Enabling Fast Lazy Learning for Data Streams

Peng Zhang ⋆†, Byron J. Gao †, Xingquan Zhu ‡, and Li Guo⋆
⋆Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100190, China
† Department of Computer Science, Texas State University, San Marcos, TX, 78666, USA
‡ QCIS, Faculty of Eng. & IT, University of Technology, Sydney, NSW 2007, Australia
zhangpeng@ict.ac.cn, bgao@txstate.edu, xqzhu@it.uts.edu.au, guoli@ict.ac.cn

Abstract—Lazy learning, such as k-nearest neighbor learning, has been widely applied to many applications. Known for well capturing data locality, lazy learning can be advantageous for highly dynamic and complex learning environments such as data streams. Yet its high memory consumption and low prediction efficiency have made it less favorable for stream oriented applications. Specifically, traditional lazy learning stores all the training data and the inductive process is deferred until a query appears, whereas in stream applications, data records flow continuously in large volumes and the prediction of class labels needs to be made in a timely manner. In this paper, we provide a systematic solution that overcomes the memory and efficiency limitations and enables fast lazy learning for concept drifting data streams. In particular, we propose a novel Lazy-tree (L-tree for short) indexing structure that dynamically maintains compact high-level summaries of historical stream records. L-trees are M-Tree [5] like, height-balanced, and can help achieve great memory consumption reduction and sub-linear time complexity for prediction. Moreover, L-trees continuously absorb new stream records and discard outdated ones, so they can naturally adapt to the dynamically changing concepts in data streams for accurate prediction. Extensive experiments on real-world and synthetic data streams demonstrate the performance of our approach.

Keywords—data stream mining, data stream classification, Spatial indexing, lazy learning, concept drifting.

I. INTRODUCTION

Data stream classification has drawn increasing attention from the data mining community in recent years with a vast amount of real-world applications. For example, in information security, data stream classification plays an important role in real-time intrusion detection, spam detection, and malicious Web page detection. In these applications, stream data flow continuously and rapidly, and the ultimate goal is to accurately predict the class label of each incoming stream record in a timely manner.

Existing data stream classification models, such as incremental learning [8] or ensemble learning models [12], [10], [19], [20], [21], all belong to the eager learning category [11]. Being eager, the training data are greedily compiled into a concise hypothesis (model) and then completely discarded. Examples of eager learning methods include decision trees, neural networks, and naive Bayes classifiers. Obviously, eager learning methods have low memory consumption and high predicting efficiency in answering queries, which are crucial for most data stream applications.

Lazy learning [3], such as k-Nearest Neighbor (kNN) classifiers, represents some instance-based and non-parametric learning methods, where the training data are simply stored in memory and the inductive process is deferred until a query is given. Compared to eager methods, lazy learning methods incur none or low computational costs during training but much higher costs in answering queries also with greater storage requirements, not scaling well to large datasets. In stream applications, data streams come continuously in large volumes, making it impractical to store all the training records. In addition, stream applications are time-critical where class prediction needs to be made in a timely manner. Lazy learning methods fall short in meeting these requirements and have not been considered for data stream classification.

In fact, lazy learning has many characteristics that are promising for data stream classification. While eager learning strives to learn a single global model that is good on average, lazy learning well captures locality and can achieve high accuracy when the learning environment is complex and dynamic. In data stream applications, stream records cannot be fully observed at a specific time stamp and it is often difficult to construct satisfactory global models based on partially observed training data. This difficulty is further aggravated by the inherent concept drifting problem in data streams, where the underlying patterns irregularly change over time resulting in much complicated decision boundary. A motivating example is shown in Example 1.

Example 1: Fig. 1 shows a typical scenario for classification in dynamic data streams. During the three continuous
time stamps, the classification boundary (concept) drifts from \( b_1 \) at time \( t_1 \) to \( b_2 \) at time \( t_2 \). We can observe that before and after the concept drifts \( i.e., \) at time \( t_1 \) and time \( t_3 \), both eager and lazy learning methods can perform equally well given the simplicity of the linear classification boundaries. However, when concept drifting occurs at time \( t_2 \), since only a small portion of examples in the red circle region are observed (the region represents emerging new concepts), the new boundary \( b_2 \) cannot be fully constructed yet. While eager learners tend to consider the examples in the red circle as noise and still return \( b_1 \) as the decision boundary, lazy learners can correctly capture partially formed new concepts in the red circle area.

Motivated by the above observations, the goal of this study is to overcome the high memory consumption and low predicting efficiency limitations and enable lazy learning on data streams. We expect that the study can unleash the potential of lazy learning in data stream classification and open up new possibilities for tackling the inherent problems and challenges in data stream applications. In particular, we propose a novel Lazy-tree (L-tree for short) indexing structure that dynamically maintains compact high-level summaries of historical stream records. L-tree is inspired by M-tree [5] that has been widely used as an indexing tool in metric spaces. When building an L-tree, historical stream records are condensed into compact high-level exemplars to reduce memory consumption. An exemplar is a sphere of certain size generalizing one or more stream examples. All the exemplars are organized into a height-balanced L-tree that can help achieve sub-linear predicting time. L-trees are associated with three key operations. The search operation traverses the L-tree to retrieve the \( k \)-nearest exemplars of an incoming stream record for classification purposes. The insertion operation adds new stream records into some exemplars in the L-tree. The deletion operation removes outdated exemplars from the L-tree. The L-tree approach achieves a logarithmic predicting time with bounded memory consumption. It also adapts quickly to new trends and patterns in stream data.

