PEARL: Probabilistic Exact Adaptive Random Forest with Lossy Counting for Data Streams

Ocean Wu(✉), Yun Sing Koh, Gillian Dobbie, and Thomas Lacombe

The University of Auckland, Auckland, New Zealand
hwu344@aucklanduni.ac.nz, ykoh@cs.auckland.ac.nz,
{g.dobbie,thomas.lacombe}@auckland.ac.nz

Abstract. In order to adapt random forests to the dynamic nature of data streams, the state-of-the-art technique discards trained trees and grows new trees when concept drifts are detected. This is particularly wasteful when recurrent patterns exist. In this work, we introduce a novel framework called PEARL, which uses both an exact technique and a probabilistic graphical model with Lossy Counting, to replace drifted trees with relevant trees built from the past. The exact technique utilizes pattern matching to find the set of drifted trees, that co-occurred in predictions in the past. Meanwhile, a probabilistic graphical model is being built to capture the tree replacements among recurrent concept drifts. Once the graphical model becomes stable, it replaces the exact technique and finds relevant trees in a probabilistic fashion. Further, Lossy Counting is applied to the graphical model which brings an added theoretical guarantee for both error rate and space complexity. We empirically show our technique outperforms baselines in terms of cumulative accuracy on both synthetic and real-world datasets.

Keywords: Random Forest · Recurring concepts · Concept drift

1 Introduction

Many applications deal with data streams. Data streams can be perceived as a continuous sequence of data instances, often arriving at a high rate. In data streams, the underlying data distribution may change over time, causing decay in the predictive ability of the machine learning models. This phenomenon is known as concept drift. For example, in a weather prediction model when there is a shift from one season to another, one may observe a decrease in prediction accuracy.

Moreover, it is common for previously seen concepts to recur in real-world data streams [3,5,12]. If a concept reappears, for example a particular weather pattern, previously learnt classifiers can be reused; thus the performance of the learning algorithm can be improved. Current techniques that deal with recurrent concept drifts [2,7] are exact methods, relying on meta-information. A drawback of these techniques is their approach to memory management in storing the classifier pool.

© Springer Nature Switzerland AG 2020
H. W. Lauw et al. (Eds.): PAKDD 2020, LNAI 12085, pp. 17–30, 2020.
https://doi.org/10.1007/978-3-030-47436-2_2

These methods create a new classifier after each drift, which leads to a large pool of concepts being created early in the process. This makes identifying previously seen concepts or states more expensive.

Motivated by this challenge, we propose a novel approach for capturing and exploiting recurring concepts in data streams using both probabilistic and exact techniques, called Probabilistic Exact Adaptive Random Forest with Lossy Counting (PEARL). We use an extended Adaptive Random Forest (ARF) as the base classifier [6]. Like ARF, it contains a set of foreground trees, each with a drift detector to track warnings and drifts, and a set of background trees that are created and trained when drift warnings are detected. Beyond that, we keep all the drifted foreground trees in an online tree repository and maintain a set of candidate trees. The candidate trees are a small subset of the online tree repository, which can potentially replace the drifted trees. When the drift warning is detected, a set of candidate trees are selected from this online tree repository by either the State Pattern Matching technique or the Tree Transition Graph. Once the actual drift is detected, the foreground trees are replaced by either their background trees or the more relevant candidate trees. In addition, we periodically update the Tree Transition Graph using a Lossy Counting [8] approximation. The benefit of Lossy Counting is the ability to control the size of the graph and expire less frequently used trees, thus adapting the graph to the dynamic environment of the evolving data.

The main contribution of this paper is a novel framework for storing and reusing concepts effectively and efficiently, by using both an exact pattern matching technique and a probabilistic graphical model. In particular, the graphical model uses the Lossy Counting approximation for improved performance and guaranteed space complexity. It is shown empirically to outperform baselines in terms of cumulative accuracy gains.

The remainder of this paper is organized as follows: in Sect. 2 we provide an overview of work related to recurring concepts. In Sect. 3 we give an overview of the PEARL framework. In Sects. 4 and 5 we discuss the implementations of the State Pattern Matching and the Tree Transition Graph in detail, followed by a theoretical analysis in Sect. 6. We then evaluate the performance of our techniques on both synthetic and real world datasets in Sect. 7. Finally, Sect. 8 concludes our work and poses directions for future work.

