Data Min Knowl Disc
DOI 10.1007/s10618-016-0472-z

Irrevocable-choice algorithms for sampling from a stream

Yan Zhu[1] · **Eamonn Keogh**[1]

Received: 28 February 2016 / Accepted: 14 June 2016

Abstract The problem of sampling from data streams has attracted significant interest in the last decade. Whichever sampling criteria is considered (*uniform* sample, *maximally diverse* sample, etc.), the challenges stem from the relatively small amount of memory available in the face of unbounded streams. In this work we consider an interesting extension of this problem, the framework of which is stimulated by recent improvements in sensing technologies and robotics. In some situations it is not only possible to *digitally* sense some aspects of the world, but to *physically* capture a tangible aspect of that world. Currently deployed examples include devices that can capture water/air samples, and devices that capture individual insects or fish. Such devices create an interesting twist on the stream sampling problem, because in most cases, the decision to take a physical sample is *irrevocable*. In this work we show how to generalize diversification sampling strategies to the irrevocable-choice setting, demonstrating our ideas on several real world domains.

Keywords Irrevocable · Sampling · Data stream · Diversification

1 Introduction

In the last decade there has been an explosion of interest in sampling, diversification, clustering, etc. on data streams. The streaming (or incremental) versions of such prob-

Responsible editor: Thomas Gärtner, Mirco Nanni, Andrea Passerini and Celine Robardet.

✉ Yan Zhu
yzhu015@ucr.edu

Eamonn Keogh
eamonn@cs.ucr.edu

1 University of California, Riverside, Riverside, USA

lems differ from their batch equivalents in that they assume that they have a finite (and generally *small*) memory, and that they will see objects arriving one at a time. As each object arrives, the algorithms can exactly or approximately incorporate the object's influence into variables that maintain statistics about the stream (means, quartiles, cardinality, etc.) and they can choose to exactly record the arriving value in one of k memory locations.

Recording the value of an object is a generally non-trivial decision for the algorithm to make. Since the algorithm only has k memory slots, each time it records an object's value there is an opportunity cost: there are other objects whose value the algorithm did not capture. The difficulty of this decision is mitigated by the fact that these writes to the k memory slots are classic *destructive writes*; the algorithms can overwrite any slot as many times as it likes, if it believes that a recently arrived object is more worthy of saving than a currently cached item.

The capability of destructive writes means that in many cases, the streaming algorithm can have a similar or identical performance to a random-access batch algorithm. For example, if the task is to record the k largest values in a stream, then the k slots simply act as k *best-so-far* variables and the streaming and batch algorithms have identical results. A less trivial example is the classic reservoir-sampling algorithm that can obtain a completely random sample of size k from a stream of length N, where N is either a very large or an unknown number (Vitter 1985). However, it is interesting to consider the case in which we have *one-time* writes, not destructive writes. In such cases, if the algorithm decides to record a value, then one of the k slots is irrevocably lost, and can never be used again. We call such algorithms *irrevocable-choice* algorithms.

While there are several types of *memory* hardware that permit Write Once Read Many (WORM) access, from EPROMs to punch cards, here we are interested in situations in which the k slots are physical devices that capture some *tangible* element of the world being sampled. Since this is a novel problem setting, we take the time to motivate it with three real world examples.

- As shown in Fig. 1 the ***Bluefin Autonomous Underwater Vehicle*** is an autonomous submarine robot platform that is used in undersea exploration in domains as varied as biodiversity studies, oil exploration and deep-submergence archaeology (Roman and Mather 2010). The standard robot is equipped with a host of sensors and the ability to make 22 irrevocable decisions to capture water samples for later analyses (Goldberg 2011). It is important to note that the *irrevocability* here is not due to a software setting or an arbitrary design choice. The physics of water pressure (441 PSI at a depth of 1000 feet) mean that once you open a sample tube, there is no

Fig. 1 The Bluefin 12S can make 22 irrevocable decisions to capture water samples for later analyses

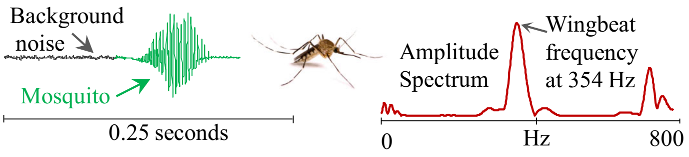


Fig. 2 (*left*) An audio clip of a mosquito flight sound can be converted into an amplitude spectrum; (*right*) this representation allows classification and clustering

practical way to empty it.
Capturing one sample per hour during a daylong mission is an obvious possibility; however, this may result in missed opportunities. Suppose that the onboard sensors detect a sudden dramatic change in temperature and salinity minutes after the 1:00 p.m. sample was taken. If we wait until 2:00 p.m. to capture the next sample, we may miss the opportunity to later investigate the interesting phenomenon (natural plume, chemical spill, etc.).

- In recent years, researchers have begun to deploy ***insect sensors*** into the field (Chen et al. 2014). As shown in Fig. 2, these sensors can optically record the flight "sound" signal of an insect.
In several ongoing projects, researchers have built devices that can *selectively* capture some of the sensed insects (Project Premonition 2015a). In some cases the device can capture insects alive and unharmed, but, as hinted at in Fig. 3, in other cases the device kills or incapacitates the insect. Note that even in the case in which the mosquito is shot down, the choice is irrevocable and expensive in terms of resources (draining power from a capacitor).
- A fish ladder is a structure on the side of large man-made dams to facilitate diadromous fishes' natural migration. These ladders enable fish to pass around the otherwise insurmountable dam by swimming and leaping up a series of relatively low steps. More than a thousand such fish ladders worldwide are augmented with video and other sensors (Zhang et al. 2015) as hinted at in Fig. 4. Thus far, fish ladders are passive devices, but researchers are experimenting with robots that can selectively remove fish for later analysis (Jonsson 2015; Zhang et al. 2015).
In addition to these examples there are many other instances of *irrevocable-choice* capture devices, including automatic water sampling devices which are deployed in the millions worldwide (Honda and Watanabe 2007), a single chemistry lab currently on Mars which can make at most twelve *irrevocable-choice* samples each day (Webster and Agle 2012), and *irrevocable-choice* blood capture devices (Fønss and Munksgaard 2008).

2 Related work

Our project combined ideas from two mature yet still very active research areas, *optimal stopping* algorithms and *streaming* algorithms. The former area is normally considered a branch of game theory or optimization, and the latter a sub-area of databases/data mining.

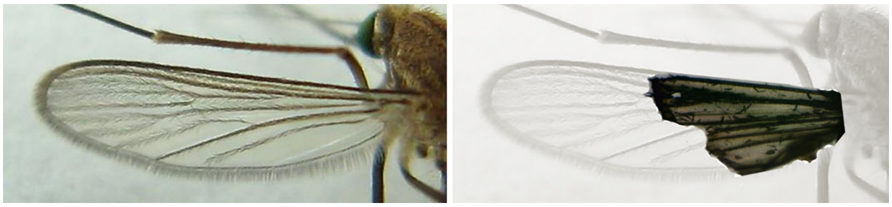
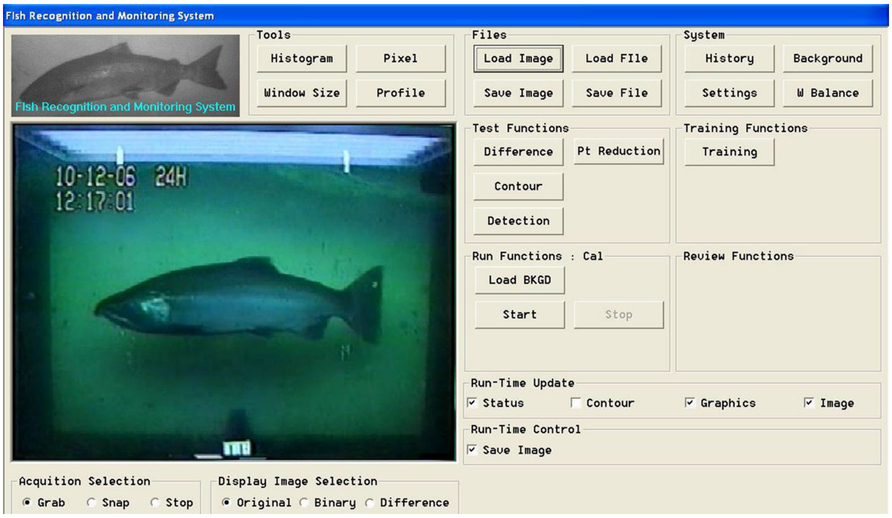


Fig. 3 (*left*) A live mosquito. The mosquito was shot down in free flight by a short laser burst that burnt off 65 % of its wing. (*right*) The decision to shoot an insect is clearly irrevocable



Run-Time Update: [✓ Status] Contour [✓ Graphics] [✓ Image]

Run-Time Control: [✓ Save Image]

Acquition Selection: [✓ Grab] Snap Stop

Display Image Selection: [✓ Original] Binary Difference

Fig. 4 A user interface showing a live video stream from a monitoring station in a fish ladder. Image courtesy of Dah-Jye Lee of Brigham Young University

2.1 Optimal stopping algorithm

Optimal stopping algorithms are concerned with the problem of choosing a time to take a given action based on sequentially observed random variables in order to maximize an expected payoff (Hill 2009). Perhaps the most famous example is the secretary problem (also known as the marriage problem, dowry problem, the fussy suitor problem, or the googol game).

