Chapter 12

Mining Data Streams

"*You never step into the same stream twice.*"—Heraclitus

12.1 Introduction

Advances in hardware technology have led to new ways of collecting data at a more rapid rate than before. For example, many transactions of everyday life, such as using a credit card or a phone, lead to automated data collection. Similarly, new ways of collecting data, such as wearable sensors and mobile devices, have added to the deluge of dynamically available data. An important assumption in these forms of data collection is that the data *continuously accumulate over time at a rapid rate*. These dynamic data sets are referred to as *data streams*.

A key assumption in the streaming paradigm is that it is no longer possible to store all the data because of resource constraints. While it is possible to archive such data using distributed "big data" frameworks, this approach comes at the expense of enormous storage costs and the loss of real-time processing capabilities. In many cases, such frameworks are not practical because of high costs and other analytical considerations. The streaming framework provides an alternative approach, where real-time analysis can often be performed with carefully designed algorithms, without a significant investment in specialized infrastructure. Some examples of application domains relevant to streaming data are as follows:

1. *Transaction streams:* Transaction streams are typically created by customer buying activity. An example is the data created by using a credit card, point-of-sale transaction at a supermarket, or the online purchase of an item.

2. *Web click-streams:* The activity of users at a popular Web site creates a Web click stream. If the site is sufficiently popular, the rate of generation of the data may be large enough to necessitate the need for a streaming approach.

C. C. Aggarwal, *Data Mining: The Textbook*, DOI 10.1007/978-3-319-14142-8_12

3. *Social streams:* Online social networks such as Twitter continuously generate massive text streams because of user activity. The speed and volume of the stream typically scale superlinearly with the number of actors in the social network.

4. *Network streams:* Communication networks contain large volumes of traffic streams. Such streams are often mined for intrusions, outliers, or other unusual activity.

Data streams present a number of unique challenges because of the processing constraints associated with the large volumes of continuously arriving data. In particular, data streaming algorithms typically need to operate under the following constraints, at least a few of which are always present, whereas others are occasionally present:

1. *One-pass constraint:* Because volumes of data are generated continuously and rapidly, it is assumed that the data can be processed *only once*. This is a hard constraint in all streaming models. The data are almost never assumed to be archived for future processing. This has significant consequences for algorithmic development in streaming applications. In particular, many data mining algorithms are inherently iterative and require multiple passes over the data. Such algorithms need to be appropriately modified to be usable in the context of the streaming model.

2. *Concept drift:* In most applications, the data may evolve over time. This means that various statistical properties, such as correlations between attributes, correlations between attributes and class labels, and cluster distributions may change over time. This aspect of data streams is almost always present in practical applications, but is not necessarily a universal assumption for all algorithms.

3. *Resource constraints:* The data stream is typically generated by an external process, over which a user may have very little control. Therefore, the user also has little control over the arrival rate of the stream. In cases, where the arrival rates vary with time, it may be difficult to execute online processing continuously during peak periods. In these cases, it may be necessary to drop tuples that cannot be processed in a timely fashion. This is referred to as *loadshedding*. Even though resource constraints are almost universal to the streaming paradigm, surprisingly few algorithms incorporate them.

4. *Massive-domain constraints:* In some cases, when the attribute values are discrete, they may have a large number of distinct values. For example, consider a scenario, where analysis of pairwise communications in an e-mail network is desired. The number of distinct pairs of e-mail addresses in an e-mail network with 10^8 participants is of the order of 10^{16}. When expressed in terms of required storage, the number of possibilities easily exceeds the *petabyte* order. In such cases, storing even simple statistics such as the counts or the number of distinct stream elements becomes very challenging. Therefore, a number of specialized data structures for synopsis construction of massive-domain data streams have been designed.

Because of the large volume of data streams, virtually all streaming methods use an online synopsis construction approach in the mining process. The basic idea is to create an online synopsis that is then leveraged for mining. Many different kinds of synopsis can be constructed depending upon the application at hand. The nature of a synopsis highly influences the type of insights that can be mined from it. Some examples of synopsis structures include random samples, bloom filters, sketches, and distinct element-counting data structures. In addition, some traditional data mining applications, such as clustering, can be leveraged to create effective synopses from the data.

This chapter is organized as follows. Section 12.2 introduces various types of synopsis construction methods for data streams. Section 12.3 discusses frequent pattern mining methods for data streams. Clustering methods are discussed in Sect. 12.4, and outlier analysis methods are discussed in Sect. 12.5. Classification methods are introduced in Sect. 12.6. Section 12.7 gives a summary.

12.2 Synopsis Data Structures for Streams

A wide variety of synopsis data structures have been designed for different applications. Synopsis data structures are of two types:

1. *Generic:* In this case, the synopsis can be used for most applications directly. The only such synopsis is a random sample of the data points, although it cannot be used for some applications such as distinct element counting. In the context of data streams, the process of maintaining a random sample from the data is also referred to as *reservoir sampling*.

2. *Specific:* In this case, the synopsis is designed for a specific task, such as frequent element counting or distinct element counting. Examples of such data structures include the Flajolet–Martin data structure for distinct element counting, and sketches for frequent element counting or moment computation.

In the following, different types of synopsis structures will be discussed.

12.2.1 Reservoir Sampling

Sampling is one of the most flexible methods for stream summarization. The main advantage of sampling over other synopsis data structures is that it can be used for an arbitrary application. After a sample of points has been drawn from the data, virtually any offline algorithm can be applied to the sample. By default, sampling should be considered the method of choice in streaming scenarios, although it does have limitations for a small number of applications such as distinct-element counting. In the context of data streams, the methodology used to maintain a dynamic sample from the data is referred to as *reservoir sampling*. The resulting sample is referred to as a *reservoir sample*. The method of reservoir sampling is introduced briefly in Sect. 2.4.1.2 of Chap. 2.

The streaming scenario creates some interesting challenges for a simple problem such as sampling. The challenge arises from the fact that one cannot store the entire stream on disk for sampling. In reservoir sampling, the goal is to *continuously* maintain a *dynamically updated* sample of k points from a data stream without explicitly storing the stream on disk at any given point in time. Therefore, for each incoming data point in the stream, one must use a set of efficiently implementable operations to *maintain* the sample. In the static case, the probability of including a data point in the sample is k/n, where k is the sample size, and n is the number of points in the data set. In the streaming scenario, the "data set" is not static, and the value of n continually increases with time. Furthermore, previously arrived data points, which are not included in the sample, have been irrevocably lost. Thus, the sampling approach works with *incomplete knowledge* about the previous history of the stream at any given moment in time. In other words, for each incoming data point in the stream, one needs to make two simple *admission control* decisions dynamically:

1. What sampling rule should be used to decide whether to include the incoming data point in the sample?

2. What rule should be used to decide how to eject a data point from the sample to "make room" for the newly inserted data point?

The reservoir sampling algorithm proceeds as follows. For a reservoir of size k, the first k data points in the stream are always included in the reservoir. Subsequently, for the nth incoming stream data point, the following two admission control decisions are applied.

1. Insert the nth incoming stream data point in the reservoir with probability k/n.
2. If the newly incoming data point was inserted, then eject one of the old k data points in the reservoir at random to make room for the newly arriving point.

It can be shown that the aforementioned rule maintains an unbiased reservoir sample from the data stream.

Lemma 12.2.1 *After n stream points have arrived, the probability of any stream point being included in the reservoir is the same, and is equal to k/n.*

Proof: This result is easy to show by induction. At initialization of the first k data points, the theorem is trivially true. Let us (inductively) assume that it is also true after $(n-1)$ data points have been received. Therefore, the probability of each point being included in the reservoir is $k/(n-1)$. The lemma is trivially true for the arriving data point because the probability of its being included in the stream is k/n. It remains to prove the result for the remaining points in the data stream. There are two *disjoint* case events that can arise for an incoming data point, and the final probability of a point being included in the reservoir is the sum of these two cases:

I: The incoming data point is not inserted into the reservoir. The probability of this is $(n-k)/n$. Because the original probability of any point being included in the reservoir by the inductive assumption, is $k/(n-1)$, the overall probability of a point being included in the reservoir *and* the occurrence of the Case I event, is the multiplicative value of $p_1 = \frac{k(n-k)}{n(n-1)}$.

II: The incoming data point is inserted into the reservoir. The probability of Case II is equal to insertion probability k/n of incoming data points. Subsequently, existing reservoir points are retained with probability $(k-1)/k$ because exactly one of them is ejected. Because the inductive assumption implies that any of the earlier points in the stream was originally present in the reservoir with probability $k/(n-1)$, it implies that the probability of a point being included in the reservoir *and* Case II event is given by the product p_2 of the three aforementioned probabilities:

$$p_2 = \left(\frac{k}{n}\right)\left(\frac{k-1}{k}\right)\left(\frac{k}{n-1}\right) = \frac{k(k-1)}{n(n-1)} \tag{12.1}$$

Therefore, the total probability of a stream point being retained in the reservoir after the nth data point arrival is given by the sum of p_1 and p_2. It can be shown that this is equal to k/n. ■

The major problem with this approach is that it cannot handle concept drift because the data is uniformly sampled without decay.

12.2.1.1 Handling Concept Drift

In streaming scenarios, recent data are generally considered more important than older data. This is because the data generating process may change over time, and the older data are often considered "stale" from the perspective of analytical insights. A uniform random sample from the reservoir will contain data points that are distributed uniformly over time. Typically, most streaming applications use a decay-based framework to regulate the relative importance of data points, so that more recent data points have a higher probability to be included in the sample. This is achieved with the use of a *bias function*.

The bias function associated with the rth data point, at the time of arrival of the nth data point, is given by $f(r, n)$. This function is related to the probability $p(r, n)$ of the rth data point belonging to the reservoir at the time of arrival of the nth data point. In other words, the value of $p(r, n)$ is proportional to $f(r, n)$. It is reasonable to assume that the function $f(r, n)$ decreases monotonically with n (for fixed r), and increases monotonically with r (for fixed n). In other words, recent data points have a higher probability of belonging to the reservoir. This kind of sampling will result in a *bias-sensitive sample* $\mathcal{S}(n)$ of data points.

Definition 12.2.1 *Let $f(r, n)$ be the bias function for the rth point at the time of arrival of the nth point. A* biased *sample $\mathcal{S}(n)$ at the time of arrival of the nth point in the stream is defined as a sample such that the relative probability $p(r, n)$ of the rth point belonging to the sample $S(n)$ (of size n) is proportional to $f(r, n)$.*

In general, it is an open problem to perform reservoir sampling with arbitrary bias functions. However, methods exist for the case of the commonly used *exponential bias* function:

$$f(r, n) = e^{-\lambda(n-r)} \tag{12.2}$$

The parameter λ defines the bias rate and typically lies in the range $[0, 1]$. In general, this parameter λ is chosen in an application-specific way. A choice of $\lambda = 0$ represents the unbiased case. The exponential bias function defines the class of *memoryless functions* in which the future probability of retaining a current point in the reservoir is independent of its past history or arrival time. It can be shown that this problem is interesting only in space-constrained scenarios, where the size of the reservoir k is strictly less than $1/\lambda$. This is because it can be shown [35] that an exponentially biased sample from a stream of infinite length, will not exceed $1/\lambda$ in expected size. This is also referred to as the *maximum reservoir requirement*. The following discussion is based on the assumption that $k < 1/\lambda$.

The algorithm starts with an empty reservoir. The following replacement policy is used to fill up the reservoir. Assume that at the time of (just before) the arrival of the nth point, the fraction of the reservoir filled is $F(n) \in [0, 1]$. When the $(n + 1)$th point arrives, it is inserted into the reservoir with insertion probability[1] $\lambda \cdot k$. However, one of the old points in the reservoir is not necessarily deleted because the reservoir is only partially full. A coin is flipped with the success probability $F(n)$. In the event of a success, one of the points in the reservoir is randomly selected and replaced with the incoming $(n + 1)$th point. In the event of a failure, there is no deletion, and the $(n + 1)$th point is added to the reservoir. In the latter case, the number of points in the reservoir (the current sample size) increases by 1. In this approach, the reservoir fills up fast early in the process, but then levels off, as it reaches near its capacity. The reader is referred to the bibliographic notes for the proof of correctness of this approach. A variant of this approach that fills up the reservoir even faster is also discussed in the same work.

[1] This value is always at most 1, because $k < 1/\lambda$.

12.2.1.2 Useful Theoretical Bounds for Sampling

While the reservoir method provides data samples, it is often desirable to obtain quality bounds on the results obtained with the use of the sampling process. A common application of sampling is the estimation of statistical aggregates with the reservoir sample. The accuracy of these aggregates is often quantified with the use of *tail inequalities*.

The simplest tail inequality is the *Markov inequality*. This inequality is defined for probability distributions that take on only nonnegative values. Let X be a random variable with the probability distribution $f_X(x)$, a mean of $E[X]$, and a variance of $Var[X]$.

Theorem 12.2.1 (Markov Inequality) *Let X be a random variable that takes on only nonnegative random values. Then, for any constant α satisfying $E[X] < \alpha$, the following is true:*

$$P(X > \alpha) \leq E[X]/\alpha \tag{12.3}$$

Proof: Let $f_X(x)$ represent the density function for the random variable X. Then, we have:

$$\begin{aligned} E[X] &= \int_x x f_X(x) \mathrm{d}x \\ &= \int_{0 \leq x \leq \alpha} x f_X(x) \mathrm{d}x + \int_{x > \alpha} x f_X(x) \mathrm{d}x \\ &\geq \int_{x > \alpha} x f_X(x) \mathrm{d}x \\ &\geq \int_{x > \alpha} \alpha f_X(x) \mathrm{d}x. \end{aligned}$$

The first inequality follows from the nonnegativity of x, and the second follows from the fact that the integral is computed only over cases where $x > \alpha$. The term on the right-hand side of the last line is exactly equal to $\alpha P(X > \alpha)$. Therefore, the following is true:

$$E[X] \geq \alpha P(X > \alpha) \tag{12.4}$$

The above inequality can be rearranged to obtain the final result. ■

The Markov inequality is defined only for probability distributions of nonnegative values and provides a bound only on the upper tail. In practice, it is often desired to bound both tails of probability distributions over both positive and negative values.

