Mining Data Streams (Part 2)

Filtering Data Streams

- Each element of data stream is a tuple
- Given a list of keys $S$
- Determine which elements of stream have keys in $S$
- Obvious solution: hash table
  - But suppose we don’t have enough memory to store all of $S$ in a hash table
  - e.g., we might be processing millions of filters on the same stream

Applications

- Example: email spam filtering
  - We know 1 billion “good” email addresses
  - If an email comes from one of these, it is NOT spam
  - Publish-subscribe
  - People express interest in certain sets of keywords
  - Determine whether each message matches a user’s interest

First Cut Solution – (1)

- Create a bit array $B$ of $m$ bits, initially all 0’s.
- Choose a hash function $h$ with range $[0,m)$
- Hash each member of $S$ to one of the bits, which is then set to 1
- Hash each element of stream and output only those that hash to a 1

First Cut Solution – (2)

First Cut Solution – (3)

- $|S| = 1$ billion, $|B| = 1GB = 8$ billion bits
- If a string is in $S$, it surely hashes to a 1, so it always gets through
- Approximately most $1/8$ of the bit array is 1, so about $1/8^{th}$ of the strings not in $S$ get through to the output (*false positives*)
  - Actually, less than $1/8^{th}$, because more than one key might hash to the same bit
**Throwing Darts**

- If we throw \( m \) darts into \( n \) equally likely targets, what is the probability that a target gets at least one dart?
- Targets = bits, darts = hash values

**Throwing Darts – (2)**

\[
\frac{m}{n} \cdot (1 - \frac{1}{n})^{m/n} = 1 - e^{-m/n}
\]

**Throwing Darts – (3)**

- Fraction of 1's in array = probability of false positive = \( 1 - e^{-m/n} \)
- Example: \( 10^9 \) darts, \( 8 \times 10^9 \) targets.
  - Fraction of 1's in B = \( 1 - e^{-1/8} \) = 0.1175.
  - Compare with our earlier estimate: \( 1/8 \) = 0.125.

**Bloom Filter**

- Say \( |S| = m \), \( |B| = n \)
- Use \( k \) independent hash functions \( h_1, \ldots, h_k \)
- Initialize B to all 0's
- Hash each element \( s \) in \( S \) using each function, and set \( B[h_i(s)] = 1 \) for \( i = 1, \ldots, k \)
- When a stream element with key \( x \) arrives
  - If \( B[h_i(x)] = 1 \) for \( i = 1, \ldots, k \), then declare that \( x \) is in \( S \)
  - Otherwise discard the element

**Bloom Filter -- Analysis**

- What fraction of bit vector B is 1’s?
  - Throwing \( km \) darts at \( n \) targets
  - So fraction of 1’s is \( (1 - e^{-km/n}) \)
- \( k \) independent hash functions
- False positive probability = \( (1 - e^{-km/n})^k \)

**Bloom Filter – Analysis (2)**

- \( m = 1 \) billion, \( n = 8 \) billion
  - \( k = 1: (1 - e^{-1/8}) = 0.1175 \)
  - \( k = 2: (1 - e^{-1/4})^2 = 0.0493 \)
- What happens as we keep increasing \( k \)?
  - “Optimal” value of \( k: n/\ln 2 \)
Bloom Filter: Wrap-up

- Bloom filters guarantee no false negatives, and use limited memory
- Great for pre-processing before more expensive checks
- E.g., Google’s BigTable, Squid web proxy
- Suitable for hardware implementation
- Hash function computations can be parallelized

Counting Distinct Elements

- Problem: a data stream consists of elements chosen from a set of size \( n \). Maintain a count of the number of distinct elements seen so far.
- Obvious approach: maintain the set of elements seen.

Applications

- How many different words are found among the Web pages being crawled at a site?
  - Unusually low or high numbers could indicate artificial pages (spam?)
- How many different Web pages does each customer request in a week?

Using Small Storage

- Real Problem: what if we do not have space to store the complete set?
- Estimate the count in an unbiased way.
- Accept that the count may be in error, but limit the probability that the error is large.