The problem can be stated as following: imagine an employer wants to hire the best secretary out of N rankable candidates for a position. The candidates are interviewed one by one in random order. A decision about each particular candidate is to be made *immediately* after the interview. Once rejected, a candidate cannot be recalled. During the interview, the administrator can rank the candidate among all candidates interviewed so far, but is unaware of the quality of the candidates he has yet to see. The objective is to maximize the probability of selecting the best candidate. If it is possible to defer decision until after the last candidate is seen, then simply maintaining a *best-so-far* pointer gives a trivial solution. The difficulty is that the decision must be made immediately, and the decision is irrevocable.

The problem has an unexpectedly simple and elegant solution. The employer should maintain a *best-so-far* value for the first N/e candidates (where e is Euler's number $\sim$2.718), and thereafter pick the first candidate with a higher ranking. This stopping algorithm gives a $1/e$ (about 37 %) chance of selecting the best candidate.

The secretary problem has been generalized in dozens of ways by adding constraints, changing the metric of success, increasing the possible actions that can be taken, and adding uncertainly to the values/rankings; see Ferguson (2006) for an overview. While the secretary problem may seem like just a contrived puzzle, it has seen concrete applications in auctions and portfolio management, where decisions (to bid/buy) are clearly irrevocable. Likewise, one can model house selling as an optimal stopping problem. Assume that you have a house and wish to sell it. Each day you need to pay k dollars to keep advertising the house, and on the ith day you are offered x_i dollar for it. If you sell your house on the nth day, you will earn y_n dollars, where $y_n = x_n - nk$. The problem is when you should sell the house to maximize y_n.

Most optimal stopping problems can be written in the form of a Bellman equation, and are thus amiable to solution by dynamic programming (Peskir and Shiryaev 2006). When the underlying process is determined Markovian transition probabilities, very powerful analytical tools provided by the theory of Markov processes can often be utilized (Peskir and Shiryaev 2006).

More general optimal stopping algorithms have seen applications in biological domains similar to our motivating examples. For example, as far back as 1979 Rasmussen and Starr asked "*when should you stop searching for a new species?*" (Rasmussen and Starr 1979).

2.2 Streaming algorithms

In the early years of database/data mining research it was largely assumed that the data of interest was *static*. However, in the last decade, our increasing ability to *dynamically* monitor data sources has given rise to an ever-growing interest in streaming algorithms (Vitter 1985; Cormode and Hadjieleftheriou 2010). Thus, virtually all useful data mining batch algorithms have been generalized to the streaming case, including classification, clustering, segmentation, repeated pattern discovery (Begum and Keogh 2014) and diversification (Drosou and Pitoura 2012a, b). The "cost" of moving from the batch case to the streaming case is highly variable; in some cases, such as certain kinds of sampling (Vitter 1985) or time series segmentation, there is no overhead for handling the streaming case. In fact, for these problems, practitioners may use the streaming algorithms even if all the data is available with random access. However, for many problem definitions, moving to the streaming case comes with the loss of performance guarantees, and we must resort to offering approximate solutions (Begum and Keogh 2014).

The problem of selecting the k most diverse items from a set of real-valued data objects is an interesting one. It is known that even the batch case is NP-complete; therefore, we must resort to approximate solutions. However, streaming algorithms have been designed that can produce essentially identical results, and offer similarly tight bounds on performance (Drosou and Pitoura 2012a).

This review of streaming algorithms was necessity brief; we refer the interested reader to Aggarwal (2006) and the references therein.

3 Definitions and assumptions

3.1 Diversification problems

Among all the variants of sampling from data streams, *result diversification* has recently attracted significant attention as a technique to increase user satisfaction in web/data base search, recommender systems, etc. (Drosou and Pitoura 2012a, b; Ghosh 1996; Minack et al. 2011). Diversification is also an important tool in scientific discovery, in the sense that we often want to capture the most representative samples in the world we are exploring. Note that in most information retrieval uses of diversification, only the *answer set*, a relatively small subset of the entire data collection, must be diversified. In contrast, in the applications we consider, we need to consider diversification with respect to the entire data collection, as there is no well-defined answer set. In this section, we first define two classical diversification problems, then extend the notation to the irrevocable-choice diversification problem we wish to solve.

3.1.1 Batch k-diversification problem

Let $U = u_1, u_2, \ldots, u_N$ be an ordered set of N items. For every possible pair of items u_i, u_j $(i \leq N, j \leq N)$, we define a distance measure $dist(u_i, u_j)$: $U \times U \rightarrow \mathbb{R}^+$. To prevent *pathological* solutions to our problem, we require $dist(u_i, u_j)$ to be symmetric, i.e. $dist(u_i, u_j) = dist(u_j, u_i)$. Euclidean distance is clearly one such eligible distance measure.

To define the k-diversification problem, we must first clarify the meaning of k. We use k as the user's choice/constraint for the number of items in the answer set. For example, in the diversification of web search results, k is often set to ten, the default number of results returned on the first page of the desktop version of most familiar search engine interfaces. In our case, k is the number of capture choices we can make given hardware constraints. The batch k-diversification problem is to find a subset $S \subseteq U : |S| = k$, such that the diversity of S is maximized. The diversity of a set S is measured by a function *divs*: $S \times dist \rightarrow \mathbb{R}^+$. There are two common variations of the *divs* function (Drosou and Pitoura 2012b; Ghosh 1996). One is called the SUM diversity, which is defined as:

$$divs_{SUM}(S, dist) = \sum_{s_i, s_j \in S, i \neq j} dist(s_i, s_j),$$

the other is called the MIN diversity, which is defined as:

$$divs_{MIN}(S, dist) = \min_{s_i, s_j \in S, i \neq j} dist(s_i, s_j).$$

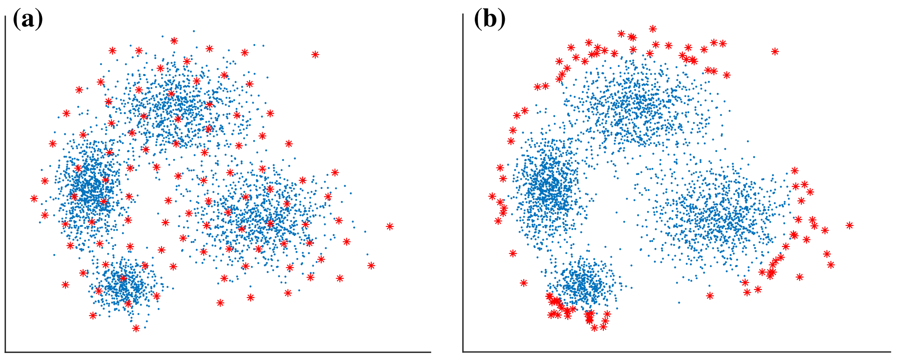


Fig. 5 MAXMIN (**a**) versus MAXSUM (**b**) solutions for $N = 3500$ and $k = 100$. Diverse items are shown in *red*/*bold*. Inspired by a similar figure in Drosou and Pitoura (2012b) (Color figure online)

Corresponding to these two different diversity definitions, the two most common variants of the diversification problem are referred to as the MAXSUM diversity problem and the MAXMIN diversity problem, respectively. In general, it has been noted that the MAXMIN diversity "*tends to select items that intuitively provide a better cover of the set*" (Drosou and Pitoura 2012b). To visually confirm and illustrate this, in Fig. 5 we show the results of both strategies on a small synthetic dataset consisting of four different sized Gaussian "balls" in two-dimensional space.

We can see that items selected by MAXMIN diversity are evenly spread among the set, while those selected by MAXSUM diversity are located mainly at the borders. Therefore, we use MAXMIN diversity in this paper, and define the batch k-diversification problem formally as finding such a subset $S^*$of U:

$$S^* = \underset{S \subseteq U, |S|=k}{\operatorname{argmax}} \; divs_{MIN}\,(S, dist)\,.$$

The batch k-diversification problem has been proved to be NP-hard (Erkut 1990; Ghosh 1996); however, demonstrably good approximation algorithms are known (Erkut et al. 1994). Beyond the intractable time requirements of the problem, most algorithms assume that when attempting to optimize subset S^*, we could store the entire set of U in memory and have random access to it (Erkut 1990; Ghosh 1996).