Consider the case where X is a random variable that is not necessarily nonnegative. The *Chebychev inequality* is a useful approach to derive symmetric tail bounds on X. The Chebychev inequality is a direct application of the Markov inequality to a nonnegative (square deviation-based) derivative of X.

Theorem 12.2.2 (Chebychev Inequality) *Let X be an arbitrary random variable. Then, for any constant α, the following is true:*

$$P(|X - E[X]| > \alpha) \leq Var[X]/\alpha^2 \tag{12.5}$$

Proof: The inequality $|X - E[X]| > \alpha$ is true if and only if $(X - E[X])^2 > \alpha^2$. By defining $Y = (X - E[X])^2$ as a (nonnegative) derivative random variable from X, it is easy to see that $E[Y] = Var[X]$. Then, the expression on the left-hand side of the theorem statement is the same as determining the probability $P(Y > \alpha^2)$. By applying the Markov inequality to the random variable Y, one can obtain the desired result. ■

The main trick used in the aforementioned proof was to apply the Markov inequality to a nonnegative function of the random variable. This technique can generally be very useful for proving other kinds of bounds, when the distribution of X has a specific form (such as

the sum of Bernoulli random variables). In such cases, a parameterized function of the random variable can be used to obtain a parameterized bound. The underlying parameter can then be optimized for the tightest possible bound. Several well-known bounds such as the Chernoff bound and the Hoeffding inequality are derived with the use of this approach. Such bounds are significantly tighter than the (much weaker) Markov and Chebychev inequalities. This is because the parameter optimization process implicitly creates a bound that is optimized for the special form of the corresponding probability distribution of the random variable X.

Many practical scenarios can be captured with the use of special families of random variables. A particular case is one in which a random variable X may be expressed as a sum of other independent bounded random variables. For example, consider the case where the data points have binary class labels associated with them and one wishes to use a stream sample to estimate the fraction of examples belonging to each class. While the fraction of points in the sample belonging to a class provides an estimate, how can one bound the probabilistic accuracy of this bound? Note that the estimated fraction can be expressed as a (scaled) sum of independent and identically distributed (i.i.d.) binary random variables, depending on the binary class associated with each sample instance. The *Chernoff bound* provides an excellent bound on the accuracy of the estimate.

A second example is one where the underlying random variables are not necessarily binary, but are bounded. For example, consider the case where the stream data points correspond to individuals of a particular age. The average age of the individuals is estimated with the use of an average of the points in the reservoir sample. Note that the age can be (realistically) assumed to be a *bounded* random variable from the range $(0, 125)$. In such cases, the *Hoeffding* bound can be used to determine a tight bound on the estimate.

First, the Chernoff bound will be introduced. Because the expressions for the lower tail and upper tail are slightly different, they will be addressed separately. The lower-tail Chernoff bound is introduced below.

Theorem 12.2.3 (Lower-Tail Chernoff Bound) *Let X be a random variable that can be expressed as the sum of n independent binary (Bernoulli) random variables, each of which takes on the value of 1 with probability p_i.*

$$X = \sum_{i=1}^{n} X_i$$

Then, for any $\delta \in (0, 1)$, we can show the following:

$$P(X < (1-\delta)E[X]) < e^{-E[X]\delta^2/2}, \tag{12.6}$$

where e is the base of the natural logarithm.

Proof: The first step is to show the following inequality:

$$P(X < (1-\delta)E[X]) < \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{E[X]} \tag{12.7}$$

The *unknown* parameter $t > 0$ is introduced to create a parameterized bound. The lower-tail inequality of X is converted into an upper-tail inequality on e^{-tX}. This can be bounded by the Markov inequality, and it provides a bound that is a function of t. This function of

t can be optimized, to obtain the tightest possible bound. By using the Markov inequality on the exponentiated form, the following can be derived:

$$P(X < (1-\delta)E[X]) \leq \frac{E[e^{-tX}]}{e^{-t(1-\delta)E[X]}}.$$

By expanding $X = \sum_{i=1}^{n} X_i$ in the exponent, the following can be obtained:

$$P(X < (1-\delta)E[X]) \leq \frac{\prod_i E[e^{-tX_i}]}{e^{-t(1-\delta)E[X]}}. \tag{12.8}$$

The aforementioned simplification uses the fact that the expectation of the product of independent variables is equal to the product of the expectations. Because each X_i is Bernoulli, the following can be shown by summing up the probabilities over the cases where $X_i = 0$ and 1, respectively:

$$E[e^{-tX_i}] = 1 + E[X_i](e^{-t} - 1) < e^{E[X_i](e^{-t}-1)}.$$

The second inequality follows from the polynomial expansion of $e^{E[X_i](e^{-t}-1)}$. By substituting this inequality back into Eq. 12.8, and using $E[X] = \sum_i E[X_i]$, the following may be obtained:

$$P(X < (1-\delta)E[X]) \leq \frac{e^{E[X](e^{-t}-1)}}{e^{-t(1-\delta)E[X]}}.$$

The expression on the right is true for any value of $t > 0$. It is desired to pick a value t that provides the *tightest possible* bound. Such a value of t may be obtained by using the standard optimization process of using the derivative of the expression with respect to t. It can be shown by working out the details of this optimization process that the optimum value of $t = t^*$ is as follows:

$$t^* = \ln(1/(1-\delta)). \tag{12.9}$$

By using this value of t^* in the inequality above, it can be shown to be equivalent to Eq. 12.7. This completes the first part of the proof.

The first two terms of the Taylor expansion of the logarithmic term in $(1-\delta)\ln(1-\delta)$ can be expanded to show that $(1-\delta)^{(1-\delta)} > e^{-\delta+\delta^2/2}$. By substituting this inequality in the denominator of Eq. 12.7, the desired result is obtained. ■

A similar result for the upper-tail Chernoff bound may be obtained that has a slightly different form.

Theorem 12.2.4 (Upper-Tail Chernoff Bound) *Let X be a random variable that can be expressed as the sum of n independent binary (Bernoulli) random variables, each of which takes on the value of 1 with probability p_i.*

$$X = \sum_{i=1}^{n} X_i.$$

Then, for any $\delta \in (0, 2e-1)$, the following is true:

$$P(X > (1+\delta)E[X]) < e^{-E[X]\delta^2/4}, \tag{12.10}$$

where e is the base of the natural logarithm.

Proof: The first step is to show the following inequality:

$$P(X > (1+\delta)E[X]) < \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{E[X]}. \tag{12.11}$$

As before, this can be done by introducing the unknown parameter $t > 0$, and converting the upper-tail inequality on X into that on e^{tX}. This can be bounded by the Markov inequality, and it provides a bound that is a function of t. This function of t can be optimized to obtain the tightest possible bound.

It can be further shown by algebraic simplification that the inequality in Eq. 12.11 provides the desired result, when $\delta \in (0, 2e-1)$. ■

Next, the Hoeffding inequality will be introduced. The Hoeffding inequality is a more general tail inequality than the Chernoff bound because it does not require the underlying data values to be Bernoulli. In this case, the ith data value needs to be drawn from *the bounded interval* $[l_i, u_i]$. The corresponding probability bound is expressed in terms of the parameters l_i and u_i. Thus, the scenario for the Chernoff bound is a special case of that for the Hoeffding's inequality. We state the Hoeffding inequality below, for which both the upper- and lower-tail inequalities are identical.

Theorem 12.2.5 (Hoeffding Inequality) *Let X be a random variable that can be expressed as the sum of n independent random variables, each of which is bounded in the range $[l_i, u_i]$.*

$$X = \sum_{i=1}^{n} X_i.$$

Then, for any $\theta > 0$, the following can be shown:

$$P(X - E[X] > \theta) \leq e^{-\frac{2\theta^2}{\sum_{i=1}^{n}(u_i - l_i)^2}} \tag{12.12}$$

$$P(E[X] - X > \theta) \leq e^{-\frac{2\theta^2}{\sum_{i=1}^{n}(u_i - l_i)^2}} \tag{12.13}$$

Proof: The proof for the upper tail will be briefly described here. The proof of the lower-tail inequality is identical. For an unknown parameter t, the following is true:

$$P(X - E[X] > \theta) = P(e^{t(X-E[X])} > e^{t\theta}) \tag{12.14}$$

The Markov inequality can be used to show that the right-hand probability is at most $E[e^{(X-E[X])}]e^{-t\theta}$. The expression within $E[e^{(X-E[X])}]$ can be expanded in terms of the individual components X_i. Because the expectation of the product is equal to the product of the expectations of independent random variables, the following can be shown:

$$P(X - E[X] > \theta) \leq e^{-t\theta} \prod_i E[e^{t(X_i - E[X_i])}]. \tag{12.15}$$

The key is to show that the value of $E[e^{t(X_i - E[X_i])}]$ is at most equal to $e^{t^2(u_i - l_i)^2/8}$. This can be shown with the use of an argument that uses the convexity of the exponential function $e^{t(X_i - E[X_i])}$ in conjunction with Taylor's theorem (see Exercise 12).

Therefore, the following is true:

$$P(X - E[X] > \theta) \leq e^{-t\theta} \prod_i e^{t^2(u_i - l_i)^2/8}. \tag{12.16}$$

Table 12.1: Comparison of different methods used to bound tail probabilities

Result	Scenario	Strength
Chebychev	Any random variable	Weak
Markov	Nonnegative random variable	Weak
Hoeffding	Sum of independent bounded random variables	Strong
Chernoff	Sum of independent Bernoulli variables	Strong

This inequality holds for any nonnegative value of t. Therefore, to find the tightest bound, the value of t that minimizes the right-hand side of the above equation needs to be determined. The optimal value of $t = t^*$ can be shown to be:

$$t^* = \frac{4\theta}{\sum_{i=1}^{n}(u_i - l_i)^2}. \tag{12.17}$$

By substituting the value of $t = t^*$ in Eq. 12.16, the desired result may be obtained. The lower-tail bound may be derived by applying the aforementioned steps to $P(E[X] - X > \theta)$ rather than $P(X - E[X] > \theta)$. ■

Thus, the different inequalities may apply to scenarios of different generality, and they may also have different levels of strength. These different scenarios are illustrated in Table 12.1.

12.2.2 Synopsis Structures for the Massive-Domain Scenario

As discussed in the introduction, many streaming applications contain discrete attributes, whose domain is drawn on a large number of distinct values. A classical example would be the value of the IP address in a network stream, or the e-mail address in an e-mail stream. Such scenarios are more common in data streams, simply because the massive number of data items in the stream are often associated with discrete identifiers of different types. E-mail addresses and IP addresses are examples of such identifiers. The streaming objects are often associated with *pairs* of identifiers. For example, each element of an e-mail stream may have both a sender and recipient. In some applications, it may be desirable to store statistics using pairwise identifiers, and therefore the pairwise combination is treated as a single integrated attribute. The *domain of possible values* can be rather large. For example, for an e-mail application with over a hundred million different senders and receivers, the number of possible pairwise combinations is 10^{16}. In such cases, *even storing simple summary statistics such as set membership, frequent item counts, and distinct element counts becomes more challenging from the perspective of space constraints.*

If the number of distinct elements were small, one might simply use an array, and update the counts in these arrays in order to create an effective summary. Such a summary could address all the aforementioned queries. However, such an approach would not be practical in the massive-domain scenario because an array with 10^{16} elements would require more than 10 petabytes. Furthermore, for many queries, such as set membership and distinct element counting, a reservoir sample would not work well. This is because the vast majority of the stream may contain infrequent elements, and the reservoir will disproportionately overrepresent the frequent elements for queries that are agnostic to the absolute frequency. Set membership and distinct-element counting are examples of such queries.

It is often difficult to create a single synopsis structure that can address all queries. Therefore, different synopsis structures are designed for different classes of queries. In the

following, a number of different synopsis structures will be described. Each of these synopsis structures is optimized to different kinds of queries. For each of the synopsis structures discussed in this chapter, the relevant queries and query-processing approach will also be described.

12.2.2.1 Bloom Filter

Bloom filters are designed for set-membership queries of discrete elements. The set-membership query is as follows:

Given a particular element, has it ever occurred in the data stream?

Bloom filters provide a way to maintain a synopsis of the stream, so that this query can be resolved with a probabilistic bound on the accuracy. One property of this data structure is that false positives are possible, but false negatives are not. In other words, if the bloom filter reports that an element does not belong to the stream, then this will always be the case. Bloom filters are referred to as "filters" because they can be used for making important selection decisions in a stream in real time. This is because the knowledge of membership of an item in a set of stream elements plays an important role in filtering decisions, such as the removal of duplicates. This will be discussed in more detail later. First, the simple case of stream membership queries will be discussed.

A bloom filter is a binary bit array of length m. Thus, the space requirement of the bloom filter is $m/8$ bytes. The elements of the bit array are indexed starting with 0 and ending at $(m-1)$. Therefore, the index range is $\{0, 1, 2, \ldots m-1\}$. The bloom filter is associated with a set of w *independent* hash functions denoted by $h_1(\cdot) \ldots h_w(\cdot)$. The argument of each of these hash functions is an element of the data stream. The hash function maps uniformly at random to an integer in the range $\{0 \ldots m-1\}$.