Flajolet-Martin* Approach

- Pick a hash function \( h \) that maps each of the \( n \) elements to at least \( \log_2 n \) bits
- For each stream element \( a \), let \( r(a) \) be the number of trailing 0’s in \( h(a) \)
- Record \( R = \) the maximum \( r(a) \) seen
- Estimate = \( 2^R \).

* Really based on a variant due to AMS (Alon, Matias, and Szegedy)

Why It Works

- The probability that a given \( h(a) \) ends in at least \( r0's \) is \( 2^{-r} \)
- Probability of NOT seeing a tail of length \( r \) among \( m \) elements is \( \left(1 - 2^{-r}\right)^m \)

Prob. All end in fewer than \( r0's \). Prob. a given \( h(a) \) ends in fewer than \( r0's \).
**Why It Works — (2)**

- Since $2^r$ is small, prob. of NOT finding a tail of length $r$ is:
  - If $m << 2^r$, tends to 1. So probability of finding a tail of length $r$ tends to 0.
  - If $m >> 2^r$, tends to 0. So probability of finding a tail of length $r$ tends to 1.
- Thus, $2^R$ will almost always be around $m$.

**Why It Doesn’t Work**

- $E(2^R)$ is actually infinite.
  - Probability halves when $R \rightarrow R + 1$, but value doubles.
  - Workaround involves using many hash functions and getting many samples.
- How are samples combined?
  - Average? What if one very large value?
  - Median? All values are a power of 2.

**Solution**

- Partition your samples into small groups
- Take the average of groups
- Then take the median of the averages

**Generalization: Moments**

- Suppose a stream has elements chosen from a set of $n$ values.
- Let $m_i$ be the number of times value $i$ occurs.
- The $k^{th}$ moment is

**Special Cases**

- $0^{th}$ moment = number of distinct elements
  - The problem just considered.
- $1^{st}$ moment = count of the numbers of elements = length of the stream.
  - Easy to compute.
- $2^{nd}$ moment = surprise number = a measure of how uneven the distribution is.

**Example: Surprise Number**

- Stream of length 100; 11 distinct values
  - Item counts: 10, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9
  - $\text{Supprise} # = 910$
- Item counts: 90, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
  - $\text{Suprise} # = 8,110$. 
AMS Method

- Works for all moments; gives an unbiased estimate.
- We’ll just concentrate on 2nd moment.
- Based on calculation of many random variables $X$.
- Each requires a count in main memory, so number is limited.

One Random Variable ($X$)

- Assume stream has length $n$.
- Pick a random time to start, so that any time is equally likely.
- Let the chosen time have element $a$ in the stream.
- Maintain a count $c$ of the number $a$’s in the stream starting at the chosen time.
- $X = n*(2c-1)$
- Store $n$ once, count of $a$’s for each $X$.

Expectation Analysis

\[ X = n(2c - 1) \]
\[ E[X] = (1/n)\sum_{all \ times} n \ (2c - 1) \]
\[ = \sum_{all \ times} (2c - 1) \]
\[ = \sum_{a} (1 + 3 + 5 + \ldots + 2m_a - 1) \]
\[ = \sum_{a} (m_a)^2 \]

Combining Samples

- Compute as many variables $X$ as can fit in available memory.
- Average them in groups.
- Take median of averages.

Problem: Streams Never End

- We assumed there was a number $n$, the number of positions in the stream.
- But real streams go on forever, so $n$ is a variable – the number of inputs seen so far.

Fixups

1. The variables $X$ have $n$ as a factor – keep $n$ separately; just hold the count in $X$.
2. Suppose we can only store $k$ counts. We must throw some $X$’s out as time goes on.
   - Objective: each starting time $t$ is selected with probability $k/n$.
   - How can we do this?
Exponentially Decaying Windows

- Stream $a_1, a_2, ...$
- Define exponentially decaying window at time $t$ to be: $\sum_{i=1,2,...,t} a_i (1-c)^{t-i}$
- $c$ is a constant, presumably tiny, like $10^{-6}$ or $10^{-9}$.

Applications

- Key use case is when the stream's statistics can vary over time
- Finding the most popular elements “currently”
  - Stream of Amazon items sold
  - Stream of topics mentioned in tweets
  - Stream of music tracks streamed