3.1.2 Streaming k-diversification problem

An interesting variant k-diversification problem is to consider it in a *streaming* setting (Cormode and Hadjieleftheriou 2010; Vitter 1985). The streaming k-diversification problem makes the random-access requirement of the batch k-diversification problem moot, as we do not have full knowledge of set U in advance. In addition, potential algorithms cannot be expected to keep an unbounded number of previously seen objects in memory, and must resort to keeping a small constant-sized buffer. Concretely, the problem setting is as follows. The N items of set U arrive one by one. We are provided k memory slots, and when an item arrives, we need to decide immediately whether to discard it or place it in memory. A discarded item cannot be retrieved later, but items in memory can be overwritten. In Minack et al. (2011) the author gives an efficient algorithm to solve this problem with a diversification quality very close to the batch algorithm.

3.1.3 Irrevocable-choice k-diversification problem

In contrast to the above instantiations of the k-diversification problem, which deals only with *digital* memory, we are interested in the situation where the memory is an *analogue* capture device which physically captures some tangible aspects of the world: an insect, a water sample, a rock sample, etc. In such cases, we can only fill each *physical* memory slot once. We call this situation the *irrevocable-choice* k-diversification problem.

More formally, we have N items from set U arriving one by one, and we wish to fill the k memory slots. As an item arrives, we can either drop it or *physically* place it in memory. A *dropped item* cannot be retrieved later. In contrast to the classic streaming k-diversification problem, the items stored in memory slots cannot be replaced. The goal of the problem is to find a subset $S^{'} \subseteq U$: $|S^{'}| = k$, such that $divs_{MIN}(S', dist)$ is as close to $divs_{MIN}$ $(S^{*}, dist)$, the solution of the batch k-diversification problem, as possible. Of course, this *digital* goal is really a proxy for our true goal of capturing diverse *physical* objects.

3.2 Assumptions

To clarify the setting of the irrevocable-choice k-diversification problem, we begin by listing the following assumptions.

Assumption 1 The size of set U (denoted as N) is known in advance.

This assumption will generally *not* be true in real world setting. However, it greatly simplifies the notations and algorithms that follow. Later we will show that as long as a reasonable and conservative (i.e. *lower bounding*) estimate of N is provided, our algorithm is robust enough against violations of this assumption.

Assumption 2 All items in U are independent and identically distributed (i.i.d.), i.e., they have the same probability distribution and are mutually independent.

Note that though all the items come from the same probability distribution, we do not know what that distribution is in advance, nor do we need to explicitly model it.

Assumption 3 N is much larger than k, and is large enough for U to approximate the probability distribution it comes from.

This assumption matches the scenarios we described in Sect. 1. For example, the first mosquito capture device built for Microsoft's Project Premonition (2015a) is able to capture 64 insects ($k = 64$) but may see more than ten thousand insects in a single night ($N = 10{,}000+$) (Silver 2008).

Assumption 4 We are given *digital* memory large enough to store all the data.

Note that this *digital* memory is different from the *physical* “memory slot” we discussed in Sect. 3.1.3. We need to select k irrevocable-choices to fill the “memory slots”. If we decide to drop a data point, we cannot retrieve it later to fill in those memory slots. But we are allowed the memory to “remember” all the discarded *digital* data to gain some insights into the entire data set (seen thus far). To better differentiate between these two usages of “memory” in the rest of the paper, where there is a possibility of ambiguity we refer the space we store digital data as the *D-Memory* (*Digital Memory*) and the space we store physical objects as the *P-Memory* (*Physical Memory*).

Assumption 5 For all items u_i, u_j, $dist(u_i, u_j)$ can be evaluated.

Here we are simply assuming that we have an appropriate distance measure defined by the digital signature of the physical objects we wish to capture. This measure will be data-dependent. For example, for flying insects, the Euclidean distance between frequency amplitude spectrums (cf. Fig. 2) has been shown to be effective (Chen et al. 2014); for water samples the absolute difference in turbidity may be appropriate; and for rock samples (both Terrestrial and Martian, Webster and Agle 2012) the *Spectral Information Divergence* or *Spectral Angle* of the spectral signatures may be appropriate (Cerra et al. 2011).

Having defined our problem and stated our assumptions, we are now in a position to introduce our solution. For clarity of presentation, we begin by introducing a “static” algorithm, which learns a key parameter by examining the distribution of the first items it encounters, and “hardcodes” this parameter until the program terminates. As we shall show, this simple algorithm often works reasonably well, but can have occasional dramatic failures with an unlucky permutation of the arrival order. To solve this problem, in Sect. 5 we generalize our algorithm to again learn the key parameter, but to *dynamically* adapt it over time as needed.

4 The static algorithm: *Simplek*

To explain our solution, we begin with an assumption that is clearly untrue, and later show that we can nevertheless approximate it. Concretely, we assume $MD = divs_{MIN}(S^*, dist)$ is known *in advance*. Given this, in order to obtain the optimal set S^*, the distance between any two data objects we capture must be at least MD. Based on this observation, we can develop a naïve solution (Strategy 1) to the problem. Suppose our irrevocable selection set is S, which is empty at the beginning. Let $T = MD$.

Strategy 1 (***NaiveStrategy***) Place the first data point u_1 in set S. Then, for any upcoming data u_i, we calculate $dist(u_i, S)$, the minimum distance between u_i and all objects in S. If $dist(u_i, S)$ is greater than or equal to T, place u_i in S. Otherwise discard u_i.

Consider the role that T plays in the above strategy. Two unwelcome possibilities are:

- T is set too small. In this case, we will quickly fill *P-Memory* (set S) with objects that are not particularly diverse.
- T is set too large. In this case, we will rarely or never place objects into *P-Memory* (set S).

Thus we must consider the "Goldilocks principle" and carefully set the value of T. Even so, this strategy can still fail. This is because the strategy is sensitive to the u_1 captured in the first move as well. For example, suppose we have $U = 2, 1, 3, 4, 5$ and $k = 3$, then $S^* = 1, 3, 5$, $MD = 2$. But since $u_1 = 2$, the items we capture are actually $S = 2, 4$; we terminate with $|S| < k$. We can prevent this with a simple policy (later formalized in lines 11–12 of Table 2). If there are E empty slots remaining in S, and only E objects remaining to be seen from the stream, then we simply capture every remaining object, given that accepting suboptimal objects is nevertheless better than having unfilled slots. Note that even if we got lucky and $u_1 \in S^*$, the strategy may *still* fail, simply because we might have captured a data point u_i, such that $dist(u_i, S) \geq MD$ but $u_i \notin S^*$ (e.g. $U = 1, 4, 2, 3, 5$, then $S^* = 1, 3, 5$, $MD = 2$, but we will actually capture $S = 1, 4$, and $4 \notin S^*$). In that case, $|S|$ will again be smaller than k, unless we resort to the "*fill the last E slots*" fix suggested above. These issues could possibly be mitigated if we set T to be *slightly* less than MD. However, this requires us to address our assumption that we know MD in the first place. The following observation explains our rationale.

Assume we divide the dataset U into two parts: $U_1 = u_1, u_2, \ldots, u_m$ and $U_2 = u_{m+1}, u_{m+2}, \ldots, u_N$. Suppose S_1^* is the solution of the batch k-diversification problem of set U_1, S_2^* is the solution of set U_2, and the MIN diversity values of S_1^* and S_2^* are MD_1 and MD_2, respectively. Since all the items in U are i.i.d., given $|U_1|$ and $|U_2|$ are much larger than k, we have $MD_1 \approx MD_2 \approx MD$.

Since this assumption is a cornerstone of our proposed algorithm, and it is somewhat surprising, we will take the time to justify it. Recall the dataset shown in Fig. 5a. The MIN diversity MD for the entire 3500 data objects is 1.37. If we randomly divide the dataset into two equal sized halves, we find $MD_1 = 1.25$ and $MD_2 = 1.28$. Thus in this case we do have $MD_1 \approx MD_2 \approx MD$.

This observation suggests a two-phase algorithm:

Step 1: Learning Phase. Allow items in U_1 to pass by, observe their values and learn an appropriate threshold T from them.

Step 2: Capture Phase. Apply T to Strategy 1 to make the k irrevocable choices in U_2.

Our proposed algorithm to learn the appropriate threshold T is outlined in Table 1. For notational convenience, in the peusdocode we use "*Data*" instead of "U".

The *FindThreshold* algorithm calculates T, the threshold we must set in Strategy 1 to capture at least k items in *Data*. Line 1 finds the extreme values in the search space of T. Then in lines 2–9 we use binary search to find T.

Table 2 explains the algorithm that exploits the learned threshold to trigger the irrevocable captures. In the *Simplek* algorithm in Table 2, lines 1 and 2 delineate the dataset into two sections. We drop the first m data *physically*, but keep their associated *digital* values in *D-Memory*. Line 3 uses the *FindThreshold* algorithm in Table 1 to calculate the threshold T. Then we apply T and Strategy 1 to the rest of the data, such that the distance between any two items in set S is no less than T. The process terminates when $|S| = k$.

