Hierarchical Clustering for Real-Time Stream Data with Noise

Philipp Kranen, Felix Reidl, Fernando Sanchez Villaamil, and Thomas Seidl

Data Management and Data Exploration Group, RWTH Aachen University, Germany
{kranen,reidl,sanchez,seidl}@cs.rwth-aachen.de

Abstract. In stream data mining, stream clustering algorithms provide summaries of the relevant data objects that arrived in the stream. The model size of the clustering, i.e. the granularity, is usually determined by the speed (data per time) of the data stream. For varying streams, e.g. daytime or seasonal changes in the amount of data, most algorithms have to heavily restrict their model size such that they can handle the minimal time allowance. Recently the first anytime stream clustering algorithm has been proposed that flexibly uses all available time and dynamically adapts its model size. However, the method exhibits several drawbacks, as no noise detection is performed, since every point is treated equally, and new concepts can only emerge within existing ones. In this paper we propose the LiarTree algorithm, which is capable of anytime clustering and at the same time robust against noise and novelty to deal with arbitrary data streams.

1 Introduction

There has been a significant amount of research on data stream mining in the past decade and the clustering problem on data streams has been frequently motivated and addressed in the literature. Recently the ClusTree algorithm has been proposed in [3] as the first anytime algorithm for stream clustering. It automatically self-adapts its model size to the speed of the data stream. Anytime in this context means that the algorithm can process an incoming stream data item at any speed, i.e. at any time allowance, without any parameterization by the user. However, the algorithm does not perform any noise detection, but treats each point equally. Moreover, it has limited capabilities to detect novel concepts, since new clusters can only be created within existing ones. In this paper we build upon the work in [3] and maintain its advantages of logarithmic time complexity and self-adaptive model size. We extend it to explicitly handle noise and improve its capabilities to detect novel concepts. While we improve the approach and add new functionality, it stays an anytime algorithm that is interruptible at any time to react to varying stream rates.

Due to a lack of space we will not repeat the motivation for stream clustering and anytime algorithms here. Neither can we recapitulate related work, especially the ClusTree presented in [3]. However, we stress that a good understanding of the work in [3] is indispensable for understanding the remainder of this paper.

J.B. Cushing, J. French, and S. Bowers (Eds.): SSDBM 2011, LNCS 6809, pp. 405–413, 2011.

We refer to [3] for motivation, related work and, most importantly, the ClusTree algorithm as a prerequisite for the following.

2 The LiarTree

In this section we describe the structure and working of our novel LiarTree. In the previously presented ClusTree algorithm [3] the following important issues are not addressed:

- **Overlapping:** the insertion of new objects followed a straight forward depth first descent to the leaf level. No optimization was incorporated regarding possible overlapping of inner entries (clusters).
- **Noise:** no noise detection was employed, since every point was treated equal and eventually inserted at leaf level. As a consequence, no distinction between noise and newly emerging clusters was performed.

We describe in the following how we tackle these issues and remove the drawbacks of the ClusTree. Section 2.6 briefly summarizes the LiarTree algorithm and inspects its time complexity.

2.1 Structure and Overview

The LiarTree summarizes the clusters on lower levels in the inner entries of the hierarchy to guide the insertion of newly arriving objects. As a structural difference to the ClusTree, every inner node of the LiarTree contains one additional entry which is called the noise buffer.

Definition 1. ***LiarTree.*** *For $m \leq k \leq M$ a LiarTree node has the structure $node = \{e_1, \ldots, e_k, CF^{(t)}_{nb}\}$, where $e_i = \{CF^{(t)}, CF^{(t)}_b\}$, $i = 1 \ldots k$ are entries as in the ClusTree and $CF^{(t)}_{nb}$ is a time weighted cluster feature that buffers noise points. The amount of available memory yields a maximal height (size) of the LiarTree.*

The noise buffer consists of a single CF which does not have a subtree underneath itself. We describe the usage of the noise buffer in Section 2.3.

