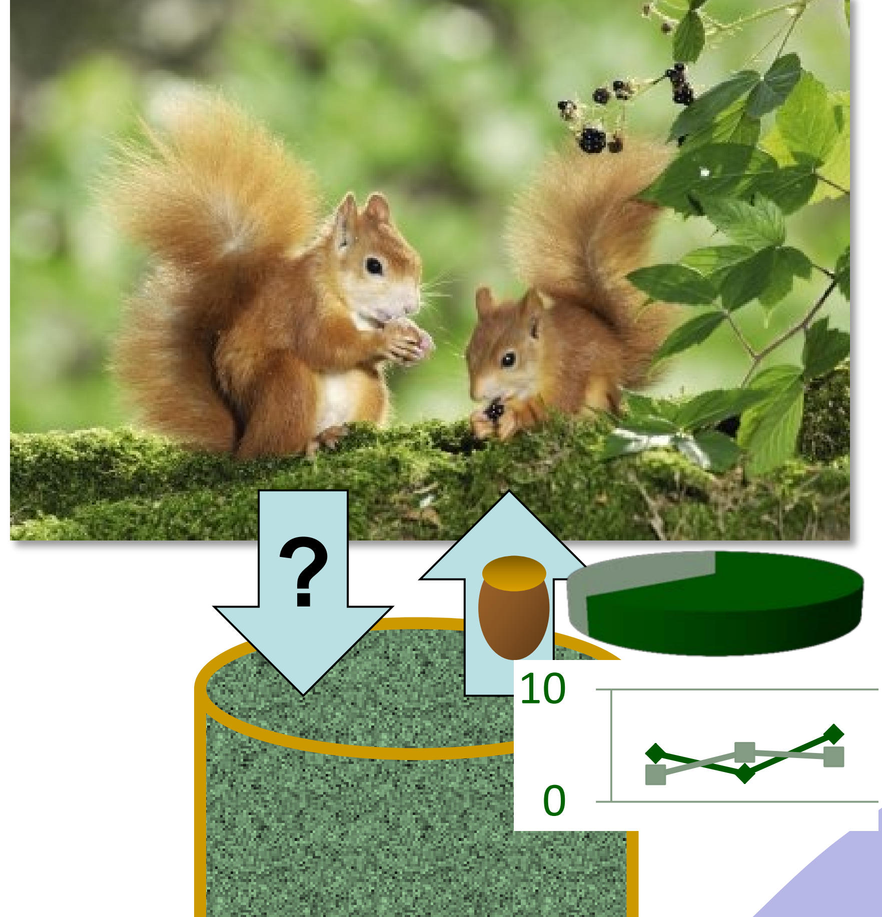
Stream mining

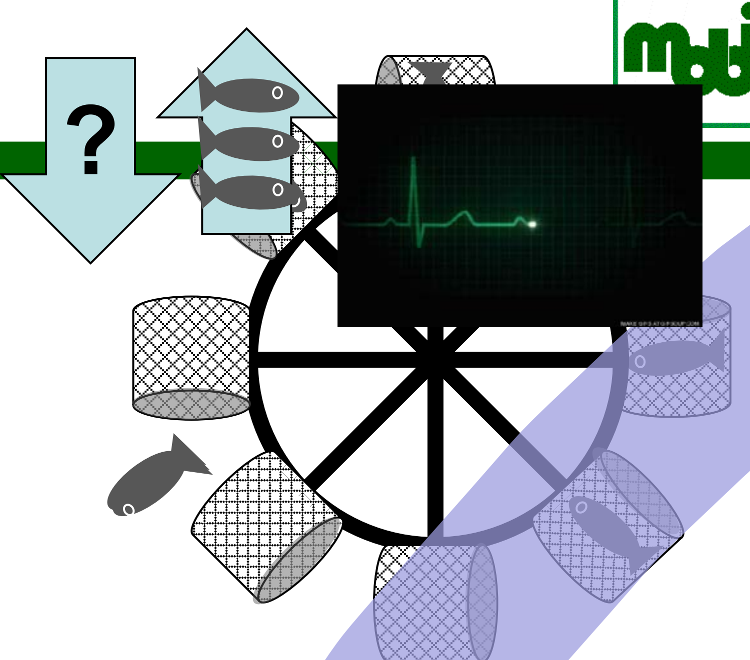
Tutorial for SummerSoc 2016

Prof. Dr. Daniela Nicklas

Lehrstuhl für Informatik,
insbes. Mobile Software Systeme / Mobilität
Otto-Friedrich-Universitaet Bamberg
An der Weberei 5, 96047 Bamberg

Data analytics approaches





Stream mining – learning from streams

Data stream management

- Introduction 1

Data Mining / Machine Learning

- Introduction 2

Stream mining

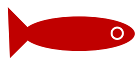
- Why is it difficult?
- How can it be done?
- Example (Frequent Pattern Mining)

1. Introduction to Data Stream Managmenent

- Terms and definitions
- Challenges
- Main concepts: windows, operators, query plans
- Systems : Odysseus, Spark, Flink

Introduction (Terms)

- Data
  - Digital representation of something (0010110100101110000)
  - With semantic: **information**
  - With interpretation: **knowledge**
- Data Stream
  - "A data stream is a **real-time**, **continuous**, **ordered** (implicitly by arrival time or explicitly by timestamp) **sequence of items**."
    Golab und Özsu [Gola03b]
- Event
  - An event is an **occurence** within a particular system or domain; it is something that has happend, or is contemplated as having happend in that domain. The word event is also used to mean a **programming entity** that represents such an occurrence in a computing system.
- Complex Event
  - An event that is defined by the occurance of a certain **pattern** of events



a fish

<salmon, 32>

data

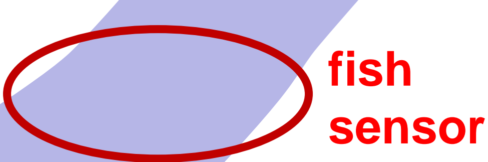
"A salmon with a length of 32cm"

information

<salmon, 32, 2014-05-20-20:29:23>

"We have seen a salmon with a length of 32cm on the 5th of May 2014 at 20:29:23

more data;
event



Can we learn this automatically?

<GoFishing, 2014-05-20-20:2

complex event

GoFishing:
more than 10 salmons
with length > 30
within one minute

complex event
definition /
pattern

More terms

- **(Complex) event processing**
  - A form of computing that performs operations on events
- **Data stream processing**
  - Continuous computing on data streams
  - Immediate action
  - Push-based (reacts on incoming data)
  - Optimized for low latency
- **Data stream management**
  - Support for data stream processing applications
  - Provides higher-level interfaces (e.g., queries or script languages)
  - Optimized for many sources, many applications / queries
  - Goal: reduce development effort / increase maintainability
  - Realizes data stream processing by operators
- **Data stream management system (DSMS)**
  - A system that provides data stream management features

DSM – Challenges

- Continuously arriving data → Potential infinite
  - Blocking behaviour of query operators not adequate for stream processing
    - E.g., how to compute the average?

    → **<u>unblocking by windows</u>**
  - Temporal relationship of streaming data
- Heterogeneity of data to be processed
  - need support for different data models, formats, data schemata
- Many stream-based applications deal with sensor data
  - need support for uncertainty and vagueness of information
  - high input / throughput rates, low latency of data tuples
- Changing input rates
  - need for ressource planning or for burst adaptions

    → prioritazation, load shedding

Features of Data Stream Management Systems

- Programming Abstraction
  - declarative: query (e.g., CQL)
  - functional: flow graph (e.g., SPL)
  - enables optimizations
    - query rewriting, parallelization, …
  - better maintanance of systems

  "Using a DSMS on data streams is like using a DBMS instead of files"

- Data flow vs. event bus as in many complex event processing (CEP) engines
  - DSMS offers early filtering to scale up event pattern processing

- Data streams can be unbounded:
  - issues with sorting, joins, aggregation
  - approximate answers
  - window semantics

CQL example

```
SELECT ego.pos
RANGE 10 secon
radar RANGE 15
WHERE ego
radar.spe
AND s2.po
```

Some DSMS provide CEP operators

executable query plan

PQL example

```
trackingdata_
KEYVALUETOTUPLE({sc
'Integer'],['data.e
'Integer'],['data.z
'String'],['data.ma
['data.battlevel',
'String'],['data.Mi
'String'],['data.te
'Integer'],['data.R
type='Beacon', keep
ACCESS({source='Source',
wrapper='GenericPush',
transport='HTTPServer', protocol='JSON',
datahandler='KeyValueObject',
options=[['path', '/'],['port',
'8080'],['schedule.delay', '2000']]}))
```

Window definitions

- Monotone window operator to split stream into segments
  - Needed for all stateful operators
  - → By definition needed for mining, too
- Window size can be based on:
  - number of elements (e.g., last 100 elements)
  - time (e.g., last 5 seconds)
    - Application time: defined by data (e.g., observation)
    - System time: defined by processor clock
      → Non-deterministic, network latency influences results!
  - predicates of elements (e.g., value between two thresholds)

- Window stride: how it moves
  - jumping or tumbling (no overlaps, data processed once)
  - sliding (continously, overlaps, data processed more than once)
  - sampling (no overlaps but gaps, some data is not processed)

More window types (by [JiAg07])

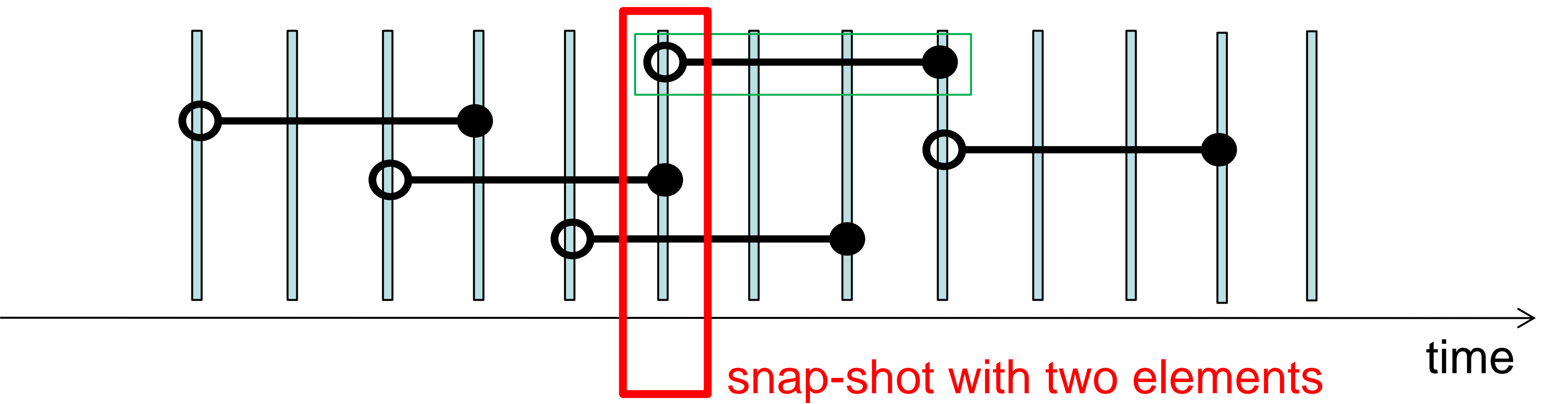
- Landmark window:
  - window from a starting point i to the current time-point t : W[i, t]
  - special case: i = 1 (from the beginning, entire data stream)
  - require efficient single-pass mining algorithm
  - each time-point is equally important
- Damped window model:
  - assign more weight to the recently arrived transactions
  - e.g.: define a decay rate [ChLe03] and use it to update the previously arrived transactions (by multiplication); count of item set is also defined based on the weight of each transaction
- Tilted-time window:
  - freqent itemsets over a set of windows corresponding to different time granularities
  - e.g.: Last hour: every minute; previous hour: every five minutes; …
  - Basis for FP-stream [GHPY02]

Window implementation: interval approach

- Processing model:
  - each data element has an validity interval
  - only elements that are concurrently valid are processed together