Note that no matter how accurate our estimate of T is, there is some danger that we will fail to capture enough physical objects, and that as we approach the end of

Table 1 Algorithm to learn the threshold value

FindThreshold(*Data*, *k*)

```
FindThreshold(Data,k)
Input: Data: dataset; k: number of items to select
Output: T: Threshold
1   Tmin←min(Distances(Data)), Tmax←max(Distances(Data))
2   l←Tmin,h←Tmax,Sl←NaiveStrategy(Data,l),Sh←NaiveStrategy(Data,h)
3   T←$(l + h)/2$, S←NaiveStrategy(Data,T)
4   while $|S| \neq k$ and $|Sl| \neq |Sh|$ do
5       if $|S| < k$ then h←T, Sh←S
6       else l←T, Sl←S
7       end if
8       T←$(l + h)/2$, S← NaiveStrategy(Data,T)
9   end while
10  return T
```

Table 2 A simple algorithm to select k diverse items

Simplek(*Data*, *k*)

```
Simplek(Data,k)
Input: Data: dataset, data coming in one by one; k: number of items to select
Output: S: Set of the k selected items
1   N←|Data|
2   m←⌊N/2⌋, Drop← Data[1:m]  //keep digital signature in D-Memory
3   T←FindThreshold(Drop, k)    // i.e. Table 1, learning phase
4   S←{Data[m+1]}
5   for i←m+2 to N do
6       if Dist(Data[i], S)≥T then
7           S← S ∪{Data[i]}
8           if |S|== k then
9               return S
10          end if
11          if k-|S|== N-i then          // Failure Case (Definition 1)
12              S← S ∪{Data[i+1:N]}  //  We were too aggressive..
13              return S                 // ..so we must take all..
14          end if                       // ..the remaining objects.
15      end if
16  end for
17  return S
```

the data stream we realize that $|S|$ will be smaller than k. To avoid this situation, lines 11–12 forces the process to place the last several data into S when it reaches the end. In this case, the MIN diversity of set S is no longer bounded by T. It can be any random value. For clarity, we define this situation as a *failure*.

Definition 1 *Failure*: If we have just E remaining data items left to process, and we have captured only $k - E$ objects, we switch to a suboptimal "*just take all of the next E items*" subroutine, and denote this situation as a *failure*.

Note that this is a *failure* in the sense that we did not make optimal choices, but we did *not* fail to capture k items. We clearly wish to reduce the probability of such failures as much as possible.

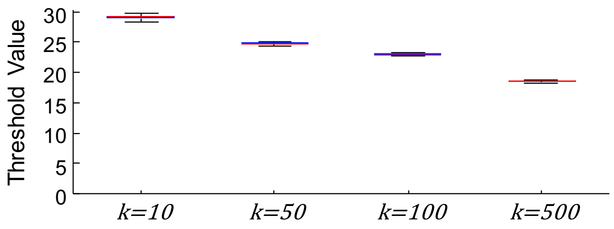


Fig. 6 The effect of the first value on the selection result. The tops of the box-and-whisker plots correspond to the *maximum* value obtained in the 100 runs, and the bottoms correspond to the *minimum* of the 100 runs

4.1 Discussion of *Simplek* algorithm on our running example

To help the reader appreciate the characteristics of the *Simplek* algorithm (and the challenges of the problem setting itself), we begin by considering a concrete problem instantiation as a running example. Suppose we will encounter 5000 random-walk time series of length 512, and need to select k diverse items from this set. We will use z-normalized random walks here for convenience, but recall that our algorithm does not know that (otherwise exploitable) fact.

4.1.1 The impact of the first value

In line 4 of Table 2, we always select the *first* item in the data stream as we reach the capture phase, is this a good strategy? It seems to open our algorithm to the possibility of occasional poor luck, where the first item affects our irrevocable-choice results significantly. Should we spend more effort in creating a better strategy for selecting a good first value?

To answer this question, we randomly shuffled the 5000 random-walk samples 100 times, and then applied the *FindThreshold* algorithm in Table 1 to see how different the resulting threshold T would be. Figure 6 shows the box-and-whisker plot of T corresponding to four different k values. The red lines represent the medians, the upper and lower bounds of the blue box represent quantiles, and the black lines stand for extreme values.

As expected, we see that as k increases, the threshold T decreases, which means the minimum distance between any two items of the selection set S in Table 1 also decreases. Figure 6 also shows that the variance of T decreases dramatically as k grows. Note that the variance of T is already very low when k is as small as 10. These results strongly suggest that as long as k is reasonably large, the arbitrariness of the first value selection will not affect the selection result significantly.

4.1.2 The setting of m

The setting of the value m offers a classic tradeoff. Given a finite pool of data, should we use more data to learn, at the expense of candidate data we could consider capturing, or should we use less data to learn, but have more candidate items to capture? In Table 2

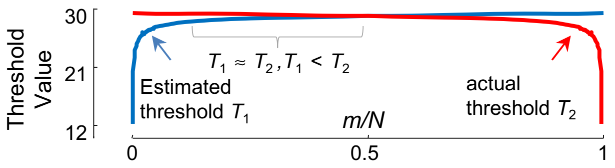


Fig. 7 The effect of m/N on the threshold learned. Note that for a large range of [0.2, 0.5], $T_1 \approx T_2$, and $T_1 < T_2$

line 2, we had simply set m as $\lfloor N/2 \rfloor$ without discussion. We repair that omission here.

If m is larger, we will use more data in the first part U_1 to learn an appropriate threshold T. As a result, less data are left in the second part U_2 for us to choose from. In contrast, if m is smaller, we will learn T from a smaller U_1, but will have more data in U_2 to choose from. Figure 7 shows how the learned threshold T would change as m/N increases when we need to choose $k = 10$ samples from the 5000 random walks. The result is averaged over 100 random shuffles of the data.

The curves in Fig. 7 are obtained by applying *FindThreshold* algorithm in Table 1 to both parts of the data. The blue curve shows the estimated threshold we learned from the first part of the data. We denote it T_1. The red curve is the *actual* threshold we should set to get at least 10 samples from the second part of the data. We denote this T_2.

When m is very small, say $m < N/5$, the value of T_1 is much less than T_2. In that case, we will obtain a very pessimistic estimate of the threshold to use for the remaining data. In contrast, when m is large, say $m > N/2$, the value of T_1 is larger than the value of T_2. This is very undesirable since it will probably lead to *failure* (cf. Definition 1).

Note that when $m = \lfloor N/2 \rfloor$, T_1 is the closest estimate of T_2. This fact can be derived without experiment. However, the figure also shows us that for a relatively large range of m $\left(\frac{m}{N} \in [0.2, 0.5]\right)$, $T_1 \approx T_2$ and $T_1 < T_2$. This is good news for us, telling us that our algorithm is not very sensitive to the parameter choice.

Finally, note that Fig. 7 shows that the appropriate threshold for $N/2$ of the data (the value of the blue curve at $m/N = 0.5$) is very close to that for the entire data set (the value of the blue curve at $m/N = 1$). This means that, in principle, we can be very competitive with an oracle algorithm that is told the optimal threshold in advance.

4.2 *Simplek* algorithm: empirical result on our running example

To evaluate the *Simplek* algorithm, we will compare the results of three algorithms:

- **Batch**: The batch k-diversification assumption outlined in Sect. 3.1.1. Note that this algorithm is able to "cheat" relative to our problem setting, by making an arbitrary number of passes over all the objects. This algorithm provides an upper bound to the performance of our algorithm, which can make only a *single* pass over the data.

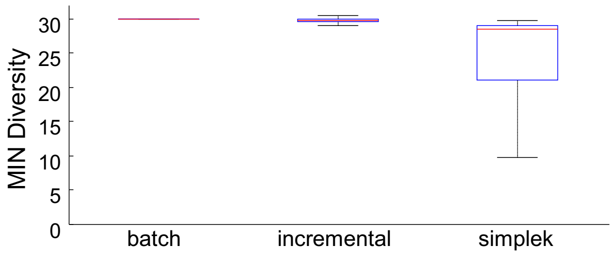


Fig. 8 A comparison of three algorithms on an irrevocable-choice k-diversification problem

- **Incremental**: The streaming k-diversification algorithm outlined in Sect. 3.1.2. Once again, this algorithm is cheating relative to our problem setting, because it is allowed to make choices that are not irrevocable. This algorithm provides a more realistic upper bound for the irrevocable streaming case in which we are interested.
- ***Simplek***: The irrevocable-choice k-diversification algorithm outlined in this section.

Note that since the batch k-diversification problem is NP-hard, we use the heuristic algorithm in Erkut et al. (1994) to calculate the approximate solution instead of the theoretical optimal one. For the streaming k-diversification assumption, we use the algorithm in Minack et al. (2011), which is the state of the art.

The task-at-hand is in making $k = 10$ diverse selections from the 5000 z-normalized random-walk samples. We compared the three algorithms by conducting 100 independent experiments. For each experiment we randomly shuffled the arriving order of the 5000 random-walk samples, and calculated the MIN diversity of the selected set corresponding to all three versions of k-diversification setting. Figure 8 summarizes the results with box-and-whisker plots.

