2009 Ninth IEEE International Conference on Data Mining

Self-Adaptive Anytime Stream Clustering

Philipp Kranen* Ira Assent† Corinna Baldauf* Thomas Seidl*

*RWTH Aachen University, Germany
{kranen, baldauf, seidl}@cs.rwth-aachen.de

†Aalborg University, Denmark
ira@cs.aau.dk

***Abstract*—Clustering streaming data requires algorithms which are capable of updating clustering results for the incoming data. As data is constantly arriving, time for processing is limited. Clustering has to be performed in a single pass over the incoming data and within the possibly varying inter-arrival times of the stream. Likewise, memory is limited, making it impossible to store all data. For clustering, we are faced with the challenge of maintaining a current result that can be presented to the user at any given time.**

In this work, we propose a parameter free algorithm that automatically adapts to the speed of the data stream. It makes best use of the time available under the current constraints to provide a clustering of the objects seen up to that point. Our approach incorporates the age of the objects to reflect the greater importance of more recent data. Moreover, we are capable of detecting concept drift, novelty and outliers in the stream. For efficient and effective handling, we introduce the ClusTree, a compact and self-adaptive index structure for maintaining stream summaries. Our experiments show that our approach is capable of handling a multitude of different stream characteristics for accurate and scalable anytime stream clustering.

***Keywords*-stream clustering, anytime algorithms, self-adaptive algorithms**

I. Introduction

Analysis of streaming data is gaining importance as sensors or other data gathering devices are widely deployed. Streams constitute data values or tuples that need to be processed as they arrive. With the wide applicability of streaming data, clustering of streaming data has recently received much attention in data mining research. The goal is to cluster the objects within the stream continuously, such that there is always an up-to-date clustering of all objects seen so far. As opposed to clustering of a fixed data set that is available entirely prior to the data mining analysis, clustering of streaming data poses additional challenges.

Single pass clustering. In streaming environments, data arrives continuously. This means that clustering streams has to be performed in a single pass over the data in an online fashion.

Limited memory. Since data streams are assumed to be endless, storing each arriving object is simply not feasible. Any streaming clustering model has to adhere to memory constraints.

Limited time. The algorithm has to be able to keep up with the speed of the data stream. Clustering of the data cannot take longer than the average time between any two objects in the stream. Clustering has to keep up with the stream to always maintain a current clustering model.

Varying time allowances. Many streams do not show a constant flow of data, but constitute bursty streams. This means that the time available to process any item in the stream may vary greatly. Examples include peak times for customer transactions or seasonal changes in consumer behavior. Existing stream clustering algorithms are not capable of handling such varying time allowances, unless they were to resort to the minimal time allowance in the stream. Clearly, this means downgrading to the worst case assumption.

Evolving data. It is important to take into account that the model underlying the data in the stream may change over time. For example, consumption patterns during holidays may differ from those that are seen the rest of the year. To capture such phenomena, stream clustering should be capable of clearly identifying such changes. Denoted as concept drift, changes in clusters should be reported separately. Likewise, newly created clusters, so-called novelty, and outliers should be detected as such.

Flexible number and size of clusters. Many clustering algorithms, e.g. from the family of partitioning algorithms, require parametrization of the number of clusters to be detected. While setting such a parameter is also difficult in traditional clustering, streams undergo changes that may cause clusters to emerge, disappear, merge, or split. As such, setting a fixed number of clusters for the stream would distort the model. Existing stream clustering algorithms have to fix the size of their model in advance, e.g. through a maximum number of micro clusters [1], even though such knowledge is usually not available apriori.

We propose a parameter free stream clustering algorithm *ClusTree* that is capable of processing the stream in a single pass, with limited memory usage. It always maintains an up-to-date cluster model, and reports concept drift, novelty, and outliers. For handling of varying time allowances, we propose an *anytime* clustering approach. Anytime algorithms denote approaches that are capable of delivering a result at any given point in time, and of using more time if it is available to refine the result. For clustering, this means that our algorithm is capable of processing even very fast streams, but also of using greater time allowances to refine

1550-4786/09 $26.00 © 2009 IEEE
DOI 10.1109/ICDM.2009.47

the clustering model. Moreover, our approach makes no apriori assumption on the size of the clustering model, but dynamically *self-adapts*. We show that our algorithm can be combined with existing techniques for aging objects in the stream using decay functions, reporting cluster snapshots at different points in time, and comparing views at different points in time [1], [2], [19]. To the best of our knowledge, our approach is the first anytime clustering algorithm for streams.

II. Related Work

There is a rich body of literature on stream clustering. Convex stream clustering approaches are based on a k-center clustering. [17] processes the data stream in chunks and clusters each chunk into k clusters using either k-means or LSEARCH, a k-median variant. The final clustering is then generated by clustering the stored results from the chunks. If the available space is exceeded, the individual results from each chunk are merged via clustering to allow reusing the space for new chunks. [20] uses k-means clustering and additionally maintains a list of objects that do not fit the current clustering w.r.t. a "global boundary" [20]. A reclustering is started once that list becomes too large. [1] maintains a fixed number of micro-clusters and assigns a new object to the closest micro-cluster if it falls within its "maximum boundary" [1]. If no match can be found, either the most outdated cluster is deleted or the closest two clusters are merged. The final clustering is computed in an offline component using a k-center clustering on the micro-clusters. [2] does not explicitly fix the number of maintained clusters, but uses a fixed total number of dimensions, i.e. the clusters are maintained only in their most significant dimensions. As in [1] a new object is checked against each maintained cluster and eventually new clusters are created and the least recently updated cluster is deleted.

Micro-clusters (also called clustering features) have been introduced earlier for non-streaming contexts for speeding up clustering in large databases. In [28], a hierarchical index is maintained for faster access. This approach does not study streams nor does it provide anytime capabilities.

Detecting clusters of arbitrary shape in streaming data has been proposed using kernels [13], graphs [16], fractal dimensions [4] and density based clustering [6], [7]. [6] maintains a number of micro-clusters in an online component and applies a common density based clustering on these micro-clusters in an offline component. [7] and [4] both follow a grid based approach and store the density for populated grid cells in an online component. [7] finds clusters of arbitrary shape by combining neighboring cells with a density based approach, while [4] uses the fractal dimension of a cluster to determine the cells that belong to it. In Section III-E we discuss how our technique can be flexibly combined with these approaches.