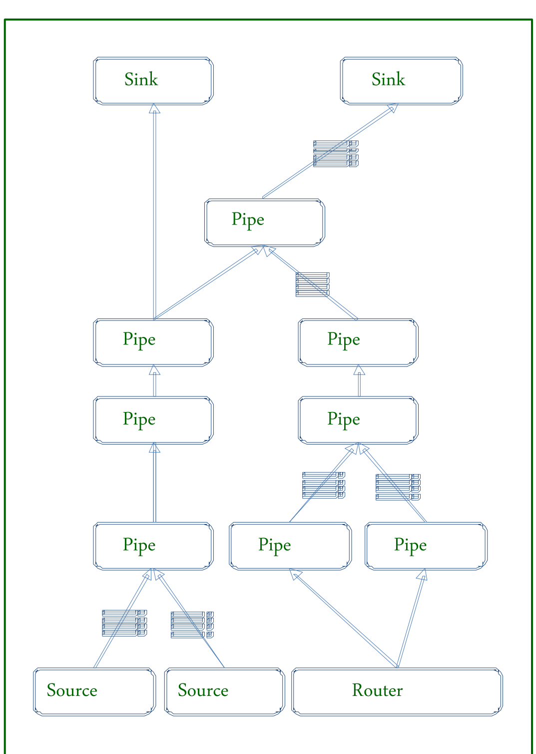
- Enables so-called snap-shot reducability
  - logical equivalence to non-stream operators, e.g., relational algebra
  - logical data stream: $S^l = \{ (e,t,n) \}$
    - (each element can occur *n* times at time t)
  - equivalent physical data stream: $S^p = \{ (e, [t_s, t_e)) \}$:

t_e is not included



Operators (in DSMS)

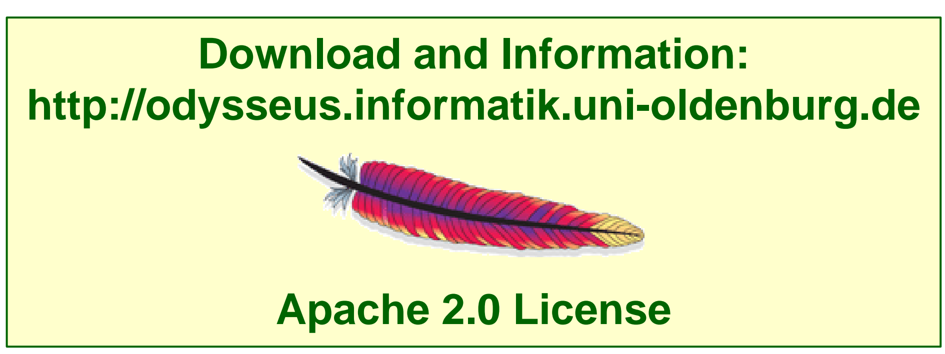
- **Operator**
  - building block for stream application computing, e.g., a relational operator or a function
  - consumes one or multiple data streams, produces one or multiple data streams
  - Examples:
    - SELECT/FILTER (filters data by predicates)
    - JOIN (joins data from two streams)
    - MY_OP (user defined operator)
    - In some DSMS: WINDOW
- **Operator graph or query plan**
  - A directed graph G = (E,V) where
    - E (edges) are operators and
    - V (vertices) are data flow connections between operators
  - Data stream queries are translated to query plans / operator graphs by a DSMS



Operator graph/ query plan

Example: Data Stream Management by Odysseus

- Flexibe Open Source Data Stream Management Framework [AGG+12]
- Time intervals as stream model
  - Semantically defined and deterministic processing
  - System time independent
  - Robust against race conditions or bursts
- Built-in optimization techniques
  - Reduction of system load and latencies
- Framework architecture (OSGi)
  - Extensible for new requirements, operators, scheduling strategies, …
  - Adaptable even at runtime
  - <u>Provides a machine learning plugin</u>



Odysseus DSMS: Architecture and basic functions

Application

SELECT ...
FROM ...
WHERE ...

Event Stream Management

Sink Sink
Op
Op Op
Op Op
Op Op Op
Source Source Router

Scheduler
Query Interface

Transform: Relational, Triple, ..., NF2

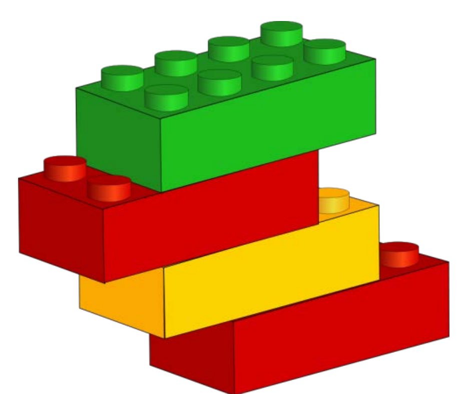
Rewrite: Relational, Triple, ..., NF2

Translate: CQL, PQL, ..., sSPARQL

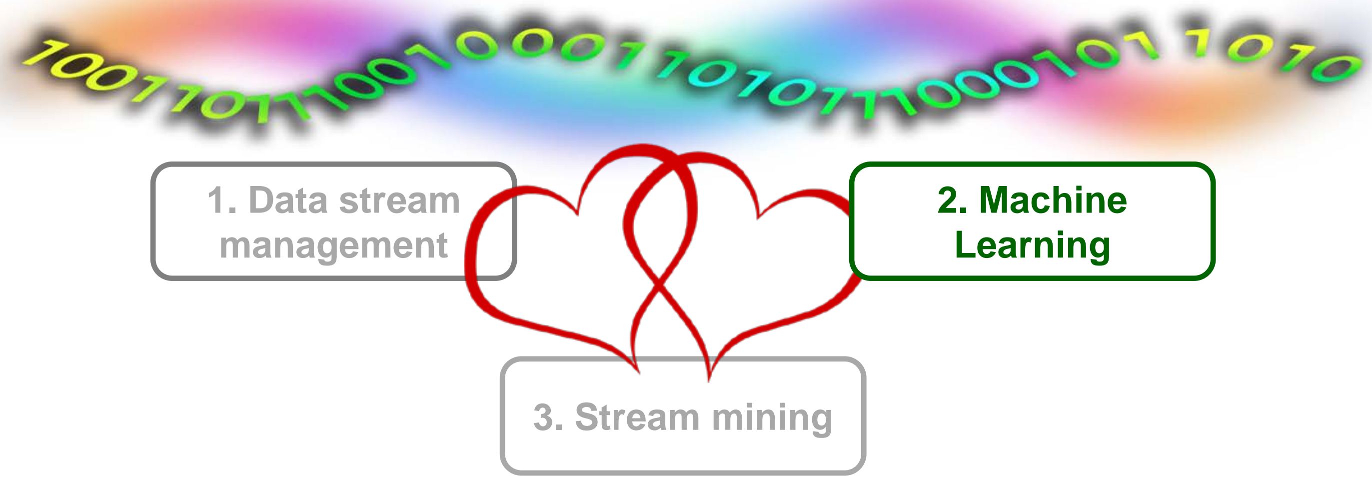
Sensors

1. Application creates query
2. Translates the query into logical query plan
3. Optimizes the logical query plan
4. Transforms the logical into a physical query plan
5. Executes the query plan

Operator architecture



- Operator: building blocks for data stream processing
- Provide interface to connect to and from other operators
  - like a Lego brick
- If the set of implemented operators of a DSMS is not sufficient, a new operator needs to be implemented
  - often called "UDO" (user defined operator)
- Design decision:
  - how to separate concerns? One big operator or many small operators?
  - big operators: less overhead, small operators: more re-use and optimization
- To implement an UDO, you always need to …
  - implement functionality of the UDO in a programming language supported by the DSMS
- Depending on the DSMS, you might also need to …
  - implement the window / validity management of the tuples
  - implement buffer management
  - implement load shedding
  - implement encryption / security mechanisms

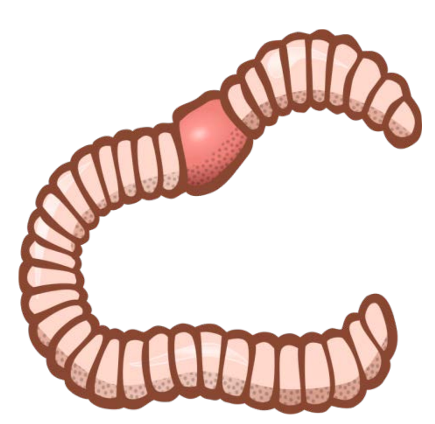


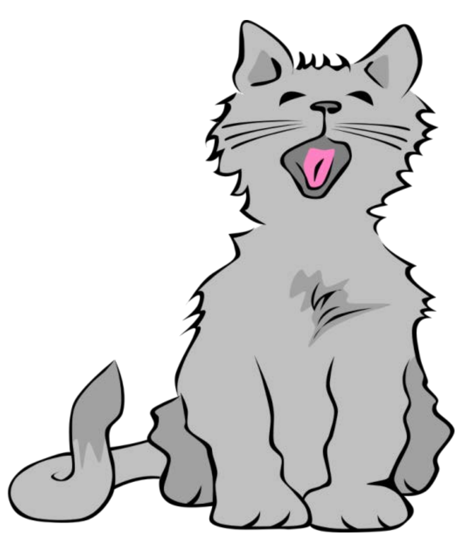
2. Introduction to Machine Learning

- Data mining, machine learning, and the KDD process
- Knowledge-base systems vs. Learning systems
- Top 10 Data mining algorithms

If an expert system–brilliantly designed, engineered and implemented–cannot learn not to repeat its mistakes, it is not as intelligent as a worm or a sea anemone or a kitten.

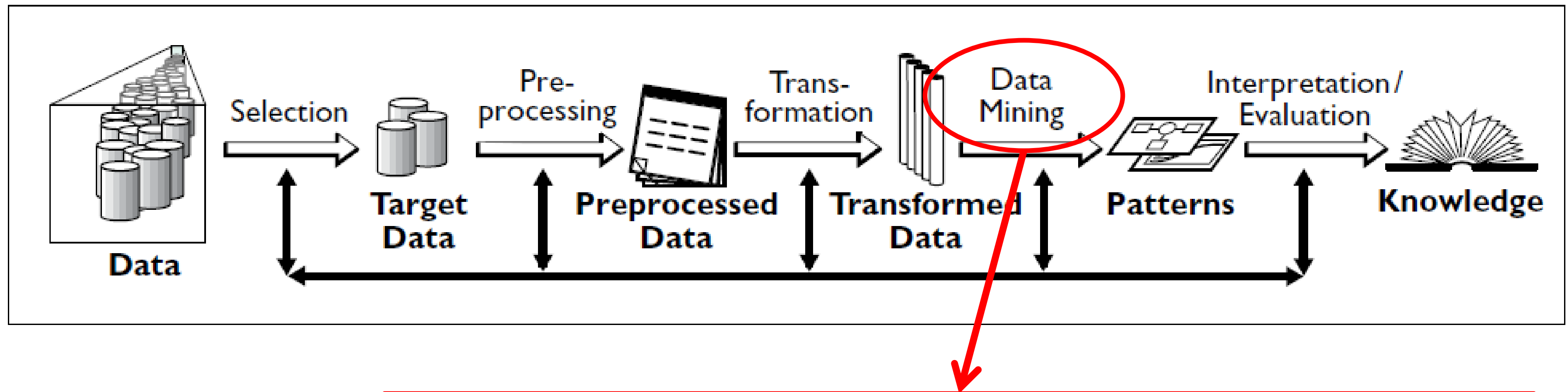
Oliver G. Selfridge, from The Gardens of Learning





Data mining and machine learning as part of the KDD process

- KDD = Knowledge discovery in databases



"A wide variety and number of data mining algorithms are described in the literature—from the fields of statistics, pattern recognition, machine learning, and databases." [FaPS96]

→ **Data mining and machine learning are often used as synonyms**

[FaPS96] Fayyad, Usama ; Piatetsky-Shapiro, Gregory ; Smyth, Padhraic: The KDD process for extracting useful knowledge from volumes of data. In: *Communications of the ACM* Bd. 39 (1996), Nr. 11, S. 27–34

Knowledge and Learning

Knowledge-based Systems: Acquisition and modeling of common-sense knowledge and expert knowledge

- ⇒ limited to given knowledge base and rule set
- ⇒ Inference: **Deduction** generates no new knowledge but makes implicitly given knowledge explicit
- ⇒ **Top-Down**: from rules to facts

Learning Systems: Extraction of knowledge and rules from examples/experience

- Teach the system vs. program the system
- Learning as **inductive process**
- ⇒ **Bottom-Up**: from facts to rules

Learning as Induction

Deduction		Induction	
All humans are mortal.	(Axiom)	Socrates is human.	(Background K.)
Socrates is human.	(Fact)	Socrates is mortal.	(Observation(s))
Conclusion:		*Generalization:*	
Socrates is mortal.		All humans are mortal.	

