28 Mining Data Streams

by **Kevin Cheung** and **Patrick Boily**

In pedagogical settings, we typically treat the data as "complete" in the sense that we assume that the data collection process is over – no new data will be added to our trove, and all of the past (available) data is used in the analyses. In a nutshell, data collection lies in the past, and data analysis lies in the future.

In practice, data is often collected **continuously** (or at least, in **batches**), and older data typically become **stale** and should perhaps not be weighed as heavily as more recent data when the analyses are **updated**.*

In complex analytical scenarios, however, it can be inefficient to replicate multiple analyses with sensibly the same data from one time point to the next. In this chapter, we discuss **data streams** and how to extract insights in such situations. More information is available in [7, 15].

28.1 Overview

We start by introducing the fundamental notions of the discipline.

28.1.1 Motivating Examples

Consider the owner of a gift shop at the Ottawa Macdonald-Cartier International Airport (YOW). She would like to keep track of the most popular items sold on each day – perhaps there are a few "hot items" that show up daily, and some outliers that only show up in the list sporadically. How could she approach this task?

Assuming that she uses some sort of electronic inventory system, she can export the daily transactions to a file for data analysis. As YOW is not a very busy airport, she might expect the file to contain only a small number of sales. Let's take a look at the data for one day.

```
df <- read.csv("dailysales.csv", stringsAsFactors = TRUE)
str(df)
df
df2 <- df['sku']  # Take just the sku column
ag <- aggregate(df2, by=list(item = df2$sku), FUN=length)
ag[ order(-ag$sku), ]  # Sort in descending order
```

* When does Netflix recognize that a user's taste in comedy have changed, say?

```
'data.frame': 20 obs. of  3 variables:
 $ sku   : Factor w/ 4 levels "A","B","C","D": 3 2 3 1 3 4 3 2 2 3 \ldots
 $ time  : Factor w/ 20 levels "08:30","08:35",..: 1 2 3 4 5 6 7 8 9 10 \ldots
 $ amount: num  9.99 4.49 9.99 12.99 9.99 \ldots
```

```
   sku  time amount
1    C 08:30   9.99
2    B 08:35   4.49
3    C 08:40   9.99
4    A 08:50  12.99
5    C 09:10   9.99
6    D 10:20  24.99
7    C 10:24   9.99
8    B 11:05   4.49
9    B 13:46   4.49
10   C 13:57   9.99
11   C 14:07   9.99
12   B 14:36   4.49
13   C 14:48   9.99
14   B 14:50   4.49
15   C 15:08   9.99
16   B 15:46   4.49
17   D 15:47  24.99
18   D 15:52  24.99
19   A 16:10  12.99
20   A 16:17  12.99
```

```
  item sku
3    C   8
2    B   6
1    A   3
4    D   3
```

But if instead of owning a small gift store at a medium-sized airport, she was the product manager of Amazon USA. She might want to see the top sold items since the beginning of the day every 10 minutes. But the solution for the YOW gift shop would be unlikely to work. Why? Well, in 2016,[1] Amazon USA had 500+ million items in their catalogue, and the site was conducting, on average, 200,000+ transactions per hour. This quickly becomes a **big data** problem.[2]

1: And it has only increased since.

2: See Chapter 30 for additional details.

28.1.2 Basic Notions

In the gift shop example, all the daily transactions can be processed in one go at the end of the day – this is an example of **batch processing**. In the Amazon USA example, the transactions are not available all at once for processing and results need to be updated in **time increments**; whether the volume of transactions is high or not, the need for **(near) real-time updates** is a characteristic of online processing.

Data stream mining algorithms are invariably **online algorithms**,[3] some of which might exploit **intermittent batch processing** over time windows. We will be looking at examples throughout this chapter.

3: "In computer science, an **online algorithm** is one that can process its input piece-by-piece in a serial fashion, i.e., in the order that the input is fed to the algorithm, without having the entire input available from the start. In contrast, an **offline algorithm** is given the whole problem data from the beginning and is required to output an answer which solves the problem at hand." [31]

Big data is typically characterized by the **5V framework**:

- we are dealing with large amounts of data (**volume**);
- with observations of different types which may not all be saved in relational databases (**variety**);
- created at high speed, but with accessibility and processing bottlenecks (**velocity**);
- whose quality and accuracy are harder to control (**veracity**), and
- which we hope to be able to turn into actionable, insightful results (**value**).

To react to information in **real-time**, the most critical component to tame is **velocity** [13]. High-velocity data streams arise, among others, in:

- communication channels;
- sensors;
- financial or sales transactions, and
- user-generated content.

Important applications (and fields of applications) include:

- autonomous vehicles;
- cybersecurity;
- fraud detection;
- health care;
- smart homes;
- mobile communication, etc.

Data Stream Models We can study data streams *via* formal mathematical models, such as an ordered pair (s, Δ) where s is a sequence of **data objects** and Δ is a sequence of **time intervals**.

Loosely speaking, a **data stream** is a sequence of data objects (usually a vector of numerical values) that arrive **online**, one at a time; we have **no control** over the **order** in which the data elements arrive. Data streams are potentially infinite in length. Access to the data objects is thus **sequential**; it is also **one-time**, meaning that once a data object has been processed, it is **archived** or **discarded**.[4]

Another important practical issue is the presence of **noise** in the data: thermostat readings from (cheap) sensors can fluctuate wildly even when the ambient temperature is constant, for instance. When real-time responses are expected (such as may be the case in autonomous vehicles, say), noisy data has to be carefully handled: **smoothing techniques** such as Kalman filters might be required.[5]

In practice, analysts use **data management systems** (DMS) to extract insight from data streams:

1. a **stream processor** (such as Apache Kafka) to manage input and output streams;
2. a **working storage** (limited in size) to hold recent/current data elements;
3. an **archival storage** (such as Apache Hadoop) to store past data elements;
4. a **query processor**, and
5. an **analytics API**.

4: Although a small number of data objects can be held in memory for debugging or validation purposes.

5: We will not, however be covering smoothing techniques in detail; see Chapter 9 (DUDADS, Volume 1) and [7, 15] for more information.

Note that we will not be using a DMS in this chapter's example.

Because of the velocity and volume of data in a typical data stream, exact analysis methods are often contra-indicated; instead, we use **probabilistic** or **approximation** algorithms.

Filtering Say we run a domain name (DNS) server for Internet protocol (IP) address lookup; we might need to filter millions of URLs for malicious websites. It becomes quite impractical to look up the database for every DNS query. A simple, well-known, and practical approach for this task is to employ a **Bloom filter**.

A probabilistic **memory-efficient** procedure that allows for **fast querying**, a Bloom filter is primarily used to **test whether an element** $s \in T$ **is also a member of a set** S; Bloom filters can result in **false positives** (falsely indicating that an element is in the set when it actually isn't), but they cannot produce **false negatives** – if a query indicates an element is not in the set, then it definitely is not.

Bloom filters are particularly useful in situations where the set of keys is large and expensive to enumerate; they are often used in situations where the **speed** and **efficiency** of the query process are more critical than the absolute accuracy of the result.

Bloom filters use **hash functions** $h_1, \ldots, h_k : T \to \{0, \ldots, m-1\}$ to map elements $s \in T$ to positions in an array F of m bits, where $m \gg 1$.[6] For each **known** $s \in S$, we compute the k hashes $h_1(s), \ldots, h_k(s) \in \{0, \ldots, m-1\}$; these values become the indices of F for which the bit entry is 1.

6: Hash functions are **deterministic**, **efficient** (the hash value can be computed quickly), **non-invertible** (it should be difficult to reverse-engineer the original input from the hash value), and **collision-resistant** (it should be improbable for two different inputs to produce the same output).

For instance, say $m = 12$ and there are $k = 3$ hash functions, with the set S of malicious IP containing only 4 elements.

URL	h_1	h_2	h_3
bad.com	0	6	4
malware.org	1	4	7
virus.ca	1	2	10
breach.biz	0	6	10

The set of hash values on all known malicious URLs is thus

$$\{h_1(s_1), \ldots, h_3(s_4)\} = \{0, 1, 2, 4, 6, 7, 10\}$$

and the array F (we count from index 0) is:

1	1	1	0	1	0	1	1	0	0	1	0

Consider an element $t \in T$. If any of the bits of F indexed by the hash values $h_1(t), \ldots, h_k(t)$ is 0, then $t \notin S$; otherwise, we "predict" $t \in S$ (but this could be a false positive).

For instance, say that a new URL t is received for DNS lookup; we compute $h_1(t) = 1$, $h_2(t) = 3$ and $h_3(t) = 7$:

1	1	1	0	1	0	1	1	0	0	1	0

As $F(3) = 0$, we would conclude that t is not malicious.

The size m of the Bloom filter and the number k of hash functions used affect its efficiency and the probability of obtaining false positives; a larger filter size generally results in a lower error rate, but the likelihood of false positives increases as more elements are tested. For more information, refer to [16, 22, 28].

Sampling Sampling from a data stream differs significantly from sampling from **static storage**, mostly with respect to **data accessibility** and **methods**.[7]

7: See Chapter 10 (DUDADS, Volume 1).

In static contexts,

- the data is **stable** and the dataset size is **known**, which simplifies sample size and estimator variance computations;
- the same dataset can be sampled **multiple times**, using traditional and straightforward sampling methods, and
- there is typically **no urgency** to analyze the data, as it does not change in real-time.

In streaming contexts,

- the data is **transient**, flowing in (near) real-time, and is not typically stored long-term (unless it is captured/archived during the stream);
- items often have a single chance to be sampled because they may not be seen again, necessitating **one-pass algorithms**;
- the sample may need to be **continuously updated** to reflect the most recent data;
- the total size of the data is not known in advance, which affects **sample size considerations**, and
- typically, the data must be analyzed **immediately**, since storing/archiving large amounts of streaming data may prove impractical or impossible.

Different sampling methods exist depending on the application. We illustrate two common sampling methods for data streams: **reservoir sampling** and **sampling *via* a hash function**.

Consider a credit card company, say, that wants to obtain a sample of 1000 transactions over the next hour drawn uniformly at random. But the company doesn't know how many transactions will occur during the hour, they cannot save too many transactions in memory, and accessing the archive of saved transactions may be too costly. How could they approach the task?

Assuming that the number of transactions is sufficiently larger than the **reservoir capacity** $\Lambda(= 1000)$, **reservoir sampling** can be shown mathematically to produce a uniform sample of all transactions. Start by preparing a **reservoir storage** R for $\Lambda(= 1000)$ transactions, and insert the first Λ transactions in R.

For each transaction t that follows in the specific time frame:

1. draw a random integer M in $\{1, \ldots, t\}$;
2. if $M \leq \Lambda$, randomly delete a transaction from R and replace it with the transaction t.

The transactions in R after the final transaction form the **reservoir sample**.

If instead we want to sample $r = 1\%$, say, of the credit card users and compute each sampled user's total spending over a specific time frame, we could use the following approach. Assume that we have a reasonably accurate estimate $\hat{N}$ of the total number of users who will conduct a credit card transaction during the time-frame.

As a transaction arrives, we check if the user is already in the sample:

- if they are, we simply add the transaction amount to the running total for that user;
- if they are not, then with probability r, we add the user (and the transaction value) to the sample.

The sampling procedure stops once there are more than $r\hat{N}$ users in the sample.

This method has a huge drawback, however: checking if a transaction's user matches one in the sample could **take too much time** relative to the rate at which the transactions arrive – data base lookup needs to be avoided as much as possible.

Instead, let h be a (quickly computable) **hash function** mapping credit card users to an integer in the range $\{0, \ldots, \hat{N} - 1\}$ and set up an array A of size $r\hat{N}$ to track the sampled users' total spendings.

When a transaction (u, a) arrives, we compute $h(u)$, where u is the user. If $h(u) < r\hat{N}$, we add the transaction amount a to the element of A indexed by $h(u)$. Otherwise, we leave A unchanged.

For example, say we expect $\hat{N} = 1000$ users to conduct transactions in the specified time frame. Then the array A is of size $r\hat{N} = 0.01(1000) = 10$ and only the spending amounts of users hashed to a value in the range $\{0, \ldots, 9\}$ would be tracked.

Time Windows In theory, data streams are **potentially infinite**.[8] In practice, we often use a **window** (i.e., a contiguous subset of the data) on the stream for analysis.

The most commonly time windows are:

1. **landmark** windows, which consider data from the beginning of the stream to the present;
2. **sliding** windows, which consider data from w time units before the present to the present;
3. **fading** windows, which assign smaller weights to older data, and
4. **tilted time** windows, which use different time granularities.[9]

Instead of figuring out the most popular items each hour (as in Section 28.1.1, for instance), we could monitor the situation **continuously** by assigning scores to items in such a way that the score of an item decreases over time if it is not purchased again. The popularity of an item can then be simply measured as the total score of the item, with popular items having the highest scores. We illustrate how a **fading window** could be used to accomplish the task.

Let c be a very small number, such as 0.0001 or $1/N$, where N is the number of items in the catalogue. Assume that there is a list of items

8: Data streams must necessarily be finite, but their size could be so large that it makes more sense to model them as infinite.

9: The fundamental idea is to create a hierarchy of time windows of varying sizes that reflect the different ages of the data. In more recent time periods, the windows are smaller, capturing data at a fine granularity, which allows for detailed analysis of the most recent events. As time progresses, the windows become larger, encapsulating older data at a coarser granularity. This approach allows for efficient storage and query of data streams, retaining more detail for recent data while still maintaining a summary of older data.

whose popularity we are tracking, each of which is assigned the initial score of 0. When a purchase takes place:

1. multiply each list item's score by $1 - c$;
2. if the purchased item is in the list of items, add 1 to its score;
3. for any item whose score falls below a threshold t, set the score to 0.

For instance, suppose that $c = 0.2$ and $t = 0.5$, and that the current list stands at:

Item	Eggs	Milk	Jam	Bread
Score	0.64	0.8	1	0

If the next purchase in the specified time frame is bread, then the list is updated to:

Item	Eggs	Milk	Jam	Bread
Score	0.512	0.64	0.8	1

If the next purchase in the specified time frame is milk, then the list becomes:

Item	Eggs	Milk	Jam	Bread
Score	0.4096	1.512	0.64	0.8

As the eggs' score falls below $t = 0.5$, it is set to 0.

Item	Eggs	Milk	Jam	Bread
Score	0	1.512	0.64	0.8

If this was the end of the specified period, the most popular purchased item would be milk, followed by bread, and then jam. Note that a score of 0 does not necessarily mean that the item in question has not been purchased (as was the case with eggs).

This algorithm is such that the sum of all scores prior to a purchase is at most $1/c$, and that there cannot be more than $2/c$ items in the list with a score above 0.

Learning and Validation Since the data objects in a data stream are unavailable for analysis or machine learning **all at once**, we need different approaches than those suggested in Chapters 6-11 ([8, vol. 1]) and Chapters 19-23 ([8, vol. 3]); instead, to get around this limitation, we typically use:

- **incremental learning**, in which the model is updated as new data arrive, or
- **two-phase learning**, where we periodically conduct (offline) learning on a synopsis/subset/summary of the data, and then update the model appropriately.