Algorithm 1 illustrates the flow of the LiarTree algorithm for an object x that arrives on the stream. The variables store the current node, the hitchhiker (h) and a boolean flag indicating whether we encourage a split in the current subtree (details below). After the initialization (lines 1 to 1) the procedure enters a loop that determines the insertion of x as follows: first the exponential decay is applied to the current node in line 1. If nothing special happens, i.e. if none of the if-statements is true, the closest entry for x is determined (line 1) and the object descends into the corresponding subtree (line 1). As in the ClusTree, the buffer of the current entry is taken along as a hitchhiker (line 1) and a hitchhiker is buffered if it has a different closest entry (lines 1 to 1). Being an anytime algorithm the insertion stops if no more time is available, buffering

Algorithm 1. Process object (x)

```
1  currentNode = root; encSplit = false;
2  h = empty; // h is the hitchhiker
3  while (true) do                          /* terminates at leaf level latest */
4  |  update time stamp for currentNode;
5  |  if (currentNode is a liar) then
6  |  |  liarProc(currentNode, x); break;
7  |  end if
8  |  e_x = calcClosestEntry(currentNode, x, encSplit);
9  |  e_h = calcClosestEntry(currentNode, h, encSplit);
10 |  if (e_x ≠ e_h) then
11 |  |  put hitchhiker into corresponding buffer;
12 |  end if
13 |  if (x is marked as noise) then
14 |  |  noiseProc(currentNode, x, encSplit); break;
15 |  end if
16 |  if (currentNode is a leaf node) then
17 |  |  leafProc(currentNode, x, h, encSplit); break;
18 |  end if
19 |  add object and hitchhiker to e_x;
20 |  if (time is up) then
21 |  |  put x and h into e_x's buffer; break;
22 |  end if
23 |  add e_x's buffer to h;
24 |  currentNode = e_x.child;
25 end while
```

x and h in the current entry's buffer (line 11). The issues listed in Section 2 are solved in the procedures *calcClosestEntry* (line 11), *liarProc* (line 11) and *noiseProc* (line 11). We detail these methods in the following subsection and start by describing how we descend and reduce overlapping of clusters using the procedure *calcClosestEntry*.

2.2 Descent and Overlap Reduction

The main task in inserting an object is to determine the next subtree to descend into, i.e. finding the closest entry. Besides determining the closest entry, the algorithm checks whether the object is classified as noise w.r.t. the current node and sets an *encSplit* flag, if a split is encouraged in the corresponding subtree.

First we check whether the current node contains an irrelevant entry. This is done as in [3], i.e. an entry e is irrelevant if it is empty (unused) or if its weight $n_e^{(t)}$ does not exceed one point per snapshot (cf. [3]). In contrast to [3], where such entries are only used to avoid split propagation, we explicitly check for irrelevant entries already during descent to actively encourage a split on lower levels, because a split below a node that contains an irrelevant entry does not

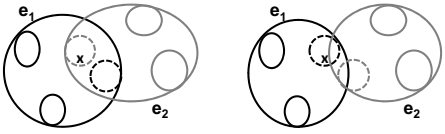

Fig. 1. Look ahead and reorganization

cause an increase of the tree height, but yields a better usage of the available memory by avoiding unused entries. In case of a leaf node we return the irrelevant entry as the one for insertion, for an inner node we set the *encSplit* flag.

Second we calculate the noise probability for the insertion object and mark it as noise if the probability exceeds a given threshold. This *noiseThreshold* constitutes a parameter of our algorithm and we evaluate it in Section 3.