This paper makes the following contributions:

- We are the first to systematically investigate lazy learning on data streams. (Section II)
- We propose a compact high-level structure exemplar to summarize stream examples and reduce memory consumption for lazy learning. (Section III)
- We propose an L-tree structure to organize exemplars and achieve sub-linear time for prediction. L-trees continuously absorb new stream records and discard outdated ones, well adapting to drifting concepts and result in accurate prediction results. (Section IV)
- We conduct extensive experiments on real-world data streams and demonstrate the performance gain of our approach compared to benchmark methods. (Section V)

II. Problem Description

We study the problem of enabling lazy learning on data streams, for which we focus on significantly reducing memory consumption and predicting time so that the critical requirements of stream applications can be satisfied.

Consider a data stream \( S \) consisting of an infinite sequence of records \( \{s_1, \ldots, s_n, \ldots\} \), where \( s_i = (x_i, y_i) \) represents a record arriving at time stamp \( t_i \). For each record \( s_i \), \( x_i \in \mathbb{R}^\tau \) represents an \( \tau \)-dimensional attribute vector, and \( y_i \in \{c_1, \ldots, c_l\} \) represents the class label. Assume that the current time stamp is \( t_\tau \) and the incoming record is denoted by \( s_n = (x_n, y_n) \) with unknown \( y_n \). Our learning objective is to accurately predict \( y_n \) as fast as possible. This is equivalent to maximizing the posterior probability in Eq. (1) with maximum efficiency.

\[
y_n = \arg\max_{c \in \{c_1, \ldots, c_l\}} P(c|x_n, s_1, \ldots, s_{n-1})
\]

Without loss of generality, we consider \( k \)-NN as the underlying lazy learning algorithm. Then, instead of using all the \((n - 1)\) stream records to predict \( y_n \), we use only \( k \) nearest neighbors, denoted by \( \{s'_1, \ldots, s'_k\} \), from all the \((n - 1)\) records for prediction. Note that \( k \ll (n - 1) \), and \( \{s'_1, \ldots, s'_k\} \subset \{s_1, \ldots, s_{n-1}\} \). This way, the posterior probability in Eq. (1) can be revised to Eq. (2),

\[
y_n = \arg\max_{c \in \{c_1, \ldots, c_l\}} P(c, s'_1, \ldots, s'_k|x_n, s_1, \ldots, s_{n-1})
\]

Decomposing the objective function in Eq. (2) into two continuous probability estimations, we have Eq. (3) below,

\[
P(s'_1, \ldots, s'_k|x_n, s_1, \ldots, s_{n-1})P(c|x_n, s'_1, \ldots, s'_k, s_1, \ldots, s_{n-1})
\]

Since \( \{s'_1, \ldots, s'_k\} \cap \{s_1, \ldots, s_{n-1}\} = \{s'_1, \ldots, s'_k\} \), Eq. (3) can be simplified as in Eq. (4),

\[
P(s'_1, \ldots, s'_k|x_n, s_1, \ldots, s_{n-1})P(c|x_n, s'_1, \ldots, s'_k)
\]

From Eq. (4), it is clear that estimating Eq. (1) using \( k \)-NN takes two steps. First, estimating \( x_n \)'s \( k \) neighbors \( \{s'_1, \ldots, s'_k\} \) from all the historical stream records \( \{s_1, \ldots, s_{n-1}\} \). Second, estimating \( x_n \)'s class label using all the \( k \) estimated neighbors.

In data stream environment, estimating these two probabilities is very challenging because of the memory consumption and predicting time constraints. Specifically, in order to estimate Eq. (4), two concerns need to be addressed:

1. How to estimate \( P(s'_1, \ldots, s'_k|x_n, s_1, \ldots, s_{n-1}) \) in a bounded memory space. In data streams, it is impractical to maintain all the \((n - 1)\) records \( \{s_1, \ldots, s_{n-1}\} \) for estimation. Thus, a memory-efficient algorithm needs to be designed for this purpose.
2. How to estimate the probability \( P(y_n|x_n, s'_1, \ldots, s'_k) \) as fast as possible. Given \( x_n \), finding its \( k \) nearest neighbors \( s'_1, \ldots, s'_k \) typically requires a linear scan of all the \((n - 1)\) records, corresponding to an \( O(n) \) time complexity, which
is unacceptable for stream applications. Thus, an efficient search algorithm is needed to reduce the predicting time to a sub-linear complexity of $O(\log(n))$.

