A Real-time Dynamic Concept Adaptive Learning Algorithm for Exploitability Prediction

A Real-time Dynamic Concept Adaptive Learning Algorithm for Exploitability Prediction

Jiao Yin[a,b], MingJian Tang[c], Jinli Cao[a,*], Hua Wang[d], Mingshan You[e]

[a]*Department of Computer Science and Information Technology, La Trobe University, Melbourne, VIC 3083, Australia*
[b]*School of Artificial Intelligence, Chongqing University of Arts and Sciences, Chongqing 402160, China*
[c]*Huawei Technologies Co. Ltd, Shenzhen 518129, China*
[d]*Institute for Sustainable Industries & Liveable Cities, Victoria University, Melbourne, VIC 3083, Australia*
[e]*College of Information Science and Engineering, Hunan University, Changsha 410082, China*

Abstract

Exploitability prediction has become increasingly important in cybersecurity, as the number of disclosed software vulnerabilities and exploits are soaring. Recently, machine learning and deep learning algorithms, including Support Vector Machine (SVM), Decision Tree, deep Neural Networks and their ensemble models, have achieved great success in vulnerability evaluation and exploitability prediction. However, they make a strong assumption that the data distribution is static over time and therefore fail to consider the concept drift problems due to the evolving system behaviours. In this work, we propose a novel consecutive batch learning algorithm, called Real-time Dynamic Concept Adaptive Learning (RDCAL), to deal with the concept drift and dynamic class imbalance problems existing in exploitability prediction. Specifically, we develop a Class Rectification Strategy (CRS) to handle the 'actual drift' in sample labels and a Balanced Window Strategy (BWS) to boost the minority class during real-time learning. Experimental results conducted on the real-world vulnerabilities collected between 1988 to 2020 show that the overall performance of classifiers, including Neural Networks, SVM, HoeffdingTree and Logistic Regression (LR),

*Corresponding author
Email address: j.cao@latrobe.edu.au (Jinli Cao)

improves over 3% by adopting our proposed RDCAL algorithm. Furthermore, RDCAL achieves state-of-the-art performance on exploitability prediction compared with other concept drift algorithms.

Keywords: Real-time learning, concept drift, class imbalance, class rectification, balanced window

1. Introduction

Background. Tens of thousands of software vulnerabilities are disclosed to the public, posing severe cybersecurity threats to the computer systems of modern organizations [1, 2, 3]. Geographically sensitive vulnerabilities are making the situation worse. In recent years, more and more software provide location-aware services [4, 5], including but not limited to navigation, recommendation and ticket reservation [6, 7]. Despite the provided convenience in many occupations, it also poses severe security threats to both individuals and organizations, due to the possible geographic mapping between cyberspace and real space [8]. People can be tracked with precise location collected by their phone's GPS sensors. Critical and valuable assets like servers, routers and other physical equipment and infrastructure can also be located and physically destroyed.

Considering the relatively limited resources for vulnerability patching and remediation, a critical challenge for security risk management is how to make a reasonable trade-off between coverage and efficiency. For one thing, firms are trying to patch as many disclosed vulnerabilities as possible to provide the highest level of protection. For another, they have to deprioritise most relatively low-risk vulnerabilities and only focus on these high-risk ones. Therefore, one of the most vital factors for effective security risk management is to identify the most likely to be exploited vulnerabilities accurately.

Literature review. In industry practice, most organizations prioritize their remediation efforts overly relying on the Common Vulnerability Scoring System

(CVSS) [1], which provides a score between 0 to 10 as the overall vulnerability assessment from multiple perspectives [9]. However, CVSS has been found to be sub-optimal as being an exploitability indicator. In some cases, it is no better than randomly choosing vulnerabilities to remediate [10, 11].

To complement CVSS, researchers seek to construct machine-learning and deep-learning-based predictive models by leveraging a large collection of multiple open-sourced datasets together [12]. For example, M. Bozorgi et al. adopted an SVM classifier to predict whether vulnerabilities will be exploited within t ($t \geq 0$) days, operating on high dimensional feature vectors extracted from the text fields, time stamps, cross-references and other entries in the existing vulnerability disclosure reports[13]. Paper [14] investigated the binary classification performance of exploitability prediction on a wide range of machine learning (ML) classifiers, including SVM, K-Nearest-Neighbors (KNN), Naive Bayes and Random Forests, achieving the best testing accuracy of 83% accuracy with SVM algorithm on data collected from National Vulnerability Database (NVD)[2] and ExploitDB[3]. In these previous works, text-related features are usually extracted by traditional statistical text processing techniques, such as Term Frequency-Inverse Document Frequency (TF-IDF) algorithm and common word counting, without capturing the context and obtain semantic features at a high level.

With recent advances in Natural Language Processing (NLP) [15, 16], techniques such as word-embedding, sen2vec and Bidirectional Encoder Representations from Transformers (BERT) are employed to extract semantic features from vulnerability descriptions[17, 18], having achieved great success in classification performance. For example, by applying transfer learning to a pre-trained BERT model, [19] achieved an accuracy of 91% in exploitability prediction.

Despite the advances achieved, upon looking closely, it is fair to say that existing approaches make strong assumptions with respect to data distributions.

[1] https://www.first.org/cvss/calculator/3.0

[2] https://nvd.nist.gov/vuln/data-feeds

[3] https://www.exploit-db.com/

In other words, they assume that all available data share the same data distribution, so that use batch learning to train the classifier and adopt hold-out evaluation to evaluate their models in a randomly separated test set. In fact, due to the evolving system behaviours and environments, concept drift exists in data distribution of both vulnerabilities and exploits. As a result, traditional batch learning and hold-out evaluation could lead to an inflated performance because of unveiling unseen data in the future to construct the predictive model.

Challenges. Considering the real-world applications of exploitability prediction, in this work, we conduct concept drift learning and train the classifier incrementally as new data become available. Besides, a prequential-evaluation or an interleaved-test-then-train evaluation mode will be used to assess the real-time performance of the classifier. It means that each new data would be used to test the classifier's performance before training the classifier. Undoubtedly, as an online learning mode, concept drift learning is more in line with the practical application than batch learning or offline learning. It also enables the classifier to capture new concepts when new data arrives. Therefore, it can provide more reliable exploitability prediction performance.

On the other hand, compared with batch learning, predicting exploitability with concept drift learning faces the following challenges due to the dynamic and incomplete nature of the evolving data.

(1) Class label drift problem. A unique trait of vulnerabilities is that their exploitability is chronologically variable. In other words, at one time slice, a vulnerability may be labelled as 'unexploitable', since no corresponding 'published exploits' exists. However, several months or years later, the vulnerability could become 'exploitable' with 'proof-of-concept' exploits available. In batch learning scenario, both vulnerabilities and exploits are collected at a certain date, thus, the time variation factor and label drift problem are ignored by previous studies. However, with concept drift learning, class label drift is a problem that has to be considered when collecting the vulnerabilities and exploits data, evaluating and updating the classifier over

time.

(2) Dynamic class imbalance problem. In batch learning scenario, all data is available and the class imbalance status is static and determined. Therefore, existing solutions, such as resampling samples, generating synthetic samples and penalizing misclassification samples, could be applied directly. However, the magnitude of data imbalance is dynamically changing in the data stream. Probably, the minority class may become the majority class in a certain time slice. Therefore, more flexible and sensitive strategies are urged for handling dynamic class imbalance problem in concept drift learning.

Contribution. In this paper, we propose an online learning algorithm to address the aforementioned two challenges in concept drift learning and improve the practicability and classification performance for the exploitability prediction task. In summary, our main contributions are as follows:

(1) We propose a general online learning algorithm, called Real-time Dynamic Concept Adaptive Learning (RDCAL), consisting of two strategies to improve the performance of concept drift learning. Specifically, a Class Rectification Strategy (CRS) is designed to handle the 'actual drift' in sample labels, along with a Balanced Window Strategy (BWS) dealing with the minority class during real-time learning.

(2) We demonstrate that RDCAL is classifier-agnostic. Experiments show that RDCAL significantly improves the performance of a wide range of classifiers, including Neural Networks, SVM, Decision Tree and Logistic Regression in a consecutive batch learning setting.

(3) We achieve state-of-the-art performance on exploitability prediction in data stream learning scenario, using the proposed RDCAL learning algorithm with a fully-connected Neural Networks (DenseNN) classifier, compared with other five adaptive data stream learning algorithms.

The rest of this paper is organized as follows. Section 2 discusses related literature, followed by a detailed description of the methodology in Section 3.

We then present the exploitability prediction results of the proposed algorithm on real-world data from 1988 to 2020 in Section 4, followed by comparison and discussion with other baseline algorithms. Section 5 concludes the paper and future work.

2. Related work

We treat the exploitability prediction problem as a concept drift leaning problem. This section discusses some of the most important and related literature in stream learning.

2.1. Concept drift

Concept drift is a phenomenon in which the statistical properties of a target domain change over time in an arbitrary way [20]. Given a set of samples at a period of time [0, t], denoted as $S_{0,t} = \{d_0, d_1, \cdots, d_t\}$, where d_i=$\{X_i, y_i\}$ is one data sample or instance observed at time step i, $X_i \in \mathbb{R}^n$ is a feature vector in an n-dimensional feature space $\mathcal{X}$ and y_i is the corresponding label. Let $S_{0,t}$ follows a certain distribution $F_{0,t}(X, y)$, if $F_{0,t}(X, y) \neq F_{t+1,\infty}(X, y)$, concept drift occurs at time step t+1. The term 'concept drift' at time step t could be defined as the change of joint probability of X and y at time step t, denoted as $\exists t : P_t(X, y) \neq P_{(t+1)}(X, y)$ [20]. Considering that $P_t(X, y)$ is determined by two parts as $P_t(X, y)$=$P_t(X) \times P_t(y|X)$, there are three main sources triggering a concept drift.

(1) Virtual drift: $P_t(X) \neq P_{(t+1)}(X)$ while $P_t(y|X)$=$P_{t+1}(y|X)$. Since $P_t(X)$ drift doesn't affect the decision boundary, it has been considered a virtual drift [20, 21].