Definition 2. ***Noise probability.*** *For a node node and an object o, the noise probability of o w.r.t. node is* $np(o) = \min_{e_i \in node} \{\{dist(o, \mu_{e_i})/r_{e_i}\} \cup \{1\}\}$ *where* e_i *are the entries of node,* r_{e_i} *the corresponding radius (standard deviation in case of cluster features) and* $dist(o, \mu_{e_i})$ *the euclidean distance from the object to the mean* μ_{e_i}*.*

Finally we determine the entry for further insertion. If the current node is a leaf node we return the entry that has the smallest distance to the insertion object. For an inner node we perform a local look ahead to avoid overlapping, i.e. we take the second closest entry e_2 into account and check whether it overlaps with the closest entry e_1. Figure 1 illustrates an example.

If an overlap occurs, we perform a local look ahead and find the closest entries e_1* and e_2* in the child nodes of candidates e_1 and e_2 (dashed circles in Figure 1 left). Next we calculate the radii of e_1 and e_2 if we would swap e_1* and e_2*. If they decrease, we perform the swapping and update the cluster features on the one level above (Figure 1 right). The closest entry that is returned is the one containing the closest child entry, i.e. e_1 in the example.

The closest entry is calculated both for the insertion object and for the hitchhiker (if any). If the two have different closest entries, the hitchhiker is stored in the buffer CF of its closest entry and the insertion objects continues alone (cf. Algorithm 1 line 1).

2.3 Noise

From the previous we know whether the current object has been marked as noise with respect to the current node. If so, the noise procedure is called. In this procedure noise items are added to the current noise buffer and it is regularly checked whether the aggregated noise within the buffer is no longer noise but a novel concept. Therefore, the identified object is first added to the noise buffer of the current node. To check whether a noise buffer has become a cluster, we calculate for the current node the average of its entries' weights $n^{(t)}$, their average density and the density of the noise buffer.

Definition 3. ***Density.*** *The density $\rho_e = n_e^{(t)}/V_e$ of an entry e is calculated as the ratio between its weighted number of points $n_e^{(t)}$ and the volume V_e that it encloses. The volume for d dimensions and a radius r is calculated using the formula for d-spheres, i.e. $V_e = C_d \cdot r^d$ with $C_d = \pi^{d/2}/\Gamma(\frac{d}{2}+1)$ where Γ is the gamma function.*

Having a representative weight and density for both the entries and the noise buffer, we can compare them to decide whether a new cluster emerged. Our intuition is, that a cluster that forms on the current level should be comparable to the existing ones in both aspects. Yet, a significantly higher density should also allow the formation of a new cluster, while a larger number of points that are not densely clustered are further on considered noise. To realize both criteria we multiply the density of the noise buffer with a sigmoid function, that takes the weights into account, before we compare it to the average density of the node's entries. As the sigmoid function we use the Gompertz function [2]

$$gompertz(n_{nb}, n_{avg}) = e^{-b(e^{-c \cdot n_{nb}})}$$

where we set the parameters b (offset) and c (slope) such that the result is close to zero ($t_0 = 10^{-4}$) if n_{nb} is 2 and close to one ($t_1 = 0.97$) if $n_{nb} = n_{avg}$ by

$$b = \frac{\ln(t_0)^{\frac{1}{1.0-(2.0/n_{avg})}}}{\ln(t_1)^{\frac{2}{n_{avg}-2}}} \qquad c = -\frac{1}{n_{avg}} \cdot \ln(-\frac{\ln(t_1)}{b})$$

Definition 4. ***Noise-to-cluster event.*** *For a node node $= (e_1, \ldots, e_k, CF_{nb}^{(t)})$ with average weight $n_{avg} = \frac{1}{k}\sum n_{e_i}^{(t)}$ and average density $\rho_{avg} = \frac{1}{k}\sum \rho_{e_i}$ the noise buffer $CF_{nb}^{(t)}$ becomes a new entry, if*

$$gompertz(n_{nb}^{(t)}, n_{avg}) \cdot \rho_n \geq \rho_{avg}$$

We check whether the noise buffer has become a cluster by now, if the encourage split flag is set to true. Note that a single inner node on the previous path with an irrelevant entry, i.e. old or empty, suffices for the encourage split flag to be true. Moreover, the exponential decay (cf. [3]) regularly yields outdated clusters. Hence, a noise buffer is likely to be checked.