2 Related Work

There has been some research using probabilistic methods for concept drift adaptation to find recurrent patterns. RePro [14] was proposed to predict concept models for future drifts using historical trends of concept change. The authors use a Markov Chain to model the concept history and have shown that this allows the learner to adjust more quickly to concept change in streams with recurring concepts. Chen et al. [4] proposed a novel drift prediction algorithm to predict the location of future drift points based on historical drift trends which they model as transitions between stream volatility patterns. The method uses a

probabilistic network to learn drift trends and is independent of the drift detection technique. ProChange [9] finds both real and virtual drifts in unlabelled transactional data streams using the Hellinger distance, and models the volatility of drifts using a probabilistic network to predict the location of future drifts. The GraphPool framework [1] refines the pool of concepts by applying a merging mechanism whenever necessary by considering the correlation among features. Then, they compare the current batch representation to the concept representations in the pool using a statistical multivariate likelihood test. All of these techniques use an exact mechanism for managing the number of transitions from one model to another, which is computationally expensive.

3 PEARL Overview

PEARL extends Adaptive Random Forest (ARF), a classification algorithm for evolving data streams that addresses the concept drift problem. In ARF, there are two types of trees, namely the foreground trees and the background trees. The **foreground trees** get trained, and make individual predictions. The majority of the individual predictions forge the final prediction output of the ARF (*i.e.* voting). Additionally, each foreground tree is equipped with two drift detectors, one for detecting drift warnings and the other for detecting actual drifts. The **background trees** are created and start training from the root when the foreground trees have detected drift warnings. When actual drifts are detected, the background trees then replace the drifted foreground trees. The background trees do not participate in voting until they replace the drifted foreground trees.

In ARF, the drifted foreground trees simply get discarded when they get replaced by the background trees. This is particularly wasteful since the same concepts may recur. Besides, the drift warning period may be too short for the background trees to model the recurred concept. In contrast to ARF, PEARL stores drifted foreground trees in an **online tree repository**, and tries to reuse these trees to replace the drifted foreground trees when concepts recur. However, as the size of the repository can grow as large as the memory allows, it is computationally expensive to evaluate all the repository trees, to find relevant trees to the potentially recurring concepts. As a result, PEARL introduces a third type of tree, called the **candidate trees**, to the random forest. The candidate trees are a small subset of the repository trees. They are potentially the most relevant trees to the next recurring concept drift. Similar to the background trees, the candidate trees may replace the drifted foreground trees, and they do not participate in voting until such replacements happen.

In the PEARL framework, both the background and the candidate trees perform predictions individually during the drift warning period. When actual drifts are detected, PEARL tries to replace the drifted foreground trees with either the background trees or the best performing candidate trees, based on the κ statistics of their individual prediction performance during the drift warning period. Figure 1 gives an overview of the PEARL framework.

To find candidate trees efficiently and accurately, PEARL uses a combination of two techniques: an exact technique called the **State Matching Process**,

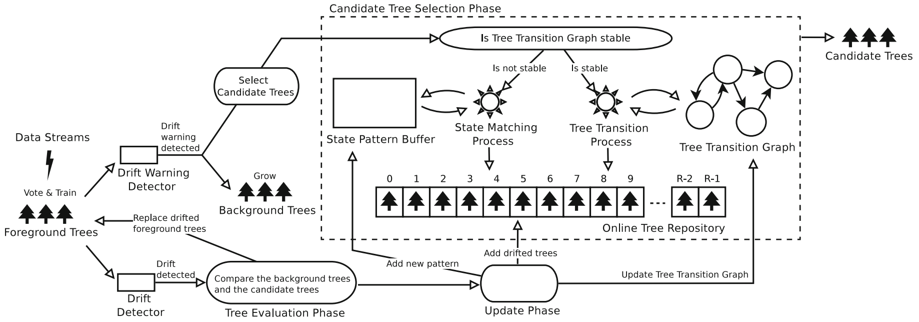

Fig. 1. PEARL overview

and a probabilistic technique called the **Tree Transition Process**, as shown in the Candidate Tree Selection Phase in Fig. 1. Initially, the state matching process finds candidate trees by matching patterns stored in a State Pattern Buffer. Each of the patterns represents a set of trees in the online repository, that co-occurred in predictions in the past. Meanwhile, PEARL constructs a probabilistic graphical model that captures the tree replacements when drifts are detected, as shown in the Update Phase in Fig. 1. When the graph model becomes stable, *i.e.*, when the reuse rate of candidate trees surpasses a user set threshold, the state matching process is replaced by the tree transition process to find potential candidate trees. Conversely, when the graph model becomes unstable, PEARL switches back to the state matching process. Details of the two processes are presented in Sects. 4 and 5.

