Fast Online Computation of the Q_n Estimator with Applications to the Detection of Outliers in Data Streams

Massimo Cafaro[a,*], Catiuscia Melle[a], Marco Pulimeno[a], Italo Epicoco[a]

[a] *University of Salento, Lecce, Italy*

Abstract

We present FQN (Fast Q_n), a novel algorithm for online computation of the Q_n scale estimator. The algorithm works in the sliding window model, cleverly computing the Q_n scale estimator in the current window. We thoroughly compare our algorithm for online Q_n with the state of the art competing algorithm by Nunkesser et al., and show that FQN (i) is faster, requiring only $O(s)$ time in the worst case where s is the length of the window (ii) its computational complexity does not depend on the input distribution and (iii) it requires less space. To the best of our knowledge, our algorithm is the first that allows online computation of the Q_n scale estimator in worst case time linear in the size of the window. As an example of a possible application, besides its use as a a robust measure of statistical dispersion, we show how to use the Q_n estimator for fast detection of outliers in data streams. Extensive experimental results on both synthetic and real datasets confirm the validity of our approach.

Keywords: data streams, sliding window model, Q_n estimator, outliers.

1. Introduction

The Q_n estimator (Rousseeuw & Croux, 1993) is a robust statistical measure of dispersion. Given a set of observations, it provides a scale estimation by determining the first quartile of the pairwise absolute differences of all data values. In statistics, there are two notions of robustness: *resistance* and *efficiency* (Mosteller & Tukey, 1977).

Resistance means that modifying a small part of the observations, even by a very large amount, does not cause the estimate to change by a large amount as well. This property is usually measured by the so-called *breakdown point*, which is defined as the fraction of incorrect observations (with arbitrarily large values) that an estimator can handle before providing an incorrect (arbitrarily large) estimate.

Efficiency means that an efficient statistic provides an estimate very close to its optimal value when the underlying distribution of the observations is known; from this perspective, a robust statistic provides high efficiency under many different conditions and it needs fewer observations than a less efficient one to achieve a given performance.

It is worth noting here that, even though several statistics have one of these two robustness properties, only a few statistics exhibit both resistance and robustness of efficiency. For instance, even though the standard deviation is the most commonly used statistic for scale estimation, and provides an efficient estimate if the observations come from a normal distribution, it is not robust. Indeed, it is enough to change just one value to dramatically alter the corresponding estimate. Therefore, the standard deviation is not resistant. Moreover, it does not provide robustness of efficiency when the observations are not normally distributed.

Among the robust statistics commonly used instead of the standard deviation, we recall here the MAD (Median Absolute Deviation about the median of the data) estimator (Hampel, 1974) and the IQR (interquartile range) (Tukey, 1977).

Given a set of observations, the MAD is the median of the absolute deviations from the median of the observations. Even though the MAD is a robust scale estimator, this statistics is affected by the following limitations: it is not highly

*Corresponding author

Email addresses: massimo.cafaro@unisalento.it (Massimo Cafaro), catiuscia.melle@unisalento.it (Catiuscia Melle), marco.pulimeno@unisalento.it (Marco Pulimeno), italo.epicoco@unisalento.it (Italo Epicoco)

efficient for normally distributed observations and is based on the implicit assumption of symmetry, since it measures the distance from the median, which is a measure of central location.

The IQR is defined as the difference between the 75th and the 25th percentiles, or between upper and lower quartiles: IQR = Q3 - Q1. This estimator is not robust, since its breakdown point is only 25% (Rousseeuw & Croux, 1992).

The Q_n estimator has been proposed by Rousseeuw and Croux and is a better alternative with regard to MAD. It is a robust estimator with a breakdown point equal to 50% and a Gaussian (normal) efficiency of about 82% but, in constrast to MAD, this estimator does not depend on symmetry.

In this paper we are concerned with the problem of computing the Q_n estimator (Rousseeuw & Croux, 1993) online. However, computing the Q_n estimator by directly using its definition is costly, so that we present here FQN (Fast Q_n) a novel algorithm that can be used in a streaming context being fast without sacrificing accuracy, i.e., the resulting algorithm provides a novel way to quickly compute exact Q_n values.

The algorithm works in the sliding window model (Datar, Gionis, Indyk & Motwani, 2002; Muthukrishnan, 2005), in which freshness of recent items is captured either by a time window, i.e., a temporal interval of fixed size in which only the most recent items are taken into account or by an item window, i.e. a window containing a predefined number of recent items. The items in the stream become stale over time, since the window periodically slides forward.

Besides being useful for online computation of exact Q_n values, as an example of possible application of our algorithm, we show how to use it for *anomaly detection*, i.e., we apply it to the problem of detecting outliers in a data stream.

A data stream σ can be thought as a sequence of n items drawn from a universe $\mathcal{U}$. In particular, the items need not be distinct, so that an item may appear multiple times in the stream. Data streams are ubiquitous, and, depending on the specific context, items may be IP addresses, graph edges, points, geographical coordinates, numbers etc.

Since the items in the input data stream come at a very high rate, and the stream may be of potentially infinite length (in which case n refers to the number of items seen so far), it is hard for an algorithm in charge of processing its items to compute an expensive function of a large piece of the input. Moreover, the algorithm is not allowed the luxury of more than one pass over the data. Finally, long term archival of the stream is usually unfeasible. A detailed presentation of data streams and streaming algorithms, discussing the underlying reasons motivating the research in this area is available to the interested reader in (Muthukrishnan, 2005).

With regard to the anomaly detection application, this work is based on the following assumptions: (i) the input is provided in a streaming fashion; (ii) the data are univariate; (iii) the data come from a parametric distribution. In practice, we are given as input a univariate data stream whose items are real or integer numbers coming from an unknown parametric distribution and are asked to determine its outliers.

Given the above assumptions, the most common and suitable outlier detection techniques are based on the so-called *extreme value analysis* (III, 1975), which is designed for one dimensional data (even though it may be generalized to cover multivariate data as well).

We use extreme value analysis along with the Q_n estimator, which is a robust statistical method for univariate data, in order to detect outliers. An outlier is an observation which markedly deviates from other members of the dataset. Looking for outliers means searching for observations which appear to be inconsistent with the rest of the data (Hodge & Austin, 2004). Outliers arise because of human or instrument errors, natural deviations in populations, fraudulent behaviour, changes or system's faults, sampling errors and data processing errors.

Detecting an outlier may indicate a system abnormal running condition such as an engine defect, an anomalous object in an image, an intrusion with malicious intension inside a system, a fault in a production line etc. An outlier detection system accomplishes the task of monitoring data in order to reveal anomalous instances. A comprehensive list of outlier detection use-cases is given in (Hodge & Austin, 2004); here, we briefly recall some of the most important uses.

In order to guarantee the security of computer systems, it is important to collect and analyze different types of data regarding the operating system calls, the network traffic and other user actions. The analysis may reveal that the data show an unusual behaviour owing to malicious activity. The process of recognizing suspect activity is referred to as intrusion detection.

Detecting fraudulent applications of credit cards is becoming increasingly important, since sensitive information related to credit cards can be easily compromised and stolen. The challenge here is to detect unauthorized uses of

credit cards, which may exhibit very different patterns. For instance, this can give rise to a burst of purchases from many different stores from a specific location or, instead, to very few but large transactions.

Another application is related to processing sensor data. Notably, sensors are becoming pervasive as connected sources of real time data. Coupled with the rise of the IoT (Internet of Things) that means an exponential increase related to the availability of streaming data. In particular, many applications in charge of analyzing such data need to to able to detect sudden changes in the underlying patterns, since these changes may refer to events of interest.

For law enforcement outlier detection can provide many insights. In this field possible applications include for instance fraud detection in financial transactions, trading activities and insurance claims. In all of the cases, being able to correctly identify unusual patterns related to criminal offences is difficult, owing to the fact that often these patterns can only be discovered over time through monitoring of multiple actions of an entity.