For similar reasons, the traditional model validation approaches (such as $k-$fold and leave-one-out validation methods) do not apply to data streams; instead, we may use **hold-out** or **prequential** validation.

In **hold-out validation**, we set aside a portion of the incoming data as a test set and use the the remainder to train the model. Since data streams are **infinite** (in theory) and **non-stationary**;[10] as such, the model may need to **update continuously**.

10: The underlying distribution of the data changes over time (see [8, vol. 1, ch. 9]).

The hold-out method is often used as a foundation for experimental frameworks in data stream analysis because it allows for **flexibility** in capturing various statistics of interest and can be adapted to track many behaviours of the model under evaluation.[11]

11: For instance, it can be set up to monitor the model's accuracy against the number of training instances processed or the actual training time, which can then be represented in **learning curves** or **performance tables**.

In practice, the hold-out method measures the **immediate accuracy** of the model at specific points in the stream (sampled according to some appropriate model) without considering the history of previous performance, which can be problematic since early poor performance can affect the accuracy measurement, even if the algorithm improves significantly over time.

But with a "**large enough**" hold-out test, we can hope to improve the reliability of the accuracy estimates; averaging the evaluation procedure multiple times over different test sets may also help in reducing the variance in the estimates.

Data mining analysts also test the performance of models on historical data. However, in real-life applications, what works well on historical data might not work well in production. When **concept/model drift** is likely to be encountered,[12] **prequential** validation (also known as the **interleaved test-then-train method**), which attempt to maintain a summary of recent samples while discarding older ones to better reflect the current distribution of the data stream, might provide a better framework for model validation.

12: That is, when the trained model might have been great for historical data, but is unlikely to be useful for new observations due to a shift in the data distribution or the way the various features interact in the data over time.

Prequential validation works as follows:

1. as each instance from the data stream arrives, it is first used to **test** the model, meaning the current model's prediction for this instance is evaluated before it is incorporated in the training set;
2. after the prediction and its evaluation, the same instance is used to further train or update the model, reflecting a real-world scenario where models must predict future or unseen data;
3. the performance of the model is tracked continuously over time, after each each prediction, using any relevant evaluation metrics.

Some variations of the prequential method involve using a **sliding window** or **fading factors** to give more weight to recent data, helping the model adapt quickly to **concept drifts** by "forgetting" older data.[13]

13: Even without these variations, the prequential approach allows the model to be constantly updated with the most recent data, which also protects against model drift.

One major advantage of using the prequential method is that it simulates a scenario where each instance can only be used once, due to the potential infinite length and high-velocity nature of streams. It **mimics the way models are used in the real world**, continuously learning and adapting from a stream of incoming data and may provide a realistic assessment of a model's predictive performance and its ability to adapt over time.

28.2 Change Detection and Maintaining Statistics

There are many phenomena/situations that exhibit **statistical** change over time:

- the demographics of a neighbourhood;
- the average room temperature in a house;
- weekly movie theatre attendance;
- the vital signs of an individual.

For each of these examples, we could expect the **summary statistics** to remain relatively stable over a "small enough" time window. However, it is clear that there are events that will cause noticeable changes in the summary statistics over a long time period.

For example, in the case of vital signs, illnesses ranging from a flu to a heart attack can trigger changes that vary from mild to wild (including the sudden disappearance of any sign).

Time Series Analysis vs. Data Streams Mining In [8, vol. 1, ch. 9], we would have tackled such situations using **time series analysis** – how does data streams mining differ? Broadly-speaking, the focus of time series analysis is on **understanding historical data** and **predicting future values** in a fixed dataset, while data streams mining is concerned with handling and extracting knowledge from an **ongoing flow** of data that may change **unpredictably** over time.[14]

14: In particular, **time series analysis** involves **ordered** sequences of **stationary** values typically measured at **successive points** in time spaced at **uniform time intervals**; the analytical aims are to understand or model the **underlying structure** and **behaviour** of the data, often to forecast future points in the series. Time series analysis uses methods that exploit **temporal correlations** (ARIMA, exponential smoothing, Fourier analysis, etc.); in general, the analysis is performed on a **complete** dataset, and it is usually **retrospective**.

In contrast, **data streams mining** deals with large volumes of (possibly **non-stationary**) data that arrive **continuously** over time, often at **high velocity**, in a potentially **never-ending process**; it aims to extract **actionable insights** from **live data** as it is generated. Data streams mining uses algorithms that process and summarize data **on the fly** (online learning, sliding windows, and approximation algorithms); in general, mining is performed in **real-time** and is often **prospective**, focusing on immediate insights and rapid decision-making.

28.2.1 Change Detection

In this section, we will take a look at **change detection** (or **concept drift detection**) over data streams. The main challenge in change detection is determining **when** a significant change occurs and modifying the data stream model appropriately to reflect the change – part of the difficulty is that **noise** may wrongly be seen as change by the untrained eye.

Change detection then needs to balance **sensitivity to change** and **robustness to noise**.

Technical Framework A data stream is typically modeled as a sequence of **data-generating distributions** from which **instances** (data objects/items) are drawn.

We denote by $\mathcal{D}_i$ the distribution for the $i-$th item in the data stream; the actual item generated from $\mathcal{D}_i$ is denoted by x_i – in other words, x_i is obtained by **sampling from** $\mathcal{D}_i$. Note that x_i could be a vector, and not necessarily just a scalar.[15]

For instance, if the distributions $\mathcal{D}_i$ are independent $\mathcal{N}(0,1)$ for $i \leq 6$, while the (remaining) distributions $\mathcal{D}_i$ are independent $\mathcal{U}(0,1)$ for $i \geq 7$, then a **realization** of the data stream could be

$$0.56, -2.43, 0.05, 1.78, -0.82, -0.21, 0.2, 0.9, 0.07, 0.54, 0.49, \ldots$$

15: In other chapters, we differentiated the scalar x from the vector $\mathbf{x}$ by using a bold font. In this chapter, we will stick to the regular font weight – if a vector is intended, the context should make it clear.

We would like to be able to detect that the initial portion ($i \leq 6$) and the tail ($i \geq 7$) of this sequence arise from **different** distributions – in other words, that a change has occurred in the mechanism to generate the data between $i = 6$ and $i = 7$.

Learning Distributions In practice, the distributions $\mathcal{D}_i$ are unknown and need to be learned from the instances x_i. When the distributions change from one instance to another in an **haphazard manner**, which is an extreme case, learning is not possible.

The other extreme case is when the distributions **remain the same** throughout the data stream. In this case, once we have "enough" data instances, we can obtain a model whose accuracy is bounded below by the theoretical limit of the **Bayes error**.[16]

16: See Chapter 21 (DUDADS, Volume 3).

In either of these extremes, there is no need for change detection algorithm (although for different reasons).

Concepts and Terminology What typically happens in a data stream is that there are stretches of the sequence in which the distributions are (nearly) identical so that a model for the underlying distribution can be learned from the examples.

We can then partition the (nearly) identical distributions into sequences of distributions.[17]

17: Of course, this may be easier said than done...

In the following sequence of distributions, let's say that the ones with the same colour are (nearly) identical:

$$\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_4, \mathcal{D}_5, \mathcal{D}_6, \mathcal{D}_7, \mathcal{D}_8, \mathcal{D}_9, \mathcal{D}_{10}, \mathcal{D}_{11}, \mathcal{D}_{12}, \ldots$$

Denote the sequences of distributions by

$$\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \ldots$$

A model that is trained with training instances from $\mathcal{S}_1$ will perform poorly on $\mathcal{S}_2$. More generally, the distributions in $\mathcal{S}_j$ will have **different characteristics** from those of the distributions $\mathcal{S}_{j+1}$. We say that each $\mathcal{S}_j$ corresponds to a data stream **concept**; the transition from $\mathcal{S}_j$ to $\mathcal{S}_{j+1}$ is where a **concept drift** appears.

Of course, it might not be easy to distinguish a truce concept drift from the presence of **temporary noise**. In [17], the authors define concept drift in terms of **consistency** and **persistency**.

Loosely speaking, **consistency** means that the changes between consecutive instances in the same concept are small enough to stay under some pre-sepecified threshold.

Persistency, on the other hand, means that the concept is present for at least p consecutive instances where $p \geq w/2$ from some pre-specified window size w.

A concept drift is **permanent** if it is both consistent and persistent.[18]

18: Note that concepts may re-occur: in the concept sequence $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \ldots$, it is possible for the first and third concept to be identical, for instance.

Drift Detection In practice, there are two main approaches to drift detection:

- monitoring the evolution of **performance indicators**, and/or
- monitoring **distributions** on two (or more) different time-windows.

Common algorithms include: **floating rough approximations** (FLORA), **cumulative sum** (CUSUM), and the **Page-Hinkley test** (PH).

FLORA: the details of the algorithm are outside the scope of the chapter but can be found in [5, 7, 27]. Note, however, that FLORA:

- uses **"rough" set theory** (a generalization of regular or "crisp" sets from computer science);
- can adapt to **changing data patterns**, which is crucial for handling concept drift in streaming data;
- supports **incremental learning**, updating its model as new data comes in;
- is designed to identify **significant deviations** from existing models;
- is efficient and scalable, making it suitable for **big data streams**;
- is capable of handling **noise** and **uncertainty** effectively, and
- allows **user-defined parameters** to tailor detection sensitivity.

CUSUM is particularly effective in identifying shifts in the mean or variance. More details about the algorithm can be found in [5, 7, 14]. As an algorithm, CUSUM:

- is aimed at detecting **shifts in statistical properties** of a data stream (especially the mean);
- tracks the **cumulative sum of deviations** from a target or reference value;
- involves **threshold setting** for change detection – exceeding these thresholds indicates potential drift;
- implements a **two-sided approach** for detecting both positive and negative shifts;
- includes a **reset mechanism** after detecting a change, allowing for continuous monitoring;
- is **simple**, has **minimal computational requirement**, and is effective in detecting **gradual changes**, but
- can be sensitive to **parameter settings** and may require tuning.

PH focuses on identifying changes in the mean [7]:

- its primary objective is to detect **shifts in the mean** of a sequence of observations;
- it is based on a **cumulative sum approach**, but focuses on rapid detection of mean changes;
- it computes the **cumulative sum of differences** from the mean and adjusts it over time;
- it uses a **threshold** for change detection; exceeding this threshold indicates a potential mean shift;
- its implementation is straightforward and it offers **timely (rapid) detection**, but
- its effectiveness depends on the **choice of threshold** and mean estimates.

28.2.2 Maintaining Statistics

It most applications, analysts are interested in tracking data stream statistics over time. In theory, as long as we have saved all of the stream's numerical values, we can compute the mean and variance (to name but two); in practice, the **direct approach** is often fraught with issues – even something as simple as identifying the number of instances in a stream can quickly devolve into a **big data problem**.

We would much rather be able to obtain a data stream's statistics **on demand** – in the lingo of data mining, this is called **maintaining** the stream's statistics – without having to compute it from scratch at any desired moment.

If we are only interested in the **mean** and the **variance** of a numeric data stream over a **landmark window** (that is, from time 0 to now), it suffices to maintain a **triple** of numbers (N, S, Q) where:

- N is the **total number** of instances seen up to now;
- S is the **sum of all the isntances** seen up to now, and
- Q is the **sum of the squares of the instances** seen up to now.

We can use the basic formulas to compute the desired statistics: the **mean** is simply S/N, while the **variance** is $Q/N - (S/N)^2$. As a new data stream instance x comes in, we only need to update the triple as follows:

- increment N by 1;
- increment S by x, and
- increment Q by x^2,

and use the basic formulas to get the mean and variance. By keeping (N, S, Q) current with the above mechanism, we get the (current) mean and variance on demand.[19]

Unfortunately, in most data stream applications, summary statistics over landmark windows are not suitable. Instead, we might be looking for summary statistics over a **recent time-window**. A mechanism for **forgetting**, **devaluing**, or **suppressing** old data is required.

ADWIN The **adaptive sliding window** (ADWIN) is an algorithm used to compute statistics over **sliding windows** [7].

It keeps a **variable-length window** of recent values, with the property that the window has the maximal length which remains statistically consistent with the hypothesis that there has been **no change** in the average value inside the window. ADWIN can thus be used to detect **change in the average value** of the data stream (non-stationarity). The basic idea is the following one.

Let W denote the window of recent values. We search for a **splitting point** that will divide W into two sub-windows W_0 and W_1, where W_0 represents **older** instances and W_1 the recent ones. We would then consider removing W_0 from W if the average value of instances over W_0 is "substantially different" from those over W_1.[20]

Consequently, we can confidently declare a change whenever the window W **shrinks**, and the mean over the existing window W is a **reliable estimate** of the current mean in the stream.

19: If we are interested in the **progression** of the mean and variance over time, we could maintain (and store) a 4−tuple $(t, N(t), S(t), Q(t))$, where t represents the **time stamp** at which new instances come in, with $N(t)$, $S(t)$, and $Q(t)$ representing N, S, and Q at t, and the mean and the variance at t being defined by $S(t)/N(t)$ and $Q(t)/N(t) - (S(t)/N(t))^2$. We may still need to store a rather large object to plot the data, say, but the mean and variance computations are not any more demanding than in the text.

20: The search for a split point (which we will describe shortly) does not need to take place upon every new arrival; it could take place every k new arrivals, for some positive integer k.

ADWIN Algorithm ADWIN requires specifying a **confidence value** $0 < \delta < 1$. We denote the sequence of data stream values by $x_1, x_2, x_3, \ldots$, where x_j only becomes available at time t_j. For simplicity's sake, we will further assume that $0 \leq x_j \leq 1$ and that x_j is drawn from a distribution D_j with **expected value** $\mathrm{E}(x_j) = \mu_j$; in practice, of course, we know neither D_j nor its expected value.

ADWIN uses a **sliding window** $W(= W^i)$, with the most recent arrival x_i going to the head of the window. We will denote the **mean of the instances over the window** W by $\hat{\mu}_W$. The size of a window W (which is to say, the number of observations in W) is denoted by $|W|$.

1. Initialize window W;
2. for each time t_j,
   a) add x_j to the head of $W(= W^j)$;
   b) while W can be split into $W = W_0 \sqcup W_1$ as in the previous discussion, with
   $$|\hat{\mu}_{W_0} - \hat{\mu}_{W_1}| > \sqrt{\frac{\ln(4|W|/\delta)}{2m}}, \quad \text{where } m = \frac{2}{\frac{1}{|W_0|} + \frac{1}{|W_1|}},$$
   then set $W = W_1$;[21]
   c) output $\hat{\mu}_{W^j}$.

21: In other words, m is the **harmonic mean** of W_0 and W_1.

Theorem: with the threshold δ, the following hold at every step t.

- False positive rate bound: if μ_j has remained constant within W up to time t, the probability that the window is shrunk is at most δ.
- False negative rate bound: if for some partition of W into $W_0 \sqcup W_1$ we have
  $$|\hat{\mu}_{W_0} - \hat{\mu}_{W_1}| > 2\sqrt{\frac{\ln(4|W|/\delta)}{2m}},$$
  then ADWIN shrinks W to W_1 (or shorter) with probability $1 - \delta$.

Histograms Loosely speaking, a **histogram** is a visual representation of frequency data as a bar graph.[22] Histograms are often used in reporting and exploratory data analysis. Constructing a histogram involves putting the observed values into **non-overlapping bins**.