In the following, we use Example 2 to illustrate the streaming lazy learning problem.

![Figure 2. An illustration of Example 2.](image)

**Example 2:** Consider a data stream $S$ having three classes ($c_1$, $c_2$, $c_3$), and each stream record has two dimensions $(\gamma_1, \gamma_2)$, where $\gamma_i \in \mathbb{R}$. Suppose that at time stamp $t_{500}$, totally 500 stream records are observed as shown in Fig. 2. For simplicity, we assume that these 500 records are distributed uniformly in five clusters $D_1, \cdots, D_5$, and all records in a cluster $D_i$ $(1 \leq i \leq 5)$ share the same class label, where class $c_1$ is denoted by the symbol “*”, class $c_2$ is denoted by “o”, and class $c_3$ is denoted by “x”. The classification objective is to predict the class label of the next incoming record (e.g., the small red circle $x = (2.2,3.5)$ in Fig.2) as fast as possible. If the original $k$-NN method is chosen as the solution, we have to maintain all the 500 stream records for prediction, which is very demanding for memory consumption. Moreover, even if memory consumption is not an issue, a straightforward approach for comparing consumption. Moreover, even if memory consumption is not an issue, a straightforward approach for comparing $\gamma_1$ and each incoming record can be efficiently estimated in a sub-linear time complexity by traversing the L-tree.

To reduce the memory consumption and improve the prediction efficiency for lazy learning on data streams, we first summarize all the historical training examples into compact exemplars, and then organize these exemplars as leaf nodes in a height-balanced L-Tree structure. In doing so, all the historical stream records can be condensed into a bounded memory space without losing much information, and each incoming record can be efficiently estimated in a sub-linear time complexity by traversing the L-tree.

### III. The Exemplar Structure

Instead of maintaining raw stream records, we cluster them into exemplars to reduce memory consumption. An **exemplar** is a sphere generalizing one or multiple records. Though inspired by micro-clusters [2], exemplars are different and more complex in that they summarize labeled data. Formally, exemplars are defined as follows.

**Definition 1:** An exemplar $M$ for a set of stream records $\{s_1, \cdots, s_n\}$ arriving at time stamps $\{t_1, \cdots, t_n\}$ is a $(d + l + 3)$-dimensional vector as in Eq. (5),

$$M = (\bar{X}, \bar{R}, \bar{C}, N, T)$$

where $\bar{X}$ is a $d$-dimensional vector that representing the center, $\bar{R}$ is the sample variance of all records in $M$, which is also the covering radius of $M$, $\bar{C} = [P(c_1|M), \cdots, P(c_i|M)]$ is an $l$-dimensional vector corresponding to the probability of $M$ having class label $c_i$ $(1 \leq i \leq l)$, $N$ is the total number of records in $M$, and $T$ represents the time stamp when $M$ was last updated.

Exemplars have several intrinsic merits for lazy learning on data streams.

- **Exemplars help summarize huge volumes of stream data into compact structures which well fit into memory.**
- **Exemplars well represent the historical data because nearby examples tend to share the same class label and can be grouped together as a prediction unit.**
- **Exemplars can be easily updated.** If a new example $x$ is absorbed into $M$, the exemplar center and radius can be conveniently updated using Eqs.(6) and (7) respectively, the class label $\bar{c}$ can be updated using Eq. (8), and the time stamp $\bar{T}$ can be updated to the current time stamp.

**Update center $\bar{X}$ and radius $\bar{R}$.** Assume $n$ records $\{(x_1, \gamma_1), \cdots, (x_n, \gamma_n)\}$ have been absorbed into an exemplar $M$. When a new stream record $x$ arrives, the center $\bar{X}$ of $M$ can be updated using Eq. (6),

$$\bar{X} \leftarrow \frac{1}{n+1} \left( \sum_{i=1}^{n} x_i + x \right) = \frac{n}{n+1} \bar{X} + \frac{1}{n+1} x,$$  \hspace{1cm} (6)$$

and the radius $\bar{R}$ of $M$ can be updated using Eq. (7),

$$\bar{R} \leftarrow \frac{1}{n} \left( \sum_{i=1}^{n} (x_i - \bar{X})^2 + (x - \bar{X})^2 \right) = \frac{n-1}{n} \bar{R} + \frac{1}{n} (x - \bar{X})^2.$$  \hspace{1cm} (7)$$

**Update class label $\bar{C}$.** Assume a new record arrives with a class label $c_p$ $(1 \leq p \leq l)$, then the class label vector $\bar{C}$ can be updated using Eq. (8),

$$\bar{C} \leftarrow [\frac{nP(c_1|M)}{n+1}, \cdots, \frac{nP(c_p|M) + 1}{n+1}, \cdots, \frac{nP(c_l|M)}{n+1}]$$ \hspace{1cm} (8)$$

Note that for each class $c_i$ $(1 \leq i \leq l)$, $P(c_i|M) = \frac{1}{n} \sum_{i=1}^{n} P(c_i|x_j)$. Adding a new $x$ with label $c_p$, we can have the following:

(i) for each $j \neq p$, $P(c_j|x) = 0$, and $P(c_j|M \cup x) = \frac{1}{n+1} \left( \sum_{i=1}^{n} P(c_i|x_i) + P(c_j|x) \right) = \frac{n}{n+1} P(c_j|M)$,

(ii) for the $j = p$, $P(c_j|x) = 1$, $P(c_j|M \cup x) = \frac{1}{n+1} \left( \sum_{i=1}^{n} P(c_i|x_i) + P(c_j|x) \right) = \frac{n}{n+1} P(c_j|M) + \frac{1}{n+1}$.