(2) Actual drift: $P_t(y|X) \neq P_{t+1}(y|X)$ while $P_t(X) = P_{(t+1)}(X)$. When actual drift happens, the actual decision boundary changes. If the classifier cannot update accordingly, the performance will decrease.

(3) Hybrid drift: a mixture of virtual drift and actual drift.

2.2. Concept drift learning

According to when to handle concept drift, there are generally two learning strategies, namely, lazy and active. For the lazy strategy, concept drift learning consists of a drift detection process and a drift adaptation process. When new data arrives, either data-distribution-based or error-rate-based detection algorithms are used to detect the occurrence of concept drift. ADaptive sliding WINdow (ADWIN) [22], for example, is a data-distribution-based detection algorithm, by calculating the absolute value of some statistics over two windows and comparing it with a pre-defined threshold to determine if drift occurs. PageHinkley [23], another example of data-distribution-based method, employs a Page-Hinkley test as a drift detector to monitor the features' magnitude of changes. Kolmogorov-Smirnov Windowing (KSWIN) is a concept change detection method based on the Kolmogorov-Smirnov (KS) statistical test [24]. Drift Detection Method (DDM) [25], Early Drift Detection Method (EDDM) [26] and Drift Detection Method based on Hoeffdings bounds (HDDM) [27] are examples of concept change detection methods based on learner's error rate. Once drift occurs, drift adaptation algorithms will adjust the classifier model accordingly. For active strategy, the classifier updates constantly and incrementally when new data is available.

Generally speaking, the performance of lazy-strategy-based concept drift learning algorithms are more effective than active-strategy-based methods, due to less frequency of updating classifier. However, the drift detection process itself is also resource-consuming. Besides, the performance of lazy strategy algorithms is greatly limited to the sensitivity of drift detection algorithms.

For both learning strategies, basically, regarding how to handle concept drift there are four strategies. Firstly, redesign base classifiers, such as redesigning the nodes of decision tree [28] or the structures of Neural Networks [29]. Secondly, retrain or fine-tune the parameters or hyperparameters of the learner[30, 9]. Thirdly, adaptively change the training set formation methods, such as adjusting training windows, training sample selection strategies and training sample weights adaptively[31, 32, 33]. Lastly, fusion rules or classifier ensemble algo-

rithms are also good choices for drift adaptation [34, 30, 35, 36].

Among these four strategies, ensemble algorithms are the most popular to reach state-of-the-art performance. On the other hand, it also has a higher computational complexity. It is worth noting that these four strategies are not separated from each other. Instead, they are often combined with each other to achieve better performance.

3. Real-time dynamic concept adaptive learning

To avoid either the possible omission of concept drift detection with lazy strategy or the frequent classifier updating with active strategy, we make a trade-off. Specifically, this work adopts a consecutive batch learning strategy as the learning framework for exploitability prediction. With respect to concept drift adaptation, the proposed RDCAL algorithm involves a combination of classifier parameter fine-tuning and training set reformation.

3.1. Consecutive batch learning framework

The workflow of the consecutive batch learning framework adopted by this work is shown in Fig. 1. As a general concept drift learning framework, it is algorithm-agnostic. In other words, the feature extraction algorithms, feature selection algorithms, classifier models and classifier updating strategies used in this framework could be flexibly selected without affecting how the whole framework works. We introduce the involved notations and main processes in the following subsections.

3.1.1. Feature extraction

As shown in Fig. 1, $\mathcal{I}=\{I^{(1)}, \cdots, I^{(k)}, \cdots\}$ is a sequence of consecutive raw input data batches in chronological order. Each data batch $I^{(k)}$ contains N_a raw samples arriving in a time slice $T^{(k)}$. Different feature extraction and feature selection algorithms could be used to extract the numerical features $\mathcal{X}=\{X^{(1)}, \cdots, X^{(k)}, \cdots\}$ from $\mathcal{I}$, where $X^{(k)}=\{X_1^{(k)}, X_2^{(k)}, \cdots, X_{N_a}^{(k)}\}$ is the feature set extracted from the k-th input raw data batch $I^{(k)}$; $X^{(k)} \in \mathbb{R}^{N_a \times n}$; N_a is the

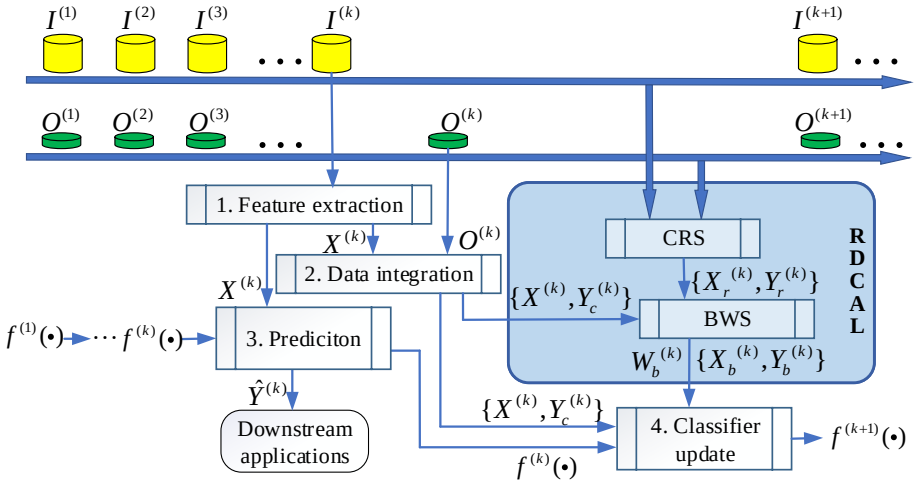

Figure 1: Consecutive batch learning framework.

number of samples in the data batch; n is the dimension of the extracted sample feature; $X_i^{(k)} \in \mathbb{R}^n$ $(i \in [1, N_a])$ is the i-th sample in the data batch $X^{(k)}$.

3.1.2. Prediciton

The notations $f^{(1)}(\cdot)$, $\cdots$, $f^{(k)}(\cdot)$, $\cdots$ in Fig. 1 represents a sequence of different status of the same classifier with different parameters, where $f^{(k)}(\cdot)$ represents the classifier used to predict the output label at the k-th time slice. $f^{(k+1)}$ is the sequential status fine-tuned from $f^{(k)}$ based on the labelled data in the k-th time slice and the learning strategies adopted. If no prior data or knowledge is available, $f^{(1)}(\cdot)$ could be initialized with random parameters. Otherwise, it could be initialized with a pre-trained model.

Once feature $X^{(k)}$ is extracted from raw data batch $I^{(k)}$, the corresponding predicted label $\widehat{Y}^{(k)}$ could be calculated by (1).

$$\widehat{Y}^{(k)} = f^{(k)}(X^{(k)}), (k \geq 1). \tag{1}$$

$\widehat{\mathcal{Y}}$=$\{\widehat{Y}^{(1)}, \widehat{Y}^{(2)}, \cdots, \widehat{Y}^{(k)}, \cdots\}$ represents the sequence of consecutive predicted label batches, which could be used by downstream applications before real labels are available.

3.1.3. Data integration

The notation $\mathcal{O}$=$\{O^{(1)}, \cdots, O^{(k)}, \cdots\}$ in Fig. 1 represents a sequence of raw output data stream, where $O^{(k)}$ is the batch collected within $T^{(k)}$, the same time period with $I^{(k)}$. Note that, the size of $O^{(k)}$ is not necessarily the same as $I^{(k)}$ and the samples in $I^{(k)}$ and $O^{(k)}$ are not in one-to-one correspondence.

Considering the situation in cybersecurity, let $I^{(k)}$= $\{I_1^{(1)}, \cdots, I_{N_a}^{(k)}\}$ be a batch of latest disclosed vulnerabilities within the time period of $T^{(k)}$, where $I_i^{(k)}$ (i=1, $\cdots$, N_a) is the i-th vulnerabilities. $O^{(k)}$ is a batch of exploits published in the same time period $T^{(k)}$. Obviously, $O^{(k)}$ can exploit vulnerabilities in $I^{(k)}$ and other historical vulnerabilities in $I^{(1)}, \cdots, I^{(k-1)}$ as well as some unknown vulnerabilities not included in $\mathcal{O}$. Exploits in $O^{(k)}$ contain the CVE-ID (a globally unique vulnerability identier) information of the exploited vulnerabilities, making it possible to integrate the exploit data batch and the vulnerability data batch. Specifically, if the vulnerabilities in $I^{(k)}$ are exploited by exploits in $O^{(k)}$, the corresponding labels are 1 ('exploitable'), otherwise, are 0 ('unexploitable').

Generally speaking, each raw data in $I^{(k)}$ has a globally unique Sample Identification (SID), which is also inherited by the data in $X^{(k)}$. Through integrating $\mathcal{I}$ and $\mathcal{O}$ with the SIDs of raw data, a sequence of class labels in batches $\mathcal{Y}_c$=$\{Y_c^{(1)}, \cdots, Y_c^{(k)}, \cdots\}$ could be obtained. The subscript c is the capital of 'current', which means the labels are obtained by integrating the current output data batch $O^{(k)}$ collected in the current time period T^k. $Y_c^{(k)}$=$\{Y_{c1}^{(k)}, \cdots, Y_{cN_a}^{(k)}\}$ is the batch labels corresponding to $I^{(k)}$ and $X^{(k)}$. The value of $Y_{ci}^{(k)}$ (i=1, 2, $\cdots$, N_a) is calculated by (2).

$$Y_{ci}^{(k)}(i = 1, 2, \cdots, N_a) = \begin{cases} 1, \text{ if } f_{SID}(X_i^{(k)}) \in f_{SID}(O^{(k)}) \\ 0, \text{ if } f_{SID}(X_i^{(k)}) \notin f_{SID}(O^{(k)}) \end{cases}, \tag{2}$$

where $X_i^{(k)}$ is the i-th data in $X^{(k)}$ and the function of $f_{SID}(\cdot)$ is to find out the appeared SID set.