Regarding health, the goal of many medical applications is to provide patients with a medical diagnosis on the basis of the available data, acquired by multiple devices such as MRI (Magnetic Resonance Imaging) scans, PET (Positron Emission Tomography) scans or ECG (electrocardiogram) time-series. In this case, unusual patterns are often signs of a disease condition.

In Earth Sciences, a huge amount of spatio-temporal data is being collected through different mechanisms such as for instance satellites and remote sensing. These data are related to weather patterns, climate changes, or land-cover patterns. Discovering anomalies may provide significant insights about human activities or environmental trends.

The rest of this paper is organized as follows. Section 2 introduces the Q_n estimator and discusses why it can be used as a robust statistical approach to outlier detection, whilst Section 3 presents related work. We introduce our FQN algorithm in Section 4. The outcomes of the experiments carried out are presented and discussed in Section 5. Finally, we draw our conclusions in Section 6.

2. The Q_n scale estimator and its use for outlier detection

In statistics, Q_n is a robust measure of dispersion (Rousseeuw & Croux, 1993) proposed by Rousseeuw and Croux; it is a rank-based estimator with its statistic based on absolute pairwise differences. The statistic does not require location estimation.

In particular, given a set $\{x_1, x_2, \ldots, x_n\}$, the value of the Q_n statistic was initially defined by the authors as

$$Q_n = 2.2219 \left\{ \left| x_i - x_j \right| ; i < j \right\}_{(k)} \tag{1}$$

where $k \approx \binom{n}{2}/4$ and the notation $\{\cdot\}_{(k)}$ denotes computing the kth order statistics on the set. However, the authors slightly modified the definition in equation (1) by taking into account that

$$\left(\begin{array}{c} h-1 \\ 2 \end{array} \right) + 1 \leq k \leq \left(\begin{array}{c} h \\ 2 \end{array} \right), \tag{2}$$

where $h = \lfloor n/2 \rfloor + 1$. The final definition is

$$Q_n = d_n \, 2.2219 \left\{ \left| x_i - x_j \right| ; i < j \right\}_{(k)}, \tag{3}$$

where $k = \binom{h}{2}$ and d_n is a correction factor which depends on n. The breakdown point (Donoho & Huber, 1983) is one of the most popular measures of robustness of a statistical procedure. Originally introduced for location functionals, it has been generalized to scale, regression and also to other situations. The breakdown point of Q_n is 50%, which means that this estimator is robust enough to counter the negative effects of almost 50% large outliers without becoming extremely biased. Moreover, Q_n exhibits a Gaussian efficiency of about 82%, i.e., it is an efficient estimator since it needs fewer observations than a less efficient one to achieve a given performance. In contrast, the MAD (Median Absolute Deviation about the median of the data) estimator (Hampel, 1974) provides an efficiency of about 36%.

It is worth noting here that for a static dataset of n items the size of the set of the absolute pairwise differences is quadratic in n, so that determining the kth order statistic using a naive approach requires in the worst case $O(n^2 \lg n)$ time by sorting the $O(n^2)$ differences. A better approach consist in using the SELECT algorithm

(Blum, Floyd, Pratt, Rivest & Tarjan, 1973) which is linear in the input size in the worst case, requiring $O(n^2)$. In practice, the QuickSelect algorithm (Floyd & Rivest, 1975; Hoare, 1961) is used instead owing to its speed, despite being linear in the input size only on average (expected computational complexity).

By means of the Q_n estimator it is possible to implement an outlier detector working in a streaming fashion, using a temporal window which slides forward one item at a time. Before providing all of the details, we review here the underlying theory on which our outlier detector is based.

The classical rule for outlier detection based on the z-scores of the observations (Grubbs, 1969) is given by $z_{score} = \frac{|x-\mu|}{\sigma}$ where x denotes the observation under test, and μ and σ denote respectively the mean and the standard deviation of the observations. The z-score rule determines the number of standard deviations by which an observation is distant from the mean. Even though the z-score test assumes that the observations come from a normal distribution, it works well even if this assumption is not verified; anyway, a common technique to solve this problem is to transform the observed data by scaling them.

The z-score test works quite well for observations coming from a univariate parametric distributions, in particular with small or medium sized datasets (it is worth noting here that, even though the stream constituting the input for our algorithm may be potentially of infinite length, we process the stream in the sliding window model, and the fixed dimension of the window is small). Another advantage is that its implementation is simple and fast, making this test suitable in a streaming context.

The outlierness test proposed in (Rousseeuw & Hubert, 2011), makes use of robust estimators such as the median and the MAD (Median Absolute Deviation from the median of the observations) (Hampel, 1974): $|x - median|/\text{MAD}$, where *median* is the median of the observations. Here, we use a slightly modified z-score, in which we substitute the Q_n estimator in place of MAD, obtaining the following outlierness test: $|x - median|/Q_n$. The reasons for preferring the Q_n estimator to MAD are its greater Gaussian efficiency and its ability to deal with skewed distributions (Rousseeuw & Croux, 1993).

Let σ be a stream for which we want to determine outliers. The algorithm, shown in pseudo-code as Algorithm 1 determines the outliers in σ. It takes as input, besides σ_i, which is the ith item arriving from the stream, two parameters w and t representing respectively the semi-window size (the full window size is $2w + 1$) and a real value acting as a multiplier of the Q_n dispersion. In practice, t is used to control the degree of outlierness of an item.

The algorithm processes the stream in windows $W =< \sigma_{i-2w}, \ldots, \sigma_i >$ of size $s = 2w + 1$. Once the first window has been processed, the new one is obtained by sliding the window one item ahead when the next item arrives from the stream. Letting $i - 2w$ be the index of the first item in a window W (i.e., the oldest one), the item under test in W is the one located at the index $i - w$.

For instance, assume that $w = 500$. In this case the first window will contain the items whose index ranges from 1 to 1001, and the first item being considered for outlierness is the item whose index is 501. After processing the window, the new one will contain the items whose index ranges from 2 to 1002 and the second item under test will be the one whose index is 502; and so on.

Denoting the item under test with $x = \sigma_{i-w}$, in order to determine whether x is an outlier we proceed as follows. We begin determining *med*, the value of W corresponding to the median order statistic. Next, we compute q, the Q_n dispersion for the window W. Then, we check the following condition (corresponding to the devised outlierness test): $|x - med| > t \cdot q$; if it is true, then x is an outlier, otherwise x is an *inlier* (i.e., a normal observation). In practice, the condition $|x - med| > t \cdot q$ identifies as outliers those points that are not within t times the Q_n dispersion from the sample median; regarding t, a commonly used value is $t = 3$.

The worst case complexity of Algorithm 1 for processing a single window is $O(s) + O(s \lg s) + O(1) = O(s \lg s)$. Indeed, determining the median of the window requires $O(s)$ (by using the QuickSelect algorithm), computing the Q_n dispersion value requires in the worst case $O(s \lg s)$ (by using the Croux and Rousseeuw (Croux & Rousseeuw, 1992) algorithm). Finally, the check for outlierness of an item can be done in $O(1)$ constant time. However, this is a basic, naive algorithm for computing the Q_n estimator. We shall show how to improve the complexity of Algorithm 1 from $O(s \lg s)$ to $O(s)$ worst case time by using our FQN algorithm to compute the Q_n estimator instead of the Croux and Rousseeuw algorithm.

Algorithm 1 Outlier Detection Using the Q_n estimator

Require: σ, the input stream; w, semi-window size; t, multiplier of the Q_n dispersion
  $Outliers \leftarrow \emptyset$
  for each window W of size $s = 2w + 1$ in σ **do**
    $x \leftarrow \sigma_{i-w}$
    $med \leftarrow \textsc{Median}(W)$
    $q \leftarrow \textsc{Qn}(W)$
    if $|x - med| > t \cdot q$ **then**
      $Outliers \leftarrow Outliers \cup x$
    end if
  end for
  return $Outliers$

3. Related Work

In this Section, we recall the most important algorithms that have been proposed for computing the Q_n estimator. An offline algorithm with worst case complexity $O(s \lg s)$ was proposed by Croux and Rousseeuw (Croux & Rousseeuw, 1992).