If the noise buffer has been classified as a new cluster, we create a new entry from it and insert this entry into the current node. Additionally we create a new empty node, which is flagged as *liar*, and direct the pointer of the new entry to this node. Figure 2 a-b) illustrate this noise to cluster event.

2.4 Novelty

In [3] new nodes were only created at the leaf level, such that the tree grew bottom up and was always balanced. The LiarTree allows noise buffers to transform to new clusters, i.e. we get new entries and, more importantly, new nodes

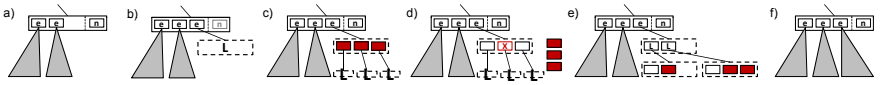

Fig. 2. The liar concept: a noise buffer can become a new cluster and the subtree below it grows top down, step by step by one node per object

Algorithm 2. liarProc (*liarNode*, x)
// refines the model to reflect novel concepts

create three new entries with *dim* dimensions $e_{new}[\,]$;
for *($d = 1$ to dim)* **do**
  $e_{new}[d \bmod 3].LS[d] = (e_{parent}.LS[d])/3 + \text{offset}_A[d]$;
  $e_{new}[(d+1) \bmod 3].LS[d] = (e_{parent}.LS[d])/3 + \text{offset}_B[d]$;
  $e_{new}[(d+2) \bmod 3].LS[d] = (e_{parent}.LS[d])/3 + \text{offset}_C[d]$;
  $e_{new}[d \bmod 3].SS[d] = F[d] + (3/e_{parent}.N) \cdot (e_{new}[d \bmod 3].LS[d])^2$;
  $e_{new}[(d+1) \bmod 3].SS[d] = F[d] + (3/e_{parent}.N) \cdot (e_{new}[(d+1) \bmod 3].LS[d])^2$;
  $e_{new}[(d+2) \bmod 3].SS[d] = F[d] + (3/e_{parent}.N) \cdot (e_{new}[(d+2) \bmod 3].LS[d])^2$;
end for
insert x into the closest of the new entries;
if *(liarNode is a liar root)* **then**
  insert new entries into *liarNode*;
else
  remove e_{parent} in parent node;
  insert new entries into parent node;
  split parent node (stop split at liar root);
end if
if *(non-empty liar nodes reach leaf level)* **then**
  remove all liar flags in correspond. subtree ;
else
  create three new empty liar nodes under $e_{new}[\,]$;
end if

within the tree. To avoid getting an increasingly unbalanced tree through noise-to-cluster events, we treat nodes and subtrees that represent novelty differently. The main idea is to let the subtrees underneath newly emerged clusters (entries) grow top down step by step with each new object that is inserted into the subtree until their leaves are on the same height as the regular tree leaves. We call leaf nodes that belong to such a subtree *liar nodes*, the root is called *liar root*. When we end up in a liar node during descend (cf. Algorithm 1), we call the liar procedure which is listed in Algorithm 2.

Definition 5. ***Liar node.*** *A liar node is a node that contains no entry. A liar root is an inner node of the liar tree that has only liar nodes as leafs in its corresponding subtree and no other liar root as ancestor.*