3.1.4. Classifier update

Let $D_t^{(k)}$=$\{X^{(k)}, Y_c{}^{(k)}\}$ be a labelled training set for the k-th time slice and $W^{(k)} = \mathbf{ones}(N_a, 1)$ be the corresponding batch sample weight. The function $\mathbf{ones}(\cdot)$ means to generate an array according to the specified dimensions, filled with 1. If no learning strategies are applied to optimize the performance of the consecutive batch learning framework, classifier would be updated from $f^{(k)}(\cdot)$ to $f^{(k+1)}(\cdot)$ by fitting $D_t^{(k)}$ with a sample weight of $W^{(k)}$.

Algorithm 1 is the pseudocode of the above-mentioned consecutive batch learning framework. Line 3, 4, 5 and 16 are four main steps executed at each data batch. Lines 6 to 15 related to the RDCAL learning strategy will be covered in Section 3.2.

3.2. Real-time dynamic concept adaptive learning

RDCAL is a general learning strategy used to improve the performance of consecutive batch leaning framework, when existing class label drift and dynamic class imbalance. Specifically, RDCAL employs a Class Rectification Strategy (CRS) to handle the 'actual drift' problem and a Balanced Window Strategy (BWS) to deal with the dynamic class imbalance problem. RDCAL achieves better performance by optimizing the labelled training set $D_t^{(k)}$ and the corresponding sample weight $W^{(k)}$ applied to update the classifier over time.

As shown in Fig. 1, the blue quadrangle named 'RDCAL' is the proposed learning strategy. The specific implementation of RDCAL is listed in line 6-15 of Algorithm 1. It is worth noting that CRS and BWS are optional for RDCAL. In real-word applications, they can be implemented separately or combinedly, depending on the existing problem in the corresponding data stream. However, if both of them are employed, CRS should be applied before BWS. The detailed implementations of CRS and BWS are given in Section 3.3 and 3.4 accordingly.

3.3. Class Rectification Strategy

CRS is designed to handle the class drift problem, which is also described as an 'actual drift' in Section 2.1.

Algorithm 1 Consecutive batch learning framework

Input: $\mathcal{I}$=$\{I^{(1)},\ \cdots,\ I^{(k)},\ \cdots\}$; N_a; $\mathcal{O}$=$\{O^{(1)},\ \cdots,\ O^{(k)},\ \cdots\}$; $f^{(1)}(\cdot)$.

Output: $\mathcal{X}$=$\{X^{(1)},\ \cdots,\ X^{(k)},\ \cdots\}$; $\widehat{\mathcal{Y}}$=$\{\widehat{Y}^{(1)},\ \widehat{Y}^{(2)},\ \cdots,\ \widehat{Y}^{(k)},\ \cdots\}$; $\mathcal{Y}_c$=$\{Y_c^{(1)},\ \cdots,\ Y_c^{(k)},\ \cdots\}$; $\mathcal{D}_t$=$\{D_t^{(1)},\ \cdots,\ D_t^{(k)},\ \cdots\}$; $f^{(2)}(\cdot)$, $f^{(3)}(\cdot)$, $\cdots$, $f^{(k+1)}(\cdot)$, $\cdots$.

$W^{(k)}$=**ones**(N_a,1); $D_t^{(k)}$=$\varnothing$
for each $k \geq 1$ **do**
  Feature extraction: extract and select features $X^{(k)} \in \mathbb{R}^{N_a \times n}$ from $I^{(k)}$
  Prediciton: predict labels $\widehat{Y}^{(k)}$ by calculating $\widehat{Y}^{(k)}$=$f^{(k)}(X^{(k)})$
  Data integration: integrate $X^{(k)}$ and $O^{(k)}$ to obtain $D_t^{(k)} = \{X^{(k)}, Y_c{}^{(k)}\}$
  if RDCAL== True **then**
    if CRS== True **then**
      run Algorithm 2 and get $D_r^{(k)}$
      $D_t^{(k)} = D_t^{(k)} \cup D_r^{(k)}$
    end if
    if BWS== True **then**
      run Algorithm 3 and get $D_b^{(k)}$ and $W_b^{(k)}$
      $D_t^{(k)}$ =$D_b^{(k)}$; $W^{(k)}$=$W_b^{(k)}$
    end if
  end if
  Classifier update: update $f^{(k)}$ to $f^{(k+1)}$ based on $D_t^{(k)}$ and $W^{(k)}$
end for

To adjust the classifier in real-time, the sample labels $Y_c{}^{(k)}$ in the training set $D_t^{(k)}$ of batch k ($k \geq 1$) are the 'current' labels determined by the current output data $O^{(k)}$. Once the label of a sample has changed in a later time, according to a vanilla consecutive batch learning framework, the classifier has no chance to learn from the 'actually drifted' sample, where 'vanilla' means a naive version without any learning strategy, i.e. Algorithm 1 when RDCAL==False.

CRS is a strategy to rectify the label of the historical data. Once a class label drift is detected, the corresponding sample will be added into a rectified set. The rectified set works as a supplement to the original training set to finetune the classifier in real-time. Specifically, as shown in Algorithm 1, in a general consecutive batch learning framework, if RDCAL==True and CRS==True, Algorithm 1 will go to Algorithm 2.

For each k-th time slice, the input for Algorithm 2 includes historical extracted feature batches $\mathcal{X}_h^{(k)} = \{X^{(1)}, \cdots, X^{(k-1)}\}$, historical label batches $\mathcal{Y}_{hc}^{(k)} = \{Y_c^{(1)}, \cdots, Y_c^{(k-1)}\}$, and the current output batch $O^{(k)}$.

To start with, initialize $S{=}\varnothing$ and $D_r^{(k)}{=}\varnothing$, where S is a temporary set to hold the SIDs found in the current output batch $O^{(k)}$; $D_r^{(k)}$ is used to hold all rectified samples in the current time slice. If k equals 1, because no historical samples exist to rectify, CRS returns an empty $D_r^{(k)}$.

When $k > 1$, there are two steps to form a rectified set. Lines 6-11 in Algorithm 2 specify the first step, obtaining all SIDs appeared in current output data $O^{(k)}$. The historical data corresponding to these SIDs may have a class label drift. Step 2, described in lines 13-20 of Algorithm 2, rectifies the historical samples one by one. Once a rectified sample x_h is identified, set its rectified label y_r to '1' and add this rectified sample $\{x_h, y_r\}$ to $D_r^{(k)}$.

Finally, a rectified set $D_r^{(k)}$ for time slice k is returned to Algorithm 1 line 8 and will be merged with the original $D_t^{(k)}$ and from a new $D_t^{(k)}$ as discribed in Algorithm 1 line 9.

Algorithm 2 Class Rectification Strategy

Input: $\mathcal{X}_h^{(k)}$=$\{X^{(1)}, \cdots, X^{(k-1)}\}$; $\mathcal{Y}_{hc}^{(k)}$=$\{Y_c^{(1)}, \cdots, Y_c^{(k-1)}\}$; $O^{(k)}$.

Output: $D_r^{(k)}$.

S=$\varnothing$; $D_r^{(k)}$=$\varnothing$
if k==1 **then**
  break
else
  # step (1): obtain SIDs in $O^{(k)}$.
  for $O_i^{(k)}$ in $O^{(k)}$ **do**
    s=$f_{SID}(O_i^{(k)})$
    if $s \neq \varnothing$ **then**
      S=$S \cup \{s\}$
    end if
  end for
  # step (2): rectify history dataset $\{\mathcal{X}_h^{(k)}, \mathcal{Y}_{hc}^{(k)}\}$.
  for s in S **do**
    for x_h, y_{hc} in **zip**($\mathcal{X}_h^{(k)}$, $\mathcal{Y}_{hc}^{(k)}$) **do**
      if s == $f_{SID}(x_h)$ **and** y_{hc}==0 **then**
        y_r=1
        $D_r^{(k)}$=$D_r^{(k)}$$\cup\{x_h, y_r\}$
      end if
    end for
  end for
end if
return $D_r^{(k)}$

3.4. Balanced Window Strategy

BWS is designed to cope with the dynamic class imbalance problem in stream learning. The basic idea behind BWS is to keep a balanced training set $D_b^{(k)}$ at each time slice k.

Basically, in the vanilla consecutive batch learning framework, the training set $D_t^{(k)}$ used to update classifier from $f^{(k)}(\cdot)$ to $f^{(k+1)}(\cdot)$ is imbalanced, no matter applying CRS or not. To make things worse, the imbalance status within $D_t^{(k)}$ changes irregularly and dynamically over time.

BWS is a strategy to keep a balanced window to dynamically hold the latest N_b negative samples and N_b positive samples as the balanced training set $D_b^{(k)}$ for the current k-th time slice. The class balance size N_b ($N_b \geq N_a$) is a hyperparameter of BWS to control the size of each class within $D_b^{(k)}$.

To keep $D_b^{(k)}$ balanced, the samples belonging to the current minority class will stay more time slices in the balanced window. To avoid the possible over-fitting caused by multiple-times training on the same minority class samples, BWS designs a balanced sample weight $W_b^{(k)}$ to control the training weight of each sample. When it is the first time slice at which a sample occurs in $D_b^{(k)}$, its corresponding balanced sample weight $w_b^{(k)}$ =1. After that, $w_b^{(k)}$ is related to N, the number of time slices that the corresponding sample has stayed in the balanced window. Specifically, $w_b^{(k)}$ can be calculated by (3).

$$w_b^{(k)} = \alpha^{N-1}, (N \geqslant 1), \tag{3}$$

where the time decay factor α ($\alpha \in (0,1]$) is another hyperparameter for BWS.

Since classes in $D_b^{(k)}$ are balanced, it can partially solve the class imbalance problem. $W_b^{(k)}$ is a mechanism to penalize those samples staying too long at $D_b^{(k)}$ to avoid over-fitting. Combining these two factors, BWS provides an effective solution to the dynamic class imbalance problem.