None of the above approaches allows for anytime clustering nor for adapting the clustering model to the stream speed in an online fashion. Anytime algorithms denote approaches that are capable of delivering a result at any given point in time, and of using more time if it is available to refine the result. Anytime data mining algorithms such as top k processing [3], anytime learning [21], [26] and anytime classification [8], [14], [15], [18], [23], [27] are a very active field of research. An anytime clustering approach for time series has been proposed in [25], which is not directly targeted at streaming data. Our approach thus constitutes the first anytime stream clustering algorithm.

Finally, different approaches have been proposed to present an up-to-date view on the clustering result or to determine and visualize concept drift and novelty in stream clustering. [24] focuses on detecting a change in the underlying stream by means of the minimal description length. The pyramidal time frame proposed in [1] enables the view on clusterings of arbitrary time horizons defined by the user. [19] categorizes cluster transitions into internal and external transitions and describes how to detect these. [2], [22], [13] employ an exponential decay function to weigh down the influence of older data, thus focusing on keeping an up to date view of the data distribution. We show the compatibility of our approach to the visualization and analysis methods from [1], [2], [19] in Section III-E.

III. Self-adaptive anytime stream clustering

We propose self-adaptive anytime stream clustering that relies on an index structure for storing and maintaining a compact view of the current clustering. The size of our clustering model automatically adapts to the stream speed, which can not be achieved by any buffering outside the storage structure. Moreover, we do not want to drop any data object, but want to preserve a complete model. Hence, any object from the stream is inserted into the index, and possibly merged with aggregates of previously inserted objects. We describe strategies for dealing with varying time constraints for anytime clustering.

As mentioned in the Introduction, stream clustering needs to maintain a representation of the current clustering in limited memory that can be easily updated incrementally. Most importantly, anytime clustering requires the possibility of interrupting the process at any given point in time.

A. Micro-clusters and anytime insert

Our approach is based on micro-clusters as compact representations of the data distribution. By maintaining measures for incremental computation of mean and variance of micro-clusters, the infeasible access to all past stream objects is no longer necessary.

Micro-clusters are a popular technique in stream clustering or scaling clustering to large data sets [28], [2], [1] to create and maintain compact representations of the current

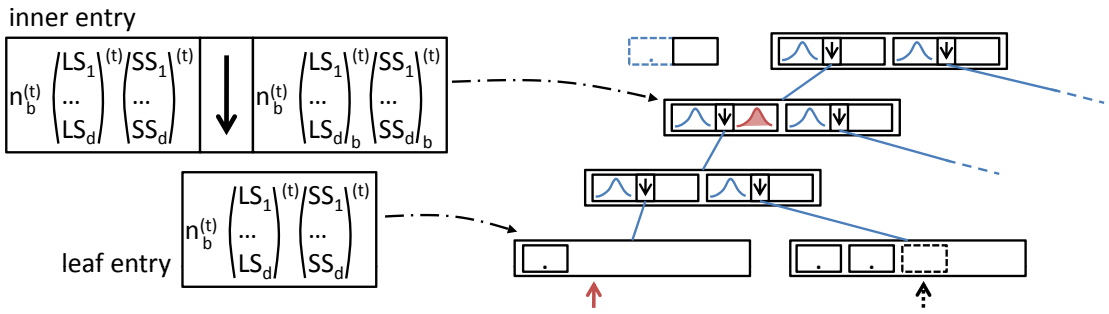


Figure 1. Left: inner node and leaf node structure. Right: insertion object, hitchhiker and buffer.

clustering. Instead of storing all incoming objects, a cluster feature tuple $CF = (n, LS, SS)$ of the number n of represented objects, their linear sum LS, and their squared sum SS is maintained. This tuple suffices for computing mean and variance, and can be incrementally updated. Any cluster feature (CF) then represents a micro-cluster, i.e. a set of objects, and the main characteristics of its distribution. With this, objects can be easily assigned to the most similar micro-cluster incrementally. Existing micro-cluster approaches lack support for varying stream inter arrival times. It is therefore crucial to provide the means for anytime clustering and self-adaptation to stream speed. We propose maintaining cluster features (CFs) by extending index structures from the R-tree family [11], [5], [18]. Such hierarchical indexing structures provide the means for efficiently locating the right place to insert any object from the stream into a micro-cluster. The idea is to build a hierarchy of micro-clusters at different levels of granularity. Given enough time, the algorithm descends the hierarchy in the index to reach the leaf entry that contains the micro-cluster that is most similar to the current object. If this micro-cluster is similar enough, it is updated incrementally by this object's values. Otherwise, a new micro-cluster may be formed.

The important observation for anytime clustering of streaming data, however, is, that there might not always be enough time to reach leaf level to insert the object. We therefore provide novel strategies for anytime inserts.

There are several possibilities for handling object arrivals before the current object insert reaches leaf level. The straightforward solution keeps a **global queue**. This approach is very simple, but it may require an infinite buffer. And we may never have the time to empty the queue, resulting in outdated clustering results. To reduce memory consumption, one could maintain a **global aggregate**, i.e. instead of the queue a single cluster feature. However, aggregating arbitrary objects loses too much information as they might be diverse.

To maintain the necessary information for clustering, and to ensure that any newly arriving object can be inserted at once, we propose interrupting the insertion process. The object has to be temporarily stored in a **local aggregate** from which we can continue at a later time. This yields foreseeable space demands like with a global aggregate, albeit slightly larger ones. For the invested space, we obtain a greater accuracy. The great advantage of local aggregates over local queues is that we can easily use the time for regular inserts to take a buffered local aggregate along as a "hitchhiker". Moreover, they can be naturally integrated into the tree structure. We will discuss this in more detail shortly, after describing the structure of our ClusTree hierarchical index for maintaining the micro-cluster information.

B. The ClusTree

Our ClusTree approach consists of a hierarchy of entries that describe the cluster feature properties of their respective subtrees. The structure of an inner entry and a leaf entry is illustrated in the left part of Figure 1: Each entry contains a cluster feature of the number of objects n that were aggregated, their dimension-wise linear sum LS, and squared sum SS, as well as a pointer to the respective subtree. We propose integrating local aggregates into the tree structure as temporary entries, so, additionally, an inner entry provides a buffer b for temporary insertions of local aggregates (CFs). Leaf nodes' entries do not contain a buffer, since inserts at leaf level are final.