4.2.1 Analysis of the Simplek algorithm result

The first thing we note is that the incremental algorithm's performance is very close to that of the heuristic batch algorithm (confirming the findings in Minack et al. 2011). This is somewhat surprising given the constraints under which the incremental algorithm runs.

The results for the *Simplek* algorithm are more mixed. The *average* performance is significantly worse, but considering only the average hides the great variability in performance. The box-and-whisker plot shows that about half of the *Simplek* results are close to those of the other two algorithms. However, more than 25 % of the results are much lower, and in extreme cases can be as low as 10, which is very undesirable.

With further investigation, we find this is a result of the too-aggressive "failure" case (cf. Definition 1). In 35 out of the 100 experiments, we are unable to capture enough items before reaching the end of the data, and forced to select the last several items. The MIN diversity obtained in those experiments is highly susceptible to poor luck.

More generally, the appropriate threshold T learned from the *FindThreshold* process in Table 1 not only depends on the data distribution, but also on their arrival order. For

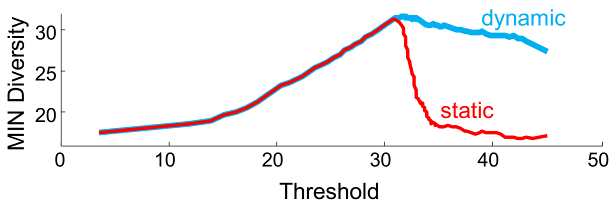


Fig. 9 Comparison of static threshold algorithm (*red*) and dynamic threshold algorithm (*blue*), as explained in Sect. 5 (Color figure online)

example, suppose $U_1 = 1, 2, 3, 4, 5$, $k = 3$ and $U_2 = 2, 4, 3, 1, 5$, a random shuffle of U_1. The appropriate threshold learned from U_1 is $T = 2$. But if we apply T to U_2, failure occurs. After capturing 2 and 4, there are no more items exceeding the threshold, and we are forced to accept 5, the last item.

To summarize the findings of this experiment, we have shown that the *Simplek* algorithm can often perform *very* close to the performance of greedy algorithm, but occasionally fails catastrophically. Moreover, these disastrous failures can be largely attributed to the threshold learning algorithm predicting a slightly too aggressive (too large) value for T. Therefore, we have developed two techniques that conservatively estimate T, one of which successfully reduced the failure rate to as low as 2 %. For brevity we are not discussing them here; we direct interested readers to Project Webpage (2016) for a detailed explanation.

4.2.2 *Limitations of the static threshold algorithm*

The algorithms discussed so far learn a *static* threshold from the first section of the stream, and then apply it to the remaining section. Though we have enhanced the basic *Simplek* algorithm with two effective strategies to find a good setting for the threshold (Project Webpage 2016), we cannot eliminate the possibility of failure cases, as we are at the mercy of an unlucky order of arrival. We can mitigate this sensitivity to order of arrival by setting the threshold even smaller; however, if the arriving order of the data is actually good, then this too-conservative threshold will decrease the MIN diversity we are able to obtain.

Thus there is a trade-off here: a too-aggressive threshold may result in failure cases, while a too-conservative threshold will limit the quality of our selection set. To better visualize this trade-off, we conducted the following experiment with the random-walk dataset. We exhaustively explored all possible threshold values. For each threshold value, we applied Strategy 1 to the latter half (latter 2500) of the random-walk dataset, captured five samples ($k = 5$) and evaluated the MIN diversity. Figure 9 shows the result averaged by 100 random shuffles of the data.

The red curve represents the result of the static threshold algorithm. We can clearly see a peak MIN diversity value when *Threshold* = 30. However, note that the MIN diversity value around the peak is very sensitive to the threshold: when threshold is smaller, MIN diversity decreases almost linearly; when threshold is larger, failure (Definition 1) occurs and the MIN diversity drops sharply. This is very undesirable, because the peak threshold value would change as the data (or just their arriving order) vary, and as we cannot see the data beforehand, a slightly suboptimal estimate of the threshold value can cause a large degradation in the MIN diversity.

Thus far our approach has assumed a single, static threshold. In the next section, we will show a *dynamic* threshold algorithm that greatly mitigates this problem. The dynamic threshold algorithm begins with an optimistic estimate of the threshold value and adjusts it (as needed) as we see additional data. The blue curve in Fig. 9 previews the result of the dynamic threshold algorithm with different initial threshold settings. The result is equal or superior to the static algorithm result for all threshold values. This is because the new algorithm is able to avoid failure by dynamically reducing the threshold value as we approach the end of a run and the algorithm realizes it is behind schedule in capturing items. To ground concrete numbers with this dataset, the range of values of the threshold for the static algorithm that give a MIN diversity that is within 1 % of optimal is 0.6171, but for the dynamic algorithm that range is 2.064. Thus we have essentially made the threshold estimation problem 3.35 times easier. The next section explains our dynamic threshold algorithm in detail.

5 A dynamic threshold algorithm

We propose the basic strategy of beginning with an optimistic (large) threshold, then *dynamically* adjusting it as we see more items. If we are capturing items too slowly, we lower the threshold, otherwise we keep the current value of the threshold.

A little introspection reveals that this *start-high-then-lower* option is the only possibility. A *start-low-then-raise* scheme results in the irrevocable selection of many items with small distances between each other before we have a chance to ratchet up the threshold.

Table 3 outlines our proposed dynamic algorithm to select k diverse items. Essentially, the dynamic algorithm measures the distances between all dropped items and the selected items, and summarizes the distribution in a histogram. The histogram is used to approximate the probability distribution of the distance between future items and selected items. The threshold is updated according to the probability distribution as more items arrive.

Similar to the *Simplek* algorithm, lines 1–3 evaluate an initial threshold T with the *FindThreshold* algorithm by learning from the first half of the data. Line 4 captures the first item of the second half into S. The variable *restk* is the remaining number of items to capture, while *restn* is the remaining number of items we expect to arrive.

In line 6, $Dist(Drop[i], S)$ is the minimum distance between $Drop[i]$ and every item in S. Lines 9–11 update the current threshold T. We use Fig. 10 to give a visual intuition of lines 9–11.

Each time after we have explored a range (defined by lines 12–18, explained below) of items, we update the histogram of the minimum distances between every dropped item and the selection set S, as Fig. 10 shows. This histogram is converted to a probability density function using simple kernel density estimation (Bowman and Azzalini 1997; Matlab 2016). Note that we need to capture at least *restk* items among the remaining *restn* items, and all these items must be at least T_d away from the current

Table 3 A dynamic algorithm to select k diverse items

Dynamic(*Data,k*)

```
Dynamic(Data,k)
Input: Data: dataset, data coming in one by one; k: number of items to select
Output: S: Set of the k selected items
1   $N \leftarrow |Data|$
2   $m \leftarrow \lfloor N/2 \rfloor$, $Drop \leftarrow Data[1{:}m]$ //keep digital signature in D-Memory
3   $T \leftarrow FindThreshold(Drop, k)$ //Table 1
4   $S \leftarrow \{Data[m+1]\}$, $startpt \leftarrow m+2$, $restk \leftarrow k$-1, $restn \leftarrow N$-$m$-1
5   for each $j$ in $Drop$
6       $DistToS[j] \leftarrow dist(Drop[j], S)$
7   end
8   while $restk \neq 0$ do
9     $hist \leftarrow histogram(DistToS)$, $pdf \leftarrow hist2pdf(hist)$
10    $T_d \leftarrow DynamicThreshold(pdf, restk/restn)$
11    $T \leftarrow \min(T, T_d)$
12    if $restk == restn$ then
13        $S \leftarrow S \cup \{Data[startpt{:}N]\}$, break   //Failure, Definition 1
14    elseif $restk > 1$ then
15        $endpt \leftarrow startpt + \lfloor restn/restk \rfloor$-1
16    else
17        $endpt \leftarrow startpt$
18    end if
19    $curend \leftarrow endpt$
20    for $i \leftarrow startpt$ to $endpt$ do
21       if $Dist(Data[i], S) \geq T$ then  //capture
22         $S \leftarrow S \cup \{Data[i]\}$, $restk \leftarrow restk$-1, $curend \leftarrow i$
23         for each $j$ in $Drop$
24             $DistToS[j] \leftarrow \min(dist(Drop[j], Data[i]), DistToS[j])$
25         end
26         break
27      else    //drop physically, but keep digital signature in D-Memory
28        $Drop \leftarrow Drop \cup \{Data[i]\}$, $DistToS[|Drop|] \leftarrow dist(Data[i], S)$
29       end if
30    end for
31    $startpt \leftarrow curend+1$, $restn \leftarrow N$-$curend$
32  end while
33  return $S$
```

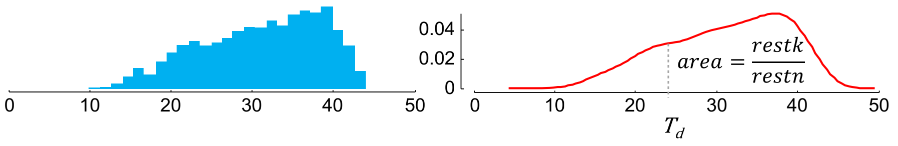


Fig. 10 (*left*) An example of the histogram of distances between all dropped items and the first selected item in the random-walk dataset. While this distribution is strongly suggestive of a reverse-lognormal, we need to make no such assumptions. (*right*) *Dynamic Threshold process* based on probability density function derived by the histogram

selection set S. Therefore T_d is determined such that the area below the pdf curve and to the right of T_d is $\frac{restk}{restn}$. Line 11 updates the current threshold T if $T_d < T$.