Their algorithm is based on a previous work of Johnson and Mizoguchi (Johnson & Mizoguchi, 1978) that allows determining the kth order statistic in a matrix of the form

$$U = X + Y = \left\{x_i + y_j; 1 \leq i, j \leq s\right\}, \tag{4}$$

which is required to have nonincreasing rows and columns.

To this end, both vectors X and Y are sorted using $O(s \lg s)$ time in the worst case. Then, the matrix U of order s is used, without being actually computed, as follows. Two arrays *left* and *right* are defined, in order to keep track of the numbers on the ith row of the matrix that must still being considered as potential candidates for being the kth order statistic. The set C of potential candidates is defined as

$$C = \left\{U_{ij};\ left\ (\mathrm{i}) \leq j \leq\ right\ (\mathrm{i}); 1 \leq i \leq s\right\}. \tag{5}$$

In practice, a pruning strategy allows discarding those numbers that can not be the kth order statistic. In each step $left(i)$ is made greater and $right(i)$ smaller by comparison with the weighted median of the medians of the rows in C (with weight equal to their length). Since each step requires $O(s)$ and there are $O(\lg s)$ steps, the worst case time required is $O(s \lg s)$.

In order to compute the Q_n estimator, Croux and Rousseeuw noted that

$$\left\{\left|x_i - x_j\right|; i < j\right\}_{(k)} = \left\{x_{(i)} - x_{(s-j+1)}; 1 \leq i, j \leq s\right\}_{(k^*)} \tag{6}$$

where $k^* = k + s + \binom{s}{2}$.

Here, $x_{(1)} \leq \ldots \leq x_{(s)}$ are the sorted observations (we recall that $x_1, \ldots, x_s$ are the unsorted observations), so that defining $X = \{x_{(1)}, \ldots, x_{(s)}\}$ and $Y = \{-x_{(s)}, \ldots, -x_{(1)}\}$, they can apply the Johnson and Mizoguchi algorithm to the matrix obtained taking into account the observations whose indexes are such that $1 \leq i, j \leq s$:

$$U = X + Y = \left(x_{(i)} - x_{(s-j+1)}\right),\ 1 \leq i, j \leq s. \tag{7}$$

Croux and Rousseeuw therefore use a different sorting order for the X and Y vectors in contrast to Johnson and Mizoguchi: these vectors are in nondecreasing order, whilst Johnson and Mizoguchi algorithm requires nonincreasing order. As a consequence, the virtual matrix U in the case of Croux and Rousseeuw exhibits both nondecreasing rows and columns, and the area of interest (containing the order statistic to be found) lies in the lower triangular matrix with regard to the antidiagonal. The upper triangular matrix with regard to the antidiagonal can be ignored (it contains negative or zero values); the antidiagonal can be ignored as well since it contains zeros. Therefore, the arrays *left* and *right* are initialized as follows: $left(\mathrm{i}) = s - i + 2$ and $right(\mathrm{i}) = s$, for all $i \geq 2$.

To recap, the aim is to search for the kth order statistic in a set containing $s(s-1)/2$ items. But, the search happens in a virtual matrix of s^2 items, of which $(s+1)s/2$ must be discarded (the ones related to the upper triangle with regard to the antidiagonal). Therefore, instead of searching for k, Croux and Rousseeuw search for the $k^* = k + s + \binom{s}{2}$ order statistic.

In (Nunkesser, Schettlinger & Fried, 2008), the authors propose a streaming algorithm for the Q_n estimator, that we denote as NUNKESSER. This algorithm handles a sliding window in which a new, incoming observation is added whilst the oldest observation is removed. This process is called a window's *update*. In order to compute the Q_n estimator during an update, they reuse the same consideration of Croux and Rousseeuw: given $X = \{x_1, \ldots, x_s\}$, $k' = \binom{\lfloor s/2 \rfloor + 1}{2}$ and $k = k' + s + \binom{s}{2}$, it holds that

$$\left\{ \left| x_i - x_j \right|, i < j \right\}_{(k')} = \left\{ x_{(i)} - x_{(s-j+1)}, 1 \leq i, j \leq s \right\}_{(k)}. \tag{8}$$

As a consequence, one must compute the kth order statistic of $U = X + (-X)$.

The NUNKESSER algorithm maintains a buffer $\mathcal{B}$ of size $b = O(s)$ that stores matrix items $u_{(k-\lfloor (b-1)/2 \rfloor)}, \ldots, u_{(k+\lfloor b/2 \rfloor)}$, centered on the kth order statistic. Initially, $\mathcal{B}$ is populated determining its items along with the kth order statistic through an adapted version of the Croux and Rousseeuw algorithm. The main data structures are AVL trees, which are balanced trees allowing inserting, deleting, finding and determining the rank of an item in $O(\lg s)$ time. These trees are used to store X, $-X$ and the buffer $\mathcal{B}$. Each time an item is deleted or inserted using the authors' procedures for these tasks, the new position of the kth order statistic in $\mathcal{B}$ is determined. The authors return the new solution or recompute $\mathcal{B}$ using the offline algorithm of Croux and Rousseeuw if the kth order statistic is not in $\mathcal{B}$ any more.

Clearly, the worst case running time of this algorithm is $O(s \lg s)$. However, the authors prove (see Theorem 1 in (Nunkesser, Schettlinger & Fried, 2008)) that "for a constant signal with stationary noise, the expected amortized time per update is $O(\lg s)$". We remark here that, in order to achieve this expected amortized time, the authors assume that the rank of each data point in the set of all data points is equiprobable. In this paper, we show how to dynamically maintain and process each of the windows originating from the input data stream in $O(s)$ worst case time. However, no assumption is made regarding the data points in each of the windows, so that our algorithm is far more general. Even though the expected amortized time per update of NUNKESSER is better than the worst case $O(s)$ running time of our algorithm, we shall show in Section 5 that FQN outperforms NUNKESSER.

4. The FQN Algorithm

Our FQN algorithm computes the Q_n estimator in a streaming fashion, without assuming anything related to the underlying distribution of the input stream. FQN works by dynamically maintaining and processing the consecutive windows originating from the input data stream. The key idea is to maintain the current window sorted. To this aim, we mimic the way InsertionSort (Cormen, Leiserson, Rivest & Stein, 2009) inserts an item.

InsertionSort requires in the worst case $O(s^2)$ to sort s items, but we do not use it to sort the windows arising from the input stream. Each time a new item arrives, we form a new window in two steps. First, we remove the least recent (in the temporal sequence of item arrivals) item. Since the previous window was already sorted, removing the least recent item leaves the window sorted. Now, we insert the incoming item using the InsertionSort insertion procedure, which requires $O(s)$ worst case time.

We need to simultaneously maintain two different permutations of the current window. One is given by the actual order in which the items arrive from the stream, the other is the sorted permutation of the items in the window. We use the notation W to denote the current window and σ_i to denote the ith item in the input stream (temporal order). The size of W is $s = 2w + 1$, where w is the semi-window size and the items belonging to W after the insertion of the item σ_i are those related to the sub-stream $\sigma_{i-2w}, \ldots, \sigma_i$. Moreover, we denote by Π the permutation of the items in W in which the items are in sorted order. Π stores the items $[\pi_1, \ldots, \pi_s]$.

Initially, the window W is empty. We insert the items in W one at a time, building the window W; we also insert the items in Π, preserving the sorted order by means of the InsertionSort insertion procedure. After inserting $s = 2w + 1$ items, the window W is full. Next, when the item σ_i arrives, we insert it into the current window and process the resulting window computing the Q_n estimator.

