Big Data & Scripting
Part II
Streaming Algorithms
Counting Distinct Elements
counting distinct elements

problem formalization

- input: stream of elements $o$ from some universe $U$
  - e.g. ids from a set of allowed ids
- elements can appear arbitrary often, not all of them appear
- question: how many different elements did appear?

- application example: counting the number of visitors (IPs) of a website

- storing the elements is not an option
  - estimate number using hashing and probabilities
Flajolet-Martin algorithm – idea

• binary function $f : X \rightarrow \{0, 1\}$, $x_i = x_j \Rightarrow f(x_i) = f(x_j)$
• $P(f(x_i) = 1) = p$ identically, independently distributed for all $x_i$

• assume we encountered $m$ different $x_i$ \(^1\)
• what is the probability $P_m$, that $f(x_i) = 0$ for all $i$?
  \(\Rightarrow P_1 = (1 - p)\) for one $x_i$, $P_m = (1 - p)^m$ for $m$ different $x_i$
• $P_m$ depends on the number of unique $x_i$, not on the total number
• exploit this to get an estimate of $m$

\(^1\)that’s actually what we try to find out
Flajolet-Martin algorithm – idea

- \((1 - p)^m = \left( (1 - p) \frac{1}{p} \right)^{mp} \approx e^{-mp} \)
  
  for small \( p \): \((1 - p) \frac{1}{p} \approx e^{-1} \) (Euler’s number)

- probability \( P_m \) that \( f(x_i) = 0 \) for \( m \) different \( x_i \)
  \( P_m = e^{-mp} \)

- if \( m \ll 1/p \): \( P_m \to 1 \)
- if \( m \gg 1/p \): \( P_m \to 0 \)
- \( p \) should be small (for derivation and for large \( m \))
Flajolet-Martin algorithm – bitmask & tail length

- use hash function \( h : X \rightarrow \{0, 1\}^K \)
  e.g. binary representation of hash values
- assume bits of \( h(x_i) \) are uniform, independent distributed

The tail

- idea: count number of trailing zeros
- \( \text{tail}(h) = \max\{j : h \mod 2^j = 0\} \)

Bitmask

- introduce a new random variable bitmask \( \in \{0, 1\}^M \)
- initialize bitmask\( (i) = 0 \) for \( i = 0 \ldots M - 1 \) (zero indexed)
- for every \( x_i \): bitmask(tail\( (h(x_i)) \)) \( \leftarrow 1 \)
Flajolet-Martin algorithm – bitmask & tail length

- bitmask() keeps tracks of tail lengths seen so far
- used to extend approximate counting by random variables
- bitmask(0) is expected to be set to 1 by 1/2 of the $x_i$
- bitmask(1) is expected to be set to 1 by 1/4 of the $x_i$, ... 
- bitmask($i$) is almost certainly 0 if $i \gg \log_2 m$
- bitmask($i$) is almost certainly 1 if $i \ll \log_2 m$
  (recall that $m$ is the number of unique elements seen)
- for counting, derive random variable $R$:
  $R($bitmask$) = \min\{i : \text{bitmask}(i) = 0\}$
- it can be shown that: $E(R) \approx \log_2 \phi m$
  where $\phi = 0.77351\ldots$
- $m$ can be approximated as $m \approx \phi^{-1}2^R$
Flajolet-Martin algorithm – accuracy

the idea so far

- probability of $f(x_i) = 1$ depends $m$
- use this for estimation
- for tail length: $k$ longest tail of hash value observed
  $\rightarrow$ estimate $2^k$ elements

problems/refinements

- $R$ is not very precise (large standard deviation $\sigma(R) \approx 1.12$)
- precision could be increased by averaging multiple functions
- but: determining $h(x_i)$ is costly
Flajolet-Martin algorithm – accuracy

- using $k$ different hash functions (and bitmasks) increases accuracy:
  \[ R = \frac{1}{k} \sum_{j=1}^{k} R(\text{bitmask}^j) \]
  has std. dev. $\approx 1.12/\sqrt{k}$
  \[ \text{estimate is exponential of } R \text{ in } \rightarrow \text{ large influence on mean value} \]