Figure 2 illustrates the liar concept, we will refer to the image when we describe the single steps. A liar node is always empty, since it has been created as an empty node underneath the entry e_{parent} that is pointing to it. Initially the liar root is created by a noise-to-cluster event (cf. Figure 2 b)). To let the subtree under e_{parent} grow in a top down manner, we have to create additional new entries e_i (cf. solid (red) entries in Figure 2). Their cluster features CF_{e_i} have to fit the CF summary of e_{parent}, i.e. their weights, linear and quadratic sums have to sum up to the same values. We create three new entries (since a fanout of three was shown to be optimal in [3]) and assign each a third of the weight from e_{parent}. We displace the new means from the parent's mean by adding three different offsets to its mean (a third of its linear sum, cf. lines 2 to 2). The offsets are calculated per dimension under the constraint that the new entries have positive variances. We set one offset to zero, i.e. $\text{offset}_A = 0$. For this special case, the remaining two offsets can be determined using the weight n_e^t and variance $\sigma_e^2[i]$ of e_{parent} per dimension as follows

$$\text{offset}_B[i] = \sqrt{\frac{1}{6} \cdot (1 - \left(\frac{1}{3}\right)^4) \cdot (n_e^t) \cdot \sigma_e^2[i]}, \qquad \text{offset}_C[i] = -\text{offset}_B[i]$$

The zero offset in the first dimension is assigned to the first new entry, in the second dimension to the second entry, and so forth using modulo counting (cf. lines 2 to 2). If we would not do so, the resulting clusters would lay on a line, not representing the parent cluster well. The squared sums of the three new entries are calculated in lines 2 to 2. The term $F[d]$ can be calculated per dimension as

$$F[d] = \frac{n_e^t}{3} \cdot \left(\frac{\sigma_e[d]}{3}\right)^4$$

Having three new entries that fit the CF summary of e_{parent}, we insert the object into the closest of these and add the new entries to the corresponding subtree (lines 2 to 2). If the current node is a liar root, we simply insert the entries (cf. Figure 2 c)). Otherwise we replace the old parent entry with the three new entries (cf. Figure 2 d)). We do so, because e_{parent} is itself also an artificially created entry. Since we have new data, i.e. new evidence, that belongs to this entry, we take this opportunity to detail the part of the data space and remove the former coarser representation. After that, overfull nodes are split (cf. Figure 2 d-e)). If an overflow occurs in the liar root, we split it and create a new liar root above, containing two entries that summarize the two nodes resulting from the split (cf. Figure 2 e)). The new liar root is then put in the place of the old liar root, whereby the height of the subtree increased by 1 and it grew top down (cf. Figure 2 e)).

In the last block we check whether the non empty leaves of the liar subtree already reach the leaf level. In that case we remove all liar flags in the subtree, such that it becomes a regular part of the tree (cf. line 2 and Figure 2 f)). If the subtree does not yet have full height, we create three new empty liar nodes (line 2), one beneath each newly created entry (cf. Figure 2 c)).

2.5 Insertion and Drift

Once the insertion object reaches a regular leaf, it is inserted using the leaf procedure (cf. algorithm 1 line 1). If there is no time left, the object and its hitchhiker are inserted such that no overflow, and hence no split, occurs. Otherwise, the hitchhiker is inserted first and, if a split is encouraged, the insertion of the hitchhiker can also yield an overflowing node. This is in contrast to the ClusTree, where a hitchhiker is merged to the closest entry to delay splits. In the LiarTree we explicitly encourage splits to make better use of the available memory (cf. Definition 1). After inserting the object we check whether an overflow occurred, split the node and propagate the split.

Three properties of the LiarTree help to effectively track drifting clusters. The first property is the aging, which is realized through the exponential decay of leaf and inner entries as in the ClusTree (cf. [3]), a proof of invariance can be found in [3]). The second property is the fine granularity of the model. Since new objects can be placed in smaller and better fitting recent clusters, older clusters are less likely to be affected through updates, which gradually decreases their weight and they eventually disappear. The third property stems from the novel liar concept, which separates points that first resemble noise and allows for transition to new clusters later on. These transitions are more frequent on levels close to the leaves, where cluster movements are captured by this process.