4 State Matching Process

In this section we detail the state matching process for finding candidate trees, as shown in Fig. 1. This process is used to find exact match for a set of trees that co-occurred in predictions in the past, and is relevant again when an old concept reappears. This process is the basis of the tree transition process in Sect. 5.

A **state pattern** is a snapshot of the status of the repository trees during a stable period *i.e.*, the interval between two consecutive drift detections. Such a snapshot is represented by a sequence of bits, denoting whether each of the repository trees is functioning as a foreground tree during the stable period. Formally, let $b_0, b_1, b_2, \ldots, b_{R-1}$ be a sequence of bits, with R being the size of the online tree repository. Each tree in the online tree repository is associated with an ID in the range of $[0, R-1]$, which corresponds to its position in the bit sequence. Here $b_i = 1$ and $b_i = 0$ denote whether the repository tree with $\texttt{ID} = i$ is functioning as a foreground tree or not, respectively (Note: $b_i = 0$ can also mean the repository has not yet allocated a spot for a tree with $\texttt{ID} = i$). For example, a pattern 0100110 indicates that the size of the online tree repository is 7 (*i.e.*, $R = 7$). The repository trees with IDs equal to 1, 4 and 5 are functioning as foreground trees, since the bits at the corresponding positions are set.

State Pattern Construction. Initially, the online tree repository allocates spots for all the foreground trees. These foreground trees get allocated the first few tree IDs, and they are represented by the first few bits in the state pattern accordingly. For example, if $R = 7$ and the random forest consists of 3 foreground trees, the state pattern is then initialized to 1110000.

The state pattern gets updated during the Update Phase as shown in Fig. 1. Following the last example, suppose we detect the very first drift on the foreground tree with $\texttt{ID} = 0$. Firstly, the pattern is updated by turning the first bit off. There are no candidate trees when the very first drift is detected, therefore we simply replace it with its background tree that started growing when the drift warning was detected. The background tree then gets allocated the next available spot in the online tree repository, as well as getting assigned a new $\texttt{ID} = 3$ which corresponds to the allocated spot. As a result, the state pattern is updated to 0111000.

At a later stage when there are candidate trees outperforming the background trees, we replace the drifted foreground trees with the candidate trees. Similar to the last example, the pattern is updated by first turning the bits representing the drifted trees off, followed by turning on the bits corresponding to the IDs of the candidate trees. Otherwise, just like the last example, a background tree gets assigned a new ID, as well as getting allocated a spot in the online tree repository corresponding to the newly assigned ID.

The updated state pattern is added to a State Pattern Buffer, which keeps all the updated states, with a user defined limit on size. Each of the state patterns is associated with the frequencies of it being added. To control the size of the buffer, the state patterns are evicted by the Least Recently Used (LRU) policy.

State Pattern Matching. Pattern matching is used to find candidate trees from the online tree repository by comparing the current state pattern with all the patterns in the State Pattern Buffer. The choice of the best matching state pattern follows three requirements: (A) the bits representing the drifted trees must be unset; (B) the pattern must have the lowest edit distance to the current state pattern; (C) the maximum edit distance must not be greater than a user defined threshold $\theta \in (d, 2|F|]$ with d being the number of drifted trees, and F being the number of foreground trees. For patterns that satisfy all the above three requirements, the one with the highest frequency gets matched.

The set bits in the matched pattern, that are unset in the current state pattern, are the IDs of the repository trees to be added to the set of candidate trees. For instance, suppose the current pattern is 010011 and $\theta = 1$. A drift warning is detected on the foreground tree with $\texttt{ID} = 1$, and the state pattern list contains the following 3 patterns with their frequencies f: (1) 010011 with $f = 5$; (2) 100101 with $f = 4$; (3) 001011 with $f = 1$. The edit distance for each pattern is 0, 2, 1, respectively.

Following rule (A), pattern 1 is not matched since the bit corresponding to the drift tree ID is set. Following rules (B) and (C), pattern 2 is not matched despite having a higher frequency than pattern 3, since its edit distance is higher.

As a result, pattern 3 is matched as its edit distance is no greater than θ. In this case, the repository tree with $\texttt{ID} = 2$ is added to the set of candidate trees.