Deduction: from general to specific ⇒ **proven** correctness

Induction: from specific to general ⇒ (**unproven**) knowledge gain

> **Induction generates hypotheses not knowledge!**

Inductive Learning Hypothesis

- As shown above inductive learning is **not** proven correct
- The learning task is to determine a hypothesis $h \in H$ identical to the target concept c for all possible instances in instance space X

$$(\forall x \in X)[h(x) = c(x)]$$

- Only training examples $D \subset X$ are available
- Inductive algorithms can at best guarantee that the output hypothesis h fits the target concept over D

$$(\forall x \in D)[h(x) = c(x)]$$

- **Inductive Learning Hypothesis**: Any hypothesis found to approximate the target concept well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples

(First) application to stream learning

Application of hypotheses ("Prediction")

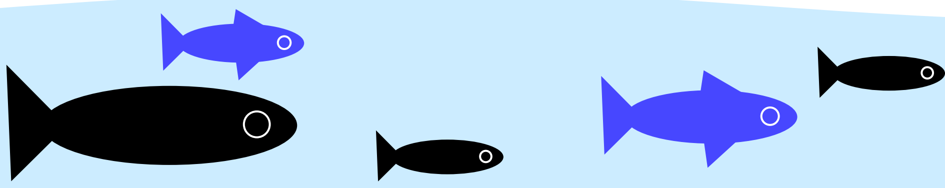
Some hypotheses do not hold
- How (and when) do we know?

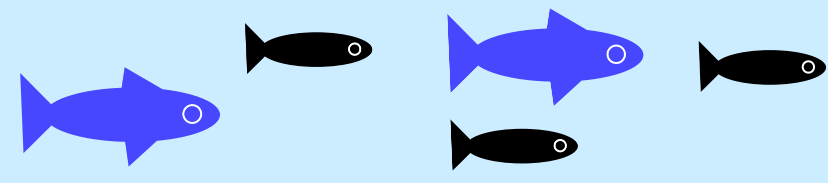
What can we do?
- Accept some errors
- Re-learning (including statistics, "92% of blue fish are bigger than black fish")
- Adaptive learning (continues re-learning)

Window of training examples

Sample hypotheses:

- Blue fish are bigger than black fish
- Blue fish have two visible vertical fins
- Blue fish taste better





Machine learning approaches

- There are MANY machine learning approaches
- In this tutorial (and in general), we need to focus!

Top 10 algorithms in data mining

X Wu, V Kumar, JR Quinlan, J Ghosh, Q Yang… - … and Information Systems, 2008 - Springer

Abstract This paper presents the **top 10 data mining algorithms** identified by the IEEE International Conference on **Data Mining** (ICDM) in December 2006: C4. 5, k-Means, SVM, Apriori, EM, PageRank, AdaBoost, k NN, Naive Bayes, and CART. These **top 10 …**

Zitiert von: 1496 Ähnliche Artikel Alle 96 Versionen Web of Science: 439 In BibTeX importieren Speiche

- Top 10 algorithms:
  - identified by the IEEE international conference on Data Mining (ICDM) 2006
  - experts nominiated up to 10 best-known algorithms with name, brief justification, reference (publication)
  - nominations with less than 50 citations were removed
  - remaining 18 nominations were organized in 10 topics: association analysis, classification, clustering, statistical learning, bagging and boosting, sequential patterns, integrated mining, rough sets, linkmining, and graph mining **http://www.cs.uvm.edu/~icdm/algorithms/CandidateList.shtml**
  - more experts were asked to vote, resulting in this top 10 list.

http://en.wikipedia.org/wiki/Machine_learning, 27.4.2015:

4 Approaches

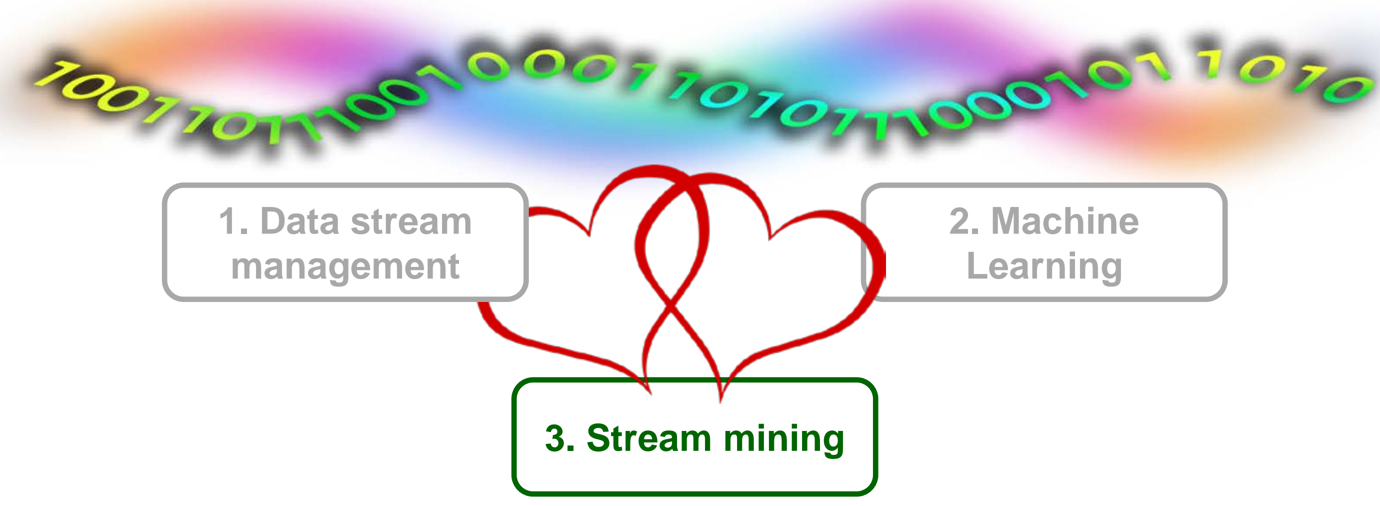
4.1 Decision tree learning
4.2 Association rule learning
4.3 Artificial neural networks
4.4 Inductive logic programming
4.5 Support vector machines
4.6 Clustering
4.7 Bayesian networks
4.8 Reinforcement learning
4.9 Representation learning
4.10 Similarity and metric learning
4.11 Sparse dictionary learning
4.12 Genetic algorithms

Top 10 algorithms of data mining (1-5)

1. C4.5:
   - generates classifiers as decision trees or rulesets
2. k-Means
   - partitions a dataset into user-specified number (k) of clusters
3. SVM (support vector machines)
   - two-class learning, find the "best" (geometrically) classification function to distinguish between members of the two classes
4. Apriori
   – finds frequent itemsets from a transaction dataset and derives association rules
5. EM (expectation maximization)
   - for data observing a random phenomen: estimate the underlying density function (e.g., sensor accuracy)

Top 10 algorithms of data mining (6-10)

6. PageRank ("the algorithm that made Google a success")
   - Search ranking algorithm using hyperlinks on the Web; learns the relevance of a search result for the user
7. AdaBoost
   – Ensemble learning: employs multiple learner to solve problem, weighs the learners to create result
8. kNN: k-nearest neigbor classification
   - finds a group of k objects in training set that are closest to the test object (labelled), computes distance of objects to labelled objects
9. Naive Bayes (aka idiot’s Bayes, simple Bayes, independence Bayes)
   - classification: based on given set of objects that belong to a known class with known vector of variables, constructs a rule to assign future objects to a class, with given probability. Naïve because it assumes independence.
10. CART: Classification and Regression Trees
    - Decision trees, tree-structured data analysis; binary recursive partitioning procedure



3. Stream mining

- Challenges
- Concept drift / Change detection
- Instance selection vs. Aggregation
- Frequent pattern mining as example

With input and slides by Ute Schmid, Machine Learning/Lernende Systeme. Thank you!
http://www.uni-bamberg.de/kogsys/services/teaching/courses/lernende-systeme/

Comparison – Static data mining vs. Stream mining

Table 1. Processing data vs data streams.

No	Parameter	Database	Data streams
1	Data access	May or may not be Sequential	Sequential
2	Available Memory	Flexible	Limited memory
3	Data spread	Need not be	Distributed
4	Computation Results	Accurate	Approximate findings
5	Data Scan	Flexible	Limited to one scan
6	Algorithms	Processing time is not a constraint	Processing time is most important as data may skip
7	Sampling	Not required	Complex to decide when to sample data
8	Data speed	Can be ignored	Data arrival rate is higher than processing rate
9	Data modelling	Persistent	Modeled as Transient Data streams
10	Data Schema	Static	Dynamic

M. S. B. PhridviRaja und C. V. GuruRao, „Data mining : past present and future - a typical survey on data streams“, *CoRR*, Bd. abs/1605.01429, 2016.

Stream mining challenges

- Scalability
  - increasing number of data, processing time must match input time
    - process a data item at most once (single-pass)
    - or limit the amount of data to be processed to allow fast multi-passes
- Temporal order
  - often, there is an inherent temporal component
  - data evolves over time (= concept drift)

  → focus on evolution of the underlying data, recognize evolution (= change detection)
- Distribution
  - often, data occurs at different locations, and limited bandwiths prohibits the transfer to a central server before mining
  - still, distributed processors may have limited resources, too (e.g., small embedded systems in sensor networks)

Stream mining requirements

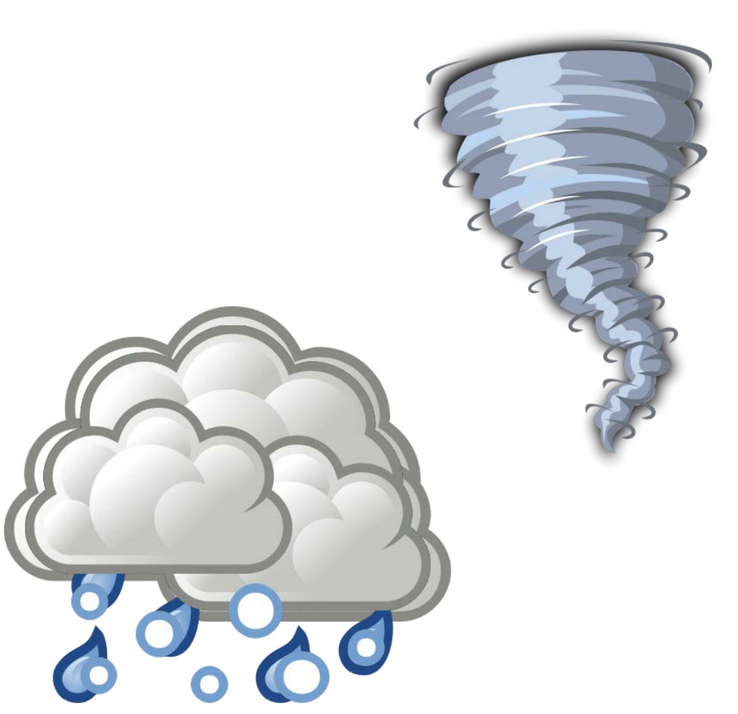
- Avoid congestion:
  - process each data element in constant time
  - average processing time needs to be less or equal to the average update rate
- Avoid memory overflow:
  - process data element with constant memory usage
- Adaptive:
  - adapt to changing characteristics of the data stream and/or phenomenon

"constant": does not increase over time

ok for streams

Concept drift

- Concept of interest: the hypothesis / thing to be predicted
  - e.g., the weather
- Concept depends on context; some context is observable
  - e.g., current temperature at the location
- Other context is hidden
  - (e.g., the weather 100km west of us)
- Concept drift: change of concept
  - often due to hidden context
- Two main classes of concept drift:
  1. sudden concept drift
     - e.g., a wind eddy
  2. gradual concept drift
     - e.g., change in seasons



Virtual concept drift

- Concept drift causes change in the underlying data distribution

  → necessity of revising the current model,

  = "virtual concept drift"
- Virtual concept drift and real concept drift often occur together
- However, virtual concept drift may occur alone:
  - e.g.: understanding of unwanted message stays the same,
    but distribution of different types of spam may change significantly over time

- In both cases (virtual and real), the model needs to be changed

Approaches to handle concept drift

- Instance selection
  - select instances relevant to current concept
  - often window-based, sometimes with changing window sizes
- Instance weighting
  - process weighted instances,
    e.g., according to age and competence with regard to current concept
  - however, sensitive to overfitting
- Ensemble learning
  - maintain a set of concept descriptions
  - combine with voting / weighted voting
  - often incremental approaches

- When to update the model?
  - can be costly, should be done only if evitable
  - different criteria can be used, e.g., average confidence or number of instances instances under a given confidence threshold

[Tsym04] Tsymbal, Alexey: The problem of concept drift: definitions and related work. In: Computer Science Department, Trinity College Dublin Bd. 106 (2004)

Learning on Data Streams: Approaches

- (Potentially) update model with every new element
- Two main approaches:
  - Aggregation / synopses / histograms
    - aggregate learned knowledge
    - Pro: can span longer time periods
    - Con: does not deal well with concept drift, accuracy drops over time
  - Window-based
    - e.g.: clustering over the last 10 minutes
    - Pro: robust with respect to concept drifts
    - Con: re-learns with every window, forgets old knowledge
- And:
  - Hybrid approaches / tilted time frames:
    - Adapt to time: younger data → more details, older data → less data

Example: Frequent Pattern Mining on Streams

1. With landmark windows and aggregation
2. With tilted time windows and aggregation
3. With sliding time windows, no aggregation

Frequent pattern mining: problem definition

- Collection of objects: Dataset

  $D = \{o_1, o_2, \ldots, o_{|D|}\}$
- Set of all possible interesting patterns occuring in D:

  $P = \{p_1, p_2, \ldots, p_n\}$
- Counting function:

  $g : P \times O \rightarrow \mathbb{N}$

  g(p, o) returns the number of times p occurs in o
- Indicator function:

  $I: \mathbb{N} \rightarrow \{0, 1\}$

  if $g(p, o_j) > 0$: $I(g(p, o_j)) = 1$ else 0 fi
- Support of pattern $p \in P$ in dataset D:

$$support(p) = \sum_{j=0}^{|D|} I(g(p, o_j))$$

Popular example:
Basket analysis,
Beer and diapers are
often bought together

Stream pattern mining: challenges

- Scalability:
  - Search space grows exponential with number of elements, and number of elements can be potentially infinite
  - Cardinality of answering set can be very large

  → Need for approximate answers and memory-efficient algorithms
- Efficiency:
  - Frequent pattern mining relys on down-closure property to prune infrequent pattern
    - this can be compute-intensive

  → Algorithm needs to keep up the space with high-speed data streams
- Quality:
  - Algorithms can only deliver approximate answers. But: how good is it?