- improve accuracy with only one hash function:
  - use $k$ different bitmaps bitmask$^0$, $\ldots$, bitmask$^{k-1}$
  - for each $x_i$ update only bitmask$^{(h(x_i) \mod k)}$
  - \[ R = \frac{1}{k} \left( \sum_{j=0}^{k-1} R(\text{bitmask}^j) \right) \]

- std. dev. $\approx 1.12/\sqrt{k}$ still holds
Flajolet-Martin – algorithm

**input:**
stream of numbers $X$ (e.g. converted strings)
precision parameter $k$

**output:** estimate $m$ of unique numbers within $X$

**algorithm:**
bitmask=bitmask$[k]$ // array of bitmasks initialized with 0
while $x_i = \text{readNext}()$
{
    $h = h(x_i)$
    update bitmask[$h \mod k$] with tail($h$)
}
mean=mean($\{R(\text{bitmask}[j]) : j = 0, \ldots, k\}$)
return($2^{\text{mean}}/0.77351$);

source and additional information: Flajolet, P.; Martin, G. N.: “Probabilistic counting algorithms for data base applications”, 1985
## Sampling in Streams

### Sample

- Descriptive subset of the original data
- Approximate true distribution

### Problem

- Selection of samples (how?)
- Introduction of bias
- Sampling over time

---

## sampling – example

We use the following example setting to demonstrate problems in sampling:

<table>
<thead>
<tr>
<th>data and problem</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>incoming data:</strong></td>
</tr>
<tr>
<td>- data points: <code>(user, query, time)</code></td>
</tr>
<tr>
<td>- e.g. DB system serving clients</td>
</tr>
<tr>
<td><strong>queries appear repeatedly</strong></td>
</tr>
<tr>
<td><strong>goal:</strong> estimate number of</td>
</tr>
<tr>
<td>- unique</td>
</tr>
<tr>
<td>- <code>2,3, ...</code> times repeated queries</td>
</tr>
</tbody>
</table>
sampling – naive approach (does not work)

- sample 10% of the data points (randomly)
- estimate repetitions from resulting sample

example: unique and double queries

- unique queries $A$ and double queries $B_1,B_2$
- $P(A \text{ in sample}) = 0.1$, analogous for $B_1$ and $B_2$
- $P(B_1 \text{ in sample and } B_2 \text{ not in sample}) = 0.1 \cdot (1 - 0.1) = 0.09$
  in the sample, these appear as unique queries!
- $P(B_1 \text{ and } B_2 \text{ in sample}) = 0.1 \cdot 0.1 = 0.01$
sampling – using hash functions

solution: frequency-independent sampling

- sample items independent of their frequency
- if item is in sample, retain all copies

important assumption: objects are approximately equally distributed to hash values (buckets)

algorithm - sample \((j/k)N\) elements

- hash function \(h: X \rightarrow \{0, \ldots, k - 1\}\)
- if \(h(x_i) < j\) add \(x_i\) to sample

- works, if \(N\) is known in advance and \(j/k\) is small enough
- can be adapted to arbitrary \(N\) and fixed sample size
## sampling – adaptive sample size

### problem

- number of distinct objects unknown in the beginning
- sample should be of fixed or limited size

### approach:

- use large number of buckets (e.g. keep all $x_i$)
- sample $x_i$ as before (store each sample in it’s bucket)
- when sample grows too large, drop buckets from sample
sampling – a concrete example

**problem:** extract 2GB sample from incoming ids

- input objects with id as string $s_0s_1\ldots s_k$,
  - $k \in \{3, \ldots, 20\}$
  - $s_i \in \{a, \ldots, z, A, \ldots, Z\} = \text{Var}$
  - $|\text{Var}| = 52$
- convert id to number:
  - assign value to each character: $^3\nu(a) = 0, \nu(b) = 1, \ldots, \nu(Z) = 51$
  - convert to number: $\text{number}(s_0\ldots s_k) = \sum_{i=1}^{k} \nu(s_i) \cdot 52^i$
  - for implementation, use *Horner’s method:*
    $\nu_0 = s_k, \nu_1 = \nu_0 \ast 52 + s_{k-1}, \ldots, \text{number}(s_0\ldots s_k) = \nu_{k+1}$
- apply hash function $h(\nu) = \nu \mod 1000$
- sample $h(\nu) < s$ starting with $s = 1000$
- when mem usage of samples $> 2\text{GB}$, $k$ is decremented by 10