5 Tree Transition Process

In the state matching process, the total number of patterns is 2^R with R being the size of the online tree repository. The patterns are evicted by the LRU scheme since it is infeasible to store all the patterns in memory. Relevant patterns may be thrown away due to memory constraints. By introducing a graph that models the tree transitions during the pattern matching process, we can retain more information with an addition of polynomial space complexity. As data streams evolve over time, some trees in the online repository may loose relevancy, while new trees are built and stored. Therefore we apply Lossy Counting on the graphical model to adapt to changes in the underlying distribution. In addition, Lossy Counting reduces the space complexity to logarithmic with an error guarantee.

Tree Transition Graph. The *Tree Transition Graph* is a directed graph $G = (V, E)$. The set of nodes $V_0, V_1, \ldots, V_{R-1}$ represents the tree IDs of the individual trees stored in the online tree repository of size R. A directed edge $(u, v) \in E$ stands for the foreground tree with $\texttt{ID} = u$ is replaced by a repository tree with $\texttt{ID} = v$, when a drift is detected on the foreground tree u. The edge weight $W(u, v)$ describes the number of times that u transitions to v. The graph is updated during the Update Phase (Fig. 1). A drifted foreground tree adds an outgoing neighbour when either its background tree or a candidate tree replaces it. If such a transition already exits in E, $W(u, v)$ is incremented by 1. If a background tree replaces the drifted tree and gets added to the online tree repository, a new node representing the background tree is added to the graph first. The Tree Transition Graph becomes stable and replaces the state matching process when the last d drifted trees have a reuse rate $\frac{c}{d}$ over a user defined $\delta \in (0, 1)$, where c denotes the number of candidate trees replacing the d drifted trees. When the foreground tree with $\texttt{ID} = u$ detects a drift warning, it is replaced by one of its outgoing neighbours randomly. The probability of a transition (u, v) is $\frac{W(u,v)}{\sum_{\forall (u,i) \in E} W(u,i)}$ where $i \in V$.

Lossy Counting. The Lossy Counting algorithm computes approximate frequency counts of elements in a data stream [11]. We apply this technique to approximate the edge weights in the Tree Transition Graph, to improve its summarization of the probabilities of tree transitions, under constant changes in the underlying distribution of the data stream.

Lossy Counting may be triggered when a drift warning is detected, *i.e.*, before either the state matching and the tree transition processes. Given an error rate ϵ, the window size of Lossy Counting is $l = \frac{1}{\epsilon}$, meaning the Lossy Counting is performed after every l drift warning trees. At the window boundaries, all the edge weights get decremented by 1. If an edge weight becomes 0, the edge is

removed from E. This may lead to nodes becoming isolated, *i.e.*, when both out-degree and in-degree is 0. Such a node is removed from G and its corresponding repository tree also gets deleted. An example is given in Fig. 2.

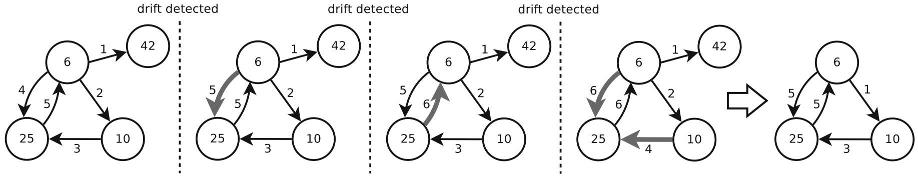

Fig. 2. Lossy Counting on the Tree Transition Graph with $l = 3$. The bold edges denote tree transitions.

Lossy Counting can potentially mitigate the side effects of undesired transitions. If a node u incorrectly transits to a less performant neighbour v and adds the corresponding repository tree v to the candidate trees, the foreground tree u is more likely to be replaced by its background tree instead, when an actual drift is detected. In this case, a new background tree v' is added to the online tree repository, and its representation node v' and edge (u, v') are added to the graph. However at the next window boundary of Lossy Counting, this newly added node and the corresponding tree are removed due to low edge weight.