Computing the kth order statistic of the absolute pairwise differences can be done in worst case $O(s)$ time as well. Following the same ideas discussed in previous work, we do not actually compute the $O(s^2)$ differences. Instead, we

determine the order statistic by using the algorithm proposed by Mirzaian and Arjomandi (Mirzaian & Arjomandi, 1985), which works as follows. Let A be a matrix of real numbers, whose order is s and in which the rows are sorted in descending order and the columns are sorted in ascending order. Moreover, let $\overline{s} = \lceil \frac{1}{2}(s+1) \rceil$. Then $\overline{A}$ is a submatrix of A of order $\overline{s}$, consisting of the odd indexed rows and columns (plus the last row and columns of A if s is even). Letting L be a list of reals and a a real number, the $rank^+$ and $rank^-$ of a in the list L are defined as follows:

$$rank^+(L, a) = |\{x \in L : x > a\}|; \tag{9}$$

$$rank^-(L, a) = |\{x \in L : x < a\}|. \tag{10}$$

For $1 \leq k \leq |L|$, a is the kth smallest item of L if and only if $rank^-(L, a) \leq k - 1$ and $rank^+(L, a) \leq |L| - k$. The selection algorithm is based on Theorem 3.1 in (Mirzaian & Arjomandi, 1985), which states that, given the matrices A and $\overline{A}$, for any real number a it holds that (i) $rank^-(A, a) \leq 4\ rank^-(\overline{A}, a)$ and (ii) $rank^+(A, a) \leq 4\ rank^+(\overline{A}, a)$.

Determining $rank^-(A, a)$ can be done in $O(s)$ taking advantage of the fact that the rows and columns of A are sorted respectively in descending and ascending order. Algorithm 2 shows how to compute $rank^-(A, a)$. Similarly, $rank^+(A, a)$ can be determined in $O(s)$ as well.

Algorithm 2 Determining $rank^-(A, a)$

Require: A, a matrix of order s, with rows and columns sorted respectively in descending and ascending order; a, a real number

```
j ← 1
x ← 0
for i = 1 to s do
    while j ≤ s and A_{i,j} ≥ a do
        j ← j + 1
    end while
    x ← x + s − j + 1
end for
return x
```

To select the kth item, the algorithm determines two items a and b with $a \geq b$ from $\overline{A}$. Letting z denote the kth order statistic of A, the algorithm ensures that (i) $b \leq z \leq a$ and (ii) the number of items of A whose value is less than a and greater than b is $O(s)$. The function `MAselect` (Mirzaian and Arjomandi Select), shown in pseudocode as Algorithm 3 determines the kth item of A in $O(s)$.

The `MAselect` function simply calls the `biselect` function with parameters s, A, k_1 and k_2, with $k_1 \geq k_2$. The pair (x, y) is returned, so that x is the k_1th item of A whilst y is the k_2th item.

Defining

$$\overline{k_1} = \begin{cases} s + 1 + \lceil \frac{1}{4} k_1 \rceil & \text{if } s \text{ is even} \\ \lceil \frac{1}{4} k_1 + 2s + 1 \rceil & \text{if } s \text{ is odd} \end{cases} \tag{11}$$

and

$$\overline{k_2} = \left\lfloor \frac{1}{4} (\mathrm{k}_2 + 3) \right\rfloor \tag{12}$$

$\overline{k_1}$ is the smallest integer such that the $\overline{k_1}$th item of $\overline{A}$ is at least as large as the k_1th item of A, and $\overline{k_2}$ is the largest integer such that the $\overline{k_2}$th item of $\overline{A}$ is no larger than the k_2th item of A.

When the matrix A is of the form $X + (-X)$ as in our algorithm, only X needs to be stored in memory, i.e., the items of A are computed when they are actually needed, so that only a small fraction of A is used ($O(s)$ instead of $O(s^2)$ items).

The matrix is derived by the array X which is in nondecreasing order, and by the array $-X$ which is in noincreasing order. Owing to the different orders of X and $-X$, the matrix A contains nonincreasing rows and nondecreasing columns. The area of interest is the lower triangle with regard to the main diagonal. Therefore, the Mirzaian and Arjomandi algorithm is applied taking into account an offset value $s + \binom{s}{2}$ as in the case of Croux and Rousseeuw, in order to limit the computation only to the lower triangle of the virtual matrix A. The `rank` and `pick` procedure are modified accordingly to achieve this goal.

Algorithm 3 MASelect

Require: A, a matrix of order s, with rows and columns sorted respectively in descending and ascending order; k, an integer number
  $(x, y) \leftarrow \text{BISELECT}(s, A, k, k)$
  return x

Algorithm 4 Biselect

Require: s, order of matrix A; A, a matrix with rows and columns sorted respectively in descending and ascending order; k_1, an integer; k_2, an integer
  if $s \leq 2$ **then**
    $(x, y) \leftarrow (k_1\text{th of } A, k_2\text{th of } A)$
  else
    $(a, b) \leftarrow \text{BISELECT}(\overline{s}, \overline{A}, \overline{k_1}, \overline{k_2})$
    $ra^- \leftarrow rank^-(A, a)$
    $rb^+ \leftarrow rank^+(A, b)$
    $L \leftarrow \{A_{ij} : b < A_{ij} < a\}$
    if $ra^- \leq k_1 - 1$ **then**
      $x \leftarrow a$
    else
      if $k_1 + rb^+ - s^2 \leq 0$ **then**
        $x \leftarrow b$
      else
        $x \leftarrow \text{QUICKSELECT}(L, k_1 + rb^+ - s^2)$
      end if
    end if
    if $ra^- \leq k_2 - 1$ **then**
      $y \leftarrow a$
    else
      if $k_2 + rb^+ - s^2 \leq 0$ **then**
        $y \leftarrow b$
      else
        $y \leftarrow \text{QUICKSELECT}(L, k_2 + rb^+ - s^2)$
      end if
    end if
  end if
  return (x, y)

In FQN updating the windows works as follows. The permutation Π is already sorted. Each time a new item arrives from the stream, the oldest one is removed and the new one is inserted in both W and Π. In particular, inserting the new item in Π in its correct position is done by using the INSERTIONSORT insertion procedure. Then, we determine the Q_n estimator for the current window as previously described.

The pseudo-code of Algorithm 5 describes how our FQN algorithm works for online computation of exact Q_n values. The same algorithm may possibly be used for outlier detection as follows. In the loop, after inserting the

item σ_i into Π, the item x to be checked is identified as $x \leftarrow \sigma_{i-w}$. The median can be easily obtained in $O(1)$ worst case time as $med \leftarrow \pi_{w+1}$, since the Π permutation is sorted. Once the Q_n value has been determined, the condition $|x - med| > t \cdot Q_n$ identifies as outliers, again in $O(1)$ worst case time, those points that are not within t times the Q_n dispersion from the median.

Algorithm 5 Fast Q_n

Require: σ_i, the current item;
  for each item σ_i **do**
    delete σ_{i-2w-1} from W and Π
    insert σ_i into W
    insert σ_i into Π using InsertionSort insertion
    $stat \leftarrow$ MASelect(Π)
    $Q_n \leftarrow d_n \cdot 2.2219 \cdot stat$
  end for
  return Q_n

Regarding our implementation, the main data structures are two arrays: one is a circular buffer, used to guarantee a consistent temporal order for the items σ_i arriving from the stream, the other is a sorted array representing Π, which is updated by means of the streaming InsertionSort procedure.

Algorithm 5 correctly determines the Q_n estimator in worst case time and space $O(s)$. Indeed, we process the input stream by handling the current sliding window W and maintaining in sorted order, through the use of incremental InsertionSort, the corresponding permutation Π. In particular, Π must be sorted as required by the Mirzaian and Arjomandi algorithm, which is used to determine the order statistics required for computing the Q_n value.

5. Experimental Results

In this Section, we present and discuss experimental results, thoroughly comparing FQN against Nunkesser et al. algorithm, that we denote as NUNKESSER. Since both algorithms correctly determine the Q_n values, we shall compare the algorithms only with regard to their performances; in particular, we take into account the number of *updates per second*.

