Lec 08–??: Mining Data Streams

COSC–254 – February 18–??, 2019
Outline

*Data streams*: motivation, applications, model(s), queries

Approximate *query answering*: reservoir sampling

Approximate *set membership*: Bloom filters

Approximate *counting*: The Flajolet-Martin approach

Approximate *counting on sliding windows*: The DGIM Algorithm
Data streams

Sensor data: *continuously* transmit (measurements of) quantities of interest

   - temperature, location, traffic, stock prices, web search queries, …

*Stream* of data elements:

\[ e_{t-2}, \quad e_{t-1}, \quad e_t, \quad e_{t+1}, \quad \ldots \quad \text{text}

Element seen at time \( t \geq 0 \)

**EXAMPLE:** elements are *tuples* of (temperature, wind speed, humidity)

3-tuple

\[ (15^\circ, 20 \text{mph}, 52\%), \quad (18^\circ, 10 \text{mph}, 64\%), \quad \ldots \]
The dataset is *never complete*: data points are appended at each timestep:

\[
\mathcal{D}_{t+1} = \mathcal{D}_t \cup \{ e_{t+1} \}
\]

\( \mathcal{D}_0 = \emptyset, \mathcal{D}_1 = \{ e_1 \}, \ldots \)

**Task:** for each \( t \), compute quantity/ies of interest \( q = f(\mathcal{D}_t) \) (standing queries).

**Example:** wind-chill at each time \( t \), average temperature over the past 7 days.
Data streams

Properties of the data that make the task hard:

1) The data is essentially *infinite*;
2) Input elements arrive *very fast*
   (think: Instagram photos, stock prices, security camera frame)

Consequences:

1) *cannot store the entire stream* accessibly;
2) must *compute query answer fast*. 
Stream processing model

Figure from slides at http://mmds.org
Queries

*Filtering*: select all elements with property $x$

*Counting distinct* elements (possibly in the last $k$ elements seen)

*Moment estimation*: estimate the average or the standard deviation
   (possibly of the last $k$ elements)

*Find frequent elements*
Applications

How many distinct users visited my website in the last month?

Mining streams of web search queries:
  what queries are *more frequent* today than yesterday?

Mining click streams:
  what web pages are getting an *unusual* number of hits in the past three hour?

Mining social network status updates:
  is there an earthquake happening right now in California? A protest in Cairo?
More applications

Sensor networks:
With a million of sensors sending 4 bytes every 1/10 of seconds,
you get a million data points per 1/10 of second, 3.5 terabytes per day.

IP packets monitored at a switch:
Is there a flow of packets that would benefit from different routing decision?
Are there unusual patterns in the flow? (denial-of-service attacks)
Query answers

Answering queries *exactly* may not always be possible because of

1) the *limited working space*
2) computing the exact answer \( q_t = f(D_t) \) may *take too long*

**Example:** Count the distinct elements.

Can’t count exactly if the set of distinct elements is larger than the number of elements I can store

**Example:** Mine the frequent itemsets from the last \( k \) elements.

Would take too long
Approximations

ISSUE: Impossible to compute the exact answer and compute it fast.

SOLUTION: Compute *approximate answer* $\tilde{q}_t = \tilde{f}(D_t)$

$$\tilde{q}_t \approx q_t \quad \text{for every } t > 0$$

**Computer scientist task:** Given a query $f$, design an algorithm $\tilde{f}$ that:

1) “approximate” $f$ for *all possible* input datasets;
2) uses a *small working space*;
3) is *fast* in computing the approximation
Why shall we be happy with approximate answers?