6 Theoretical Analysis

First we discuss the Lossy Counting guarantees. Let N be the number of trees with drift warnings, $\epsilon \in (0, 1)$ be the user specified error rate, and s be the support threshold of the edges such that $s > \epsilon$, where $(u, v) \in E$ if $W(u, v) > (s - \epsilon)N$. The window size is $\frac{1}{\epsilon}$. The edge weight count is underestimated by at most ϵN, *i.e.*, $\frac{N}{window\ size} = \frac{N}{1/\epsilon} = \epsilon N$. There are no false negatives. If the tree transition is genuinely frequent, $(u, v) \in E$ since $\frac{W(u,v)}{N} > \epsilon$. False positives may occur before the edge weight decay. It is guaranteed to have true edge weight at least $(s - \epsilon)N$. Secondly we discuss the memory analysis. The memory size for pattern matching is $O(R \cdot P)$, with R being the size of online tree repository, and P being the capacity of the State Pattern Buffer. The Tree Transition Graph without Lossy Counting introduces an additional polynomial space complexity $O(R \cdot |E|)$ where $|E|$ has an upper bound of $R(R-1)$. With Lossy Counting the space complexity is guaranteed to be reduced by $\frac{1}{\epsilon} \log(\epsilon N)$ [11]. As a result, the Tree Transition Graph with Lossy Counting is $O(\frac{1}{\epsilon} \log(\epsilon N) \cdot R \cdot |E|)$ in terms of space complexity. Finally we discuss the runtime. One transition in the Tree Transition Graph only takes $O(R)$ as the transition is randomly selected. One execution of State Pattern Matching takes $O(N \cdot L)$ where L is a user defined variable denoting the maximum number of patterns that can be stored in the State Pattern Buffer.

7 Experimental Evaluation

We evaluate the performance of PEARL by classification performance (accuracy and kappa statistics), runtime and memory usage. The classification performance is based on prequential evaluation. In our implementation, both the State Pattern Matching and the Tree Transition Graph can be either turned on or off for evaluation. We compare these approaches with ARF, which is simply PEARL having both the State Pattern Matching and Tree Transition Graph turned off. Additionally, we compare with the State Pattern Matching only PEARL(PO).

The experimentation is performed on the following machine and architecture: Dell PowerEdge R740 with 40 CPUs, 125.42 GiB (Swap 976.00 MiB) and Ubuntu 18.04 with AMD64 4.15.0-46-generic. Our code, synthetic dataset generators and test scripts are available here[1] for reproducible results. In our experiments, the size of foreground and candidate trees is set to 60 each, and the size of the online tree repository is set to 9600. The accuracy/kappa sample frequency is set to 1000 data instances.

Datasets. We use both synthetic and real world datasets in our experiments. The synthetic data sets include recurrent abrupt and gradual drifts. The sequence of concepts are generated by the Poisson distribution with $\lambda = 1$. The abrupt and gradual drift width are set to 1 and 3000 data instances, respectively. Each of the drift types are generated with either 3 or 6 concepts, denoted as Agrawal 3 or Agrawal 6 in the tables. The real world datasets have been thoroughly used in the literature to assess the classification performance of data stream classifiers: Covertype, Electricity, Airlines[2], and Rain[3].

Accuracy and Kappa Evaluation. We ran all experiments ten times with varying seeds for every synthetic dataset configuration. The set of parameters for PEARL has been well tuned although more optimal sets may exist. Agrawal generator is used, since it is the most sophisticated synthetic data generator involving classification functions with different complexities. We generate 400,000 instances for each of the Agrawal datasets.

Apart from measuring the accuracy and kappa mean, we calculate the cumulative gain for both accuracy and kappa against ARF over the entire dataset *e.g.*, $\sum$((accuracy(PEARL) − accuracy(ARF)). A positive value indicates that the PEARL approach obtained a higher accuracy/kappa as compared to the baseline ARF. The cumulative gain does not only track the working characteristics over the course of the stream, which is important for evaluating new solutions to dynamic data streams [10]; but also takes into account the performance decay caused by model overfitting at the drift points. We notice that with our technique, we obtained higher accuracy/kappa mean and positive cumulative gain values compared with ARF (Table 1).

[1] https://github.com/ingako/PEARL.
[2] https://moa.cms.waikato.ac.nz/datasets.
[3] https://www.kaggle.com/jsphyg/weather-dataset-rattle-package.

