FAKULTÄT FÜR MATHEMATIK, INFORMATIK UND STATISTIK
INSTITUT FÜR INFORMATIK

LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

LEHRSTUHL FÜR DATENBANKSYSTEME UND DATA MINING

Lecture Notes to

Big Data Management and Analytics

Winter Term 2018/2019

Stream Analytics

DBS

Outlook

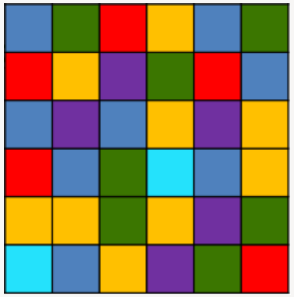
- Maintaining Histograms
- Change Detection
- Frequent Itemset Mining
- Clustering
- Classification

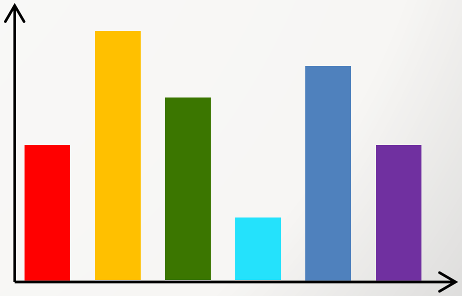
Maintaining Histograms

- Histograms are *graphical representations* of the distribution of numerical data

- Histograms estimate the probability distribution of a random variable

- Used for approximate query processing with error guarantees





Maintaining Histograms

- Histograms are defined by non-overlapping intervals or buckets

- A bucket is defined by its boundaries and its frequency count

- In case of streams:
  One never observes all values of a random variable

→ k-bucket histogram defined as
$]-\infty, b_1],]b_1, b_2], \dots,]b_{k-1}, \infty[$ buckets
with frequency counts $f_1, f_2, \dots, f_k$

Maintaining Histograms

In general: two types of histogram maintenance techniques

1. *Equal-width* histograms:
   The range of observed values is divided into equi-sized intervals $(\forall i, j: (b_i, b_{i+1}) = (b_j, b_{j+1}))$

2. *Equal-frequency* histograms:
   The range of observed values is divided into k intervals such that the counts in each interval are equal $(\forall i, j: (f_i = f_j))$

Maintaining Histograms

K-buckets Histograms (Gibbons et al., 1997)

- Incremental maintenance of histograms applicable for *Insert-Delete Models*

- Setting: Pre-defined number of intervals k and continuously occurring inserts and deletes as given in a sliding window approach

- Histogram maintenance based on two operations
  - Split & Merge Operation
  - Merge & Split Operation

Maintaining Histograms

K-buckets Histograms (Gibbons et al., 1997)

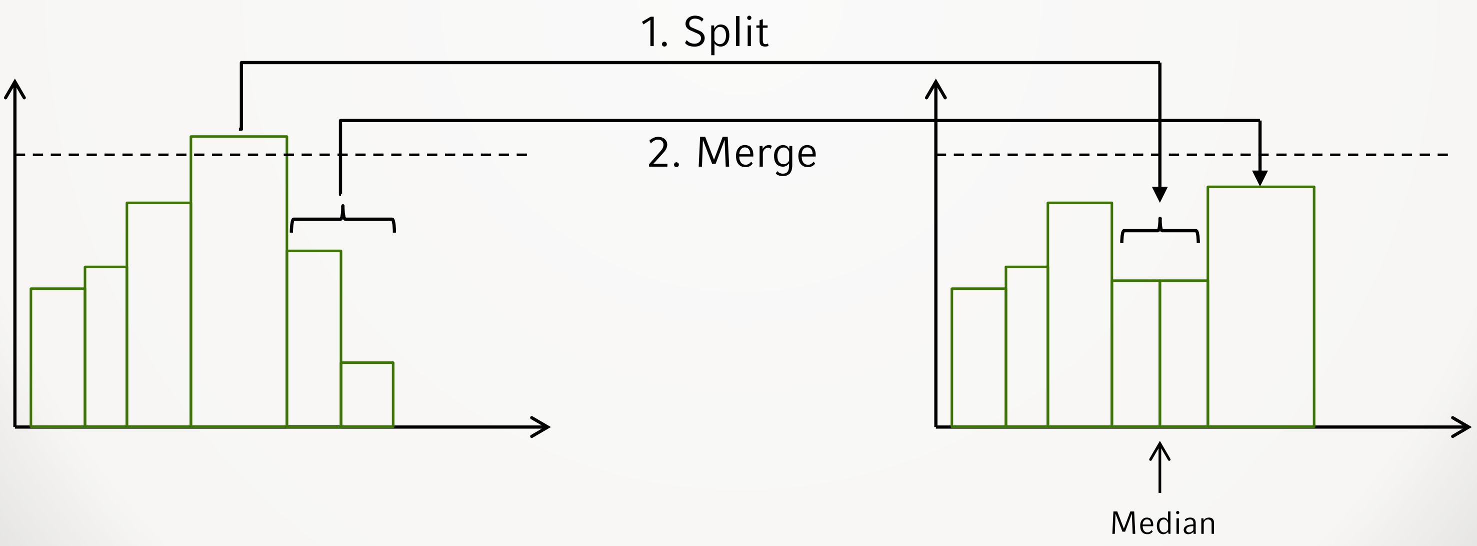
1. *Split & Merge Operation*:
   - Occurs with inserts
   - Triggered whenever the count in a bucket is greater than a given threshold
   - Split overflowed bucket into two and merge two consecutive buckets

Stream Applications and Algorithms

Maintaining Histograms

K-buckets Histograms (Gibbons et al., 1997)

1. *Split & Merge Operation*:



Maintaining Histograms

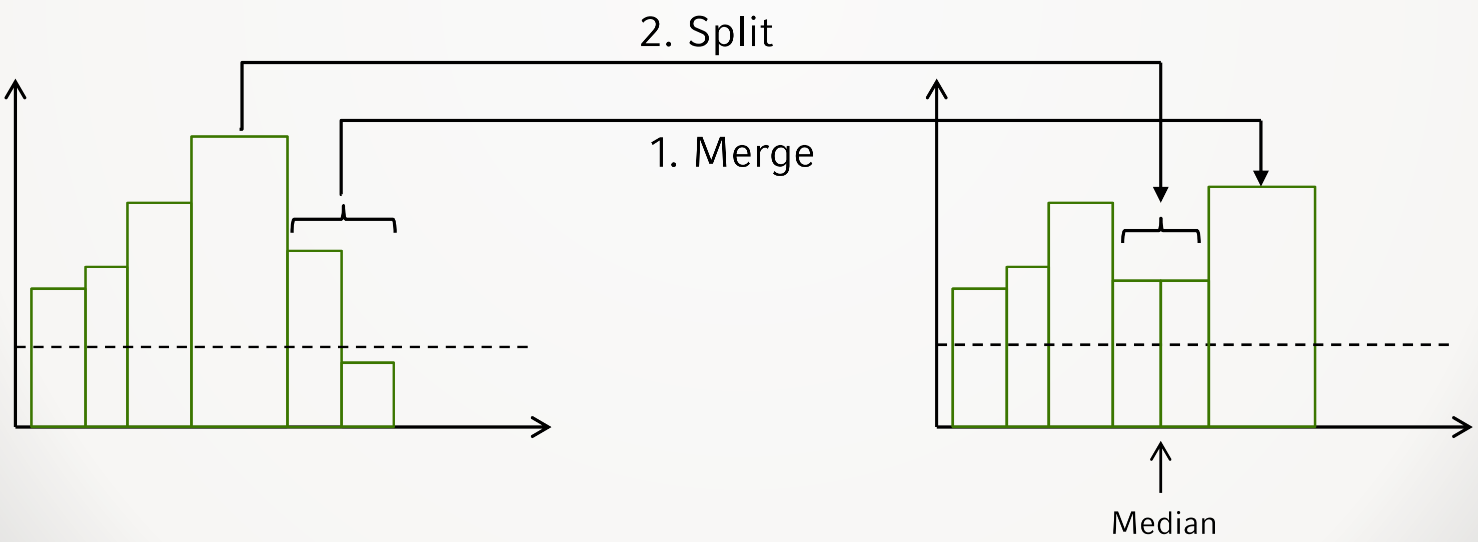
K-buckets Histograms (Gibbons et al., 1997)

1. *Merge & Split Operation*:
   - occurs with deletes
   - triggered whenever the count in a bucket is below a given threshold
   - merge "underflowed" bucket with a neighbor bucket and split the bucket with the highest count

Maintaining Histograms

K-buckets Histograms (Gibbons et al., 1997)

1. *Merge & Split Operation*:



Maintaining Histograms

Exponential Histograms (Datar et al., 2002)

- task: count the number of 1s among the last N readings of a bit stream
- trivial solution: maintain a window of N Bits as a ring buffer
  => remove last bit and decrement count if it was a 1
  => insert new bit and increment count if it is a 1

Can we reduce the memory requirement to log(N)?