  → Need for user-controllable quality parameters

Frequent itemset mining by Jin et al. [JiAg07]

- Basis: find frequent elements by Karp et al. [KaSP03]

Basis: algorithm to find frequent elements

- Based on work by Karp, Papadimitriou and Shenker to find frequent elements [KaSP03]:
- Problem:
  - given a sequence of length N and a threshold Θ ($0 < \Theta < 1$): determine elements that occur with frequency $> N\,\Theta$
- Example: (a,n,a,l,y,t,i,c,a,l)
  - sequence of length 10, $\Theta = 0.2$
  - frequent elements with frequence > 10*0.2 = 2: {a, l}
- Trivial algorithm:
  - count frequency of all distinct elements
  - check if any of them has the desired frequency
  - requires $O(n)$ memory

Find frequent elements by Karp et al. [KaSP03]

- Finding the majority element in a sequence

  = appears more than half the time in an entire sequence ($\Theta > 0.5$)

  - find two distinct elements and eliminate them from the sequence
  - repeat this process only one distinct element remains
  - if there is a majority element, it will remain (but not vice versa)

- Generalization:

  - Find any $1/\Theta$ distinct elements in the sequence and eliminate them together
  - Repeat until no more than $1/\Theta$ distinct elements remain in the sequence
  - This can be only done at most $N / (1/\Theta) = N\Theta$ times

  → Any element that appears more than $N\Theta$ times will be left in the sequence

  – however, the elements left do not necessarily appear with frequency greater than $N\Theta$

- Requires only $O(1/\Theta)$ memory

Issues in frequent itemset mining

... as addressed by Jin et al. [JiAg07]

- Dealing with transaction sequences
  - Karp et al. [KaSP03] assumes that a sequence compromises single elements (1-items)
  - For frequent itemset mining, each transaction has a number of items
  - Length of transactions can be different, but easier with fixed length
- Providing an accuracy bound
  - Karp et al. [KaSP03] can provable find a superset of frequent items
  - However, no accuracy bound is provided for the item(set)s in the superset
- Dealing with k-itemsets
  - Karp et al. [KaSP03] only finds the frequent items, or 1-itemsets
  - For frequent itemset mining, we need to find all k-itemsts, $k \geq 1$, in a single pass

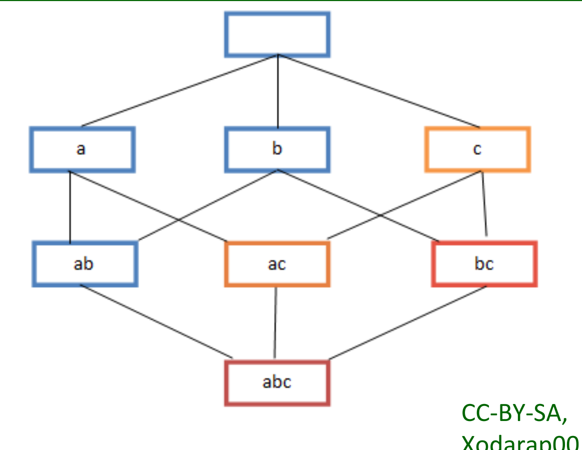
→ extending the algorithm step by step to cope with all challenges

Providing an accuracy bound

- Bound: user-defined parameter *e,* with $0 < e <= 1$
- Reported frequent itemsets do occur more than
  $(1-e)*\Theta*|D|$
  times in the data set
- Properties:
  1. if an itemset has frequency more than ϴ, it will be reported
  2. if an itemset is reported as a potential frequent itemset,
     it must have a frequence more than ϴ(1-e)

Lattice and basic routines

- Keep frequent itemsets in Lattices (because of down-closure property)
  - $L = L_1 \cup L_2 \cup \ldots \cup L_k$
  - k: largest frequent itemset,
    L_i are the potential frequent i-itemsets



CC-BY-SA, Xodarap00

- Subroutines (used by all extensions of the algorithm):

```
define update(transaction t, Lattice L, i):
  for all i-subsets s of t:
     if s in Li: s.count++
     elif i <= 2: Li.insert(s)
     elif all i-1-subsets of s in Li-1: Li.insert(s)

define ReducFreq(Lattice L, i):
  foreach i itemsets s ∊ Li:
     s.count--
     if s.count == 0: Li.delete(s)
```

Deal with variable length transactions

- Variable f:
  - weighted average of the number of 2-itemsets in each transaction processed so far
  - recent transactions get higher weights
- Algorithm for f:

```
define TwoItemsetPerTransaction(Transaction t):
  global X # number of 2 itemset
  global N # number of Transactions
  local f
  N++
  X = X + (|t| choose 2)
  f = ⌈X/N⌉
  if |L₂| >= ⌈1/Θ * e⌉ * f:
     N = N - ⌈1/Θ * e⌉
     X = X - ⌈1/Θ * e⌉ * f
  return f
```

Binomial coefficient:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

$$\binom{n}{0} = \binom{n}{n} = 1$$

Full algorithm for itemset mining

```
define StreamMining (Stream D, Θ, e):
     global Lattice L = ∅
     global Buffer T = ∅
     local Transaction t = "next arriving transaction"
     f = 0 # weighted average of 2-itemsets in transaction
     c = 0 # number of ReducFreq invocations
     foreach (t in D)
        T = T ∪ {t}
        Update(t, L, 1)
        Update(t, L, 2)
        f = TwoItemsetPerTransaction(t)
        if |L2| >= ⌈1/Θ e⌉ * f:
           ReducFreq(L, 2)
           c++
           i = 2
           while Li != ∅:
              i++
              foreach (t in T): update (t, L, i)
              ReducFreq(L, i)
           T = ∅
     # remove all items whose reported frequency is too low
     foreach s in L:
        if s.count <= Θ*|D| - c: Li.delete(s)
     return(L)
```

FP-stream by Giannella [GHPY02]

- Basis: FP-Tree [HaPY00]

FP-Stream - Overview

- Stream-based algorithm for finding frequent itemsets by Giannella et al. [GHPY02]
- Hybrid: approach based on both instance selection and aggregation
- Main concepts:
  - Data structure inspired by FP-tree [HaPY00]
  - Tilted time-window [CDHW02]
- In contrast to Jin et al. [JiAg07]:
  - no landmark window, but tilted time-window
    - →more robust to concept drift
    - →still keeps some history

FP-Tree

- Han, Pei and Yin 2000 [HaPY00]

FP-Tree: Overview

- From FP-Growth, an alternative algorithm to Aprioi [AgSO94], which …
  - mines from stored data
  - uses compressed data base to increase scan performance
  - uses FP-Tree as central data structure
- Terms:
  - *support* (or occurence frequency) of an itemset I: absolute number of transactions containing I
  - *frequent pattern* A: Item set with a support no less than predefined *minimum support threshold* ξ

FP-Tree: Definition

- FP-Tree is a tree structure that contains ...
  - a root labelled as "null"
  - a set of *item prefix subtrees* as children of the root
  - a *frequent-item header table*
- Nodes in the *Item prefix subtree* consist of three fields:
  1. item-name: which item is represented?
  2. count: number of transactions represented by the portion of the path reaching this node
  3. node-link: points to the next node in the FP-tree carrying the same item-name, or null if there is none
- Entries in the *Frequent-item header table* consist of two fields:
  1. item-name: which item is represented?
  2. head of node-link: points to the first node in the FP-tree carrying the item-name

FP-Tree: Construction algorithm

- Input: a transaction database DB and a minimum support ξ
- Output: its frequent pattern tree FP-tree
- Method:
  1. scan DB, collect set of frequent items ("1-itemsets") and their supports;
     L = F.sort(descending order of support) # list of frequent items
     T = new(FP-tree) # root is labelled as "null"
  2. For each transaction t in DB do:
     - R = select and sort the frequent items in t according to L
     - call insert_tree(R, Tree)
- insert_tree(R,Tree):
  - r = R[0]
  - if Tree has a child N with N.item-name = r.item-name: N.count++
    else: create new node N with
    - N.count = 1
    - N.parent-link → Tree
    - node-link → nodes with same item-name
  - if R[1:] is not empty: call insert_tree(R[1:], N)

Example: Step 1

- minimum support ξ = 3.

t	Items
100	f, a, c, d, g, i, m, p
200	a, b, c, f, l, m, o
300	b, f, h, j, o
400	b, c, k, s, p
500	a, f, c, e, l, p, m, n

FP-tree → T: root

L:

item-name
f
c
a
b
m
p

← list of frequent items (sorted descending)

Example: Step 2.100.1

- minimum support ξ = 3.

t	Items	(Ordered) frequent items of t: R
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	
300	b, f, h, j, o	
400	b, c, k, s, p	
500	a, f, c, e, l, p, m, n	

R: f, c, a, m, p

insert_tree(R, T):
- r = R[0] = 'f'
- Child? No!
  - create node N
- R[1:] = [c, a, m, p]
  - not empty:
    call insert_tree(R[1:], N)

item-name	head of node-links
f	
c	
a	
b	
m	
p	

Example: Step 2.100.2

- minimum support ξ = 3.

t	Items	(Ordered) frequent items of t: R
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	
300	b, f, h, j, o	
400	b, c, k, s, p	
500	a, f, c, e, l, p, m, n	

R: c, a, m, p

insert_tree(R, T):

- r = R[0] = 'c'
- Child? No!
  - create node N
- R[1:] = [a, m, p]
  - not empty:
    call insert_tree(R[1:], N)

item-name	head of node-links
f	
c	
a	
b	
m	
p	

root

T

f:1

c:1

N

Example: Step 2.100.3

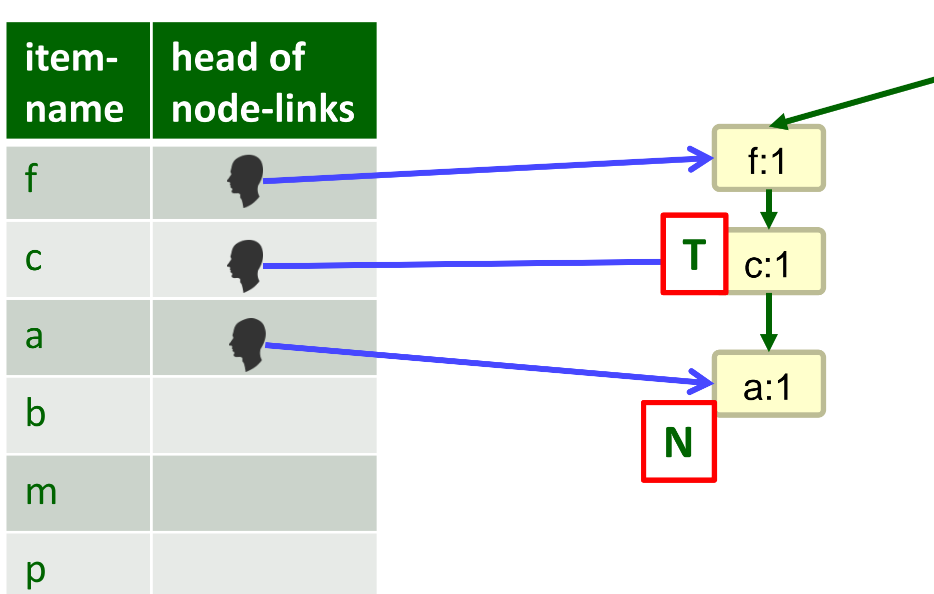
- minimum support ξ = 3.