The FQN and NUNKESSER algorithms have been implemented in C. The source code has been compiled using the Intel C compiler v19.0.4.243 on linux CentOS 7 with the following flags: -O3 -std=c99. The tests have been carried out on a workstation equipped with 64 GB of RAM and two 2.0 GHz exa-core Intel Xeon CPU E5-2620 with 15 MB of cache level 3. The source code is freely available for inspection and for reproducibility of results[1]. The tests have been performed on both synthetic and real datasets.

5.1. Synthetic datasets

Synthetic datasets consist of items generated according to the distributions shown in Table 1.

For each distribution, the algorithms have been executed three times and we report here the mean number of updates per second varying w, the semi-window size from 100 to 500 in steps of 100. We fix the number of items to be processed (i.e., checked to verify if they are outliers) to 100000. Of course, for a given value of w, in order to process 100000 items, the dataset length must be $100000 + 2w + 1$.

Results are depicted in Figure 1. As shown, FQN clearly outperforms NUNKESSER in all of the experiments with the only notable exception related to the uniform distribution. As discussed in Section 3, Nunkesser et al. proved that *for a constant signal with stationary noise, the expected amortized time per update is* $O(\lg s)$. This bound on the expected amortized time, requires the assumption that the rank of each data point in the set of all data points is equiprobable. Clearly, this is the case for the uniform distribution. On other distributions this strong assumption is not satisfied, so that the NUNKESSER algorithm is subject to its worst case running time, which is $O(s \lg s)$. On the contrary, FQN does

[1] https://github.com/cafaro/FQN

Table 1: Synthetic data: experiments carried out

Distribution	Parameters
beta	$\alpha = 2, \beta = 1/4$
chi-squared	$\nu = 3$
exponential	$\lambda = 1/2$
gamma	$\alpha = 1, \beta = 2$
half-normal	$\theta = 1/2$
inverse gaussian	$\mu = 2, \lambda = 1$
log-normal	$\mu = 1, \sigma = 3$
normal	$\mu = 1, \sigma = 3$
Pareto	$k = 3, \alpha = 0.75$
Poisson	$\mu = 3$
uniform	$min = 0, max = 100000$
Zipf	$n = 100000000, \rho = 1.2$

not make any assumption on the underlying input distribution, and can dynamically maintain and process each of the windows in $O(s)$ worst case time.

We thoroughly analyze the NUNKESSER algorithm in Figure 2. We report the size of the buffer $\mathcal{B}$ and the percentage of executions of the Croux and Rousseeuw algorithm; in particular, besides the normal distribution, we deal here only with the following distributions: log-normal, Poisson and Zipf. The results obtained for the remaining distributions are similar and we do not report them in order to save space. As shown, the running time of NUNKESSER can be ascribed to two main factors: the dimension of the buffer $\mathcal{B}$ and the number of executions of the Croux and Rousseeuw algorithm, which is executed when the kth order statistic is not found within the buffer.

Two different behaviours are clearly depicted in the plots. For continuous distributions (log-normal and normal) the buffer size is linear in s so that when the kth order statistic is within the buffer, it can be determined quickly. Otherwise, NUNKESSER executes the Croux and Rousseeuw algorithm, which is $O(s \lg s)$ in the worst case. The percentage of executions, as shown, is not negligible and is the main factor affecting the overall running time. For discrete distributions (Poisson and Zipf), whose number of distinct items is much smaller than in the continuous case, the buffer size exhibits a quadratic increase with regard to s. In particular, all of the time is spent searching for the kth order statistic within a huge buffer. Indeed, as can be seen in the plots, for both the Poisson and Zipf distributions, the Croux and Rousseeuw algorithm is never executed.

For completeness, we also discuss here a variation proposed by Nunkesser et al. in their paper (in Section 2.1 Online Algorithm). Indeed, they state: *We may also introduce bounds on the size of $\mathcal{B}$ in order to maintain linear size and to recompute $\mathcal{B}$ if these bounds are violated*. We note here that in their paper Nunkesser et al. do not provide any result regarding this variation.

We have implemented and tested this variation, denoted by NUNKESSER B, in which we maintain the size of $\mathcal{B}$ linear by imposing the constraint that the buffer size can not exceed $2s$. The experimental results show that the performances of this variation are slightly worse with regard to the original algorithm on all of the input distributions but the Poisson and Zipf in which the variation provides better results. However, in all of the cases, our FQN algorithm always outperforms this variation of the NUNKESSER algorithm. In Figure 3, we depict the results for the log-normal, normal, Poisson and Zipf distributions.

A detailed analysis of NUNKESSER B with limited buffer $\mathcal{B}$ is shown in Figure 4. We only report the results obtained for the log-normal, normal, Poisson and Zipf distributions. As shown, since the maximum buffer size is limited to $2s$ we report the mean buffer size. For the continuous distributions the mean buffer size is linear in s as expected. For the Poisson and Zipf distributions, the mean buffer size is zero: in practice, for these distributions the buffer is never used and the Croux and Rousseeuw algorithm is always executed.

Finally, regarding the space used, our algorithm only needs to store two arrays of size s, the circular buffer and the sorted array representing Π which takes on the role of X and is used as input to the `MAselect` procedure. Therefore, FQN requires $O(s)$ space. As shown, depending on the input distribution, NUNKESSER may require instead up to $O(s^2)$

Table 2: Statistical characteristics of the real datasets

Name	Min	Max	Mean	Median	stdDev	Skew
Accidents	1	275	46.1599	33	37.7459	1.63804
Kosarak	1	11018	1758.17	632	2433.03	1.73603
Nasa	0	28474	480.919	190	923.676	5.60017
Q148	15	149464496	4570.15	63	471361	315.665
Retail	0	8563	1967.7	1274	2026.74	1.07606
Webdocs	1	14842	2688.35	1181	3414.45	1.75281

space, whilst the variation NUNKESSER B in which the buffer is restricted to be of size at most $2s$ requires $O(s)$ space but provides worst performances for the majority of the input distributions. From a practical perspective, NUNKESSER needs to maintain three data structures. These are three AVL trees, one for the X array ($O(s)$ space), one for the Y array ($O(s)$ space) and one for the buffer $\mathcal{B}$ (with space required between $\Omega(s)$ and $O(s^2)$). Besides the actual values, these trees also need to store several pointers, consuming additional space.

5.2. Real datasets

The real datasets have been selected, among the ones which are publicly available, in order to represent a variety of different application domains. Moreover, some of these datasets are widely used for many different tasks in the data mining literature. Each of the original datasets has been pre-processed when necessary, and only the first 1,001,001 entries were retained and used for the tests. We report the most important statistical information characterizing the datasets obtained after the pre-processing step in Table 2. A description of the datasets follows.

Accidents: This dataset contains (anonymized) traffic accident data. In particular, the data have been obtained from the National Institute of Statistics (NIS) for the region of Flanders (Belgium) for the period 1991-2000. More specifically, the data are obtained from the Belgian Analysis Form for Traffic Accidents that should be filled out by a police officer for each traffic accident that occurs with injured or deadly wounded casualties on a public road in Belgium.

Kosarak: This is a click-stream dataset of a Hungarian online news portal. It has been anonymized, and consists of transactions, each of which is comprised of several integer items.

Nasa: Compliments of NASA and the Voyager 2 Triaxial Fluxgate Magnetometer principal investigator, Dr. Norman F. Ness, this dataset contains several data. We selected the Field Magnitude (F1) and Field Modulus (F2) attributes from the Voyager 2 spacecraft Hourly Average Interplanetary Magnetic Field Data. A pre-processing step was required for this dataset: having selected the data for the years 1977-2004, we removed the unknown values (marked as 999), and multiplied all of the values by 1000 to convert them to integers (since the original values were real numbers with precision of 3 decimal points). The values of the two attributes were finally concatenated. In our experiments, we read all of the values of the attribute F1, followed by all of the values of the attribute F2.