Algorithm 1 shows the procedure of constructing and updating a set $E$ of exemplars on data stream $S$. Initially, a small portion of records $S_0$ are read from stream $S$, ...
and clustered into $u$ clusters using K-Means, forming the initial exemplar set $E$. For each incoming stream record $x$, its nearest exemplar $e$ is retrieved from $E$. If the distance between $e$ and $x$ is larger than the given threshold $\gamma$, a new exemplar $e_{new}$ will be created and inserted into $E$. Otherwise, $x$ will be absorbed into $e$ using the updating rules.

For better understanding, we use Example 3 to illustrate the procedure of summarizing the 500 stream records given in Example 2 into exemplars.

Example 3: The 500 stream records can be summarized into five exemplars $\{M_1, \cdots, M_5\}$ as shown in Fig. 3. The detailed information of the five exemplars is listed in Table I. Compared to preserving all raw stream records, the exemplars consume only 1% of the memory space.

![Algorithm 1: Create and maintain exemplars.](image)

<table>
<thead>
<tr>
<th>ID</th>
<th>$X$</th>
<th>$R$</th>
<th>$C$</th>
<th>$N$</th>
<th>$T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1$</td>
<td>(1,1,1)</td>
<td>0.5</td>
<td>(1,0,0)</td>
<td>100</td>
<td>t100</td>
</tr>
<tr>
<td>$M_2$</td>
<td>(1,2,5)</td>
<td>0.5</td>
<td>(0,1,0)</td>
<td>100</td>
<td>t200</td>
</tr>
<tr>
<td>$M_3$</td>
<td>(2,5,2)</td>
<td>0.5</td>
<td>(0,0,1)</td>
<td>100</td>
<td>t300</td>
</tr>
<tr>
<td>$M_4$</td>
<td>(4,5,3)</td>
<td>0.5</td>
<td>(0,1,1)</td>
<td>100</td>
<td>t400</td>
</tr>
<tr>
<td>$M_5$</td>
<td>(3,4)</td>
<td>0.5</td>
<td>(0,1,0)</td>
<td>100</td>
<td>t500</td>
</tr>
</tbody>
</table>

From Example 3, we can also observe that the number of comparisons for predicting a testing record is reduced from 500 to only 5. Formally, by using exemplars, the estimate function in Eq. (4) can be converted to Eq. (9).

$$P(e_1|\mathcal{M}_1, \cdots, \mathcal{M}_n, M_1, \cdots, M_{n-1})P(c|x_n, M_1, \cdots, M_k)$$

where $\{M'_1, \cdots, M'_{n-1}\}$ are exemplars, and $M'_{1}, \cdots, M'_{k}$ are $x_n$'s $k$ nearest exemplars.

Figure 3. An illustration of Example 3.

A possible limitation of estimating Eq. (9) is that the number of exemplars continuously increases with time, a linear scan of all exemplars for prediction is still inefficient for time-critical stream applications. This motivates our height-balanced L-tree structure for further improvement of predicting efficiency.

IV. L-Tree Indexing

In this section, we introduce the L-tree structure and its three key operations: Search, Insertion, and Deletion.

A. The L-Tree Structure

L-Trees extend M-Trees [5]. While M-trees index objects in metric spaces such as voice, video, image, text, and numerical data, L-trees are extended to labeled data on data streams, for which additional information needs to be stored such as class labels and time stamps. L-trees index exemplars that are spherical spatial objects. This is different from other spatial indexing structures such as R-Trees and R*-Trees that index rectangular spatial objects.

![Figure 4. An illustration of the L-tree structure.](image)

An L-Tree mainly consists of two components as shown in Fig. 4: (1) an M-tree like structure on the right-hand side storing all exemplars, and (2) a table structure on the left-hand side storing time stamps of all the exemplars. The two structures are connected by linking each time stamp in the table to its corresponding exemplar in the tree.

The tree structure consists of two different types of nodes: leaf nodes and routing nodes. The root node can be considered as a special routing node that has no parent. A
leaf node contains a batch of exemplars represented in the form of,
\[(pointer, distance),\]
where `pointer` references the memory location of an exemplar, `distance` indicates the distance between the exemplar and its parent node. On the other hand, a routing node in the tree structure contains entries in the form of,
\[(\mu, r, child, distance),\]
where \(\mu\) represents the center of the covering space, \(r\) represents the covering radius, `child` is a pointer that references its child node, and `distance` denotes the distance of the entry to its parent node. Two important parameters of L-Trees are \(M\) and \(m\), which denote the maximum (\(M\)) and the minimum (\(m\)) number of entries in a node.