Table 1. Accuracy and kappa mean, cumulative accuracy gain and kappa gain (%)

Dataset	Drift type	Accuracy mean			Kappa mean		
		ARF	PEARL (PO)	PEARL	ARF	PEARL (PO)	PEARL
Agrawal 3	Abrupt	78.50 ± 0.42	87.93 ± 0.51	88.60 ± 0.55	27.38 ± 2.68	65.63 ± 2.66	68.74 ± 1.00
	Gradual	78.09 ± 0.85	86.90 ± 1.04	87.25 ± 1.10	29.36 ± 2.98	67.64 ± 1.58	68.78 ± 1.88
Agrawal 6	Abrupt	77.35 ± 0.63	86.02 ± 0.70	86.13 ± 0.80	26.27 ± 1.87	64.34 ± 1.39	64.55 ± 1.57
	Gradual	76.94 ± 0.30	84.83 ± 0.74	85.11 ± 0.88	28.11 ± 2.10	64.18 ± 1.19	64.73 ± 1.39
Dataset	Drift type	Accuracy gain			Kappa gain		
		–	PEARL (PO)	PEARL	–	PEARL (PO)	PEARL
Agrawal 3	Abrupt	–	3763.90 ± 262.78	4027.70 ± 221.49	–	15259.90 ± 1637.55	16503.57 ± 1058.39
	Gradual	–	2633.44 ± 234.39	2737.71 ± 265.92	–	11443.73 ± 936.03	11786.61 ± 925.50
Agrawal 6	Abrupt	–	3457.69 ± 355.08	3502.30 ± 354.55	–	15187.32 ± 856.60	15271.74 ± 889.47
	Gradual	–	3158.37 ± 283.48	3267.47 ± 319.81	–	14429.91 ± 1083.39	14648.21 ± 1139.64

NOTE: PEARL (PO) is PEARL with pattern matching only. The ARF columns have been removed from the gain tables, since its cumulative accuracy/kappa is subtracted by both PEARL (PO) and PEARL, for calculating gains.

Memory and Runtime Evaluation. Similar to scikit-multiflow [13] we estimate memory consumption, by adding up the key data structures used by PEARL and PEARL (PO) (Table 2).

Table 2. Memory and runtime

Dataset	Drift type	Memory (KB)		Runtime (min)		
		PEARL (PO)	PEARL	ARF	PEARL (PO)	PEARL
Agrawal 3	Abrupt	166.10 ± 4.26	347.49 ± 157.69	166.63 ± 11.33	217.92 ± 9.54	218.09 ± 12.65
	Gradual	109.88 ± 27.65	243.07 ± 40.52	157.63 ± 36.63	198.89 ± 42.70	199.13 ± 42.95
Agrawal 6	Abrupt	172.26 ± 4.23	274.08 ± 110.19	202.34 ± 74.92	276.96 ± 89.85	276.04 ± 87.77
	Gradual	165.09 ± 2.21	301.44 ± 10.92	177.35 ± 9.30	230.67 ± 17.57	216.04 ± 14.24

Lossy Counting Evaluation. Table 3 shows that with Lossy Counting, PEARL consistently obtains a higher gain in both accuracy while consuming less memory across 10 different seeds, comparing to Lossy Counting disabled. The State Matching parameters are fixed while the Tree Transition parameters have been tuned differently on each seed. We examine both 0%, 33% and 66% concept shifts for evaluating the performance under decaying concepts. 33% concept shift means 33% of expiring concepts after an interval (*i.e.*, a number of data instances). For example, if there are 3 concepts we are transitioning between, a 33% concept shift will expire 1 concept while adding 1 new concept after an interval. Each interval is constituted of predetermined concepts appearing according to a Poisson distribution of $\lambda = 1$. For each type of concept shift, we generate a total of 300,000 data instances with 2 concept shifts happening at positions 200,000 and 250,000. Table 3 shows that with concept shifts, the graph with Lossy Counting was able to gain a higher accuracy and kappa performance within a similar amount of time, while consuming less memory.

Table 3. Agrawal gradual concept shift

Shift type	w/lossy counting	Acc. gain	Kappa gain	Memory (KB)	Runtime (min)
0%	False	2927.21 ± 187.76	12177.98 ± 885.27	376.08 ± 17.28	254.19 ± 3.67
	True	3065.14 ± 142.82	12435.13 ± 885.75	279.96 ± 57.25	257.27 ± 2.16
33%	False	1924.44 ± 221.70	8490.98 ± 709.92	312.27 ± 8.86	209.92 ± 5.76
	True	2153.76 ± 263.04	8899.14 ± 811.47	277.18 ± 19.20	209.03 ± 7.04
66%	False	1697.06 ± 435.99	8077.79 ± 1156.34	312.28 ± 16.12	229.90 ± 30.49
	True	2017.86 ± 477.47	8700.64 ± 1129.36	277.46 ± 20.34	221.94 ± 33.57

Case Study. To better understand the benefits of the probabilistic graphical model and Lossy Counting, we performed a case study on a 400,000 instance dataset generated by the Agrawal data generator with abrupt drifts on 3 concepts. We evaluate the three different configurations of PEARL: State Pattern

Matching only, Tree Transition Graph only, and Tree Transition Graph with LC (Lossy Counting). Both of the Tree Transition Graph configurations include State Pattern Matching as it is the basis of graph construction.