t	Items	(Ordered) frequent items of t: R
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	
300	b, f, h, j, o	
400	b, c, k, s, p	
500	a, f, c, e, l, p, m, n	

R: a, m, p

insert_tree(R, T):
- r = R[0] = 'a'
- Child? No!
  - create node N
- R[1:] = [m, p]
  - not empty:
    call insert_tree(R[1:], N)

item-name	head of node-links
f	
c	
a	
b	
m	
p	



Example: Step 2.100.4

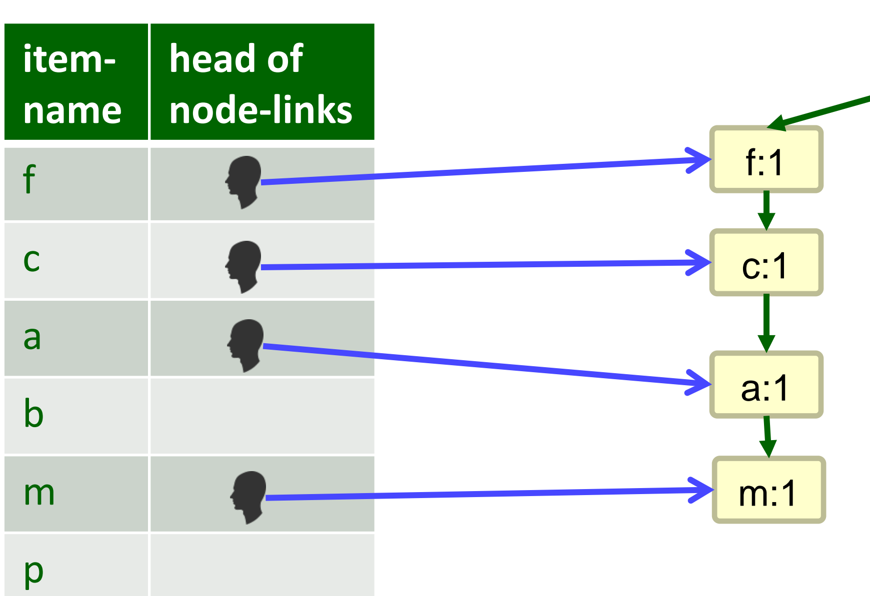
- minimum support ξ = 3.

t	Items	(Ordered) frequent items of t: R
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	
300	b, f, h, j, o	
400	b, c, k, s, p	
500	a, f, c, e, l, p, m, n	

R: m, p

insert_tree(R, T):
- r = R[0] = 'm'
- Child? No!
  - create node N
- R[1:] = [p]
  - not empty: call insert_tree(R[1:], N)

item-name	head of node-links
f	
c	
a	
b	
m	
p	



Example: Step 2.100.5

- minimum support ξ = 3.

t	Items	(Ordered) frequent items of t: R
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	
300	b, f, h, j, o	
400	b, c, k, s, p	
500	a, f, c, e, l, p, m, n	

insert_tree(R, T): **R: p**

- r = R[0] = 'p'
- Child? No!
  - create node N
- R[1:] = []
  - empty!

item-name	head of node-links
f	
c	
a	
b	
m	
p	

Example: Step 2.200.1

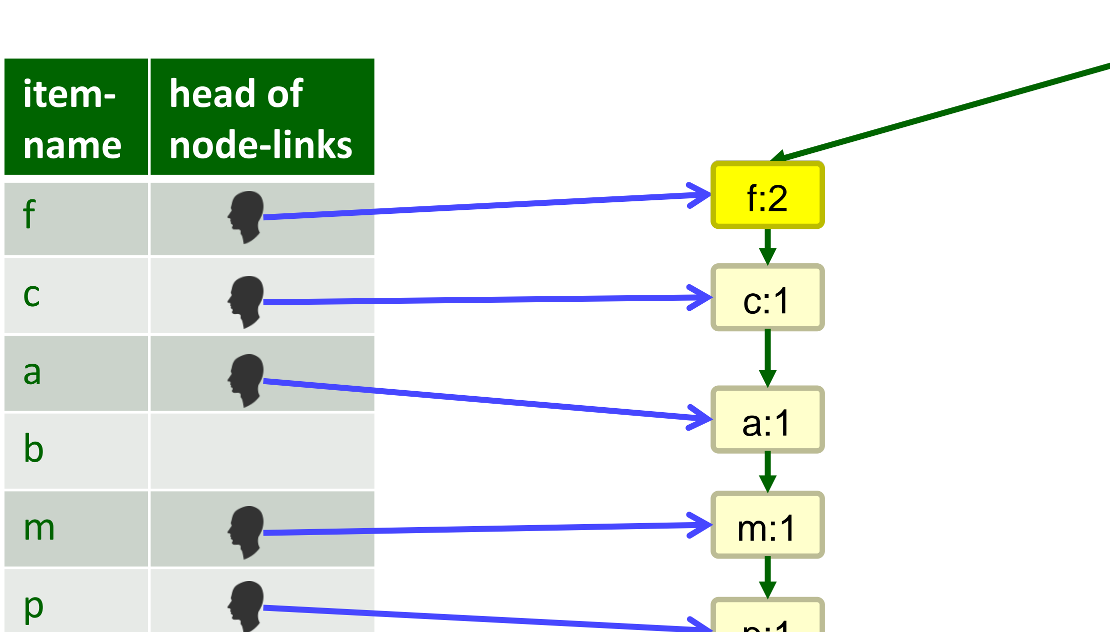
- minimum support ξ = 3.

t	Items	(Ordered) frequent items of t: R
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	
400	b, c, k, s, p	
500	a, f, c, e, l, p, m, n	

insert_tree(R, T):

R: f, c, a, b, m

- r = R[0]
- Child?
  - no? create node N
  - yes? N.count++
- R[1:] not empty?
  - call insert_tree(R[1:], N)



Example: Step 2.200.2

- minimum support ξ = 3.

t	Items	(Ordered) frequent items of t: R
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	
400	b, c, k, s, p	
500	a, f, c, e, l, p, m, n	

insert_tree(R, T):

R: c, a, b, m

- r = R[0]
- Child?
  - no? create node N
  - yes? N.count++
- R[1:] not empty?
  - call insert_tree(R[1:], N)

item-name	head of node-links
f	
c	
a	
b	
m	
p	

root

f:2 T

c:2

a:1

m:1

p:1

Example: Step 2.200.3

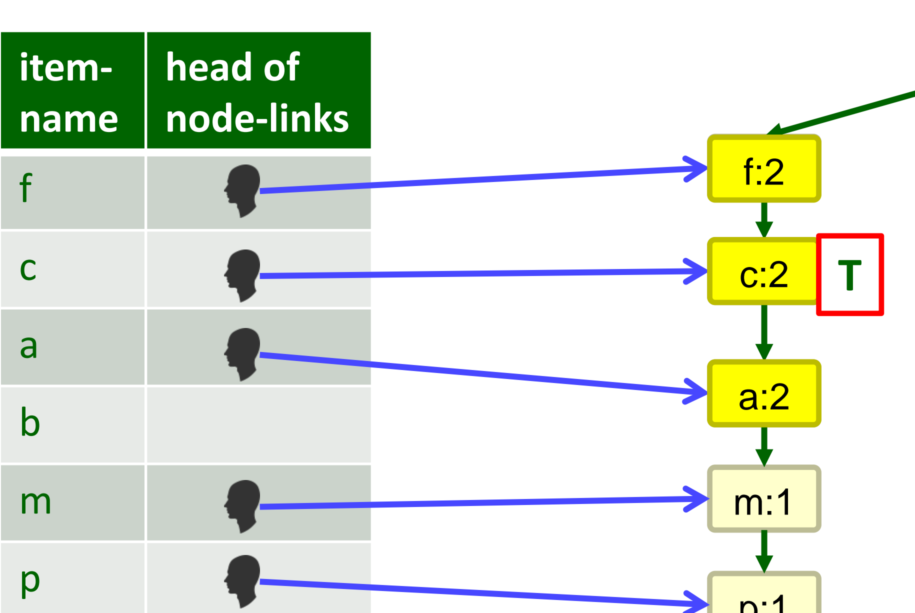
- minimum support ξ = 3.

t	Items	(Ordered) frequent items of t: R
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	
400	b, c, k, s, p	
500	a, f, c, e, l, p, m, n	

insert_tree(R, T): **R: a, b, m**

- r = R[0]
- Child?
  - no? create node N
  - yes? N.count++
- R[1:] not empty?
  - call insert_tree(R[1:], N)

item-name	head of node-links
f	
c	
a	
b	
m	
p	



Example: Step 2.200.4

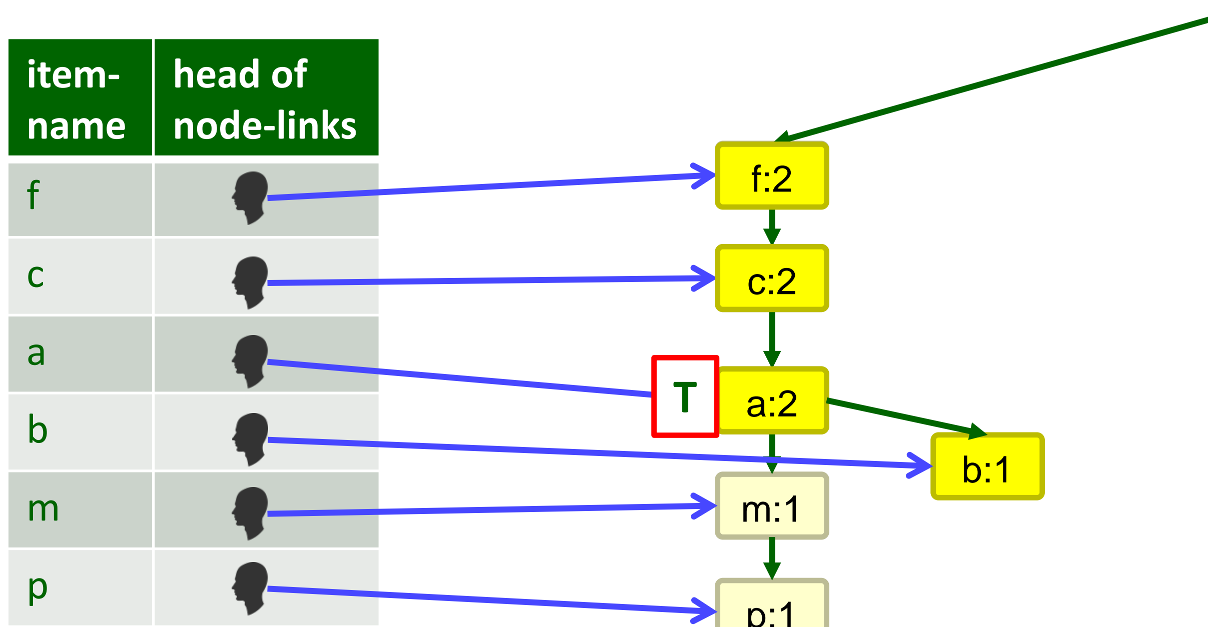
- minimum support ξ = 3.

t	Items	(Ordered) frequent items of t: R
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	
400	b, c, k, s, p	
500	a, f, c, e, l, p, m, n	

insert_tree(R, T):

R: b, m

- r = R[0]
- Child?
  - no? create node N
  - yes? N.count++
- R[1:] not empty?
  - call insert_tree(R[1:], N)



Example: Step 2.200.5

- minimum support ξ = 3.

t	Items	(Ordered) frequent items of t: R
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	
400	b, c, k, s, p	
500	a, f, c, e, l, p, m, n	

insert_tree(R, T):

- r = R[0]
- Child?
  - no? create node N
  - yes? N.count++
- R[1:] not empty?
  - call insert_tree(R[1:], N)