22: We have discussed such charts in Chapter 7 (DUDADS, Volume 1), Chapter 18 (DUDADS, Volume 2), and [9].

In a data stream context, **up-to-date** histograms could lead to actionable insights in **real-time**. But constructing histograms from data streams can be challenging:

- since we do not know ahead of time what values new instances could take, the histogram's (long-term) **range** is unknown;
- consequently, its **number of bins** might need to be determined dynamically (which is to say, it may change as new instances arrive);
- equal-frequency bins require **sorting**, which cannot be achieved in **linear time**, and
- up-to-date histograms need to be available **on demand**, should the user request them.

In practice, we must settle for **approximate** histograms – as long as the approximations remain "decent", we can overlook the lack of exactness as the cost of obtaining real-time "insights" **when requested**.

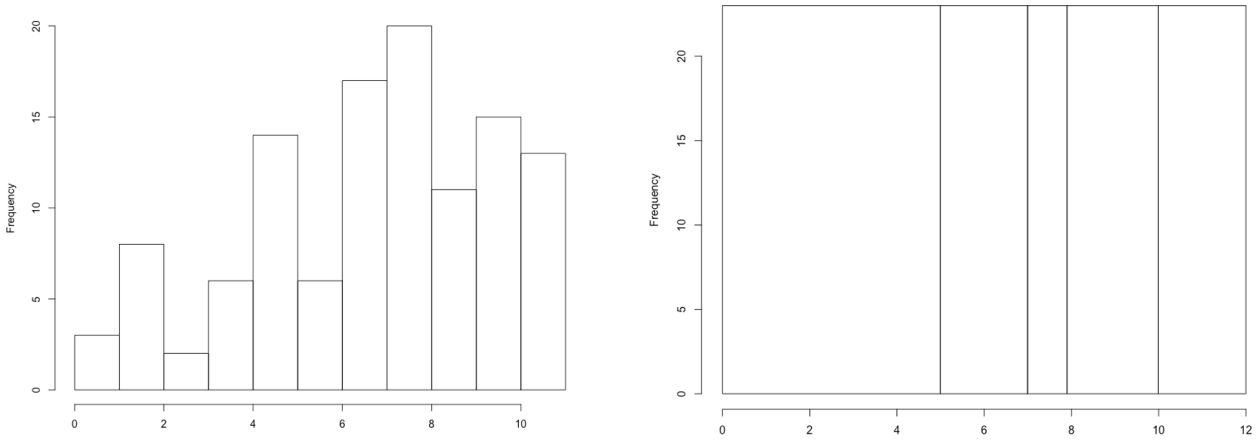


Figure 28.1: The same data is represented in two different types of histograms: **equal-width** (on the left), for which the height of the bar is proportional to the number of values in the bin, and **equal-frequency** (on the right), for which the width of the bar is proportional to the number of values in the bin.

PID The **partition incremental discretization** (PID) algorithm provides one way to maintain a data stream's histogram.

PID uses a **two-layer** (two-phase) approach:

- layer 1 obtains a **fine-grained summary** of the data objects as they arrive, while
- layer 2 **aggregates** the summary.

The idea of first obtaining a summary of the data and then performing an aggregation at lower granularity for the final result is quite common and will appear again in Sections 28.4 and 28.3.

Layer 1 We can think of layer 1 as an **intermediate histogram** summarizing the incoming data at a relatively fine granularity. It is initialized before any data is seen as follows:

1. pick a tentative but **sufficiently wide** range for the values (it can be extended later, as needed), and
2. partition the range into a **high number of equal-width intervals** (i.e., the histogram will have a large number of bins), with the first bin extending to ∞ to the left and the last bin to ∞ to the right.

Upon receiving a new instance x, we find the bin in which it falls and add 1 to that bin's **count**. If the bin count exceeds a **pre-specified threshold** (that is, if it contains more than a pre-specified percentage of the total number of values seen so far), then:

1. if the bin in question is the **first bin**, insert a bin with default width to the left of the first break point and distribute the count evenly between this new bin and the left-most bin;
2. if it is the **last bin**, insert a bin with default width to the right of the last break point and distribute the count evenly between this new bin and the right-most bin;
3. **otherwise**, split the bin into two bins with equal width and distribute the count evenly between the two bins.

The process of updating layer 1 is illustrated in Figure 28.2. As more and more instances arrive, the bins in the updated first layer are **unlikely to**

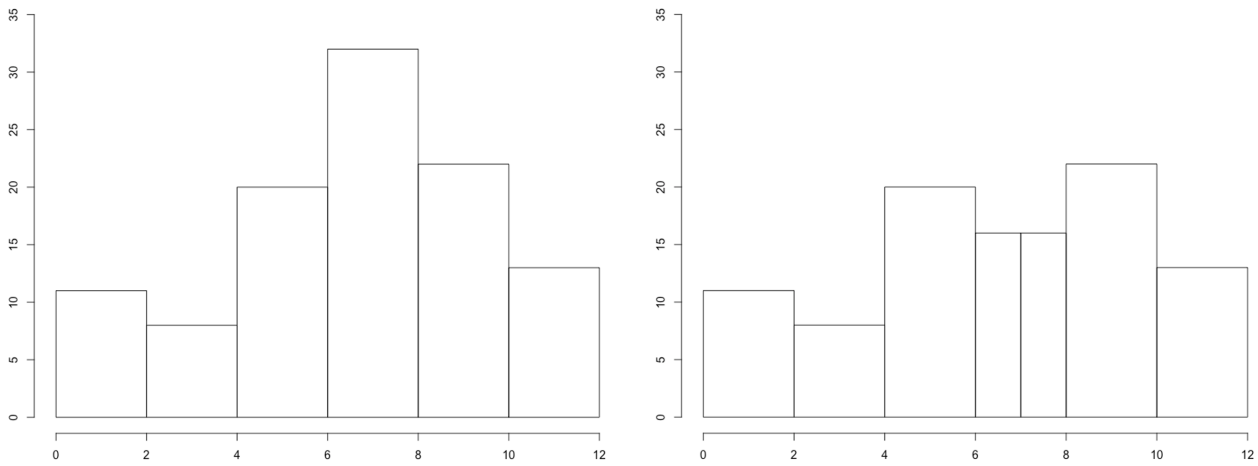


Figure 28.2: Maintaining a histogram: updating layer 1. Assume that the histogram on the left has bin $(6, 8]$ at threshold. If the next value arriving from the stream is 7.5, a bin split is triggered: the updated layer 1 is shown on the right.

be of equal width, and their number will **far exceed** the actual number of bins that would ever be displayed, both of which are acceptable as layer 1 is updated incrementally and typically stays **hidden** from the user; recall that layer 2 is only constructed from layer 1 **upon request**.

Layer 2: Equal-Width Histograms To build an **equal-width** layer 2 histogram, we select a bin width that is an **integer multiple** of the default width used for layer 1; layer 2 thus has fewer bins than the layer 1 histogram. For each bin B in the layer 2, we set the **bin count** to the tally of the bin counts in layer 1 for bins that are within the interval defined by B. For instance, consider the layer 1 histogram below.

Interval	Count
(-∞, 1]	4
(1, 2]	12
(2, 3]	8
(3, 4]	15
(4, 5]	17
(5, 6]	9
(6, 7]	10
(7, ∞)	9
(-∞, ∞)	60

Table 28.1: Layer 1 histogram for a data stream at the time of a display request from user.

Each bin in the first layer is of length 1 (apart from the first and last bin, which are of infinite length); we chose a layer 2 bin length of $3 \times 1 = 3$; the layer 2 equal-width histogram is thus as shown in Table 28.2.

Interval	Count
(-∞, 1]	4
(1, 4]	35
(4, 7)	36
(7, ∞)	9

Table 28.2: Equal-width layer 2 histogram for a data stream at the time of a display request from user.

Layer 2: Equal-Frequency Histograms To build an **equal-frequency** layer 2 histogram, we select a desired number of bins k. Let $C = N/k$, where N is the **total number of instances** captured by layer 1.

The bins in the second layer 2 are created as follows:

1. tally the bin counts in Layer 1 starting from the left-most bin;
2. as soon as the tally meets or exceeds C, insert a new breakpoint at the **right breakpoint** of the last bin of layer 1 used in the tally;
3. resume tallying bin counts starting at the **next bin** in layer 1;
4. repeat this process until all the bins in Layer 1 have been processed.

With the layer 1 histogram of Table 28.1 and $k = 3$, we have

$$C = N/k = 60/3 = 20;$$

the equal-frequency layer 2 histogram is thus as shown in Table 28.3.

Interval	Count
(-∞, 1]	4
(1, 2]	12
(2, 3]	8
(3, 4]	15
(4, 5]	17
(5, 6]	9
(6, 7]	10
(7, ∞)	9
(-∞, ∞)	60

Interval	Count
(-∞, 3]	30
(3, 5]	32
(5, ∞)	28

Table 28.3: Equal-frequency layer 2 histogram for a data stream at the time of a display request from user.

PID Revisited The bad news is that PID does not generate **exact histograms** in general because of the **granularity** of the first layer.

In particular, there could be issues with the **set of boundaries** (layer 2 breakpoints are restricted to the set of breakpoints available in layer 1) and the **frequency counts** (in layer 1, the bin counts are exact **only if no splitting takes place**).

The good news is that both layers can be constructed **efficiently**, however. When mining data streams then, as in life in general, you win some and you lose some.

PID and Change Detection One way to detect change *via* data stream histograms is to maintain a layer 2 histogram over a **reference time window** and compare with a layer 2 histogram over a **recent time window**, as distributions.

Provided that the number of bins and the widths of the two histograms are the same, the **Kolmogorov-Smirnov test** [29] can be used. Alternatively, one can also compute the **Kullback-Leibler divergence** [30] which measures the "distance" between two probability distributions.

If the histograms are deemed to be **statistically different**, then concept drift has occurred.

28.3 Clustering

Clustering is the process of partitioning data points into different groups so that data points within each group have **high similarity** whereas those from different groups have **low similarity**.

We have discussed these notions and presented various algorithms to cluster offline data in Chapters 19 and 22 of [8, vol. 3] (see also [2]).

28.3.1 Basics and Challenges

But when data is arriving **continuously** (or in small chunks), it might be useful to incorporate the new data into the existing models without re-computing everything from scratch (especially when the underlying data sets are large).

This is often conducted in **batches**: the data is collected until some threshold quantity is met, and each batch of new data is then **combined with the existing model** to create a new model.

But there are technical challenges, as we have seen: if the data arrive **continuously** in real-time (and at high velocity), then there could be time constraints. More worryingly, clusters could evolve over time, which is to say that new clusters might **emerge** or old clusters might **merge** or disappear altogether.

28.3.2 Approaches

There are two main approaches for clustering over data streams.

Partitioning In this approach, we start with k clusters obtained from an initial batch of data from the stream and we continuously update the clusters as new data points arrive, thus maintaining k clusters throughout.

In the $k-$means context, for instance, after batch t have been processed and clustered, let:

- $c_i(t)$ be the ith cluster centroid;
- $n_i(t)$ be the number of points assigned to the ith cluster;
- $x_i(t)$ be the centroid of the points batch $t+1$ closer to $c_i(t)$ than to any other centroid $c_j(t)$, and
- $m_i(t)$ be the number of points in batch $t+1$ closer to $c_i(t)$ than to any other centroid $c_j(t)$.

The **streaming** $k-$**means** algorithm updates $c_i(t)$ and $n_i(t)$ as follows:

$$c_i(t+1) = \frac{\alpha c_i(t) n_i(t) + x_i(t) m_i(t)}{\alpha n_i(t) + m_i(t)}$$
$$n_i(t+1) = n_i(t) + m_i(t),$$

where $\alpha \in (0, 1]$ is a **decay factor** used to weigh older data less relative to new data.[23]

23: Some other popular algorithms allow the number of clusters to deviate from k within a small multiplicative factor to obtain better clustering performance.

Two-Phase In this approach, we maintain many small clusters (sometimes called **micro-clusters**) that summarize the incoming data, and perform **offline clustering** on these small clusters, treating them as **weighted virtual data points**.

We will consider two methods in particular: CluStream and DenStream.

Technical Assumptions Throughout, we assume that:

- data points belong to $\mathbb{R}^d$ for some positive integer d;[24]
- the Euclidean distance is used to measure the dissimilarity between points,[25] and
- we describe clusters using a **representative**, the **number of points** in the cluster, its **density** and/or **shape**, and its **distances** to other clusters.[26]

Distance-based methods usually result in **spherical** (blob-like) clusters; density-based methods can result in **non-convex** clusters.[27] The application at hand dictates which type of cluster is most desirable.

In a data stream setting, we are unlikely to keep all the data in memory; we work instead with **data summaries**. One key idea found in many clustering algorithms is the concept of **cluster features** (CF). The simplest CF structure in use is a triple $(N, \mathbf{S}, \mathbf{Q})$, one per cluster, where:

- N is the number of points assigned to a cluster;
- $\mathbf{S} \in \mathbb{R}^d$ is vector of the sum of the variables for the N observations in the cluster, and
- $\mathbf{Q} \in \mathbb{R}^d$ is the vector of the sum of the squares of the variables for the N observations in the cluster.

For instance, suppose that the $N = 3$ points $(1, 2)$, $(2, -1)$, $(0, 1)$ are assigned to the same cluster. Then

$$\mathbf{S} = (1 + 2 + 0, 2 + (-1) + 1) = (3, 2)$$
$$\mathbf{Q} = (1^2 + 2^2 + 0^2, 2^2 + (-1)^2 + 1^2) = (5, 6),$$

so the cluster's CF is $(3, (3, 2), (5, 6))$, and that is the cluster summary that would be stored, the actual points being discarded or archived.

However the CF is defined, it should have two properties: **incrementality** and **additivity**. If a new point x is assigned to an existing cluster with CF $(N, \mathbf{S}, \mathbf{Q})$, then the new CF for the cluster is simply $(N + 1, \mathbf{S} + x, \mathbf{Q} + x^2)$ where x^2 is obtained from x by squaring each component.[28] If we merge clusters A and B, with corresponding CF $(N_A, \mathbf{S}_A, \mathbf{Q}_A)$ and $(N_B, \mathbf{S}_B, \mathbf{Q}_B)$, then $(N_A + N_B, \mathbf{S}_A + \mathbf{S}_B, \mathbf{Q}_A + \mathbf{Q}_B)$ is the CF of the merged cluster.

Being able to update CF quickly when new points are added to a cluster or when clusters merge is **critical** for being able to cluster high-velocity data streams. The importance of additivity will become apparent when we discuss CluStream.[29]

The simple CF defined above provides sufficient information to compute some **cluster metrics** that can help us decide when to merge or create new clusters.

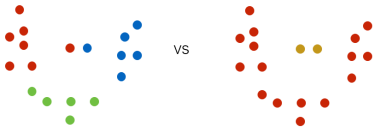
- The **centroid** (mean) of a cluster is $\mathbf{S}/N$.

24: Thus, we must embed nonnumerical data (such as text) before we can cluster the data – see Chapter 32 for examples.

25: Other options are available, of course.

26: We typically use cluster-level summary statistics to do so.