- exponential histograms estimate the number of 1s
- memory consumption is log(N)
- compresses older readings stronger than fresh ones

Maintaining Histograms

Exponential Histograms (Datar et al., 2002)

- varying bucket sizes and interval sizes
- each bucket consists of $size$ and $timestamp$
- uses two additional variables, i.e. $LAST$ and $TOTAL$, to estimate the number of elements in the sliding window

Maintaining Histograms

Exponential Histograms (Datar et al., 2002)

Algorithm Exponential Histogram Maintenance
Input: data stream S, window size N, error param. ϵ
begin
  $TOTAL \coloneqq 0$
  $LAST \coloneqq 0$
  while S **do**
    $x_i \coloneqq S.next$
    if $x_i == 1$ **do**
      create new bucket b_i with timestamp t_i
      $TOTAL += 1$
      while $t_l \leq t_i - N$ **do**
        $TOTAL -= b_l.size$
        drop the oldest bucket b_l
        $b_l \coloneqq b_{l-1}$
        $LAST \coloneqq b_l.size$
      while *exist* $|1/\epsilon|/2 + 2$ *buckets of the same size* **do**
        merge the two oldest buckets of the same size with the largest timestamp of both buckets
        if *last bucket was merged* **do**
          $LAST \coloneqq$ size of the new created last bucket
end

Algorithm Exponential Histogram Count Estimation
Input: current Exponential Histogram EH
Output: estimate number of 1's within $EH.N$
begin
  return $\mathrm{EH.TOTAL} - \mathrm{EH.LAST}/2$
end

Maintaining Histograms

S = (1,0,1,0,1,1,1,0,0,0,1,1)

N = 8 (2^3=> 3 Buckets), $\epsilon = \frac{1}{2}$

Timest. t_i	Buckets b_i	Element x_i	TOTAL	LAST	$\# \, buckets$ $of \; same \; size = \tau$?
1	1_1	1	1	0	no
2	1_1	0	1	0	no
3	1_1, 1_3	1	2	0	no
4	1_1, 1_3	0	2	0	no
5	1_1, 1_3, 1_5 $2_3, 1_5$	1	3	2	yes
6	$2_3, 1_5$, 1_6	1	4	2	no
7	$2_3, 1_5$, 1_6, 1_7 → $2_3, 2_6$, 1_7	1	5	2	yes
8	…	…	…	…	…

Change Detection

General Assumptions:

- for static datasets:
  - data generated by a fixed process
  - data is a sample of a fixed distribution

- for data streams:
  - additional temporal dimension
  - underlying process can change over time

→ challenge: detection and quantification of changes

Change Detection

Impact of changes on data processing algorithms:

- Data Mining:
  data that arrived before a change can bias the model due to characteristics that no longer hold after the change

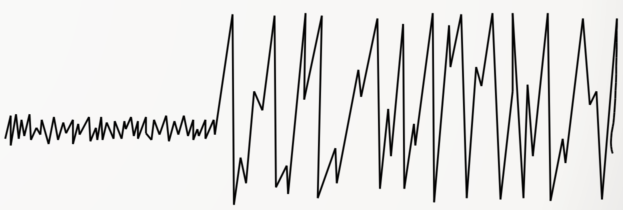
- Query Processing:
  query answers for time intervals with stable underlying data distributions might be more meaningful

Change Detection

The nature of changes

- Concept Drifts:
  gradual change in target concept

- Concept Shifts:
  abrupt change in target concept



Change Detection

Two general approaches

- Monitoring the evolution of performance indicators (Klinkenberg et al., 1998), e.g.
  - accuracy of the current classifier
  - attribute value distribution
  - monitoring *top* attributes (according to any ranking)

- monitoring distribution on two different time-windows

Change Detection

CUSUM Algorithm (Page, 1954)

- monitors the **cu**mulative **sum** of instances of a random variable

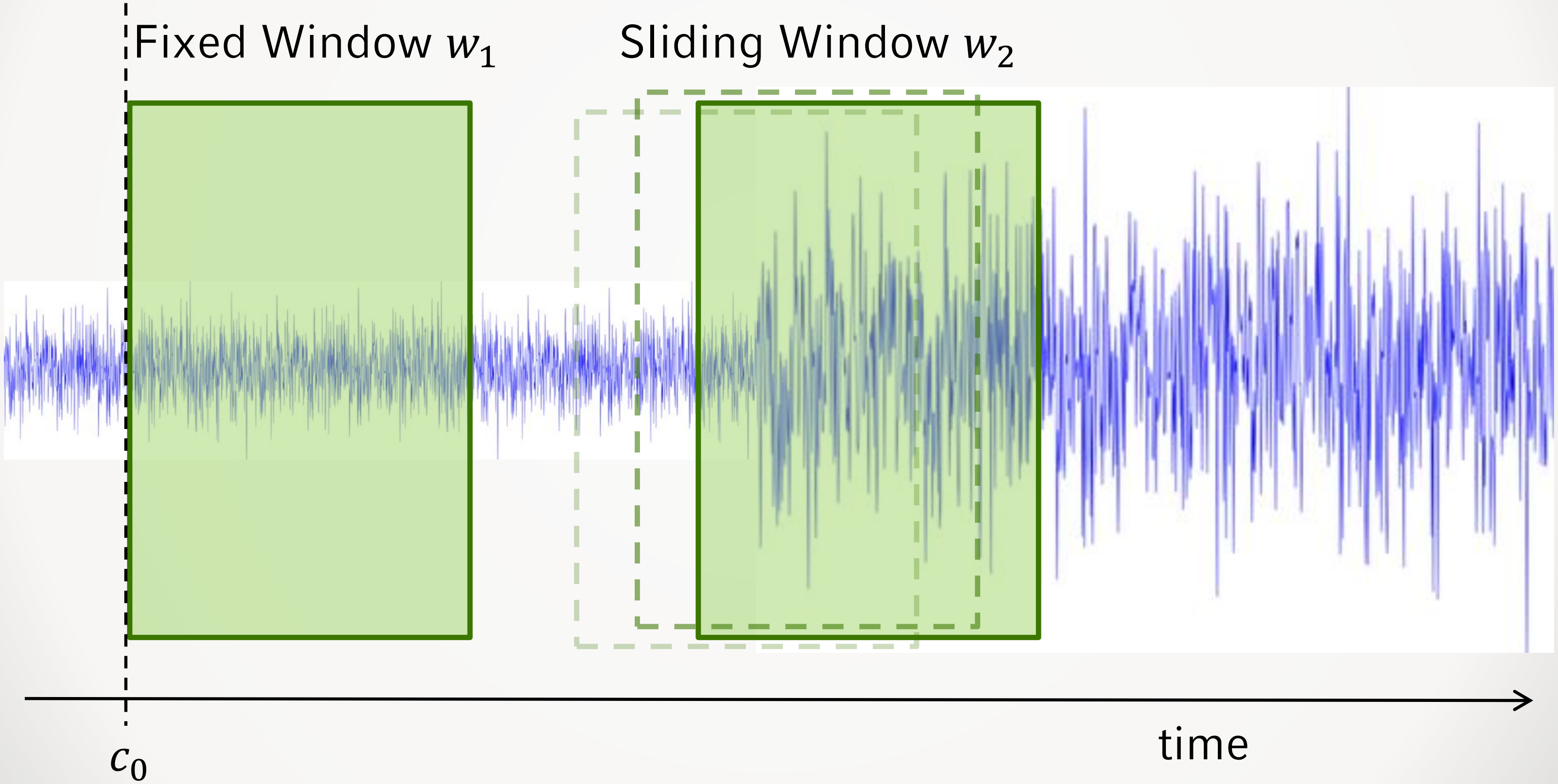
- detects a change if the cumulative difference between observation and likelihood gets larger than threshold α

Algorithm CUSUM
Input: data stream S, threshold param. α
begin
  $G_0 \coloneqq 0$
  while S **do**
  $x_t \coloneqq$ next instance of S
  compute likelihood ω_t
  $G_t \coloneqq \max(0, G_{t-1} - \omega_t + x_t)$
  if $G_t > \alpha$ **then**
    report change at time t
    $G_t \coloneqq 0$
end

- ω_t commonly represents the likelihood function

Change Detection

Two Windows Approach (Kifer et al., 2004)



Change Detection

Two Windows Approach (Kifer et al., 2004)

Algorithm Two Windows Approach
Input: data stream S, window sizes m_1 and m_2, distance func. $d: D \times D \to R$, threshold param. α
begin
  $c_0 \coloneqq 0$
  $W_1 \coloneqq$ first m_1 points from time c_0
  $W_2 \coloneqq$ most recent m_2 points from S
  while S **do**
    slide W_2 by 1 point
    if $d(W_1, W_2) > \alpha$ **then**
      $c_0 \coloneqq$ current time
      report change at time c_0
      $W_1 \coloneqq$ first m_1 points from time c_0
      $W_2 \coloneqq$ most recent m_2 points from S
end

d measures the distance between two probability distributions

Frequent Itemset Mining

- Let $A = \{a_1, a_2, \dots, a_n\}$ be a set of *items* (e.g. products)
- Any subset $I \subseteq A$ is called an *itemset*
- Let $T = (t_1, t_2, \dots, t_m)$ be a set of *transactions* with t_i being a pair $\langle TID_i, I_i \rangle$ where $I_i \subseteq A$ is a set of items (e.g. the set of products bought by a customer within a certain period in time)
- The *support* σ_{min} of an itemset $I \subseteq A$ is the number/fraction of transactions $t_i \in T$ that contain I

Frequent Itemset Mining

Example:
Given the set of items $A = \{a, b, c, d, e\}$, the set of transactions T, and a relative support $\sigma_{min} = 0.3$, determine the set of frequent item sets that is $\{I \subseteq A | \sigma_T(I) \geq \sigma_{min}\}$.

