changing stream generator on the top of the multiple concept drifts stream generators. The aim of this generator is to synthesise a data stream with changing volatility. Stream volatility refers to the frequency of concept drifts, so a volatility-changing stream can by obtained by concatenating various multiple concept drift streams with different drift densities. For instance, we use multiple concept drift stream generators to generate two multiple concept drifts streams dS_1 and dS_2 with equal length l. We create d_1 drifts in dS_1 and d_2 drifts in dS_2 given $d_1 < d_2$. The stream $dS_1 \cdot dS_2$ obtained by concatenating dS_1 and dS_2 is a volatility-changing stream in which the volatility level is increasing (occurrences of concept drifts are getting more frequent as time goes on).

A volatility-changing stream generator follows the method described above to generate volatility-changing streams. The user specifies C as the count of sub-streams with equal lengths composing the output stream. Then, the number of instances of each sub-stream L is given. After that, the user inputs a list D with length C, denoting the number of concept drifts in each sub-stream respectively. The generator finally produces a volatility-changing stream with $C \times L$ instances.

We can show how the generator works through an example. If we want to generate a stream with increasing volatility. We can set $L = 10,000$ and $C = 3$. This means that we generate the stream by concatenating 3 sub-streams. Each sub-stream has 10,000 instances. Then we let D be $\{10, 20, 30\}$. This means that the first sub-stream has 10 drifts, the second one has 20 drifts, and the last one has 30 drifts. As a result, the output is a stream with 30,000 instances with increasing volatility. D can be understood as the volatility pattern that the volatility-changing stream follows.

4.5 Default Classifiers Parameters

In this subsection, we specify the general parameters for learners. All ADWIN drift detectors' δ value in VACS, HAT-ADWIN and VFDT-ADWIN is set to 0.002. This is the default ADWIN parameter used in the MOA framework [6].

VACS's parameters can be seen in Table 4.2.

Table 4.2: VACS Default Parameters Setting

Parameter Name	**Parameter Value**
Size of reservoirs (L)	200
Size of sliding window for measuring current volatility levels (T)	300000
Means' difference threshold (λ)	15
Interval length between each volatility level measurement (τ)	10000
Drift detector	ADWIN
High volatility classifier	HAT-ADWIN
Low volatility classifier	VFDT-ADWIN
Starting classifier	VFDT-ADWIN

All experiments share these parameters unless specified differently.

5 Results

In this chapter, we present the results of our experiments along with interpretations.

We compare prediction quality, time and memory among VACS, VFDT-ADWIN and HAT-ADWIN on various datasets. We then explore how different parameters affect the behaviour of VACS. Results show that VACS successfully solves our proposed problem.

5.1 Abbreviations

We use abbreviations for performance indicators in our results' tables to make the presentation clearer. In this section, we map abbreviations to their original phrases. For meanings of those measurements, refer to Section 4.3.

Acc

Average Prediction Accuracy

dAcc

Average Prediction Accuracy in Drifting Periods

Mem

Average Memory Usage

Max Mem

Maximal Memory Usage

Time

Total Training Time Usage

PICEC

Percentage of Instances Classified by Expected Classifier

5.2 Statistical Test

We run the algorithm on 20 sample datasets generated by different random seeds for each experiment. We use Welch's t-test [26] to test whether two means are different. Note that Welch's t-test is specifically suitable for comparing samples with different variances. We set 95% as the confidence level. We conclude that there is a difference between two populations if we find the difference in the 95% confidence interval.

When we state a difference between two measurements in this chapter, it means the difference is significant in the Welch's t-test. We also attach the standard deviation of each sample mean in our results.

5.3 Experiments on Data From Mutating Random Tree Generator

In this section, we show the evaluation results run on data generated by the mutating random tree generator. We intend to compare performance differences among three algorithms and inspect the behaviour of VACS. We use the volatility-changing stream generator together with the mutating random tree generator to generate volatility-changing streams.

5.3.1 Data Generator Default Parameters

In this subsection, we specify the default parameter settings for our mutating random tree generator. The data has 10 attributes and 2 classes. The maximal depth of the random tree is 5. We add 5% noise to the data (when generating an instance from the random tree, there is 5% probability that the instance's class label is flipped). The mutating probability of the mutating random tree is 0.2. All experiments share these parameters unless specified differently.

At the drift point, we interleave two streams by a window with size 100 (parameter W for the MOA concept drift generator).

5.3.2 Performance Evaluations of VACS, HAT-ADWIN, and VFDT-ADWIN

The aim of this set of experiments is to show and compare performances of HAT-ADWIN, VFDT-ADWIN and VACS. We measure indicators specified in Section 4.3 on streams with different volatility-changing patterns.

In this experiment, we evaluate performances of three algorithms using the synthetic data with four different volatility changing patterns explained in the following part.

The synthetic data stream is composed of 28 blocks. Each block is composed of 1 million instances (the stream contains 28,000,000 instances in total). We obtain volatility-changing streams by adding a different number of drifts in each block. For each pattern, we generate 20 such stream samples with different random seeds.

Stream with balanced volatility changes

Streams have balanced volatility changes constantly fluctuating between high and low volatility levels and the lengths of each low volatility period and high volatility period are equal. We produce the stream using this process: we add 100 mutations in each of the first two blocks, and 5 mutations in each of the consecutive two blocks. We repeated this pattern 7 times to obtain a stream with 28 blocks.

Streams with the majority of low volatility periods

Lengths of low volatility periods are longer than the high volatility periods in this type of stream. This stream is created by adding 100 mutations in the first 2 blocks, and 5 mutations in the consecutive 5 blocks. We repeated this pattern 4 times to get a stream with 28 blocks.

Streams with the majority of high volatility periods

Lengths of low volatility periods are shorter than the high volatility periods in this type of stream.

We add 100 mutations in each of the first 5 blocks, and 5 mutations in each of the consecutive 2 blocks. We repeated this pattern 4 times to get a stream with 28 blocks.

Streams with composed patterns

We also aim to evaluate learners' performance with irregular volatility changing patterns. Thus, we generate a stream with the composed pattern. This stream is generated by concatenating the first half of *Streams with the majority of low volatility periods* and the second half of *Streams with the majority of high volatility periods.*

Results Analysis

Experimental results are shown in Table 5.1. The bold font denotes the worst performance mean (the least accurate prediction, the longest training time, or the highest memory usage) among three algorithms.

Generally, experiments in all four types of experiments show that the prediction accuracy of VACS is close to the prediction accuracy of HAT-ADWIN in drifting periods (dAcc). So it has similar stableness to HAT-ADWIN when drifts occur. However, compared with HAT-ADWIN, VACS effectively saves training time and memory usage. To see the average prediction accuracy (Acc) of the entire stream, results show that VACS has a better average prediction accuracy than VFDT-ADWIN and a slightly worse prediction accuracy than HAT-ADWIN. This is the expected result since VACS's prediction accuracy is produced by the hybrid system composed by HAT-ADWIN and VFDT-ADWIN.

VFDT-ADWIN has the worst prediction accuracy because it is non-adaptive to changes in the stream, but it has the lowest time and memory overhead. This satisfies our assumptions for VFDT-ADWIN discussed in Section 3.1.

Experiments show the best case for applying VACS is when the stream has a majority of low volatility periods and experiences short high volatility periods occasionally. In this case, VACS can achieve the most effective training time and memory saving. In the results of experiments with the long low volatility periods stream samples, samples' mean training time of VACS is only 48.7% of HAT-ADWIN. VACS's maximal memory usage and average memory is 41% and 46% of HAT-ADWIN. The maximal memory usage of VACS is even close to VFDT-ADWIN, which is considered to be the cheap classifier option without the adaptive power.

The worst case of applying VACS is when the stream has a majority of high volatility periods. However, VACS can still achieve a speed-up and memory usage saving. In experimental results with the sample dataset, training time is 77% of HAT-ADWIN. The maximal memory and average memory cost of VACS are 44% and 65% of HAT-ADWIN. Note that the worst case's maximal memory saving is still close to the saving in the best case.

In the balanced volatility changing sample datasets, the percentage of time and memory reduction is in between the best and the worst cases. The training time of VACS is around 63.2% of HAT-ADWIN's training time. The average memory usage of VACS is 53.2% of HAT-ADWIN's. The maximum memory usage of VACS is 43% of HAT-ADWIN's. Similarly, the reduction of maximal memory usage is close to the results in the best case datasets. Experiments show that VACS's reductions on the maximal memory usage dataset are equally effective regardless of the patterns of volatility changes.

In summary, VACS achieved a reduction in training time and memory compared with the adaptive learner in the above synthetic datasets' experiments. Meanwhile, it maintains a prediction accuracy close to the adaptive learner.

Table 5.1: Performance

Dataset	Balanced		Majority of Low Vol		Majority of High Vol		Composed	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
VACS								
Acc %	84.86	0.28	86.71	0.456	83.43	0.38	84.98	0.34
dAcc %	81.2	0.41	81.85	0.49	80.9	0.55	81.45	0.51
Mem (bytes)	413178.93	30237.52	515455.88	4891.55	338391.59	22452.62	439684.71	28795.49
Max Mem (bytes)	2200813.6	540997.42	2372156	430484.32	2108248	480220.7	2625750.4	630048.8
Time (sec)	230.31	5.83	190.78	2.75	256.93	6.11	237.31	7.41
VFDT-ADWIN								
Acc %	**83.5**	0.24	**85.62**	0.5	**81.78**	0.4	**83.37**	0.31
dAcc %	**78.85**	0.37	**79.45**	0.6	**78.85**	0.37	**78.9**	0.31
Mem (bytes)	275669.66	29343.87	392995.25	55100	194866.07	24074.3	289343.15	35649.62
Max Mem (bytes)	2068718.4	519383.28	2274702	485691.92	1987230.4	469304.53	2327595.6	649198.25
Time (seconds)	123.72	2.04	126.31	2.62	116.47	2.34	126.25	1.13
HAT-ADWIN								
Acc %	85.77	0.25	87.68	0.41	84.06	0.29	85.77	0.32
dAcc %	81.8	0.41	82.35	0.49	81.3	0.47	81.75	0.44
Mem (bytes)	**775754.48**	86571.52	**1115474.7**	158771.26	**520137.73**	77229.29	**867333.47**	119946.78
Max Mem (bytes)	**5139417.6**	1143583.34	**5754436**	1218915.82	**4763293.2**	959179.97	**5810193.6**	1493533.96
Time (seconds)	**364.69**	6.55	**392.04**	11.84	**333.41**	8.87	**364.71**	9.21

5.3.3 Evaluate Learner Selection Quality Of VACS

The objective of these experiments is to evaluate whether VACS selects appropriate learners as expected. We test VACS with the four different types of datasets (20 samples in each type). We calculate PICEC for each type of dataset. PICEC is an important indicator to see whether VACS applies appropriate learners. We also count the percentage of periods in which VACS applies each learner in a stream. The results are shown in Tables 5.2 to 5.5.

Results show decent PICECs for all types of data. PICECs for all datasets' experiments reach around 90%. The percentage of applying the low volatility learner and the high volatility learner are also consistent with the type of data. For example, in streams with a majority of low volatility periods, 72% of the instances are classified by the low volatility classifier (VFDT-ADWIN) and 28% of the instances are classified by the high volatility classifier (HAT-ADWIN).

To better understand the behaviour of VACS, we plot its measured volatility levels γ and the mean value μ of double reservoirs in samples from each type of datasets so that we can visualise the relationship between these two values. See Figures 5.1, 5.2, 5.3, and 5.4. The X-axis of the plots indicates indices of instances in a data stream. The Y-axis of the plots is volatility levels. The solid line shows VACS's measured volatility γ and the dotted line is μ. The plots show that μ is unstable at the beginning of the stream, but after a short period, it successfully "cuts" γ value into upper and lower parts indicating high and low volatilities.

Even though PICECs are reasonably high, the classifier selection has errors. By investigating the results, we found that the main source of these errors is volatility measurement delay. For example, when the stream volatility is entering the high period from the low period, the sliding volatility measurement window (discussed in Section 3.3) cannot immediately detect the transition, because the sliding window still contains old volatility information. The sliding volatility measurement window cannot produce accurate results until most of the out-dated information is removed by sliding them out. The other source of errors is that the volatility measurement quality is affected by random noise. For example, in Figure 5.3, some high volatility levels in the high volatility period go below the double reservoirs mean due to the randomness of the generated stream.