[^3]: or use their ASCII/UTF value
sampling – problems

• characters are often not equally distributed (e.g. in languages)
  – we ignore that for now
  – assume, hashing takes care of that

• hash values get very large (e.g. $52^3 = 140608$)
  – problem: primitive data types (int, long) are limited
  – counter measure: apply $h()$ to $v_i$ in every step
    works if $a = 0$ because of modulo arithmetic

• what if $k = 1$ and sample is still $>2\text{GB}$?
  – rerun, use more buckets (e.g. 10000)
  – or use second stage of hashing:
    apply new hash function to selected samples
    (those passing the first filter only)
maintaining sets – motivating scenario

• a sample \( S \subset X \) can be used to learn properties of the overall distribution
  – example: queries and their frequencies
• in the following, we derive further statistics about the samples
  – e.g.: frequency distributions (how many queries have certain frequencies)
• we need to decide, whether \( x_i \in S \)
• we could exploit the filters resulting from the sampling process
• problem: new (unseen) objects could end up in “sample buckets”
• aim: decide set membership without further information
maintaining sets – a first approach

- use fine grained hashing to split samples into small chunks
- remember buckets that contain $x_i \in S$
- discard elements not in these buckets

- result: “dirty” filter
  - if $x_i \in S$ it is retained
  - if $x_i \notin S$ it **might** be discarded

- in the following: improve this general idea using hash functions

⇒ Bloom Filter
Bloom filter – idea and introduction

- input: large set $S$ of objects
- problem: decide whether $x_i \in S$ without storing $S$

formalizing the idea - (not the Bloom filter yet)

- hash function $h : X \rightarrow H$, $|H|$ very large
- bit array $B$ with $B[i] = \begin{cases} 1 , & \text{if } \exists s \in S : h(s) = i \\ 0 , & \text{else} \end{cases}$
- for each $s \in S \rightarrow B[h(s)] == 1$
- for each $s \notin S \rightarrow B[h(s)] == 1$ (false positive) or $B[h(s)] == 0$ (reject)
- elements of $S$ pass, others might pass but hopefully not all

next: try to minimize false positives
Bloom filter – definition and usage

Bloom filter

- set $S$ and bit array $B$ as before
- **collection** of hash functions: $h_1, \ldots, h_k$ mapping to $H$
- set $B[i] = 1$, if for any $s \in S$ and any $h_j$: $h_j(s) = i$
- object $o$ passes, if $B[h_i(o)] = 1$ for **all** $i \in \{1, \ldots, k\}$
- $B[h_i(o)] = 0$ for any $i$, $o$ does not pass

- spreading to more hash functions avoids collisions
  - sufficient, if only one $h_i$ does not produce a collision
- allows derivation of probabilistic properties
  - e.g. how many $h_i$, and size of $H$ for a given $S$
Bloom filter: properties and extensions

- array size $|A| = m$ (memory usage) e.g. $10^6$
- $k$ hash functions (time consumption of test) e.g. 5
- $|S| = n$ elements stored (yield in a sense) e.g. $10^5$
- approximate probability of false positive: $\left(1 - e^{-kn/m}\right)^k$
  - example: $\left(1 - e^{-5 \cdot 10^5/10^6}\right)^5 \approx 0.0094$ (0.94%)
  - mem usage: $10^6$ bits $\approx 122$ kbyte (10 bits/id)
  - assume id has 10 bytes: $10^7$ bytes $\approx 9.5$ MB

conclusions:

- more memory improves probability, lesser elements ($n$), too
- more hash functions **not necessarily**!
- e.g. (values from above) $k = 7$ prob$\approx 0.82\%$, $k = 10$ prob$\approx 1\%$
- each $m, n$ pair has optimal $k = \frac{n}{m} \ln(2)$
wrap up: sampling and filtering

- straightforward sampling can go wrong
- correct sample by fixing selection
  e.g. limit to certain users/ids
- decide if in sample or not with hash function
  - does that solve our “count duplicates” problem?
- filter large set of objects with Bloom filter
insertion: difference between streaming and online algorithms:

• online scenario:
  – unlimited stream
  – one pass
  – answers ready all the time (e.g. sliding window)

• streaming scenario
  – several passes possible, usually avoided
  – answers usually at end of execution