Q148: Derived from the KDD Cup 2000 data, compliments of Blue Martini, this dataset contains several data. The ones we use for our experiments are the values of the attribute Request Processing Time Sum (attribute number 148), coming from the "clicks" dataset. The pre-processing phase required replacing of all the missing values (appearing as question marks) with the value 0.

Retail: This dataset contains retail market basket data coming from an anonymous Belgian store.

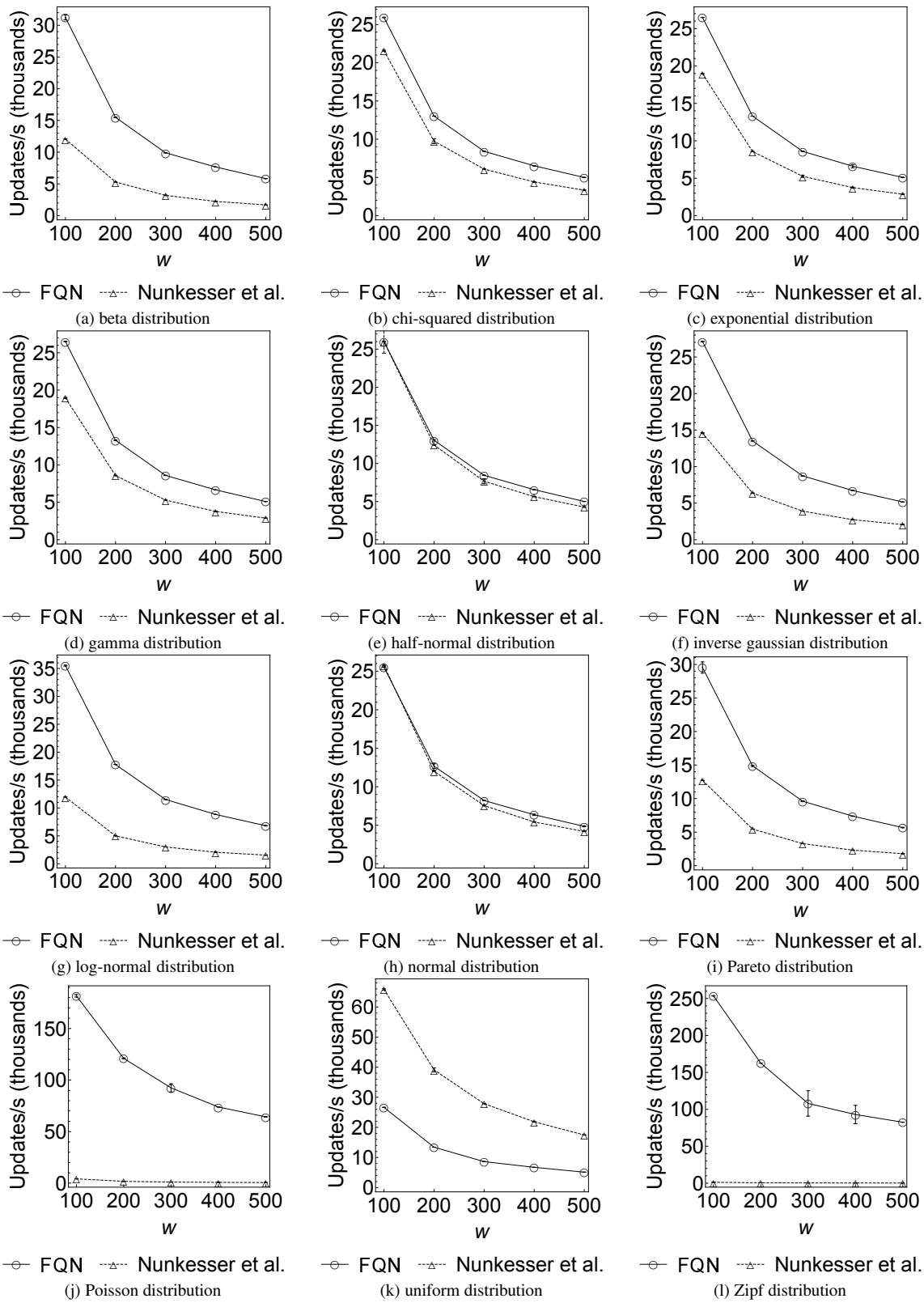
Webdocs: This dataset derives from a spidered collection of web html documents. All the web documents were preliminarly filtered by removing HTML tags and the most common words (stopwords), and by applying a stemming algorithm. Then, a distinct transaction containing the set of all the distinct terms (items) appearing within the document itself has been generated from each document.

The experiments carried out on these real datasets are the same executed on synthetic data, that is, varying the window size. The FQN algorithm was compared against the NUNKESSER algorithm and its variant NUNKESSER B, in which a fixed memory bound is used. For all the real datasets under test, the experimental results clearly show that FQN outperforms both NUNKESSER and its variant NUNKESSER B. In particular, Figures 5 and 6 depict respectively the number of updates per second (in thousands) and the actual running time (in seconds). We observe that FQN provides

at least two times the number of updates per second (respectively, concerning the running time it is at least two times faster) with regard to NUNKESSER and NUNKESSER B.

6. Conclusions

We have introduced FQN (Fast Q_n), a novel algorithm for online computation of the Q_n scale estimator, and showed a possible application in the context of outlier detection in data streams. Our algorithm works in the sliding window model, by cleverly computing the Q_n scale estimator in the current window. We have shown, through extensive experimental results on both synthetic and real datasets, that our algorithm for online Q_n is faster than the state of the art competing algorithm by Nunkesser et al. To the best of our knowledge, our algorithm is the first that allows online computation of the Q_n scale estimator in worst case time linear in the size of the window. Moreover, the computational complexity of FQN does not depend on the input distribution. Finally, our algorithm requires less space.



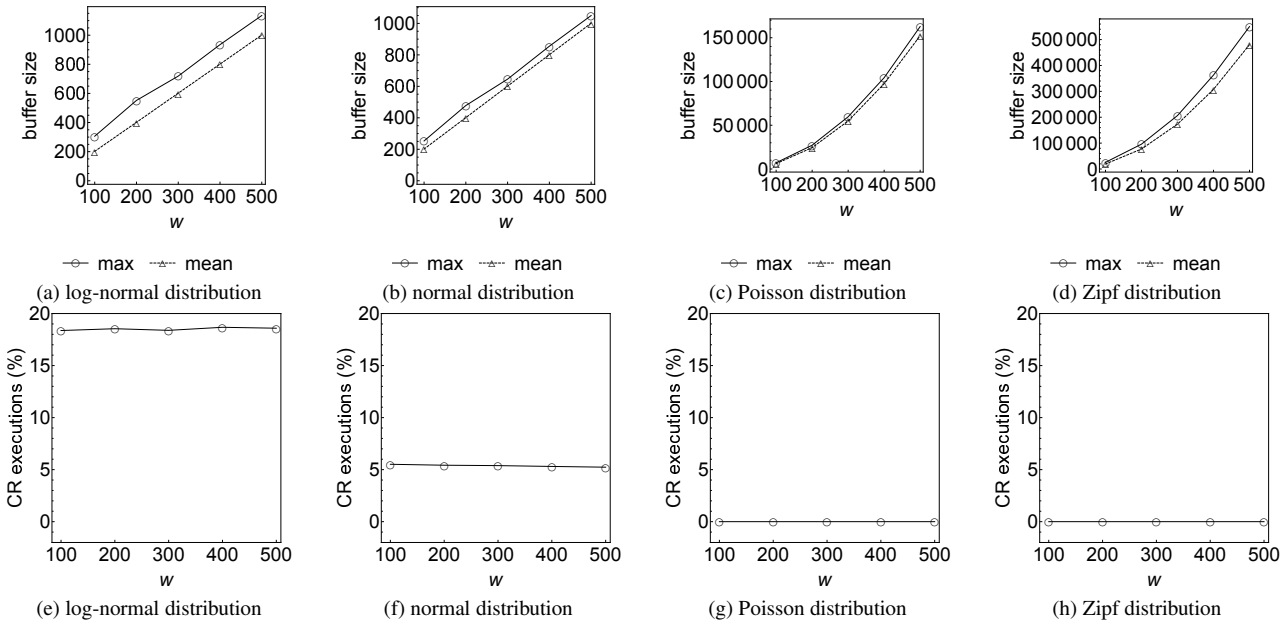
(a) beta distribution (b) chi-squared distribution (c) exponential distribution