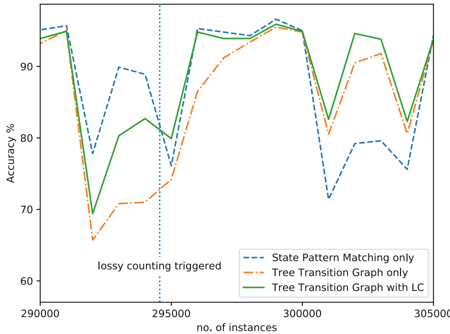

Fig. 3. Accuracy

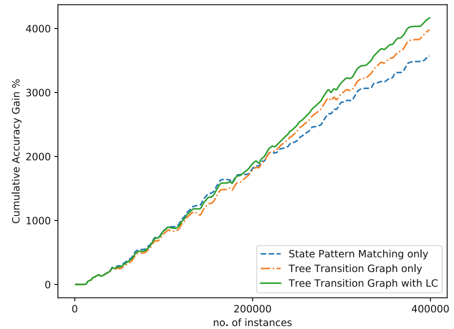

Fig. 4. Cumulative accuracy against ARF

Figure 3 is an example of an execution of Lossy Counting. In this case, before Lossy Counting was triggered, both the Tree Transition Graph only and the Tree Transition Graph with Lossy Counting configurations show a dip in accuracy. This is due to undesired graph transitions, and these undesired transitions happen more frequently when there are too many similar outgoing neighbours. However after an execution of Lossy Counting, the graph with Lossy Counting recovers faster than the graph only configuration. During this recovery period, the pattern matching process was triggered for the Tree Transition Graph with Lossy Counting because some nodes were isolated after the Lossy Counting removed rarely used outgoing neighbours. After that, the graph with Lossy Counting configuration starts to utilize the Tree Transition Graph, and it continues to outperform the Tree Transition Graph only configuration. Intuitively, there are two factors which contribute to such performance: firstly, the brief pattern matching activation after the Lossy Counting helps with the graph construction, which strengthens the stability of the graph; secondly, confusing similar outgoing neighbours were removed by Lossy Counting. Figure 4 captures the accuracy gain of the three types of configurations of PEARL against ARF. All configurations show a continuous gain in accuracy, but the graph with Lossy Counting tends to gain the most accuracy over time.

Real World Dataset. Table 4 shows the accuracy/kappa mean and cumulative gain values. In these experiments, we used ECPF with Hoeffding Tree (*i.e.* ECPF(HT)) and ECPF with Adaptive Random Forest (*i.e.* ECPF (ARF)). We noticed that our technique has positive gains on three out of the five datasets.

Table 4. Accuracy and kappa mean, cumulative accuracy and kappa gain (%)

Dataset	#instances	Accuracy mean				Kappa mean			
		ARF	ECPF (HT)	ECPF (ARF)	PEARL	ARF	ECPF (HT)	ECPF (ARF)	PEARL
Electricity	45,312	86.04 ± 3.08	85.94 ± 3.60	88.43 ± 2.68	86.13 ± 3.17	68.53 ± 7.71	70.69 ± 7.43	75.87 ± 5.58	68.93 ± 7.33
Rain	142,193	93.24 ± 2.06	96.25 ± 2.66	93.93 ± 1.66	93.75 ± 1.99	78.30 ± 7.98	88.79 ± 7.50	81.47 ± 3.98	80.17 ± 7.75
Airlines	539,383	66.07 ± 5.65	65.56 ± 5.79	65.42 ± 6.12	66.85 ± 5.43	15.59 ± 11.55	17.81 ± 10.22	19.00 ± 9.37	19.16 ± 11.64
Covtype	581,012	88.77 ± 5.89	86.53 ± 5.05	88.48 ± 5.16	90.12 ± 5.41	74.28 ± 15.38	73.32 ± 9.37	76.85 ± 10.53	77.43 ± 14.36
Pokerhand	829,012	74.07 ± 6.25	73.78 ± 6.48	90.07 ± 6.76	90.08 ± 9.00	26.72 ± 17.41	39.52 ± 15.28	77.28 ± 17.33	72.89 ± 27.58
Dataset	#instances	Accuracy gain				Kappa gain			
		–	ECPF (HT)	ECPF (ARF)	PEARL	–	ECPF (HT)	ECPF (ARF)	PEARL
Electricity	45,312	–	−7	**106**	4	–	91	**326**	18
Rain	142,193	–	**424**	96	73	–	**1481**	446	265
Airlines	539,383	–	−271	−343	**419**	–	1195	1843	**1924**
Covtype	581,012	–	−1311	−181	**783**	–	−576	1475	**1830**
Pokerhand	829,012	–	−248	13261	**13267**	–	10588	**41922**	38271