Similar to M-Trees, L-Trees have the following properties:
- Routing node. A routing node has between \(m\) and \(M\) number of entries unless it is the root. Each entry in a routing node covers the tightest spatial area of its child nodes.
- Leaf node. A Leaf node contains between \(m\) and \(M\) number of exemplars unless it is the root node, and all leaf nodes are at the same level.
- Root node. The root node is a special routing node, which has at least two entries unless it is a leaf.

**Example 4:** Fig. 5 illustrates the L-tree structure for the exemplars in Example 3. For simplicity, the time stamp table is omitted.

![L-tree for the exemplars](image)

**B. Search**

Each time a new record \(x\) arrives, a search operation is invoked to calculate the class label for \(x\). The search algorithm first traverses the L-tree to find its \(k\) nearest exemplars in leaf nodes. Then it calculates the class label for \(x\) by combining label information from all retrieved \(k\) exemplars using a majority voting scheme.

Compared to a linear scan of all exemplars as shown in Algorithm 1, organizing the exemplars in a height-balanced tree can significantly reduce the search time cost from \(O(N)\) to \(O(\log(N))\), where \(N\) is the total number of exemplars in the L-tree. Such a search method can be further improved by using a branch-and-bound technique.

The bound \(b\) is defined as follows. Assume that the search algorithm has traversed \(u\) routing entries \(\{O_1, \cdots, O_u\}\) (\(u \ge k\)), with the distance between each \(O_i\) and \(x\) represented as \(d = [d_1, \cdots, d_u]\). The bound \(b\) is defined as the maximal distance of the \(k\) smallest distances in \(d = [d_1, \cdots, d_u]\) as in Eq. (12).

\[b = \max \{\min_i[d_1, \cdots, d_u]\}\]  

![An illustration of the bound in Eq.(13)](image)
where \( d(e_p, x) \) denotes the distance between \( e_p \) and \( x \), \( e_c, d \) is the distance between \( e_c \) and \( e_p \), \( e_c, r \) is \( e_c \)’s covering radius, and \( b \) is the bound. Obviously, all the above distances are pre-computed, and Eq. \((13)\) can be easily estimated.

Algorithm 2 contains detailed procedures of the branch-
and-bound search. A priority queue \( Q \) is used to perform
breath-first search. In addition, an array \( U \) is used to preserve
all the \( k \) results. The main purpose of the algorithm is to re-
trieve the \( k \) results using minimized number of comparisons
by making full use of the bound \( b \). For each routing entry \( O_i \),
if and only if the pruning condition in Eq. \((13)\) is satisfied,
the node will be traversed. The function \( UpdateArray(U, e) \)
guarantees that the array \( U \) always contains the \( k \) results by
continuously removing the unsatisfied ones.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure7.png}
\caption{An illustration of search in Example 4.}
\end{figure}

**Example 5:** Consider an incoming testing record \( x \) = 
\((2.2, 3.5)\), we need to traverse the L-tree in Fig. 5 to predict
its class label. Assume the parameter \( k \) is set to 1. The
search algorithm initially pushes entries \( e_1 \) and \( e_2 \) in query
\( Q \). Then, it calculates the distance \( d(x, e_1) = 1.75 \)
and \( d(x, e_2) = 1.55 \), and updates the bound \( b \) to the smaller one
of 1.55 and traverses along \( e_2 \). Next, it sequentially compares
\( x \) with entries \( e_6 \) and \( e_7 \) in the leaf node \( O_3 \) and obtains
\( d(x, e_6) = 2.55 \) and \( d(x, e_7) = 0.94 \). Thus, \( e_7 \) is taken as the
1-nearest neighbor, and the class label \( y_x \) is set to \( c_2 \). The
comparison path is shown in Fig. 7. In this example, the
search would involve four comparisons in the worst case.
Compared to the linear scan that requires five comparisons,
L-tree achieves 20\% improvement in the worst case.

\section{C. Insertion}

Insertion operations are used to absorb new stream records
into L-trees, so that they can quickly adapt to new trends and
patterns in data streams.

Algorithm 3 lists detailed procedures of the insertion
operation. For each incoming record \( x \), a search algorithm is
invoked to find its nearest leaf node \( O \). When inserting \( x \)
in the retrieved leaf node \( O \), three different situations need
to be considered:

\begin{itemize}
\item \( x \) can be absorbed in one of the entries \( e \in O \). This is
the ideal situation, and the algorithm updates the leaf
node defined in Eq. \((1)\) according to updating rules.
\item \( x \) cannot be absorbed in any entry, and the leaf node \( O \)
is not full. In this case, a new entry \( e_{\text{new}} \) is generated
and inserted into the leaf node \( O \).
\item \( x \) cannot be absorbed in any entry, and the leaf node \( O \)
is full. In this case, a new entry \( e_{\text{new}} \) is generated,
and then a node splitting operation is invoked to obtain
spare room for insertion.
\end{itemize}