Regarding implementation, as shown in Algorithm 1, in a general consecutive batch learning framework, if RDCAL==True and BWS==True, Algorithm 1 will go to Algorithm 3. For the k-th time slice, the inputs of BWS include

existing training set $D_t^{(k)}$ and sample weight $W^{(k)}$, the class balance size N_b and the time decay factor α.

To start with, if k equals 1, no historical balanced dataset and sample weight are available to work as old data. Therefore, both D_{old} and W_{old} are initialized as $\varnothing$, as shown in line 2 of Algorithm 3. Otherwise, the ${D_b}^{(k)}$ will act as the D_{old} for the next time slice, and ${W_b}^{(k)}$ will decay in a rate α and then work as W_{old} for the next time slice, as shown in line 15 of Algorithm 3.

For each time slice k, as shown in line 4 of Algorithm 3, we first concatenate the old data and current existing data to generate the initial balanced dataset ${D_b}^{(k)}$ and the corresponding sample weight ${W_b}^{(k)}$. Then, check if the sample size of each class in ${D_b}^{(k)}$ is bigger than the pre-set class balance size N_b. If yes, only keep the latest N_b samples and their balanced sample weights for each class, as shown in lines 5-14 in Algorithm 3.

Finally, the ${D_b}^{(k)}$ and ${W_b}^{(k)}$ is returned to Algorithm 1 and worked as the new training set $D_t^{(k)}$ and sample weight $W^{(k)}$ to update the classifier, as shown in lines 12-13 in Algorithm 1.

Algorithm 3 Balanced Window Strategy

Input: $D_t^{(k)}$; $W^{(k)}$; N_b; α.

Output: $D_b^{(k)}$; $W_b^{(k)}$.

if k=1 **then**

  $D_{old} = \varnothing$; $W_{old} = \varnothing$

end if

$D_b^{(k)}$ = concatenate(D_{old}, $D_t^{(k)}$); $W_b^{(k)}$ = concatenate(W_{old}, $W^{(k)}$)

get the indexes of all negative samples $idx0$

if len($idx0 > N_b$) **then**

  $idx0 = idx0$[-N_b : -1]

end if

get the indexes of all positive samples $idx1$

if len($idx1 > N_b$) **then**

  $idx1 = idx1$[-N_b : -1]

end if

$idx = idx0 \cup idx1$

$D_b^{(k)} = D_b^{(k)}[idx]$; $W_b^{(k)} = W_b^{(k)}[idx]$

$D_{old} = D_b^{(k)}$; $W_{old} = \alpha * W_b^{(k)}$

return $D_b^{(k)}$; $W_b^{(k)}$

4. Experimental study

In this part, we validate the performance of RDCAL to learn the real-time dynamic patterns on a real-world vulnerability dataset. First, we set the experiments in Section 4.1. Then, we compare the performance of four classifiers, namely, DensNN, HoeffdingTree, SVM and LR, when applying and without applying RDCAL in Section 4.2, to verify if RDCAL is classifier-agnostic. Furthermore, we compared the performance of RDCAL and other five drift adaptation algorithms on the task of exploitability prediction in Section 4.3. Finally, we analyse the effects of hyperparameters of RDCAL in Section 4.4.

4.1. Experimental setting

4.1.1. Dataset

Data source. We validate RDCAL on a real-world dataset containing 140,758 vulnerabilities disclosed between 1988 and 2020. 23,413 of them have found corresponding exploits in ExploitDB, recognized as positive sample. Specifically, the National Vulnerability Database works as the consecutive raw input data stream $\mathcal{I}$, while ExploitDB provides the consecutive raw output data stream $\mathcal{O}$. The CVE-ID works as the SID to integrate NVD and ExploitDB.

Data stream trend. Fig. 2 shows the monthly number of disclosed vulnerabilities and exploits from 1988 to 2020, demonstrating the number of disclosed vulnerabilities are soaring and much more than available exploits in recent years. Therefore, accurate exploitability prediction is of importance to improve the remediation efficiency through filtering out 'low-risk' vulnerabilities.

Dynamic class imbalance status. When dealing with the collected vulnerabilities and exploits as a real-time data stream, the real-time dynamic class proportion of the current label and rectified label is shown in Fig. 3. The dynamic class proportion is calculated by the Sliding Window Imbalance Factor Technique, proposed by [9], setting the window size $z = 1000$. The current label y_c of each vulnerability is obtained following the formula (2), where Na sets to 1. The

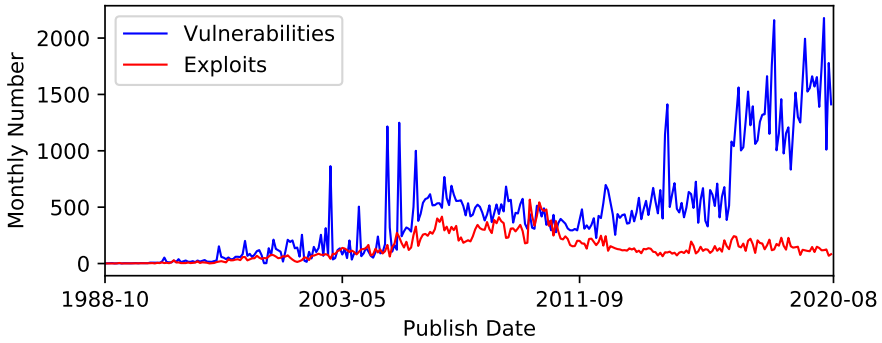

Figure 2: Monthly number of disclosed vulnerabilities and exploits from 1988 to 2020.

rectified label y_r of each vulnerability is obtained following Algorithm 3, where Na is also set to 1.

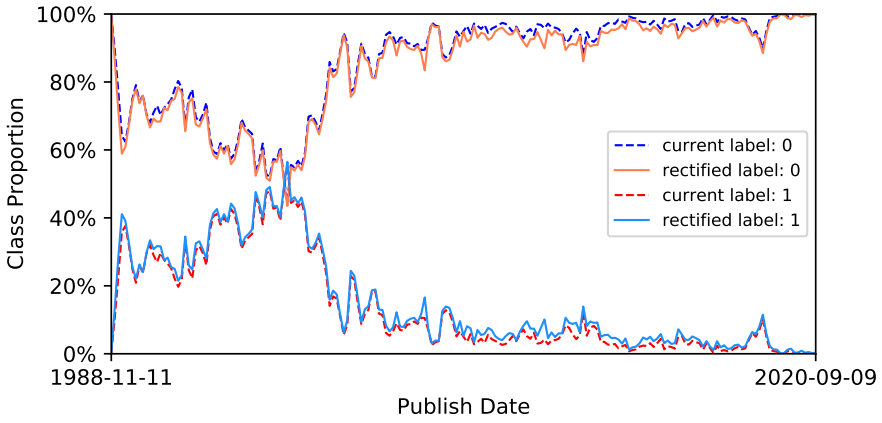

Figure 3: Real-time dynamic class proportion status comparison of current label and rectified label from 1988 to 2020.

On one hand, Fig. 3 shows the dynamic changing trends of class imbalance status. At first, the proportion of class 0 was higher than class 1, and then decreased moderately to about 40%, which is lower than class 1. But, it went up to 80% shortly and then stabilized and in the interval of [80%, 100%]. On the other hand, Fig. 3 illustrates only a small portion of vulnerabilities suffers class drift problem. Therefore the proportion of 'rectified label 1' drawn in blue

line is only a bit higher than the 'current label 1' drawn in red dashed line.

Class rectification. To visualize the real-time class rectified vulnerabilities, we draw Fig. 4. Set $N_a = 1$, and once a vulnerability in $\mathcal{I}$ was disclosed, we will search for the corresponding exploits published in the same current time slice (the time period between current vulnerability and last vulnerability disclosed) in $\mathcal{O}$. These corresponding exploits will be used to rectify all existing vulnerabilities. The horizontal axis in Fig. 4 is the exploit publish date and is also the class rectification date. The vertical axis is the publish date of these rectified vulnerabilities.

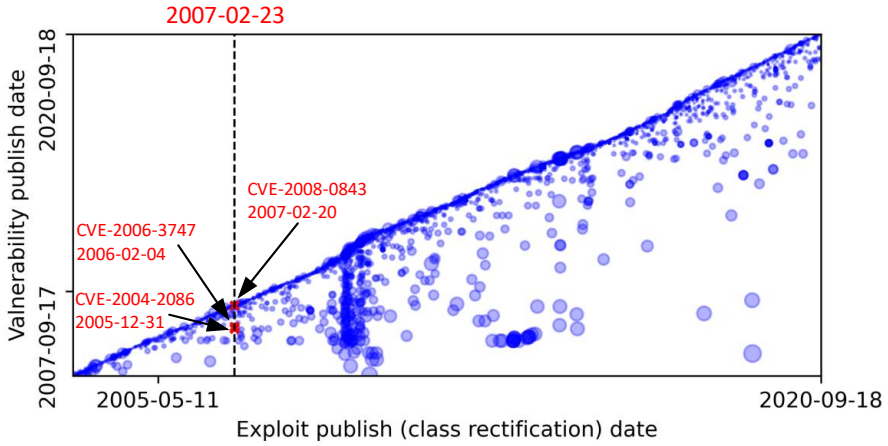

Figure 4: Real-time class rectification results.