(d) gamma distribution (e) half-normal distribution (f) inverse gaussian distribution

(g) log-normal distribution (h) normal distribution (i) Pareto distribution

(j) Poisson distribution (k) uniform distribution (l) Zipf distribution

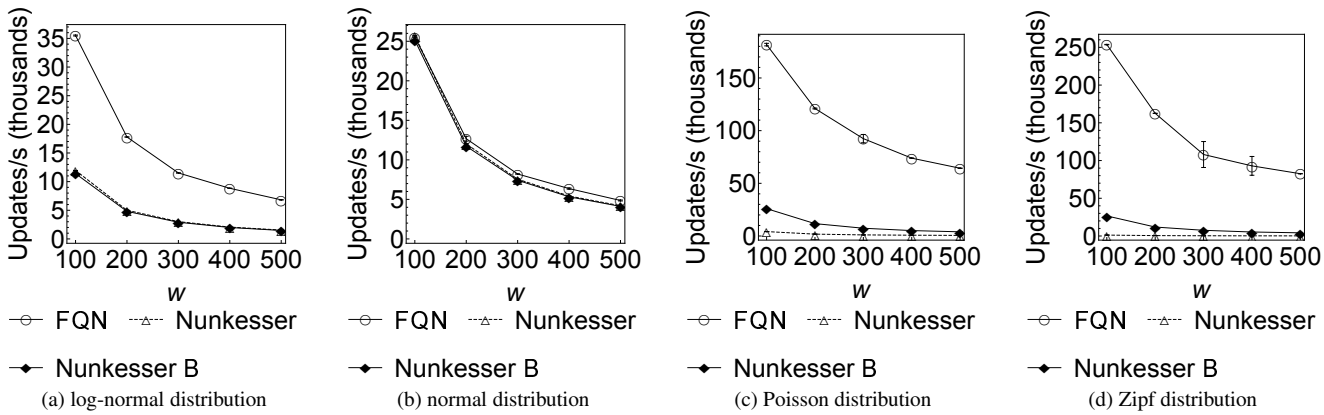
Figure 1: Updates per second (mean and confidence interval)



(a) log-normal distribution (b) normal distribution (c) Poisson distribution (d) Zipf distribution

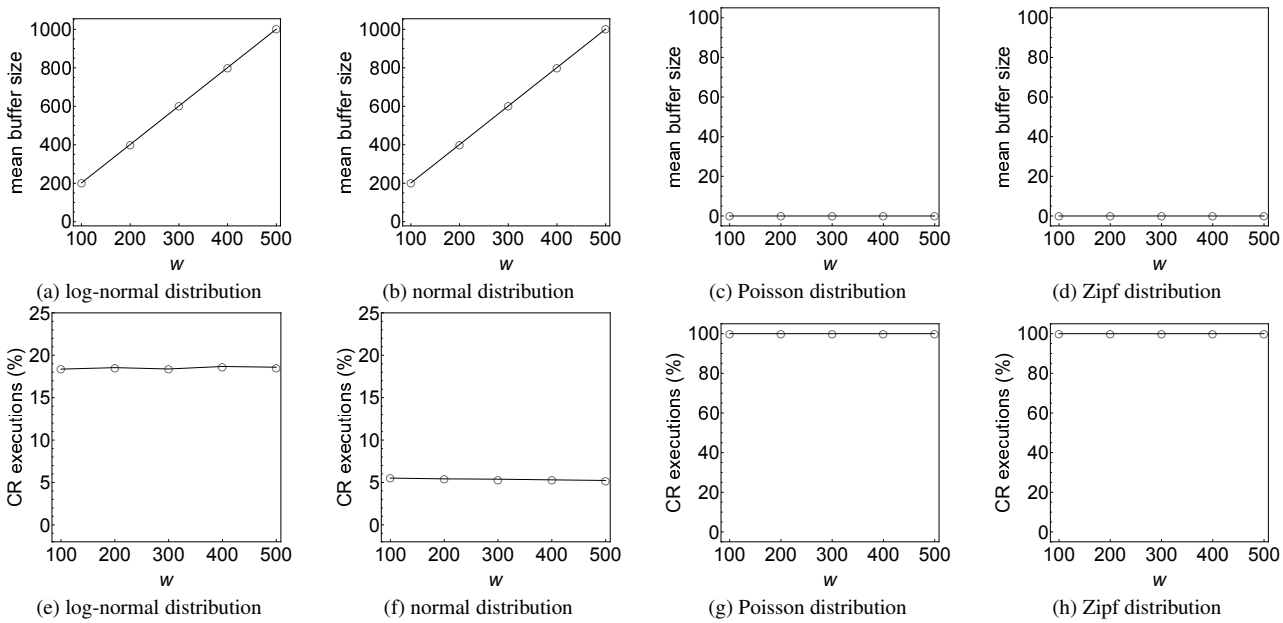
(e) log-normal distribution (f) normal distribution (g) Poisson distribution (h) Zipf distribution

Figure 2: Detailed analysis of NUNKESSER algorithm



(a) log-normal distribution (b) normal distribution (c) Poisson distribution (d) Zipf distribution

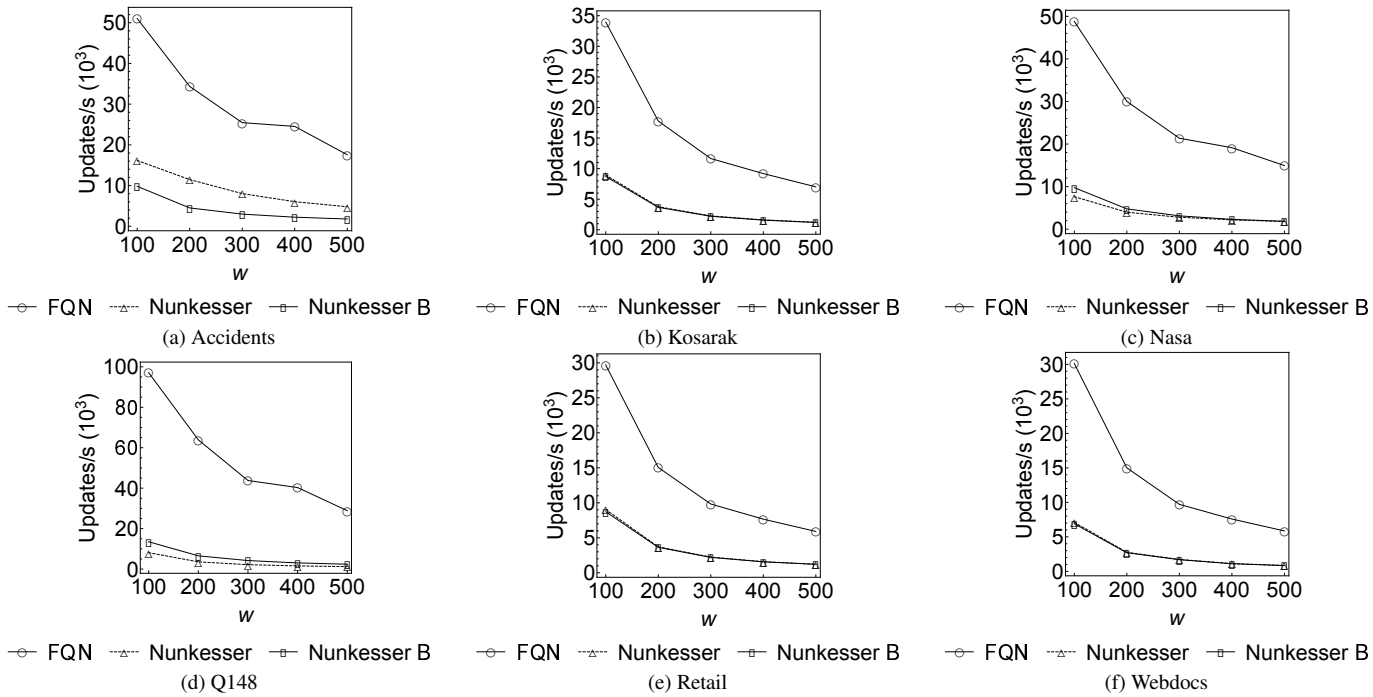
Figure 3: Updates per second including NUNKESSER algorithm with limited buffer $\mathcal{B}$ (mean and confidence interval)