In lines 12–18, [*startpt, endpt*] define the range of items to explore with the current threshold T. When $restk > 1$ (lines 14–15), we explore the next $\lfloor restn/restk \rfloor$ items with the current threshold T unchanged. This is because if T is appropriate, there is a high probability we will capture at least one item within this range. If we capture nothing in this range, then the threshold needs to be adjusted. Note that when $restk = 1$ (lines 16–17), we have $\lfloor restn/restk \rfloor = restn$, but it is undesirable to explore all of the remaining items with a static threshold T. Thus we set the length of the range to be 1, allowing us to adjust threshold T after the arrival of every single item.

Lines 20–30 explore items in range [*startpt, endpt*] according to the current threshold T. We call this a static-threshold exploring session. If an item is captured before we reach *endpt*, we end the exploration process instantly, update *restk, restn* and set *startpt* as the next item to arrive. Otherwise, we set *startpt* as the item following *endpt* and update *restn*.

5.1 Complexity and scalability of the algorithm

As our goal is to capture physical items in a streaming setting, the algorithm needs to be fast and space-efficient. Therefore, before moving on to empirical evaluation, we analyze the complexity of the algorithm in Table 3. The space complexity of the algorithm is $O(Nd)$, where d is the length of the physical signature of an item. In the learning phase (line 2), we simply store the digital signature of the first m items in memory, so the processing time for each item is $O(d)$. After that the *FindThreshold* process in line 3 takes $O(N^2d)$ time to calculate an initial threshold value. In the capture phase (lines 4–30), the time used to process each item differs in three cases. If the item is dropped and we have not reached the end of a static-threshold exploring session, we simply spend $O(kd)$ time to store its digital signature in memory and update *DistToS* (line 28). If the item is dropped and we have reached the end of a static-threshold exploring session, we run through lines 9–11 to obtain a new threshold value with $O(N)$ time. If the item is captured, we need to update both *DistToS* (lines 23–25) and the threshold value (lines 9–11); in this case the time spent is $O(Nd)$.

In all our real-life applications in Sect. 6, the processing speed of each item is at least two orders of magnitude faster than its arrival rate. However, if N is too large, the data cannot easily be stored in *D-Memory*, and the *FindThreshold* process in line 3 becomes intractable. Consider an insect-capture experiment that lasts for a week. If the insects arrive two per second, then we must examine the digital signature of 1,209,600 insects. If the digital signature of each insect sample consists of 1000 data points with double precision, then the dynamic algorithm would require an infeasible 9GB of memory. Nevertheless we can easily handle this situation, because we do not need to keep the digital signature of all the data in *D-Memory* to obtain high-quality MIN diversity result. We will see later that Fig. 13 (incidentally) provides a strong evidence for the claim. Randomly sampling a small fraction of a massive dataset allows us to tightly estimate MIN diversity of the *entire* dataset.

Thus all we have to do to handle such cases is replace the data we store in *D-Memory* (cf. the variable *Drop* in Table 3) with a tractable random sampling of them. For brevity, we explain and test this idea in detail at Project Webpage (2016), showing that our ideas easily scale to millions of objects.

5.2 Result of the dynamic threshold algorithm on the running example

Figure 11 shows the result of the dynamic algorithm on our running example problem. This time we have no failure case, and in 3/4 of the runs, the MIN diversity obtained

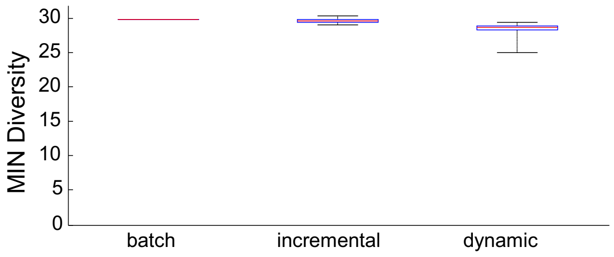


Fig. 11 The effectiveness of the dynamic threshold algorithm. Compare to Fig. 8

is within 5 % of the approximate batch MIN diversity. This is a stunningly small performance gap, given the very restrictive constraints of the irrevocable-choice problem setting.

While the results in Fig. 11 bode well for our proposed algorithm, this is just a single problem setting. In the next section, we will empirically evaluate the algorithm under more diverse conditions including three real world case studies.

6 Empirical evaluation

We begin by noting that all our experiments (including all the figures above) are completely reproducible. All experimental code and data (and additional experiments omitted for brevity) are archived in perpetuity at Project Webpage (2016). The streaming and dynamic natures of our algorithm lend themselves to visual intuition, and we have created short videos to illustrate this work (Project Webpage 2016).

Our proposed algorithm performed well on the running example toy-problem in the previous sections. Here we want to see how the algorithm performs for more diverse problems, and for problems in which our simplifying assumptions no longer hold.

6.1 The Effect of Varying k

As all our previous examples consider only a single value for k, the number of diverse items to capture, we will begin by testing performance of the algorithm by various values of k. We continue to consider our running example of 5000 z-normalized random-walk time series of length 512, and again randomly shuffle the dataset 100 times to allow averaging over 100 experiments. Figure 12 shows the resulting average MIN diversity of three algorithms: the approximate batch algorithm, the incremental algorithm and our proposed dynamic algorithm.

The sharp-eyed reader may notice that the average MIN diversity results of the batch algorithm is very slightly lower than the incremental algorithm when k is small. That is because the batch result is obtained by the approximate heuristic algorithm Erkut et al. (1994). The incremental algorithm Minack et al. (2011) can perform better by chance. Our proposed algorithm also performs better in some of the experiments. More importantly, no matter how k varies, the average MIN diversity obtained with

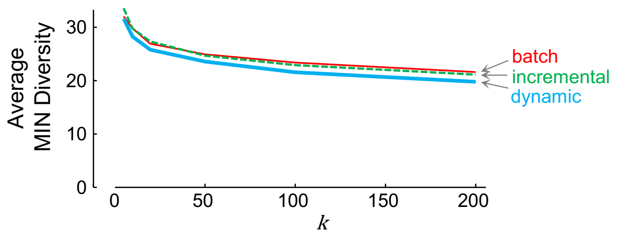


Fig. 12 The performance of three algorithms with various values of k. The difference in the performance of three algorithms is difficult to tell, which is the *point* of this plot

our proposed algorithm (dynamic) is always more than 92 % of the approximate batch MIN diversity.

6.2 The Effect of the *N is Known* Assumption

We envision that in real-life deployment, the value of k will be known in advance. For example, the IARPA-funded, Microsoft-led project on "*using mosquitoes as pathogen sensors*" has converged on a device that will make 64 irrevocable mosquito captures ($k = 64$) (Project Premonition 2015a).

However, our assumption that we can know the exact value of N ahead of time is clearly unrealistic. We typically will only be able to estimate the value of N. For example, for the mosquito capture problem, examining historical (analogue) trap captures might allow us to estimate that trap in a particular location, during a particular season, might expect to see approximately 3000–5000 insects.

Thus we will explore how incorrect estimations of N will affect the performance of our algorithm. Imagine we have *overestimated* N. If we have reached the end of data before we complete the learning phase of the algorithm, the result will be very undesirable: no item will be captured. Even if we are lucky enough to reach the capture phase, we will still be prone to failure, since the overestimation of N will result in over-setting the threshold value. Thus there is a very high probability that we will not be able to capture k items.

Now let us consider the opposite situation. If we have *underestimated* N, then we will clearly be able to capture k items, however we will suffer from an *opportunity cost*. How large is this opportunity cost?

To answer this, we used 10,000 z-normalized random-walk time series of length 512 as a dataset to test the proposed dynamic algorithm and set $k = 10$. We varied the actual value of N in range [1000, 10,000], and always told the batch algorithm the true N. In contrast, we always told the dynamic algorithm that the estimated N was 1000. That is to say, the dynamic algorithm would only be able to "see" the first 1000 time series. When it reached the end of the first 1000 time series and "thought" failure happened, it would apply the strategy in Definition 1.