Definition 1: **ClusTree.**

A ClusTree with fanout parameters m, M and leaf node capacity parameters l, L is a balanced multi-dimensional indexing structure with the following properties:

- an inner node $node_s$ contains between m and M entries. Leaf nodes contain between l and L entries. The root has at least one entry.
- an entry in an inner node of a ClusTree stores:
  - a cluster feature of the objects it summarizes.
  - a cluster feature of the objects in the buffer. (May be empty.)
  - a pointer to its child node.
- an entry in a leaf of a ClusTree stores a cluster feature of the object(s) it represents.
- a path from the root to any leaf node has always the same length (balanced).

The tree is created and updated like any multidimensional index such as R-tree, R*-tree, etc. [11], [5], [18]. Unlike the minimum bounding rectangles that they maintain in addition to the objects, we store only CFs in the ClusTree. For insertion, we descent into the subtree with the closest mean with respect to Euclidean distance. Splitting is based on pairwise distances between the entries, where entries are combined into two groups such that the sum of the intra-group distances is minimal. We will show in Section IV that $M = 3$ is a good choice, hence there are maximally six pairwise distances per node yielding a fast split operation.

The important property that reflects anytime capability of the ClusTree is its buffer in each entry. It serves as a temporary storage place of aggregates or objects that do not reach leaf level during insertion. Whenever insertion is interrupted, the current CF is simply stored in the buffer of the entry that corresponds to the subtree into which to descend next. At any future time when this subtree is next accessed, the temporary entry in the buffer is taken along as a "hitchhiker". This makes sure that future descent down the same subtree is used for continuing the insertion process. Whenever the descent destination of the current insertion CF and the hitchhiker differ, the latter is placed in the corresponding buffer again to wait for the next ride down the tree.

The right part of Figure 1 illustrates this process. Assume that the insertion object (drawn blue in the dashed box to the left of the root) belongs to the leaf that is marked by the dashed arrow (at the second leaf). Assume also, that the leftmost entry on the second level has a filled buffer (second distribution symbol in the entry), which belongs to a different leaf than the insertion object (indicated by the red solid arrow at the first leaf). The insertion object first descends to the second level, and next descends into the left entry. It picks up the left entry's buffer in its buffer CF for hitchhikers (depicted as the solid box at the right of the insertion object). The insertion object descends to level three, taking the hitchhiker along. Because the hitchhiker and the insertion object belong to different subtrees, the hitchhiker is stored in the buffer of the left entry on the third level (to be taken along further down in the future) and the insertion object descends into the right entry alone to become (part of) a leaf entry.

Our buffer concept and the algorithmic idea of taking hitchhikers along are key to our anytime clustering algorithm. It allows the algorithm to be interrupted at any point in time and making best use of future descents down the tree. Moreover, unlike global aggregates, objects are kept separate as long as time permits.

When a leaf node is reached and the insertion would cause a split, the algorithm checks whether there is still time left. If there is no time for a split, the closest two entries are merged. For tracking of concept drift, novelty, etc. in the output clustering, leaf node entries contain a unique id. When they are created they are assigned a unique number in increasing order. When entries are merged, this is recorded as a pair of ids in a merging list.

The ClusTree can be initialized to improve the starting structure of the tree. Given an initial set of objects, each is transformed to a new CF. The leafs and internal nodes may be ordered for best structural properties through recursive top-down partitioning along the dimension with the largest variance and such that each partition contains equally many objects. Or any clustering algorithm, e.g. expectation maximization (EM) [9] or k-means, can be used to group the objects in a top-down or bottom-up fashion to initialize the tree. Please note that our focus is not on optimizing the initialization phase, and our experiments are performed without it.

C. Maintaining an up-to-date clustering

In order to maintain an up-to-date view, we would like new objects to be more important than older objects. A common solution is to weigh objects with an exponential time-dependent decay function $\omega(\Delta t) = \beta^{-\lambda \Delta t}$. The decay rate λ controls how much more one favors new objects compared to old ones. The higher λ is, the faster the algorithm "forgets" old data. We chose to set $\beta = 2$. For this basis the half life of objects is $\frac{1}{\lambda}$.

To incorporate decay, temporal information has to be added to the ClusTree nodes. We ensure that the inner entries of the ClusTree still summarize their subtrees accurately by making elements of a cluster feature vector dependent on the current time t:

$$n^{(t)} = \sum_{i=1}^{n} \omega(t - ts_i), \quad LS^{(t)} = \sum_{i=1}^{n} \omega(t - ts_i) \cdot x_i,$$

$$SS^{(t)} = \sum_{i=1}^{n} \omega(t - ts_i) \cdot x_i^2$$

n denotes the (unweighted) number of contributing objects and ts_i is the timestamp at which object x_i was added to the CF.

We know that additive properties of cluster features are preserved, and also temporal multiplicity [2]: If no object is added to a $CF^{(t)}$ during the time interval $[t, t + \Delta t]$ then

$$CF^{(t+\Delta t)} = \omega(\Delta t) \cdot CF^{(t)}.$$

Details on this property and the corresponding proof can be found in [2].

Each insertion object x carries the timestamp ts_x of its arrival time. Furthermore, each entry e_s has a timestamp $e_s.ts$ specifying its last update. We use it to compute the time that passed between the last update of an object and t_x, which is the input of the decay function. Upon descending into a node, we update all entries e_s in the node to t_x by position-wise multiplication with the decay function and resetting the timestamp: $e_s.CF \leftarrow \omega(t_x - e_s.\text{ts}) \cdot e_s.CF, \quad e_s.\textit{buffer} \leftarrow$

$\omega(t_x - e_s.\text{ts}) \cdot e_s.\textit{buffer}$, $e_s.\text{ts} \leftarrow t_x$. Please note that entries in the same node always have the same timestamp, as we update all entries in the node we descend into.

We now show that inner entries summarize their subtrees correctly. We derive an invariant that incorporates the time aspect. The cluster feature of a parent entry e_s that was last updated at $t + \Delta t$ equals the sum of the CFs of the entries in its child node updated from time t of their last update to the parent's time plus the parent's buffer.