In summary, VACS behaves as expected given measurements of PICECs in these datasets. VACS is able to change expected classifiers in different volatility periods.

Table 5.2: VACS Behaviour in balanced volatility changes stream.

	Mean	SD
PICEC %	89	1.8
Low vol learner %	52	2.1
High vol learner %	48	2.1

Table 5.3: VACS Behaviour in streams with majority of low volatility periods

	Mean	SD
PICEC %	94	1.5
Low vol learner %	72	1.3
High vol learner %	28	1.3

Table 5.4: VACS Behaviour in streams with majority of high volatility periods

	Mean	SD
PICEC %	88	2.4
Low vol learner %	37	1.9
High vol learner %	63	1.9

Table 5.5: VACS Behaviour in composed volatility stream

	Mean	SD
PICEC %	93	1.6
Low vol learner %	51	2.8
High Vol learner %	49	2.8

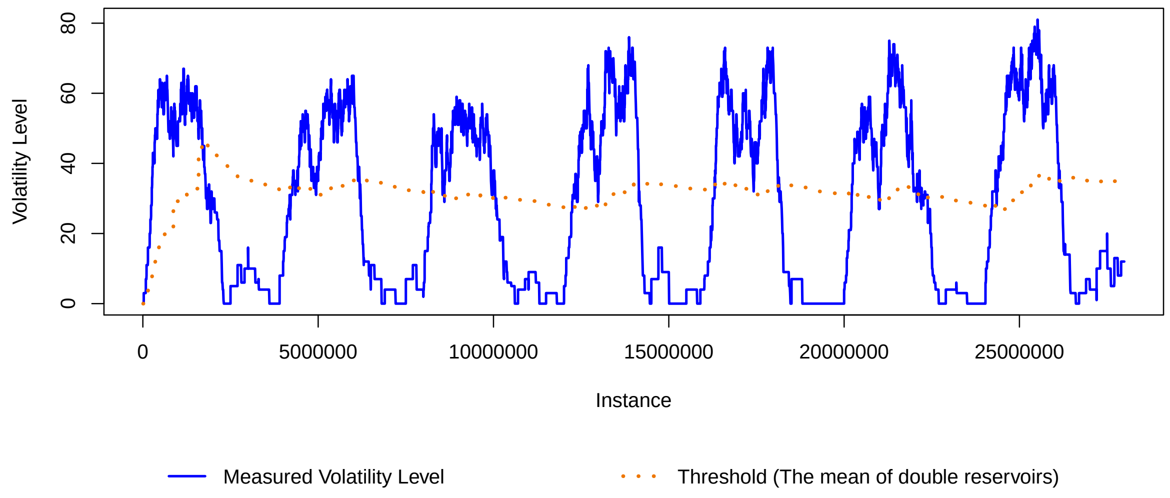

Figure 5.1: Volatility measurements in balanced volatility changes stream

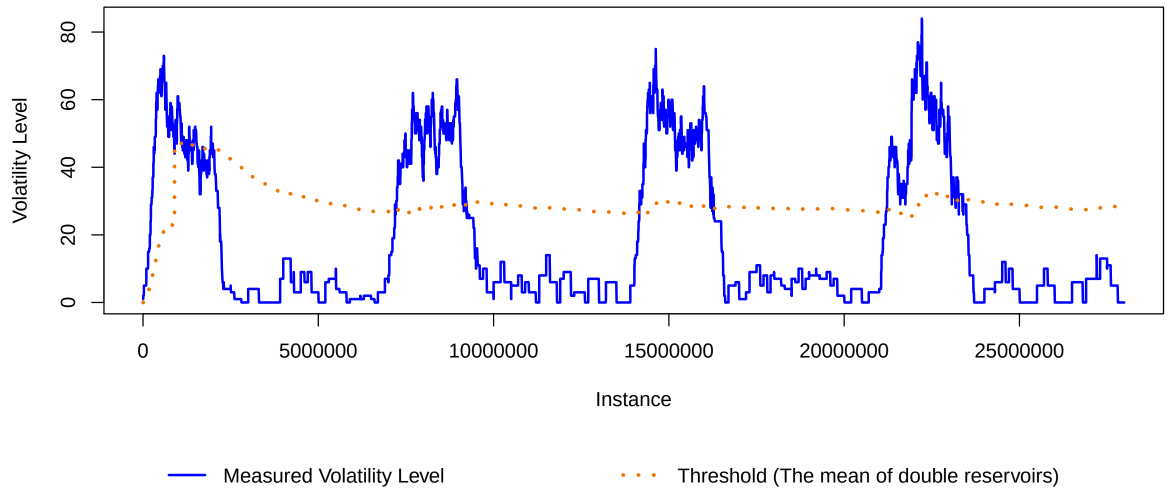

Figure 5.2: Volatility measurements in streams with majority of low volatility periods

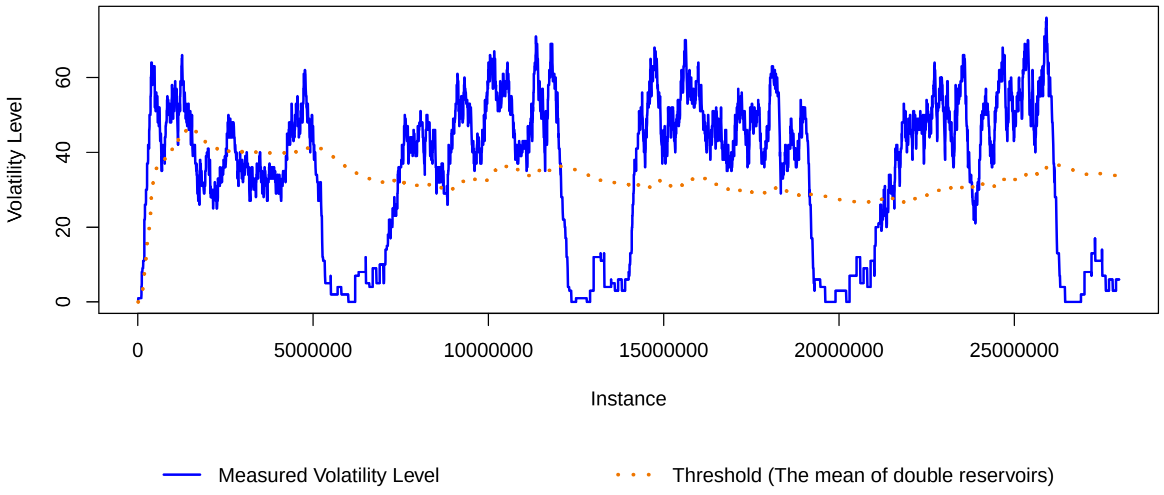

Figure 5.3: Volatility measurements in streams with majority of high volatility periods

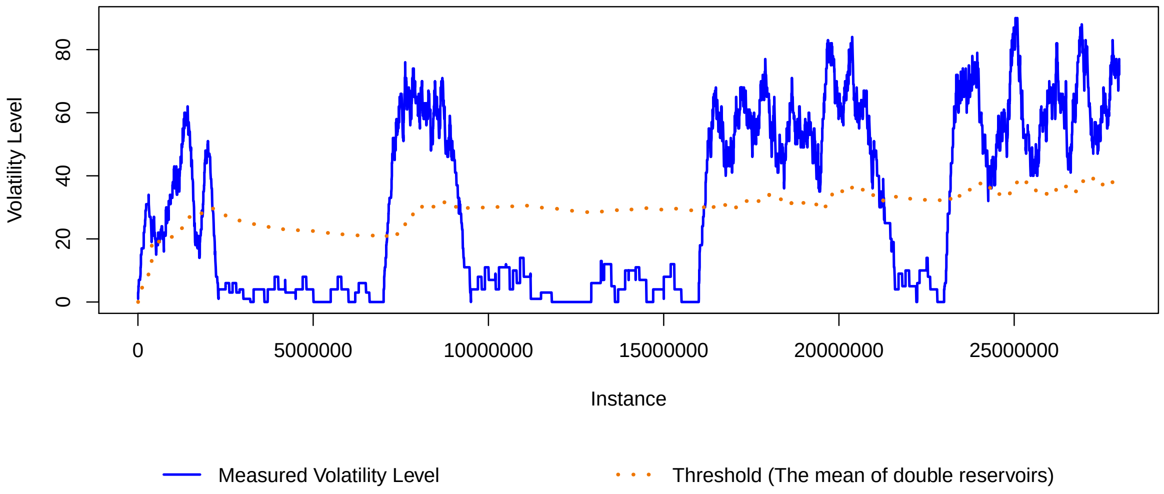

Figure 5.4: Volatility measurements in composed volatility stream

5.3.4 Experiments on Different Volatility Measurement Window Size

The objective of these experiments is to inspect the influences of different Volatility Measurement Window Size T (See Section 3.3 for Volatility Measurement Window) on the learner selection quality of VACS.

In previous experiments, we use 300,000 as the default value for T. In this experiment, we vary T by both increasing and decreasing from its default value.

We run VACS on 20 samples of the balanced volatility stream. The experimental results are shown in Table 5.6.

In the table, we see that when we set T to 3,000 and 30,000, both PICEC are 50% without any variation in all samples in our balanced volatility dataset. This is because the double reservoirs classifier selector is never active in these two cases, so there is no learner switch happening in VACS. Specifically, we set the starting classifier as VFDT-ADWIN (low volatility classifier), and if double reservoirs are not active, VACS uses VFDT-ADWIN for the whole time. Moreover, the balanced dataset has equal periods of high and low volatility. Thus, all the low volatility periods are correctly covered by the low volatility classifier but none of the high volatility periods is covered by the high volatility classifier, which gives a 50% PICEC. We also found the reason the double reservoirs is never active in these two cases - a small volatility measurement window size cannot differentiate high and low volatility periods because a small window can only count a small number of drifts occurred in a short period. Thus, the largest differences between high volatility level and low volatility level are bounded by small values. Results also show that a small window makes VACS problematically sensitive to volatility change caused by noise. This is because the size of the window is the fixed period that we take volatility measurements, and if in a short period, the detections of drift suddenly become frequent due to the random noise, VACS with a small window size may take the change seriously and switch to the high volatility classifier. In contrast, with a larger window, increasing the frequency of drifts in a short period will not signal a great volatility change because the measured random volatility level soar is dampened by other recently measured volatility levels in the window.

In Figure 5.5, we plot VACS's volatility measurement and reservoirs mean when $T = 3,000$ in one sample dataset to visualise its behaviour. Compared with Figure 5.1 when $T = 300,000$. We can see that the measured volatility level dramatically fluctuates due to its sensitiveness. Additionally, measurements in the case with a small $T = 3,000$ are bounded to a smaller value. One the Y-axis, we can see that the maximal measured volatility level is 12. In contrast, Figure 5.1 has around 80.

Table 5.6 shows that a large T will also result in issues. When $T = 3,000,000$, the sample mean PICEC is only 29%, which is considered to be poor. This is because the window with large size always includes periods of various volatility levels. This long period always contains information from both high volatility and low volatility simultaneously, and it may never measure the period with one level of volatility. For example, in our experiment, the balanced volatility dataset fluctuates between high and low volatility for every 2 million instances, but if the measurement window size is 3 million, the window always estimates drifts collected from both high and low volatility periods. As a result, the measured volatility levels cannot accurately represent its actual level. Problematic measurement delays are also possible in this case. Figure 5.6 plots a sample result for the volatility measurements of VACS when $T = 3,000,000$ and $T = 300,000$ along with their double reservoirs means ($T = 300,000$'s scatter line is same as the one from Figure 5.1 which has a decent PICEC). The higher solid line shows volatility measurements of $T = 3,000,000$. In this plot, it can be seen that compared with smaller $T = 300,000$, VACS with $T = 3,000,000$ experiences the measurement delay in the first few waves. In the last wave of $T = 3,000,000$ VACS' volatility levels, the measured levels do not go over the double reservoir means, which was not expected. Additionally, the window with the large size produces higher measurement values because it can count more drifts in a longer period. It can be evident from the plot that $T = 3,000,000$ has a higher measurement value than $T = 300,000$.

We can summarise that classifier selection quality of VACS is influenced by the volatility measurement window size T. Setting the value T to either too small or too large may cause undesirable classifier selection behaviour. T should be large enough such that the measurement is not strongly influenced by the volatility fluctuation of randomness. Moreover, T should not be too large such that VACS can measure the period of one volatility level in most cases. One suggestion of selecting T is to run a testing on the stream before starting VACS. A user can start from a small T value and gradually increase T meanwhile evaluating whether measurements are susceptible to volatility fluctuations caused by noise. When VACS performs as expected, stop increasing T and use this value. We assume that the T value obtained in the pre-experiment period from a continuous stream is also appropriate in the upcoming period in that stream. This assumption holds true in all of our experiments.