Taking the date '2007-02-23' as an example. A vulnerability 'CVE-2007-1083' was published on that date. To obtain the current label, we check all the exploits published on '2007-02-23', E_{list} = ['EXPLOIT-DB:25452', 'EXPLOIT-DB:3362', 'EXPLOIT-DB:3363', 'EXPLOIT-DB:3364', 'EXPLOIT-DB:3365', 'EXPLOIT-DB:3366', 'EXPLOIT-DB:3367', 'EXPLOIT-DB:29641', 'EXPLOIT-DB:29642', 'EXPLOIT-DB:29640', 'EXPLOIT-DB:29643']. Execute $f_{SID}(E_{list})$, all exploited vulnerabilities are found, V_{list}= ['CVE-2005-4832', 'CVE-2006-5276', 'CVE-2006-0549', 'CVE-2005-4832', 'CVE-2007-1133', 'CVE-2007-1130', 'CVE-2007-1131', 'CVE-2007-1126', 'CVE-2007-1124', 'CVE-2007-1127', 'CVE-

2007-1125']. Since 'CVE-2007-1083' is not in the list, we can set its current label as 'unexploitable'. Check the exploitability of vulnerabilities in V_{list}, we find that only three of them, namely 'CVE-2005-4832', 'CVE-2006-0549', 'CVE-2006-5276' are labelled as 'unexploitable'. Therefore, their class labels are rectified from 'unexploitable' to 'exploitable' on '2007-02-23'. Specifically, we mark them with red '$\times$' markers and also annotate their 'CVE-ID' and publish date in Fig. 4.

4.1.2. Evaluation metrics.

As exploitability prediction is binary-classification, we adopt four widely used evaluation metrics, namely, Accuracy, Precision, Recall and F1 score as evaluation metrics. Besides, as we deal with stream data, to further evaluate algorithms over different time slices, the geometric mean (G-mean) is also calculated as an evaluation metric, following [9]. The definition of G-mean is shown in (4),

$$\text{G-mean}(x_1, x_2, \cdots, x_n) = \sqrt[n]{x_1 \times x_2 \times \cdots \times x_n} \tag{4}$$

where x_1, x_2, $\cdots$, x_n are the n elements to calculate the G-mean of them [9].

4.1.3. Feature extraction and selection

We extract features from raw input database NVD. Both vulnerability description and CVSS metrics are available. On one hand, we follow [19] and apply a fine-tuned BERT model to extract semantic features from vulnerability description. On the other hand, we select the identical CVSS V2.0 metrics with [9] as tabular features, applying one-hot encoding to transfer categorical features into one-hot numeric arrays. Finally, to reduce the computational complexity, 10 features from each side are selected via Principal Component Analysis (PCA) to concatenate a 20-dimensional feature set for exploitability prediction.

4.2. RDCAL versus vanilla learning

To verify the effectiveness of RDCAL, we compare the performance of four classifiers, namely, DensNN, HoeffdingTree, SVM and LR, when using RDCAL

and without using RDCAL. Specifically, DensNN is a fully-connected Neural Network with a 10-node-hidden layer, implemented with Keras[4]; HoeffdingTree is an incremental decision tree induction algorithm, implemented with a python package, scikit-multiflow[5]; SVM and LR are two traditional machine learning classifiers, implemented with scikit-learn [6]. The parameters for these classifiers keep default setting. When applying RDCAL, the hyperparameters are set as N_a=200, N_b=200 and α=0.4.

Table 1 summarises the experimental results, where classifier with a '_Vanilla' means using Algorithm 1 without RDCAL, while classifier with a '_RDCAL' means using Algorithm 1 with RDCAL. All classifiers in Table 1 are pre-trained with the first 100 samples, and then are evaluated in an interleaved-test-then-train evaluation mode.

Table 1: Overall performance comparison between Vanilla and RDCAL strategy

Classifier	Accuracy	Precision	Recall	F1 Score	G-mean	△G-mean
DenseNN_Vanilla	89.53%	84.21%	74.03%	78.70%	81.41%	0
DenseNN_RDCAL	90.31%	81.01%	82.88%	81.92%	83.95%	3.12%
HoeffdingTree_Vanilla	89.73%	**84.76%**	74.32%	79.15%	81.78%	0
HoeffdingTree_RDCAL	**90.59%**	81.61%	**83.25%**	**82.37%**	**84.37%**	3.16%
SVM_Vanilla	88.61%	83.03%	70.88%	76.41%	79.45%	0
SVM_RDCAL	89.73%	79.06%	82.88%	80.92%	83.05%	**4.53%**
LR_Vanilla	88.10%	80.99%	71.22%	75.77%	78.77%	0
LR_RDCAL	88.79%	76.49%	82.11%	79.19%	81.52%	3.49%

Values in columns 'Accuracy', 'Precision', 'Recall' and 'F1 Score' are the geometric mean of these metrics over all time-slice. Columns 'G-mean' represents the G-mean of these four metrics. The last column '△G-mean' is calculated by

[4]https://keras.io/api/layers/core_layers/dense/

[5]https://scikit-multiflow.readthedocs.io/en/stable/api/generated/skmultiflow.trees.HoeffdingTreeClassifier.html#skmultiflow.trees.HoeffdingTreeClassifier

[6]https://scikit-learn.org/stable/modules/classes.html

(5).

$$\triangle\text{G-mean} = \frac{\text{G-mean(C\_RDCAL)} - \text{G-mean(C\_Vanilla)}}{\text{G-mean(C\_Vanilla)}} \times 100\% \qquad (5)$$

Where 'C_Vanilla' is the Vanilla learning version with the classifier C and 'C_RDCAL' is the RDCAL learning version.

As shown in Table 1 and Fig. 5, RDCAL can improve the overall performance of all these four classifiers for more than 3%. Especially, for SVM, RDCAL learning strategy makes a significant improvement of 4.53%. Therefore, RDCAL is classifier-agnostic for improving the performance of concept drift learning with class drift problem and dynamic class imbalance problem.

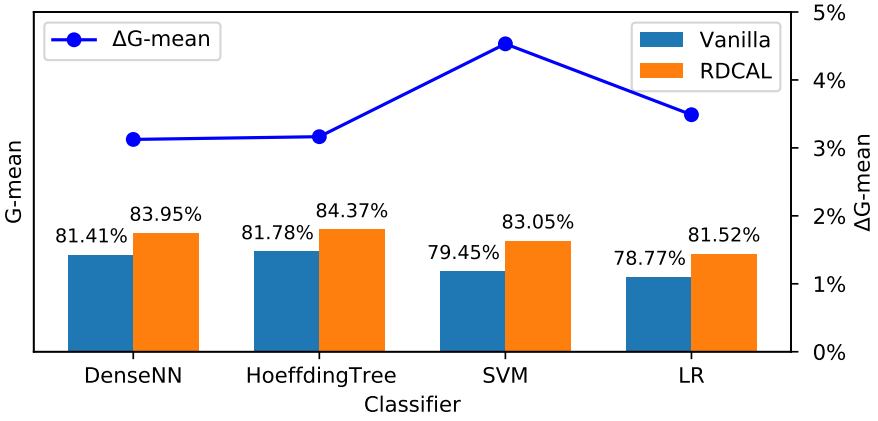

Figure 5: Overall G-mean and △G-mean comparison between Vanilla and RDCAL learning strategy over four different classifiers

Table 1 gives more details. For all these four classifiers, RDCAL can increase the 'Accuracy', 'Recall' and 'F1 Score', but causes a decrease in the 'Precision'. For the exploitability prediction problem, the positive samples are more valuable than negative samples. In other words, 'Recall' is a more important metric than 'Precision'. Therefore, the reduction in 'Precision' caused by the promotion of 'Recall' is acceptable. By comparing the G-mean of these four algorithms, we can see that HoeffdingTree achieves the best performance among both the

'Vanilla' version classifiers and the 'RDCAL' version classifiers. DenseNN takes the second place. LR, limited by its simplicity, performs worst.

Fig. 6 presents the corresponding real-time performance of these 'RDCAL' version classifiers. The publish date starts from '1995-11-01' instead of from 1988, because the first 100 samples are used to pre-train the classifiers and thus are not reported. An interesting result reflected in Fig. 6 is that, in the early years, HoeffdingTree had an obvious advantage over DenseNN. However, in recent years, DenseNN gradually catches up and passes HoeffdingTree. A reasonable guess is that with the accumulation of learning data, the neural network algorithm gradually shows its advantages on learning complex data patterns. Another thing is that the performance of 'Precision', 'Recall' and 'F1 Score' have a similar fluctuating trend with label 1 in Fig. 3.

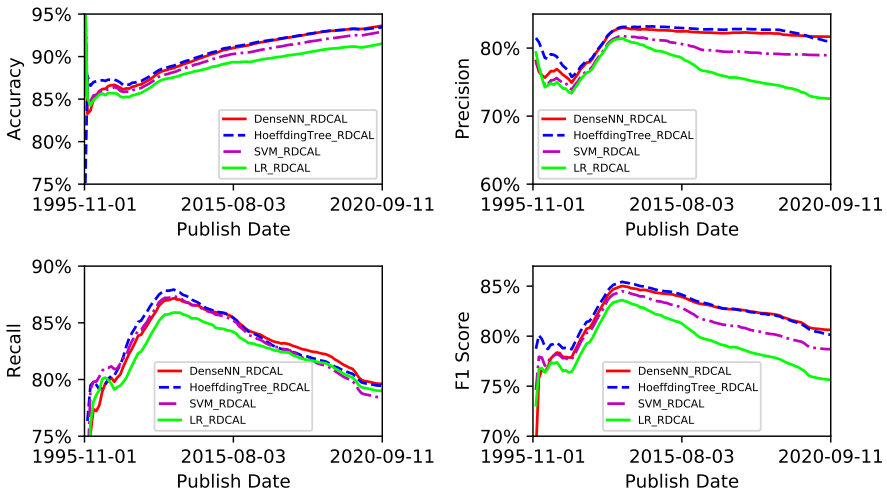

Figure 6: Real-time performance comparison between four different classifiers with RDCAL learning strategy

4.3. RDCAL versus other drift adaptive algorithms

In this section, we compared the performance of RDCAL with five other concept drift adaptation algorithms on the task of exploitability prediction. The experimental setting for all involved algorithms is specified below.

- RDCAL is the proposed consecutive batch learning with RDCAL strategy. The classifier used in this section is DenseNN and hyperparameters are set as N_a=200, N_b=500 and α=0.9.

- SAMKNN is the Self Adjusting Memory [31] model which builds an ensemble with models targeting current or former concepts.