Lemma 1 (ClusTree Invariant): For each inner entry e_s with timestamp $t + \Delta t$ and decay function $\omega(\Delta t) = 2^{-\lambda \Delta t}$ it holds

$$e_s.CF^{(t+\Delta t)} = (\omega(\Delta t) \cdot \sum_{i=1}^{\nu_s} e_{s \circ i}.CF^{(t)}) + e_s.\textit{buffer}^{(t+\Delta t)}$$

Proof: Each inner entry e_s is created first due to a split. Its summary is calculated directly as the sum of the cluster features in its child node entries $e_{s \circ i}$. The child node entries are all on the same time, because we update all entries in a node. The timestamp of the children is the insertion time t of the object x that caused the split. There can only be a change in one of the $e_{s \circ i}$, if there was first a change in e_s, because we always start from the root and descend downwards.

Take the case of updating parent entry e_s (with filled buffer) to the new time $t + \Delta t + \Gamma t$, and addition of object y, where y descends into $node_s$ and gives the buffer a lift. Upon descending into $node_s$, all its entries $e_{s \circ i}$ are updated and y is added to the CF of exactly one of the $e_{s \circ i}$. e_s has a buffer, which y takes along. The buffer is also added to exactly one of the child entries' cluster features.

Following the above reasoning, we know that after updating e_s it holds that:

$$\begin{aligned} e_s.CF^{(t+\Delta t+\Gamma t)} &= (\omega(\Delta t + \Gamma t) \cdot \sum_{i=1}^{\nu_s} e_{s \circ i}.CF^{(t)}) \\ &\quad + e_s.\text{buffer}^{(t+\Delta t+\Gamma t)}. \end{aligned}$$

Because y descends into $node_s$, we update the child entries:

$$= \sum_{i=1}^{\nu_s} e_{s \circ i}.CF^{(t+\Delta t+\Gamma t)} + e_s.\text{buffer}^{(t+\Delta t+\Gamma t)}.$$

Now we give $e_s.\text{buffer}^{(t+\Delta t+\Gamma t)}$ a lift. Afterwards $e_s.\text{buffer}^{(t+\Delta t+\Gamma t)}$ contains zeros, and the values that it held before are added to the cluster feature of one of the child entries. Also adding y on both sides of the equation, once to $e_s.CF$ and once to the CF of one of the child node entries, leaves the invariant unchanged. This is also true for "hitchhiking" objects temporarily in a buffer o (replace $y.CF^{(t+\Delta t+\Gamma t)}$ with $y.CF^{(t+\Delta t+\Gamma t)} + o.CF^{(t+\Delta t+\Gamma t)}$), and if $node_s$ is a leaf node.

The last case in which we need to check violate the invariant is a split. Let us consider the split of a leaf node $node_{s \circ i}$. Then two summaries in $node_s$ are computed from scratch. One overwrites the existing entry $e_{s \circ i}$ that pointed to the split node. The other one is the start of a new entry. The two new summaries naturally fulfill the invariant. The invariant also holds true for e_s, the entry pointing to $node_s$, because only the distribution of the summaries changed on the levels below e_s, not the total of the values. ■

Thus, maintaining a single additional field in each node with a timestamp value of its last update, and weighing according to the above scheme, ensures that decay with time is correctly captured. Note that weighing does not require additional memory; the weighted CFs simply replace the non-weighted cluster features in Def. 1.

Weighing with time provides us with an interesting way of avoiding splits, to save valuable time. If a node is about to be split, our algorithm checks whether the least significant entry can be discarded, because it no longer contributes significantly to the clustering. Assuming that a snapshot of the ClusTree is taken regularly after t_{snap} time, the significance is tested by checking whether the entry $\hat{e}$ with the smallest $n_{\hat{e}}^{(t)}$ satisfies

$$n_{\hat{e}}^{(t)} < \beta^{-\lambda \cdot t_{snap}} \tag{1}$$

If this is the case, $\hat{e}$ is discarded, making room for the entry to be inserted, and avoiding a split. The summary statistics of $\hat{e}$ are subtracted from the corresponding path up to the root. Note that according to Equation 1 no entry is discarded if a new object has been added to it after the last snapshot has been taken. Moreover, Equation 1 guarantees that each entry is stored in at least one snapshot.

We discuss in Section III-E how the ClusTree results can be used to detect clusters of arbitrary shape and time horizon. Moreover, applicability of recent approaches to concept drift detection is shown.

D. Speed-up through aggregation

What happens to our index structure when it faces an exceptionally fast data stream? If insertion is interrupted at the top levels most of the time, the root and upper levels of the tree aggregate a lot of objects in their buffers that have little chance of getting a lift down to a leaf. Worse yet, dissimilar objects which belong to different subtrees and leaves become inseparable in a buffer. The quality of our results is bound to deteriorate if we are constantly interrupted on higher levels.

We propose a speed-up through aggregation before insertion: If we do not insert each object individually, there is more time to descend deeper with an aggregate of objects. Naively, one could add up a certain number m of incoming objects, insert the aggregate, sum up the next m objects, and so on. This is essentially a global aggregate with the problem of merging arbitrary objects, even very dissimilar ones, to the same aggregate.

Clearly we need to exercise some control over which objects should – literally – "go together". Ideally, we want

to aggregate objects that would end up in the same leaf if we could descend the tree with them. Most probably, the arriving objects are not all similar to each other, but we expect subgroups with inter-similarity – representatives of the clusters we also find in the tree.

Our solution is to create several aggregates for dissimilar objects. This makes sure that the objects summarized in the same aggregate are similar. To this end, we set a max_{radius} for the maximum distance of objects in the aggregate. max_{radius} does not need to be set by the user. We propose determining its value from the leaf level, as the average variance of the leaves. This way, the aggregates for fast streams do not deteriorate the quality of the tree disproportionally.