R: m

item-name	head of node-links
f	
c	
a	
b	
m	
p	

Example: Step 2.300.1

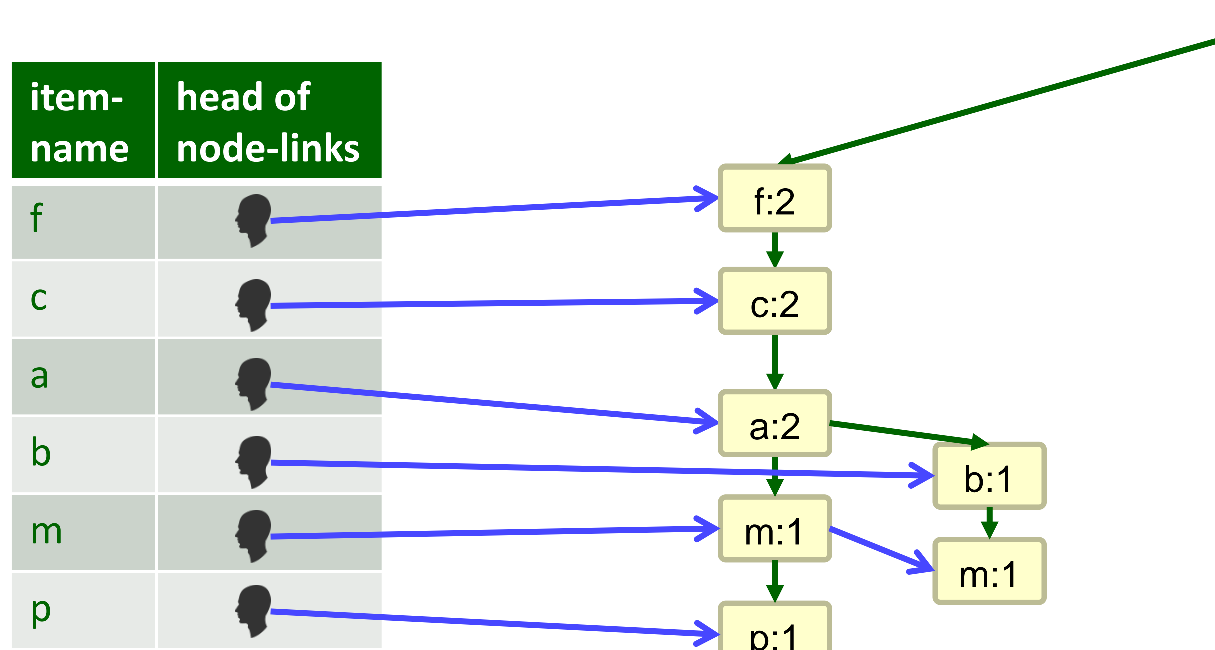
- minimum support ξ = 3.

t	Items	(Ordered) frequent items of t: R
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	
500	a, f, c, e, l, p, m, n	

insert_tree(R, T):

R: f, b

- r = R[0]
- Child?
  - no? create node N
  - yes? N.count++
- R[1:] not empty?
  - call insert_tree(R[1:], N)



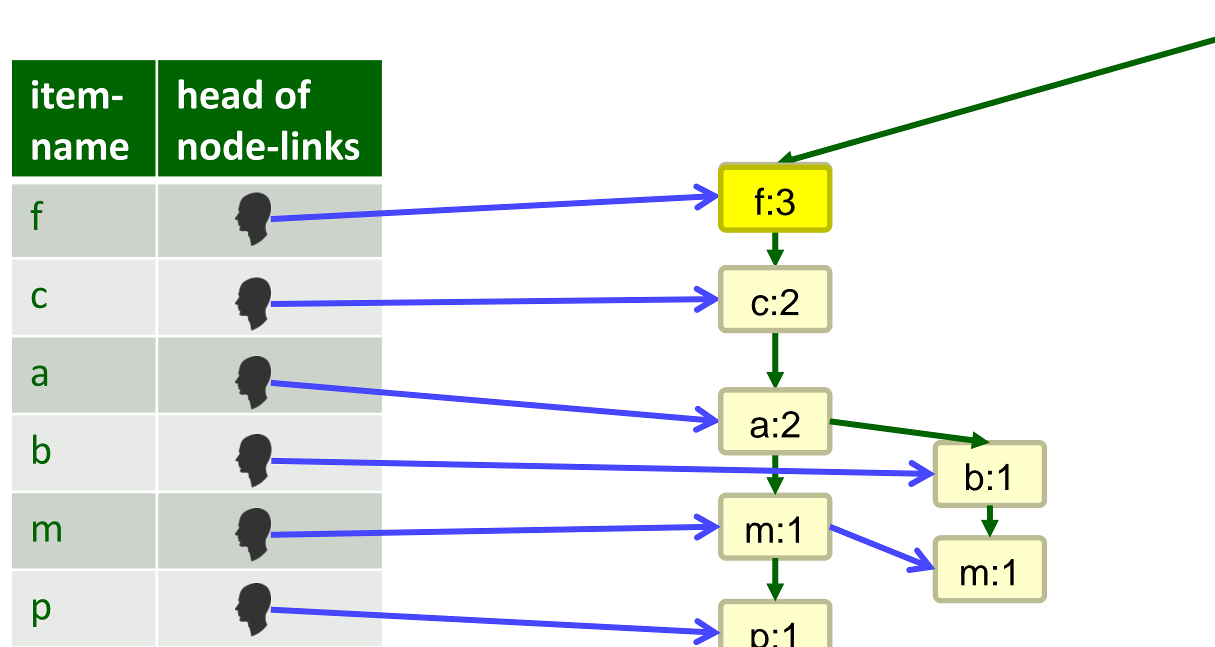
Example: Step 2.300.1

- minimum support ξ = 3.

t	Items	(Ordered) frequent items of t: R
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	
500	a, f, c, e, l, p, m, n	

insert_tree(R, T):
- r = R[0]
- Child?
  - no? create node N
  - yes? N.count++
- R[1:] not empty?
  - call insert_tree(R[1:], N)

R: f, b



Example: Step 2.300.2

- minimum support ξ = 3.

t	Items	(Ordered) frequent items of t: R
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	
500	a, f, c, e, l, p, m, n	

insert_tree(R, T): **R: b**

- r = R[0]
- Child?
  - no? create node N
  - yes? N.count++
- R[1:] not empty?
  - call insert_tree(R[1:], N)

item-name	head of node-links
f	
c	
a	
b	
m	
p	

root

T

f:3

c:2

b:1

a:2

b:1

m:1

m:1

p:1

Example: Step 2.400.1

- minimum support ξ = 3.

t	Items	(Ordered) frequent items of t: R
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	a, f, c, e, l, p, m, n	

insert_tree(R, T):
- r = R[0]
- Child?
  - no? create node N
  - yes? N.count++
- R[1:] not empty?
  - call insert_tree(R[1:], N)

R: c, b, p

T root

item-name	head of node-links
f	
c	
a	
b	
m	
p	

f:3
c:2
a:2
m:1
p:1
b:1
b:1
m:1
c:1

Example: Step 2.400.2

- minimum support ξ = 3.

t	Items	(Ordered) frequent items of t: R
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	a, f, c, e, l, p, m, n	

insert_tree(R, T):

- r = R[0]
- Child?
  - no? create node N
  - yes? N.count++
- R[1:] not empty?
  - call insert_tree(R[1:], N)

R: b, p

item-name	head of node-links
f	
c	
a	
b	
m	
p	

root

f:3
c:2
a:2
m:1
p:1
b:1
b:1
m:1
c:1
T
b:1

Example: Step 2.400.3

- minimum support ξ = 3.

t	Items	(Ordered) frequent items of t: R
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	a, f, c, e, l, p, m, n	

insert_tree(R, T):
- r = R[0]
- Child?
  - no? create node N
  - yes? N.count++
- R[1:] not empty?
  - call insert_tree(R[1:], N)

R: p

item-name	head of node-links
f	
c	
a	
b	
m	
p	

root
f:3
c:2
a:2
m:1
p:1
b:1
b:1
m:1
c:1
b:1
T
p:1

Example: Step 2.500.1

- minimum support ξ = 3.

t	Items	(Ordered) frequent items of t: R
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	a, f, c, e, l, p, m, n	f, c, a, m, p

insert_tree(R, T): **R: f, c, a, m, p**

- r = R[0]
- Child?
  - no? create node N
  - yes? N.count++
- R[1:] not empty?
  - call insert_tree(R[1:], N)

item-name	head of node-links
f	
c	
a	
b	
m	
p	

Example: Step 2.500.2

- minimum support ξ = 3.

t	Items	(Ordered) frequent items of t: R
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	a, f, c, e, l, p, m, n	f, c, a, m, p

insert_tree(R, T): **R: c, a, m, p**
- r = R[0]
- Child?
  - no? create node N
  - yes? N.count++
- R[1:] not empty?
  - call insert_tree(R[1:], N)

item-name	head of node-links
f	
c	
a	
b	
m	
p	

root
T
f:4
c:3
a:2
m:1
p:1
b:1
b:1
m:1
c:1
b:1
p:1

Example: Step 2.500.5

- minimum support ξ = 3.

t	Items	(Ordered) frequent items of t: R
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	a, f, c, e, l, p, m, n	f, c, a, m, p

R: p

insert_tree(R, T):
- r = R[0]
- Child?
  - no? create node N
  - yes? N.count++
- R[1:] not empty?
  - call insert_tree(R[1:], N)

item-name	head of node-links
f	
c	
a	
b	
m	
p	

root

f:4
c:3
a:3
m:2
p:2
b:1
b:1
m:1
c:1
b:1
p:1
T

Result

- Information in FP-tree:
  - for every node: count gives support for the whole path along the FP-Tree vertices
  - e.g.: support(fca) = 3

item-name	head of node-links
f	
c	
a	
b	
m	
p	

Tilted Time Windows

- Chen et al. 2002 [CDHW02]
- Giannella et al. [GHPY02]

Tilt time frame (Chen et al. 2002 [CDHW02])

- Motivation:
  - in stream data analysis, recent data is often of more interest
  - should be analyzed at a finer scale
  - older data could be more aggregated
- Natural tilted time frame:
  - aggregation / analysis intervals are aligned with "natural" time intervals (i.e., hours, weeks, …)

time frames:

t_4	t_2	t_1	t_0

12 months | 31 days | 24 hours | 4 qtrs

Time → Now

Figure 4: A tilt time frame model **[CDHW02]**

Frequent patterns in tilted-time frames [GHPY02]

- For each tilted-time window, a set of frequent patterns is maintained
- Sample queries that can be answered:
  - What is the frequent pattern set over the period t_2 and t_3?
  - What are the periods when (a,b) is frequent?
  - Does the support of (a) change dramatically in the period from t_3 to t_0?

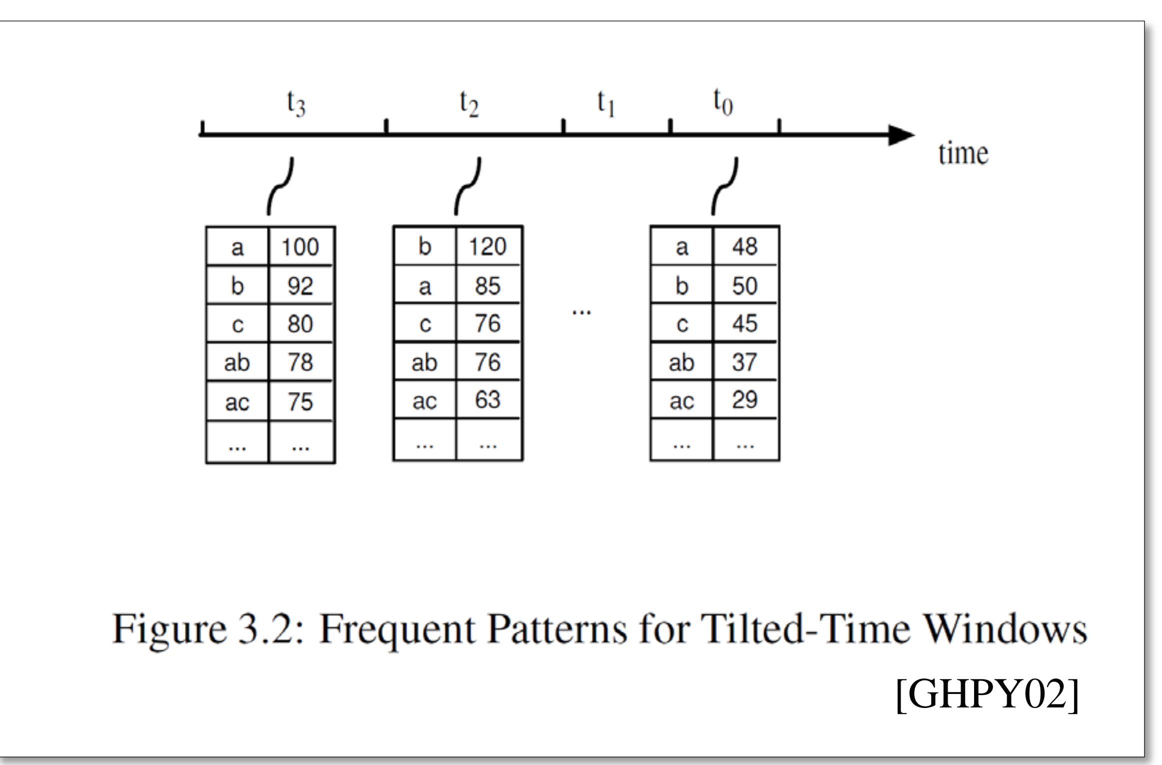


Figure 3.2: Frequent Patterns for Tilted-Time Windows

[GHPY02]

Itemset representation: pattern tree (similar to FP-Tree)

frequent pattern	support
a	100
b	92
c	80
ab	78
ac	75
bc	70
abc	63

Frequent Patterns

⇨

a: 100
b: 92
c :80
b: 78
c :75
c :70
c :63

Pattern Tree

Figure 3.3: Pattern Tree

[GHPY02]

Extend pattern tree for tilted-time windows

a
b
c
c
d
...