Figure 13 shows the average MIN diversity result of both algorithms. Each data point is obtained by averaging 100 random shuffling results of the dataset. Due to the strategy employed in Definition 1, the result of the dynamic algorithm remains

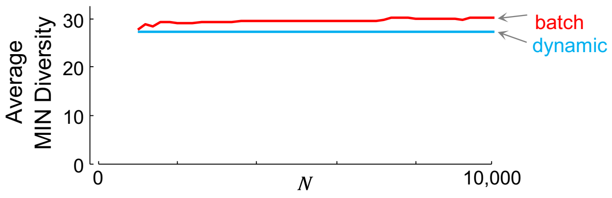


Fig. 13 The relative insensitivity of the dynamic algorithm to the situation when N is underestimated

constant as the true N goes beyond 1000. The batch algorithm result improves as N increases, but *very* slowly. Even when the true N reaches 10,000, which is ten times greater than the estimated N, the dynamic result is no more than 10 % worse than the batch result. So long as the estimated N is still large enough to reflect the distribution of the dataset, the performance of our proposed algorithm is assured.

To summarize, the effect of N on our proposed algorithm is highly asymmetric. If N is overestimated, we may not be able to capture enough items and that is very undesirable. In contrast, if N is underestimated, even by an order of magnitude, the performance of our algorithm will not be harmed much. Thus it is always desirable to give a slightly pessimistic estimate of N. In almost all cases this "*within an order of magnitude*" requirement will be easy to achieve. For example, in the insect sampling problem discussed in Sect. 6.3.1, we can use historical trap counts to estimate new trap counts within a factor of two (Silver 2008).

6.3 Case Studies

In this section, we will investigate how our proposed algorithm performs in three case studies. As before, we compare the numeric values of the MIN diversity with the batch "oracle". In addition, for these experiments we have access to class labels, so we measure the diversity in the more concrete sense of number of unique classes in the captured set, in this case comparing to the only obvious strawman in the literature, random sampling.

6.3.1 Case Study: Insect Sampling

The Intelligence Advanced Research Projects Activity (IARPA) has recently funded an ambitious 5-year project on pathogen surveillance (Project Premonition 2015a). Project *Premonition* seeks to detect pathogens in animals before these pathogens make people sick. It does this by treating a mosquito as a "device" that can find animals/humans and sample their blood. Project Premonition uses drones and new robotic mosquito traps to capture mosquitoes from the environment, and then analyzes their stomach contents for pathogens. Pathogens are detected by gene sequencing collected mosquitoes and searching for known and unknown pathogens in sequenced genetic material. A key element of the project is the ability to sense and capture individual mosquitoes. In some cases it may be useful to target particular species; however, in the first exploratory visit to a new site, it is necessary to capture a representative sample of the species present. In many cases we have only the vaguest understanding of which of the approximately 3600 species of mosquitoes are present in the target area. Capturing a representative sample is an ideal application for our algorithm.

Note that this domain violates one of our assumptions, that the distribution during the learning phrase is approximately the same as the distribution during the capture phase. This is because many insects, *especially* mosquitos (Chen et al. 2014), have widely varying flight behaviors, usually synchronized around the timing of dawn/dusk. For example, in a region where there are only *Culex tarsalis* and *Aedes aegypti*, at sunrise we are about three times more likely to encounter a *tarsalis*, but at sunset we are about fifteen times more likely to encounter *aegypti* (see Fig. 5 of Chen et al. 2014). There are two ways to deal with this. First, we can perform our learning phase at a fixed time, say 7:00 a.m. to 8:00 a.m. on one day, and perform our capture phase at the same time the next day. The other solution is simply to make sure the total time for the learning and capture phases is relatively short, say under 5 min. In the initial tests in Granada in 2015 (Project Premonition 2015b), the sensors detected insects at the rate of up to one per-second, so the short-window solution seems tenable.

One difficulty in evaluating our algorithm's efficacy in this domain is in obtaining ground truth. However, we are able to address this issue in the following way. We reared cohorts of insects in the same insectary, but in separate yet adjacent cages. Each cage had its own sensor, which could only sense the insects in that cage. However, we simultaneously fed all sensors into a single data recorder. Thus, the former gives us ground truth and the latter stream is the only data our algorithm has access to. Note that we *simulated* the capture of insects in this experiment, pending the fabrication of the insect capture device (Project Premonition 2015a). The six species we consider are:

• *Aedes aegypti* ♂	• *Culex quinque*fa*sciatus* ♂	• *Culex stigmatosoma* ♂
• *Aedes aegypti* ♀	• *Culex quinque*fa*sciatus* ♀	• *Culex tarsalis* ♀

For simplicity, we assume that the insects are equally frequent and their arrival order is random. Our task is as follows: we will see 1200 insects and we have the ability to capture exactly eight of them (i.e. $N = 1200, k = 8$). Thus we have the possibility to capture one of each insect type, plus two "spare" captures. If we had a random capture policy we would expect to capture a full set of insects only 11.5 % of the time. However, with our proposed dynamic algorithm, the probability of success is much higher. To show this, we randomly shuffle the arriving order of insects 500 times, and each time we capture eight items using both algorithms.

We then measure the number of distinct species captured (*NoDSC*) in each experiment. Ideally, $NoDSC = 6$, so that we capture at least one sample from each species, but sometimes we are not that lucky. Figure 14 shows the quality of the MIN diversity result of the proposed algorithm, and summarizes the 500 *NoDSC* results of both the random policy and our proposed algorithm.

Figure 14a compares the MIN diversity of our algorithm with both the batch algorithm and the random policy in the 500 experiments. The lowest MIN diversity value

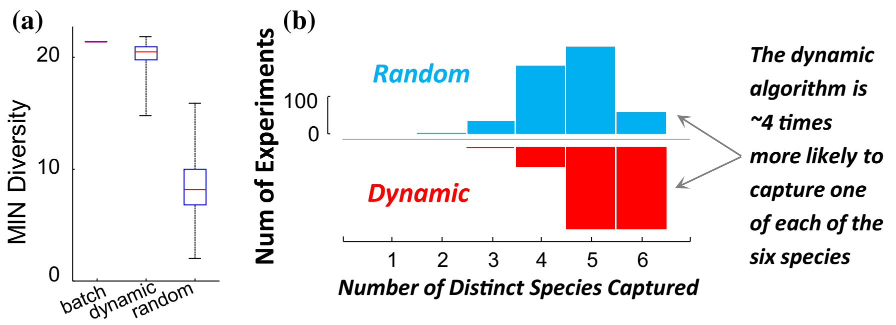


Fig. 14 **a** The effectiveness of the dynamic algorithm on insect data. **b** The insect *NoDSC* result of the random policy (*blue*) versus the proposed dynamic algorithm (*red*) (Color figure online)

of the dynamic algorithm results from two failure cases. The remaining 498 experiments have achieved over 90 % the quality of batch algorithm, and the average MIN diversity is about two times better than the random policy result, validating the diversification quality of our proposed algorithm. Furthermore, Figure 14b shows that with random policy, only 54 out of 500 experiments succeed in capturing all six species (agreeing with our theoretical estimate), while with our proposed dynamic algorithm we succeed more than four times as often (224 out of 500 experiments). The random algorithm captures three or less species about 6.4 % of the time, whereas for the dynamic algorithm that number is just 0.2 %.

6.3.2 *Case Study: Fish Sampling*

Following our success with insects, we consider a very similar experiment with fish, inspired by the example of fish ladders discussed in Sect. 1 (cf. Fig. 4). While the scenario of robotically capturing *individually* selected fish may seem farfetched, there are multiple ongoing research efforts in this direction, both for edible fish capture (Jonsson 2015) and invasive species mitigation (Zhang et al. 2015).

Our dataset includes four different fish species, with fifty examples of each. We assume that the fish enter the fish ladders in random order. We will see 200 fish passing by and we are allowed to capture exactly four of them (i.e. $N = 200, k = 4$). We randomly shuffle the arriving order of the fish 500 times, and each time we measure the number of distinct species captured (*NoDSC*).

Figure 15 shows the diversification quality of all three methods, and compares the *NoDSC* histogram of our proposed algorithm with the random policy.

From Fig. 15b we can see that although N is relatively small, the dynamic algorithm still achieves over 75 % of the diversification quality of batch algorithm in more than 75 % experiments. Only seven out of 500 experiments end in failure (which determines the lowest dynamic MIN diversity value), and the MIN diversity result is more than twice of the random policy result by average. The *NODSC* result in Fig. 15c shows that the random policy succeeds in capturing all four species 8.8 % of the time, while our algorithm succeeds 21.6 % of the time. In about 35.6 % experiments the random

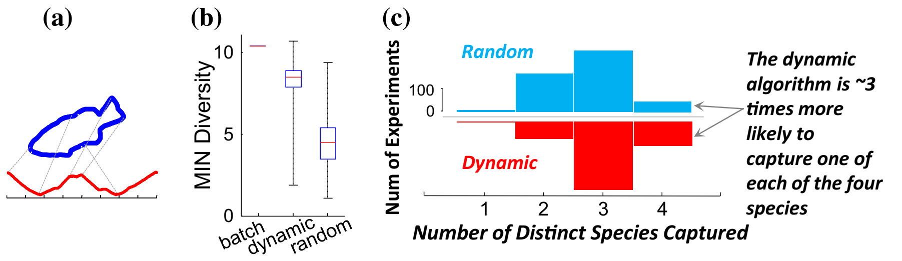


Fig. 15 **a** We convert the fish outline to a pseudo time series, an effective representation for this domain. **b** The effectiveness of the dynamic algorithm on fish data. **c** The fish *NoDSC* result of the random policy (*blue*) versus the proposed dynamic algorithm (*red*) (Color figure online)

policy captures no more than two fish species, but our algorithm reduces the rate of this undesirable occurrence by more than a half to just 16.2 %.