- DWM is the dynamic weighted majority algorithm [34], which keeps a dynamic online weighted learner ensemble by operations like training, weighting, removing and adding base learners to cope with concept drift.

- HTA is the HoeffdingTree Classifier [37] employing ADWIN [22] to detect concept drift and bootstrapping strategy to get better performance.

- LPPNSE is the Learn++.NSE ensemble classifier [35], which is an incremental learning algorithm for all kinds of concept drift, including addition or deletion of concept classes.

- VFDRC is the Very Fast Decision Rules classifier [32], which is an incremental rule learning classifier to adapt with concept drift.

All of the above-mentioned adaptive algorithms except DenseNN are implemented using the python package, scikit-multiflow[7] in this paper.

The average results of 10 times of independent experiments are shown in Table 2 and Fig. 7. Regarding the overall G-mean, RDCAL performs best at 83.20±0.05%. The second place is HTA at 82.16±0.46%. Next are SAMKNN and DWM, achieving 81.35±0.07% and 81.05±0.09%. Both LPPNSE and VFDRC have poor performance on this task, only obtaining around 74% of G-mean. As for single metric, Table 2 shows that RDCAL achieves the best Recall at 86.05±0.10% and best F1 Score at 81.11±0.05% with a large margin above others. However, HTA achieves the best Accuracy with a small advantage over others. It also achieves the best Precision at 81.52±0.59%.

[7] https://scikit-multiflow.readthedocs.io/en/stable/api/api.html

Table 2: Overall performance comparison between RDCAL and other five drift adaptation algorithms

Algorithms	Accuracy	Precision	Recall	F1 Score	G-mean
RDCAL	89.48±0.03%	76.72±0.06%	**86.05±0.10%**	**81.11±0.05%**	**83.20±0.05%**
SAMKNN	89.19±0.04%	81.04±0.06%	76.83±0.11%	78.86±0.08%	81.35±0.07%
DWM	88.22±0.12%	75.30±0.24%	82.52±0.08%	78.73±0.09%	81.05±0.09%
HTA	**89.57±0.19%**	**81.52±0.59%**	78.26±1.18%	79.74±0.61%	82.16±0.46%
LPPNSE	84.73±0.12%	70.77±0.34%	71.20±0.16%	70.98±0.18%	74.19±0.17%
VFDRC	84.38±0.75%	69.63±2.08%	72.51±2.12%	70.98±1.09%	74.14±0.99%

Since HTA is an improvement based on HoeffdingTree, we can compare the performance of HTA with HoeffdingTree_Vanilla and HoeffdingTree_RDCAL in Table 1. The overall G-mean of HoeffdingTree_Vanilla is 81.78%, lower than HTA at 82.16%, showing the effectiveness of HTA in improving the performance of HoeffdingTree classifier. However, compared with the G-mean of HoeffdingTree_RDCAL at 84.37%, it is obvious that RDCAL has a much better effect on improving the performance of HoeffdingTree.

Fig. 7 shows the real-time performance on these four metrics. Consistent with Table 2, RDCAL, SAMKNN, DWM achieve equivalent best performance on Accuracy; HTA and SAMKNN are the best on Precision; RDCAL alone achieves the best Recall with an overwhelming advantage; RDCAL and HAT achieve equivalent good performance on F1 score, followed by SAMKNN and DWM. Although, in recent years, the proportion of class 1 goes down sharply to about 10%, we can see that the Recall of RDCAL is still quite stable, compared with other algorithms.

Both Table 2 and Fig. 7 demonstrate that RDCAL is the best concept adaptation algorithm in this exploitability prediction task. RDCAL only adopt a single classifier during the whole online learning process. However, its performance is even better than ensemble algorithms with multiple classifiers, such as SAMKNN, DWM and LPPNSE.

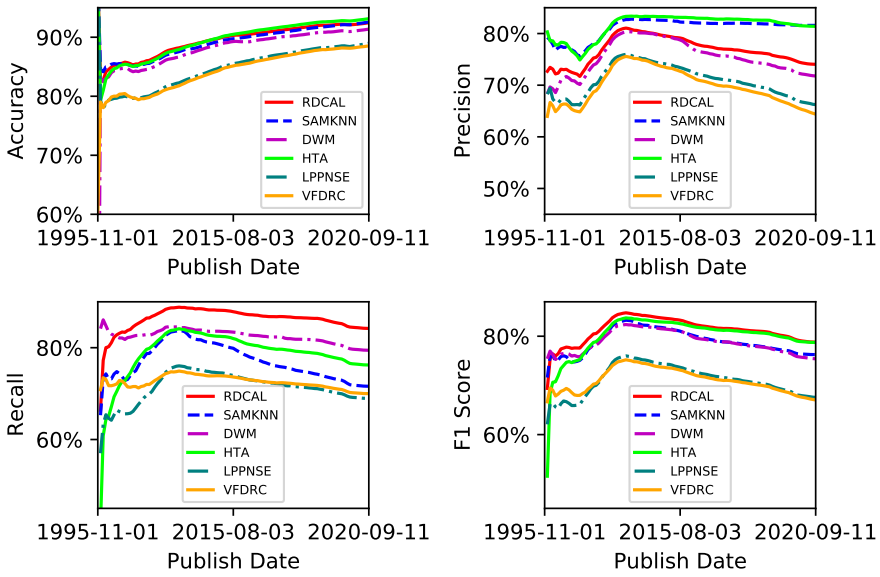

Figure 7: Real-time performance comparison between RDCAL and other five drift adaptation algorithms

4.4. Hyperparameter influence analysis

The parameters of classifiers can be learnt from data automatically. However, the hyperparameters of RDCAL, namely, N_a, N_b and α, should be elaborately adjusted. As shown in Table 1, DenseNN_RDCAL achieves G-mean of 83.95% by setting N_a=200, N_b=200 and α=0.4. However, in Table 2, with the same DenseNN classifier, RDCAL only achieves 83.20% on G-mean, when setting N_a=200, N_b=500 and α=0.9. Therefore, in this section, we discuss how these three hyperparameters affect the performance of RDCAL. Besides, since RDCAL consists of two optional learning strategies, CRS and BWS, we also discuss the effect of them separately.

Therefore, we study CRS and BWS separately under different settings of N_a, N_b and α in the following subsections. All experiments adopt the same consecutive batch learning framework described in Algorithm 1, employing an identical DenseNN as the classifier. Baseline is Algorithm 1 with neither CRS nor BWS, setting the consecutive batch size N_a to 200.

4.4.1. Class Rectification Strategy and N_a

To discuss the effect of CRS and N_a, we conduct a series of experiments to learn the performance of CRS when N_a traverses in [50, 100, 200, 500, 1000].

The the real-time performance of different settings is shown in Fig. 8. We can see that all settings adopting CRS have quite similar performance. Although Baseline has a higher Precision, solutions with CRS are much better in terms of Accuracy, Recall and F1 Score, regardless the value of N_a. Therefore, CRS alone is useful in improving the performance of concept drift learning.

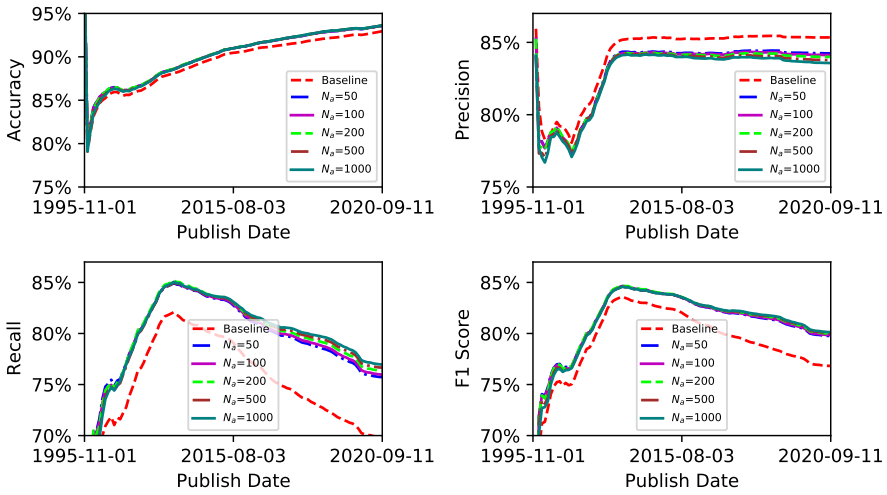

Figure 8: Real-time performance comparison of CRS with different N_a

Table 3 shows the overall performance comparison. When adopting CRS, N_a=200 achieves the best Accuracy, Recall, F1 Score and the overall G-mean. All other N_a settings achieve over 2% improvement in G-mean than Baseline.

Fig. 9 shows the G-mean and ΔG-mean of CRS with different N_a. Obviously, the best performance is achieved when N_a=200, where G-mean is 83.31$\pm$0.07% and ΔG-mean is 2.21%.

4.4.2. Balanced Window Strategy and N_b

To discuss the effect of BWS and N_b, α is fixed to 1. Experiments are designed to compare the performance of BWS when N_b traverses in [50, 100,

Table 3: Overall performance comparison of CRS with different N_a

N_a	Accuracy	Precision	Recall	F1 Score	G-mean	ΔG-mean
Baseline	89.56±0.22%	**84.02±0.24%**	74.52±0.90%	78.89±0.51%	81.56±0.44%	0
50	90.19±0.05%	83.06±0.07%	79.11±0.18%	80.99±0.11%	83.23±0.10%	2.12%
100	90.18±0.02%	83.01±0.10%	79.18±0.15%	80.99±0.05%	83.24±0.04%	2.13%
200	**90.21±0.03%**	82.96±0.05%	**79.40±0.15%**	**81.08±0.08%**	**83.31±0.07%**	**2.21%**
500	90.15±0.03%	82.81±0.08%	79.32±0.13%	80.97±0.06%	83.21±0.05%	2.09%
1000	90.12±0.02%	82.68±0.06%	79.36±0.11%	80.91±0.05%	83.17±0.04%	2.04%

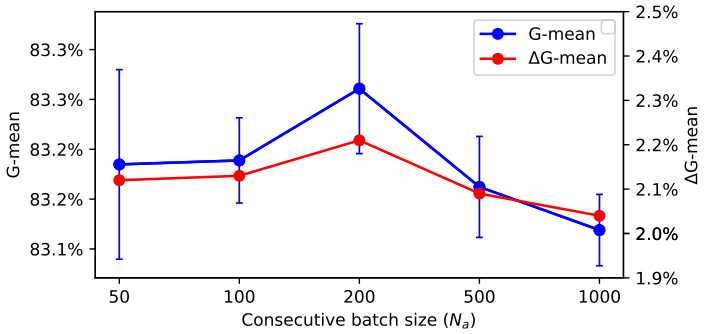

Figure 9: Performance improvement comparison of CRS with N_a

200, 500, 1000].