Node splitting is the most critical step in the insertion
operation. Similar to M-trees, a basic principle in splitting
is that the split leaf nodes should have the minimized spatial
expansion. This is equivalent to minimizing Eq. \((14)\),
\begin{equation}
< O_1, O_2 > = \arg \min_{<X,Y>|X,Y \subset O \cup e_{\text{new}}} (X. r + Y. r), \quad (14)
\end{equation}
where \( X \) and \( Y \) are variables, and \( O_1 \) and \( O_2 \) are the new
leaf nodes containing \( e_{\text{new}} \) and all entries in \( O \). Obviously,
solving Eq. \((14)\) requires examining all possible combinations
of all entries in \( O \cup e_{\text{new}} \), which is very difficult especially
when \( M \) is large. Alternatively, a greedy heuristic would first
randomly select two entries in \( O \cup e_{\text{new}} \) that has the largest
distance, and then cluster all the remaining \( M - 1 \) entries in
\( O \cup e_{\text{new}} \) into the given two groups.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure8.png}
\caption{An illustration of L-tree in Example 3 after inserting a new
stream record \( x \).}
\end{figure}
Example 6: Consider inserting $x = (2.2, 3.5)$ into the L-tree in Example 3. First of all, the search algorithm locates entry $e_7$ in the leaf node $O_1$ as the target node. Then, it examines the distance between $x$ and $e_7$, which equals to 0.94 and is larger than the covering radius of $e_7$. Thus, a new entry $e_8$ is generated containing a new exemplar $M_6$ as follows: $M_6, X = (2.2, 3.5), M_6, R = 0.1, M_6, T = (0, 1, 0), M_6, N = 1, M_6, T = t_{001}$. Since the leaf node $O$ has only two entries, which is less than its capacity $M = 3$, then $e_8$ is inserted into $O$ directly. In addition, the covering space of entry $e_2$ in the parent node is enlarged to $e_2, r = 1.66$. The updated L-tree structure is shown in Fig. 8 and Fig. 9.

D. Deletion

The deletion operation discards outdated exemplars when the L-tree reaches its capacity. For example, if the largest tree size is set to four in Example 4, $e_3$ will be discarded from the L-tree when $e_8$ is generated, this is because $e_3$ has the earliest time stamp $t_{000}$, which means it has not been updated for a long time, and is possibly outdated.

The detailed procedures of the deletion operation are shown in Algorithm 4. First of all, the outdated entry in a leaf node is discovered by scanning the time table, and deleted from the L-tree. After the deletion, there are two different situations.

- The number of entries in the leaf node is larger than $m$. In this case, the algorithm iteratively adjusts the covering radius $r$ of its parent entries, making these nodes to be more compact.
- The number of entries in the leaf node is smaller than $m$. In this case, a delete-then-insert method will be used. Similar methods are commonly used in spatial indexing structures. It first iteratively deletes node(s) having entries less than $m$, and re-inserts their entries into the tree using the insertion operation in Algorithm 3. This method is advantageous in that: (1) It is easy to implement. (2) Re-insertion will incrementally refine the spatial structure of the tree.

Algorithm 4: Deletion

<table>
<thead>
<tr>
<th>Input</th>
<th>L-tree $T$, parameters $m$, $M$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Updated L-Tree $T'$</td>
</tr>
</tbody>
</table>

// locate a leaf node from the time table ;
pointer <- Locate(Time_table) ;
$O$ <- delete($T$, pointer) ;

// iterative deletions ;
if $O$ is root then
  if $O$.entries() == 1 then // the root node
    $T'$ <- $O$.child ;
    delete($O$) ;
  else // non-root nodes
    if $O$.numberOfEntries() < $m$ then
      foreach entry $e$ $\in$ $O$
        delete($e$, $O$) ;
      $T'$ <- insert($e$, $O$) ;

Output $T'$ ;

V. Experiments

In this section, we present extensive experiments on benchmark datasets to validate the performance of L-trees with respect to time efficiency for prediction, memory consumption, and predicting accuracy. All experiments are implemented in Java on a Microsoft XP machine with 3GHz CPU and 2GB memory.

A. Experimental Settings

Benchmark data sets. Twelve data sets from the UCI data repository [4] and stream data mining repository [22] are used in our experiments. Table II lists the basic information of the data sets. Due to the page limit, please refer to [4, 22] for detailed descriptions. The synthetic data stream contains a gradually changing concept (decision boundary) defined by Eq. (15):

$$ g(x) = \sum_{i=1}^{r-1} a_i \cdot \frac{(x_i + x_{i+1})}{x_i} \quad (15) $$

where $a_i$ controls the shape of the decision surface and $g(x)$ determines the class label of each record $x$. Concept drifting can be controlled by adjusting $a_1$ to $a_1 + ah$ after generating $D$ records, where $\alpha$ defines the direction of change and $h \in [0, 1]$ defines the magnitude of change. $\alpha = -1$ with probability of $\rho$. In our experiments, we set $\rho = 0.2$, $h = 0.1$, $D = 2000$, $r = 3$, and generate $10^5$ stream records from five classes. Parameter $m$, if not specially mentioned, is set to 1.