Consider a stream of discrete elements. These discrete elements could be e-mail addresses (either individually or sender–receiver pairs), IP addresses, or another set of discrete values drawn on a massive domain of possible values. The bits on the bloom filter are used to keep track of the distinct values encountered. The hash functions are used to map the stream elements to the bits in the bloom filter. For the following discussion, it will be assumed that the bloom filter data structure is denoted by $\mathcal{B}$.

The bloom filter is constructed from a stream $\mathcal{S}$ of values as follows. All bits in the bloom filter are initialized to 0. For each incoming stream element x, the functions $h_1(x) \ldots h_w(x)$ are applied to it. For each $i \in \{1 \ldots w\}$, the element $h_i(x)$ in the bloom filter is set to 1. In many cases, the value of some of these bits might already be 1. In such cases, the value does not need to be changed. A pictorial representation of the bloom filter and the update process is illustrated in Fig. 12.1. The pseudocode for the overall update process is illustrated in Fig. 12.2. In the pseudocode, the stream is denoted by $\mathcal{S}$, and the bloom filter data structure is denoted by $\mathcal{B}$. The input parameters include the size of the bloom filter m, and the number of hash functions w. It is important to note that multiple elements can map onto the same bit in the bloom filter. This is referred to as a *collision*. As discussed later, collisions may lead to false positives in set-membership checking.

The bloom filter can be used to check for membership of an item y in the data stream. The first step is to compute the hash functions $h_1(y) \ldots h_w(y)$. Then, it is checked whether the $h_i(y)$th element is 1. If at least one of the these values is 0, we are *guaranteed* that the element has not occurred in the data stream. This is because, if that element had occurred in the stream, the entry would have already been set to 1. Thus, false negatives

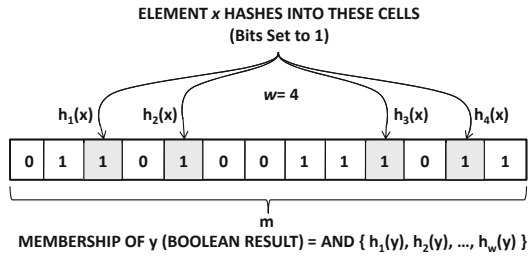

Figure 12.1: The bloom filter

Algorithm *BloomConstruct*(Stream: $\mathcal{S}$, Size: m, Num. Hash Functions: w)
begin
  Initialize all elements in a bit array $\mathcal{B}$ of size m to 0;
  repeat
    Receive next stream element $x \in \mathcal{S}$;
    for $i = 1$ to w **do**
      Update $h_i(x)$th element in bit array $\mathcal{B}$ to 1;
  until end of stream $\mathcal{S}$;
  return $\mathcal{B}$;
end

Figure 12.2: Update of bloom filter

are not possible with the bloom filter. On the other hand, if all the entries $h_1(y) \ldots h_w(y)$ in the bit array have a value of 1, then it is reported that y has occurred in the data stream. This can be checked efficiently by applying an "AND" logical operator to all the bit entries corresponding to the indices $h_1(y) \ldots h_w(y)$ in the bit array. The overall procedure of membership checks is illustrated in Fig. 12.3. The binary result for the decision problem for checking membership is tracked by the variable *BooleanFlag*. This binary flag is reported at the end of the procedure.

The bloom filter approach can lead to false positives, but not false negatives. A false positive occurs, if all the w different bloom filter array elements $h_i(y)$ for $i \in \{1 \ldots w\}$ have been set to 1 by some spurious element other than y. This is a direct result of collisions. As the number of elements in the data stream increases, all elements in the bloom filter are eventually set to 1. In such a case, all set-membership queries will yield a positive response. This is, of course, not a useful application of the bloom filter. Therefore, it is instructive to bound the false positive probability in terms of the size of the filter and the number of distinct elements in the data stream.

Lemma 12.2.2 *Consider a bloom filter $\mathcal{B}$ with m elements, and w different hash functions. Let n be the number of distinct elements in the stream $\mathcal{S}$ so far. Consider an element y, which has not appeared in the stream so far. Then, the probability F that an element y is reported as a false positive is given by the following:*

$$F = \left[1 - \left(1 - \frac{1}{m}\right)^{w \cdot n}\right]^w \tag{12.18}$$

Algorithm *BloomQuery*(Element: y, Bloom Filter: $\mathcal{B}$)
begin
  Initialize $BooleanFlag = 1$;
  for $i = 1$ to w **do**
    $BooleanFlag = BooleanFlag$ AND $h_i(y)$;
  return $BooleanFlag$;
end

Figure 12.3: Membership check using bloom filter

Proof: Consider a particular bit corresponding to the bit array element $h_r(y)$ for some fixed value of the index $r \in \{1 \ldots w\}$. Each element $x \in \mathcal{S}$ sets w different bits $h_1(x) \ldots h_w(x)$ to 1. The probability that *none* of these bits is the same as $h_r(y)$ is given by $(1-1/m)^w$. Over n distinct stream elements, this probability is $(1-1/m)^{w \cdot n}$. Therefore, the probability that the bit array index $h_r(y)$ is set to 1, by at least one of the n spurious elements in $\mathcal{S}$ is given by $Q = 1-(1-1/m)^{w \cdot n}$. A false positive occurs, when *all* bit array indices $h_r(y)$ (over varying values of $r \in \{1 \ldots w\}$) have been set to 1. The probability of this is $F = Q^w$. The result follows. ■

While the false-positive probability is expressed above in terms of the number of *distinct* stream elements, it is trivially true for the total number of stream elements (including repetitions), as an upper bound.

The expression in the aforementioned lemma can be simplified by observing that $(1-1/m)^m \approx e^{-1}$, where e is the base of the natural logarithm. Correspondingly, the expression can be rewritten as follows:

$$F = (1 - e^{-n \cdot w/m})^w. \tag{12.19}$$

Values of w that are too small or too large lead to poor performance. The value of w needs be selected optimally in terms of m and n to minimize the number of false positives. The number of false positives is minimized, when $w = m \cdot \ln(2)/n$. Substituting this value in Eq. 12.19, it can be shown that the probability of false positives for optimal number of hash functions is:

$$F = 2^{-m \cdot \ln(2)/n}. \tag{12.20}$$

The expression above can be written purely as an expression of m/n. Therefore, *for a fixed value of the false-positive probability F, the length of the bloom filter m needs to be proportional to the number of distinct elements n in the data stream.* Furthermore, the constant of proportionality for a particular false-positive probability F can be shown to be $\frac{m}{n} = \frac{\ln(1/F)}{(\ln(2))^2}$. While this may not seem like a significant compression, it needs to be pointed out that bloom filters use elementary *bits* to track the membership of arbitrary elements, such as strings. Furthermore, because of bitwise operations, which can be implemented very efficiently with low-level implementations, the overall approach is generally very efficient.

It does need to be kept in mind that the value of n is not known in advance for many applications. Therefore, one strategy is to use a cascade of bloom filters for geometrically increasing values of w, and to use a logical AND of the membership query result over different bloom filters. This is a practical approach that provides more stable performance over the life of the data stream.

The bloom filter is referred to as a "filter" because it is often used to make decisions on which elements to exclude from a data stream, when they meet the membership condition.

For example, if one wanted to filter all duplicates from the data stream, the bloom filter is an effective strategy. Another strategy is to filter forbidden elements from a universe of values, such as a set of spammer e-mail addresses in an e-mail stream. In such a case, the bloom filter needs to be constructed up front with the spam e-mail addresses.

Many variations of the basic bloom filter strategy provide different capabilities and trade-offs:

1. The bloom filter can be used to approximate the number of distinct elements in a data stream. If $m_0 < m$ is the number of bits with a value of 0 in the bloom filter, then the number of distinct elements n can be estimated as follows (see Exercise 13):

$$n \approx \frac{m \cdot \ln(m/m_0)}{w} \tag{12.21}$$

   The accuracy of this estimate reduces drastically, as the bloom filter fills up. When $m_0 = 0$, the value of n is estimated to be ∞, and therefore the estimate is practically useless.

2. The bloom filter can be used to estimate the size of the intersection and union of sets corresponding to different streams, by creating one bloom filter for each stream. To determine the size of the union, the bitwise OR of the two filters is determined. The bitwise OR of the filter can be shown to be *exactly* the same as the bloom filter representation of the union of the two sets. Then, the formula of Eq. 12.21 is used. However, such an approach cannot be used for determining the size of the intersection. While the intersection of two sets can be approximated by using a bitwise AND operation on the two filters, the resulting bit positions in the filter will not be the same as that obtained by constructing the filter directly on the intersection. The resulting filter might contain false negatives, and, therefore, such an approach is *lossy*. To estimate the size of the intersection, one can first estimate the size of the union and then use the following simple setwise relationship:

$$|\mathcal{S}_1 \cap \mathcal{S}_2| = |\mathcal{S}_1| + |\mathcal{S}_2| - |\mathcal{S}_1 \cup \mathcal{S}_2| \tag{12.22}$$

3. The bloom filter is primarily designed for membership queries, and is not the most space-efficient data structure, when used purely for distinct element counting. In a later section, a space-efficient technique, referred to as the Flajolet–Martin algorithm, will be discussed.

4. The bloom filter can allow a limited (one-time) tracking of deletions by setting the corresponding bit elements to zero, when an element is deleted. In such a case, false negatives are also possible.

5. Variants of the bloom filter can be designed in which the w hash functions can map onto separate bit arrays. A further generalization of this principle is to track counts of elements rather than simply binary bit values to enable richer queries. This generalization, discussed in the next section, is also referred to as the *count-min* sketch.

Bloom filters are commonly used in many streaming settings in the text domain.

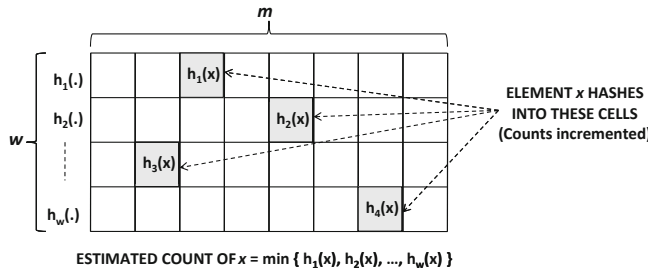

Figure 12.4: The count-min sketch

12.2.2.2 Count-Min Sketch

While the bloom filter is effective for set-membership queries, it is not designed for methods that require *count-based* summaries. This is because the bloom filter tracks only binary values. The count-min sketch is designed for resolving such queries and is intuitively related to the bloom filter. A count-min sketch consists of a set of w different *numeric* arrays, each of which has a length m. Thus, the space requirement of the count-min sketch is equal to $m \cdot w$ cells containing *numeric values*. The elements of each of the w numeric arrays are indexed starting with 0, corresponding to an index range of $\{0 \ldots m-1\}$. The count-min sketch can also be viewed as a $w \times m$ 2-dimensional array of cells.

Each of the w numeric arrays corresponds to a hash function. The ith numeric array corresponds to the ith hash function $h_i(\cdot)$. The hash functions have the following properties:

1. The ith hash function $h_i(\cdot)$ maps a stream element to an integer in the range $[0 \ldots m-1]$. This mapping can also be viewed as one of the index values in the ith numeric array.

2. The w hash functions $h_1(\cdot) \ldots h_w(\cdot)$ are fully independent of one another, but *pairwise* independent over different arguments. In other words, for any two values x_1 and x_2, $h_i(x_1)$ and $h_i(x_2)$ are independent.

The *pairwise* independence requirement is a weaker one than the full independence requirement. This is a convenient property of the count-min sketch because it is usually easier to construct pairwise independent hash functions rather than fully independent ones.

The procedure for updating the sketch is as follows. All $m \cdot w$ entries in the count-min sketch are initialized to 0. For each incoming stream element x, the functions $h_1(x) \ldots h_w(x)$ are applied to it. For the ith array, the element $h_i(x)$ is incremented by 1. Thus, if the count-min sketch $\mathcal{CM}$ is visualized as a 2-dimensional $w \times m$ numeric array, then the element $(i, h_i(x))$ is incremented[2] by 1. Note that the value of $h_i(x)$ maps to an integer in the range $[0, m-1]$. This is also the range of the indices of each numeric array. A pictorial illustration of the count-min sketch and the corresponding update process is provided in Fig. 12.4. The pseudocode for the overall update process is illustrated in Fig. 12.5. In the pseudocode, the stream is denoted by $\mathcal{S}$, and the count-min sketch data structure is denoted by $\mathcal{CM}$. The inputs to the algorithm are the stream $\mathcal{S}$ and two parameters (w, m) specifying the size of the 2-dimensional array for the count-min sketch. A 2-dimensional $w \times m$ array $\mathcal{CM}$ is initialized with all values set to 0. For each incoming stream element, the counts of all the

[2]In the event that each distinct element is associated with a nonnegative frequency, the count-min sketch can be updated with the frequency value. Only the simple case of unit updates is discussed here.