Table 5.6: VACS with different volatility measurement window size

T	PICEC % (Mean)	PICEC % (SD)
3000	50	0
30000	50	0
100000	88	1.6
200000	90	1.6
300000	89	1.8
400000	87	1.3
3000000	29	4.3

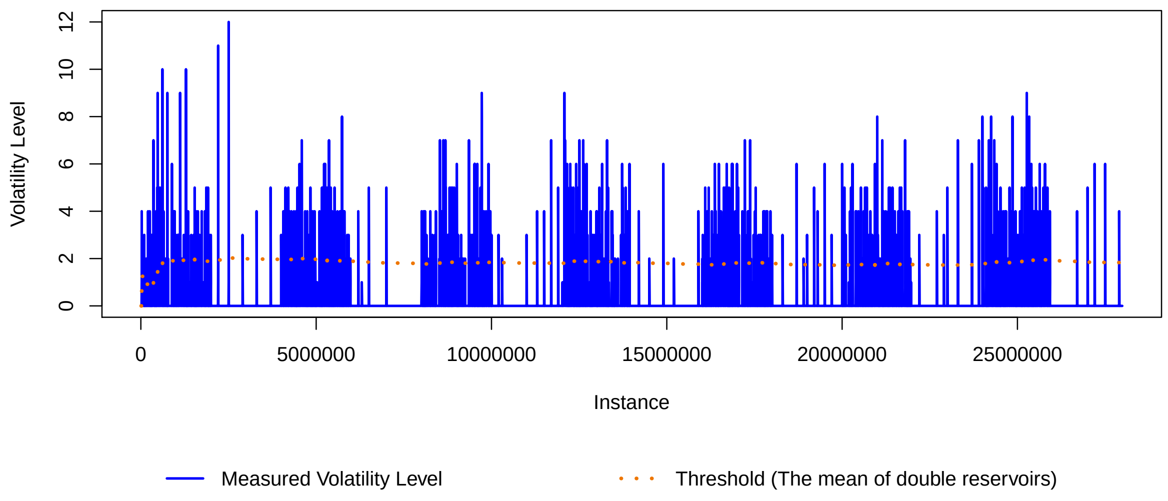

Figure 5.5: VACS with $T = 3000$

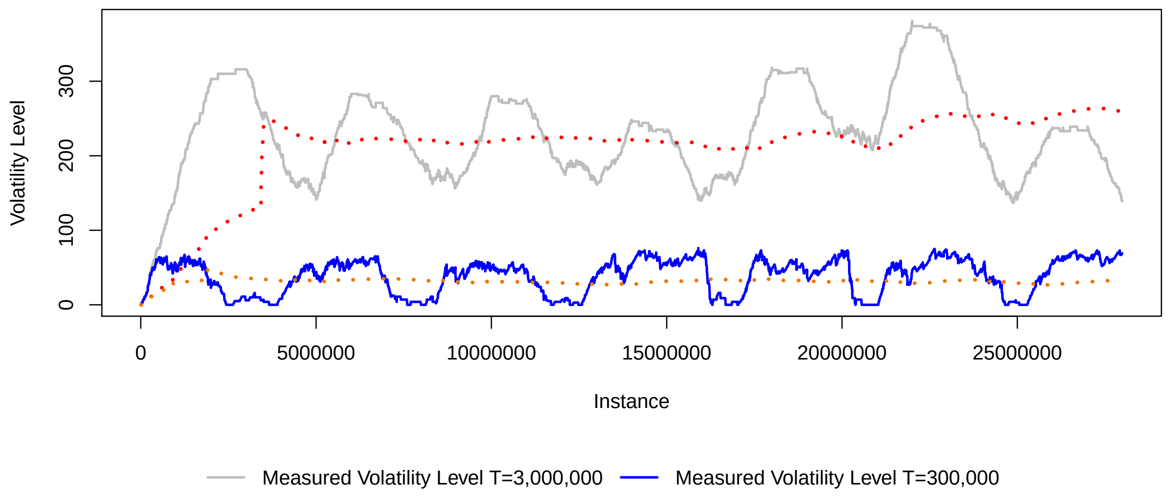

Figure 5.6: VACS with and $T = 300000$ $T = 3000000$

5.3.5 Experiments on Different Reservoir Size

In this section, we aim to evaluate the influences of different double reservoirs' sizes L of DRCS on the learner selection quality of VACS.

In our earlier experiments, we use $L = 200$ as the default reservoir size. In this experiment, we run VACS with small L and large L compared with its default value and inspect how it affects the behaviour and PICEC of VACS. We run VACS in 20 sample datasets from our balanced volatility-changing stream type.

We run VACS when $L = 5$ and $L = 10$ for small reservoir size tests and $L = 1,000$ and $L = 10,000$ for large reservoir size tests. Table 5.7 shows experimental results. When L is set to small values 5 and 10, the PICEC is relatively poor. This is because the size of double reservoirs is too small to estimate the overall volatility of the stream. As a result, the reservoirs mean cannot be used as an effective threshold to distinguish between high and low volatility periods in a stream with respect to the overall volatility changes in the stream. In Figure 5.7, we plot the volatility measurements and double reservoir mean of VACS when $L = 10$. It can be seen that the reservoirs mean moves with the pattern of the measured volatility levels and it cannot clearly separate the measured volatility levels into high and low classes.

However, when L is large, we do not see a great change in PICEC.

In summary, a small size of double reservoirs negatively affects VACS's quality of selecting learners, but a large L may not negatively influence its quality. However, the overly large double reservoirs should be used with caution if we only want VACS to compare the current volatility level with the volatility levels samples in a near period to make classifier selection decisions. The overly large double reservoirs will result in longer time for out-dated volatility levels to be removed from them.

Table 5.7: VACS with different double reservoir size

L	PICEC % (Mean)	PICEC % (SD)
5	59	4.8
10	59	3.7
200	89	1.8
1000	87	1.7
10000	87	1.8

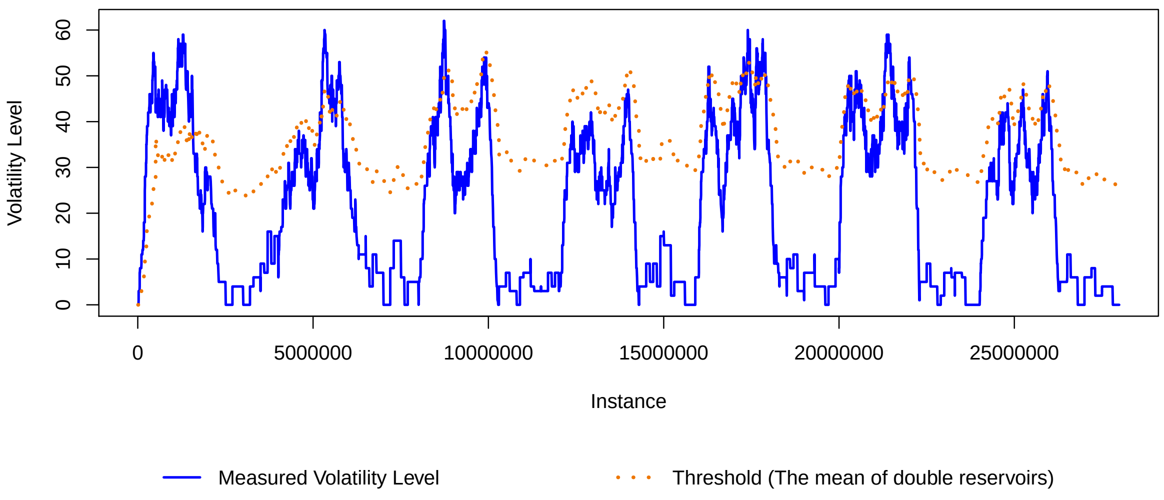

Figure 5.7: VACS with double reservoir size 10

5.3.6 Performances Evaluations of VACS, HAT-ADWIN, and VFDT-ADWIN On Full Drift Datasets

The aim of these experiments is to evaluate performances of VACS, HAT-ADWIN, and VFDT-ADWIN when all attributes' distributions in the dataset are changed during concept drifts. We generate special volatility-changing datasets to compare the performances of three different classifiers by simply dropping the whole random tree and rebuilding a new tree at the drifting point with a new random seed. Such a generation process creates full drifts datasets where all attributes' distributions in the data are changed.

We generate four types of datasets with same volatility changing pattern as experiments in Section 5.3.2. For each high volatility period block, we add 50 drifts. For each low volatility block, we add 5 drifts. Note in the high volatility blocks used in Section 5.3.2, we add 100 mutations. Mutations are different from full drifts because they are nondeterministic, which may result in a full drift, partial drift or no drift in a stream. Intuitively, full drifts cause stronger changes in the dream than mutations.

Experimental results are shown in Table 5.8. The bold font denotes the worst performance mean (the least accurate prediction, the longest training time, or the highest memory usage) among three algorithms. The results are similar to those in Section 5.3.2. In all four datasets, VFDT-ADWIN has the poorest prediction accuracy. VACS uses less training time than HAT-ADWIN while having slightly worse prediction accuracy than HAT-ADWIN. The best cases on applying VACS is the streams with the majority of low volatility periods. What is different in these experiments from those in Section 5.3.2 is that HAT-ADWIN does not experience an effective prediction quality improvement than VFDT-ADWIN. The reason might be that HAT-ADWIN needs to rebuild its whole model in order to learn the evolved concepts caused by full drifts, which is same as VFDT-ADWIN. Thus, HAT-ADWIN loses its advantages of partial model adaptations to maintain a higher prediction reliability when drifts happen. VACS has a higher memory cost than HAT-ADWIN on datasets with majority of high volatility periods. This is because VACS has a larger fixed memory cost than HAT-ADWIN. We investigate VACS's fixed memory cost in later experiments. On other three datasets, VACS has less memory cost than HAT-ADWIN.

The learner selection quality of VACS indicated by PICECs is decent in all four data types. See Tables 5.9, to 5.12.

Overall, in experiments with full drifts datasets, we get the similar conclusion to Section 5.3.2. VACS achieved a high prediction accuracy close to the adaptive learner. And in most cases, VACS had a reduction in training and and memory compared with the adaptive learner.

Table 5.8: Performance with full drifts datasets

Dataset	Balanced		Majority of Low Vol		Majority of High Vol		Composed	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
VACS								
Acc %	83	0.21	84.65	0.23	81.28	0.15	83.04	0.24
dAcc %	78.85	0.37	78.95	0.22	78.9	0.31	78.95	0.22
Mem (bytes)	251366	2211.81	289074.78	1644.9	**214891.71**	2187.39	251548.09	1457.81
Max Mem (bytes)	1036073.6	128215.02	1070771.6	63123.16	929577.2	105318.91	963778	118083.14
Time (sec)	206.76	4.71	175.09	1.88	241.69	10.25	205.69	2.6
VFDT-ADWIN								
Acc %	**82.27**	0.25	**83.98**	0.24	**80.3**	0.13	**82.24**	0.25
dAcc %	**77.45**	0.51	**77.45**	0.51	**77.25**	0.44	**77.45**	0.51
Mem (bytes)	135817.53	1084.04	176949.95	1519.21	94689.76	570.0	135844.06	950.13
Max Mem (bytes)	575813.6	15715.44	589674.8	18281.75	570573.2	16585.42	578405.6	17200.34
Time (seconds)	119.02	1.67	125.02	1.67	119.12	1.77	118.2	1.12
HAT-ADWIN								
Acc %	83.37	0.2	85.06	0.21	81.48	0.12	83.33	0.23
dAcc %	78.9	0.31	79	0.22	78.9	0.31	78.95	0.22
Mem (bytes)	**272390.2**	1854.77	**363111.16**	2780.24	183707.28	1214.94	**272586.71**	1911.32
Max Mem (bytes)	**1304497.2**	44084.06	**1343990**	59456.15	**1305687.2**	41685.42	**1303521.6**	37435.93
Time (seconds)	**304.33**	3.04	**330.26**	2.4	**284.86**	2.57	**300.06**	2.97

Table 5.9: VACS Behaviour in balanced volatility changes stream (Full drift).