Because N is much smaller than the insect sampling case, it is easier for the random algorithm to achieve a higher *NoDSC*. Moreover, because the data size is no longer large enough to robustly represent the distribution of each fish species' shape, the distributions of the learning phase and capture phase can differ significantly. Nevertheless, our experimental results provide strong evidence that our algorithm outperforms the random policy significantly.

6.3.3 Case Study: Rock Sampling

Our final case study is inspired by the capabilities and limitations of NASA's Mars Curiosity Rover, as shown in Fig. 16a (Anderson et al. 2010). While the previous case studies both assume that the classes are approximately equally frequent, this will clearly not be generally true. Imagine that Curiosity is exploring a small region of Mars in which there are four types of rocks:

• *mafic* (26)	• *quartzite* (11)	• *marble* (14)	• *schist* (19)

The values in parenthesis are the number of rocks in each type. The robot can detect the spectral reflectance of rocks with low cost (Cerra et al. 2011), but must make only five irrevocable choice to drill into a rock for further analysis, moving the powdered samples from the rocks' interiors to the onboard laboratory instruments. The choices are irrevocable because of the limited chemical supplies in the onboard laboratory and the inevitable wear on the drill bit. Thus, in this case, $N = 70$ and $k = 5$ (we have the possibility to capture all four types of rock, plus one spare capture).

We use the data from ASTER Spectral Library (Baldridge et al. 2009) to simulate this situation. Similar to the previous case studies, we randomly shuffle the order of the rocks 500 times, and compare the MIN diversity results of the batch algorithm, the proposed dynamic algorithm and the random policy. We also compare the *NoDRTC*

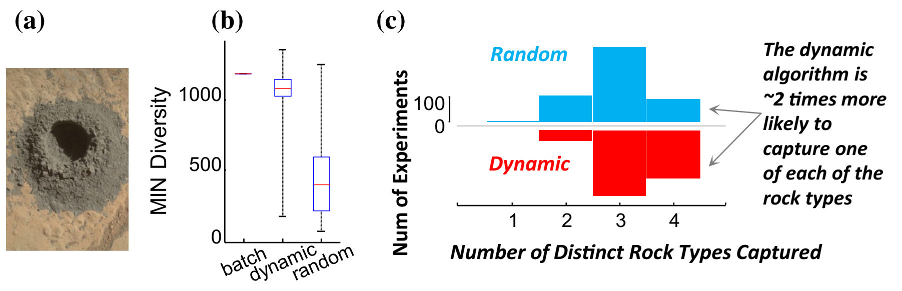


Fig. 16 **a** The first hole drilled by Rover. **b** The effectiveness of the dynamic algorithm on rock data. **c** The insect *NoDRTC* result of the random policy (*blue*) versus the proposed dynamic algorithm (*red*) (Color figure online)

("*R*ock *T*ypes") result of the proposed algorithm with the random policy. The results are shown in Fig. 16.

Because N is very small, our proposed algorithm has a failure case in 22 out of 500 experiments. Nevertheless, the MIN diversity result is within 8 % of the batch result in over 75 % experiments, and the average MIN diversity is about three times that of the random policy. The dynamic algorithm captures all four types of rocks in 39 % of the experiments, which is more than twice the random policy result (18.4 %).

7 Conclusion

We introduced the problem of irrevocable-choice diversification sampling, and showed that it maps well onto diverse real-world problems of commercial and scientific interest. We further demonstrated a somewhat surprising result, that in spite of the severe constraints of the problem setting (streaming data, irrevocable-choices), we can typically produce results that are very close to those of an oracle algorithm that sees all the data at once. We have several plans for future work. First, we would like to investigate the problem from a more theoretical point of view. Second, we will have several opportunities to do irrevocable-choice diversification sampling of insects in the coming years (Project Premonition 2015a), and it will be interesting to see how our results in Sect. 6.3 generalize to field settings. Finally, we wish to consider streaming versions of other algorithms under the irrevocable-choice constraints, including clustering and outlier detection.

References

Aggarwal CC (2006) Data streams: models and algorithms (advances in database systems). Springer, New York

Anderson R et al (2010) Mars Science Laboratory participating scientists program proposal information package. NASA/Jet Propulsion Laboratory, Pasadena

Baldridge AM, Hook SJ, Grove CI, Rivera G (2009) The ASTER spectral library version 2.0. Remote Sens Environ 113(4):711–715

Begum N, Keogh E (2014) Rare time series motif discovery from unbounded streams. Proc VLDB Endow 8(2):149–160

Bowman A, Azzalini A (1997) Applied smoothing techniques for data analysis: the kernel approach with S-Plus illustrations. Oxford University Press, New York

Cerra D, Bieniarz J, Avbelj J, Reinartz P, Mueller R (2011) Compression-based unsupervised clustering of spectral signatures. Whispers, Oro Valley

Chen Y, Why A, Batista G, Mafra-Neto A, Keogh E (2014) Flying insect classification with inexpensive sensors. J Insect Behav 27(5):657–677

Cormode G, Hadjieleftheriou M (2010) Methods for finding frequent items in data streams. VLDB J 19(1):3–20

Drosou M, Pitoura E (2012a) Disc diversity: result diversification based on dissimilarity and coverage. Proc VLDB Endow 6(1):13–24

Drosou M, Pitoura E (2012b) Dynamic diversification of continuous data. In: Proceedings of the 15th EDBT/ICDT, ACM, pp 216–227

Erkut E (1990) The discrete p-dispersion problem. Eur J Oper Res 46(1):48–60

Erkut E, Ülküsal Y, Yenicerioğlu O (1994) A comparison of p-dispersion heuristics. Comput Oper Res 21(10):1103–1113

Ferguson TS (2006) Optimal stopping and applications. Online Book. www.math.ucla.edu/~tom/Stopping/Contents.html

Fønss A, Munksgaard L (2008) Automatic blood sampling in dairy cows. Comput Electron Agric 64(1):27–33

Ghosh JB (1996) Computational aspects of the maximum diversity problem. Oper Res Lett 19(4):175–181

Goldberg D (2011) Huxley: a flexible robot control architecture for autonomous underwater vehicles. In: Proceedings of IEEE OCEANS conference (Spain, 2011), pp 1–10

Hill TP (2009) Knowing when to stop: how to gamble if you must—the mathematics of optimal stopping. Am Sci 97(2):126–133

Honda MC, Watanabe S (2007) Utility of an automatic water sampler to observe seasonal variability in nutrients and DIC in the Northwestern North Pacific. J Oceanogr 63(3):349–362

Jonsson F (2015) Real-time fish type recognition in underwater images for sustainable fishing. Technical report. Uppsala University, Uppsala

Matlab ksdensity function (2016) http://www.mathworks.com/help/stats/ksdensity.html

Minack E, Siberski W, Nejdl W (2011) Incremental diversification for very large sets: a streaming-based approach. In: ACM SIGIR (July 2011), pp 585–594

Peskir G, Shiryaev A (2006) Optimal stopping and free-boundary problems. Lectures in Mathematics. ETH, Zürich

Project Premonition (2015a) http://www.research.microsoft.com/en-us/um/redmond/projects/projectpremonition/default.aspx. Accessed 2 Aug 2015

Project Premonition (2015b) URL of Video of First Trials in Granada. Seeking to prevent disease outbreaks. https://www.youtube.com/watch?v=v8uG82Z7VLM

Project Webpage (2016) https://sites.google.com/site/irrevocablestreamingdata/

Rasmussen SL, Starr N (1979) Optimal and adaptive stopping in the search for new species. J Am Stat Assoc 74(367):661–667

Roman C, Mather R (2010) Autonomous underwater vehicles as tools for deep-submergence archaeology. Eng Marit Environ 224(4):327–340

Silver JB (2008) Chapter 14: estimating the size of the adult population. Mosquito ecology field sampling methods, 3rd edn. Springer, New York

Vitter JS (1985) Random sampling with a reservoir. ACM Trans. Math Softw (TOMS) 11(1):37–57

Webster G, Agle DC (2012) Mars Science Laboratory/Curiosity Mission status report. NASA, New York

Zhang D et al (2015) Automatic fish taxonomy using evolution-constructed features for invasive species removal. Pattern Anal Appl 18(2):451–459