Algorithm *CountMinConstruct*(Stream: $\mathcal{S}$, Width: w, Height: m)
begin
  Initialize all entries of $w \times m$ array $\mathcal{CM}$ to 0;
  repeat
    Receive next stream element $x \in \mathcal{S}$;
    for $i = 1$ to w **do**
      Increment $(i, h_i(x))$th element in $\mathcal{CM}$ by 1;
  until end of stream $\mathcal{S}$;
  return $\mathcal{CM}$;
end

Figure 12.5: Update of count-min sketch

Algorithm *CountMinQuery*(Element: y, Count-min Sketch: $\mathcal{CM}$)
begin
  Initialize $Estimate = \infty$;
  for $i = 1$ to w **do**
    $Estimate = \min\{Estimate, V_i(y)\}$;
  { $V_i(y)$ is the count of the $(i, h_i(y))$th element in $\mathcal{CM}$ }
  return $Estimate$;
end

Figure 12.6: Frequency queries for count-min sketch

cells $(i, h_i(x))$ are updated for $i \in \{1 \ldots w\}$. In the pseudocode description, the resulting sketch $\mathcal{CM}$ is returned after processing all the stream elements. In practice, the count-min sketch can be used at any time during the progression of the stream $\mathcal{S}$. As in the case of the bloom filter, it is possible for multiple stream elements to map to the same cell in the count-min sketch. Therefore, different stream elements will increment the same cell, and the resulting cell counts are always overestimates of one or more stream element counts.

The count-min sketch can be used for many different queries. The simplest query is to determine the frequency of an element y. The first step is to compute the hash functions $h_1(y) \ldots h_w(y)$. For the ith numeric array in $\mathcal{CM}$, the value $V_i(y)$ of the $(i, h_i(y))$th array element is retrieved. Each value $V_i(y)$ is an overestimate of the true frequency of y because of potential collisions. Therefore, the tightest possible estimate may be obtained by using the minimum value $\min_i\{V_i(y)\}$ over the different hash functions. The overall procedure for frequency estimation is illustrated in Fig. 12.6.

The count-min sketch causes an overestimation of frequency values because of collisions of nonnegative frequency counts of distinct stream items. It is therefore helpful to determine an upper bound on the estimation quality.

Lemma 12.2.3 *Let $E(y)$ be the estimate of the frequency of the item y, using a count-min sketch of size $w \times m$. Let n_f be the total frequencies of all items received so far, and $G(y)$ be true frequency of item y. Then, with probability at least $1 - e^{-w}$, the upper bound on the estimate $E(y)$ is as follows:*

$$E(y) \leq G(y) + \frac{n_f \cdot e}{m}. \tag{12.23}$$

Here, e represents the base of the natural logarithm.

Proof: The expected number of spurious items hashed to the cells belonging to item y is about[3] n_f/m, if all spurious items are hashed uniformly at random to the different cells. This result uses pairwise independence of hash functions because it relies on the fact that the mapping of y to a cell does not affect the distribution of another spurious item in its cells. The probability of the number of spurious items exceeding $n_f \cdot e/m$ in any of the w cells belonging to y is given by at most e^{-1} by the Markov inequality. For $E(y)$ to exceed the upper bound of Eq. 12.23, this violation needs to be repeated for all the w cells to which y is mapped. The probability of a violation of Eq. 12.23 is therefore e^{-w}. The result follows. ■

In many cases, it is desirable to directly control the error level ϵ and the error probability δ. By setting $m = e/\epsilon$ and $w = \ln(1/\delta)$, it is possible to bound the error with a user-defined tolerance $n_f \cdot \epsilon$ and probability at least $1-\delta$. Two natural generalizations of the point query can be implemented as follows:

1. If the stream elements have arbitrary positive frequencies associated with them, the only change required is to the update operation, where the counts are incremented by the relevant frequency. The frequency bound is identical to Eq. 12.23, with n_f representing the sum of the frequencies of the stream items.

2. If the stream elements have either positive or negative frequencies associated with them, then a further change is required to the query procedure. In this case, the *median* of the counts is reported. The corresponding error bound of Eq. 12.23 now needs to be modified. With a probability of at least $1-e^{-w/4}$, the estimated frequency $E(y)$ of item y lies in the following ranges:

$$G(y) - \frac{3n_f \cdot e}{m} \leq E(y) \leq G(y) + \frac{3n_f \cdot e}{m}. \tag{12.24}$$

   In this case, n_f represents the sum of the *absolute* frequencies of the incoming items in the data stream. The bounds in this case are much weaker than those for nonnegative elements.

A useful application is to determine the dot product of the frequency counts of the discrete attribute values in two data streams. This has a useful application in estimating the join size on the massive-domain attribute in two data streams. The dot product between the frequency counts of the items in a pair of nonnegative data streams can be estimated by first constructing a count-min sketch representation for each of the two data streams in a separate $w \times m$ count-min data structure. The same hash functions are used for both sketches. The dot product of their corresponding count-min arrays for each hash function is computed. The minimum value of the dot product over the w different arrays is reported as the estimation. As in the previous case, this is an overestimate, and an upper bound on the estimate may be obtained with a probability of at least $1 - e^{-w}$. The corresponding error tolerance for the upper bound is $n_f^1 \cdot n_f^2 \cdot e/m$, where n_f^1 and n_f^2 are the aggregate frequencies of the items in each of the two streams. Other useful queries with the use of the count-min sketch include the determination of quantiles and frequent elements. Frequent elements are also referred to as *heavy hitters*. The bibliographic notes contain pointers to various queries and applications that can be addressed with the use of the count-min sketch.

[3]It is exactly equal to n_s/m, where n_s is the frequency of all items other than y. However, n_s is less than n_f by the frequency of y.

12.2.2.3 AMS Sketch

As discussed at the beginning of this section, different synopsis structures are designed for different kinds of queries. While the bloom filter and count-min sketch provide good estimations of various queries, some queries, such as second moments, can be better addressed with the *Alon–Matias–Szegedy (AMS)* sketch. In the AMS sketch, a random binary value is generated from $\{-1, 1\}$ for each stream element by applying a hash function to the stream element. These binary values are assumed to be 4-wise independent. This means that, if at most four values generated from the same hash function are sampled, they will be statistically independent of one another. It is easier to design a 4-wise independent hash function than a fully independent hash function. The details of 4-wise independent hash functions may be found in the bibliographic notes.

Consider a stream in which the ith stream element is associated with the *aggregate* frequency f_i. The second-order moment F_2 of the data stream, for a stream with n *distinct* elements, is defined as follows:

$$F_2 = \sum_{i=1}^{n} f_i^2 \tag{12.25}$$

In the massive-domain scenario, where the number of distinct elements is large, this quantity is hard to estimate because running counts of the frequencies f_i cannot be maintained with an array. However, it can be estimated effectively using the AMS sketch. As a practical application, the second-order moment yields an estimate of the self-join size of a data stream with respect to the massive-domain attribute. The second-order moment can also be viewed as a variant of the Gini index, which measures the level of frequency skew over different items in the data stream. When the skew is large, the value of F_2 is large, and very close to its upper bound $\left(\sum_{i=1}^{n} f_i\right)^2$.

The AMS sketch contains m different sketch components, each of which is associated with an independent hash function. Each hash function generates its corresponding sketch component as follows. A random binary value, with equal probability for both realizations, is generated for the incoming stream element. This binary value is denoted by $r \in \{-1, 1\}$, and is generated using the hash function for that component. The frequency of each incoming stream element is multiplied by r, and added to the corresponding component of the sketch. Let $r_i \in \{-1, 1\}$ be the random value generated by a particular hash function for the ith distinct element. Then, the corresponding component Q of the sketch, for a stream of n distinct elements with aggregate frequencies $f_1 \ldots f_n$, can be shown to be equal to the following:

$$Q = \sum_{i=1}^{n} f_i \cdot r_i. \tag{12.26}$$

This relationship is because of the incremental way in which Q is updated, each time a stream item is received. Note that the value of Q is a random variable, dependent on how the binary random values $r_1 \ldots r_n$ are generated by the hash function. The value of Q is useful in estimating the second moment.

Lemma 12.2.4 *The second moment of the data stream can be estimated by the square of the AMS sketch component Q:*

$$F_2 = E[Q^2]. \tag{12.27}$$

Proof: It is easy to see that $Q^2 = \sum_{i=1}^{n} f_i^2 r_i^2 + 2\sum_{i=1}^{n}\sum_{j=1}^{n} f_i \cdot f_j \cdot r_i \cdot r_j$. For any pair of hash values r_i, r_j, we have $r_i^2 = r_j^2 = 1$ and $E[r_i \cdot r_j] = E[r_i] \cdot E[r_j] = 0$. The last of

these results uses 2-wise independence, which is implied by 4-wise independence. Therefore, $E[Q^2] = \sum_{i=1}^{n} f_i^2 = F_2$. ■

The 4-wise independence can also be used to bound the variance of the estimate (see Exercise 16).

Lemma 12.2.5 *The variance of the square of a component Q of the AMS sketch is bounded above by twice the frequency moment.*

$$Var[Q^2] \leq 2 \cdot F_2^2 \tag{12.28}$$

The bound on the variance can be reduced further by averaging over the m different sketch components $Q_1 \ldots Q_m$. The reduced variance can be used to create a (weak) probabilistic estimate on the quality of the second moment estimate with the Chebychev inequality. This can be tightened further by using a "mean–median combination trick" that is commonly used in such a probabilistic analysis. This trick can be used to robustly estimate a random variable, whenever its variance is no larger than a modest factor of the square of its expected value. This is the case for the random variable Q^2.

The mean–median combination trick works as follows. It is desired to establish a bound with probability at least $(1 - \delta)$ that the second moment can be estimated to within a multiplicative factor of $1 \pm \epsilon$. Let $Q_1 \ldots Q_m$ be m different sketch components, each of which is generated using a different hash function. The value of m is chosen to be $O(\ln(1/\delta)/\epsilon^2)$. These m sketch components are further partitioned into $O(\ln(1/\delta))$ different groups of size $O(1/\epsilon^2)$ each. The sketch values in each group are averaged. The median of these $O(\ln(1/\delta))$ averages is reported. A combination of the Chebychev inequality and the Chernoff bounds can be used to show the following result:

Lemma 12.2.6 *By selecting the median of $O(ln(1/\delta))$ averages of $O(1/\epsilon^2)$ copies of Q_i^2, it is possible to guarantee the accuracy of the sketch-based second-moment approximation to within $1 \pm \epsilon$ with a probability of at least $1 - \delta$.*

Proof: According to Lemma 12.2.5, the variance of each sketch component is at most $2 \cdot F_2^2$. By using the average of $16/\epsilon^2$ independent sketch components, the variance of the averaged estimate can be reduced to $F_2^2 \cdot \epsilon^2/8$. In this case, the Chebychev inequality shows that the ϵ-bound is violated by the averaged estimate with probability at most 1/8. Assume that a total of $4 \cdot \ln(1/\delta)$ such averaged and independent estimates are available. The random variable Y is defined as the sum of the Bernoulli indicator variables of ϵ-bound violations over these $q = 4 \cdot \ln(1/\delta)$ averages. The expected value of Y is $q/8 = \ln(1/\delta)/2$. The Chernoff bound is used to show the following:

$$P(Y > q/2) = P(Y > (1+3) \cdot q/8) = P(Y > (1+3)E[Y]) \leq e^{-3^2 \cdot \ln(1/\delta)/8} = \delta^{9/8} \leq \delta.$$

The median can violate the ϵ-bound only when more than half the averages violate the bound. The probability of this event is exactly $P(Y > q/2)$. Therefore, the median violates the ϵ-bound with probability at most δ. ■

The AMS sketch can be used to estimate many other values in a similar way, with corresponding quality bounds. For example, consider two streams with the sketch components Q_i and R_i.

1. The dot product between the frequency counts of the items in a pair of streams is estimated as the product of the corresponding sketch components Q_i and R_i. By using

the median of $O(\ln(1/\delta))$ averages of different sets of $O(1/\epsilon^2)$ values of $Q_i \cdot R_i$, it is possible to bound the approximation within $1 \pm \epsilon$ with probability at least $1 - \delta$. This estimation can be performed using the count-min sketch as well, though with a different bound.

2. The Euclidean distance between the frequency counts of a pair of streams can be estimated as $Q_i^2 + R_i^2 - 2Q_i \cdot R_i$. The Euclidean distance can be viewed as a linear combination of three different dot products (including self-products) between the frequency counts of the two streams. Because each dot product is itself bounded using the "mean–median trick" discussed above, the approach can be used to determine similar quality bounds in this case as well.

3. Like the count-min sketch, the AMS sketch can be used to estimate frequency values. For the jth distinct stream element with frequency f_j, the product of the random variable r_j and Q_i provides an estimate of the frequency.

$$E[f_j] = r_j \cdot Q_i. \tag{12.29}$$

The mean, median, or mean–median combination of these values over different sketch components Q_i can be reported as a robust estimate. The AMS sketch can also be used to identify heavy hitters from the data stream.

Some of the queries resolved by the AMS and count-min sketch are similar, although others are different. The bounds provided by the two techniques are also different, although none of them is strictly better than the other in all scenarios. The count-min sketch does have the advantage of being intuitively easy to interpret because of its natural hash-table data structure. As a result, it can be more naturally integrated in data mining applications such as clustering and classification in a seamless way.

12.2.2.4 Flajolet–Martin Algorithm for Distinct Element Counting

Sketches are designed for determining stream statistics that are dominated by *large aggregate signals* of frequent items. However, they are not optimized for estimating stream statistics that are dominated by infrequently occurring items. Problems such as distinct element counting are more directly influenced by the much larger number of infrequent items in a data stream. Distinct element counting can be performed efficiently with the Flajolet–Martin algorithm.