Benchmark methods. For comparison purposes, we implemented two lazy learning and two eager learning models.

1. Global k-NN. This is the traditional k-NN method that maintains all historical stream records for prediction.
2. Local k-NN. In this method, only a small portion of the most recent stream records are preserved for prediction. Compared to our method, this method simply uses a linear scan
to retrieve the $k$ nearest neighbors. (3) Incremental Decision Tree. This method belongs to the eager learning category. It is based on the method proposed in [8]. The source code can be downloaded from www.cs.washington.edu/dm/vml.

(4) Incremental Naive Bayes. This is another standard eager learning model.

**Measures.** We use three measures in our experiments. (1) Predicting time. By using a height-balanced tree to index all exemplars, L-trees are expected to achieve lower computational costs than other two lazy learning models. (2) Memory consumption. L-trees are expected to consume much less memory space than Global $k$-NN. (3) Predicting accuracy. L-trees are expected to achieve similar predicting accuracy to Global $k$-NN, and better accuracy than Local $k$-NN.

### B. Parameter Study on Synthetic Data Streams

**Parameter $\gamma$.** This parameter denotes the maximum radius threshold of exemplars in an L-tree. Fig. 10 shows the predicting time and predicting accuracy w.r.t. different $\gamma$ values. From the results we can observe that both the predicting time and predicting accuracy decreases with increasing $\gamma$. This is because the larger $\gamma$, the fewer exemplars that are generated. As a result, both the predicting time and memory consumption are reduced. On the other hand, generalizing stream records into fewer exemplars leads to more information loss and reduced predicting accuracy.

<table>
<thead>
<tr>
<th>Name</th>
<th>Records</th>
<th>Attr. clas.</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensor</td>
<td>$2.0 \times 10^5$</td>
<td>5</td>
<td>54</td>
</tr>
<tr>
<td>KDDCUP99</td>
<td>$4.9 \times 10^6$</td>
<td>41</td>
<td>23</td>
</tr>
<tr>
<td>Powersupply</td>
<td>$2.9 \times 10^5$</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>Waveform</td>
<td>$5.0 \times 10^5$</td>
<td>40</td>
<td>3</td>
</tr>
<tr>
<td>hallofame</td>
<td>$1.3 \times 10^5$</td>
<td>17</td>
<td>3</td>
</tr>
<tr>
<td>kr-vs-kp</td>
<td>$3.1 \times 10^5$</td>
<td>37</td>
<td>2</td>
</tr>
<tr>
<td>sick</td>
<td>$3.7 \times 10^5$</td>
<td>30</td>
<td>2</td>
</tr>
<tr>
<td>hypothyroid</td>
<td>$3.7 \times 10^5$</td>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>mushroom</td>
<td>$8.1 \times 10^5$</td>
<td>23</td>
<td>2</td>
</tr>
<tr>
<td>spice</td>
<td>$3.1 \times 10^5$</td>
<td>61</td>
<td>3</td>
</tr>
<tr>
<td>nursery</td>
<td>$1.2 \times 10^6$</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>musk</td>
<td>$6.5 \times 10^5$</td>
<td>167</td>
<td>2</td>
</tr>
</tbody>
</table>

![Figure 10. Comparisons with respect to different $\gamma$ values. $M=10$.](image-url)

![Figure 12. Comparisons between eager and lazy learning models on 40 continuous data chunks.](image-url)

**Parameter $M$.** This parameter denotes the maximum number of entries in each node of an L-tree. Fig. 11 shows the predicting time and memory consumption with respect to different $M$ values. From the results we have the following observations. When $M$ increases at the very early stage, both the predicting time and memory consumption decrease significantly. After that, the benefit becomes marginal and then turns negative with increasing $M$. This is because increasing $M$ at an early stage reduces the number of routing nodes and leaf nodes. As a result, the $k$-nearest neighbors of an incoming query are likely to be stored in the same node, leading to reduced search and storage costs. However, when $M$ continues to increase, exemplars that slightly overlap with each other will be mistakenly stored in the same node, leading to extra comparisons on each node and increased search time. Therefore, a desirable $M$ value should neither be too large nor too small.

**C. Performance Study on Real-world and Synthetic Data Streams**

Table III presents the comparison results between three lazy learning models. The parameter $k$ is set to 3, other parameters are listed in Table II. From the results we can conclude that: (1) Compared to Global $k$-NN, L-tree is more efficient with respect to predicting time and memory consumption, and can achieve similar predicting accuracy. This is because L-trees condense historical stream records into compact exemplars and indexes them into a high-
balanced tree structure. In doing so, the memory cost and predicting time can be significantly reduced without losing much useful information for prediction. (2) Compared to Local $k$-NN, L-tree can achieve better predicting accuracy and efficiency given the same memory consumption. This is because L-tree maintains more information for prediction than local $k$-NN. In addition, by using a high-balanced tree structure, L-tree can achieve better predicting efficiency.