1) We cannot compute anything else;

2) *High-quality* approximations are still *very useful*;

3) Exact answers have *little value* in a streaming setting;
Outline

- **Data streams**: motivation, applications, model(s), queries

  Approximate *query answering*: reservoir sampling

  Approximate *set membership*: Bloom filters

  Approximate *counting*: The Flajolet-Martin approach

  Approximate *counting on sliding windows*: The DGIM Algorithm
We cannot store the whole stream? Let’s store a subset $S_t$ of $D_t$

How to compute the approximate answer $\tilde{q}_t$?
Possible answer: $\tilde{q}_t = f(S_t)$ (i.e., use the same $f$)

**Example:**

$f = \text{average}: \tilde{q}_t$ works well on some subsets;
We need a way to keep a representative subset
Easiest way to build a representative subset: select one \textit{(uniformly) at random}

Each element has the same probability of being in the sample (being \textit{sampled})

How to create a random sample?

Approach 1: select a \textit{fixed proportion} of elements in the stream (1 in 10)

Approach 2: Maintain a random sample of fixed size size
Sampling a fixed proportion

Scenario: Web search query stream:

\((\text{user}_t, \text{search}_t), (\text{user}_{t-1}, \text{search}_{t+1}), \ldots\)

Query: What fraction of the typical user’s queries are repeated?

Naïve approach to build the sample:

For each \(t\), generate a random integer \(i_t\) from \([0 \ldots 9]\)
Add the element \(e_t\) to \(S_t\) if \(i_t = 0\).

Answering the query:

for each user \(u\), count the fraction \(r_u\) of repeated queries in \(S_t\),
then take the average of \(r_u\) over the users
Issues with the naïve approach

Suppose each user issues $x$ queries once, and $d$ queries twice (total $x + 2d$ queries)

Correct query answer: $\frac{d}{x + d}$

The typical sample will contain:
- $x/10$ of the singleton queries
- $d/100$ pairs of duplicates ($d/100 = d \times \left(\frac{1}{10} \times \frac{1}{10}\right)$)
- $18d/100$ of the $d$ duplicates, each appearing exactly once. ($18d/100 = d \times \left(\frac{1}{10} \times \frac{9}{10} \right) + \left(\frac{9}{10} \times \frac{1}{10}\right)$)

Naïve approach answer:

$$\frac{x/10 + d/100 + 18d/100}{10x + 19d} = \frac{d}{10x + 19d}$$
Solution: sample users!

Pick $1/10^{th}$ of users and add all their searches to the sample.

How to decide whether a user is one of the “sampled” one?

Use a hash function $h$ that hashes user names uniformly into 10 buckets. If the $h(user_t) = 0$, add $e_t = (user_t, search_t)$ to $S_t$. 
Generalized solution

Stream of tuples with keys:

Key is a subset of the components of each tuple (e.g., user\textsubscript{t})
Choice of key depends on application

To get a sample of $a/b$ fraction of the stream:

Hash each tuple’s key uniformly into $b$ buckets $[0, \ldots, b - 1]$
Add the tuple to the sample if the hash value is less than $a$

Example: To generate a 30\% sample, what is $b$ and what is $a$?

$b = 10$ and $a = 3$. 
A problem with the previous approach is that the size of the sample grows with time.

Our memory may not grow as fast. It may even be fixed to exactly \( s \) tuples.

How to build a fixed-size random sample that is representative of all elements seen so far?

For all time steps \( k \),

each of the \( k \) elements seen so far must have the same probability of being in \( S_k \).
Reservoir sampling

Algorithm:

\[ S \leftarrow \emptyset \]

If \( t \leq s \), store the \( e_t \) in \( S \)

Else \hspace{1em} // i.e., when \( t > s \)

\hspace{1em} \text{flip a biased coin that has probability of head equal to } \frac{s}{t}

\hspace{1em} \text{If outcome is tail, discard } e_t

\hspace{1em} \text{Else } \hspace{1em} // \text{i.e., when outcome is head}

\hspace{2em} \text{choose an element of } S \text{ uniformly at random and replace it with } e_t

Lemma:

At each time \( t \), the sample \( S_t \) is such that each element \( e_k, k \leq t \) has probability \( \min\{1, \frac{s}{t}\} \) of being in \( S_t \).

Proof: Next time