	Mean	SD
PICEC %	87	2.1
Low vol learner %	55	2.4
High vol learner %	45	2.4

Table 5.10: VACS Behaviour in streams with majority of low volatility periods (Full drift).

	Mean	SD
PICEC %	92	0.6
Low vol learner %	75	0.8
High vol learner %	25	0.8

Table 5.11: VACS Behaviour in streams with majority of high volatility periods (Full drifts)

	Mean	SD
PICEC %	89	5.3
Low vol learner %	35	5.6
High vol learner %	65	5.6

Table 5.12: VACS Behaviour in composed volatility stream (Full drifts)

	Mean	SD
PICEC %	93	1.4
Low vol learner %	54	1.5
High vol learner %	46	1.5

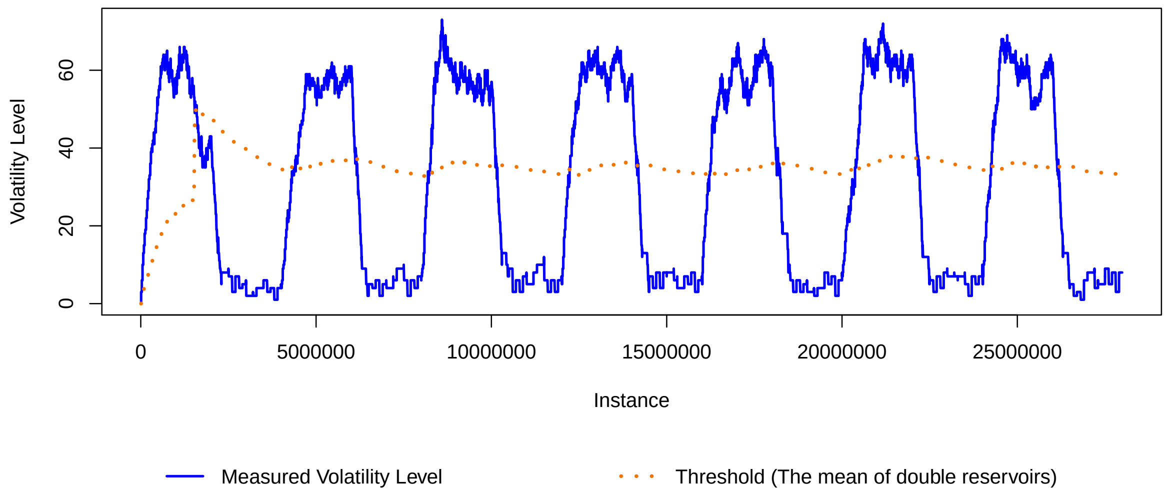

Figure 5.8: Volatility measurements in balanced volatility changes stream (Full drift)

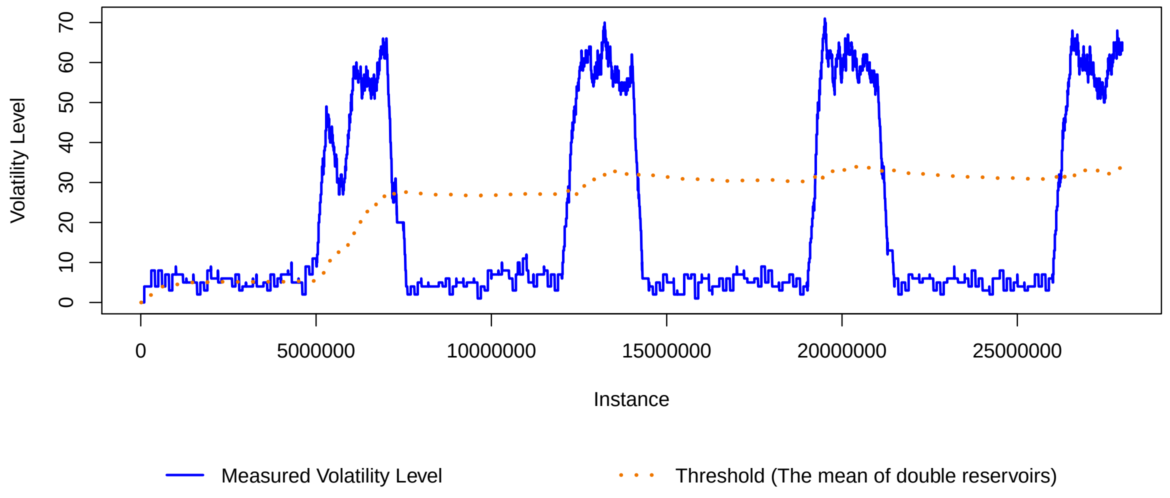

Figure 5.9: Volatility measurements in streams with majority of low volatility periods (Full drift)

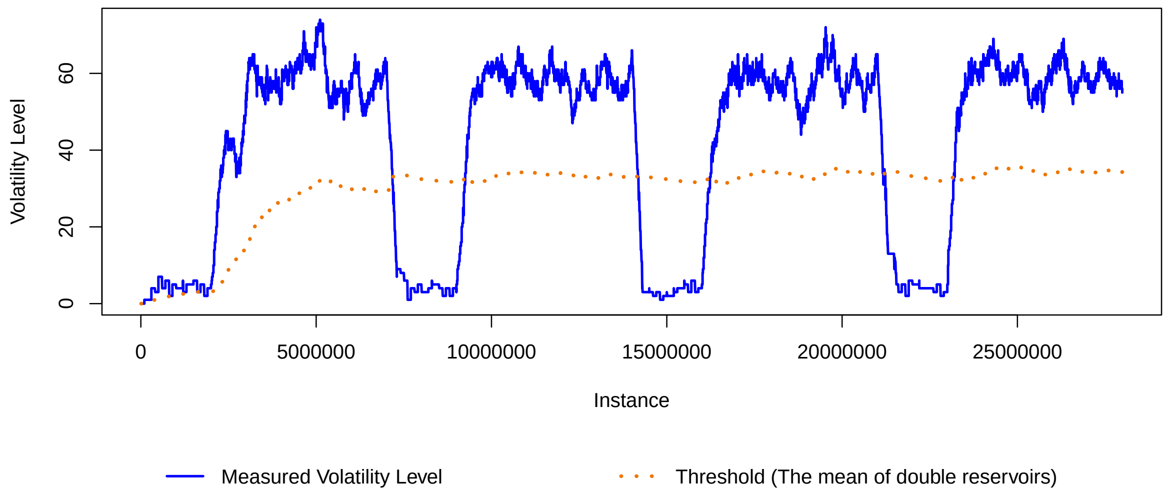

Figure 5.10: Volatility measurements in streams with majority of high volatility periods (Full drifts)

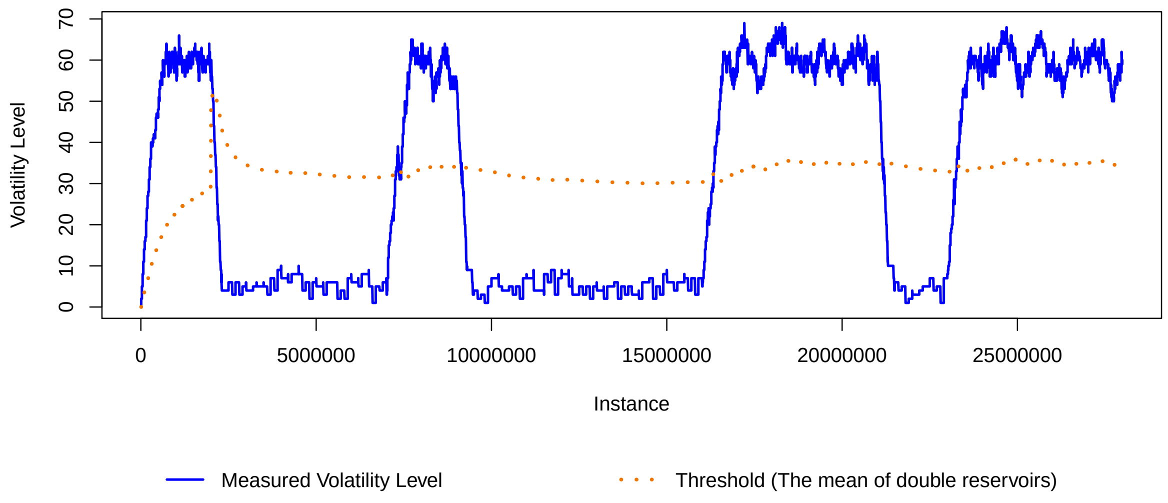

Figure 5.11: Volatility measurements in composed volatility stream (Full drifts)

5.3.7 Performances Evaluations of VACS, HAT-ADWIN, and VFDT-ADWIN on One-Direction Volatility-Changing Data

In these experiments, we run three algorithms in two extreme cases and compare these algorithms' performances. The first case is when the volatility in a stream is always increasing without decreasing. The second one is when the volatility in a stream is decreasing without increasing.

We use the volatility-changing stream generator along with mutating random tree generator to generate such datasets by composing three stream blocks.

In Case 1, we add 5 mutations in the first block, 50 mutations in the second block and 100 mutations in the last block (Mutation is defined in Section 4.4.1). Each block has 1 million instances. We concatenate three blocks to obtain an increasing volatility stream. VACS uses low volatility classifier as the starting classifier.

In Case 2, we reverse the volatility changing the pattern of the first case. We add 100 mutations in the first block, 50 mutations in the second block and 5 mutations in the last block with same lengths as the case 1. Finally, we get the decreasing volatility stream. VACS uses the high volatility classifier as the starting classifier.

We run experiments on 20 samples for each type of data. Evaluations of three algorithms are in Table 5.13. The bold font denotes the worst performance mean (the least accurate prediction, the longest training time, or the highest memory usage) among three algorithms. In these two extreme case, we still find similar conclusions as earlier experiments in Section 5.3.2. VFDT-ADWIN has the poorest prediction accuracy. VACS experiences slight prediction accuracy drop compared with HAT-ADWIN, but it has notable saving on training time and memory usage. In Case 1, VACS's training time is 54.1% of HAT-ADWIN, and its average memory and maximal memory usage are 57.5% and 43.6% of HAT-ADWIN respectively. In Case 2, VACS's training time is 63.5% of HAT-ADWIN, and its average memory and maximal memory usage are 44.7% and 62% of HAT-ADWIN respectively.

We also evaluate classifier selection behaviours of VACS in these datasets. We expect to see that in Case 1, VACS can apply the high volatility classifier in the end, and in Case 2, VACS can apply the low volatility classifier in the end. Experiments show that VACS meets the expectation in all 20 datasets of each case. Figures 5.12 and 5.13 show VACS's volatility measurement and double reservoirs' means plots in samples from two cases. In Figure 5.12 of the dataset with increasing volatility, we can see that VACS's measured volatility levels eventually goes above the threshold and switch to the high volatility classifier. In Figure 5.13, we see the reversed pattern and VACS ends up at the low volatility classifier.

In conclusion, in special cases where the volatility changes in one direction, VACS can work as expected by switching to a different learner eventually. VACS also has the reduction in training time and memory compared with the adaptive learner.