27: Convex clusters (left) vs. non-convex clusters (right), below.



28: Remember that x is a vector observation.

29: In a nutshell, the property allows us to "subtract" older micro-clusters from recent micro-clusters when clustering over a time window.

- The **radius** of a cluster is

$$\sqrt{\frac{s(\mathbf{Q})}{N} - \left\|\frac{\mathbf{S}}{N}\right\|^2},$$

  where $s(\mathbf{X}) = X_1 + \cdots + X_d$.
- The **diameter** of a cluster is

$$\sqrt{\frac{s(\mathbf{Q})N - (s(\mathbf{S}))^2}{\binom{N}{2}}}.$$

Note that these differ from the usual geometrical definitions of the radius (the average distance of the points to the centroid) and the diameter (the maximum distance between any two points in the set).

28.3.3 Evaluation

Evaluating the **quality of a clustering outcome** is of course important – we would definitely like to know how "good" the clusters returned by a clustering method are. Unfortunately, it can be difficult to come up with evaluation measures in unsupervised learning settings.

Many evaluation measures have been proposed, including **internal measures** and **external measures**.[30]

30: See discussion in Section 22.3 (see [8, vol. 3]) for more details.

Internal Measures

- **Cohesion:** the average distance from a point to the centroid of the cluster to which it is assigned (the smaller the better).
- **WSS**: the sum of squared distances from data points to their assigned centroids (the smaller the better).
- **Separation:** the average distance from a point to the points assigned to other clusters (the larger the better).

External Measures External measures require knowledge of the **ground truth**, which is to say, the "true" clustering of the data points. In practice, the ground truth is usually unavailable (assuming that there is even one in the first place); in research settings, known clusters are generated to test clustering methods.

- **Accuracy:** the fraction of the points assigned to their correct cluster.
- **Recall:** the fraction of the points of a cluster that are actually assigned to it.
- **Precision:** the fraction of the points assigned to a cluster that truly belong to it.
- **Purity:** $(f_1 + f_2 + \cdots + f_c)/N$ where f_i is the maximum number of points in the computed cluster i belonging to the same true cluster; the highest possible value is 1.[31]

31: A drawback of this measure is that it can be gamed by assigning each point to its own cluster.

28.3.4 Algorithms

Current popular data stream clustering methods follow a framework similar to the one proposed in [3]. The framework consists of **two phases**:

1. we first create and maintain a **data abstraction** (summary structure, sketch, microclusters, etc.) to hold a summary of the incoming data at a reasonably **fine granularity**;[32]
2. next, we invoke a traditional clustering algorithm on the data abstraction to obtain a small number of **global macro-clusters**, upon request by the user.

32: For example, there can be millions of micro-clusters even though we may want no more than 10 clusters in the end.

CluStream The original algorithm of [3] maintains multiple **micro-clusters** and it clusters over an **approximate time window**.[33]

33: We describe an updated version of CluStream which uses the **geometric time frame**, which is simpler to implement than the original **pyramidal time frame**. The former is described in [4].

CluStream requires an extended CF which includes **temporal summary statistics**; for each micro-cluster, T is the sum of the time stamps of the points assigned to the micro-cluster and T_Q is the sum of the squares of those same time stamps.

At any given time, CluStream maintains q micro-clusters, where q could be quite large, as long as we can store the micro-clusters in memory. The q micro-clusters are **initialized** by applying an offline clustering method (such as $k-$means, say) to the first batch in the data stream, and they are assigned a unique ID.

After initialization, whenever a new data point arrives, it is either

- **absorbed** into an existing micro-cluster, or
- a micro-cluster is **created** solely for this point.

Which of these options is selected depends on the distance of the new point to the existing micro-clusters.

When a new micro-cluster is called for, we need to either **delete an old cluster** or **merge two neighbouring micro-clusters** into a single one so that we do not exceed the total number of maintained micro-clusters.[34] The decision to delete an old cluster is based on the temporal summary statistics in the extended CF of the micro-clusters.

34: A merged micro-cluster contains the IDs of the micro-clusters that were merged – these IDs are needed when clustering over a time window.

Every once in a while, a **snapshot** of the micro-clusters is taken and is stored away. The snapshot schedule follows a **geometric time frame**, leading to snapshots stored at different levels of granularity depending on how recent the data is.

If one unit of clock time (say 1 second) is the finest granularity, then each snapshot is classified with a **frame number**, a value from 0 to $\log_2(T)$, where T is the maximum length of the stream (such as $T = 20$ yrs $\approx 6.3 \times 10^8$ sec).

The snapshots with frame number i are then stored at clock times divisible by 2^i but not by 2^{i+1}. For instance, frame number 0 would contain snapshots at odd clock times, frame number 1 would contain even clock times not divisible by 4, frame number 2 would contain snapshots at clock times divisible by 4 but not by 8, frame number 3 would contain snapshots at clock times divisible by 8 but not by 16, and so on.

Let M be a relatively small positive integer. We retain/store only up to the M most recent snapshots for each frame number that will be stored; the older ones are deleted. For instance, suppose that $M = 3$ and the current clock time is 70. The table below shows the clock times of snapshots for various frame numbers.

Frame number	Clock times of snapshots
0	69, 67, 65
1	70, 66, 62
2	68, 60, 52
3	56, 40, 24
4	48, 16
5	64, 32

This approach maintains a large number of micro-clusters on-line, takes **intermittent snapshots** of the micro-clusters and stores them away according to a **geometric time frame**, and uses an offline clustering algorithm on the micro-clusters to obtain macro clusters when requested by the user.

An important property of the geometric time window is that for any time window, at least one stored snapshot can be found within a factor of 2 of the specified horizon.

Lemma: let $[h, t_c]$ be a user-specified time window where t_c is the current time. If $M \geq 2$, then a snapshot exists at a time t_s for which $h/2 \leq t_c - t_s \leq 2h$.

To cluster over the time window $[h, t_c]$ where t_c is the current time, we find a snapshot at a time t_s for which $h/2 \leq t_c - t_s \leq 2h$ (whose existence is guaranteed by the lemma) and consider the micro-clusters at t_c.

For each micro-cluster A at t_c, we identify all the micro-cluster ID associated with this micro-cluster and subtract away the CF summary stats of the corresponding micro-clusters in the older snapshot from the CF of A.

For instance, let's assume that micro-cluster A contains the micro-cluster ID 2 and 9 at t_c.[35] Next, we look up the micro-clusters with ID 2 and 9: their respective CF are $(N_2, \mathbf{S}_2, \mathbf{Q}_2)$ and $(N_9, \mathbf{S}_9, \mathbf{Q}_9)$.[36] We adjust the CF of A, $(N_A, \mathbf{S}_A, \mathbf{Q}_A)$, to $(N_A - N_2 - N_9, \mathbf{S}_A - \mathbf{S}_2 - \mathbf{S}_9, \mathbf{Q}_A - \mathbf{Q}_2 - \mathbf{Q}_9)$.

Finally, we perform **(weighted)** $k-$**means** on the adjusted micro-clusters, treating their CF as **virtual** data points: these **macro-clusters** would be the ones that are reported on demand.

The result will be a clustering over a time window that is not too far off from the user-specified time-window $[h, t_c]$.[37]

35: That is, A was formed by merging micro-clusters with IDs 2 and 9 at some point in the past.

36: We are ignoring the temporal summary stats for the current illustration but they would be tackled in the same manner.

37: Obviously, t_c could be replaced by an earlier time for which we have a snapshot. So we are not always restricted to a most current time window. This may prove useful for change detection.

DenStream We will not get into the details of DenStream, other than to say that this algorithm was proposed in [10] (see also [7]); the authors claim that it can overcome CluStream's limitation in handling **noise**, **outliers**, and **non-spherical clusters**.

DenStream uses a **fading function** to calculate micro-cluster **weights**; points that are more distant in time contribute less to the weight. Unlike CluStream, DenStream does not allow users to obtain clusters over a particular time window.

There are many other clustering algorithms over data streams – we have covered only some basic ideas and a selection of methods. This is a field of active research, with different methods addressing the demands of different applications.

Unfortunately, many algorithms published in research papers still do not have readily accessible implementations for use in production.[38]

38: This is also the case for classification algorithms.

28.4 Classification

In machine learning, **classification** is the task of assigning a category or label to a new data object.[39]

39: For instance:

- Is an image that of a cat?
- Is an e-mail message spam?
- Is a credit card transaction fraudulent?
- Is a student at risk of failing?
- Is an HTTP request a network attack?

We have discussed these notions and presented various algorithms to classify offline data in Chapters 19 and 21 of [8, vol. 3] (see also [1]).

28.4.1 Basics and Challenges

Classifiers are trained on a pre-defined set of examples. In the case of spam filtering, e-mail messages that are known to be spam and those known not to be spam (a.k.a. ham) are used for training. They key is that the categories (true labels) of the training data are known. Classifier training, unlike clustering, is a case of **supervised learning**.

Challenges The need for the true labels for the training data seems to lead to a dilemma for building classifiers over data streams. The point of having a classifier is to **predict the label** of a **new data object**. A trained classifier can predict over static data or from a data stream; in other words, once we have a trained classifier, it can be put into operation whether or not the data objects come from streams.

For data coming from a data stream, where do the true labels come from? They cannot come immediately with the data or there would be no need to build classifiers. If the true labels come after the fact, how do classifiers **adapt** to concept drift? For example, spam filters cannot be static as new types of spam messages can arise and old ones can go out of circulation.

Settings In a batch setting, examples with their true labels are stored in a data base, and **random access** to the data is assumed. A portion of these examples is set aside to form the **test set**; once a classifier is trained, it can be put into operation for prediction over a data stream or a data base.

But in a data stream setting, not all the examples may be available at once. True labels might not be available right away and may only be available some time after the rest of the data arrive.

For example, say an e-mail message arrives and the existing spam filter misclassifies it. The user marks it with the correct label (e.g., by clicking the 'Is Spam' or 'Not Spam' button) and the classifier can then be updated with this new example or along with other accumulated examples, assuming that an **incremental training method** is used.

Scenarios But we don't necessarily need methods to train classifiers over data stream only because we need to build classifiers **on the fly** – it could happen that the training examples are only accessible in a **stream-like fashion** (e.g., in a MapReduce framework) in which case multiple access to the data is very costly or infeasible.

Another reason for having training methods that operate over data streams is to deal with **concept drift**: for instance, e-mail spam can **evolve over time**, as can network attacks, fraudulent transaction patterns, etc.

We thus need training methods that can adapt from new data – we do not want to retrain from scratch every time we think the current model is out-of-date.

Obtaining True Labels In general, obtaining **true labels** is a challenging task. In the past, labeling was done manually. For example, image labels were entered by human beings looking at the image and typing in the correct label. Nowadays, crowdsourcing is one method employed by companies that offer image-sharing platforms; they scrap information from tags or captions provided by the platform users. **Clustering algorithms** are sometimes used to speed up the manual labeling process in what is called **semi-supervised learning**.

There is on-going research on how to obtain proper labels with few examples: promising avenues include **generalized adversarial networks** (GANs) and **one-shot learning** (see the work of [25]).

Evaluation It is important to be able to compare the performance of different classifiers and training methods. However, classifier training often involves **massive** amounts of data and does not come cheap. Hence, we search for a "good" **return on investment:** good performance should not come at too steep a price.

In practice, this eliminates the traditional **cross-validation** method as being too costly, computationally, in favour of other approaches, such as: **holdout**; **interleaved test-then-train**; **prequential**, or **interleaved chunks**.

28.4.2 Approaches

There are tons of classification algorithms for batch settings [8, ch.21], but deriving versions of these that can be trained from data streams is not a trivial endeavour.

In the rest of this section, we provide a brief description of two algorithms: **Hoeffding trees** and **on-demand** k**NN** (an extension of CluStream). We conclude with a few remarks on **ensemble classifiers**.

Hoeffding Trees In a classification tree, at each **intermediate** node (i.e., not a leaf), we query a specific data feature to determine which **child** to descend/**branch out** to. Once a leaf is reached, the label associated with the leaf is the label assigned to the data object under consideration.

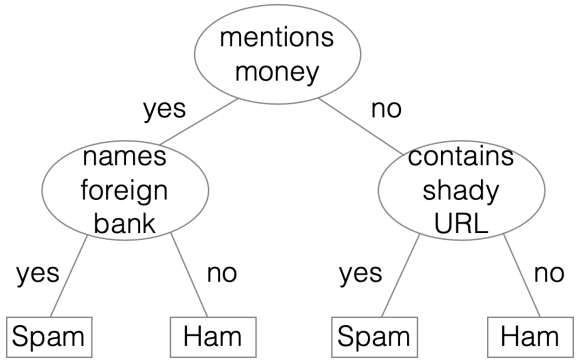


Figure 28.3: Example of a spam-filtering decision tree.

Suppose that we want to build a decision tree classifier from a set of examples with a fixed set of features. Using Shannon's information theory, we can compute the **entropy** of this set (see [8, sec 19.4.3] for details).[40]

The process of building a decision tree for classification involves selecting a feature and a condition for splitting the set into two smaller sets so that the total entropy of the two subsets is less than the entropy of the set taken as a whole. Usually, at each stage, we choose the feature that leads to the **maximum reduction of entropy**.

The process typically stops way before we reach a stage where no further splits can reduce the entropy – there are heuristics to determine when the time is ripe to **stop creating new decision nodes**.

Decision trees are very easy to build once the entire training is available and the results are **interpretable**, but once a model is found on the training data, it does not usually generalize very well to new data.[41]

In a data stream setting, we do not have access to the entire training set at any given time, however. The idea behind a **Hoeffding** tree is not to train on the entire training set (coming through a stream) but to first accumulate a "sufficient" number of observations before making decisions about splitting.

The algorithm is shown in Figure 28.4; the precise details are fairly technical and can be found in [12]. The method depends on a well-known result in probability theory called the **Hoeffding bound**, which provides a statistical guarantee that a decision made from a small data sample matches the decision that would be made if all the data was available.[42]

40: In a nutshell, the entropy of a set can be thought of as the minimum number of bits required to represent the **disorder** in the data.

41: The addition of a few new training instances could dramatically change the topology of the model Extensions (such as **boosting** or **random forests**) can be used to attempt to mitigate this shortcoming.

42: Similarly, a few flips of a coin (fewer than a hundred, certainly) are usually sufficient to give an idea of the true odds of flipping 'Heads' for that coin.

Inputs:	S	is a sequence of examples,
	$\mathbf{X}$	is a set of discrete attributes,
	$G(.)$	is a split evaluation function,
	δ	is one minus the desired probability of choosing the correct attribute at any given node.
Output:	HT	is a decision tree.