Table IV and Fig. 12 show the comparisons between L-tree, incremental Decision Tree, and incremental Naive Bayes learning models. From the results we can observe that: (1) In terms of time cost, L-tree incurs no training time but more predicting time. By combining training and predicting time together, we can see that L-tree is even more efficient than the two eager learning models. (2) In terms of predicting accuracy, L-tree performed similarly to the eager learning models on the static data sets. L-tree did not show advantage over eager models in these experiments because the datasets did not exhibit significant concept drifting. As shown in Fig. 12, on the synthetic data streams that exhibit significant concept drifting and complex decision boundaries, L-tree outperformed the eager learning models because it adapts faster to the drifting concepts in data streams.

In summary, the proposed L-tree approach enables fast lazy learning on data streams by significantly reducing predicting time and memory consumption. Moreover, compared to eager learners, lazy learning can achieve better predicting accuracy for concept drifting data streams.

VI. Related work

Eager learning and lazy learning. Decision trees [13] are typical eager learners and $k$-nearest neighbor (knn) classifiers [7] exemplify the simplest form of lazy learners. The distinguishing characteristics of eager and lazy learners were identified in [3]. Eager learners are model-based and parametric, where the training data are greedily compiled into a concise hypothesis (model) and then completely discarded. Lazy learners are instance-based and non-parametric, where the training data are simply stored in memory and the inductive process is deferred until a query is given. Obviously, lazy learners incur lower computational costs during training but much higher costs in answering queries also with greater storage requirements, not scaling well to large datasets. However, by retraining all the training data, lazy learners do not lose information in the training data, which make them capable of quickly adapting to the changing data distributions in dynamic learning environments, such as data streams. On the other hand, for lazy learning, a prediction model is built for each individual test record. As a result, the decision is customized according to the data characteristics of each test record. In concept drifting environments, if data distributions or concepts drift rapidly, lazy learning has the advantage of relying on the loci of each test example to derive decision models and outperform global models.

Data stream classification. Data stream classification [1] has vast real-world applications, which are usually time-critical and require fast predictions. Many sophisticated learning methods have been proposed, such as incremental learning [8] and ensemble learning [12]. These methods belong to the eager learning category, which aims at building a global model from historical stream data for prediction. Our work differs from existing approaches in that we propose and study lazy learning for data stream classification.

K-NN query on data streams. A number of $k$-NN query methods on data streams have been proposed with focus on various types of data, such as relational data, uncertain data, semi-structured data, spatial data, fuzzy data, and time series data [18], [16]. Our work differs from theirs in two ways. First, we query data streams for classification purposes while they do not. Second, existing $k$-NN query methods on data streams use a sliding window to reduce the query space for efficiency, which can be considered as local $k$-NN. In contrast, we maintain high-level compact summaries of stream records, which is an approximation of global $k$-NN on data streams.

Indexing techniques on databases. $k$-NN query on
datasets has been extensively studied in the databases community. Many efficient indexing and hashing techniques have been proposed to partition the query space and achieve \( O(\log(N)) \) query time. Examples of such techniques include k-d tree [14], vp-tree [17], to name a few. Our work differs from theirs in that we index dynamically changing data streams while they index static data.

**Data stream summarization.** In order to query or mine stream data, many synopsis structures [6] exist to transform large volume stream data into memory-economic compact summaries that can be rapidly updated as the stream records arrive. Typical synopses include sampling techniques and sketching methods. Our work differs from theirs in that we summarize labeled data for classification purposes.

**VII. Conclusions**

Lazy learning can be advantageous in dynamic and complex learning environments such as data streams. However, due to its high memory consumption and low efficiency for prediction, lazy learning is not favorable for stream applications where rapid predictions are essential. To overcome these limitations and enable fast lazy learning for data streams, we proposed a novel Lazy-tree indexing structure that dynamically maintains compact summaries of historical stream records to facilitate accurate and fast predictions. Experiments and comparisons demonstrated the performance gain (in terms of memory consumption, time efficiency for prediction, and classification accuracy) on both synthetic and real-world data streams.

There are many interesting topics for future work. For example, sophisticated class-aware summarization techniques [9] can be used to generate exemplars. As another example, the temporal dimension can be included in the distance computation for nearest neighbors, and various weighting schemes can also be investigated beyond the basic majority voting. Last but not least, we hope our study can help unleash the potential of lazy learning in data stream classification and open up new possibilities for tackling the inherent problems and challenges in data stream applications.

**VIII. Acknowledgements**

This research was supported by the National Science Foundation of China (NSFC) Grant (61003167), Basic Research Program of China 973 Grant (2007CB311100), National High Technology Research and Development Program of China 863 Grant (2011AA010705), Texas Normal Hackerman Advanced Research Program Grant (003656-0035-2009), and Australian Research Council (ARC) Future Fellowship Grant (FT100100971).

**References**