Table 5.13: Performance with One-Direction Volatility-Changing datasets

Dataset	Increasing Vol		Decreasing Vol	
	Mean	SD	Mean	SD
VACS				
Acc %	84.17	1.16	83.77	1
dAcc %	81.7	0.86	81.156	0.99
Mem (bytes)	343791.11	83070.2	347242.9	72669.44
Max Mem (bytes)	1365596	467427.9	1326631.6	456492.98
Time (sec)	19.38	2.39	21.84	1.72
VFDT-ADWIN				
Acc %	**82.82**	1.24	**82.61**	1.02
dAcc %	**79.95**	0.94	**79.7**	0.8
Mem (bytes)	219550.01	79447.1	220034.93	74110.42
Max Mem (bytes)	1254376.8	480149.06	1218673.6	487807.22
Time (seconds)	12.63	0.42	12.71	0.55
HAT-ADWIN				
Acc %	85.23	0.88	84.78	0.93
dAcc %	82.75	0.79	82.2	0.83
Mem (bytes)	**597726.79**	192213.09	**546851.47**	159047.64
Max Mem (bytes)	**3133163.6**	997313.91	**2965717.2**	901221.27
Time (seconds)	**35.85**	1.64	**35.25**	1.24

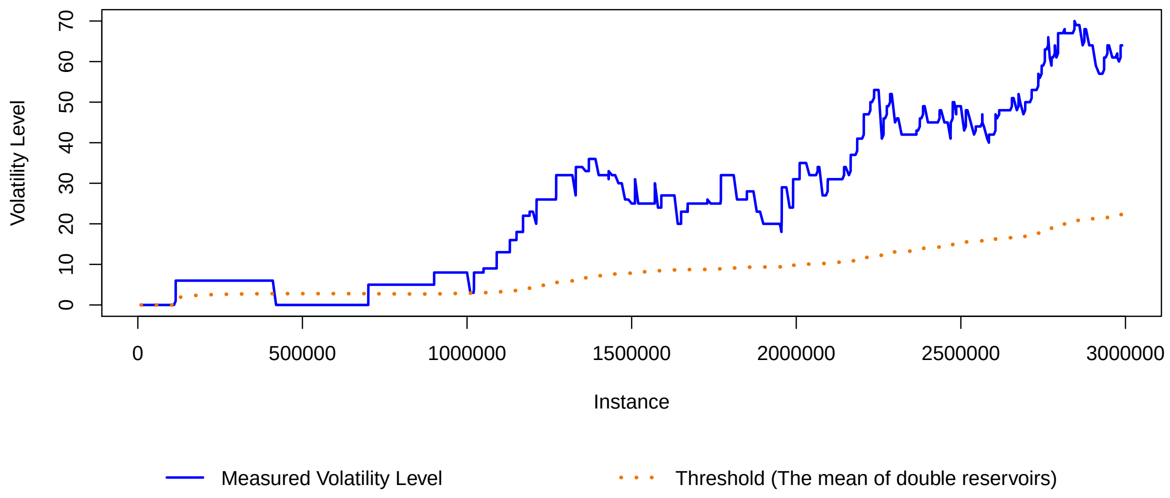

Figure 5.12: Volatility measurements in increasing volatility streams (Case 1)

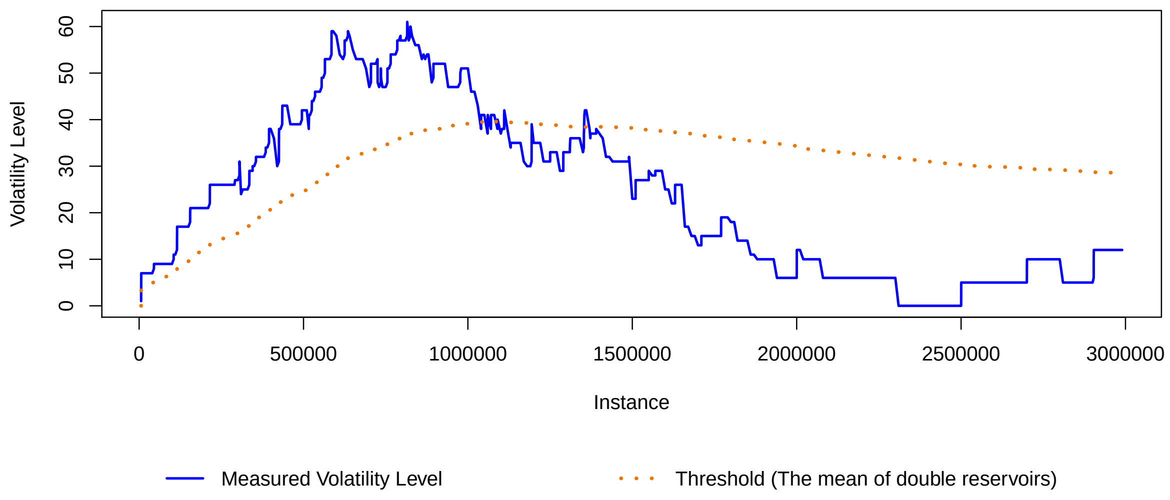

Figure 5.13: Volatility measurements in decreasing volatility streams (Case 2)

5.4 Experiments on Data From SEA Generator

The aim of this set of experiments is to evaluate the performances of HAT-ADWIN, VFDT-ADWIN, and VACS on different data generators. We use SEA Generator as our second data generator.

For SEA generator, the instance has 3 attributes and 2 classes by default. We generate the same four types of volatility-changing data streams with the same volatility patterns as the first experiment in Section 5.3.2.

The experimental results can be viewed in Table 5.14. The bold font denotes the worst performance mean (the least accurate prediction, the longest training time, or the highest memory usage) among three algorithms.

Results show a similar conclusion for reducing training time. VACS uses less training time than HAT-ADWIN. Moreover, in streams composed of a majority of low volatility periods, VACS has the most effective reduction in time.

Three algorithms have the similar prediction accuracy. All prediction accuracies are also higher than the case of mutating random tree generator. This is because the SEA generator's datasets have only 3 attributes, which is significantly fewer than earlier experiments with 10 attributes. Trees can grow into simpler structures and consume fewer data to generalise enough stream concepts. As a consequence, either an adaptive model or non-adaptive model can react efficiently to newly evolved concepts.

One interesting finding is that VACS does not gain average memory usage reductions over HAT-ADWIN in these experiments. This is because VACS has a larger fixed memory cost than HAT-ADWIN since VACS is composed of HAT-ADWIN, VFDT-ADWIN and other components discussed in the methodology chapter. VACS's memory usage becomes dominant in the datasets when its component learners' memory usages are low for all time. VACS's fixed memory dominance is evident in the case when the attribute number is small or the volatility is high, so the tree model will neither grow into complex structures or frequently rebuild themselves due to the high volatility. As a result, component learners consume a relatively small amount of memory, but VACS still needs to consume a fixed amount of memory for each component. We plot the memory usage of VACS and VFDT-ADWIN in a sample stream with balanced volatility changing pattern generated by SEA. The plot can be viewed in Figure 5.14. The Y-axis is the memory usage in bytes and the X-axis is the count of trained instances from the stream. The blue line is HAT-ADWIN, and the orange line is VACS. In this stream, the first 2 million instances are in the high volatility period and the second 2 million instances are in the low volatility period. This volatility fluctuation pattern repeats until the end of the stream is reached. It can be seen that bytes of memory usage are anti-correlated with volatility levels in the stream. More importantly, in the high volatility periods, VACS uses more bytes than

HAT-ADWIN. Moreover, it can also be seen that VACS's memory usage has a fixed lower bound, which causes its high memory usage. However, in high volatility periods, HAT-ADWIN consumes more memory, because VACS uses VFDT-ADWIN in these periods which has a lower memory overhead than HAT-ADWIN.

We also evaluate VACS's learner selection qualities in these datasets. We show the results in Tables 5.15, 5.16, 5.17, 5.18. VACS in all four types of datasets return around 90% values, which is similar to earlier experiments in mutating random tree datasets. This tells us the learner selection qualities of VACS is also decent in SEA generated datasets.

We can conclude that it might not be desirable to use either VACS or the adaptive learners (HAT-ADWIN) in the dataset where the non-adaptive learner (VFDT-ADWIN) has already performed well. For example, in the dataset where the decision tree does not need to grow large to generalise the concept and either non-adaptive or adaptive learner can respond to the concept drifts efficiently. If applying the adaptive learners or VACS in this scenario, we would not gain considerable improvements on prediction quality by using the adaptive learner but still need to take its large overhead. However, if we do not know any prior information about the stream, we can use VACS as a middle choice to mitigate the risk of the low prediction accuracy if accidentally applying the inappropriate learner. Additionally, we can achieve a speed-up compared with the adaptive learner. Experiments also reveal VACS has a larger fixed memory usage than other two learners. It shows the caution to be considered before using VACS. Specifically, in the stream where VACS's component learners' memory consumption is more dominant than VACS's fixed memory usage, for example, in the case where the decision tree learner can grow large, it is suggested to use VACS to save the memory cost. Otherwise, VACS may cost more memory than the adaptive learner.

Figure 5.14: Memory usage of VACS in SEA generated data streams

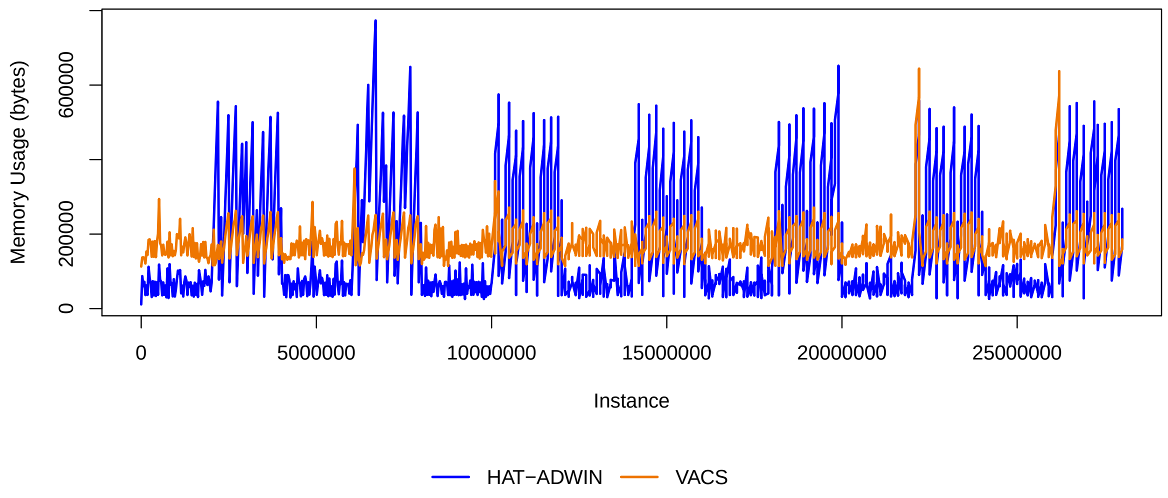

Table 5.14: Performance with SEA generator datasets

Dataset	Balanced		Majority of Low Vol		Majority of High Vol		Composed	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
VACS								
Acc %	92.14	0.08	92.62	0.07	91.69	0.19	92.15	0.11
dAcc %	90.85	0.37	90.9	0.31	90.75	0.44	90.65	0.49
Mem (bytes)	**173849.41**	3951.98	172856.88	2138.16	**163834.01**	2091.83	**170996.52**	2359.62
Max Mem (bytes)	572069.6	56281.64	576905.2	58845.55	532899.6	57047.26	554536.8	61840.14
Time (sec)	127.33	4.33	105.41	2.39	143.82	3.3	126.62	2.2
VFDT-ADWIN								
Acc %	**92.09**	0.08	**92.52**	0.07	**91.67**	0.16	**92.09**	0.12
dAcc %	**90.85**	0.37	**90.8**	0.41	**90.65**	0.49	**90.6**	0.5
Mem (bytes)	56843.41	1871.21	63692.83	1191.53	41673.5	1123.55	56476.84	1246.22
Max Mem (bytes)	186323.2	15153.59	177403.6	9168.92	175032	5283.07	182452.4	11580.66
Time (seconds)	64.33	1.64	65.06	1.04	62.41	1.26	64.72	1.4
HAT-ADWIN								
Acc %	92.2	0.08	92.66	0.07	91.71	0.19	92.18	0.11
dAcc %	90.85	0.37	90.9	0.31	90.75	0.44	90.65	0.49
Mem (bytes)	161413.3	4963.52	**201208.75**	3811.19	112874.55	4215.42	160569.33	5425.01
Max Mem (bytes)	**666700**	71667.62	**688492.4**	88055.91	**637625.2**	67176.21	**698805.2**	74618.09
Time (seconds)	**188.22**	3.38	**209.84**	3.4	**171.86**	2.82	**193.54**	2.73

Table 5.15: VACS Behaviour in balanced volatility changes stream (SEA).

	Mean	SD
PICEC %	90	2.6
Low vol learner %	49	2.9
High vol learner %	51	2.9

Table 5.16: VACS Behaviour in streams with majority of low volatility periods (SEA).