Procedure HoeffdingTree $(S, \mathbf{X}, G, \delta)$
Let HT be a tree with a single leaf l_1 (the root).
Let $\mathbf{X_1} = \mathbf{X} \cup \{X_\emptyset\}$.
Let $\overline{G}_1(X_\emptyset)$ be the $\overline{G}$ obtained by predicting the most frequent class in S.
For each class y_k
  For each value x_{ij} of each attribute $X_i \in \mathbf{X}$
    Let $n_{ijk}(l_1) = 0$.
For each example $(\mathbf{x}, y_k)$ in S
  Sort $(\mathbf{x}, y)$ into a leaf l using HT.
  For each x_{ij} in $\mathbf{x}$ such that $X_i \in \mathbf{X}_l$
    Increment $n_{ijk}(l)$.
  Label l with the majority class among the examples seen so far at l.
  If the examples seen so far at l are not all of the same class, then
    Compute $\overline{G}_l(X_i)$ for each attribute $X_i \in \mathbf{X}_l - \{X_\emptyset\}$ using the counts $n_{ijk}(l)$.
    Let X_a be the attribute with highest $\overline{G_l}$.
    Let X_b be the attribute with second-highest $\overline{G}_l$.
    Compute ϵ using Equation 1.
    If $\overline{G}_l(X_a) - \overline{G}_l(X_b) > \epsilon$ and $X_a \neq X_\emptyset$, then
      Replace l by an internal node that splits on X_a.
      For each branch of the split
        Add a new leaf l_m, and let $\mathbf{X_m} = \mathbf{X} - \{X_a\}$.
        Let $\overline{G}_m(X_\emptyset)$ be the $\overline{G}$ obtained by predicting the most frequent class at l_m.
        For each class y_k and each value x_{ij} of each attribute $X_i \in \mathbf{X_m} - \{X_\emptyset\}$
          Let $n_{ijk}(l_m) = 0$.
Return HT.

Figure 28.4: The Hoeffding tree algorithm.

If $\delta > 0$ is the decision procedure's **confidence level** and $n \geq 1$ is the **number of training observations**, then the error margin ε is

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

if $r \in [0, 1]$ is the true probability that an observation's class is A, say, and if the model built on n training observations suggests that the probability of the observations is $\overline{r} \in [0, 1]$, then

$$P(|\overline{r} - r| < \varepsilon) = 1 - \delta.$$

Practically speaking, a Hoeffding tree can confidently (at a pre-determined level δ) split an intermediate node **without having access to all the data**; as more data comes in, it may refine its decisions but it will **not reverse to an earlier state** unless there arises a "major" reason to do so.

Several versions of **Hoeffding Adaptive Trees** (which extend Hoeffding trees) are described in [6]: HAT-INC, HAT-EWMA, HAT-ADWIN; more recent work investigates the usage of drift detectors in HATs [23].

On-Demand kNN In [4], the authors proposed a framework for **on-demand classification** of evolving data streams using a **two-phase approach** as in CluStream:

- phase 1 involves the on-line maintenance of **micro-clusters**, while
- phase 2 involves an efficient **batch classifier** (kNN with small k).

kNN is simple; the user is only required to select a **measure of distance** between data points and a **positive integer** k. The training examples are stored; when a new point needs to be classified, we look for the k nearest stored examples and find their most frequent label – this becomes the predicted class for the new observation (see Figure 28.5).

But as we have seen, not all training examples are available at once in a stream setting. Additionally, there may be too many points to store in memory. Instead, we build a **summary** of the training examples in terms of **labeled micro-clusters**.[43]

When new micro-clusters are created, we may have to **delete** an old cluster or **merge** two nearby micro-clusters with the same label so the maximum number of **maintained micro-clusters** is not exceeded.[44]

As in CluStream, ID are needed when clustering over a time window. The decision to delete an old cluster is based on the temporal summary statistics in the **extended cluster features** (CF) of the micro-clusters.

To classify a new point using a classifier built with examples in the time window $[h, t_c]$, we find a snapshot at time t_s such that $h/2 \leq t_c - t_s \leq 2h$ (guaranteed by the lemma on page 1843) and the micro-clusters at the **current time**.

For each micro-cluster A at time t_c, we identify all the micro-cluster ID associated with A and subtract away the CF summary stats of the corresponding micro-clusters in the **older** snapshot provided by the CF of A.

The new point is then assigned the label of the **nearest micro-cluster**.

43: As were defined/used in CluStream, in the previous section.

44: A merged micro-cluster contains the ID of the micro-clusters that are merged.

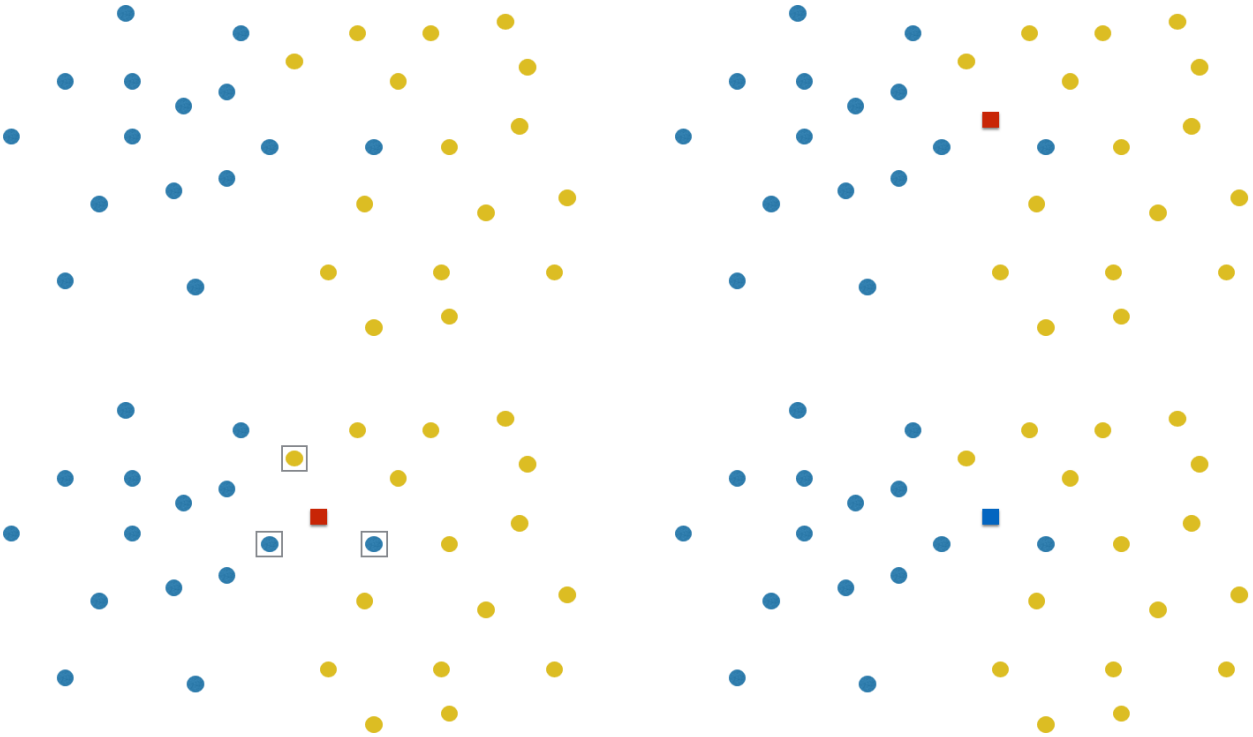


Figure 28.5: A review of kNN classification: a two-class dataset (top left); a new observation (in red) which must be classified (top right); comparing with $k = 3$ nearest neighbours (bottom left); classifying the observations as a blue square (bottom right).

28.4.3 Ensemble Classifiers

As the name suggests, an **ensemble classifier** consists of a set of individual classifiers. They do not all have to be of the same type: those that are, however, are trained on **slightly different data** and/or with **slightly different parameters**. A **(weighted) majority** of the individual classifiers provides the **ensemble classification**.[45]

45: See [8, sec. 21.5] for more details.

As an example, suppose we have an ensemble classifier for email spam consisting of three tree-based classifiers. When a new e-mail message is received, each of these three classifiers independently predicts whether the e-mail message is 'spam' or 'ham'; if **two classifiers or more** predict that the message is 'spam', then the ensemble prediction is that the e-mail message is 'spam', **otherwise** the ensemble prediction is 'ham'.

Adapting to Change The main advantage of maintaining an ensemble of classifiers in the streaming setting is that it provides the ability to handle **concept drift**.

If the performance of a classifier in the ensemble declines over time, say, then we can **decrease** its weight in the ensemble decision; if its performance becomes completely unacceptable, then it can be **removed** from the ensemble outright.

Similarly, we can add new classifiers to the ensemble to improve its overall performance without having to discard any individual classifier with (still) acceptable performance.

An early attempt to address concept drift using ensemble classifiers is provided in [24]; a more general framework is proposed in [26].

28.5 Frequent Itemset Mining

Say we have a data source of **transactions**, each of which consists of a set of items – we may be interested in discovering the common collections of items that occur among the transactions. Such item collections are called **frequent itemsets**.

Frequent itemset mining powers **association rule mining** [8, sec. 19.3]), which used to be common component of **recommender systems**.[46] An **association rule** takes the form $X \rightarrow Y$ where X and Y are sets. One possible interpretation of $X \rightarrow Y$ is "If we see X, we are likely to see Y as well."

46: In recent years, **collaborative filtering** has become more popular than association rule mining for recommenders (see Chapter 34).

It would be a mistake to treat such rules as **causal relationships**, however; data mining algorithms are typically incapable of learning **causality** (see Chapter 36 for more on this topic). At best, association rules reflect **correlations** found in data.

Example Suppose that a small supermarket sells only 4 types of products: apples, bread, cheese, and watermelons. There are $2^4 - 1 = 15$ possible itemsets of purchased goods, as can be seen in the table below.

Itemset size	Possible itemsets
1	{🍎}, {🍞}, {🧀}, {🍉}
2	{🍎,🍞}, {🍎,🧀}, {🍎,🍉}, {🍞,🧀}, {🍞,🍉},{🧀,🍉}
3	{🍎,🍞,🧀},{🍎,🍞,🍉},{🍎,🧀,🍉},{🍞,🧀,🍉}
4	{🍎,🍞,🧀,🍉}

Table 28.4: Possible itemsets in the supermarket example, when only 4 items can be purchased.

In practical situations, the number of possible itemsets is **much larger** than the total number of items – a typical supermarket easily carries 1000+ different items, so there are already at least $2^{1000} - 1 \approx 10^{301}$ **non-empty itemsets**!

In general, the number of possible itemsets is **exponential** in the number of items, and it becomes highly impractical (if not impossible) to keep track of that many itemsets. But ways have been devised to mitigate the combinatorial explosion of itemsets (see [8, sec. 9.3] and [19]); as we cannot have our cake and eat it too, some of these methods sacrifice **exactness** and can lead to **false positives/negatives**.[47]

47: This trade-off between resource consumption and accuracy of results is par for the course in machine learning and data science.

One concrete way to tame the combinatorial explosion requires drastically cutting down the number of "interesting" itemsets. A **frequency threshold** $\sigma \in (0, 1)$ is first specified; an itemset X is considered **frequent** if it occurs in at least σ (viewed as a percentage) of the available transactions. For instance, when $\sigma = 20\%$, an itemset X is considered frequent if it occurred in at least 20% of all transactions.

The **support** of an itemset is the number of transactions that contain it; if an itemset appears in two transactions, say, then its support is 2.

We say that a frequent itemset is **closed** if it has no frequent **superset** with the same support.[48] A frequent itemset is **maximal** if it has no frequent superset.

Theorem: any maximal frequent itemset is also closed, but closed frequent itemsets need not be maximal.[49]

Example Consider the same small supermarket as before, after it introduced a fifth product: cookies. Five customers have made purchases today: their transactions are summarized in the table below.

Transaction	Items in transaction
t1	{🍎, 🍞, 🧀, 🍉}
t2	{🍞, 🍉, 🍪}
t3	{🍎, 🍞, 🍉, 🍪}
t4	{🍎, 🍞, 🧀, 🍉, 🍪}
t5	{🍞, 🧀, 🍉}

Table 28.5: Transactions in the supermarket example.

Suppose that the frequency threshold is $\sigma = 0.4$ and consider the itemset $X = \{\text{bread}, \text{cheese}\}$. The support of X is 3 since it is a subset of transactions t_1, t_4, and t_5, but not of transactions t_2 and t_3; X is a frequent itemset since its support is greater than $0.4 \times 5 = 2$.

The supersets of X with three items are

$$\begin{aligned} X_a &= \{\text{bread}, \text{cheese}, \text{apple}\}, \\ X_w &= \{\text{bread}, \text{cheese}, \text{watermelon}\}, \\ X_c &= \{\text{bread}, \text{cheese}, \text{cookie}\}. \end{aligned}$$

As X_a appears only in t_1, t_4, its support is 2; as X_2 appears only in t_1, t_4, t_5, its support is 3; as X_c appears only in t_4, its support is 1. That means that the X_a and X_w are frequent itemsets for this data, but that X_c isn't; in particular, X is neither closed nor maximal.[50] □

Apriori Algorithm Before we look at frequent itemset mining in data streams, we briefly discuss the **apriori algorithm** in a batch setting (see [8, sec. 19.3] for more details). This algorithm exploits the **apriori property**, which states that all subsets of a frequent itemset are themselves also frequent.[51]

48: If $A \subseteq B$, then A is a subset of B and B is a superset of A.

49: **Proof:** for the first statement, let X be a maximal frequent itemset that is not closed. Then there is a superset $Y \supseteq X$ with the same support as X. Thus, Y must be a frequent itemset, contradicting that X has no frequent superset. For the second statement, suppose that the support level for being frequent is 3. It could be that there is some closed frequent itemset X with support 4, but every frequent superset of X has support 3. ■

50: Note however that

{bread, watermelon, cookie}

is maximal.

51: The converse of that statement is that all supersets of an infrequent itemset are themselves also infrequent.

Suppose that $X = \{\text{bread}, \text{cheese}, \text{apple}\}$ is a frequent itemset in a transaction dataset; according to the apriori property, the following itemsets are also frequent:

$$\{\text{bread}, \text{cheese}\}, \{\text{bread}, \text{apple}\}, \{\text{cheese}, \text{apple}\},$$
$$\{\text{bread}\}, \{\text{cheese}\}, \{\text{apple}\}.$$

The idea behind the apriori algorithm is to build up frequent itemsets starting with small sets.

1. First, we make a pass over the database and identify all the **frequent items** (frequent itemsets with only 1 item);
2. then we enumerate the 2−item sets by adding one element to each of the frequent 1−item sets, in all possible ways;
3. from these generated 2−item sets, only those that are frequent are retained.
4. This procedure is then repeated to obtain the frequent 3−item sets, and so on, until there are no new frequent itemsets on which to build.

Example Suppose that apple, bread, cheese, and watermelon are frequent items in a transaction dataset. Then we only need to consider 1−item sets selected from the family

$$L_1 = \{\text{apple}, \text{bread}, \text{cheese}, \text{watermelon}\}.$$

All other items can be ignored because no superset that contains them can be frequent according to the apriori property.

The candidates 2−item sets are thus:

$$\{\text{apple}, \text{bread}\}, \{\text{apple}, \text{cheese}\}, \{\text{apple}, \text{watermelon}\},$$
$$\{\text{bread}, \text{cheese}\}, \{\text{bread}, \text{watermelon}\}, \{\text{cheese}, \text{watermelon}\};$$

those that are frequent in that list are retained to form the family L_2, and so on. □

General Algorithm The general aprior algorithm is simple: let L_k denote the family of all frequent k−item sets. Form all possible $(k+1)$−item sets (the **candidates**) that contain a k−item set of L_k; those that are also frequent are placed into L_{k+1}.