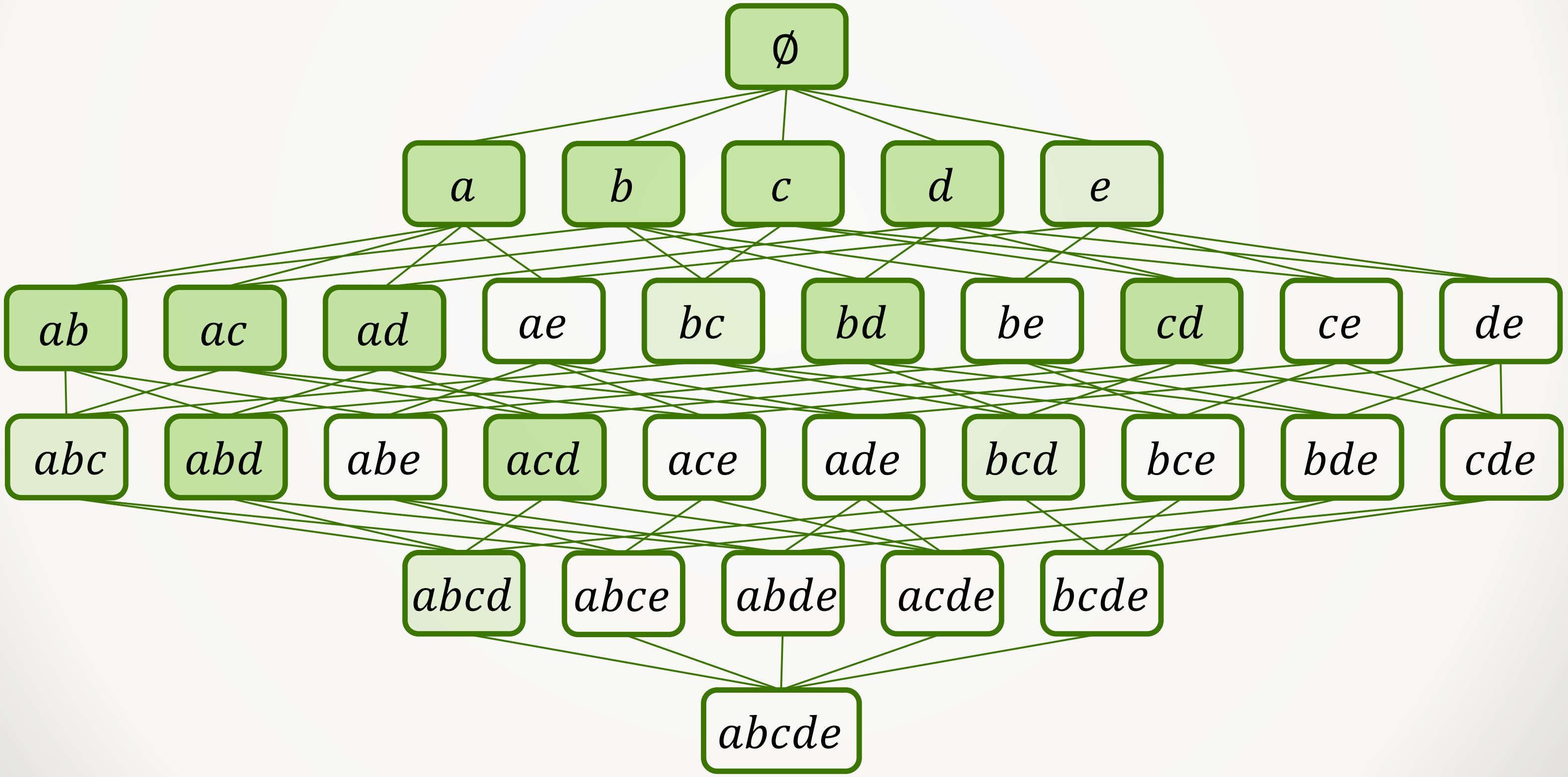
T:

TID_i	I_i
1	$\{a, b, c, d\}$
2	$\{b, d, e\}$
3	$\{a, b, d\}$
4	$\{a, b, c, d, e\}$
5	$\{a, c\}$
6	$\{c, d\}$
7	$\{a, c, d\}$

0 items	1 item	2 items	3 items
$\emptyset: 7$	$\{a\}: 5$	$\{a, b\}: 3$	$\{a, c, d\}: 3$
	$\{b\}: 5$	$\{a, c\}: 4$	$\{a, b, d\}: 3$
	$\{c\}: 5$	$\{a, d\}: 4$	
	$\{d\}: 6$	$\{b, d\}: 4$	
		$\{c, d\}: 4$	

Frequent Itemset Mining

search space



Frequent Itemset Mining

LossyCounting Algorithm (Manku et al., 2002)

- One-pass algorithm for computing frequency counts that exceed a user-specified threshold

- Approximate error but guaranteed to be below a user-specified boundary

→ Two parameters:
  - Support threshold $s \in [0,1]$
  - Error threshold $\epsilon \in [0,1]$
  - $\epsilon \ll s$

Frequent Itemset Mining

LossyCounting Algorithm (Manku et al., 2002)

- Setup:
  - Stream S is divided into buckets of width $\omega = \left\lceil \frac{1}{\epsilon} \right\rceil$
  - The current bucket id $b_{curr} = \left\lceil \frac{N}{\omega} \right\rceil$
  - For element e, the true frequency seen so far is f_e
  - The data structure D is a set of entries of the form (e, f, Δ)
    - e is the element
    - f is the frequency seen since e is in D
    - Δ is the maximum possible error, resp. the estimated frequency of e in buckets $b = 1$ to b_{curr}-1

Frequent Itemset Mining

LossyCounting Algorithm (Manku et al., 2002)

Algorithm LossyCounting
Input: data stream S, error threshold ϵ
begin
  $D = \emptyset$, $N = 0$, $\omega = \left\lceil \frac{1}{\epsilon} \right\rceil$
  while S **do**
    $e_i :=$ next object from S
    $N \mathrel{+}= 1$
    $b_{curr} = \left\lceil \frac{N}{\omega} \right\rceil$
    if $e_i \in D$ **then**
      increment e_i's frequency by 1
    else
      $D.add\big((e_i, 1, b_{curr} - 1)\big)$
    whenever $N \equiv 0 \mod \omega$ **do**
      foreach entry (e, f, Δ) in D **do**
        if $f + \Delta \leq b_{curr}$ **then**
          delete (e, f, Δ)
end

Algorithm LossyCounting – User request
Input: lookup table D, support threshold s
begin
  $S = \emptyset$
  foreach entry (e, f, Δ) in D **do**
    if $f \geq (s - \epsilon)N$ **then**
      add (e, f, Δ) to S
  return S
end

f is the exact frequency count of e since the entry was inserted into D

Δ is the maximum number of times e could have occurred in the first $b_{curr} - 1$ buckets

Clustering from Data Streams

Clustering is the process of grouping objects into different groups, such that the similarity of data in each subset is high, and between different subsets is low.

Clustering from data streams aims at maintaining a continuously consistent good clustering of the sequence observed so far, using a small amount of memory and time.

Clustering from Data Streams

General approaches to clustering

- *Partitioning*: Fixed number of clusters, new object is assigned to closest cluster center (k-means/k-medoid)
- *Density-based*: Take connectivity and density functions into account (DBSCAN)
- *Hierarchical*: Find a tree-like structure representing the hierarchy of the cluster model (Single Link/Complete Link)
- *Grid-based*: Partition the space into grid cells (STING)
- *Model-based*: Take a model and find the best fit clustering (COBWEB)

Clustering from Data Streams

Requirements for stream clustering algorithms

- Compactness of representation
- Fast, incremental processing (one-pass)
- Tracking cluster changes
  (as clusters might (dis-)appear over time)
- Clear and fast identification of outliers

Clustering from Data Streams

LEADER algorithm (Spath, 1980)

- Simplest form of partitioning based clustering applicable to data streams
- Depends on the order of incoming objects
- Depends on a good choice of the threshold parameter δ

```
Algorithm LEADER
Input: data stream S, threshold param. δ
begin
  while S do
    x_i := next object from S
    find closest cluster c_clos to x_i
    if d(c_clos, x_i) < δ then
      assign x_i to c_clos
    else
      create new cluster with x_i
end
```

Clustering from Data Streams

Stream K-means (O'Callaghan et al., 2002)

- Partition data stream S into chunks $X_1, \dots, X_n, \dots$ so that each chunk fits in memory
- Apply k-means for each chunk X_i and retrieve k cluster centers each weighted with the number of points it compresses
- Apply k-means on the cluster centers to get an overall k-means clustering when demanded

Clustering from Data Streams

Microcluster-based Clustering

- Common approach to capture temporal information for being able to deal with cluster evolution

- A *microcluster* (or *cluster feature CF*) is a triple (N, LS, SS) that stores the sufficient information of a set of points
  - N is the number of points
  - LS is the linear sum of the N points, i.e. $\sum_{i=1}^{N} \overrightarrow{x_i}$
  - SS is the square sum of the N points, i.e. $\sum_{i=1}^{N} \overrightarrow{x_i}^2$

Clustering from Data Streams

Microcluster-based Clustering

- The properties of cluster features are:
  - Incrementality:
    $\mathrm{N_i} = \mathrm{N_i} + 1, \qquad LS_i = LS_i + \vec{x}, \qquad SS_i = SS_i + \vec{x}^2$
  - Additivity:
    $\mathrm{N_k} = \mathrm{N_i} + \mathrm{N_j}, \qquad LS_k = LS_i + LS_j, \qquad SS_k = SS_i + SS_j$
  - Centroid: $\overrightarrow{X_c} = \frac{LS_i}{N}$
  - Radius: $r = \sqrt{\frac{SS_i}{N_i} - \left(\frac{LS_i}{N_i}\right)^2}$