	Mean	SD
PICEC %	94	1.6
Low vol learner %	71	1.4
High vol learner %	29	1.4

Table 5.17: VACS Behaviour in streams with majority of high volatility periods (SEA)

	Mean	SD
PICEC %	91	2.7
Low vol learner %	33	1.8
High vol learner %	67	1.8

Table 5.18: VACS Behaviour in composed volatility stream (SEA)

	Mean	SD
PICEC %	95	0.9
Low vol learner %	51	1.3
High vol learner %	49	1.3

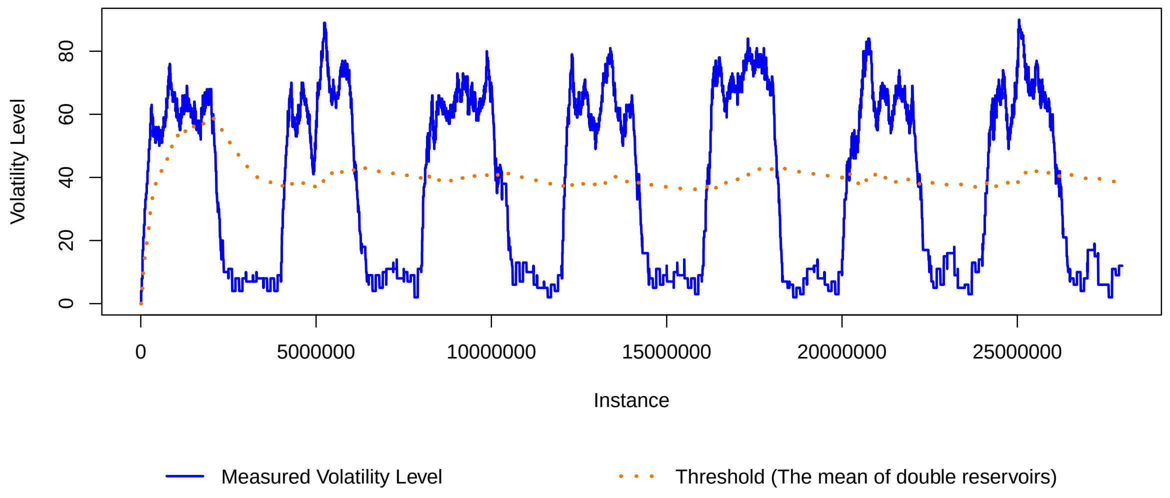

Figure 5.15: Volatility measurements in balanced volatility changes stream (SEA)

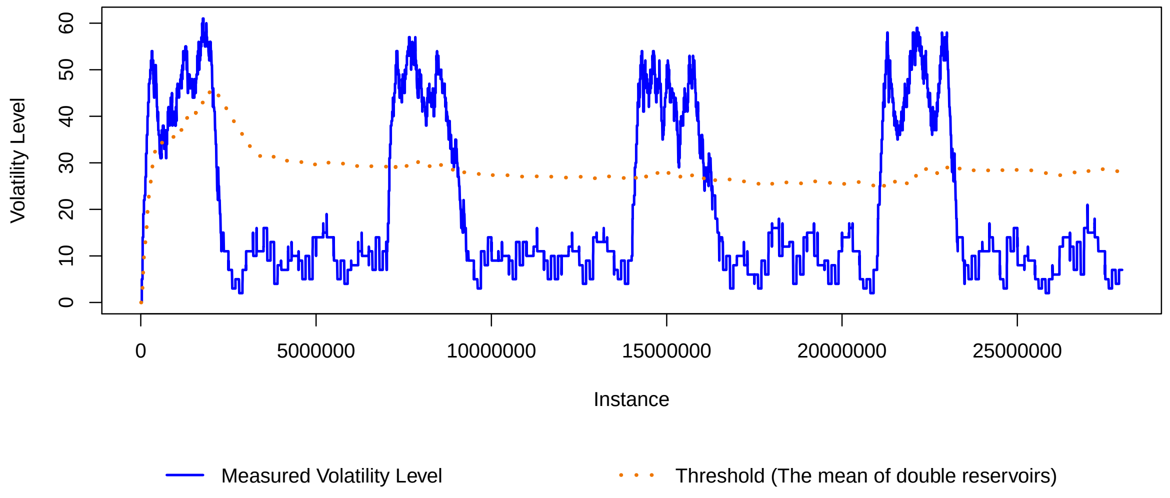

Figure 5.16: Volatility measurements in streams with majority of low volatility periods (SEA)

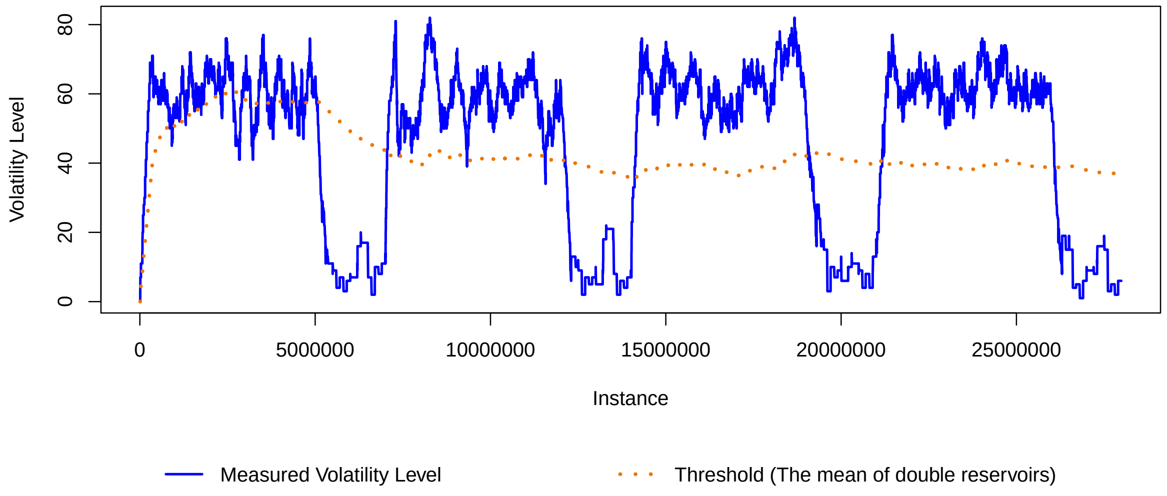

Figure 5.17: Volatility measurements in streams with majority of high volatility periods (SEA)

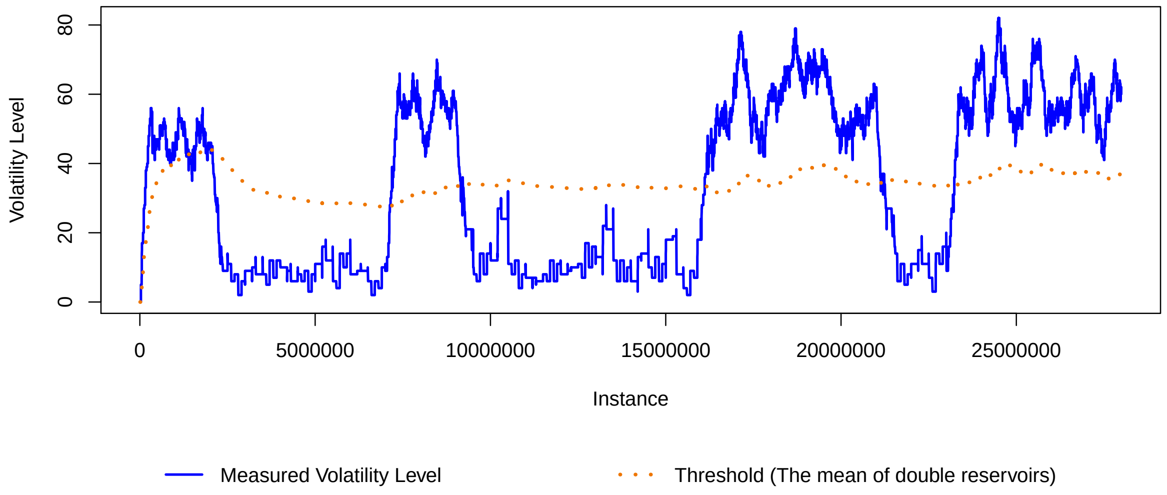

Figure 5.18: Volatility measurements in composed volatility stream (SEA)

5.5 Experiments on Real-World Data

The aim of this set of experiments is to evaluate and compare the performances of three algorithms in real-world datasets. We chose 3 real world datasets available on MOA [6] Datasets[1] which contain volatility changes.

Poker Hand

This dataset contains 1,000,000 instances with 10 attributes. Each instance is a hand consisting of five playing cards drawn from a deck of standard poker. Each two attribute represent one card by recording its suit and rank. The prediction class has 10 possible labels describing the poker hand.

Forest Cover Type

The dataset contains information about forest cover type for 30 x 30 meter cells. It contains 581,012 instances with 54 attributes. The prediction class has 7 possible labels.

Airlines

The dataset contains information about flights, and the prediction label is whether it will be delayed (binary class). It has 539,383 instances with 4 attributes.

Results Analysis

We run three algorithms on three datasets, and we measure those indicators in Section 4.3 excluding the accuracy during drifts. The reason that we exclude that measurement is that we did not add synthetic drifts in the real world data. We run the experiment only once because the experimental is deterministic. Table 5.19 shows the experiment's results. The bold font denotes the worst performance mean (the least accurate prediction, the longest training time, or the highest memory usage) among three algorithms.

We show experimental results in Table 5.19. We also plot VACS' volatility measurements in Figure 5.19 to 5.21 to demonstrate the classifier selection behaviour of VACS.

We follow the method of choosing volatility measurement window size T proposed in Section 5.3.4 and set $T = 20,000$ for VACS running on three datasets. Becuase a smaller T is suggested to be coupled with a smaller λ, we set λ to 3.

In the experiment's results of Poker Hand and Forest Cover Type datasets, we have two main findings. Firstly, VACS has the highest prediction accuracy among the three, and HAT-ADWIN has the worst prediction. Secondly, VACS has the largest memory consumption among others. The reason VACS has the highest prediction accuracy is that

[1]http://moa.cms.waikato.ac.nz/datasets/

we found in these datasets that HAT-ADWIN's accuracy is better than VFDT-ADWIN's when the volatility is high, but its accuracy is lower than VFDT-ADWIN's when the volatility is low. VACS has the largest memory usage. The memory usage for the adaptive learner is lower than VACS because in these real-world datasets, so VACS's component learners do not consume a large portion of memory due to frequent tree reconstructions and VACS's fixed memory usage dominates the overall memory usage. Figure 5.22 shows the memory usage of HAT-ADWIN and VACS in the Forest Cover Type dataset. VACS has a higher fixed memory cost such that its overall memory usage is higher than HAT-ADWIN's. The reason behind this phenomenon is that the drift detector detects much more frequent drifts in these two real-world datasets than our synthetic data. This can be shown by comparing the volatility measurements between synthetic dataset experiments and real world dataset experiments. For example, in Figure 5.20, the majority of measured volatility levels ranged between 20 to 40 given $T = 20,000$, which means there are mostly 20 to 40 drifts detected in the window with size 20,000. Moreover, in Figure 5.1 of the synthetic dataset, the maximal measured volatility level is about 80 given $T = 300,000$, we can estimate that there are only at most 5 drifts detected suppose the window has size 20,000 in our synthetic dataset. One side conclusion is that HAT-ADWIN may not always have better prediction accuracy than VFDT-ADWIN's even though HAT-ADWIN is an adaptive algorithm and is expected to have a better prediction quality.

In the experiment in the Airline dataset, we find a different result from the above cases. VFDT-ADWIN performs better than HAT-ADWIN regarding the prediction accuracy. In this case, VACS has better prediction accuracy than HAT-ADWIN's but slightly worse than VFDT-ADWIN's. The reason is that we found that VFDT-ADWIN predicts more accurately than HAT-ADWIN in most periods of this stream. VFDT-ADWIN also has the lowest overhead. Thus, VFDT-ADWIN is the best choice for learning from and predicting on this dataset. Because in the previous two real-world datasets' experiments, we found that VFDT-ADWIN has better prediction accuracy than HAT-ADWIN when the volatility is low, one reason that VFDT-ADWIN has a better overall prediction accuracy might be that the Airline dataset has less frequent drifts than the other two real-world datasets. It can be evident by comparing Figure 5.19 to 5.21. With the same $T = 20,000$, the maximal volatility levels measured for Airline datasets is only 14, while the majority of volatility levels measured for the other two datasets ranged between 20 to 40. In this dataset, VACS consumes less memory than HAT-ADWIN because tree models are able to grow large. This is because the Airline dataset has two nominal attributes "flightFrom" and "flightTo" which have a large number of possible values (the names of different cities). As a result, when the decision tree splits on these attributes, it will grow large numbers of branches thus has a large memory consumption. Therefore, if the component learners' memory is more dominant, VACS has a memory saving compared with the adaptive learner, which is consistent with our results from synthetic data.