For very fast streams, we store interrupted objects in their closest aggregate with respect to the distance to the mean, if this distance is below max_{radius}. If max_{radius} is exceeded, we open up a new aggregate. We insert aggregates, just as we insert single objects. Whenever the insertion thread is idle, it simply picks the next aggregate (ordered first by the number of objects in the aggregates and then by their age). The number of aggregates is limited by the stream speed, i.e. it cannot exceed the number of distance computations that can be done between two arriving items. In the case of a varying data stream the maximum number of aggregates has to be set by the user, constituting the only parameter of our approach. The aggregation is done by a different thread, i.e. the insertion of aggregates works in parallel and is not affected in terms of processing time. If no aggregate violates the max-radius constraint, the fullest aggregate is inserted. If several aggregates are equally full, the oldest of these is inserted.

Figure 2 summarizes the complete ClusTree algorithm in a flow chart.

E. Cluster shapes and cluster transitions

The clustering resulting from the ClusTree is the set of CFs stored at leaf level, i.e. the finest representation maintainable w.r.t. the speed of the data stream. This can be seen as our online component and it allows for using various offline clustering approaches. Taking the means of the CFs as representatives we can apply a k-center clustering as in [17] or density based clustering as proposed in [6] to detect clusters of arbitrary shape. One main advantage of our approach is that we can maintain a way larger number of micro-clusters compared to other approaches [1], [2], [17], [6] and hence the offline clustering, e.g. density based, has finer input granularity.

Regarding cluster transitions, e.g. concept drift or novelty, many approaches proposed in the literature can directly be applied to the output of our ClusTree. Using the unique ids to every new leaf entry, we are able to track micro-clusters. Pyramidal time frames [1] allow the user to view clusterings of arbitrary time horizons. Furthermore, the ids allow us also

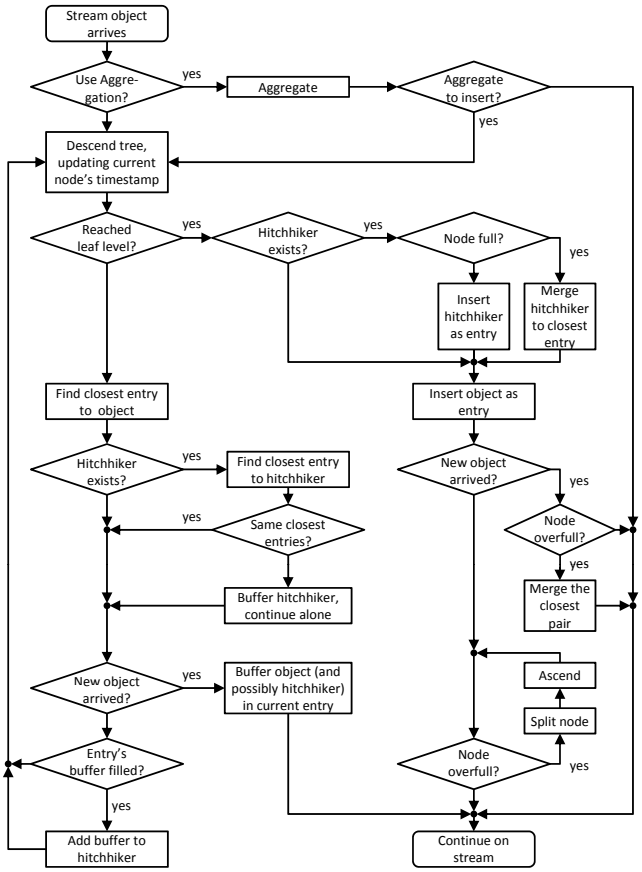


Figure 2. Flow chart of the ClusTree algorithm.

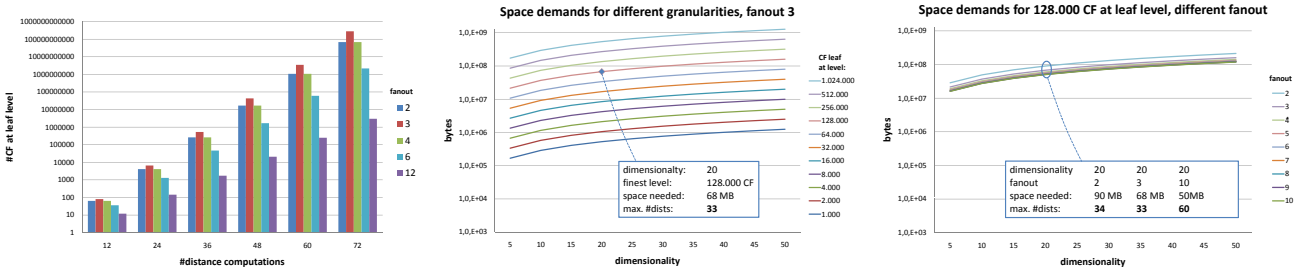
to apply the transition detection and distinction techniques described in [19], including outlier, novelty and concept drift detection.

IV. Analysis and Experiments

We assess the performance of the ClusTree in the following. First we examine the time and space complexity of building and maintaining a ClusTree in Section IV-A. In Section IV-B we evaluate the anytime clustering property of the ClusTree and show the benefits of our speed-up through aggregation. Finally, we demonstrate the adaptive clustering performance in Section IV-C by comparing our results against CluStream [1] and DenStream [6]. The algorithms were implemented in C, all experiments were run on Windows machines with 3GHz.

A. Time and space complexity

Our goal for efficient and effective clustering is a high granularity with low processing costs. Therefore we investigate the effect of the fanout and of the number of distance computations required to insert an object from the stream on the granularity, i.e. the number of cluster features (CFs) at leaf level. Figure 3(a) shows the results for fanouts from



(a) (b) (c)

Figure 3. a: Granularity (number of CF at leaf level) w.r.t. fanout and number of distance computations. b&c: Space consumption w.r.t granularity, fanout and dimensionality

2 to 12. Depending on the speed of the stream, 12 to 72 distance computations are possible before interruption (we chose multiples of 12 on the x-axis because it is the smallest multiple of all tested fanouts). As can clearly be seen in all groups of bars, a fanout of three yields the highest granularity independent of the number of distance computations, i.e. of stream speed.