Unfortunately, the apriori algorithm is unsuitable for data streams because the algorithm requires **multiple scans of the database**, and resources are often wasted on the **generation of candidate**, many of which ultimately ending up **discarded**, in typical applications.

A number of additional challenges also arise when mining frequent items in data streams:

- the data often must be processed in a **single pass**;
- computer memory is limited and cannot always store all of the stream's data, and
- frequent itemsets can be **time-sensitive** (concept drift).

Popular Algorithms In a streaming context, frequent itemset algorithms are classified based on a number of characteristics:

- **approximate** or **exact** (allows false positives/negatives or not)
- **per-batch** or **per-transaction** processing
- **incremental**, **sliding window**, or **adaptive**
- **frequent**, **closed**, or **maximal** itemsets

Some popular algorithms are shown in the table below (see [18] for a more comprehensive table).

Algorithm	Window	Batch?	Accuracy	Itemsets
LossyCounting	Landmark	Yes	False+	All
FP-stream	Tilted	Yes	False+	All
FDPM-1	Landmark	Yes	False-	All
MOMENT	Tilted	Yes	Exact	Closed
IncMine	Tilted	Yes	False+	Closed
estDec+	Landmark/ Damped	No	False+/-	Maximal

Table 28.6: Popular frequent itemset mining data streams algorithm.

LossyCounting Proposed in 2002 [21], **LossyCounting** is the first one-pass algorithm to find all frequent itemsets over a data stream. It does not allow false negatives and provides a **theoretical guarantee** on false positives.

For a user-defined **error parameter** $0 \leq \varepsilon \leq 1$ and **support threshold** $\theta \in (0, 1)$, all itemsets with a relative frequency in excess of θ will be found; conversely, no itemset whose true relative frequency is less than $\theta - \varepsilon$ will be included.

In a data stream, the LossyCounting algorithm for frequent 1−item sets works as follows:

1. the stream is divided into **buckets**, each of size $\lceil 1/\varepsilon \rceil$;
2. the buckets are **indexed sequentially** starting at 1;
3. counters of the form $\langle$`element`, `count`, `bucket_id`$\rangle$ are maintained, where `bucket_id` denotes the ID of the bucket being processed when the counter was created;
4. at the end of each bucket, we check if `count` + `bucket_id` $\leq \varepsilon N$, where N denotes the number of transactions seen so far – if the inequality holds, the counter is deleted.

The modification to handle frequent k−item sets is a bit more complicated. Assume that we have a **data structure** D, initially empty, consisting of a set of entries of the form (`itemset`, f, Δ) where f is the **estimated frequency** of the itemset and Δ is the **maximum possible error** in f.

1. The stream is divided into **buckets** of $w = \lceil 1/\varepsilon \rceil$ transactions each;
2. the buckets are **indexed sequentially** starting at 1
3. the current bucket id is denoted by `bcurrent`;

4. the transactions are not processed individually but in **batches**, each containing as many transactions as memory allows;
5. let β be the number of buckets in **main memory** in the batch being **currently processed** (β must be relatively "large");
6. D is updated as follows:
   a) for each entry (`itemset`, f, Δ) in D, increment f by the number of occurrences of `itemset` in the **current batch**;
   b) if $f + \Delta \leq$ `bcurrent`, the entry is **removed** from D;
   c) if an `itemset` has frequency f at least β times in the current batch and `itemset` does not occur in D, a new entry (`itemset`, f, `bcurrent` $- \beta$) is added to D.

We observe that a set `itemset` with true frequency at least εN must have an entry (`itemset`, f, Δ) in D. Furthermore, the true frequency is at least f and at most $f + \Delta$.

Note that we can also provide a list of itemsets with support threshold θ by outputting the entries in D with $f \geq (\theta - \varepsilon)N$.

Conceptually, LossyCounting is a rather simple algorithm, but its **efficient implementation** requires the use of **non-trivial data structures** whose descriptions are beyond the scope of these notes.[52]

52: A common implementation makes use of a "trie" (a **prefix tree**), which is a search tree for strings that can be dynamically updated. A full description of LossyCounting's implementation details can be found in [20].

FDPM-1 Unlike LossyCounting, the **frequent datastream pattern mining** algorithm FDPM-1 does not allow false positives, but it also has a high probability of finding **truly** frequent itemsets [32].

Users must specify parameters $0 \leq \delta \leq 1$ and $\theta \geq 1$ such that the resulting family P contains no itemset whose frequency is below the specified **support level** θ and includes any frequent $\theta-$item set with probability at least $1 - \delta$.[53]

53: FDPM-1 uses the **Chernoff bound** to achieve the probabilistic guarantee by modeling the appearance of an itemset in P as a **binomial random variable** – in each transaction, either the itemset appears or it does not. This requires the transactions to be **independent**, which is not necessarily valid in real applications.

$n \leftarrow 0$, $P \leftarrow \emptyset$;
while a new transaction t arrives **do**
  if $t \in P$ **then**
    increase t's count by 1;
  else
    if $|P| > n_0$ **then**
      calculate the running ϵ_n for the n observations;
      delete all entries in P that are potentially infrequent;
    end if
    insert t with an initial count 1 into P;
  end if
  $n \leftarrow n + 1$;
  output P on demand;
end while

Figure 28.6: Frequent datastream pattern mining FDPM-1, which produces the family P of frequent itemsets.

It can be shown that the **required number of transactions per batch** is

$$n_0 = 2(1 + \ln(2/\delta))/\theta;$$

the **running variable** is $\varepsilon_n = \sqrt{2\theta \ln(2/\delta)/n}$, where n is the number of transactions **seen so far**.

IncMine Proposed in 2008 [11], **IncMine** uses **sliding windows** to mine **frequent closed itemsets** (FCI), with a controlled number of false negatives. The algorithm can handle concept drift, has a high throughput, and only requires **modest** memory usage.

The sliding window looks at the w **most recent time units**.[54] The following illustration shows 4 time units: t_1, t_2, t_3, t_4, with 3, 2, 3, 4 transactions, respectively. Using $w = 2$, we have 3 **complete time windows** (t_1, t_2), (t_2, t_3), (t_3, t_4) over the 4 time units, containing 5, 5, 7 transactions, respectively.

54: The number of transactions arriving in each time unit may vary.

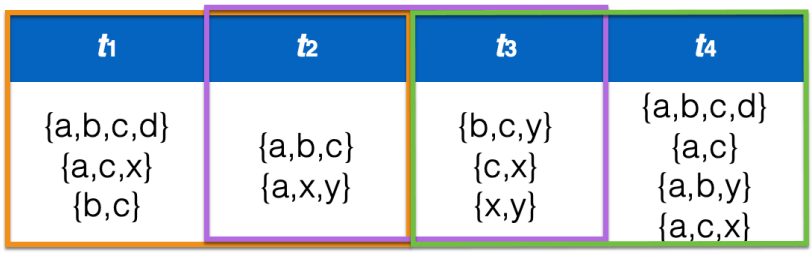


Figure 28.7: An illustration of a sliding window with $w = 2$ over 4 time units.

IncMine introduce a novel idea for controlling the number of itemsets that could eventually become frequent; the basic idea of **semi-FCI** is that an itemset whose first appearance in the window happens earlier than one with a later first appearance needs a **higher support** in order to be kept. Intuitively, if a **low support** itemset has been in the stream for a while, it is unlikely to become a **frequent** itemset.[55]

55: In other words, IncMine assessment of an itemset's support is **time-dependent**.

The set of semi-FCI is updated incrementally.[56] Briefly, consider the three semi-FCIs F, L, and C, where F is the set of semi-FCI over the **current time unit**, L is the set of semi-FCI over the **previous window**, and C is the set of semi-FCI over the **current window**. The incremental update task is to construct C by modifying L using F, dropping unpromising itemsets and adding new semi-FCI according to some protocol.

56: The details of the actual algorithm and its efficient implementation are rather technical and fall beyond our scope; details are available in [11, sec. 5].

Significantly faster processing times and smaller memory consumption have been reported, in comparison to other algorithms.[57] Of course, the No Free Lunch theorems are still in play, so *caveat emptor*.

57: IncMine was able to process up to 40,000 transactions per second for the t10i4 data stream (produced by a modified IBM data generator), outperforming LossyCounting and MOMENT by over two orders of magnitude. Memory consumption was found to be much below that of LossyCounting and MOMENT, and was relatively constant over a range of values of minimum support threshold. [11]

Mining Other Patterns Interesting patterns do not always appear as frequent subsets of a larger set. In a supermarket, for instance, we might be interested in tracking the **order** in which a customer picks up items (e.g., milk before eggs); these **precedence relations** might provide insight on merchandise layout, say. **Sequence mining** also has applications to bioinformatics, text mining, telecommunications, fraud detection, and cybersecurity.

Graph mining is another common pattern mining task; it subsumes itemset mining since itemsets can be modeled as a **complete subgraphs**. Notable graph mining algorithms include:

- **IncGraphMiner** (a variant of IncMine);
- **WinGraphMiner** (which maintains the frequent closed graphs in a fixed-size sliding window), and
- **AdaGraphMiner** (an extension of WinGraphMiner which can adapt to changes in the stream).

28.6 Examples

Various data streams mining notions discussed in this chapter are illustrated in the following examples, using R.

28.6.1 Obtaining Statistics

We got some practice doing batch analysis when working with the dailysales.csv dataset in Section 28.1.1. In this first example, we will work with the related weeklysales.csv dataset, which contains all the sales over a week.

First, we read in the data.

```
dfw <- read.csv("weeklysales.csv", stringsAsFactors = TRUE)
str(dfw)
head(dfw)
```

```
'data.frame': 154 obs. of  4 variables:
 $ sku   : Factor w/ 4 levels "A","B","C","D": 3 1 2 1 3 2 2 3 3 2 ...
 $ day   : Factor w/ 7 levels "Fri","Mon","Sat",..: 4 4 4 4 4 4 4 4 4 4 ...
 $ time  : Factor w/ 112 levels "08:25","08:30",..: 25 28 30 34 35 37 43 46 50 56 ...
 $ amount: num  9.99 12.99 4.49 12.99 9.99 ...
```

Suppose that we are interested in the most popular item sold on Sunday.

We start by isolating the Sunday sales.

```
(sun <- dfw[dfw$day=='Sun',])
```

```
   sku day  time amount
1    C Sun 10:20   9.99
2    A Sun 10:28  12.99
3    B Sun 10:43   4.49
4    A Sun 11:17  12.99
5    C Sun 11:23   9.99
6    B Sun 11:30   4.49
7    B Sun 12:09   4.49
8    C Sun 12:22   9.99
9    C Sun 12:30   9.99
10   B Sun 13:00   4.49
11   C Sun 13:12   9.99
12   C Sun 13:20   9.99
13   B Sun 13:39   4.49
14   A Sun 16:11  12.99
15   D Sun 16:27  24.99
16   A Sun 16:35  12.99
17   A Sun 16:55  12.99
18   A Sun 16:59  12.99
```

Next, we only retain the sku column.

```
df2 <- sun['sku']
```

We find the summary count statistics as follows:

```
(ag <- aggregate(df2, by=list(item = df2$sku), FUN=length))
```

```
  item sku
1    A   6
2    B   5
3    C   6
4    D   1
```

We rename the columns for consistency's sake and exhibit the most popular item(s).

```
(agsun <- setNames(ag, c('sku','count')))
agsun[agsun$count==max(agsun$count),]
```

```
  sku count
1   A     6
3   C     6
```

28.6.2 Bloom Filter

In this example, we create a Bloom filter for a list of malware URLs excerpted from malwaredomainlist.com, found in `malwaredomainlist.csv`.

Recall that such a filter can produce false positives; that is, it can flag URL that do not appear on the list of URL used to create the filter.

We start by loading the data.[58]

58: This block of code could be replaced by a scraper that collects up-to-date information from a host file containing thousands of adware/malware URL into a list of strings ready for hashing.

```
badURLs <- read.csv('malwaredomainlist.csv')
head(badURLs)
```

```
                                                URL
1                          CPROHWIN0190.locaweb.com.br
2                          PEMLINWEB133.blacknight.com
3                    Static-IP-18150024815.cable.net.co
4 a184-50-238-184.deploy.static.akamaitechnologies.com
5                                  acf113.rev.netart.pl
6                                  acs169.rev.netart.pl
```

Next, we set the filter's parameters.

```
k <- 4      # Number of hash functions
m <- 100000 # Size of the filter array
```

The next stage is to build a "helper" function that creates a hash function from a sample URL (salt). The range of output values of the created hash function is $1, \ldots, m$ (inclusively). The `digest()` function (in package `digest`) creates hash digests of arbitrary `R` objects – in this case, URL strings.

```
makehash <- function ( salt, m ) {
    function( url ) {
        1 + strtoi(substr(digest::digest(paste(url,as.character(salt)),
                                         algo='xxhash32'), 1 , 5), 16L) %% m
    }
}
```

Then we create the k hash functions, by applying `makehash` to our sample URL k times.

```
f <- lapply( seq(1:k) , function(x) {makehash(x,m)})
```

For instance, we can apply the k hash functions to the sample URL `uottawa.ca`.

```
sampleURL = 'uottawa.ca'
for (i in 1:k) print(f[[i]](sampleURL))
```

```
[1] 17275
[1] 11876
[1] 29530
[1] 18068
```

We can now construct the Bloom filter from `badURLs`.

```
bf <- as.vector(rep(FALSE, m))
for (i in 1:k) {
    hashed <- lapply( badURLs$URL, f[[i]] )
    for (j in hashed) bf[j] = TRUE
}
```

The object `bf` is a logical vector of length m. In this example, we have the following distribution.

```
table(bf)
```

```
bf
FALSE  TRUE
97757  2243
```

We write a function that uses the Bloom filter to flag bad URL.

```
isBadURL <- function( url ) {
    rv <- TRUE
    for (i in 1:k) {
        hash <- f[[i]]( url )
        if (bf[hash] == FALSE) {
            rv <- FALSE
        }
      break
    }
    rv
}
```

We can now use the Bloom filter on a list of known safe sites obtained from moz.com/top500 (stored in `moz_dot_com_top500.csv`) to determine the percentage of false positives.

```
top <- read.csv('moz_dot_com_top500.csv')
(n <- dim(top)[1]) # number of rows
head(top)
```

```
[1] 500

URL
1  facebook.com
2   twitter.com
3    google.com
4   youtube.com
5 instagram.com
6  linkedin.com
```

We can compute the number of false positives in `top` as follows:

```
falsePositive <- 0
(c <- Reduce('+', lapply( top$URL, isBadURL)))
```

```
[1] 12
```

The proportion of false positives is thus quite low:

```
c / n
```

```
[1] 0.024
```

We can also determine which safe sites were considered "bad" by the Bloom filter built from the list of malware sites.

```
index = which(lapply( top$URL, isBadURL) == TRUE)
top[index,]
```

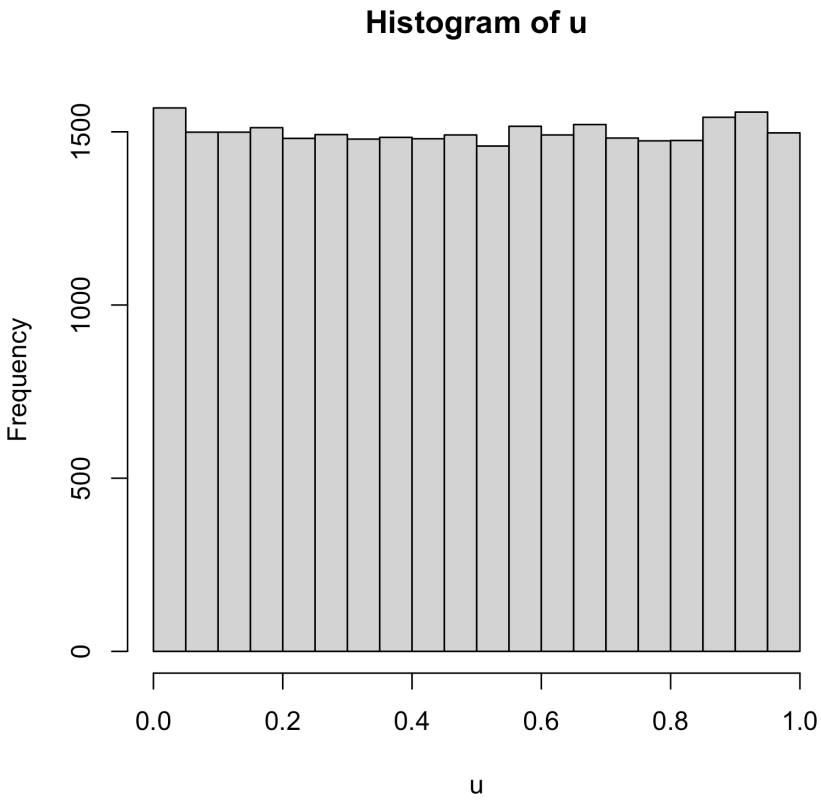
```
 [1] "wixsite.com"          "archive.org"
 [3] "time.com"             "constantcontact.com"
 [5] "webs.com"             "networksolutions.com"
 [7] "youku.com"            "yale.edu"
 [9] "loopia.com"           "mozilla.com"
[11] "allaboutcookies.org"  "dot.gov"
```

That is not great news for Yale University, not gonna lie...