We can visualise the behaviour of VACS in Figure 5.19, 5.20, 5.21. It can be seen that VACS produces a threshold (denoted by the dotted line) based on the sampled volatility levels in the double reservoirs. It separates the high volatility and the low volatility periods in these real-world datasets. When the volatility level goes above the threshold, VACS applies the adaptive classifier, otherwise, the non-adaptive classifier is used. These results satisfy our expectations.

In all experiments, VACS achieves a training time reduction compared with HAT-ADWIN.

In summary, in these real-world experiments, VACS does not produce worse prediction accuracy than other two learners, but can even produce better prediction accuracy than the other two, which exceeds our expectation. Additionally, VACS can still achieve a training time reduction compared with the adaptive learner (HAT-ADWIN). We also notice one finding beyond our research scope: the adaptive learner may produce worse prediction accuracy than the non-adaptive learner. So the intention of using an adaptive learner to obtain a high-quality prediction may fail. Specifically, in these real-world datasets, VFDT-ADWIN predicts better than HAT-ADWIN when drifts are not frequent. Results suggest that VACS can be a better option when users do not know which learner to choose to learn from the incoming stream because results show that VACS has the potential to give a prediction accuracy at least better than one of its component learners. Meanwhile, it also saves training overhead compared with the adaptive learner.

Table 5.19: Performance with Real-World datasets

Dataset	PokerHand	ForestCovType	Airline
VACS			
Acc %	72.0	82.14	64.98
Mem (bytes)	**124684.11**	**176295.35**	2642422.83
Mem (bytes)	**163392**	**357528**	10468520
Time (seconds)	4.29	11.62	10.14
VFDT-ADWIN			
Acc %	69.68	81.77	65.28
Mem (bytes)	28676.98	67016.01	1832277.51
Mem (bytes)	40192	131256	7855680
Time (seconds)	2.36	5.94	9.05
HAT-ADWIN			
Acc %	**66.42**	**81.42**	**63.37**
Mem (bytes)	23741.62	98633.23	**7024953.36**
Mem (bytes)	83600	304024	**13914984**
Time (seconds)	**5.12**	**14.37**	**15.82**

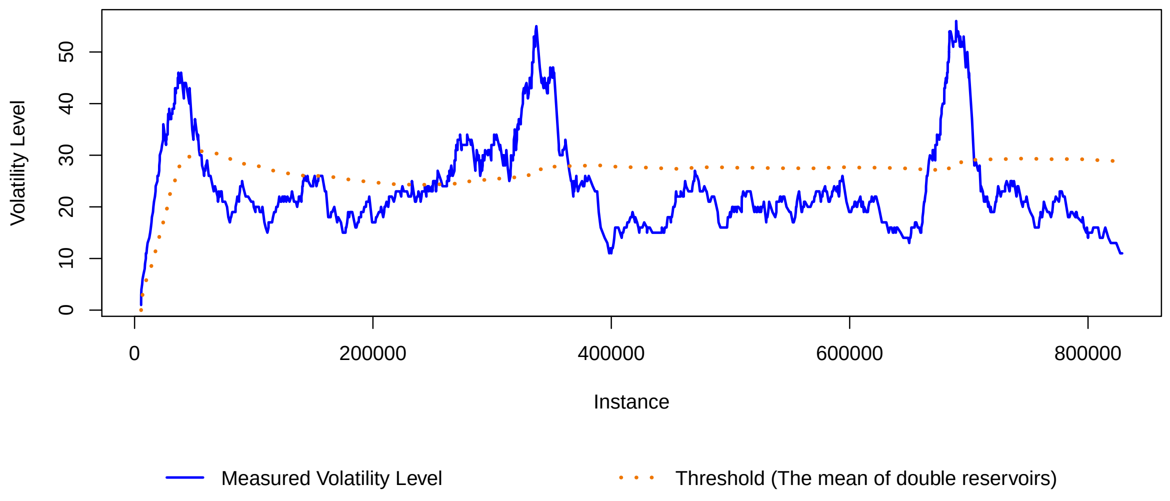

Figure 5.19: Volatility measurements in the Poker Hand dataset

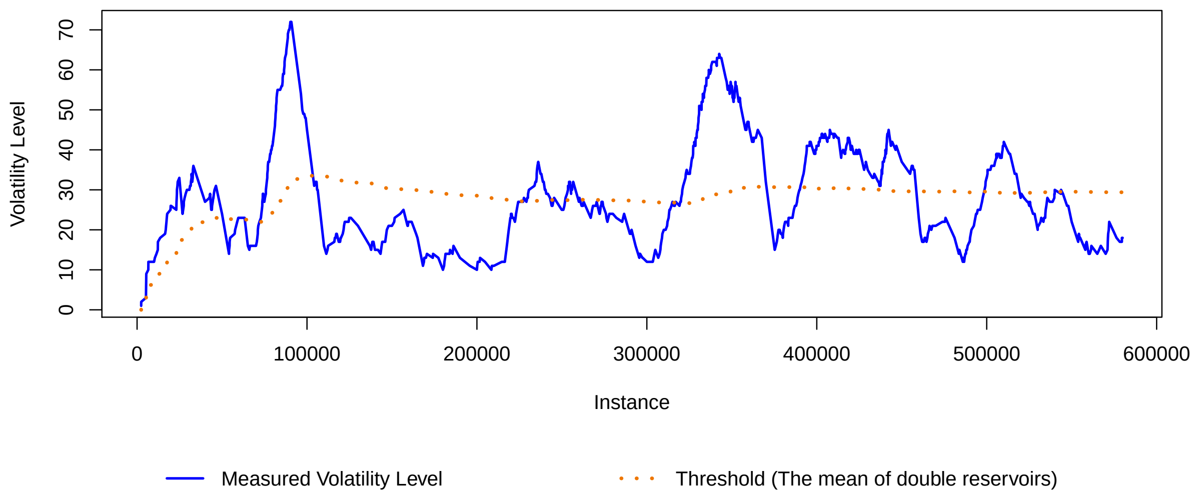

Figure 5.20: Volatility measurements in the Forest Cover Type dataset

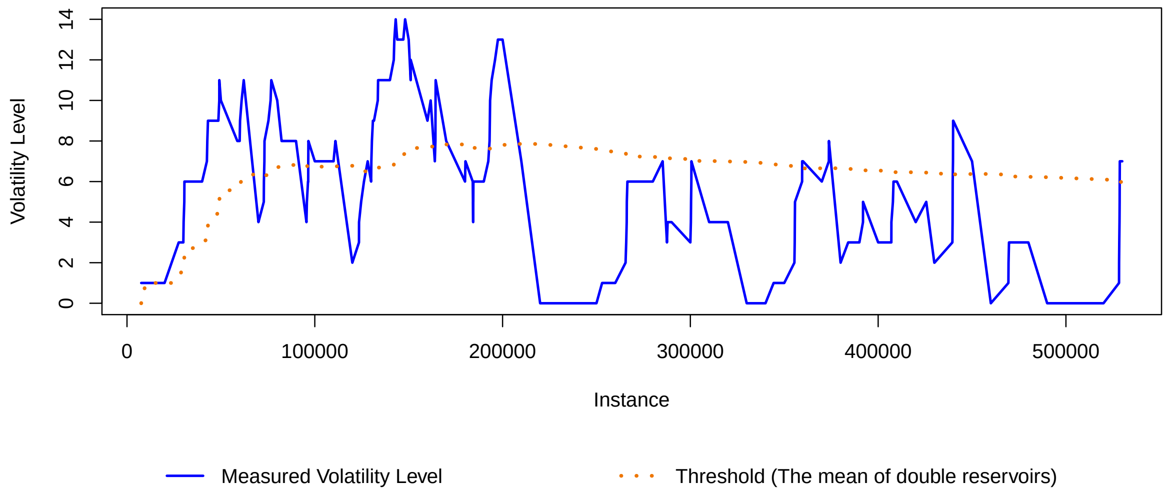

Figure 5.21: Volatility measurements in the Airline dataset

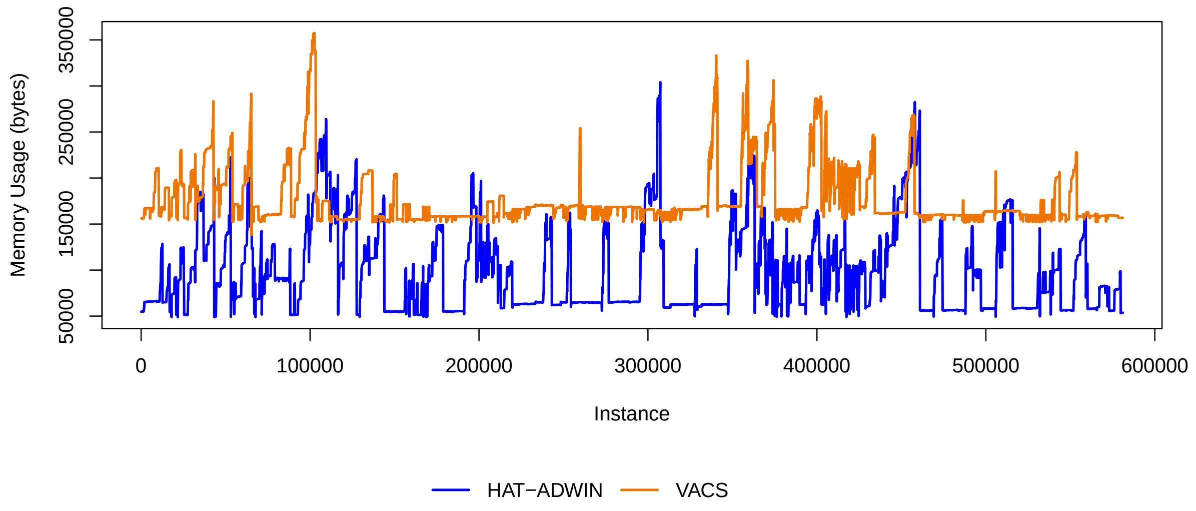

Figure 5.22: VACS's Memory Usage in the Forest Cover Type dataset

6 Discussion

In this chapter, we summarise our experiments.

Based on the results of experiments, we discuss some suggestions when one wants to apply VACS to learn a data stream effectively.

Additionally, RCD model has some similar aspects to our VACS, however, RCD model is fundamentally different from our framework. We also compare and contrast RCD model with our VACS in this chapter.

6.1 Summary on Experimental Results

In this section, we review our experimental results.

In our experiments with synthetic data, we showed the performance differences of three learners: VFDT-ADWIN, HAT-ADWIN, VACS on various datasets. The data are generated by the mutating random tree generator. This generator can produce partial drift data. The adaptive learner is expected to perform better with such partial drift data than the non-adaptive learner. We generate four groups of synthetic datasets with different volatility changing patterns: streams with balanced volatility changes, streams with majority of low volatility periods, streams with the majority of high volatility periods, and streams with composed patterns. VFDT-ADWIN is a non-adaptive learner with low overheads while HAT-ADWIN is an adaptive learner with high overheads. In these experiments, we found that VACS can reduce both training time and memory usage compared with the individual adaptive learner while maintaining decent prediction accuracy close to the adaptive learner. The best case of applying VACS is in streams with longer low volatility periods. In this case, it only applies the costly adaptive learner in the short periods where drifts are more frequent and uses the cheap non-adaptive learner most of the time when drifts are not very frequent. The result is that VACS can produce decent overall prediction accuracies but does not need to spend unnecessary time and memory on training the adaptive learners when drifts are not frequent. The worst case of using VACS is when streams contain long high volatility periods. In this case, memory and time reduction is not notable, because it uses the adaptive learner most of the time. We also generate variant datasets of the above four types by adding full drifts (all attributes experience the concept drift) in the streams. We find a similar conclusion on time and memory reduction with reasonable prediction accuracy for VACS compared with the individual adaptive learner. Finally, we generate same four types of volatility-changing data streams using the SEA generator. The SEA generator has fewer attributes and decision tree models do not need to grow large to generalise the concept. In this case, the adaptive learner's prediction quality improvement is not very notable, and the VFDT-ADWIN is expected to be used given its high prediction accuracy and low overhead. Thus, VACS might not be desirable to use if the user already knows the non-adaptive learner is the good choice. However, without any prior information about the stream, VACS can be a middle option to mitigate the adverse effect of using the wrong learner. We also found that VACS has a large fixed memory overhead, and it may lose memory reduction benefit if its fixed memory overhead dominates its component learners' costs.