The Flajolet–Martin algorithm uses a hash function $h(\cdot)$ to render a mapping from a given element x in the data stream to an integer in the range $[0, 2^L - 1]$. The value of L is selected to be large enough, so that 2^L is an upper bound on the number of distinct elements. Usually, the value L is selected to be 64 for implementation convenience, and because the value of 2^{64} is large enough for most practical applications. Therefore, the binary representation of the integer $h(x)$ will have length L. The position[4] R of the rightmost 1 bit of the binary representation of the integer $h(x)$ is determined. Thus, the value of R represents the number of trailing zeros in this binary representation. Let R_{max} be the maximum value of R over all stream elements. The value of R_{max} can be maintained incrementally in the streaming scenario by applying the hash function to each incoming stream element, determining its rightmost bit, and then updating R_{max}. The key idea in the Flajolet–Martin algorithm is that the dynamically maintained value of R_{max} is logarithmically related to the number of distinct elements encountered so far in the stream.

[4]The position of the least significant bit is 0, the next most significant bit is 1, and so on.

The intuition behind this result is quite simple. For a uniformly distributed hash function, the probability of R trailing zeros in the binary representation of a stream element is equal to 2^{-R-1}. Therefore, for n distinct elements and a fixed value of R, the expected number of times that exactly R trailing zeros are achieved is equal to $2^{-R-1} \cdot n$. Therefore, for values of R larger than $\log(n)$, the expected number of such bitstrings falls off exponentially less than 1. Of course, in our application, the value of R is not fixed, but it is a random variable that is generated by the outcome of the hash function. It has been rigorously shown that the expected value $E[R_{max}]$ of the maximum value of R over all stream elements is logarithmically related to the number of distinct elements as follows:

$$E[R_{max}] = \log_2(\phi n), \quad \phi = 0.77351. \tag{12.30}$$

The standard deviation is $\sigma(R_{max}) = 1.12$. Therefore, the value of $2^{R_{max}}/\phi$ provides an estimate for the number of distinct elements n. To further improve the estimate of R_{max}, the following techniques can be used:

1. Multiple hash functions can be used, and the average value of R_{max} over the different hash functions is used.

2. The averages are still somewhat susceptible to large variations. Therefore, the "mean–median trick" may be used. The medians of a set of averages are reported. Note that this is similar to the trick used in the AMS sketch. As in that case, a combination of the Chebychev inequality and Chernoff bounds can be used to establish qualitative guarantees.

It should be pointed out that the bloom filter can also be used to estimate the number of distinct elements. However, the bloom filter is not a space-efficient way to count the number of distinct elements when set-membership queries are not required.

12.3 Frequent Pattern Mining in Data Streams

The frequent pattern mining problem in data streams is studied in the context of two different scenarios. The first scenario is the massive-domain scenario, in which the number of possible items is very large. In such cases, even the problem of finding frequent items becomes difficult. Frequent items are also referred to as *heavy hitters*. The second case is the conventional scenario of a large (but manageable) number of items that fit in main memory. In such cases, the frequent item problem is no longer quite as interesting, because the frequent counts can be directly maintained in an array. In such cases, one is more interested in determining frequent *patterns*. This is a difficult problem, because most frequent pattern mining algorithms require multiple passes over the entire data set. The one-pass constraint of the streaming scenario makes this difficult. In the following, two different approaches will be described. The first of these approaches leverages generic synopsis structures in conjunction with traditional frequent pattern mining algorithms and the second designs streaming versions of frequent pattern mining algorithms.

12.3.1 Leveraging Synopsis Structures

Synopsis structures can be used effectively in most streaming data mining problems, including frequent pattern mining. In the context of frequent pattern mining methods, synopsis structures are particularly attractive because of the ability to use a wider array of algorithms, or for incorporating temporal decay into the frequent pattern mining process.

12.3.1.1 Reservoir Sampling

Reservoir sampling is the most flexible approach for frequent pattern mining in data streams. It can be used either for frequent *item* mining (in the massive-domain scenario) or for frequent pattern mining. The basic idea in using reservoir sampling is simple:

1. Maintain a reservoir sample S from the data stream.
2. Apply a frequent pattern mining algorithm to the reservoir sample S and report the patterns.

It is possible to derive qualitative guarantees on the frequent patterns mined as a function of the sample size S. The probability of a pattern being a false positive can be determined by using the Chernoff bound. By using modestly lower support thresholds, it is also possible to obtain a guaranteed reduction in the number of false negatives. The bibliographic notes contain pointers to such guarantees. Reservoir sampling has several flexibility advantages because of its clean separation of the sampling and the mining process. Virtually, any efficient frequent pattern mining algorithm can be used on the memory-resident reservoir sample. Furthermore, different variations of pattern mining algorithms, such as constrained pattern mining or interesting pattern mining, can be applied as well. Concept drift is also relatively easy to address. The use of a decay-biased reservoir sample with off-the-shelf frequent pattern mining methods translates to a decay-weighted definition of the support.

12.3.1.2 Sketches

Sketches can be used for determining frequent *items*, though they cannot be used for determining frequent *itemsets* quite as easily. The core idea is that sketches are generally much better at estimating the counts of more frequent items accurately on a relative basis. This is because the bound on the frequency estimation of any item is an absolute one, in which the error depends on the aggregate frequency of the stream items rather than that of the item itself. This is evident from Lemma 12.2.3. As a result, the frequencies of heavy hitters can generally be estimated more accurately on a relative basis. Both the AMS sketch and the count-min sketch can be used to determine the heavy hitters. The bibliographic notes contain pointers to some of these algorithms.

12.3.2 Lossy Counting Algorithm

The lossy counting algorithm can be used either for frequent item, or frequent itemset counting. The approach divides the stream into segments $S_1 \ldots S_i \ldots$ such that each segment S_i has a size $w = \lfloor 1/\epsilon \rfloor$. The parameter ϵ is a user-defined tolerance on the required accuracy.

First, the easier problem of frequent item mining will be described. The algorithm maintains the frequencies of all the items in an array and increments them as new items arrive. If the number of distinct items is not very large, then one can maintain all the counts and report the frequent ones. The problem arises when the total available space is less than that required to maintain the counts of the distinct items. In such cases, whenever the boundary of a segment S_i is reached, infrequent items are dropped. This results in the removal of many items because the vast majority of the items in the stream are infrequent in practice. How does one decide which items should be dropped, to retain a quality bound on the approximation? For this purpose, a *decremental* trick is used.

Whenever the boundary of a segment S_i is reached, the frequency count of *every* item in the array is decreased by 1. After the decrease, items with zero frequencies are pruned from

the array. Consider the situation where n items have already been processed. Because each segment contains w items, a total of $r = O(n/w) = O(n \cdot \epsilon)$ segments have been processed. This implies that any particular item has been decremented at most $r = O(n \cdot \epsilon)$ times. Therefore, if $\lfloor n \cdot \epsilon \rfloor$ were to be added to the counts of the items after processing n items, then no count will be underestimated. Furthermore, this is a good overestimate on the frequency that is proportional to the user-defined tolerance ϵ. If the frequent items are reported with the use of this overestimate, it may result in some false positives, but no false negatives. Under some uniformity assumptions, it has been shown that the lossy counting algorithm requires $O(1/\epsilon)$ space.

The approach can be generalized to the case of frequent patterns by *batching* multiple segments, each of size $w = \lfloor 1/\epsilon \rfloor$. In this case, arrays containing counts of patterns (rather than items) are maintained. However, patterns can obviously not be generated efficiently from individual transactions. The idea here is to batch η segments that are read into main memory. The value of η is decided on the basis of memory availability. When the η segments have been read in, the frequent patterns with (absolute) support of at least η are determined using any memory-based frequent pattern mining algorithm. First, all the old counts in the array are decremented by η, and then the counts of the corresponding patterns from the current segment are added to the array. Those itemsets with zero or negative supports are removed from the array. Over the entire processing of the stream of length n, the count of any itemset is decreased by at most $\lfloor \epsilon \cdot n \rfloor$. Therefore, by adding $\lfloor \epsilon \cdot n \rfloor$ to all array counts at the end of the process, no counts would be underestimated. The overestimate is the same as in the previous case. Thus, it is possible to report the frequent patterns with no false negatives, and false positives that are regulated by user-defined tolerance ϵ. Conceptually, the main difference of this algorithm for frequent *itemset* counting from the aforementioned algorithm for frequent *item* counting is that batching is used. The main goal of batching is to reduce the number of frequent patterns generated at support level of η during the application of the frequent pattern mining algorithm. If batching is not used, then a large number of irrelevant frequent patterns will be generated at an absolute support level of 1. The main shortcoming of lossy counting is that it cannot adjust to concept drift. In this sense, reservoir sampling has a number of advantages over the lossy counting algorithm.

12.4 Clustering Data Streams

The problem of clustering is especially significant in the data stream scenario because of its ability to provide a compact synopsis of the data stream. A clustering of the data stream can often be used as a heuristic substitute for reservoir sampling, especially if a fine-grained clustering is used. For these reasons, stream clustering is often used as a precursor to other applications such as streaming classification. In the following, a few representative stream clustering algorithms will be discussed.

12.4.1 STREAM Algorithm

The *STREAM* algorithm is based on the k-medians clustering methodology. The core idea is to break the stream into smaller memory-resident segments. Thus, the original data stream $\mathcal{S}$ is divided into segments $S_1 \ldots S_r$. Each segment contains at most m data points. The value of m is fixed on the basis of a predefined memory budget.

Because each segment S_i fits in main memory, a more complex clustering algorithm can be applied to it, without worrying about the one-pass constraint. One can use a variety

of different k-medians[5] style algorithms for this purpose. In k-medians algorithms, a set $\mathcal{Y}$ of k representatives from each chunk S_i is selected, and each point in S_i is assigned to its closest representative. The goal is to select the representatives to minimize the *sum of squared distances (SSQ)* of the assigned data points to these representatives. For a set of m data points $\overline{X_1} \ldots \overline{X_m}$ in segment S, and a set of k representatives $\mathcal{Y} = \overline{Y_1} \ldots \overline{Y_k}$, the objective function is defined as follows:

$$Objective(S, \mathcal{Y}) = \sum_{\overline{X_i} \in S, \overline{X_i} \Leftarrow \overline{Y_{j_i}}} dist(\overline{X_i}, \overline{Y_{j_i}}). \tag{12.31}$$

The assignment operator is denoted by "$\Leftarrow$" above. The squared distance between a data point and its assigned cluster center is denoted by $dist(\overline{X_i}, \overline{Y_{j_i}})$, where the data record $\overline{X_i}$ is assigned to the representative $\overline{Y_{j_i}}$. In principle, any partitioning algorithm, such as k-means or k-medoids, can be applied to the segment S_i in order to determine the representatives $\overline{Y_1} \ldots \overline{Y_k}$. For the purpose of discussion, this algorithm will be treated as a black box.

After the first segment S_1 has been processed, we now have a set of k medians that are stored away. The number of points assigned to each representative is stored as a "weight" for that representative. Such representatives are considered *level-1* representatives. The next segment S_2 is independently processed to find its k optimal median representatives. Thus, at the end of processing the second segment, one will have $2 \cdot k$ such representatives. Thus, the memory requirement for storing the representatives also increases with time, and after processing r segments, one will have a total of $r \cdot k$ representatives. When the number of representatives exceeds m, a second level of clustering is applied to these set of $r \cdot k$ points, except that the stored weights on the representatives are also used in the clustering process. The resulting representatives are stored as level-2 representatives. In general, when the number of representatives of level-p reaches m, they are converted to k level-$(p+1)$ representatives. Thus, the process will result in increasing the number of representatives of all levels, though the number of representatives at higher levels will increase exponentially slower than those at the lower levels. At the end of processing the entire data stream (or when a specific need for the clustering result arises), all remaining representatives of different levels are clustered together in one final application of the k-medians subroutine.

The specific choice of the algorithm used for the k-medians problem is critical in ensuring a high-quality clustering. The other factor that affects the quality of the final output is the effect of the problem decomposition into chunks followed by hierarchical clustering. How does such a problem decomposition affect the final quality of the output? It has been shown in the *STREAM* paper [240], that the final quality of the output cannot be arbitrarily worse than the particular subroutine that is used at the intermediate stage for k-medians clustering.

Lemma 12.4.1 *Let the subroutine used for k-medians clustering in the STREAM algorithm have an approximation factor of c. Then, the STREAM algorithm will have an approximation factor of no worse than $5 \cdot c$.*

A variety of solutions are possible for the k-medians problem. In principle, virtually any approximation algorithm can be used as a black box. A particularly effective solution is based on the problem of facility location. The reader is referred to the bibliographic notes for pointers to the relevant approach.

[5]This terminology is different from the k-medians approach introduced in Chap. 6. The relevant subroutines in the *STREAM* algorithm are more similar to a k-medoids algorithm. Nevertheless, the "k-medians" terminology is used here to ensure consistency with the original research paper describing *STREAM* [240].

A major limitation of the *STREAM* algorithm is that it is not particularly sensitive to evolution in the underlying data stream. In many cases, the patterns in the underlying stream may evolve and change significantly. Therefore, it is critical for the clustering process to be able to adapt to such changes and provide insights over different time horizons. In this sense, the *CluStream* algorithm is able to provide significantly better insights at different levels of temporal granularity.

12.4.2 CluStream Algorithm

The concept drift in an evolving data stream changes the clusters significantly over time. The clusters over the past day are very different from the clusters over the past month. In many data mining applications, analysts may wish to have the flexibility to determine the clusters based on one or more time horizons, which are unknown at the beginning of the stream clustering process. Because stream data naturally imposes a one-pass constraint on the design of the algorithms, it is difficult to compute clusters over different time horizons using conventional algorithms. A direct extension of the *STREAM* algorithm to such a case would require the simultaneous maintenance of the intermediate results of clustering algorithms over all possible time horizons. The computational burden of such an approach increases with progression of the data stream and can rapidly become a bottleneck for online implementation.