NOTE: The ARF columns have been removed from the gain tables, since its cumulative accuracy/kappa is subtracted by ECPF and PEARL, for calculating gains.

8 Conclusions and Future Work

We presented a novel framework, PEARL, that handles recurrent concept drifts by capturing the recurrent concepts using probabilistic and exact approaches. PEARL uses an extended random forest as a base classifier. We applied Lossy Counting to the Tree Transition Graph to approximate the recurrent drifts. In the real word experiments, PEARL had a cumulative accuracy gain up to 13267% compared to the ARF baseline.

As future work, we will adapt the window size of Lossy Counting to increase the effectiveness of the Tree Transition Graph. Beyond that we can utilize a memory constrained budget to limit the number of concepts we store from the stream. In addition, we will explore the feasibility of using other ensemble-based methods within the PEARL framework.

Acknowledgment. We would like to thank Dr Hiekeun Ko, Science Director at Office of Naval Research Global, for his support of this project. This work was funded in part by the Office of Naval Research Global grant (N62909-19-1-2042).

References

1. Ahmadi, Z., Kramer, S.: Modeling recurring concepts in data streams: a graph-based framework. Knowl. Inf. Syst. **55**(1), 15–44 (2017). https://doi.org/10.1007/s10115-017-1070-0
2. Anderson, R., Koh, Y.S., Dobbie, G., Bifet, A.: Recurring concept meta-learning for evolving data streams. Expert Syst. Appl. **138**, 112832 (2019)
3. Ángel, A.M., Bartolo, G.J., Ernestina, M.: Predicting recurring concepts on data-streams by means of a meta-model and a fuzzy similarity function. Expert Syst. Appl. **46**, 87–105 (2016)
4. Chen, K., Koh, Y.S., Riddle, P.: Proactive drift detection: predicting concept drifts in data streams using probabilistic networks. In: IJCNN, pp. 780–787. IEEE (2016)
5. Chiu, C.W., Minku, L.L.: Diversity-based pool of models for dealing with recurring concepts. In: 2018 IJCNN, pp. 1–8. IEEE (2018)
6. Gomes, H.M., et al.: Adaptive random forests for evolving data stream classification. Mach. Learn. **106**(9-10), 1469–1495 (2017)
7. Gonçalves Jr., P.M., Barros, R.S.M.D.: RCD: a recurring concept drift framework. Pattern Recogn. Lett. **34**(9), 1018–1025 (2013)
8. Goyal, A., Daumé, H.: Lossy conservative update (LCU) sketch: succinct approximate count storage. In: 25th AAAI (2011)
9. Koh, Y.S., Huang, D.T.J., Pearce, C., Dobbie, G.: Volatility drift prediction for transactional data streams. In: 2018 IEEE ICDM, pp. 1091–1096. IEEE (2018)
10. Krawczyk, B., Minku, L.L., Gama, J., Stefanowski, J., Woźniak, M.: Ensemble learning for data stream analysis: a survey. Inf. Fusion **37**, 132–156 (2017)
11. Manku, G.S., Motwani, R.: Approximate frequency counts over data streams. In: Proceedings of the 28th VLDB, VLDB 2002, pp. 346–357. VLDB Endowment (2002)
12. Masud, M.M., et al.: Detecting recurring and novel classes in concept-drifting data streams. In: 2011 IEEE 11th ICDM, pp. 1176–1181. IEEE (2011)

13. Montiel, J., Read, J., Bifet, A., Abdessalem, T.: Scikit-multiflow: a multi-output streaming framework. J. Mach. Learn. Res. **19**(72), 1–5 (2018)
14. Yang, Y., Wu, X., Zhu, X.: Mining in anticipation for concept change: proactive-reactive prediction in data streams. Data Min. Knowl. Disc. **13**(3), 261–289 (2006)