Pattern Tree

ac

tilt window	support
t_3	75
t_2	63
t_1	32
t_0	29

Tilted-time Window Table

Figure 3.4: Pattern-Tree with Tilted-Time Windows Embedded in FP-stream

[GHPY02]

Logarithmic Tilted-time Frame Windows [GHPY02]

- Current quarter, next two quarters, next four quarters, …

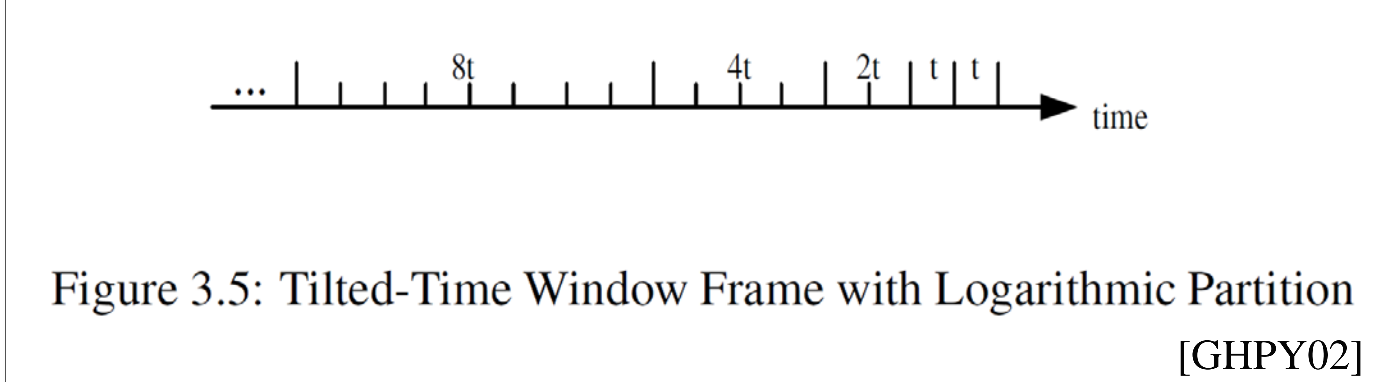


Figure 3.5: Tilted-Time Window Frame with Logarithmic Partition

[GHPY02]

- Set of transactions broken into fixed-sized batches $\boldsymbol{B_1}, \boldsymbol{B_2}, \dots, \boldsymbol{B_n}$
  - B_n is the most current batch, B_1 the oldest
- $B(i,j) = \bigcup_{k=j}^{i} B_k \ , i \geq j$ is a set of batches between i and j
- For a given itemset I: $f_I(i,j)$ is the frequency of I in $B(i,j)$ (number of times I occurs in B)
- In a logarithmic tilted-time window, the following frequencies are kept:

$$f(n,n); f(n-1,n-1); f(n-2,n-3); f(n-4,n-7); \dots$$

- Growth rate of window size:
  - Ratio r between the size of two neighbor tilted-time windows

FP-Stream Algorithm

- Giannella et al. [GHPY02]

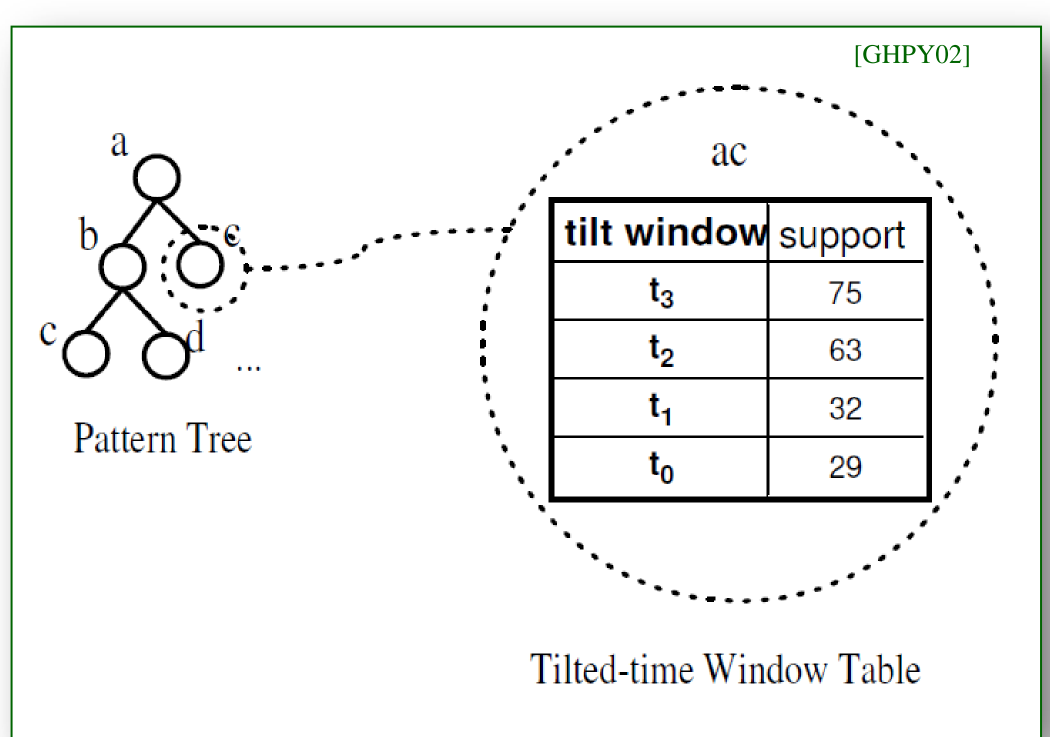
FP-stream algorithm (init)

- Constructs and maintains the FP-stream data structure
  - Pattern tree with tilted time window information
- Is bulky: updates only when enough incoming transaction have arrived to form a new batch B_n

- First batch B_1 is used for initialization:
  - compute frequencies for all items
  - store transactions in main memory
  - create ordered list *f_list* with items in decreasing frequency (as for the FP-tree) – remains fixed for all remaining batches!
  - all transactions from B_1 are used to create an FP-tree
    - prune all items with frequency less than $\varepsilon|B_1|$ (error rate * size of B_1)

FP-stream algorithm (signature of update)

- Input:
  - an FP-stream structure
  - a min_support threshold σ
  - an error rate ε
  - a new incoming batch B_i
- Output:
  - the updated FP-stream structure



tilt window	support
t_3	75
t_2	63
t_1	32
t_0	29

FP-stream algorithm (update)

1. Initialize the FP-tree to empty
2. Sort each incoming transaction t according to *f_list* and insert it into the FP-tree without pruning
3. When all $t \in B_i$ are accumulated, update the FP-stream structure:
   a) Mine itemsets out of the FP-tree.
      For all Itemset I:
      if I is in the FP-stream structure:
      i. Add $f_I(B_i)$ to the tilted-time window table for I
      ii. Conduct tail pruning
      iii. If the table is empty, stop FP-growth (Type II Pruning); else continue with supersets of I

      else:

      if $f_I(B_i) \geq \varepsilon|B_i|$: insert I into structure
      else stop mining supersets of I (Type I Pruning)

FP-stream algorithm (update) (cont).

b) Scan the FP-stream structure (depth-first search)
For each itemset I:

i. if I was not updated in mining B_i: insert 0 into I's tilted-time window table (because I did not occur in B_i)
Prune I's table by tail pruning.

If I is a leaf that has an empty tilted-time window table: drop the leaf

If there are siblings:

continue with siblings

else:

return to the parent and continue with its siblings

Frequent itemset mining in Odysseus

- As datastream operator

Incremental Learning on Data Streams

- Update model with every new element
- Main approaches:
  - Aggregation / synopses / histograms
    - aggregate learned knowledge
    - Pro: can span longer time periods
    - Con: does not deal well with concept drift, accuracy drops over time
  - Window-based
    - e.g.: clustering over the last 10 minutes
    - Pro: robust with respect to concept drifts
    - Con: re-learns with every window, forgets old knowledge
  - Hybrid approaches / tilted time frames:
    - Adapt to time: younger data → more details, older data → less data

Introduction

- Approach by Dennis Geesen [Gees13], implemented in Odysseus
- Idea: extend DSMS to cope with any "traditional" machine learning approach
- Logical data stream:

$$S^l := \{(e, t, n) | e \in \Omega_{\mathcal{A}} \wedge t \in T \wedge n \in \mathbb{N} \wedge n > 0\}$$

  - Logical data stream S^l is a potential infinite multi set of tuples
  - e is the tuple, t the time stamp, n the number of occurences of tuple at time t
  - logical view enables snapshot reducability
- Physical data stream:

$$S^p := \{(e, [t_s, t_e)) | e \in \Omega_{\mathcal{A}} \wedge t_s, t_e \in T \wedge t_s \leq t_e\}$$

  - Physical data stream (implementation) based on time intervals
  - Physical view enables efficient implementation

[Gees13] Geesen, Dennis: Maschinelles Lernen in Datenstrommanagementsystemen. Auflage: 1., Auflage. Aufl. Edewecht : OlWIR Verlag für Wirtschaft, Informatik und Recht, 2013 — ISBN 9783955990015

FREQUENTITEMSET operator in Odysseus

- Operator that creates frequent item sets from a given stream.
- The result stream creates a tuple with 3 attributes:
  - id: the number (a simple counter) of the pattern
  - set: the frequent pattern, which is a list of tuples (a nested attribute ~ NF^2)
  - support: the support of the pattern
- **Parameter**
  - SUPPORT: The minimal support that defines what is frequent. This can be either a total number > 1.0 or a double between 0.0 and 1.0. The double indicates the percent in terms of the number of transactions.
  - TRANSACTIONS: A number of transactions that should be investigated
    - A transaction is a snap-shot of a window:
      Each time when a window changes, there is a new transaction
  - LEARNER: the algorithm that is used
    Currently implemented: **fpgrowth**, Weka (which in turn has further algorithms)
  - ALGORITHM: A set of options to describe the algorithm

Snapshot reducability

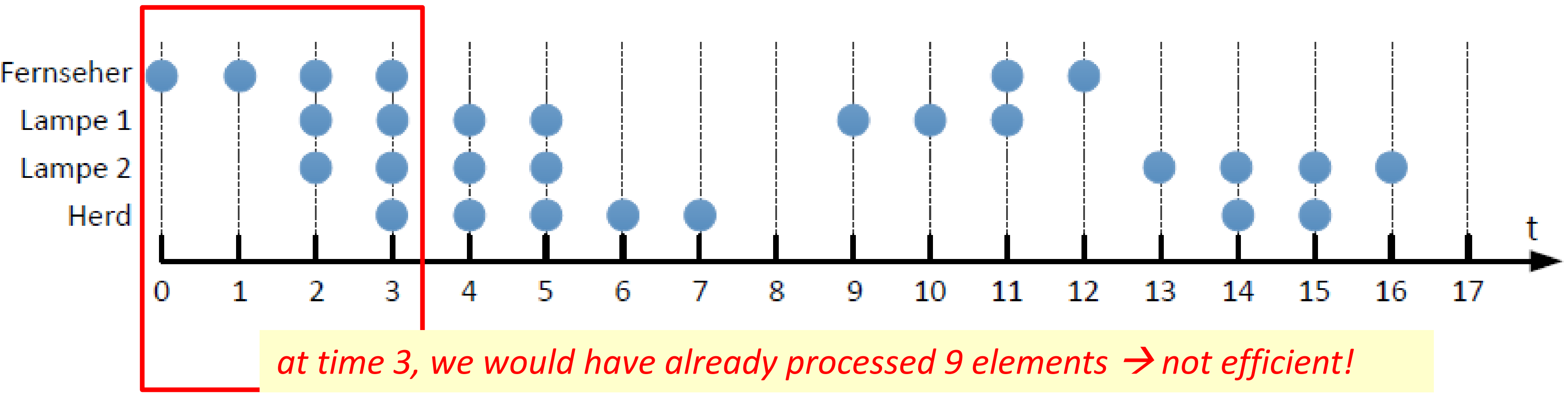
Logical data stream (validity of elements):