We also investigated VACS's behaviour and whether it can apply the appropriate classifier on periods with different volatility levels. We see decent results on these experiments and VACS can expectedly perform learner selection. We have found that the value of volatility measurement window's size T is related to the quality of selecting learners. We proposed a method to choose T in Section 5.3.4.

In experiments on real-world data, we found that VACS uses less training time than the adaptive learner. However, VACS does not always gain reductions in memory usage due to its large fixed memory overhead. We also found that the adaptive learner's prediction may not always be more accurate than the non-adaptive one. In our experiments, VACS can be more accurate than at least one individual learner. VACS can produce better prediction accuracy than both individual learners if the adaptive learner predicts more accurately in relatively high volatility periods while the VFDT-ADWIN performs more accurately in relatively low volatility periods, which is evident in Poker Hand and Forest Cover Type dataset. Results may suggest that VACS is a better choice when users do not know prior information about the stream and cannot choose the appropriate learners, and VACS has the potential to produce the better results than at least one of the learners while taking less training time than the adaptive learner.

In all of our experiments, we don't find evidence suggesting that VACS is worse than both individual learners regarding prediction accuracy. VACS also uses less training time than the adaptive learner in all experimental cases.

We can conclude that VACS successfully addresses the problem stated in Chapter 1. VACS uses stream volatility to select appropriate learners to reduce unnecessary overhead in data stream mining tasks while maintaining high enough prediction accuracy. However, VACS should be used with caution because it may use more memory than both individual learners. From experiments, we also spot some current limitations of VACS which will be discussed in the later chapter.

6.2 Discussion on VACS's Application

In this section, we discuss how to effectively apply VACS in data stream mining tasks after studying the experimental results.

The main purpose of using VACS is reducing unnecessary computational overhead when mining a stream, meanwhile maintaining a high enough prediction accuracy.

Firstly, we discuss the case where we have prior information about the stream. It means that we approximately know the prediction quality, memory usage and time of different learning algorithms in the given stream. In this case, when choosing two component learners for VACS, it is suggested to apply the adaptive learner as the high volatility learner that can produce better prediction accuracy on the stream because of its model adaptation. The low volatility learner is a non-adaptive learner which is a "budget" learner with low time and memory complexity but may predict with worse accuracy when drifts occur. Additionally, there is one preferable property for the low volatility component learner: it is preferred to predict more accurately than the high volatility learner when drifts are not frequent. In this case, VACS can produce better prediction accuracy than both learners.

After starting up, VACS can be applied in the given stream with the volatility changing. The best case of applying VACS is when the stream experiences constant volatility, but occasionally the volatility increases to a high level and then bounces back in a short period. VACS uses the non-adaptive learner when drifts are not frequent, so it does not need to use unnecessary overheads to prepare for adapting the drifts when they do not frequently appear most of the time. When drifts are getting frequent, we switch to the adaptive learner to improve the overall prediction quality. VACS is expected to reduce both time and memory overhead compared with the adaptive learner.

When using VACS, its fixed memory cost should be considered in the case where memory efficiency is the top priority. In the stream where VACS's component learners never consume more memory than its fixed memory usage, VACS cannot take less memory than the adaptive learner.

In the case where the user either has any prior information about the stream or knows behaviours of the adaptive learner and the non-adaptive learner on the given stream, it will be risky for users to choose the adaptive learner since it may not give a better prediction accuracy but still takes more overhead than the non-adaptive learner. In this case, VACS is suggested to be used as a middle choice, because experiments show that VACS uses less training overhead than the adaptive learner. And in all of our experiments, VACS can predict more accurately than at least one of the non-adaptive and the adaptive learners.

6.3 Compare and Contrast between VACS & RCD

In Chapter 2, we reviewed RCD model, which is a learning framework also containing more than one learning models.

RCD and VACS have some similarities. Firstly, both VACS and RCD store more than one learner in its framework when performing learning tasks. Secondly, both frameworks activate one model to learn from the stream and may switch between models.

However, RCD and VACS are different in many aspects. The major difference is that VACS and RCD are used in distinctive contexts and purposes. RCD is applied when there are recurrent concepts in the stream, so RCD can learn multiple concepts and adapt to concept drifts by switching to the corresponding models. The main purpose of using RCD is to improve the prediction accuracy when recurrent concepts appear in the stream. In contrast, VACS is used when the stream has changing volatility. The purpose of VACS is to reduce the training cost while maintaining high enough prediction accuracy. Secondly, VACS and RCD use different criteria to switch among the various models. RCD chooses models based on different stream concepts while VACS chooses models based on different volatility levels. Thirdly, VACS uses different algorithms for each model, and RCD uses the same algorithm for each model with different configurations. This is because VACS mainly benefits from combining natural properties of different algorithms to optimise the learning task. For example, in our case, one algorithm is low-cost, and the other is high-cost. Thus, by combining these two algorithms in the learning task, we can achieve a reduction in cost compared with the application of the high-cost model. In contrast, RCD can perform well because its models are the same algorithm with different configurations. Thus, it can generalise different concepts in the stream and handle recurrent concepts.

We can summarise that even though VACS and RCD are similar because of they use more than one learning model, they are still two distinctive frameworks used in different scenarios.

7 Conclusions

In this research, we contribute a novel framework known as VACS that can reduce computational cost when mining data streams. In this chapter, we summarise the result that has been accomplished in this research.

We also discovered several limitations of the current VACS, and we will discuss them in this chapter. Based on these limitations, we propose two potential improvements for VACS.

7.1 Research Conclusion

The problem that is addressed in this research is to develop a system that can automatically choose the most suitable classifier between adaptive and non-adaptive algorithms in real time when mining a stream with changing volatility. The system should apply the adaptive learner when the volatility is high and the non-adaptive learner when the volatility is low. It reduces the learning costs while maintaining high enough prediction accuracy.

We developed VACS which contains an adaptive (high volatility classifier) and a non-adaptive classifier (low volatility classifier) to learn from the data stream. VACS samples the changing stream volatility levels by using double reservoirs and compares the current volatility level with the sampled volatility levels to decide whether the current volatility level is high or low. Then, VACS select the appropriate classifier based on the comparison.

We tested VACS on both synthetic and real-world data with changing volatility. We used two synthetic data generators: mutating random tree generator and SEA generator. We generated streams with four different patterns of changing volatility. In most experiments on synthetic data, VACS effectively reduces the time and memory cost compared with the adaptive learner. Meanwhile, VACS maintains a high prediction accuracy similar to the adaptive learner. Additionally, VACS switched as expectedly between two classifiers given different stream volatility levels. In the real-world data, we test VACS on three datasets: Poker Hand, Forest Cover Type, and Airline. VACS reduces training time cost compared with the adaptive learner. In the case of Poker Hand and Forest Cover Type datasets, VACS produces better prediction than both adaptive learner and non-adaptive learner, which exceeds our expectation. In all of our experiments, VACS reduces the training time compared with the adaptive learner, and VACS's prediction accuracy is at least better than one of its component learners.

We are confident to conclude that VACS has met our requirements. It chooses the most suitable classifier when learning a stream with volatility changes, and it achieves the computational cost reduction compared with the adaptive learner while maintain a close prediction quality to the adaptive learner. We have shown that stream mining can be improved by considering the stream volatility. To the best of our knowledge, VACS is the first data stream learning framework that combines distinctive learners in order to reduce training overheads.

There are still potential improvements on VACS. We will discuss VACS's limitations and propose our future works on VACS in following sections.

7.2 Limitations

In this section, we discuss some limitations of the current VACS.

The first limitation is that users must specify a volatility measurement window size as a parameter such that VACS can work properly. We have proposed a method to choose the window size in Section 5.3.4. However, it requires an additional test on the stream before applying VACS. Also, the volatility measurement window reacts to the stream volatility with delays which can cause inaccurate measurement of the current volatility level.

The second limitation is that currently VACS only supports equal weights for high volatility and low volatility learners. So it is not possible to assign a higher preference to a particular learner.

Also, VACS has a large fixed memory overhead. This is because VACS uses VFDT-ADWIN and HAT-ADWIN along with other components. It may consume more memory than the adaptive algorithm if VACS's fixed memory usage dominate the overall memory usage. Therefore, this makes VACS's memory reduction conditional.

7.3 Future Works

We identify two potential improvements on VACS at this stage.

One possible improvement is for the volatility measurement window. The window can be replaced by an adaptive window. The window is expected to shrink its size when the volatility changes notably and enlarge its size when the volatility is stable. The first benefit is to enable VACS to decide its window size without the user's input. Secondly, the improvement can also reduce the volatility measurement delay when volatility is changing notably. Because when the volatility largely changes, it can remove the outdated elements by shrinking the window size such that it can react quickly to the changed volatility level. Meanwhile, it is still robust to the random noise because it increases its size to reduce the effect of any noise when the volatility change is not significant. Existing algorithms such as ADWIN [4] have already implemented a sliding window with similar properties to the above description. However, the challenge of applying such a window is an efficiency problem because VACS is mainly designed to reduce computational costs when mining a stream. Such a window will be time and memory efficient such that it will not significantly influence the computational cost reduction of VACS. Thus, to apply the adaptive window in VACS, we will analyse and test different existing adaptive window algorithms to choose one of them appropriately. One alternative suggestion that can mitigate the volatility measurement delay is to use a predictive approach. For instance, one can build a learning model to generalise the volatility change pattern of a stream and signal alarms when the volatility is about to change. Corresponding actions such as reducing the sliding window size can be taken when this alarm is signalled.

The second improvement is to enable VACS to add different weights to each component learners. One way to implement this is to multiply the threshold (the mean of double reservoirs) with a weighting coefficient. The coefficient shall increase the threshold value if we want to assign a higher preference to the high volatility learner and decrease the threshold value if we prefer the low volatility learner.

Bibliography

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

[2] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: the stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 665–665. ACM, 2003.

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

[4] Albert Bifet and Ricard Gavalda. Learning from time-changing data with adaptive windowing. In *SDM*, volume 7, pages 443–448. SIAM, 2007.

[5] Albert Bifet and Ricard Gavaldà. Adaptive learning from evolving data streams. In *International Symposium on Intelligent Data Analysis*, pages 249–260. Springer, 2009.

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

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

[8] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 71–80. ACM, 2000.

[9] Pedro M Domingos and Geoff Hulten. Catching up with the data: Research issues in mining data streams. In *DMKD*, 2001.

[10] Joao Gama. *Knowledge discovery from data streams.* CRC Press, 2010.

[11] Joao Gama, Pedro Medas, Gladys Castillo, and Pedro Rodrigues. Learning with drift detection. In *Brazilian Symposium on Artificial Intelligence*, pages 286–295. Springer, 2004.

[12] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM Computing Surveys (CSUR)*, 46(4):44, 2014.

[13] Paulo Mauricio Gonçalves Jr and Roberto Souto Maior De Barros. RCD: A recurring concept drift framework. *Pattern Recognition Letters*, 34(9):1018–1025, 2013.

[14] Michael Bonnell Harries, Claude Sammut, and Kim Horn. Extracting hidden context. *Machine learning*, 32(2):101–126, 1998.

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

[16] David Tse Jung Huang, Yun Sing Koh, Gillian Dobbie, and Albert Bifet. Drift detection using stream volatility. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 417–432. Springer, 2015.

[17] David Tse Jung Huang, Yun Sing Koh, Gillian Dobbie, and Russel Pears. Detecting volatility shift in data streams. In *2014 IEEE International Conference on Data Mining*, pages 863–868. IEEE, 2014.

[18] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Proceedings of the seventh ACM SIGKDD International Conference on Knowledge Discovery and Data mining*, pages 97–106. ACM, 2001.

[19] Ralf Klinkenberg. Learning drifting concepts: Example selection vs. example weighting. *Intelligent Data Analysis*, 8(3):281–300, 2004.

[20] Ivan Koychev. Gradual forgetting for adaptation to concept drift. Proceedings of ECAI 2000 Workshop on Current Issues in Spatio-Temporal Reasoning,, 2000.

[21] H Mouss, D Mouss, N Mouss, and L Sefouhi. Test of Page-Hinckley, an approach for fault detection in an agro-alimentary production system. In *5th Asian Control Conference, 2004.*, volume 2, pages 815–818. IEEE, 2004.

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

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

[24] W Nick Street and YongSeog Kim. A streaming ensemble algorithm (sea) for large-scale classification. In *Proceedings of the seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 377–382. ACM, 2001.

[25] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[26] Bernard L Welch. The generalization ofstudent's' problem when several different population variances are involved. *Biometrika*, 34(1/2):28–35, 1947.