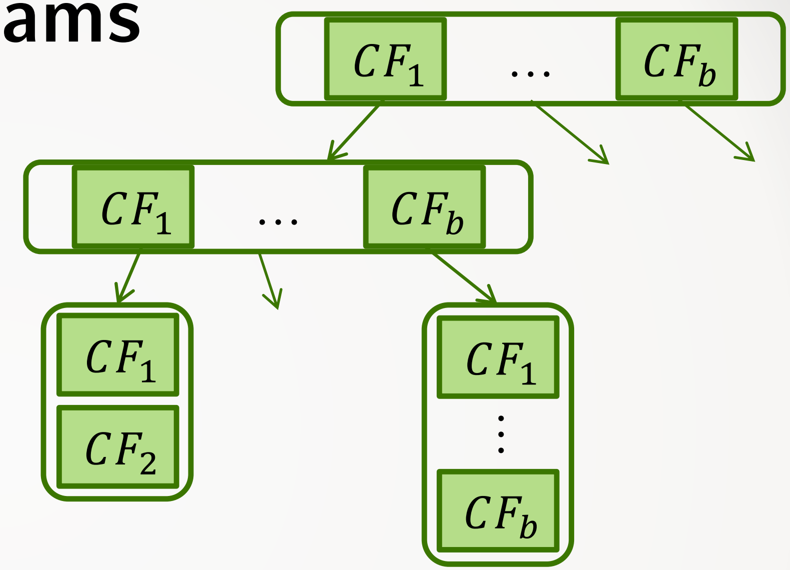
Clustering from Data Streams

BIRCH (Zhang et al., 1996)

- Usage of Microclusters within CF-Tree
  - B^+-Tree like structure
  - Two user specified parameters:
    - Branching factor B
    - Maximum diameter (or radius) T of a CF
  - Each non-leaf node contains at most B entries of the form $[CF_i, child_i]$ where
    - CF_i is the CF representing the subcluster that child forms
    - $child_i$ is a pointer to the i-th child node
  - Each leaf node contains entries of the form $[CF_i, prev, next]$

Stream Applications and Algorithms

Clustering from Data Streams



BIRCH (Zhang et al., 1996)

- Inserts into CF-Tree
  - At each non-leaf node, the new object follows the *closest-CF* path
  - At leaf node level, the *closest-CF* tries to *absorb* the object (which depends on diameter threshold T and the page size)
    - If possible: update *closest-CF*
    - If not possible: make a new CF entry in the leaf node (split the parent node if there is no space)

Clustering from Data Streams

BIRCH (Zhang et al., 1996)

Two step algorithm:

1. Online component:
   - Microclusters are kept locally
   - Maintenance of the hierarchical structure
   - Optional: Condense by building smaller CF-Tree (requires scan over leaf entries)

2. Offline component:
   - Apply global clustering to all leaf entries
   - Optional: Cluster refinement to the cost of additional passes (use centroids retrieved by global clustering and re-assign data points)

CluStream [AggEtAl03]

Assume that the data stream consists of a set of multi-dimensional records $X_1,\ldots X_n,\ldots$, arriving at $T_1,\ldots,T_n,\ldots$: $X_i = (x_i^1,\ldots,x_i^d)$

- The micro-cluster summary for a set of d-dimensional points (X_1, X_2, …, X_n) arriving at time points T_1, T_2, …, T_n is defined as:
  - CFT = ($CF2^x$, $CF1^x$, $CF2^t$, $CF1^t$, n)

$$CF2^x = \sum_{i=1}^{N} X_i^2 \qquad CF1^x = \sum_{i=1}^{N} X_i \qquad CF2^t = \sum_{i=1}^{N} T_i^2 \qquad CF1^t = \sum_{i=1}^{N} T_i$$

- Easy calculation of basic measures to characterize a cluster:

•

- Center: $\frac{CF1^x}{n}$
- Radius: $\sqrt{\frac{CF2^x}{n} - \left(\frac{CF1^x}{n}\right)^2}$

- Important properties of micro-clusters:
  - Incrementality: CFT(C_1 U p) = CFT(C_1) + p
  - Additivity: CFT(C_1 U C_2) = CFT(C_1) + CFT(C_2)
  - Subtractivity: CFT(C_1 - C_2) = CFT(C_1) - CFT(C_2), $C_1 \supseteq C_2$

CluStream: overview

- A fixed number of *q* micro-clusters is maintained over time
- **Initialize**: apply *q*-Means over *initPoints*, built a summary for each cluster
- Online micro-cluster maintenance as a new point *p* arrives from the stream
  - Find the closest micro-cluster *clu* for the new point *p*
    - If *p* is within the max-boundary of *clu*, *p* is <u>absorbed</u> by *clu*
    - o.w., a <u>new cluster</u> is created with *p*
  - The number of micro-clusters should not exceed q
    - <u>Delete</u> most obsolete micro-cluster or <u>merge</u> the two closest ones
- Periodic storage of micro-clusters snapshots into disk
  - At different levels of granularity depending upon their recency
- Offline macro-clustering
  - Input: A user defined time horizon *h* and number of macro-clusters *k* to be detected
  - Locate the valid micro-clusters during *h*
  - Apply k-Means upon these micro-clusters → k macro-clusters

CluStream: Initialization step

- Initialization
  - Done using an offline process in the beginning
  - Wait for the first *InitNumber* points to arrive
  - Apply a standard *k*-Means algorithm to create *q* clusters
    - For each discovered cluster, assign it a unique ID and create its micro-cluster summary.

- Comments on the choice of *q*
  - much larger than the natural number of clusters
  - much smaller than the total number of points arrived

CluStream: Online step

- A fixed number of *q* micro-clusters is maintained over time
- Whenever a new point *p* arrives from the stream
  - Compute distance between *p* and each of the *q* maintained micro-cluster centroids
  - *clu* ← the closest micro-cluster to *p*
  - Find the max boundary of *clu*
    - It is defined as a factor of *t* of *clu* radius
  - If p falls within the maximum boundary of *clu*
    - *p* is absorbed by *clu*
    - Update clu statistics (incremental property)
  - Else, create a new micro-cluster with p, assign it a new ID, initialize its statistics
    - To keep the total number of micro-clusters fixed (i.e., *q*):
      - Delete the most obsolete micro-cluster or
        - If its safe based on its time statistics
      - Merge the two closest ones (Additivity property)
        - When two micro-clusters are merged, a list of ids is created. This way, we can identify the component micro-clusters that comprise a micro-cluster.

CluStream: storing micro-cluster storage

- all micro-clusters are saved to disc for a given time frame (snapshot)
- not all snapshots are kept. goal: provide a high granularity for recent snapshots and maintain a rough picture of older snapshots

$\Rightarrow$ Pyramidal Storage

idea: Maintain the last a^i+1 snapshots for multiple levels of storage granularity. => larger level cover a longer history with less granularity

- For each level i, we store a snapshot if the current timestamp $t \bmod a^i = 0$, *except* $t \bmod a^{i+1} = 0$ *holds as well (no redundancy)*
- At most a^b+1 snapshots are stored at each order; if a new snapshot arrives the oldest one is deleted.

level	clock times
0	~~60~~ 59 ~~58~~ 57 ~~56~~
1	~~60~~ 58 ~~56~~ 54 ~~52~~
2	60 56 52 ~~48~~ 44
3	~~48~~ 40 ~~32~~ 24 ~~16~~
4	48 ~~32~~ 16
5	32

Snapshots stored at t = 60, a=2, b=2

- #levels: $\log_a(t)$
- #stored snapshots: $(a^b+1)\log_a(t)$

CluStream: Offline step

- The offline step is applied on demand
- User input: time horizon *h*, # macro-clusters k to be detected
- Step 1: Find the active micro-clusters during *h*:
  - We exploit the subtractivity property to find the active micro-clusters during *h*:
    - Suppose current time is t_c. Let S(t_c) be the set of micro-clusters at t_c.
    - Find the stored snapshot which occurs just before time t_c-*h*. We can always find such a snapshot *h'*. Let S(t_c–h') be the set of micro-clusters.
    - For each micro-cluster in the current set S(t_c), we find the list of its component micro-cluster ids. For each of the list of ids, find the corresponding micro-clusters in S(t_c–h').
    - Subtract the CF vectors for the corresponding micro-clusters in S(t_c–h')
    - This ensures that the micro-clusters created before the user-specified horizon do not dominate the result of clustering process
- Step 2: Apply k-Means over the active micro-clusters in *h* to derive the *k* macro-clusters
  - Initialization: centers are not picked up randomly, rather sampled with probability proportional to the number of points in a given micro-cluster
  - Distance is the centroid distance
  - New centers are defined as the weighted centroids of the micro-clusters in that partition

CluStream: discussion

- \+ CluStream clusters large evolving data streams
- \+ Views the stream as a changing process over time, rather than clustering the whole stream at a time
- \+ Can characterize clusters over different time horizons in changing environment
- \+ Provides flexibility to an analyst in a real-time and changing environment

- – Fixed number of micro-clusters maintained over time
- – Sensitive to outliers/ noise

The (batch) classification process

Model construction

Training data

NAME	RANK	YEARS	TENURED
Mike	Assistant Prof	3	no
Mary	Assistant Prof	7	yes
Bill	Professor	2	yes
Jim	Associate Prof	7	yes
Dave	Assistant Prof	6	no
Anne	Associate Prof	3	no

Predictive attributes

Class attribute

Classification Algorithm

Classifier (Model)

IF rank = 'professor' OR years > 6 THEN tenured = 'yes'

IF (rank!='professor') AND (years < 6) THEN tenured = 'no'

Prediction

Unseen data

NAME	RANK	YEARS	TENURED		
Jeff	Professor	4	?	Tenured?	Yes
Patrick	Assistant Professor	8	?	Tenured?	?
Maria	Assistant Professor	2	?	Tenured?	?