28.6.3 Sampling With a Reservoir

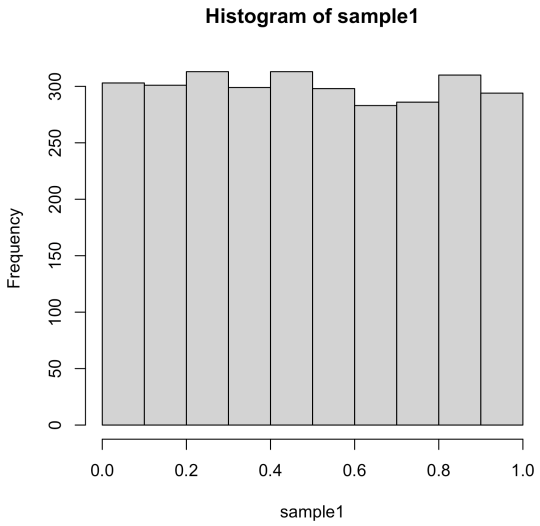
In the next two sections, we will compare sampling with reservoir and with hash. We will not be working with actual data streams, however. Instead, we will generate a long list of integers and process the numbers sequentially.

```
n <- 30000
set.seed(n) # for replicability
u <- runif( n, 0, 1 )  # uniform distribution
hist(u)
```



We see that the distribution is quite uniform. If we batch select 10% of the points of u uniformly randomly, we expect the sample distribution to also be uniform.

```
c <- as.integer(0.10 * n)
sample1 <- sample(u, c)
hist(sample1)
```



How would we implement sampling with a reservoir in R? First, we create the reservoir.

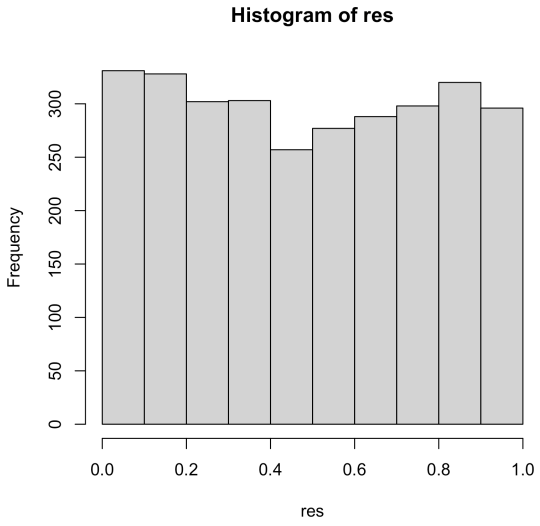
```
res <- u[1:c]
```

The reservoir sampling routine is shown below.

```
for (t in seq( c+1, length(u) )) {
    m <- sample( 1 , t , 1)
    if (m[1] <= c) {
        # select a random position in the reservoir
        # and replace with u[t]
        i <- sample( 1 , c , 1)
        u[i[1]] = u[t]
    }
}
```

Once again, the histogram shows a uniform distribution.

```
hist(res)
```



28.6.4 Sampling With a Hash Function

Sample with a hash function, we should once again expect to see a uniform sample distribution.

```
hashNumeric <- function (x) { 1 + strtoi(substr(digest::digest(as.character(x),
                                                               algo='xxhash32'),1,7),
                                         16L) %% n}
hashes <- setNames( as.data.frame(sapply(u, hashNumeric)) , c('hash') )
```

The `hashes` object is a vector with the same length as u.

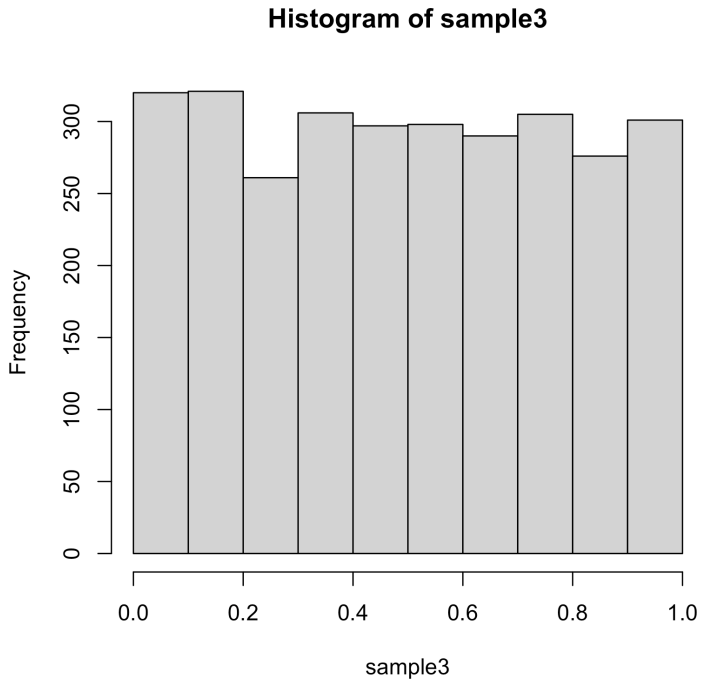
```
head(hashes)
```

```
   hash
1 28126
2 13495
3  1506
4 28238
5 15920
6 11666
```

The has sample may not necessarily contain exactly 3000 observations, however.

```
sample3 <- u[hashes$hash <= c]
length(sample3)
hist(sample3)
```

```
[1] 2975
```



28.6.5 Fading Window

We now turn to the simple fading window example discussed on pp.1828-1829 to identify recent most popular items. We start by setting the number of possible items in the data.

```
set.seed(2000) # for replicability
numItems <- 100
```

Next, we create a data "stream" of 10,000 sales.

```
p <- sort( runif( numItems-1 , 0 , 1 ))
r <- runif(5000,0,1)

findIndex <- function (x, d) {
    rv <- 1
    # Identity the interval with breakpoints given by d in which x lies
    for (i in 1:length(d)) {
        if (x > d[i]) { rv <- i+1 }
    }
    rv
}
```

The stream is built in such a way that popular items in the first half are not popular in the second half and vice versa.

```
items <- c(lapply(r, function(x) findIndex(x, p)),
           lapply(r, function(x) { numItems + 1 - findIndex(x, p) } ))
```

The top products in the first half of the stream are shown below.

```
firstHalf.df <- setNames(data.frame(unlist(items[1:length(items)/2])), c('item'))
ag1 <- setNames(aggregate(firstHalf.df, by=list(item = firstHalf.df$item), FUN=length),
                c('item', 'count'))
head(ag1[ order(-ag1$count),])
```

```
item count
  26   556
  50   550
  87   336
  90   294
  39   288
  20   286
```

The top products in the second half of the stream are shown below.

```
secondHalf.df <- setNames(data.frame(unlist(items[(length(items)/2+1):length(items)])),
                          c('item'))
ag2 <- setNames(aggregate(secondHalf.df, by=list(item = secondHalf.df$item), FUN=length),
                c('item', 'count'))
head(ag2[ order(-ag2$count),])
```

```
item count
  75   278
  51   275
  14   168
  11   147
  62   144
  81   143
```

We set up the parameters for the fading window algorithm.

```
t = 0.5  # threshold
c = 1/length(items)  # decaying factor
scores <- rep(0.0, numItems)
```

We now process the first half the data "stream" with the fading window algorithm.

```
for (i in 1:(length(items)/2)) {
    scores <- (1-c)*scores
    j <- items[[i]]
    scores[[j]] <- scores[[j]] + 1
}
```

The top scores are shown below.

```
scores.df <- setNames(data.frame(c(1:length(scores)),unlist(scores)), c('item','score'))
head(scores.df[ order(-scores.df$score),])
```

```
item    score
  26 219.6582
  50 215.2115
  87 132.1868
  90 114.2575
  20 112.9336
  39 111.5845
```

The same items appear, although the order of the 5th and 6th items has been switched. And of course, we can do the same for the second half of the "stream".

```
scores <- rep(0.0, numItems)

for (i in (length(items)/2+1):length(items)) {
    scores <- (1-c)*scores
    j <- items[[i]]
    scores[[j]] <- scores[[j]] + 1
}

scores.df <- setNames(data.frame(c(1:length(scores)),unlist(scores)), c('item','score'))
head(scores.df[ order(-scores.df$score),])
```

```
item    score
  75 219.6582
  51 215.2115
  14 132.1868
  11 114.2575
  81 112.9336
  62 111.5845
```

The same items appear, although the order of the 5th and 6th items has been switched.

28.6.6 Adaptive Sliding Window Algorithm

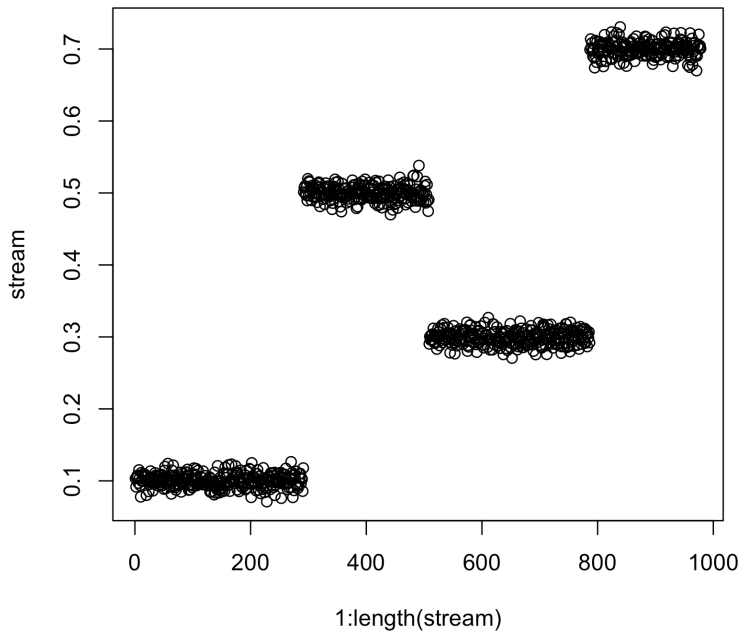
In this example, we play with the ADWIN algorithm of Section 28.2.2.

First, we create a random data "stream" with $n = 4$ concepts.

```
set.seed(0) # for replicability
nConcepts <- 4
minLen <- 150
maxLen <- 300
noiseLevel <- 0.01
len <- sample(minLen:maxLen, nConcepts)
means <- c(0.1, 0.5, 0.3, 0.7)
stream <- list()
for (i in 1:nConcepts) {
    t <- rep(means[[i]], len[[i]]) + noiseLevel * rnorm(len[[i]], 0, 1)
    stream <- c(stream, t)
}
```

We display the complete stream below.

```
plot( 1:length(stream), stream )
```



We can obtain the empirical mean of the stream along each concept.

```
means.2 <- c()
cumlen = cumsum(unlist(len))
means.2[1] <- mean(unlist(stream[1:cumlen[1]]))

for(t in 2:nConcepts){
        means.2[t] <- mean(unlist(
                        stream[cumlen[t-1]+1:cumlen[t]])
                       )
    }
means.2
```

```
[1] 0.1002887 0.3952116 0.4630066 0.7000306
```

Let's see if ADWIN can detect the different concept means. First, we implement the threshold function: $\varepsilon(W) = \sqrt{\frac{1}{2m} \ln \frac{4|W|}{\delta}}$ where W is a list of numbers (the variable-length window) and $m = \frac{2}{1/|W_0|+1/|W_1|}$. In the code below, i represents the splitting indices.

```
delta = 0.5 # A value between 0 and 1; a value of 0.5
            # favours neither FP nor FN

threshold <- function (W, i) {
    W0 <- W[1:i]
    W1 <- W[(i+1):length(W)]
    m <- 2/(1/length(W0)+1/length(W1))
    eps <- sqrt(1/(2*m)*log(4*length(W))/delta)
    eps
}
```

Next, we Initialize the window W to contain only the first observation in the stream.[59]

59: Normally, one should consider a sizeable initial chunk of the stream, as in the exercises.

```
W <- stream[1]
border <- 0

for (t in 2:length(stream)) {
    W <- c(unlist(W), stream[[t]])

    if (length(W) > 1) {
        i <- 0
        repeat {
            i <- i+1
            if (i >= length(W)) break;

            u0 <- mean(W[1:i])
            u1 <- mean(W[(i+1):length(W)])
            thd <- threshold(W, i)
            if (abs(u0 - u1) > thd) {
                border = border + i
```

```
                print(paste('Detected new mean:',
                    as.character(u1),
                    'between indices', border,
                    'and', border+1))
                W <- W[(i+1):length(W)]
                i <- 0
            }
        }
    }
}
```

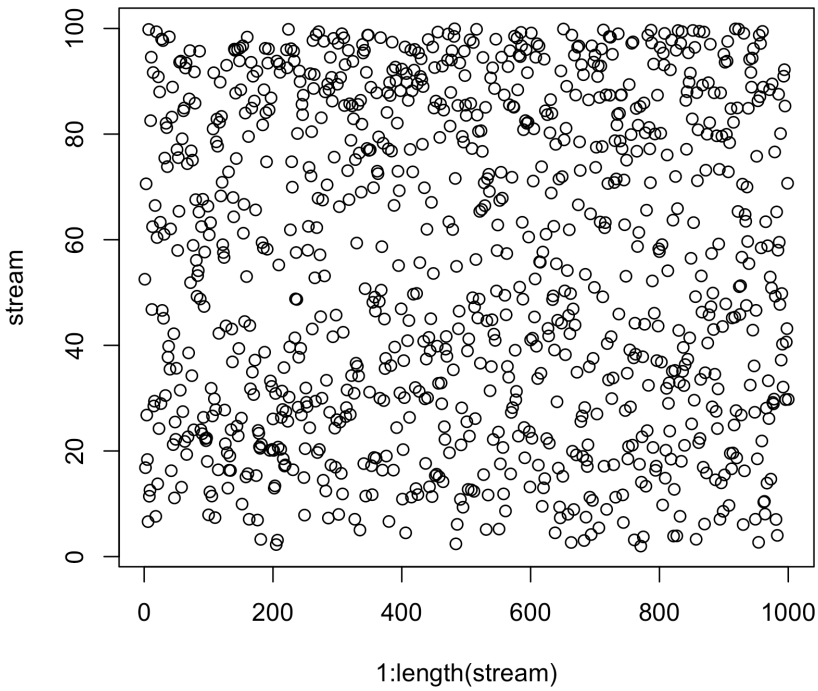
```
[1] "Detected new mean: 0.502436993351689 between indices 291 and 292"
[1] "Detected new mean: 0.299156929106440 between indices 508 and 509"
[1] "Detected new mean: 0.697717000444021 between indices 786 and 787"
```

28.6.7 Partition Incremental Discretization Algorithm

In this example, we study the mechanics of PID (of pp.1836-1838).