Next we evaluate the space demands with respect to the dimensionality and granularity in Figure 3(b). It shows the results for a fanout of 3 (assuming 4 Bytes per value). The space demands for the ClusTree are moderate even for high granularities and high dimensionality. For 128.000 CF at leaf level and 20 dimensions the ClusTree only needs 68 MB space, while the number of distance computations to reach the leaf level is only 33. Fanout 3, dimensionality 20 and one million CF at the finest level consume roughly 500 MB, i.e. still main memory, and the number of distance computations is still less than 40. This is opposed to any stream clustering algorithm that maintains one million micro-clusters and checking a new item against each of these. CluStream [1] for example stores q micro-clusters and hence has to calculate q distances (plus possible delete $O(q)$ and merge $O(q^2)$ checks). We only need $O(log(q))$ many distance calculations and only store $O(q)$ CF (cf. Figure 3).

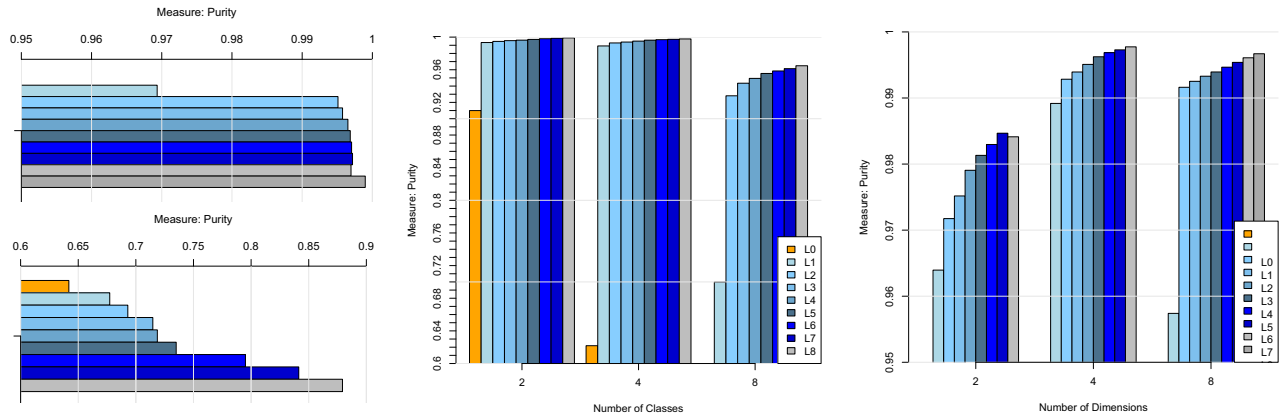
Figure 3(c) shows the space demand of the ClusTree for 128.000 CF at leaf level, different dimensionality and different fanout values. While a higher fanout yields less space demands, the number of distance computations that are necessary to reach the same granularity is significantly higher. Combining the results from Figure 3 we conclude that a fanout of 3 is the best choice in terms of time and space complexity.

Given the fanout of 3, the costs for a single split are low: 4 entries are present during split, hence 6 distances are calculated. A new node and one new entry for the parent node are created, and the old node and the old entry pointing to it are updated. In the worst case, the number of splits is equal to the height of the tree. Moreover, once the tree size is adapted to the stream speed and decay invalidates old entries, the number of splits is practically zero.

B. Anytime clustering and aggregation

To evaluate the clustering quality of the ClusTree we evaluate the average purity of the clusters on the different levels of the tree. To determine the purity we use synthetic as well as real world data that contains objects labeled with one of several classes. For a set K of CFs the purity is then calculated as the weighted average purity of all CFs in K: $\sum_{k=1}^{|K|} \frac{n_k}{n} . \frac{\max_c(n_{ck})}{n_k} = \frac{1}{n} \sum_{k=1}^{|K|} \max_c(n_{ck})$, where n_k is the number of objects in the CF k, n_{ck} those belonging to class c and $n = \sum_{k \in K} n_k$. The real world data set Forest Covertype is available from [12] and contains roughly 580.000 objects from 7 classes and 10 continuous attributes. To investigate the scalability of the ClusTree in terms of dimensionality and the number of clusters we use synthetic data sets containing 550.000 objects each (including 5% noise) and a varying number of attributes and classes. The clusters are generated as a hierarchy of Gaussians, where centers lie at a uniformly distributed angle and distance from their parents. To simulate a varying stream we generated the arrival intervals according to a Poisson process, a stochastic model that is often used to model random arrivals [10]. For the anytime experiments we generated a stream with an expected number of 90,000 points per second, i.e. $\lambda = 1/90000$.

Figure 4(a) shows the results for Forest Covertype (bottom) and the synthetic data set containing four classes and four dimensions (top). The results shown are the purity values after the complete data set has been processed. The top most bar (orange) represents the purity value at the root level and each following bar corresponds to the next deeper level. (Please note that for synthetic data the root level bar is not visible as the axis has been formatted to show the difference on the lower levels.) The resulting ClusTree had ten levels for the synthetic data and 9 levels for the Forest Covertype data set. The most interesting purity value is that of the leaf level representing the finest micro-clustering granularity. It is above 99% for the synthetic data and still 88% for Forest Covertype. The purity values on the higher levels of the tree give an indication for the clustering quality for higher stream speeds. We show further results on varying stream speed in Section IV-C. Except for the leaf level, the



(a) (b) (c)

Figure 4. a: Clustering purity on synthetic data (top) and on Forest Covertype [12] (bottom). b: Scalability w.r.t. the number of clusters; c: Scalability w.r.t. the number of dimensions.

purity values on the synthetic data set are above 95% on all levels, showing that the noise objects have been seperated very well. The purity decreases more significantly for the Forest Covertype data, but is still above 70% even three levels underneath the root.

Figures 4(b) and 4(c) show the results regarding scalability using the same anytime stream as before. We varied the number of classes from 2 to 8 at four dimensions (left) and the number of dimensions from 2 to 8 using 4 classes (right). For 2 and 4 classes the quality is consistently high on all levels, just the root level purity drops at 4 classes, and further (below the shown area) for 8 classes. Increasing the number of classes to 8 shows a higher impact on the root level and also one level below the root. Although the purity decreases also on the other levels it is still above 95% on six levels, indicating a good seperation of classes and even of noise objects. Comparing the results on different dimensionalities shows that the quality is similar for 4 to 8 dimensions, but lower in the 2-dimension case. This is due the fact that the overlapping of the classes is higher if the dimensionality decreases. However, once again the majority of the levels has a purity above 95%.