Stream vs batch classification 1/2

- So far, classification as a batch/ static task
  - The whole training set is given as input to the algorithm for the generation of the classification model.
  - The classification model is static (does not change)
  - When the performance of the model drops, a new model is generated from scratch over a new training set.

- But, in a dynamic environment data change continuously
  - Batch model re-generation is not appropriate/sufficient anymore

Stream vs batch classification 2/2

Need for new classification algorithms that

- have the ability to *incorporate new data*
- deal with non-stationary data generation processes (concept drift/shift)
  - ability to *remove obsolete data*
- subject to:
  - resource constraints (processing time, memory)
  - single scan of the data (one look, no random access)

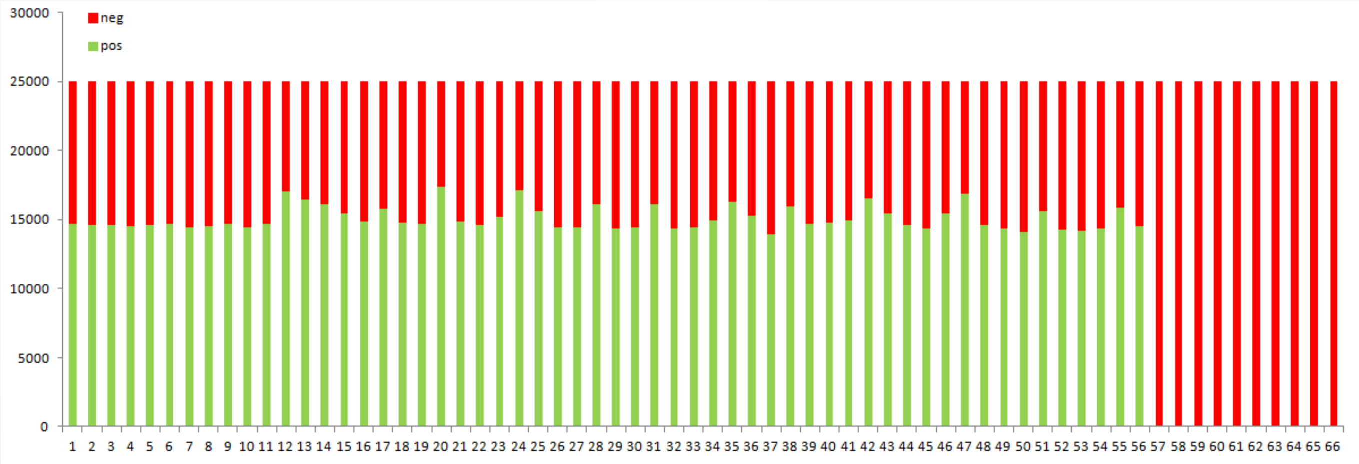
Non-stationary data distribution → Concept drift

- In dynamically changing and non-stationary environments, the data distribution might change over time yielding the phenomenon of concept drift
- Different forms of change:
  - The input data characteristics might change over time
  - The relation between the input data and the target variable might change over time
- Concept drift between t_0 and t_1 can be defined as $\exists X : p_{t_0}(X,y) \neq p_{t_1}(X,y)$
  - P(X,y): the joint distribution between X and y

- According to the Bayesian Decision Theory: $p(y|X) = \frac{p(y)p(X|y)}{p(X)}$
- So, changes in data can be characterized as changes in:
  - The prior probabilities of the classes p(y)
  - The class conditional probabilities p(X|y).
  - The posterior p(y|X) might change

Example: Evolving class priors

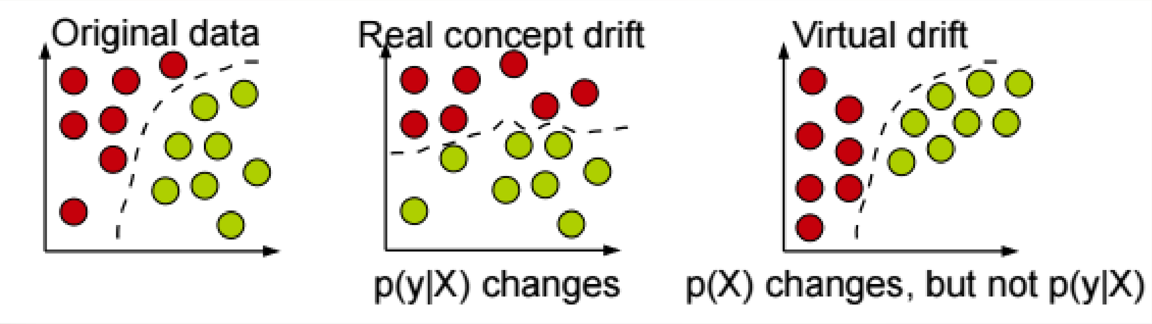
- E.g., evolving class distribution
  - The class distribution might change over time
  - Example: Twitter sentiment dataset
    - 1.600.000 instances split in 67 chunks of 25.000 tweets per chunk
    - Balanced dataset (800.000 positive, 800.000 negative tweets)
    - The distribution of the classes changes over time
    - Dataset online at: *https://sites.google.com/site/twittersentimenthelp/for-researchers*



Evolving class distribution [Sinelnikova12]

Real vs virtual drift

- Real concept drift
  - Refers to changes in p(y|X). Such changes can happen with or without change in p(X).
  - E.g., “I am not interested in tech posts anymore”
- Virtual concept drift
  - If the p(X) changes without affecting p(y|X)

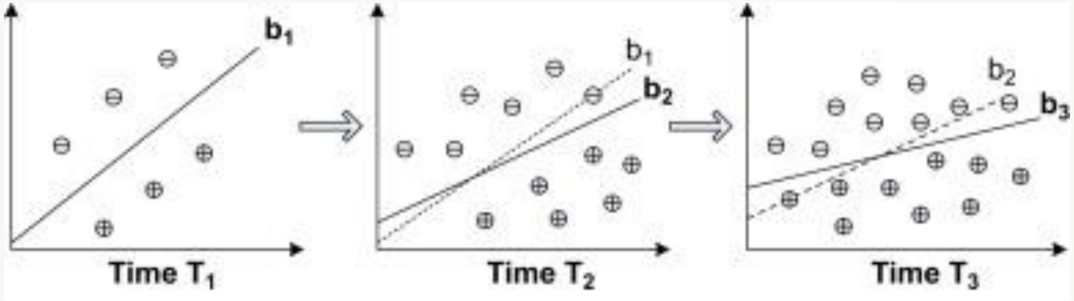


Source: [GamaETAl13]

  - Drifts (and shifts)
    - Drift more associated to gradual changes
    - Shift refers to abrupt changes

Model adaptation

- As data evolve over time, the classifier should be updated to “reflect” the evolving data
  - Update by incorporating new data
  - Update by forgetting obsolete data



The classification boundary gradually drifts from b_1 (at T_1) to b_2 (at T_2) and finally to b_3 (at T_3). (Source: A framework for application-driven classification of data streams, Zhang et al, Journal Neurocomputing 2012)

Data stream classifiers

- The batch classification problem:
  - Given a finite training set D={(x,y)} , where y={y_1, y_2, …, y_k}, |D|=n, find a function y=f(x) that can predict the y value for an unseen instance x

- The data stream classification problem:
  - Given an infinite sequence of pairs of the form (x,y) where y={y_1, y_2, …, y_k}, find a function y=f(x) that can predict the y value for an unseen instance x
    - the label y of x is not available during the prediction time
    - but it is available shortly after for model update

Supervised scenario

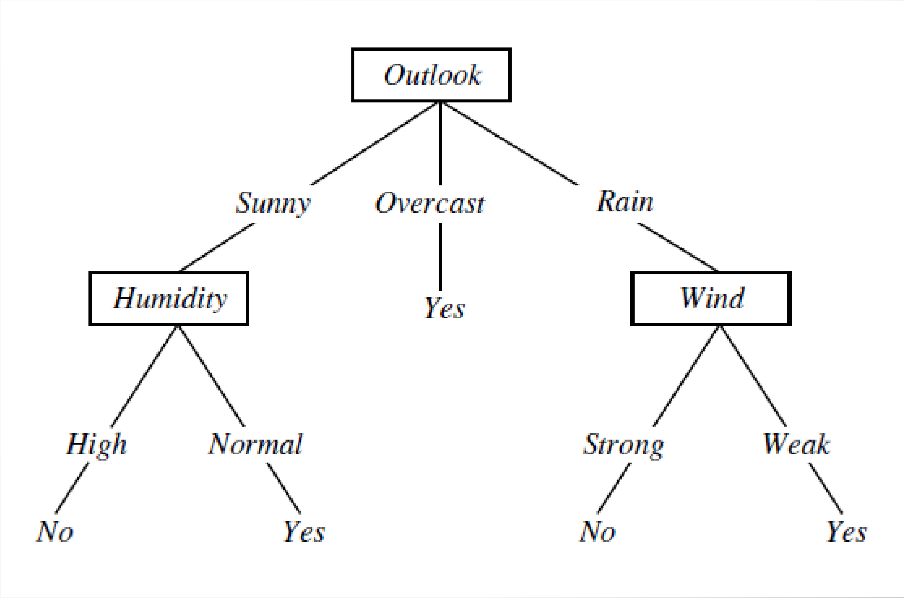
- Example applications:
  - Fraud detection in credit card transactions
  - Churn prediction in a telecommunication company
  - Sentiment classification in the Twitter stream
  - Topic classification in a news aggregation site, e.g. Google news
  - …