Fig. 10 shows the real-time performance comparison of BWS with different selections of N_b. Baseline wins the best Accuracy and Precision, but gets the worst Recall. As for the F1 Score, Baseline is also among the best. Different settings distinguished each other at the beginning, and then the gaps were gradually narrowing down with the two classes became much more balanced. However, in recent years, the gaps began to widen due to the severe class imbalance.

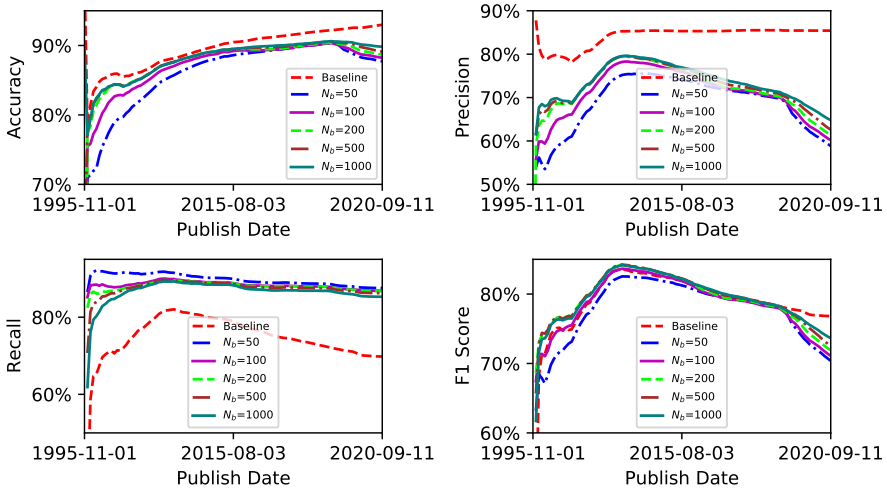

Figure 10: Real-time performance comparison of BWS with different N_b ($\alpha = 1$)

Table 4 shows the overall performance comparison. N_b=500 achieves the best F1 Score at 79.33±0.09% and the overall G-mean at 81.64±0.08%. Baseline performs the best on Accuracy and Precision, while N_b=50 achieves the best Recall. With respect to ΔG-mean, only N_b=500 and N_b=1000 have a weak advantage over Baseline. Considering their extraordinary performance on Recall, the reason should be the over-fitting to minority class. BWS will keep the minority class samples in the balanced window more than one time slice, therefore they will be used to update the classifier more than once. Without a time decay weight, it is very likely to result in over-fitting.

Fig. 11 shows the G-mean and ΔG-mean of BWS with different N_b. The

Table 4: Overall performance comparison of BWS with different N_b ($\alpha = 1$)

N_b	Accuracy	Precision	Recall	F1 Score	G-mean	ΔG-mean
Baseline	**89.58±0.11%**	**84.17±0.13%**	74.32±0.42%	78.83±0.24%	81.52±0.21%	0
50	86.00±0.80%	68.44±1.76%	**89.73±1.55%**	77.50±0.47%	79.97±0.47%	-1.90%
100	87.30±0.12%	70.70±0.23%	88.32±0.25%	78.48±0.07%	80.87±0.06%	-0.80%
200	87.89±0.07%	72.02±0.16%	87.97±0.08%	79.16±0.10%	81.48±0.09%	-0.05%
500	88.15±0.11%	72.64±0.26%	87.44±0.23%	**79.33±0.09%**	**81.64±0.08%**	**0.14%**
1000	88.26±0.14%	73.21±0.27%	86.51±0.13%	79.29±0.13%	81.59±0.13%	0.09%

best performance is achieved when N_b=500, which is only 0.14% better than Baseline. Therefore, it is vital to set an appropriate time decay factor to weaken the influence of old data when using BWS.

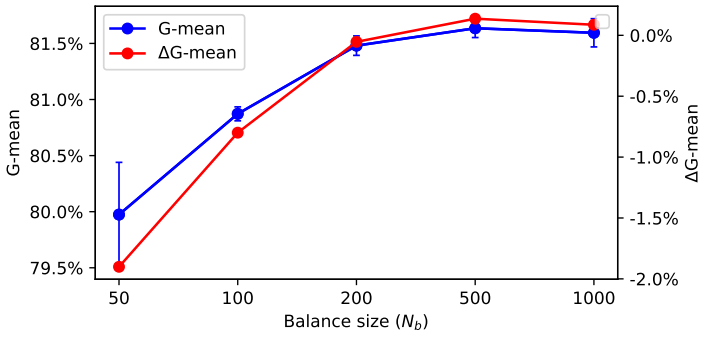

Figure 11: Performance improvement comparison of BWS with different N_b ($\alpha = 1$)

4.4.3. Balanced Window Strategy and α

Similarly, to discuss the effect of BWS and α, N_b is fixed to 500. Experiments are designed to compare the performance of BWS when α traverses in [0.3, 0.5, 0.7, 0.8, 0.9, 1].

Fig. 12 shows the real-time performance comparison of BWS with different selections of α. In terms of Accuracy, all settings except for α=1 achieves equivalent results. For Precision and Recall, the results are quite different. Generally speaking, settings achieving good precision usually have poor performance on

Recall and vice versa. Performance on F1 Score is highly related to the class proportion. For example, when class 1 and class 0 are almost half to half, all settings have good results, while when the two classes are highly imbalanced, different settings can make a big difference.

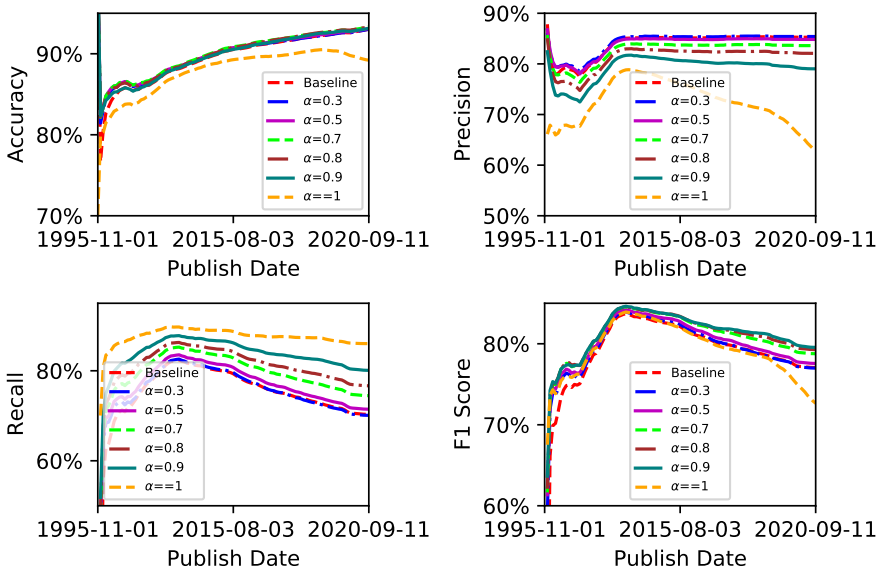

Figure 12: Real-time performance comparison of BWS with different α (N_b=500)

Table 5 shows the overall performance comparison. We can see that the best overall G-mean is achieved by α=0.9, which is 2.19% better than Baseline. Followed by α=0.8 and 0.7. The worst is α=1, which means that BWS only keeps a balanced window but not weakens sample weights overtime. As we analysed before, in this case, the performance can decrease because of over-fitting to the minority class. α=0.3 only gets limited improvement due to the too fast time decay of minority samples in the balanced window.

Fig. 13 shows the G-mean and ΔG-mean of BWS with different values of α. The best performance is achieved with α=0.9, which is 2.19% better than the Baseline.

According to Tables 3, 4 and 5, the best N_a is 200, the best N_b is 500 and the best α is 0.9. However, these results are obtained by ablation studies, which

Table 5: Overall performance comparison of BWS with different α (N_b=500)

α	Accuracy	Precision	Recall	F1 Score	G-mean	ΔG-mean
Baseline	89.54±0.12%	83.91±0.50%	74.46±0.35%	78.74±0.15%	81.47±0.14%	0
0.3	89.82±0.05%	**84.23±0.09%**	75.37±0.20%	79.50±0.14%	82.06±0.11%	0.72%
0.5	89.95±0.02%	83.76±0.08%	76.64±0.08%	79.99±0.03%	82.44±0.02%	1.19%
0.7	**90.09±0.03%**	82.57±0.06%	79.14±0.05%	80.77±0.05%	83.04±0.04%	1.93%
0.8	90.01±0.03%	81.28±0.07%	80.82±0.09%	81.00±0.06%	83.19±0.06%	2.11%
0.9	89.78±0.03%	79.15±0.06%	83.32±0.10%	81.14±0.05%	**83.25±0.05%**	**2.19%**
1	87.98±0.23%	72.28±0.61%	**87.45±0.42%**	79.11±0.19%	81.44±0.18%	-0.03%

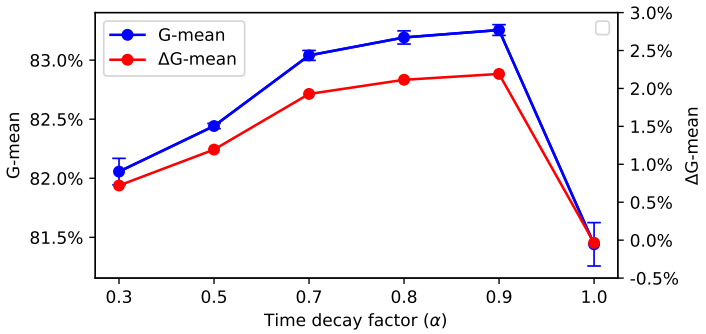

Figure 13: Performance improvement comparison of BWS with different α (N_b=500)

means only one factor is applied to the consecutive batch learning framework each time. When applying CRS and BWS at the same time, the best choice for these three hyperparameters should be tested simultaneously.