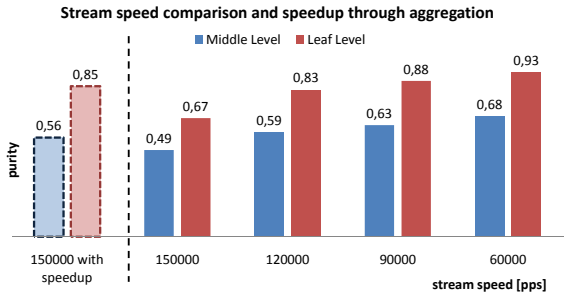


Figure 5. Purity with and without aggregation w.r.t. stream speed for Forest Covertype.

Finally, we evaluated different stream speeds, i.e. we varied the expected number of points per second (*pps*) from 60,000 to 150,000. Figure 5 shows the resulting purity values for the leaf level and the middle level of the ClusTree for Forest Covertype. For the slowest stream the purity on the leaf level reaches 93%. While the purity is still very good (87%) at 120,000 pps it drops below 70% for even faster streams with 150,000 pps. For our proposed speed-up through aggregation (cf. Section III), the results for 150,000 pps are shown in the left part of Figure 5. Thanks to the aggregation the purity on the leaf level is significantly improved.

C. Adaptive clustering

To evaluate the adaptive clustering behavior of the ClusTree we simulated constant data streams with different numbers of points per second using the Forest Covertype data set. We compare our performance against CluStream [1] and DenStream [6]. For all approaches we report the results of the online component, i.e. we analyze the properties of the resulting micro clusters and do not employ an additional offline component afterwards.

First of all we investigate the number of micro clusters that can be maintained by the individual approaches for different stream speeds. The results are shown in Figure 6, exact numbers are listed in Figure 7. As indicated, the ClusTree can maintain roughly 430,000 micro clusters at 49,000 pps. With a stream speed of 140,000 pps the ClusTree can still maintain 435 micro clusters. The competing approaches on the other hand can only process less than 10,000 pps when maintaining 500 micro clusters. This drastic difference is due to the hierarchical structure of the ClusTree which yields only a logarithmic amount of distance computations. In other words, the number of micro clusters we can maintain is exponential compared to CluStream or DenStream. This large number is beneficial,

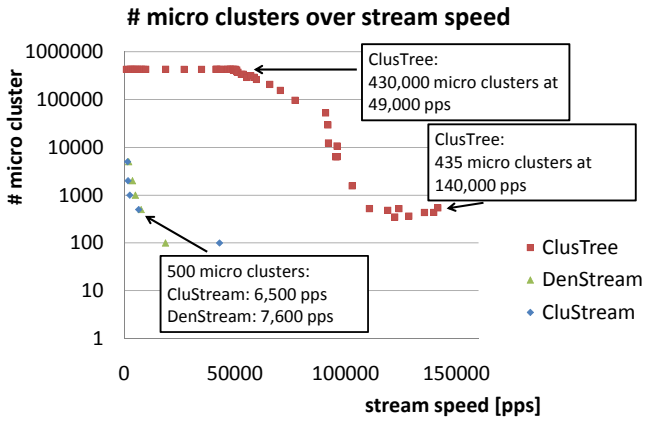


Figure 6. Number of micro clusters that can be maintained w.r.t. stream speed.

since the output of the online component is given to the offline component to compute the final clustering (using a clustering method of choice). A more detailed input to the final clustering enables more accurate results and detection of possible outliers. Moreover, the major advantage here is that the ClusTree automatically self-adapts to the stream speed without parametrization.

The question is at which price comes this benefit? Does the quality of the individual micro clusters deteriorate, because new points might not be added to the optimal micro cluster? To answer this question we evaluated the radius and the purity of the resulting micro clusters from all three approaches. Figure 8 shows the results for the ClusTree, results for CluStream and DenStream are listed in Figure 7.

For the radius we report the maximum as well as the 75 percentile and the median in Figure 8. Since the actual numbers are skewed we plot a moving average value. Naturally, with increasing stream speed, and hence decreasing number of micro clusters, the radii generally become larger. However, while we see a constant increase in the maximum value, the median and even the 75 percentile stays very low even for 100,000 to 150,000 pps. While DenStream produces larger micro clusters, CluStream shows a similar performance for the same amount of micro clusters. However, to maintain this amount of micro clusters CluStream can again only process slow streams where it is outperformed by our approach.

# MC	pps	radius (median)	radius (max.)	purity
DenStream				
5000	2000	0.21	151.8	0.53
2000	3700	0.24	195.1	0.55
1000	5000	3.35	160.9	0.66
500	7600	14.01	83.7	0.53
CluStream				
5000	1500	0.02	37.4	0.70
2000	1700	0.03	224.4	0.87
1000	2500	0.33	238.8	0.90
500	6500	0.58	177.6	0.62
ClusTree				
5000	80,000	0.44	13.8	0.72
2000	94,000	0.51	18.9	0.71
1000	105,000	0.55	21.8	0.70
500	120,000	2.25	29.7	0.67

Figure 7. Overall results on Forest Covertype.

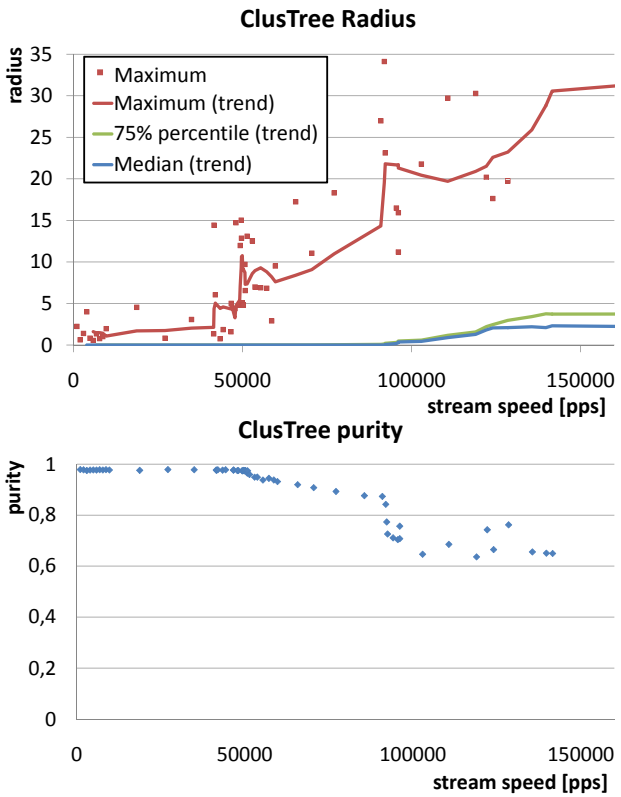


Figure 8. Radius (top) and purity (bottom) for ClusTree micro clusters w.r.t stream speed.