A natural approach to address this issue is to apply the clustering process with a two-stage methodology, including an online microclustering stage, and an offline macroclustering stage. The online microclustering stage processes the stream in real time to continuously maintain summarized but detailed cluster statistics of the stream. These are referred to as *microclusters*. The offline macroclustering stage further summarizes these detailed clusters to provide the user with a more concise understanding of the clusters over different time horizons and levels of temporal granularity. This is achieved by retaining sufficiently detailed statistics in the microclusters, so that it is possible to re-cluster these detailed representations over user-specified time horizons.

12.4.2.1 Microcluster Definition

It is assumed that the multidimensional records in the data stream are denoted by $\overline{X_1} \ldots \overline{X_k} \ldots$, arriving at time stamps $T_1 \ldots T_k \ldots$. Each $\overline{X_i}$ is a multidimensional record containing d dimensions that are denoted by $\overline{X_i} = (x_i^1 \ldots x_i^d)$. The microclusters capture summary statistics of the data stream to facilitate clustering and analysis over different time horizons. These summary statistics are defined by the following structures:

1. *Microclusters:* The microclusters are defined as a temporal extension of the *cluster feature vector* used in the *BIRCH* algorithm of Chap. 7. This concept can be viewed as a temporally optimized representation of the CF-vector specifically designed for the streaming scenario. To achieve this goal, the microclusters contain temporal statistics in addition to the feature statistics.

2. *Pyramidal Time Frame:* The microclusters are stored at snapshots in time that follow a pyramidal pattern. This pattern provides an effective trade-off between the storage requirements and the ability to recall summary statistics from different time horizons. This is important for enabling the ability to re-cluster the data over different time horizons.

Microclusters are defined as follows.

Definition 12.4.1 *A microcluster for a set of d-dimensional points $X_{i_1} \ldots X_{i_n}$ with time stamps $T_{i_1} \ldots T_{i_n}$ is the $(2 \cdot d + 3)$ tuple $(\overline{CF2^x}, \overline{CF1^x}, CF2^t, CF1^t, n)$, wherein $\overline{CF2^x}$ and $\overline{CF1^x}$ each correspond to a vector of d entries. The definition of each of these entries is as follows:*

1. *For each dimension, the sum of the squares of the data values is maintained in $\overline{CF2^x}$. Thus, $\overline{CF2^x}$ contains d values. The p-th entry of $\overline{CF2^x}$ is equal to $\sum_{j=1}^{n} (x_{i_j}^p)^2$.*
2. *For each dimension, the sum of the data values is maintained in $\overline{CF1^x}$. Thus, $\overline{CF1^x}$ contains d values. The p-th entry of $\overline{CF1^x}$ is equal to $\sum_{j=1}^{n} x_{i_j}^p$.*
3. *The sum of the squares of the time stamps $T_{i_1} \ldots T_{i_n}$ is maintained in $CF2^t$.*
4. *The sum of the time stamps $T_{i_1} \ldots T_{i_n}$ is maintained in $CF1^t$.*
5. *The number of data points is maintained in n.*

An important property of microclusters is that they are *additive*. In other words, the microclusters can be updated by purely additive operations. Note that each of the $2 \cdot d + 3$ components of the microcluster can be expressed as a linearly separable sum over the constituent data points in the microcluster. This is an important property for enabling the efficient maintenance of the microclusters in the online streaming scenario. When a data point $\overline{X_i}$ is added to a microcluster, the corresponding statistics of the data point $\overline{X_i}$ need to be added to each of the $(2 \cdot d + 3)$ components. Similarly, the microclusters for the stream period (t_1, t_2) can be obtained by subtracting the microclusters at time t_1 from those at time t_2. This property is important for enabling the computation of the higher-level macroclusters over an arbitrary time horizon (t_1, t_2) from the microclusters stored at different times.

12.4.2.2 Microclustering Algorithm

The data stream clustering algorithm can generate approximate clusters in any user-specified length of history from the current instant. This is achieved by storing the microclusters at particular moments in the stream that are referred to as *snapshots*. At the same time, the current snapshot of microclusters is always maintained by the algorithm. The additive property can be used to extract microclusters from any time horizon. The macroclustering phase is applied to this representation.

The input to the algorithm is the number of microclusters, denoted by k. The online phase of the algorithm works in an iterative fashion, by always maintaining a current set of microclusters. Whenever a new data point $\overline{X_i}$ arrives, the microclusters are updated to reflect the changes. Each data point either needs to be absorbed by a microcluster, or it needs to be put in a cluster of its own. The first preference is to absorb the data point into a currently existing microcluster. The distance of the data point to the current microcluster centroids $\mathcal{M}_1 \ldots \mathcal{M}_k$ is determined. The distance value of the data point $\overline{X_i}$ to the centroid of the microcluster $\mathcal{M}_j$ is denoted by $dist(\mathcal{M}_j, \overline{X_i})$. Because the centroid of the microcluster can be derived from the cluster feature vector, this distance value can be computed easily. The closest centroid $\mathcal{M}_p$ is determined. The data point $\overline{X_i}$ is assigned to its closest cluster $\mathcal{M}_p$, unless it is deemed that the data point does not "naturally" belong to that (or any other) microcluster. In such cases, the data point $\overline{X_i}$ needs to be assigned a (new) microcluster of its own. Therefore, before assigning a data point to a microcluster, it first needs to be decided whether it naturally belongs to its closest microcluster centroid $\mathcal{M}_p$.

To make this decision, the cluster feature vector of $\mathcal{M}_p$ is used to decide if this data point falls within the *maximum boundary* of the microcluster $\mathcal{M}_p$. If so, then the data point $\overline{X_i}$ is added to the microcluster $\mathcal{M}_p$ by using the additivity property of microclusters. The maximum boundary of the microcluster $\mathcal{M}_p$ is defined as a factor t of the root-mean-square deviation of the data points in $\mathcal{M}_p$ from the centroid. The value of t is a user-defined parameter, and it is typically set to 3.

If the data point does not lie within the maximum boundary of the nearest microcluster, then a new microcluster must be created containing the data point $\overline{X_i}$. However, to create this new microcluster, the number of other microclusters must be reduced by 1 to free memory availability. This can be achieved by either deleting an old microcluster or merging two of the older clusters. This decision is made by examining the staleness of the different clusters, and the number of points in them. The time-stamp statistics of the microclusters are examined to determine whether one of them is "sufficiently" stale to merit removal. If this is not the case, then a merging of the two microclusters is initiated.

How is staleness of a microcluster determined? The microclusters are used to approximate the average time-stamp of the last m data points of the cluster $\mathcal{M}$. This value is not known explicitly because the last m data points are not explicitly retained in order to minimize memory requirements. The mean μ and variance σ^2 of the time-stamps in the microcluster can be used together with a normal distribution assumption of the distribution of time stamps to *estimate* this value. Thus, if the cluster contains $m_0 > m$ data points, then the $m/(2 \cdot m_0)$th percentile of the normal distribution with mean μ and variance σ^2 may be used as the estimate. This value is referred to as the *relevance stamp* of cluster $\mathcal{M}$. Note that μ and σ^2 can be computed from the temporal components of the cluster feature vectors. When the smallest such relevance stamp of any microcluster is below a user-defined threshold δ, it can be eliminated. In cases where no microclusters can be safely deleted, the closest microclusters are merged. The merging operation can be effectively performed because of the existence of the cluster feature vector. Distances between microclusters can be easily computed using the cluster-feature vector. When two microclusters are merged, their statistics are added together, because of the additivity property of microclusters.

12.4.2.3 Pyramidal Time Frame

The microclusters statistics are stored periodically to enable horizon-specific analysis of the clusters. This maintenance is performed during the microclustering phase. In this approach, the microcluster snapshots are stored at varying levels of granularity depending on the recency of the snapshot. Snapshots are classified into different *orders* that can vary from 1 to $\log(T)$, where T is the clock time elapsed since the beginning of the stream. The order of a snapshot regulates the level of temporal granularity at which it is stored, according to the following rules:

- Snapshots of the ith order are stored at time intervals of α^i, where α is an integer and $\alpha \geq 1$. Specifically, each snapshot of the ith order is stored when the clock value is exactly divisible by α^i.
- At any given time, only the last $\alpha^l + 1$ snapshots of order i are stored.

The aforementioned definition allows for considerable redundancy in storage of snapshots. For example, the clock time of 8 is divisible by 2^0, 2^1, 2^2, and 2^3 (where $\alpha = 2$). Therefore, the state of the microclusters at a clock time of 8 simultaneously corresponds to order 0, order 1, order 2, and order 3 snapshots. From an implementation point of view, a snapshot needs to be maintained only once.

Table 12.2: An example [39] of snapshots stored for $\alpha = 2$ and $l = 2$

Order of Snapshots	Clock times (last five snapshots)
0	55 54 53 52 51
1	54 52 50 48 46
2	52 48 44 40 36
3	48 40 32 24 16
4	48 32 16
5	32

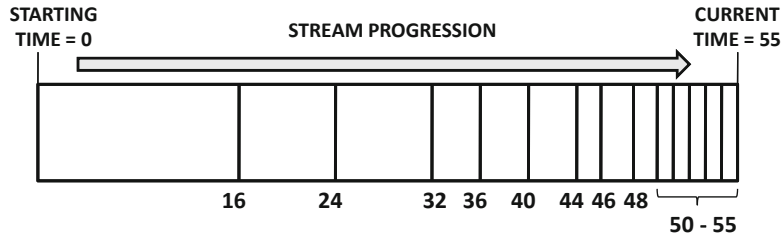

Figure 12.7: Recent snapshots are stored more frequently by pyramidal time frame

To illustrate the snapshots, an example will be used. Consider the case when the stream has been running starting at a clock time of 1, and a use of $\alpha = 2$ and $l = 2$. Therefore, $2^2 + 1 = 5$ snapshots of each order are stored. Then, at a clock time of 55, snapshots at the clock times illustrated in Table 12.2 are stored. While some snapshots are redundant in this case, they are not stored in a redundant way. The corresponding pattern of storage is illustrated in Fig. 12.7. It is evident that recent snapshots are stored more frequently in the pyramidal pattern of storage.

The following observations are true at any moment in time over the course of the data stream:

- The maximum order of any snapshot stored at T time units since the beginning of the stream mining process is $\log_\alpha(T)$.
- The maximum number of snapshots maintained at T time units since the beginning of the stream mining process is $(\alpha^l + 1) \cdot \log_\alpha(T)$.
- For any user-specified time horizon h, at least one stored snapshot can be found, which corresponds to a horizon of length within a factor $(1 + 1/\alpha^{l-1})$ units of the desired value h. This property is important because the microcluster statistics of time horizon $(t_c - h, t_c)$ can be constructed by subtracting the statistics at time $(t_c - h)$ from those at time t_c. Therefore, the microcluster within the approximate temporal locality of $(t_c - h)$ can be used instead. This enables the approximate clustering of data stream points within an arbitrary time horizon $(t_c - h, t_c)$ from the stored pyramidal pattern of microcluster statistics.

For larger values of l, the time horizon can be approximated as closely as desired. It is instructive to use an example to illustrate the combination of the effectiveness and compactness achieved by the pyramidal pattern of snapshot storage. For example, by choosing

$l = 10$, it is possible to approximate any time horizon within 0.2 %. At the same time, a total of only $(2^{10} + 1) \cdot \log_2(100 * 365 * 24 * 60 * 60) \approx 32,343$ snapshots are required for a stream with a clock granularity of 1 s and running over 100 years. If each snapshot of size $k \cdot (2 \cdot d + 3)$ requires storage of less than a megabyte, the overall storage required is of order of a few gigabytes. Because historical snapshots can be stored on disk and only the current snapshot needs to be maintained in main memory, this requirement is modest from a practical point of view. As the clustering algorithm progresses, only the relevant snapshots according to the pyramidal time frame are maintained. The remaining snapshots are discarded. This enables the computation of horizon-specific clusters at a modest storage cost.

12.4.3 Massive-Domain Stream Clustering

As discussed earlier, the massive-domain scenario is ubiquitous in the stream context. In many cases, one may need to work with a multidimensional data stream, in which the individual attributes are drawn on a massive domain of possible values. In such cases, stream analysis becomes much more difficult because "concise" summaries of the clusters become much more space-intensive. This is also the motivation for many synopsis structures, such as the bloom filter, the count-min sketch, the AMS sketch, and the Flajolet–Martin algorithm.

The data clustering problem also becomes more challenging in the massive-domain scenario, because of the difficulty in maintaining concise statistics of the clusters. A recent method *CSketch* has been designed for clustering massive-domain data streams. The idea in this method is to use a count-min sketch to store the frequencies of attribute–value combinations in each cluster. Thus, the number of count-min sketches used is equal to the number of clusters. An online k-means style clustering is applied, in which the sketch is used as the representative for the (discrete) attributes in the cluster. For any incoming data point, a dot product is computed with respect to each cluster.

The computation is performed as follows. For each attribute–value combination in the d-dimensions, the hash function $h_r(\cdot)$ is applied to it for a particular value of r. The frequency of the corresponding sketch cell is determined. The frequencies of all the relevant sketch cells for the d different dimensions are added together. This provides an estimate of the dot product. To obtain a tighter estimate, the minimum value over different hash functions (different values of r) is used. The dot product is divided by the total frequency of items in the cluster, to avoid bias towards clusters with many data items.