4.5. Concluding remarks

We validate the effectiveness of RDCAL by three series of experiments. First, the experiments in Section 4.2 demonstrate that RDCAL can improve the performance of four different classifiers in a consecutive batch learning scenario by more than 3%. Then, the experiments in Section 4.3 show that RDCAL performs the best among six concept drift adaptation algorithms. Finally, we discuss the influence of hyperparameters on RDCAL and demonstrate the effectiveness of CRS and BWS separately. Therefore, RDCAL is indeed classifier-agnostic and a state-of-the-art concept drift learning algorithm in dealing with the exploitability prediction problem.

5. Conclusion

We propose a novel Real-time Dynamic Concept Adaptive Learning algorithm under a consecutive batch learning setting. Specifically, RDCAL consists of two strategies, namely, Class Rectification Strategy (CRS) and Balanced Window Strategy (BWS). CRS is designed to handle the 'actual drift' in sample labels and BWS is a strategy to deal with the dynamic class imbalance problem.

Our comprehensive experiments show that RDCAL can significantly improve the performance of a wide range of classifiers, including Neural Networks, SVM, Decision Tree and Logistic Regression in exploitability prediction. Besides, RDCAL achieves state-of-the-art performance on a real-world dataset containing 140,758 vulnerabilities, compared with other five adaptive data stream learning algorithms.

The effectiveness of concept drift adaptation algorithms, including RDCAL, highly depends on the characteristics of data. In the future, we will further explore the generalization of RDCAL to applications in other domains.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

The work of Jiao Yin was partially supported by the Science and Technology Research Program of Chongqing University of Arts and Sciences, China (Grant No. P2020RG08), the Science and Technology Research Program of Chongqing Municipal Education Commission of China (Grant No. KJ202001344049955) and the Natural Science Foundation of Chongqing Science and Technology Commission, China (Grant No. cstc2019jcyj-msxmX034).

References

[1] M. Tang, J. Yin, M. Alazab, J. C. Cao, Y. Luo, Modelling of extreme vulnerability disclosure in smart city industrial environments, IEEE Transactions on Industrial Informatics (2020) 1–1.

[2] M. Tang, M. Alazab, Y. Luo, Big data for cybersecurity: Vulnerability disclosure trends and dependencies, IEEE Transactions on Big Data 5 (3) (2019) 317–329.

[3] E. Zhang, F.-H. Liu, Q. Lai, G. Jin, Y. Li, Efficient multi-party private set intersection against malicious adversaries, in: Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop, 2019, pp. 93–104.

[4] N. A. H. Haldar, J. Li, M. Reynolds, T. Sellis, J. X. Yu, Location prediction in large-scale social networks: an in-depth benchmarking study, The VLDB Journal 28 (5) (2019) 623–648.

[5] J. Li, K. Deng, X. Huang, J. Xu, Analysis and applications of location-aware big complex network data, Complexity 2019.

[6] Q. Zeng, M. Zhong, Y. Zhu, J. Li, Business location selection based on geo-social networks, in: International Conference on Database Systems for Advanced Applications, Springer, 2020, pp. 36–52.

[7] P. Vimalachandran, H. Liu, Y. Lin, K. Ji, H. Wang, Y. Zhang, Improving accessibility of the australian my health records while preserving privacy and security of the system, Health Information Science and Systems 8 (1) (2020) 1–9.

[8] E. Zhang, M. Li, S.-M. Yiu, J. Du, J.-Z. Zhu, G.-G. Jin, Fair hierarchical secret sharing scheme based on smart contract, Information Sciences 546 (2020) 166–176.

[9] J. Yin, M. Tang, J. Cao, H. Wang, M. You, Y. Lin, Adaptive online learning for vulnerability exploitation time prediction, in: Web Information Systems Engineering – WISE 2020, Springer, 2020, pp. 252–266.

[10] J. Ruohonen, A look at the time delays in cvss vulnerability scoring, Applied Computing and Informatics 15 (2) (2019) 129–135.

[11] J. Jacobs, S. Romanosky, I. Adjerid, W. Baker, Improving vulnerability remediation through better exploit prediction, Journal of Cybersecurity 6 (1) (2020) tyaa015.

[12] M. Alazab, M. Tang, Deep Learning Applications for Cyber Security, Springer, 2019.

[13] M. Bozorgi, L. K. Saul, S. Savage, G. M. Voelker, Beyond heuristics: learning to classify vulnerabilities and predict exploits, in: Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2010, pp. 105–114.

[14] M. Edkrantz, A. Said, Predicting cyber vulnerability exploits with machine learning., in: SCAI, 2015, pp. 48–57.

[15] J. Huang, M. Peng, H. Wang, J. Cao, W. Gao, X. Zhang, A probabilistic method for emerging topic tracking in microblog stream, World Wide Web 20 (2) (2017) 325–350.

[16] H. Jiang, R. Zhou, L. Zhang, H. Wang, Y. Zhang, Sentence level topic models for associated topics extraction, World Wide Web 22 (6) (2019) 2545–2560.

[17] E. R. Russo, A. Di Sorbo, C. A. Visaggio, G. Canfora, Summarizing vulnerabilities descriptions to support experts during vulnerability assessment activities, Journal of Systems and Software 156 (2019) 84–99.

[18] J. Du, S. Michalska, S. Subramani, H. Wang, Y. Zhang, Neural attention with character embeddings for hay fever detection from twitter, Health information science and systems 7 (1) (2019) 21.

[19] J. Yin, M. Tang, J. Cao, H. Wang, Apply transfer learning to cybersecurity: Predicting exploitability of vulnerabilities by description, Knowledge-Based Systems (2020) 106529doi:https://doi.org/10.1016/j.knosys.2020.106529.

[20] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, G. Zhang, Learning under concept drift: A review, IEEE Transactions on Knowledge and Data Engineering 31 (12) (2018) 2346–2363.

[21] S. Ramírez-Gallego, B. Krawczyk, S. García, M. Woźniak, F. Herrera, A survey on data preprocessing for data stream mining: Current status and future directions, Neurocomputing 239 (2017) 39–57.

[22] A. Bifet, R. Gavalda, Learning from time-changing data with adaptive windowing, in: Proceedings of the 2007 SIAM international conference on data mining, SIAM, 2007, pp. 443–448.

[23] J. Gama, R. Sebastião, P. P. Rodrigues, On evaluating stream learning algorithms, Machine learning 90 (3) (2013) 317–346.

[24] C. Raab, M. Heusinger, F.-M. Schleif, Reactive soft prototype computing for concept drift streams, Neurocomputing.

[25] J. Gama, P. Medas, G. Castillo, P. Rodrigues, Learning with drift detection, in: Brazilian symposium on artificial intelligence, Springer, 2004, pp. 286–295.

[26] M. Baena-Garcıa, J. del Campo-Ávila, R. Fidalgo, A. Bifet, R. Gavalda, R. Morales-Bueno, Early drift detection method, in: Fourth international workshop on knowledge discovery from data streams, Vol. 6, 2006, pp. 77–86.

[27] I. Frías-Blanco, J. del Campo-Ávila, G. Ramos-Jimenez, R. Morales-Bueno, A. Ortiz-Díaz, Y. Caballero-Mota, Online and non-parametric drift detection methods based on hoeffdings bounds, IEEE Transactions on Knowledge and Data Engineering 27 (3) (2014) 810–823.

[28] A. Bifet, R. Gavaldà, Adaptive learning from evolving data streams, in: International Symposium on Intelligent Data Analysis, Springer, 2009, pp. 249–260.

[29] J. Yin, M. You, J. Cao, H. Wang, M. Tang, Y.-F. Ge, Data-driven hierarchical neural network modeling for high-pressure feedwater heater group, in: Australasian Database Conference, Springer, 2020, pp. 225–233.

[30] J. Montiel, J. Read, A. Bifet, T. Abdessalem, Scikit-multiflow: A multi-output streaming framework, Journal of Machine Learning Research 19 (72) (2018) 1–5.
URL `http://jmlr.org/papers/v19/18-251.html`

[31] V. Losing, B. Hammer, H. Wersing, Knn classifier with self adjusting memory for heterogeneous concept drift, in: 2016 IEEE 16th international conference on data mining (ICDM), IEEE, 2016, pp. 291–300.

[32] P. Kosina, J. Gama, Very fast decision rules for classification in data streams, Data Mining and Knowledge Discovery 29 (1) (2015) 168–202.

[33] S. Ren, B. Liao, W. Zhu, Z. Li, W. Liu, K. Li, The gradual resampling ensemble for mining imbalanced data streams with concept drift, Neurocomputing 286 (2018) 150–166.

[34] J. Z. Kolter, M. A. Maloof, Dynamic weighted majority: An ensemble method for drifting concepts, Journal of Machine Learning Research 8 (Dec) (2007) 2755–2790.

[35] R. Elwell, R. Polikar, Incremental learning of concept drift in nonstationary environments, IEEE Transactions on Neural Networks 22 (10) (2011) 1517–1531.

[36] H. Li, Y. Wang, H. Wang, B. Zhou, Multi-window based ensemble learning for classification of imbalanced streaming data, World Wide Web 20 (6) (2017) 1507–1525.

[37] G. Hulten, L. Spencer, P. Domingos, Mining time-changing data streams, in: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, 2001, pp. 97–106.