(Batch) Decision Trees (DTs)

- Training set: D = {(x,y)}
  - predictive attributes: $x=<x_1, x_2, \ldots, x_d>$
  - class attribute: $y=\{y_1, y_2, \ldots, y_k\}$
- Goal: find y=f(x)
- Decision tree model
  - nodes contain tests on the predictive attributes
  - leaves contain predictions on the class attribute

Training set

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No



(Batch) DTs: Selecting the splitting attribute

- Basic algorithm (ID3, Quinlan 1986)
  - Tree is constructed in a top-down recursive divide-and-conquer manner
  - At start, all the training examples are at the root node

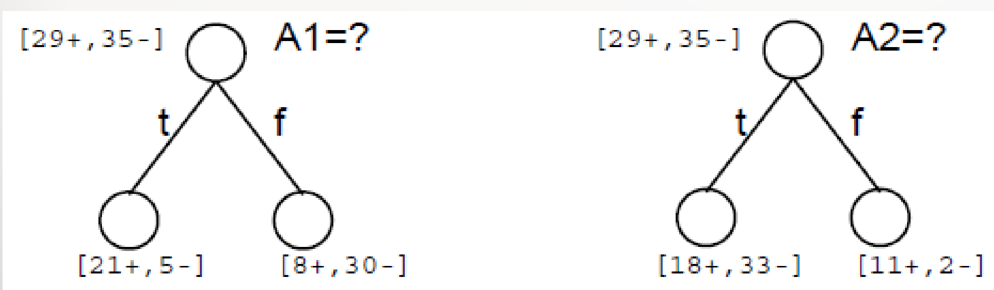
Main loop:

1. $A \leftarrow$ the "best" decision attribute for next *node*
2. Assign A as decision attribute for *node*
3. For each value of A, create new descendant of *node*
4. Sort training examples to leaf nodes
5. If training examples perfectly classified, Then STOP, Else iterate over new leaf nodes

  - But, which attribute is the best?

Attribute selection measures:

- Information gain
- Gain ratio
- Gini index



Goal: select the most "useful" attribute

- i.e., the one resulting in the purest partitioning

(Batch) DTs: Information gain

- Used in ID3
- It uses entropy, a measure of pureness of the data
- The information gain Gain(S,A) of an attribute A relative to a collection of examples S measures the gain reduction in S due to splitting on A:

$$Gain(S,A) = \boxed{Entropy(S)} - \boxed{\sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)}$$

Before splitting

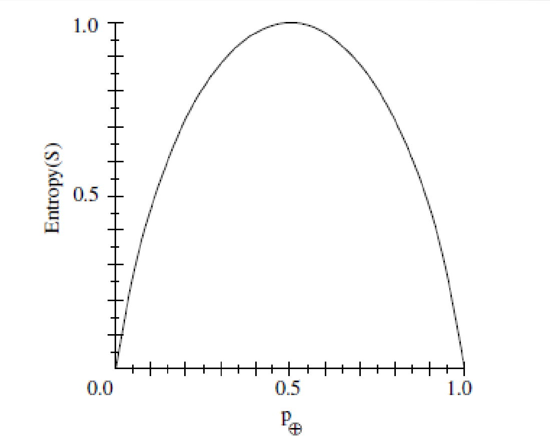
After splitting on A

- Gain measures the expected reduction in entropy due to splitting on A

- The attribute with the higher entropy reduction is chosen

(Batch) DTs: Entropy

- Let S be a collection of positive and negative examples for a binary classification problem, C={+, -}.
- p_+: the percentage of positive examples in S
- p_-: the percentage of negative examples in S
- Entropy measures the impurity of S:



$$Entropy(S) = -p_+ \log_2(p_+) - p_- \log_2(p_-)$$

- Examples :
  - Let S: [9+,5-] $Entropy(S) = -\frac{9}{14}\log_2(\frac{9}{14}) - \frac{5}{14}\log_2(\frac{5}{14}) = 0.940$
  - Let S: [7+,7-] $Entropy(S) = -\frac{7}{14}\log_2(\frac{7}{14}) - \frac{7}{14}\log_2(\frac{7}{14}) = 1$
  - Let S: [14+,0-] $Entropy(S) = -\frac{14}{14}\log_2(\frac{14}{14}) - \frac{0}{14}\log_2(\frac{0}{14}) = 0$

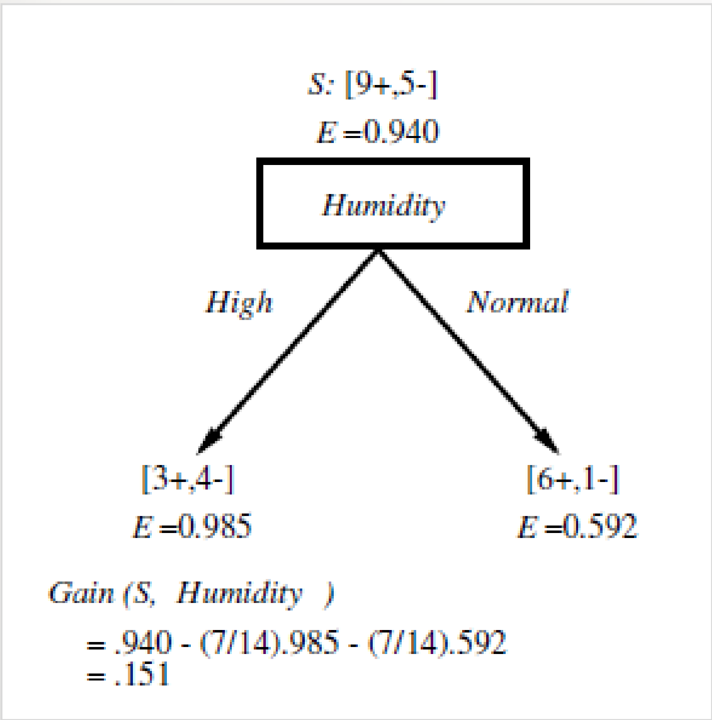
in the general case
(*k-classification problem)*

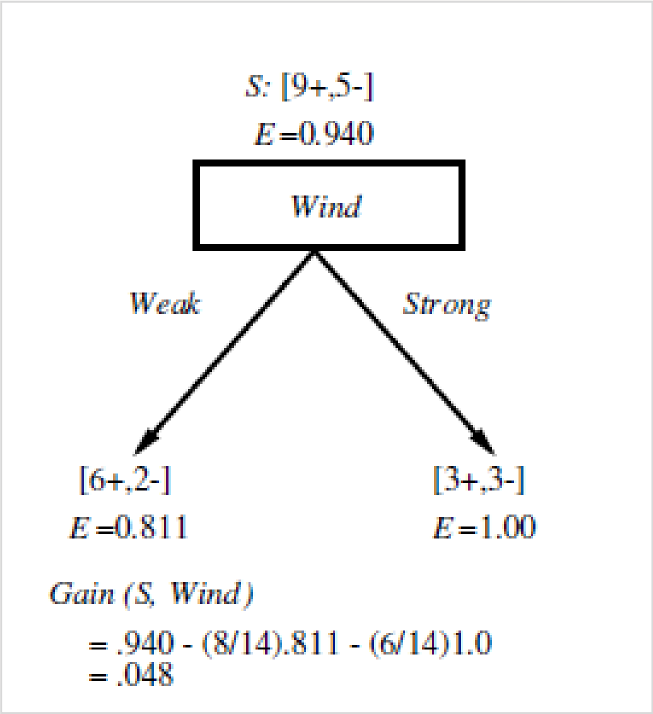
$$Entropy(S) = \sum_{i=1}^{k} - p_i \log_2(p_i)$$

- Entropy = 0, when all members belong to the same class
- Entropy = 1, when there is an equal number of positive and negative examples

(Batch) DTs: Information gain example

- Which attribute to choose next?





From batch to stream DT induction

- Thus far, in order to decide on which attribute to use for splitting in a node (essential operation for building a DT), we need to have all the training set instances resulting in this node.

- But, in a data stream environment
  - The stream is infinite
  - We can’t wait for ever in a node

- Can we make a valid decision based on some data?
  - Hoeffding Tree or Very Fast Decision Tree (VFDT) [DomingosHulten00]

Hoeffding Tree [DomingosHulten00]

- **Idea**: In order to pick the best split attribute for a node, it may be sufficient to consider only a small subset of the training examples that pass through that node.
  - No need to look at the whole dataset
  - (which is infinite in case of streams)

- **Problem**: How many instances are necessary?
  - Use the Hoeffding **bound**!