This computation can be performed accurately because the count-min sketch can compute the dot product accurately in a small space. The data point is assigned to the cluster with which it has the largest dot product. The statistics in the sketch representing that particular cluster are then updated. Thus, this approach shares a common characteristic with microclustering in terms of how data points are incrementally assigned to clusters. However, it does not implement the merging and removal steps. Furthermore, the sketch representation is used instead of the microcluster representation for cluster statistics maintenance. Theoretical guarantees can be shown on clustering quality, with respect to a clustering that has infinite space availability. The bibliographic notes contain pointers to these results.

12.5 Streaming Outlier Detection

The problem of streaming outlier detection typically arises either in the context of multidimensional data or time-series data streams. Outlier detection in multidimensional data streams is generally quite different from time series outlier detection. In the latter case,

each time series is treated as a unit, whereas temporal correlations are much weaker for multidimensional data. This chapter will address only multidimensional streaming outlier detection, whereas time-series methods will be addressed in Chap. 14.

The multidimensional stream scenario is similar to static multidimensional outlier analysis. The only difference is the addition of a temporal component to the analysis, though this temporal component is much weaker than in the case of time series data. In the context of multidimensional data streams, efficiency is an important concern because the outliers need to be discovered quickly. There are two kinds of outliers that may arise in the context of multidimensional data streams.

1. One is based on the outlier detection of individual records. For example, a first news story on a specific topic represents an outlier of this type. Such an outlier is also referred to as a *novelty*.

2. The second is based on changes in the *aggregate trends* of the multidimensional data. For example, an unusual event such as a terrorist attack may lead to a burst of news stories on a specific topic. This represents an aggregated outlier based on a specific time window. The second kind of change point almost always begins with an individual outlier of the first type. However, an individual outlier of the first type may not always develop into an aggregate change point. This is closely related to the concept of concept drift. While concept drift is generally gentle, an abrupt change may be viewed as an outlier *instant in time* rather than an outlier *data point*.

Both kinds of outliers (or change points) will be discussed in this section.

12.5.1 Individual Data Points as Outliers

The problem of detecting individual data points as outliers is closely related to the problem of *unsupervised novelty detection*, especially when the entire history of the data stream is used. This problem is studied extensively in the text domain in the context of the problem of *first story detection*. Such novelties are often trend setters and may eventually become a part of the normal data. However, when an individual record is declared an outlier in the context of a *window* of data points, it may not necessarily be a novelty. In this context, proximity-based algorithms are particularly easy to generalize to the incremental scenario by almost direct applications of the corresponding algorithms to the window of data points.

Distance-based algorithms can be easily generalized to the streaming scenario. The original distance-based definition of outliers is modified in the following way:

The outlier score of a data point is defined in terms of its k-nearest neighbor distance to data points in a time window of length W.

Note that this is a relatively straightforward modification of the original distance-based definition. When the entire window of data points can be maintained in main memory, it is fairly easy to determine the outliers by computing the score of every data point in the window. However, incremental maintenance of the scores of data points is more challenging because of the addition and removal of data points from the window. Furthermore, some algorithms such as *LOF* require the re-computation of statistics such as reachability distances. The *LOF* algorithm has been extended to the incremental scenario. Two steps are performed in the process:

1. The statistics of the newly inserted data points are computed such as its reachability distance and *LOF* score.

2. The *LOF* scores of the existing points in the window are updated along with their densities and reachability distances. In other words, the scores of many of the existing data points need to be updated because they are affected by the addition of a new data point. However, not all scores need to be updated because only the locality of the new data point is affected. Similarly, when data points are deleted, only the *LOF* scores in the locality of the deleted point are affected.

Because distance-based methods are well-known to be computationally expensive, many of the aforementioned methods are still quite expensive in the context of the data stream. Therefore, the complexity of the outlier detection process can be greatly improved by using an online clustering-based approach. The microclustering approach discussed earlier in this chapter automatically discovers outliers, together with clusters.

While clustering-based methods are generally not advisable when the number of data points are limited, this is not the case in streaming analysis. In the context of a data stream, a sufficient number of data points are typically available to maintain the clusters at a very high level of granularity. *In the context of a streaming clustering algorithm, the formation of new clusters is often associated with unsupervised novelties.* For example, the *CluStream* algorithm explicitly regulates the creation of new clusters in the data stream when an incoming data point does not lie within a specified statistical radius of the existing clusters in the data. Such data points may be considered outliers. In many cases, this is the beginning of a new trend, as more data points are added to the cluster at later stages of the algorithm. In some cases, such data points may correspond to novelties, and in other cases, they may correspond to trends that were seen a long time ago, but are no longer reflected in the current clusters. In either case, such data points are interesting outliers. However, it is not possible to distinguish between these different kinds of outliers unless one is willing to allow the number of clusters in the stream to increase over time.

12.5.2 Aggregate Change Points as Outliers

The sudden changes in aggregate local and global trends in the underlying data are often indicative of anomalous events in the data. Many methods also provide statistical ways of quantifying the level of the changes in the underlying data stream. One way of measuring concept drift is to use the concept of velocity density. The idea in velocity density estimation is to construct a density-based velocity profile of the data. This is analogous to the concept of kernel density estimation in static data sets. The kernel density estimation $\overline{f}(\overline{X})$ for n data points and kernel function $K'_h(\cdot)$ is defined as follows:

$$f(\overline{X}) = \frac{1}{n} \sum_{i=1}^{n} K'_h(\overline{X} - \overline{X_i})$$

The kernel function used is a Gaussian kernel with width h.

$$K'_h(\overline{X} - \overline{X_i}) \propto e^{-||\overline{X}-\overline{X_i}||^2/(2h^2)}$$

The estimation error is defined by the kernel width h that is chosen in a data-driven manner based on Silverman's approximation rule [471].

The velocity density computations are performed over a temporal window of size h_t. Intuitively, the value of h_t defines the time horizon over which the evolution is measured.

Thus, if h_t is chosen to be large, then the velocity density estimation technique provides long term trends, whereas if h_t is chosen to be small then the trends are relatively short term. This provides the user flexibility in analyzing the changes in the data over different time horizons. In addition, a spatial smoothing parameter h_s is used that is analogous to the kernel width h in conventional kernel density estimation.

Let t be the current instant and S be the set of data points that have arrived in the time window $(t - h_t, t)$. The rate of increase in density at spatial location $\overline{X}$ and time t is estimated with two measures the *forward time-slice density estimate* and the *reverse time-slice density estimate*. Intuitively, the forward time-slice estimate measures the density function for all spatial locations at a given time t based on the set of data points that have arrived in the *past* time window $(t-h_t, t)$. Similarly, the reverse time-slice estimate measures the density function at a given time t based on the set of data points that will arrive in the *future* time window $(t, t + h_t)$. Obviously, this value cannot be computed until these points have actually arrived.

It is assumed that the ith data point in S is denoted by $(\overline{X_i}, t_i)$, where i varies from 1 to $|S|$. Then, the forward time-slice estimate $F_{(h_s,h_t)}(X, t)$ of the set S at the spatial location $\overline{X}$ and time t is given by:

$$F_{(h_s,h_t)}(\overline{X}, t) = C_f \cdot \sum_{i=1}^{|S|} K_{(h_s,h_t)}(\overline{X} - \overline{X_i}, t - t_i).$$

Here $K_{(h_s,h_t)}(\cdot, \cdot)$ is a spatiotemporal kernel smoothing function, h_s is the spatial kernel vector, and h_t is temporal kernel width. The kernel function $K_{(h_s,h_t)}(\overline{X} - \overline{X_i}, t - t_i)$ is a smooth distribution that decreases with increasing value of $t - t_i$. The value of C_f is a suitably chosen normalization constant, so that the entire density over the spatial plane is one unit. Thus, C_f is defined as follows:

$$\int_{\text{All } X} F_{(h_s,h_t)}(\overline{X}, t)\delta X = 1.$$

The reverse time-slice density estimate is calculated differently from the forward time-slice density estimate. Assume that the set of points in the time interval $(t, t + h_t)$ is denoted by U. As before, the value of C_r is chosen as a normalization constant. Correspondingly, the reverse time-slice density estimate $R_{(h_s,h_t)}(\overline{X}, t)$ is defined as follows:

$$R_{(h_s,h_t)}(\overline{X}, t) = C_r \cdot \sum_{i=1}^{|U|} K_{(h_s,h_t)}(\overline{X} - \overline{X_i}, t_i - t).$$

In this case, $t_i - t$ is used in the argument instead of $t - t_i$. Thus, the reverse time-slice density in the interval $(t, t+h_t)$ would be exactly the same as the forward time-slice density, if time were reversed, and the data stream arrived in reverse order, starting at $t + h_t$ and ending at t.

The velocity density $V_{(h_s,h_t)}(\overline{X}, T)$ at spatial location $\overline{X}$ and time T is defined as follows:

$$V_{(h_s,h_t)}(\overline{X}, T) = \frac{F_{(h_s,h_t)}(X, T) - R_{(h_s,h_t)}(\overline{X}, T - h_t)}{h_t}.$$

Note that the reverse time-slice density estimate is defined with a temporal argument of $(T - h_t)$, and therefore the future points *with respect to* $(T - h_t)$ are known at time T. A

positive value of the velocity density corresponds to an increase in the data density at a given point. A negative value of the velocity density corresponds to a reduction in the data density at a given point. In general, it has been shown that when the spatiotemporal kernel function is defined as below, then the velocity density is directly proportional to a rate of change of the data density at a given point.

$$K_{(h_s,h_t)}(X,t) = (1 - t/h_t) \cdot K'_{h_s}(X).$$

This kernel function is defined only for values of t in the range $(0, h_t)$. The Gaussian spatial kernel function $K'_{h_s}(\cdot)$ was used because of its well-known effectiveness. Specifically, $K'_{h_s}(\cdot)$ is the product of d identical gaussian kernel functions, and $h_s = (h_s^1, \ldots h_s^d)$, where h_s^i is the smoothing parameter for dimension i.

The velocity density is associated with a data point as well as a time instant, and therefore this definition allows the labeling of both data points and time instants as outliers. However, the interpretation of a data point as an outlier in the context of aggregate change analysis is slightly different from the previous definitions in this section. An outlier is defined on an aggregate basis, rather than in a specific way for that point. Because outliers are data points in regions where abrupt change has occurred, *outliers are defined as data points* $\overline{X}$ *at time instants* t *with unusually large absolute values of the* **local** *velocity density.* If desired, a normal distribution could be used to determine the extreme values among the absolute velocity density values. Thus, the velocity density approach is able to convert the multidimensional data distributions into a quantification that can be used in conjunction with extreme-value analysis.

It is important to note that the data point $\overline{X}$ is an outlier only in the context of *aggregate* changes occurring in its locality, rather than its own properties as an outlier. In the context of the news-story example, this corresponds to a news story belonging to a particular burst of related articles. Thus, such an approach could detect the sudden emergence of local clusters in the data, and report the corresponding data points in a timely fashion. Furthermore, it is also possible to compute the aggregate absolute level of change (over all regions) occurring in the underlying data stream. This is achieved by computing the average *absolute* velocity density over the entire data space by summing the changes at sample points in the space. Time instants with large values of the aggregate velocity density may be declared as outliers.

12.6 Streaming Classification

The problem of streaming classification is especially challenging because of the impact of concept drift. One simple approach is to use a reservoir sample to create a concise representation of the training data. This concise representation can be used to create an offline model. If desired, a decay-based reservoir sample can be used to handle concept drift. Such an approach has the advantage that any conventional classification algorithm can be used since the challenges associated with the streaming paradigm have already been addressed at the sampling stage. A number of dedicated methods have also been proposed for streaming classification.

12.6.1 VFDT Family

Very fast decision trees (VFDT) are based on the principle of *Hoeffding trees*. The basic idea is that a decision tree can be constructed on a sample of a very large data set, using a carefully designed approach, so that the resulting tree is the same as what would have been

achieved with the original data set with high probability. The Hoeffding bound is used to estimate this probability, and therefore the intermediate steps of the approach are designed with this bound in mind. This is the reason that such trees are referred to as *Hoeffding trees*.

The Hoeffding tree can be constructed incrementally by growing the tree simultaneously with stream arrival. An important assumption is that the stream does not evolve, and therefore the currently arrived set of points can be viewed as a sample of the full stream. The higher levels of the tree are constructed at earlier stages of the stream, when enough tuples have been collected to quantify the accuracy of the corresponding split criteria. The lower level nodes are constructed later because statistics about lower level nodes can be collected only after the higher level nodes have been constructed. Thus, successive levels of the tree are constructed, as more examples stream in and the tree continues to grow. The key in the Hoeffding tree algorithm is to quantify the point at which statistically sufficient tuples have been collected in order to perform a split, so that the split is approximately the same as what would have been performed with knowledge of the full stream.

The same decision tree will be constructed on the current stream sample and the full stream, as long as the same splits are used at each stage. Therefore, the goal of the approach is to ensure that the splits on the sample are identical to the splits on the full stream. For ease in discussion, consider the case where each attribute[6] is binary. In this case, two algorithms will produce exactly the same tree, as long as the same split attribute is selected at each point. The split attribute is selected using a measure such as the Gini index. Consider a particular node in the tree constructed on the original data, and the same node constructed on the sampled data. What is the probability that the same attribute will be selected for the stream sample as for the full stream?