Fernseher
Lampe 1
Lampe 2
Herd
t
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

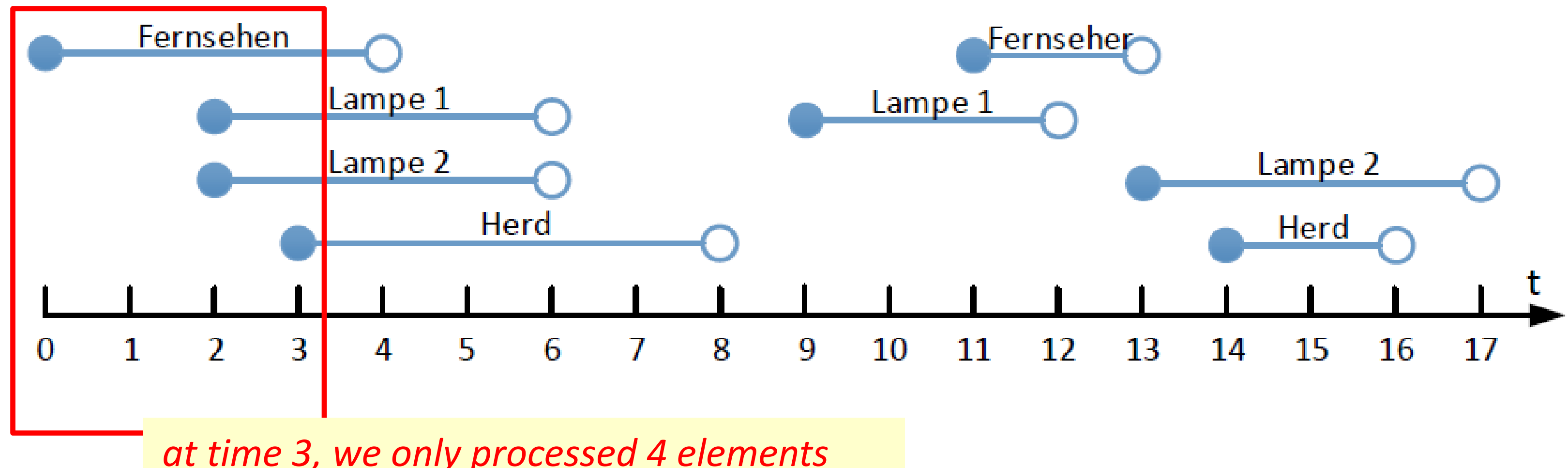
Each time stamp can be viewed a static data source with the current contents

Examples for logical and physical data streams

Logical data stream (validity of elements):



Physical data stream (implementation: tuples with validity intervals):



Example

/// support is 3 out of 1000 transactions

fpm = FREQUENTITEMSET({support=3.0, transactions=1000, learner = 'fpgrowth'}, inputoperator)

/// support is 60% out of 1000 transactions, so it is equal to a support of 600.0

fpm = FREQUENTITEMSET({support=0.6, transactions=1000, learner = 'fpgrowth'}, inputoperator)

Example (from [Gees13])

Input tuples

0	$\{(Fernseher)\}$
1	$\{(Fernseher)\}$
2	$\{(Fernseher), (Lampe\ 1), (Lampe\ 2)\}$
3	$\{(Fernseher), (Lampe\ 1), (Lampe\ 2), (Herd)\}$
4	$\{(Lampe\ 1), (Lampe\ 2), (Herd)\}$
5	$\{(Lampe\ 1), (Lampe\ 2), (Herd)\}$
6	$\{(Herd)\}$
7	$\{(Herd)\}$
8	$\{\}$
9	$\{(Lampe\ 1)\}$
10	$\{(Lampe\ 1)\}$
11	$\{(Fernseher), (Lampe\ 1)\}$
12	$\{(Fernseher)\}$
13	$\{(Lampe\ 2)\}$
14	$\{(Lampe\ 2), (Herd)\}$
15	$\{(Lampe\ 2), (Herd)\}$
16	$\{(Lampe\ 2)\}$
17	$\{\}$

Candidates for window size=5 and t=6

{Fernseher}	2
{Herd}	4
{Lampe 1}	4
{Lampe 2}	4
{Fernseher, Herd}	1
{Fernseher, Lampe 1}	2
{Fernseher, Lampe 2}	2
{Herd, Lampe 1}	3
{Herd, Lampe 2}	3
{Lampe 1, Lampe 2}	4
{Fernseher, Herd, Lampe 1}	1
{Fernseher, Herd, Lampe 2}	1
{Fernseher, Lampe 1, Lampe 2}	2
{Herd, Lampe 1, Lampe 2}	3
{Fernseher, Herd, Lampe 1, Lampe 2}	1

Example (from [Gees13])

Input tuples

0	{(Fernseher)}
1	{(Fernseher)}
2	{(Fernseher), (Lampe 1), (Lampe 2)}
3	{(Fernseher), (Lampe 1), (Lampe 2), (Herd)}
4	{(Lampe 1), (Lampe 2), (Herd)}
5	{(Lampe 1), (Lampe 2), (Herd)}
6	{(Herd)}
7	{(Herd)}
8	{}
9	{(Lampe 1)}
10	{(Lampe 1)}
11	{(Fernseher), (Lampe 1)}
12	{(Fernseher)}
13	{(Lampe 2)}
14	{(Lampe 2), (Herd)}
15	{(Lampe 2), (Herd)}
16	{(Lampe 2)}
17	{}

Result for window size=5 and t=6 and min_support = 3

{Herd}	4
{Lampe 1}	4
{Lampe 2}	4
{Herd, Lampe 1}	3
{Herd, Lampe 2}	3
{Lampe 1, Lampe 2}	4
{Herd, Lampe 1, Lampe 2}	3

Example (from [Gees13])

Input tuples

0	$\{(Fernseher)\}$
1	$\{(Fernseher)\}$
2	$\{(Fernseher), (Lampe\ 1), (Lampe\ 2)\}$
3	$\{(Fernseher), (Lampe\ 1), (Lampe\ 2), (Herd)\}$
4	$\{(Lampe\ 1), (Lampe\ 2), (Herd)\}$
5	$\{(Lampe\ 1), (Lampe\ 2), (Herd)\}$
6	$\{(Herd)\}$
7	$\{(Herd)\}$
8	$\{\}$
9	$\{(Lampe\ 1)\}$
10	$\{(Lampe\ 1)\}$
11	$\{(Fernseher), (Lampe\ 1)\}$
12	$\{(Fernseher)\}$
13	$\{(Lampe\ 2)\}$
14	$\{(Lampe\ 2), (Herd)\}$
15	$\{(Lampe\ 2), (Herd)\}$
16	$\{(Lampe\ 2)\}$
17	$\{\}$

Result for whole stream (result pattern in NF²⁾

0	{ }
1	{ }
2	{(Fernseher)}
3	{(Fernseher)}
4	{(Fernseher), (Lampe 1), (Lampe 2), (Lampe 1, Lampe 2)}
5	{(Fernseher), (Lampe 1), (Lampe 2), (Herd), (Herd, Lampe 1) (Herd, Lampe 2), (Lampe 1, Lampe 2), (Lampe 1, Lampe 2, Herd)}
6	{(Lampe 1), (Lampe 2), (Herd), (Herd, Lampe 1) (Herd, Lampe 2), (Lampe 1, Lampe 2), (Lampe 1, Lampe 2, Herd)}
7	{(Lampe 1), (Lampe 2), (Herd), (Herd, Lampe 1) (Herd, Lampe 2), (Lampe 1, Lampe 2), (Lampe 1, Lampe 2, Herd)}
8	{(Herd)}
9	{(Herd)}
10	{ }
11	{(Lampe 1)}
12	{(Lampe 1)}
13	{(Lampe 1)}
14	{ }
15	{(Lampe 2)}
16	{(Lampe 2)}
17	{(Lampe 2)}

Stream mining – summary

Data stream management

- Introduction 1

Data Mining / Machine Learning

- Introduction 2

Stream mining

- Why is it difficult?
- How can it be done?
- Example (Frequent Pattern Mining)

Stream mining

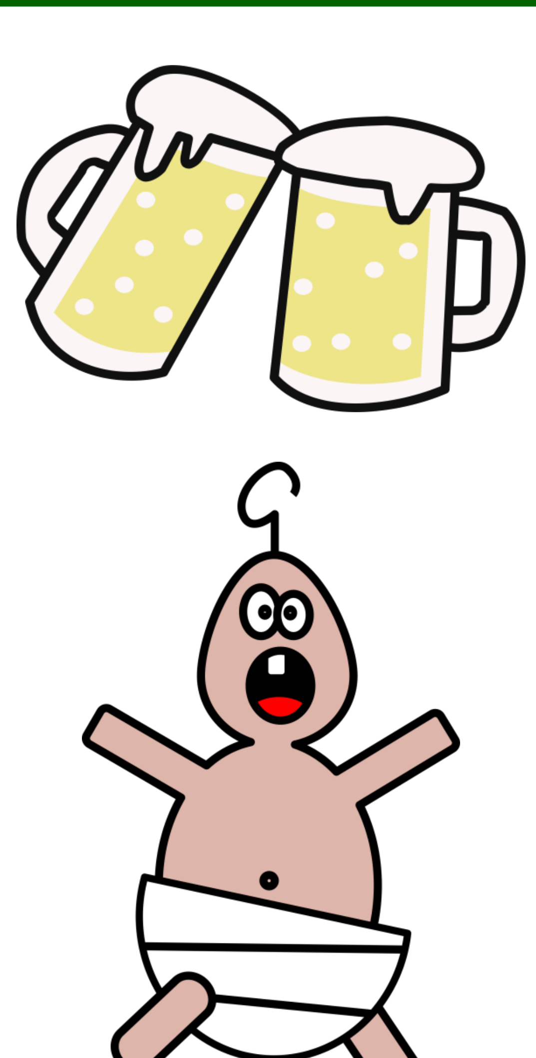
- Why is it difficult?
  - Unbounded data
  - Bounded memory
  - Concept drift

- How can it be done?
  - Invent new algorithms that aggregate over time
  - Select instances (windows) and perform existing algorithms

- Example
  Frequent Pattern Mining

Frequent Itemset Stream Mining – Summary

- Frequent itemset mining by Jin et al. [JiAg07]
  - Landmark window (from beginning)
  - Aggregations
- FP-stream by Giannella [GHPY02]
  - Tilted-time windows
  - FP-tree
- Frequent pattern mining in Odysseus
  - Sliding window!
  - Different learners
    - Can use any non-streaming algorithm
    - Integrates WEKA framework



Literature

[Gees13] Geesen, Dennis: Maschinelles Lernen in Datenstrommanagementsystemen. Auflage: 1., Auflage. Aufl. Edewecht : OlWIR Verlag für Wirtschaft, Informatik und Recht, 2013 — ISBN 9783955990015

[Agga07] AGGARWAL, C. C. (Hrsg.): *Data Streams - Models and Algorithms*, *Advances in Database Systems*. Bd. 31 : Springer, 2007 — ISBN 978-0-387-28759-1

[JiAg07] Jin, Ruoming ; Agrawal, Gagan: Frequent Pattern Mining in Data Streams. In: Aggarwal, C. C. (Hrsg.): Data Streams, Advances in Database Systems : Springer US, 2007 — ISBN 978-0-387-28759-1, 978-0-387-47534-9, S. 61–84

[ChLe03] Chang, Joong Hyuk ; Lee, Won Suk: Finding Recent Frequent Itemsets Adaptively over Online Data Streams. In: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD ’03. New York, NY, USA : ACM, 2003 — ISBN 1-58113-737-0, S. 487–492

[GHPY02] Giannella, Chris ; Han, Jiawei ; Pei, Jian ; Yan, Xifeng ; Yu, Philip S.: Mining Frequent Patterns in Data Streams at Multiple Time Granularities, 2002

[KaSP03] Karp, Richard M. ; Shenker, Scott ; Papadimitriou, Christos H.: A Simple Algorithm for Finding Frequent Elements in Streams and Bags. In: ACM Trans. Database Syst. Bd. 28 (2003), Nr. 1, S. 51–55