The Hoeffding bound

- Consider a real-valued random variable *r* whose range is *R*
  - e.g., for a probability the range is 1
  - for information gain the range is $\log_2(c)$, where c is the number of classes
- Suppose we have *n* independent observations of *r* and we compute its mean $\bar{r}$
- The Hoeffding bound states that with confidence *1-δ* the true mean of the variable, μ_r, is at least $\bar{r}$-ε, i.e., $P(\mu_r \geq \bar{r} - \varepsilon) = 1 - \delta$
- The ε is given by:

$$\varepsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}$$

- This bound holds true regardless of the distribution generating the values, and depends only on the range of values, number of observations and desired confidence.
  - A disadvantage of being so general is that it is more conservative than a distribution-dependent bound

Using the Hoeffding bound to select the best split at a node

- Let *G()* be the heuristic measure for choosing the split attribute at a node
- After seeing *n* instances at this node, let
  - X_a : be the attribute with the highest observed *G()*
  - X_b : be the attribute with the second-highest observed *G()*
- $\Delta\bar{G} = \bar{G}(X_a) - \bar{G}(X_b) \geq 0$ the difference between the 2 best attributes
- $\Delta\bar{G}$ is the random variable being estimated by the Hoeffding bound
- Given a desired δ, if $\Delta\bar{G} > \varepsilon$ after seeing n instances at the node
  - the Hoeffding bound guarantees that with probability $1\text{-}\delta$, $\Delta\bar{G} \geq \Delta\bar{G}\text{-}\varepsilon > 0$.
  - Therefore we can confidently choose X_a for splitting at this node
- Otherwise, i.e., if $\Delta\bar{G} < \varepsilon$, the sample size is not enough for a stable decision.
  - With R and δ fixed, the only variable left to change ε is n
  - We need to extend the sample by seeing more instances, until ε becomes smaller than $\Delta\bar{G}$

Hoeffding Tree algorithm

- **Input:** δ desired probability level.
- **Output:** $\mathcal{T}$ A decision Tree
- **Init:** $\mathcal{T} \leftarrow$ Empty Leaf (Root)
- While (TRUE)
  - Read next Example
  - Propagate Example through the Tree from the Root till a leaf
  - Update Sufficient Statistics at leaf
  - If *leaf(#examples)mod* N_{min}*=0*
    - Evaluate the merit of each attribute
    - Let A_1 the best attribute and A_2 the second best
    - Let $\epsilon = \sqrt{R^2 ln(1/\delta)/(2n)}$
    - If $G(A_1) - G(A_2) > \epsilon$
    - Install a splitting test based on A_1
    - Expand the tree with two descendant leaves

Those needed by the heuristic evaluation function G()

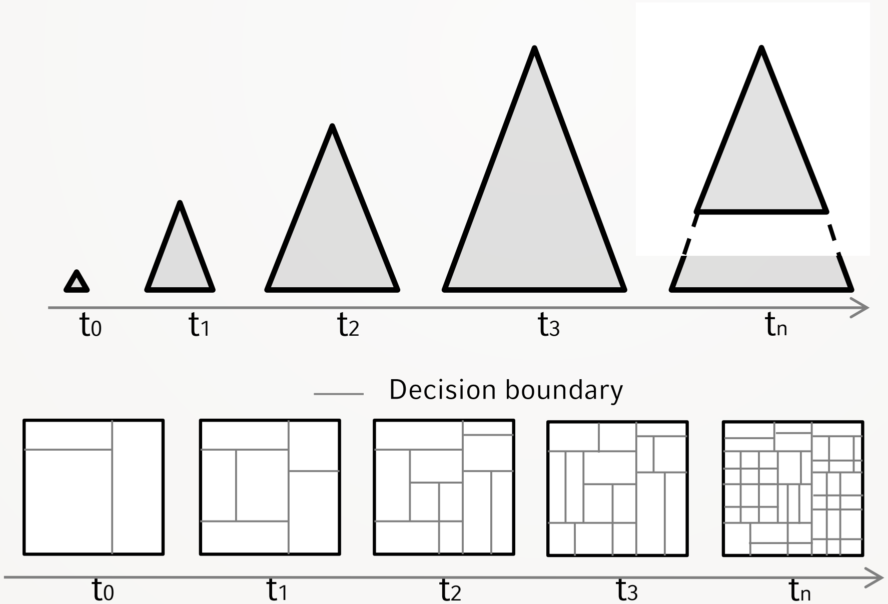
The evaluation of G() after each instance is very expensive.
→ Evaluate G() only after N_{min} instances have been observed since the last evaluation.

Hoeffding tree algorithm more details

- Breaking ties
  - When ≥2 attributes have very similar G's, potentially many examples will be required to decide between them with high confidence.
  - This is presumably wasteful, as it makes little difference which is chosen.
  - Break it by splitting on current best if $\Delta G < \varepsilon < \tau$, τ a user-specified threshold
- Grace period (MOA's term)
  - Recomputing G() after each instance is too expensive.
  - A user can specify # instances in a node that must be observed before attempting a new split

Hoeffding Tree overview

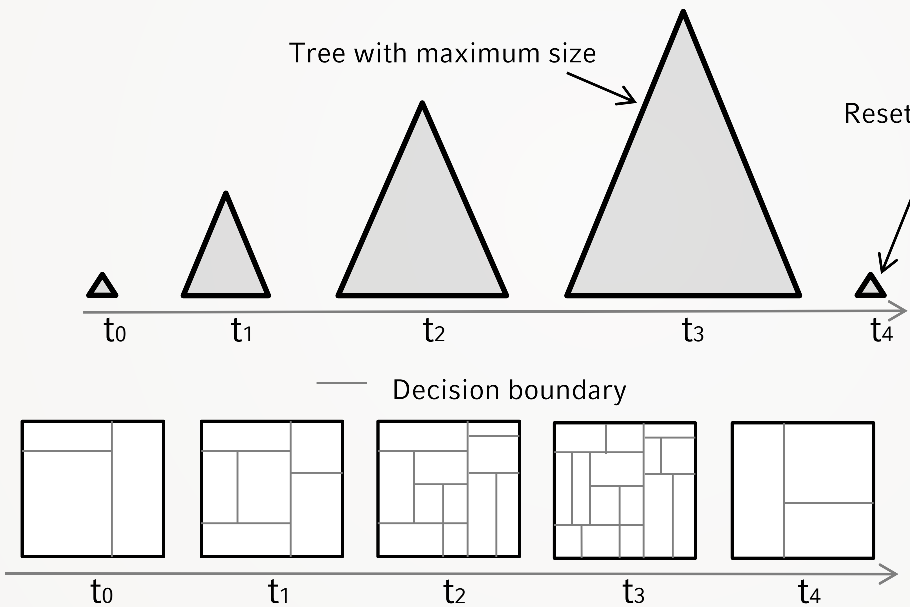
- The HT accommodates new instances from the stream
- But, doesn't delete anything (doesn't forget!)
- With time
  - The tree becomes more complex (overfitting is possible)
  - The historical data dominate its decisions (difficult to adapt to changes)



HT over time [Mahmud15]

Adaptive Size Hoeffding Tree (ASHT) [BifetEtAl09]

- Introduces a maximum size (#splitting nodes) bound
- When the limit is reached, the tree is reset
  - Test for the limit, after node's split

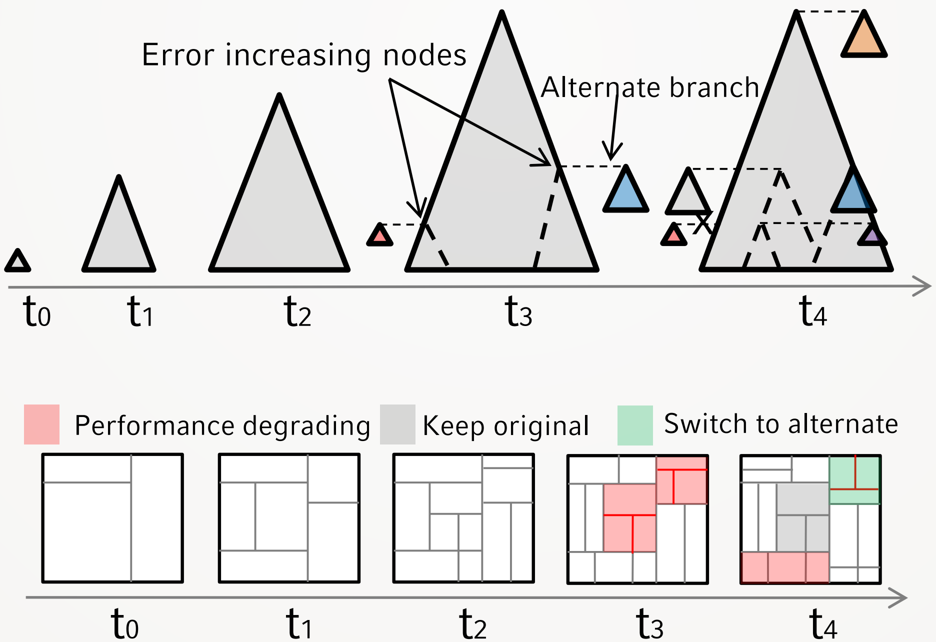


ASHT over time [Mahmud15]

- The tree forgets
  - but, due to the reset, it looses all information learned thus far

Concept-Adapting Hoeffding Tree [HultenEtAl01]

- Starts maintaining an alternate sub-tree when the performance of a node decays
- When the new sub-tree starts performing better, it replaces the original one
- If original sub-tree keeps performing better, the alternate sub-tree is deleted and the original one is kept