Consider the best and second-best attributes for a split, indexed by i and j, respectively, in the sampled data. Let G_i an G_i' be the Gini index values of the split attribute i, as computed on the full stream, and the sampled data, respectively. Because the attribute i was selected for a split in the sampled data, it is evident that $G_i' < G_j'$. The problem is that the sampling might cause an error. In other words, for the *original data*, it might be the case that $G_j < G_i$. Let the difference $G_j' - G_i'$ between G_j' and G_i' be $\epsilon > 0$. If the number of samples n for evaluating the split is large enough, then it can be shown with the use of the Hoeffding bound that the undesirable case where $G_j < G_i$ will not occur with at least a user-defined probability $1-\delta$. The required value of n would be a function of ϵ and δ. In the context of data streams with continuously accumulating samples, the key is to *wait* for a large enough sample size n before performing the split. In the Hoeffding tree, the Hoeffding bound is used to determine the value of n in terms of ϵ and δ as follows:

$$n = \frac{R^2 \cdot \ln(1/\delta)}{2\epsilon^2}. \tag{12.32}$$

The value of R denotes the range of the split criterion. For the Gini index, the value of R is 1, and for the entropy criterion, the value is $\log(k)$, where k is the number of classes. Near ties in the split criterion correspond to small values of ϵ. According to Eq. 12.32, such ties will lead to large sample size requirements, and therefore a larger waiting time until one can be sufficiently confident of performing a split with the available stream sample.

The Hoeffding tree approach determines whether the current difference in the Gini index between the best and second-best split attributes is at least $\sqrt{\frac{R^2 \cdot \ln(1/\delta)}{2n}}$ in order to initiate a split. This provides a guarantee on the quality of a split at a particular node. In cases,

[6]The argument also applies to general attributes by first transforming them to binary data with discretization and binarization.

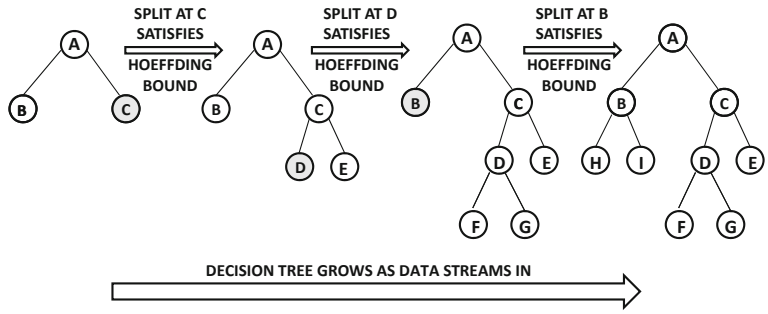

Figure 12.8: Incremental process of Hoeffding tree construction

where there are near ties in split quality (very small values of ϵ), the algorithm will need to wait for a larger value of n until the aforementioned split condition is satisfied. It can be shown that the probability that the Hoeffding tree makes the same classification on the instance as a tree constructed with infinite data is given by at least $1 - \delta/p$, where p is the probability that the instance will be assigned to a particular leaf. The memory requirements are modest because only the counts of the different discrete values of the attributes (over different classes) need to be maintained at various nodes to make split decisions.

The major theoretical implication of the Hoeffding tree algorithm is that one does not need all the data to grow exactly the same tree as what would be constructed by a potentially infinite data stream. Rather, the total number of required tuples is limited once the probabilistic certainty level δ is fixed. The major bottleneck of the approach is that the construction of some of nodes is delayed because of near ties during tree construction. Most of the time is spent in breaking near ties. In the Hoeffding tree algorithm, once a decision is made about a split (and it is a poor one), it cannot be reversed. The incremental process of Hoeffding tree construction is illustrated in Fig. 12.8. It is noteworthy that test instance classification can be performed at any point during stream progression, but the size of the tree increases over time together with classification accuracy.

The *VFDT* approach improves over the Hoeffding tree algorithm by breaking ties more aggressively and through the deactivation of less promising leaf nodes. It also has a number of optimizations to improve accuracy, such as the dropping of poor splitting attributes, and batching intermediate computations over multiple data points. However, it is not designed to handle concept drift. The *CVFDT* approach was subsequently designed to address concept drift. *CVFDT* incorporates two main ideas to address the additional challenges of drift:

1. A sliding window of training items is used to limit the impact of historical behavior.
2. Alternate subtrees at each internal node i are constructed to account for the fact that the best split attribute may no longer remain the top choice because of stream evolution.

Because of the sliding window approach, a difference from the previous method is in the update of the attribute frequency statistics at the nodes, as the sliding window moves forward. For the incoming items, their statistics are added to the attribute value frequencies in the current window, and the expiring items at the other end of the window are removed from the statistics as well. Therefore, when these statistics are updated, some nodes may no longer meet the Hoeffding bound. Such nodes are replaced. *CVFDT* associates each internal node i with a list of alternate subtrees corresponding to splits on different attributes. These

alternate subtrees are grown along with the main tree used for classification. These alternate trees are used periodically to perform the replacement once the best split attribute has changed. Experimental results show that the *CVFDT* approach generally achieves higher accuracy in concept-drifting data streams.

12.6.2 Supervised Microcluster Approach

The supervised microcluster is essentially an instance-based classification approach. In this model, it is assumed that a training and a test stream are simultaneously received over time. Because of concept drift, it is important to adjust the model dynamically over time.

In the nearest-neighbor classification approach, the dominant class label among the top-k nearest neighbors is reported as the relevant result. In the streaming scenario, it is difficult to efficiently compute the k nearest neighbors for a particular test instance because of the increasing size of the stream. However, fine-grained microclustering can be used to create a fixed-size summary of the data stream that does not increase with stream progression. A supervised variant of microclustering is used in which data points of different classes are not allowed to mix within clusters. It is relatively easy to maintain such microclusters with minor changes to the *CluStream* algorithm. The main difference is that data points are assigned to microclusters belonging to the same class during the cluster update process. Thus, labels are associated with microclusters rather than individual data points. The dominant label of the top-k nearest microclusters is reported as the relevant label.

This does not, however, account for the changes that need to be made to the algorithm as a result of concept drift. Because of concept drift, the trends in the stream will change. Therefore, it is more relevant to use microclusters from specific time horizons to increase accuracy. While the most recent horizon may often be relevant, this may sometimes not be the case when the trends in the stream revert back suddenly to older trends. Therefore, a part of the training stream is separated out as the validation stream. Recent parts of the validation stream are utilized as test cases to evaluate the accuracy over different time horizons. The optimal horizon is selected. The k-nearest neighbor approach is applied to test instances over this optimally selected horizon.

12.6.3 Ensemble Method

A robust ensemble method was also proposed for the classification of data streams. The method is also designed to handle concept drift because it can effectively account for evolution in the underlying data. The data stream is partitioned into chunks, and multiple classifiers are trained on each of these chunks. The final classification score is computed as a function of the score on each of these chunks. In particular, ensembles of classification models are scored, such as C4.5, RIPPER, naive Bayesian, from sequential chunks of the data stream. The classifiers in the ensemble are weighted based on their expected classification accuracy under the time-evolving environment. This ensures that the approach is able to achieve a higher degree of accuracy because the classifiers are dynamically tuned to optimize the accuracy for that part of the data stream. It was shown that an ensemble classifier produces a smaller error than a single classifier if the weights of all classifiers are assigned based on their expected classification accuracy.

12.6.4 Massive-Domain Streaming Classification

Many streaming applications contain multidimensional discrete attributes with very high cardinality. In such cases, it becomes difficult to use conventional classifiers because of memory limitations. The count-min sketch can be used to address these challenges. Each class is associated with a sketch that is used to track frequent r-combinations of items in the training data, where r is bounded above by a small number k. For each incoming training data point, all possible r-combinations (for $r \leq k$) are treated as pseudo-items that are added to the sketch of the relevant class. Different classes will have different relevant pseudo-items that will show up in the varying frequencies of the cells belonging to sketches of different classes. These differences can be used to determine the most discriminative cells in the different sketches. The frequent discriminative pseudo-items are determined to create *implicit* rules relating the pseudo-items to the different classes. These rules are implicit because they are not actually materialized, but implicitly stored in the sketches. They are retrieved only at the time of the classification of a test instance. For a given test instance, it is determined, which pseudo-items correspond to the combination of items inside them. The discriminative ones among them are determined by retrieving their statistics from the class-specific sketches. These are then used to perform the classification of the test instance, using the same general approach as a rule-based classifier. The bibliographic notes contain pointers to details of the massive-domain classification work.

12.7 Summary

In this chapter, algorithms for stream mining were presented. Streams present several challenges related to high volume, concept drift, the massive-domain nature of data items, and resource constraints. In this context, synopsis construction is one of the most fundamental problems in the streaming scenario. As long as a high-quality stream synopsis can be constructed, it can be leveraged for stream mining algorithms. The major issue with the use of synopsis methods is that different synopsis structures are suited to different applications. The most common synopsis structures used with data streams are reservoir samples and sketches. Reservoir samples provide the greatest flexibility and should be used where possible.

The core problems of frequent pattern mining, clustering, outlier detection, and classification have also been addressed in the streaming scenario. Most of these problems can be addressed with reservoir sampling effectively, where approximate solutions are desired. In the particular case of outlier detection, numerous variations of the problem definition are possible in the streaming scenario.

12.8 Bibliographic Notes

A detailed discussion of streaming algorithms may be found in [40]. The reservoir-sampling method was originally proposed in [498]. The biased reservoir sampling approach with decay was proposed in [35]. The count-min sketch was described in [165]. Numerous other applications of the count-min sketch are discussed in the same work. The AMS sketch was proposed in [72]. The Flajolet–Martin data structure for distinct element counting was proposed in [208]. A survey of synopsis construction algorithms in data streams is provided in [40]. A detailed discussion of the capabilities of some of these data structures may also be found in the same work.

The lossy frequent itemset counting algorithm was proposed in [376]. Surveys on streaming frequent pattern mining may be found in [34, 40]. The *STREAM* algorithm was proposed in [240]. The massive-domain scenario for stream clustering was addressed in [36]. A survey on stream clustering algorithms may be found in [32]. The *STORM* algorithm for point outlier detection was discussed in [67], and the extension of the *LOF* algorithm to data streams was proposed in [426]. The aggregate change detection algorithm was proposed in [21]. Methods for outlier detection in data streams are discussed in [5]. The *VFDT* and *CVFDT* algorithms were proposed in [176, 279]. The microcluster-based classification method was discussed in [20], and the ensemble method was discussed in [503]. The massive-domain scenario for streaming classification was discussed in [47]. A survey on stream classification methods may be found in [33].

12.9 Exercises

1. Let X be a random variable in $[0, 1]$ with mean of 0.5. Show that $P(X > 0.9) \leq 5/9$.

2. Suppose the standard deviation of a random variable X is r times its mean. Here, r can be any constant. Show how to combine the Chebychev inequality and Chernoff bound to show that repeated i.i.d. samples can be used to create a well-bounded estimate of X. In other words, we would like to create another random variable Z (using the multiple i.i.d. samples) with the same expected value of X, such that for small δ, we would like to show that:

$$P(|Z - E[Z]) > \alpha \cdot E[Z]) \leq \delta$$

(Hint: This is the "mean–median trick" discussed in the chapter.)

3. Discuss scenarios in which both the Hoeffding inequality and the Chernoff bound can be used. Which one applies more generally?

4. Suppose that you have a reservoir of size $k = 1000$, and you have a sample of a stream containing an exactly equal distribution of two classes. Use the upper-tail Chernoff bound to determine the probability that the reservoir contains more than 600 samples of one of the two classes. Can the lower tail be used?

5. (Difficult) Work out the full proof of the biased reservoir sampling algorithm.

6. (Difficult) Work out the proof of correctness of the dot-product estimate obtained with the use of the count-min sketch.

7. Discuss the generality of different synopsis construction methods to various stream mining problems. Why is it difficult to apply these methods to outlier analysis?

8. Implement the *CluStream* algorithm.

9. Extend the implementation of the previous exercise to the problem of classification with the microclustering method.

10. Implement the Flajolet–Martin algorithm for distinct element counting.

11. Suppose that X is a random variable, which always lies in the range $[1, 64]$. Suppose that Y is the geometric mean of a large number n of independent and identical realizations of X. Establish a bound on $\log_2(Y)$. Assume that you know the expected value of $\log_2(X)$.

12. Let Z be a random variable satisfying $E[Z] = 0$, and $Z \in [a, b]$.

(a) Show that $E[e^{t \cdot Z}] \leq e^{t^2 \cdot (b-a)^2/8}$.

(b) Use the aforementioned result to complete the proof of the Hoeffding inequality.

13. Suppose that n distinct items are loaded into a bloom filter of length m with w hash functions.

(a) Show that the probability of a bit taking on the value of 0 is equal to $(1-1/m)^{nw}$.

(b) Show that the probability in (a) is approximately equal to $e^{-nw/m}$.

(c) Show that the expected number of 0-bits m_0 in the bloom filter is related to n, m, and w as follows:

$$n \approx \frac{m \cdot \ln(m/m_0)}{w}$$

14. Show the proof of the bound discussed in the chapter for the count-min sketch when items with negative counts are included in the sketch.

15. Let a single component of an AMS sketch be constructed for each of two streams with the same hash-function. Show that the expected value of the product of these components is equal to the dot product of the frequency vector of distinct items in the two streams.

16. Show that the variance of the square of an AMS sketch component is bounded above by twice the square of the second-order moment of the items in the data stream.

17. Show the correctness of AMS point query frequency estimation methodology discussed in the chapter. In other words, the expected value of the $r_i \cdot Q$ should be equal to the point query result.