We first simulate a data stream of 1000 random values drawn from 3 beta distributions.

```
set.seed(1) # for replicability
n <- 1000
maxVal <- 100
stream <- sample( c(rbeta(2*n, 2, 5),
                    rbeta(2*n, 5, 1),
                    rbeta(n, 2, 2)), 1000 ,
                  replace = FALSE) * maxVal

plot( 1:length(stream), stream )
```



We construct Layer 1 from the simulated stream; for simplicity, we use intervals with integer breakpoints of width 1.[60]

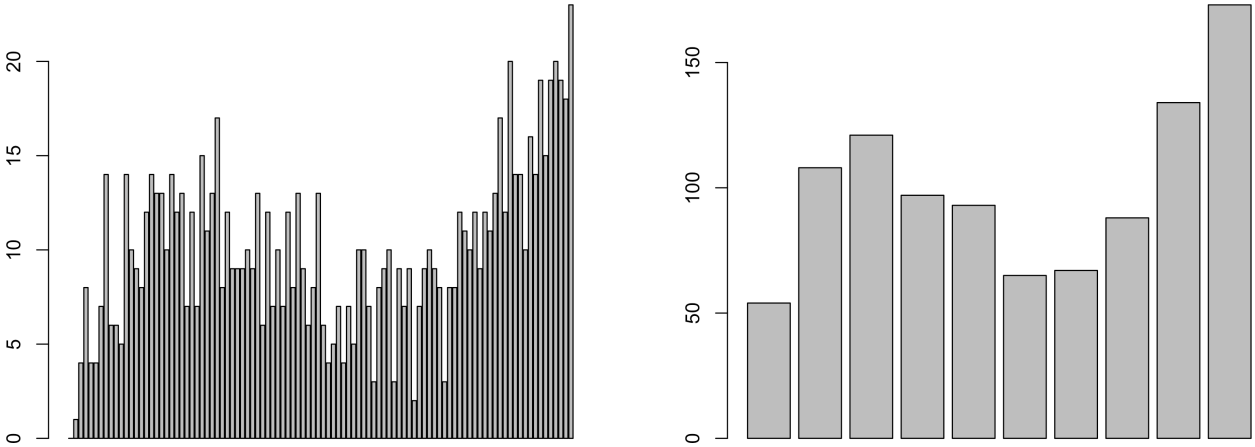
60: We could also use the split operator.

```
bins1 <- rep(0, maxVal)

for (i in 1:length(stream)) {
    # Increment the count for the bin in which the ith
    # element of stream falls
    b <- ceiling(stream[[i]])
    bins1[[b]] <- bins1[[b]] + 1
}
```

Next, we construct a $k = 10$ equal-width Layer 2 histogram from Layer 1's bin counts.[61]

61: Obtaining an equal-frequency histogram is left as an exercise.

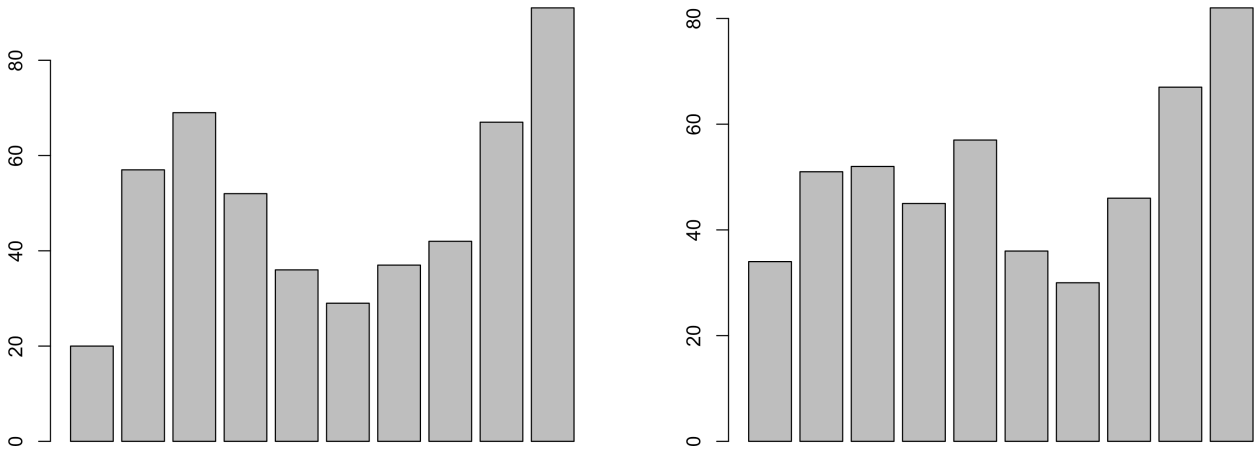
```
k <- 10
width <- ceiling (maxVal / k)
l2bins <- rep(0, k)

for (i in 1:k) {
    lb <- (i-1)*width + 1
    ub <- i*width
    for (j in lb:ub) {
        l2bins[[i]] <- l2bins[[i]] + bins[[j]]
    }
}

barplot(bins1); barplot(l2bins)
```



28.6.8 Histogram Drift

Using the same stream as in the previous section, we explore if there is a difference between the histogram for either halves of the stream.

```
bins1 <- rep(0, maxVal); bins2 <- rep(0, maxVal)

# Layer 1 - first half
for (i in 1:(length(stream)/2)) {
    # Increment the count for the bin in which the ith element of stream falls
    b <- ceiling(stream[[i]])
    bins1[[b]] <- bins1[[b]] + 1
}

# Layer 1 - second half
for (i in (length(stream)/2+1):length(stream)) {
    b <- ceiling(stream[[i]])
    bins2[[b]] <- bins2[[b]] + 1
}

# Layer 2 - both halves simultaneously
l2bins1 <- rep(0, k); l2bins2 <- rep(0, k)

for (i in 1:k) {
    lb <- (i-1)*width + 1
    ub <- i*width
    for (j in lb:ub) {
        l2bins1[[i]] <- l2bins1[[i]] + bins1[[j]]
        l2bins2[[i]] <- l2bins2[[i]] + bins2[[j]]
    }
}

barplot(l2bins1); barplot(l2bins2)
```



What about it? Are the two half-stream histograms similar or different?

28.7 Exercises

1. Find the top two items sold between 1pm and 5pm in `dailsales.csv` (note the time format in the dataset).
2. Find the most popular and the least popular items sold on each day of the week in `weeklysales.csv`.
3. Consider the Bloom filter of Section 28.6.2. Change the values of k and m to see how the rate of false positives changes. Do the results change significantly if we use updated lists of malware and safe sites? Implement your own hash functions and compare the results with those using the given hash function.
4. Recreate the examples from 28.6.3 and 28.6.4 using other distributions to generate u (such as `rnorm()`, etc.). Implement your own hash functions and compare the results with those using the given hash function.
5. We revisit the fading window example 28.6.5. How did the results from the fading window algorithm over the entire "stream" compare with the actual top items for the two halves? Apply the fading window algorithm to the entire stream instead; what can be said about the results? If we adjust the value of c (e.g., $c = 0.01$), `numItems`, and/or the number of transactions in the data "stream"; how do the results change?
6. In the ADWIN example of Section 28.6.6, instead of hardcoding the means, generate random means and see how the algorithm performs. Experiment with different values for `nConcepts`, `minLen`, `maxLen`, and `noiseLevel`. Do we encounter false positives? False negatives? As it stands, the ADWIN code above does *not* show the mean of the initial portion of the data "stream". Modify the code so that it does show the first mean.
7. In the PID example of Section 28.6.7, construct an equal-frequency Layer 2 histogram from Layer 1; experiment with different values of k, `maxVal`, and choices of distributions. Write a function that takes Layer 1 bin counts and other appropriate arguments and outputs Layer 2 bin counts.
8. Use a suitable statistical test on the two sets of bin counts to see if the histograms of Section 28.6.8 are different, with statistical significance.
9. Consider the dataset `flights1_2019_1.csv`.
    a) Transform the dataset so that the observational units are the airports and date. Create variables as needed. Assume that each new day's observations constitute a new batch. Does the stream exhibit concept drift? Maintain and display statistics and histograms for some of the variables in the stream.
    b) Conduct a streaming $k-$means clustering of the data. Play with values of k and α. Conduct a streaming two-phase clustering of the data, under the same conditions as above. Play with parameter values in CluStream and DenStream.[†] Comment on the results.

Chapter References

[1] C.C. Aggarwal, ed. *Data Classification: Algorithms and Applications*. CRC Press, 2015.

[2] C.C. Aggarwal and C.K. Reddy, eds. *Data Clustering: Algorithms and Applications*. CRC Press, 2014.

[3] C.C. Aggarwal et al. 'A framework for clustering evolving data streams'. In: *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*. 2003, pp. 81–92.

[4] C.C. Aggarwal et al. 'A framework for on-demand classification of evolving data streams'. In: *IEEE Trans. Knowl. Data Eng.* 18 (May 2006), pp. 577–589.

[5] M. Basseville, I.V. Nikiforov, et al. *Detection of Abrupt Changes: Theory and Application*. Vol. 104. Prentice Hall Englewood Cliffs, 1993.

[6] A. Bifet and R. Gavaldà. 'Adaptive learning from evolving data streams'. In: *Advances in Intelligent Data Analysis VIII*. Ed. by Niall M. Adams et al. Springer Berlin Heidelberg, 2009, pp. 249–260.

[7] A. Bifet et al. *Machine Learning for Data Streams: with Practical Examples in MOA*. Adaptive Computation and Machine Learning series. MIT Press, 2018.

[†] You may need to consult appropriate references for the algorithm details and implementation.

[8] P. Boily. *Data Understanding, Data Analysis, and Data Science (Course Notes)*, *volumes 1-5*. QED/Idlewyld, 2026.

[9] P. Boily, S. Davies, and J. Schellinck. *The Practice of Data Visualization*. Data Action Lab, 2023.

[10] F. Cao et al. 'Density-based clustering over an evolving data stream with noise'. In: *Proceedings of the 2006 SIAM International Conference on Data Mining (SDM)*, pp. 328–339.

[11] J. Cheng, Y. Ke, and W. Ng. 'Maintaining frequent closed itemsets over a sliding window'. In: *J. Intell. Inf. Syst.* 31.3 (Dec. 2008), pp. 191–215.

[12] P. Domingos and G. Hulten. 'Mining high-speed data streams'. In: *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA: Association for Computing Machinery, 2000, pp. 71–80.

[13] B. Dykes. 'Big Data: Forget Volume and Variety, Focus On Velocity'. In: *Forbes* (June 2017).

[14] R. El Sibai et al. 'An in-depth analysis of CUSUM algorithm for the detection of mean and variability deviation in time series'. In: *Web and Wireless Geographical Information Systems*. Ed. by M. R. Luaces and F. Karimipour. Springer, 2018, pp. 25–40.

[15] J. Gama. *Knowledge Discovery from Data Streams*. Chapman & Hall/CRC Data Mining and Knowledge Discovery Series. CRC Press, 2010.

[16] V. Kanaujia and S. Chakraborty. 'Exploring probabilistic data structures: Bloom filters'. In: *OpenSourceForU* (May 2018).

[17] M. Lazarescu, S. Venkatesh, and H. Bui. 'Using multiple windows to track concept drift'. In: *Intell. Data Anal.* 8 (Mar. 2004), pp. 29–59.

[18] V.E. Lee, R. Jin, and G. Agrawal. 'Frequent pattern mining in data streams'. In: *Frequent Pattern Mining*. Ed. by Charu C. Aggarwal. Springer International Publishing, 2007, pp. 199–224.

[19] J. Leskovec, A. Rajamaran, and J.D. Ullman. *Mining of Massive Datasets*. Cambridge Press, 2014.

[20] G.S. Manku. 'Frequent Itemset Mining over Data Streams'. In: *Data Stream Management - Processing High-Speed Data Streams*. Ed. by Minos N. Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Data-Centric Systems and Applications. Springer, 2016, pp. 209–219.

[21] G.S. Manku and R. Motwani. 'Approximate frequency counts over data streams'. In: *Proceedings of the 28th International Conference on Very Large Data Bases*. VLDB Endowment, 2002, pp. 346–357.

[22] M. Miner and A. Shook. *MapReduce Design Patterns*. O'Reilly Media, Inc., 2012.

[23] M. Stirling et al. 'Concept drift detector selection for Hoeffding adaptive trees'. In: *AI 2018: Advances in Artificial Intelligence*. Ed. by Tanja Mitrovic, Bing Xue, and Xiaodong Li. Springer International Publishing, 2018, pp. 730–736.

[24] W.N. Street and Y.S. Kim. 'A streaming ensemble algorithm (SEA) for large-scale classification'. In: *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Association for Computing Machinery, 2001, pp. 377–382.

[25] J. Tenenbaum. *Computational Cognitive Science Group at MIT*.

[26] H. Wang et al. 'Mining concept-drifting data streams using ensemble classifiers'. In: *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Association for Computing Machinery, 2003, pp. 226–235.

[27] G. Widmer and M. Kubat. 'Learning in the presence of concept drift and hidden contexts'. In: *Machine Learning* 23.1 (1996), pp. 69–101.

[28] Wikipedia. *Bloom filter*.

[29] Wikipedia. *Kolmogorov-Smirnov test*.

[30] Wikipedia. *Kullback-Leibler divergence*.

[31] Wikipedia. *Online algorithm*.

[32] Jeffery Xu Yu et al. 'False positive or false negative: mining frequent itemsets from high speed transactional data streams'. In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*. VLDB Endowment, 2004, pp. 204–215.