The purity values for CluStream, DenStream and our novel ClusTree approach underline the above findings (cf. Figure 7). DenStream does not exceed an average purity of 70%. Clustream shows a higher purity than the ClusTree for 1000 micro clusters (90% for CluStream vs. 78% for the ClusTree), but again these numbers are not comparable due to the huge difference in terms of points per second. In conclusion it can be said that the ClusTree can maintain an equal amount of micro clusters on streams that are faster by orders of magnitude and that it can maintain an exponential amount of micro clusters at equal stream speed while providing good results in terms of cluster size (radius) and quality (purity).

V. Conclusion

Clustering streaming data is of increasing importance in many applications. In this work, we proposed a parameter free index-based approach that self-adapts to varying stream speed and is capable of anytime clustering. Our ClusTree maintains the values necessary for computing mean and variance of micro-clusters. By incorporating local aggregates, i.e. temporary buffers for "hitchhikers", we provide a novel solution for easy interruption of the insertion process that can be simply resumed at any later point in time. For very fast streams, aggregates of similar objects allow insertion of groups instead of single objects for even faster processing. In comparison to recent approaches we have shown that the ClusTree can maintain the same amount of micro clusters at stream speeds that are faster by orders of magnitude and that for equal stream speeds our granularity is exponential w.r.t. competing approaches. Moreover, we discussed compatibility of our approach to finding clusters of arbitrary shape and to modeling cluster transitions and data evolution using recent approaches.

Acknowledgment

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

References

[1] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for clustering evolving data streams," in *VLDB*, 2003.

[2] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for projected clustering of high dimensional data streams," in *VLDB*, 2004, pp. 852–863.

[3] B. Arai, G. Das, D. Gunopulos, and N. Koudas, "Anytime measures for top-k algorithms," in *VLDB*, 2007, pp. 914–925.

[4] D. Barbará and P. Chen, "Using the fractal dimension to cluster datasets," in *KDD*, 2000, pp. 260–264.

[5] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," in *SIGMOD*, 1990, pp. 322–331.

[6] F. Cao, M. Ester, W. Qian, and A. Zhou, "Density-based clustering over an evolving data stream with noise," in *SDM*, 2006.

[7] Y. Chen and L. Tu, "Density-based clustering for real-time stream data," in *KDD*, 2007, pp. 133–142.

[8] D. DeCoste, "Anytime interval-valued outputs for kernel machines: Fast support vector machine classification via distance geometry," in *ICML*, 2002, pp. 99–106.

[9] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *J Royal Stat. Soc., B*, vol. 39, no. 1, pp. 1–38, 1977.

[10] R. Duda, P. Hart, and D. Stork, *Pattern Classification (2nd Ed.)*. Wiley, 2000.

[11] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD*, 1984, pp. 47–57.

[12] S. Hettich and S. Bay, "The UCI KDD archive http://kdd.ics.uci.edu," 1999.

[13] A. Jain, Z. Zhang, and E. Y. Chang, "Adaptive non-linear clustering in data streams," in *CIKM*, 2006, pp. 122–131.

[14] P. Kranen, "Using index structures for anytime stream mining," in *PhD Workshop VLDB, Lyon, France*, 2009.

[15] P. Kranen and T. Seidl, "Harnessing the strengths of anytime algorithms for constant data streams," *DMKD J, Special Issue on Selected Papers from ECML PKDD, Vol. 19, No. 2,*, pp. 245–260, 2009.

[16] S. Lühr and M. Lazarescu, "Incremental clustering of dynamic data streams using connectivity based representative points," *Data Knowl. Eng.*, vol. 68, no. 1, pp. 1–27, 2009.

[17] L. O'Callaghan, A. Meyerson, R. Motwani, N. Mishra, and S. Guha, "Streaming-data algorithms for high-quality clustering," in *ICDE*, 2002.

[18] T. Seidl, I. Assent, P. Kranen, R. Krieger, and J. Herrmann, "Indexing density models for incremental learning and anytime classification on data streams," in *EDBT*, 2009.

[19] M. Spiliopoulou, I. Ntoutsi, Y. Theodoridis, and R. Schult, "Monic: modeling and monitoring cluster transitions," in *KDD*, 2006, pp. 706–711.

[20] E. J. Spinosa, A. C. Ponce de Leon Ferreira de Carvalho, and J. Gama, "Olindda: a cluster-based approach for detecting novelty and concept drift in data streams," in *SAC*, 2007.

[21] W. N. Street and Y. Kim, "A streaming ensemble algorithm (sea) for large-scale classification," in *KDD*, 2001.

[22] K. Udommanetanakit, T. Rakthanmanon, and K. Waiyamai, "E-stream: Evolution-based technique for stream clustering," in *ADMA*, 2007, pp. 605–615.

[23] K. Ueno, X. Xi, E. J. Keogh, and D.-Y. Lee, "Anytime classification using the nearest neighbor algorithm with applications to stream mining," in *ICDM*, 2006, pp. 623–632.

[24] M. van Leeuwen and A. Siebes, "Streamkrimp: Detecting change in data streams," in *ECML/PKDD*, 2008.

[25] M. Vlachos, J. Lin, E. Keogh, and D. Gunopulos, "A wavelet-based anytime algorithm for k-means clustering of time series," in *WS Clust. High Dim. Data & App. (at ICDM)*, 2003.

[26] H. Wang, W. Fan, P. S. Yu, and J. Han, "Mining concept-drifting data streams using ensemble classifiers," in *KDD*, 2003, pp. 226–235.

[27] Y. Yang, G. I. Webb, K. B. Korb, and K. M. Ting, "Classifying under computational resource constraints: anytime classification using probabilistic estimators," *Machine Learning*, vol. 69, no. 1, 2007.

[28] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: an efficient data clustering method for very large databases," in *SIGMOD*, 1996.