2.6 Summary

To insert a new object, the closest entry in the current node is calculated. While doing this, a local look ahead is performed to possibly improve the clustering quality by reduction of overlap through local reorganization. If an object is classified as noise, it is added to the current node's noise buffer. Noise buffers can become new clusters (entries) if they are comparable to the existing clusters on their level. Subtrees below newly emerged clusters grow top down through the liar concept until their leaves reach the regular leaf level.

Obviously the LiarTree algorithm has time complexity logarithmic in its model size, i.e. the number of entries at leaf level, since the tree is balanced (logarithmic height), the loop has only one iteration per level (cf. Alg. 1) and any procedure is maximally called once followed directly by a **break** statement.

3 Experiments

We compare our performance to the ClusTree algorithm [3] and to the well known CluStream algorithm from [1] using synthetic data as in [3]. To compare to the CluStream approach we used a maximal tree height of 7 and allowed CluStream to maintain 2000 micro clusters. We calculate precision and recall using a Monte Carlo approach, i.e. for the recall we generate points inside the ground truth and check whether these are included in the found clustering, for the precision we reverse this process, i.e. we generate points inside the found clustering and check whether they are inside the ground truth. Figure 3 shows

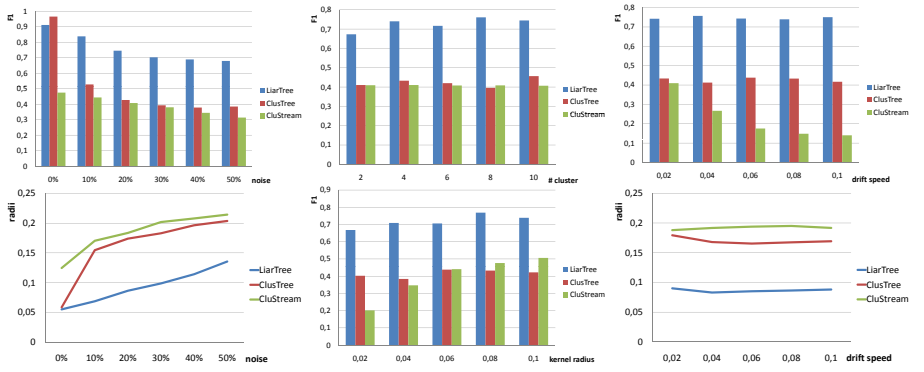

Fig. 3. Left: F1 measure and resulting radii for LiarTree, ClusTree and CluStream for different noise levels. Middle: Varying the data stream's number of clusters and their radius. Rigth: Varying the drift speed for LiarTree, ClusTree and CluStream

the resulting F1 measure and the resulting average radii of the clusters for the three approaches. In the left graphs we see that the LiarTree outperforms both competing approaches in the presence of noise, proving its novel concepts to be effective. Varying the parameters of the data stream (cf. remaining graphs) does not impair the dominance of the LiarTree.

4 Conclusions

In this paper we presented a novel algorithm for anytime stream clustering called LiarTree, which automatically adapts its model size to the stream speed. It consists of a tree structure that represents detailed information in its leaf nodes and coarser summaries in its inner nodes. The LiarTree avoids overlapping through local look ahead and reorganization and incorporates explicit noise handling on all levels of the hierarchy. It allows the transition from local noise buffers to new entries (micro clusters) and grows novel subtrees top down using its liar concept, which makes it robust against noise and changes in the distribution of the underlying stream.

Acknowledgments. This work has been supported by the UMIC Research Centre, RWTH Aachen University, Germany.

References

1. Aggarwal, C.C., Han, J., Wang, J., Yu, P.S.: A framework for clustering evolving data streams. In: VLDB, pp. 81–92 (2003)
2. Bowers, N.L., Gerber, H.U., Hickman, J.C., Jones, D.A., Nesbitt, C.J.: Actuarial Mathematics. Society of Actuaries, Itasca (1997)
3. Kranen, P., Assent, I., Baldauf, C., Seidl, T.: Self-adaptive anytime stream clustering. In: IEEE ICDM, pp. 249–258 (2009)