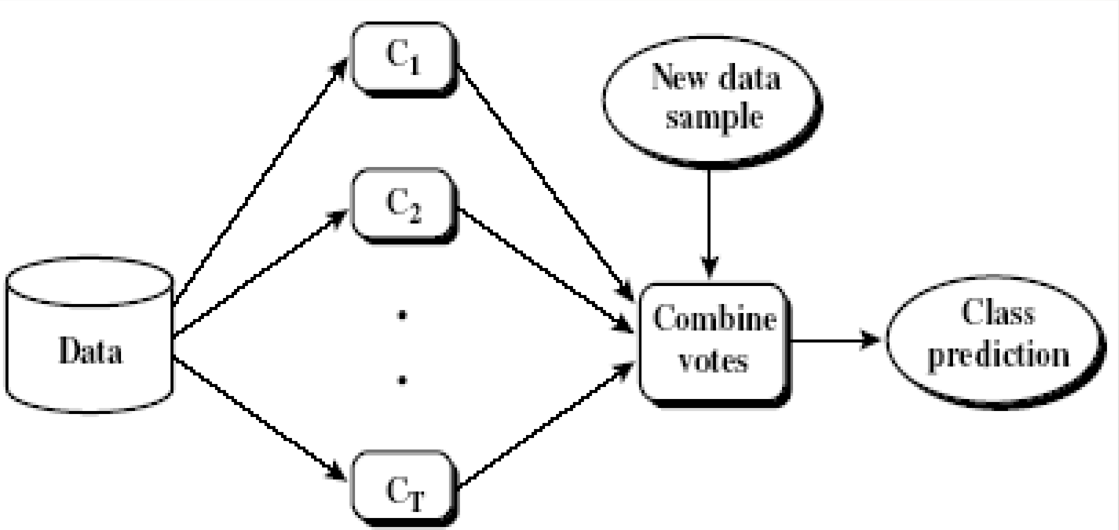


AdaHT over time [Mahmud15]

Ensemble of classifiers

Idea:

- Instead of a single model, use a combination of models to increase accuracy
- Combine a series of *T* learned models, $M_1, M_2, \ldots, M_T$, with the aim of creating an improved model M^*
- To predict the class of previously unseen records, aggregate the predictions of the ensemble

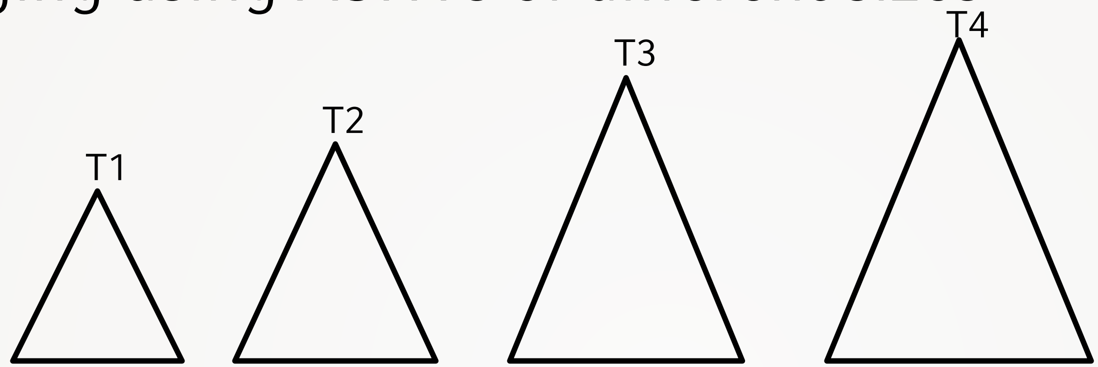


Many methods

- Bagging
  - Generate training samples by sampling with replacement (bootstrap)
  - Learn one model at each sample
- Boosting
  - At each round, increase the weights of misclassified examples
- Stacking
  - Apply multiple base learners
  - Meta learner input = base learner predictions

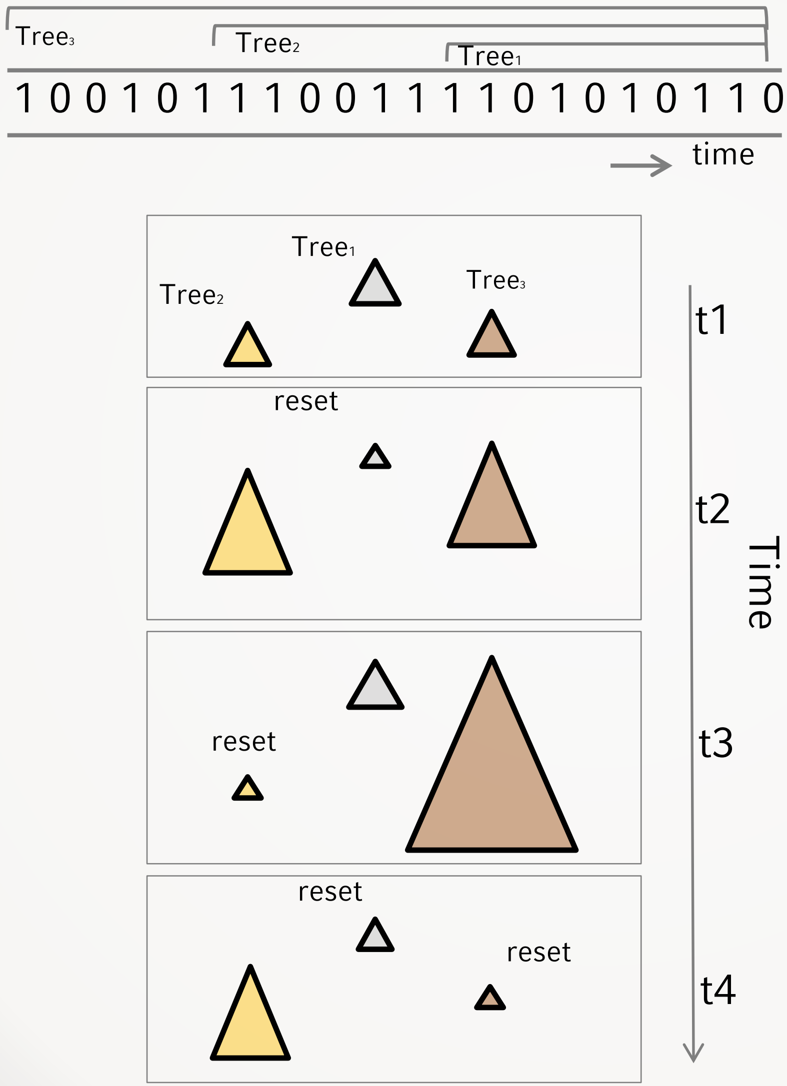
Ensemble of Adaptive Size Hoeffding Trees (ASHT) [BifetEtAl09] 1/2

- Bagging using ASHTs of different sizes



  - Smaller trees adapt more quickly to changes
  - Larger trees perform better during periods with no or little change
  - The max allowed size for the n^{th} ASHT tree is twice the max allowed size for the $(n-1)^{th}$ tree.
  - Each tree has a weight proportional to the inverse of the square of its error
  - The goal is to increase bagging performance by tree diversity

Ensemble of Adaptive Size Hoeffding Trees (ASHT) [BifetEtAl09] 1/2



Hoeffding Tree family overview

- All HT, AdaHT, ASHT accommodate new instances from the stream
- HT does not forget
- ASHT forgets by resetting the tree once its size reaches its limit
- AdaHT forgets my replacing sub-trees with new ones
- Bagging ASHT uses varying size trees that respond differently to change

Summary: Stream Classification

- Extending traditional classification methods for data streams implies that
  - They should accommodate new instances
  - They should forget obsolete instances
- Typically, all methods incorporate new instances from the model
- They differ mainly on how do they forget
  - No forgetting, sliding window forgetting, damped window forgetting,...
- and which part of the model is affected
  - Complete model reset, partial reset, ...
- So far, we focused on fully-supervised learning and we assumed availability of class labels for all stream instances
  - Semi-supervised learning
  - Active learning
- Dealing with class imbalances, rare-classes
- Dealing with dynamic feature spaces

further reading

- Joao Gama: *Knowledge Discovery from Data Streams* (http://www.liaad.up.pt/area/jgama/DataStreamsCRC.pdf)
- Gibbons, Phillip B., Yossi Matias, and Viswanath Poosala. *Fast incremental maintenance of approximate histograms*. VLDB. Vol. 97 (1997)
- Datar, Mayur, et al. *Maintaining stream statistics over sliding windows*. SIAM Journal on Computing 31.6 (2002)
- Klinkenberg, R., and Renz I. *Adaptive information filtering: Learning drifting concepts*. Proc. of AAAI-98/ICML-98 workshop Learning for Text Categorization (1998)
- Page, E. S. *Continuous Inspection Scheme*. Biometrika 41 (1954)
- Kifer, Daniel, Shai Ben-David, and Johannes Gehrke. *Detecting change in data streams*. VLDB. (2004)

further reading

- Spath, H. *Cluster Analysis Algorithms for Data Reduction and Classification*. Ellis Horwood (1980)
- L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, R. Motwani: *Streaming-Data Algorithms for High-Quality Clustering*. ICDE. (2002)
- Zhang, Tian, Raghu Ramakrishnan, and Miron Livny. *BIRCH: an efficient data clustering method for very large databases*. ACM SIGMOD (1996)
- Aggarwal, Charu C., et al. *A framework for clustering evolving data streams*. Proc. VLDB (2003)
- Manku, Gurmeet Singh, and Rajeev Motwani. *Approximate frequency counts over data streams*. Proc. VLDB. (2002)