(a) log-normal distribution (b) normal distribution (c) Poisson distribution (d) Zipf distribution

(e) log-normal distribution (f) normal distribution (g) Poisson distribution (h) Zipf distribution

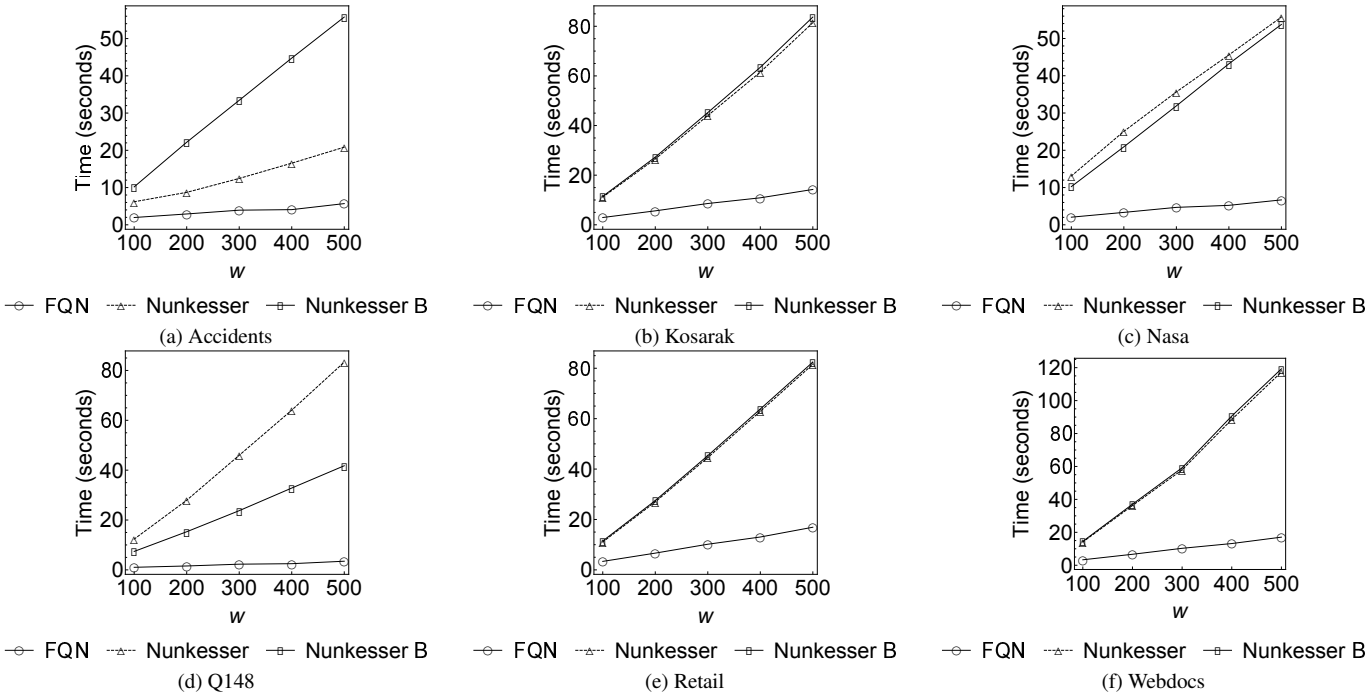
Figure 4: Detailed analysis of NUNKESSER algorithm with limited buffer $\mathcal{B}$



(a) Accidents (b) Kosarak (c) Nasa

(d) Q148 (e) Retail (f) Webdocs

Figure 5: Updates per seconds on real datasets, varying the window size



(a) Accidents (b) Kosarak (c) Nasa

(d) Q148 (e) Retail (f) Webdocs

Figure 6: Running times (in seconds) on real datasets, varying the window size

References

Blum, M., Floyd, R. W., Pratt, V. R., Rivest, R. L., & Tarjan, R. E. (1973). Time bounds for selection. *J. Comput. Syst. Sci.*, *7*(4), 448–461.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

Croux, C. & Rousseeuw, P. J. (1992). Time-efficient algorithms for two highly robust estimators of scale. In Dodge, Y. & Whittaker, J. (Eds.), *Computational Statistics*, (pp. 411–428)., Heidelberg. Physica-Verlag HD.

Datar, M., Gionis, A., Indyk, P., & Motwani, R. (2002). Maintaining stream statistics over sliding windows: (extended abstract). In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, (pp. 635–644)., Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Donoho, D. L. & Huber, P. J. (1983). The notion of breakdown point. In *A Festschrift for Erich Lehmann (P.J. Bickel, K. Doksum and J.L. Hodges, Jr., Eds.)*, (pp. 157–184). Wadsworth, Belmont, CA.

Floyd, R. W. & Rivest, R. L. (1975). Expected time bounds for selection. *Commun. ACM*, *18*(3), 165–172.

Grubbs, F. E. (1969). Procedures for detecting outlying observations in samples. *Technometrics*, *11*(1), 1–21.

Hampel, F. R. (1974). The influence curve and its role in robust estimation. *Journal of the American Statistical Association*, *69*(346), 383–393.

Hoare, C. A. R. (1961). Algorithm 65: find. *Commun. ACM*, *4*(7), 321–322.

Hodge, V. & Austin, J. (2004). A survey of outlier detection methodologies. *Artif. Intell. Rev.*, *22*(2), 85–126.

III, J. P. (1975). Statistical inference using extreme order statistics. *Ann. Statist.*, *3*(1), 119–131.

Johnson, D. & Mizoguchi, T. (1978). Selecting the k-th element in $x + y$ and $x\_1 + x\_2 + \cdots + x\_n$. *SIAM Journal on Computing*, *7*(2), 147–153.

Mirzaian, A. & Arjomandi, E. (1985). Selection in x + y and matrices with sorted rows and columns. *Information Processing Letters*, *20*(1), 13 – 17.

Mosteller, F. & Tukey, J. (1977). *Data Analysis and Regression: A Second Course in Statistics*. Addison-Wesley series in behavioral science. Addison-Wesley Publishing Company.

Muthukrishnan, S. (2005). Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science*, *1*(2), 117–236.

Nunkesser, R., Schettlinger, K., & Fried, R. (2008). Applying the qn estimator online. In Preisach, C., Burkhardt, H., Schmidt-Thieme, L., & Decker, R. (Eds.), *Data Analysis, Machine Learning and Applications*, (pp. 277–284)., Berlin, Heidelberg. Springer Berlin Heidelberg.

Rousseeuw, P. J. & Croux, C. (1992). Explicit scale estimators with high breakdown point. *L1-Statistical analysis and related methods*, *1*, 77–92.

Rousseeuw, P. J. & Croux, C. (1993). Alternatives to the median absolute deviation. *Journal of the American Statistical Association*, *88*(424), 1273–1283.

Rousseeuw, P. J. & Hubert, M. (2011). Robust statistics for outlier detection. *WIREs Data Mining and Knowledge Discovery*, *1*(1), 73–79.

Tukey, J. W. (1977). *Exploratory Data Analysis*, volume 2. Addison-Wesley Publishing Company, Reading, Massachusetts.