

COMPSCI 514: ALGORITHMS FOR DATA SCIENCE

Cameron Musco

University of Massachusetts Amherst. Fall 2019.

Lecture 5

- Problem Set 1 was posted on Friday. Due next Thursday 9/26 in Gradescope, before class.
- Don't leave until the last minute.

Last Class We Covered:

- **Bloom Filters:**
 - Random hashing to maintain a large sets in very small space.
 - Discussed applications and how the false positive rate is determined.
- **Streaming Algorithms and Distinct Elements:**
 - Started on streaming algorithms and one of the most fundamental examples: estimating the number of **distinct items** in a data stream.
 - Introduced an algorithm for doing this via a min-of-hashes approach.

Finish Distinct Elements:

- Finish analyzing distinct elements algorithm. Learn the ‘median trick’.
- Discuss variants and practical implementations.

MinHashing For Set Similarity:

- See how a min-of-hashes approach (MinHash) is used to estimate the overlap between two bit vectors.
- A key idea behind audio fingerprint search (Shazam), document search (plagiarism and copyright violation detection), recommendation systems, etc.

First an observation about Bloom filters:

$$\text{False Positive Rate: } \delta \approx \left(1 - e^{-\frac{kn}{m}}\right)^k.$$

For an m -bit bloom filter holding n items, optimal number of hash functions k is: $k = \ln 2 \cdot \frac{m}{n}$.

If we want a false positive rate $< \frac{1}{2}$ how big does m need to be in comparison to n ?

$$m = O(\log n), \quad m = O(\sqrt{n}), \quad m = O(n), \quad m = O(n^2)?$$

If $m = \frac{n}{\ln 2}$, optimal $k = 1$, and failure rate is:

$$\delta = \left(1 - e^{-\frac{n/\ln 2}{n}}\right)^1 = \left(1 - \frac{1}{2}\right)^1 = \frac{1}{2}.$$

I.e., storing n items in a bloom filter requires $O(n)$ space. So what's the point? Truly $O(n)$ bits, rather than $O(n \cdot \text{item size})$.

HASHING FOR DISTINCT ELEMENTS

Distinct Elements (Count-Distinct) Problem: Given a stream x_1, \dots, x_n , estimate the number of distinct elements.

Hashing for Distinct Elements:

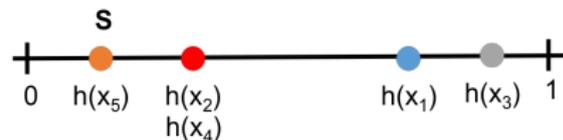
- Let $h : U \rightarrow [0, 1]$ be a random hash function (continuous output).

- $s := 1$

- For $i = 1, \dots, n$

- $s := \min(s, h(x_i))$

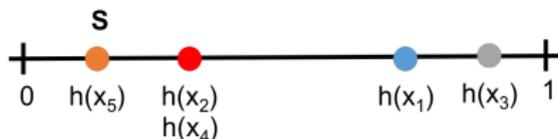
- Return $\hat{d} = \frac{1}{s} - 1$



- After all items are processed, s is the minimum of d points chosen uniformly at random on $[0, 1]$. Where $d = \#$ distinct elements.
- **Intuition:** The larger d is, the smaller we expect s to be.
- **Notice:** Output does not depend on n at all.

PERFORMANCE IN EXPECTATION

s is the minimum of d points chosen uniformly at random on $[0, 1]$.
Where $d = \#$ distinct elements.



$$\mathbb{E}[s] = \frac{1}{d+1} \text{ (using } \mathbb{E}(s) = \int_0^\infty \Pr(s > x)dx \text{ + calculus)}$$

- So estimate of $\hat{d} = \frac{1}{s} - 1$ output by the algorithm is correct if s exactly equals its expectation. Does this mean $\mathbb{E}[\hat{d}] = d$? No, but:
- **Approximation is robust:** if $|s - \mathbb{E}[s]| \leq \epsilon \cdot \mathbb{E}[s]$ for any $\epsilon \in (0, 1/2)$:

$$(1 - 2\epsilon)d \leq \hat{d} \leq (1 + 4\epsilon)d$$

So question is how well \mathbf{s} concentrates around its mean.

$$\mathbb{E}[\mathbf{s}] = \frac{1}{d+1} \text{ and } \text{Var}[\mathbf{s}] \leq \frac{1}{(d+1)^2} \text{ (could compute via calculus).}$$

Chebyshev's Inequality:

$$\Pr [|\mathbf{s} - \mathbb{E}[\mathbf{s}]| \geq \epsilon \mathbb{E}[\mathbf{s}]] \leq \frac{\text{Var}[\mathbf{s}]}{(\epsilon \mathbb{E}[\mathbf{s}])^2} = \frac{1}{\epsilon^2}.$$

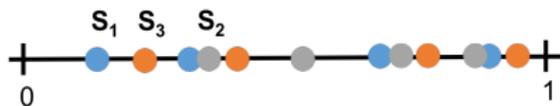
Bound is vacuous for any $\epsilon < 1$. **How can we improve accuracy?**

\mathbf{s} : minimum of d distinct hashes chosen randomly over $[0, 1]$, computed by hashing algorithm. $\hat{\mathbf{d}} = \frac{1}{\mathbf{s}} - 1$: estimate of # distinct elements d .

Leverage the law of large numbers: improve accuracy via repeated independent trials.

Hashing for Distinct Elements (Improved):

- Let $h_1, h_2, \dots, h_k : U \rightarrow [0, 1]$ be random hash functions
- $s_1, s_2, \dots, s_k := 1$
- For $i = 1, \dots, n$
 - For $j=1, \dots, k$, $s_j := \min(s_j, h_j(x_i))$
- $s := \frac{1}{k} \sum_{j=1}^k s_j$
- Return $\hat{d} = \frac{1}{s} - 1$



$\mathbf{s} = \frac{1}{k} \sum_{j=1}^k \mathbf{s}_j$. Have already shown that for $j = 1, \dots, k$:

$$\mathbb{E}[\mathbf{s}_j] = \frac{1}{d+1} \implies \mathbb{E}[\mathbf{s}] = \frac{1}{d+1} \text{ (linearity of expectation)}$$

$$\text{Var}[\mathbf{s}_j] \leq \frac{1}{(d+1)^2} \implies \text{Var}[\mathbf{s}] \leq \frac{1}{k \cdot (d+1)^2} \text{ (linearity of variance)}$$

Chebyshev Inequality:

$$\Pr \left[\left| d - \widehat{\mathbf{d}} \right| \geq 4\epsilon \cdot d \right] \leq \frac{\text{Var}[\mathbf{s}]}{(\epsilon \mathbb{E}[\mathbf{s}])^2} = \frac{\mathbb{E}[\mathbf{s}]^2/k}{\epsilon^2 \mathbb{E}[\mathbf{s}]^2} = \frac{1}{k \cdot \epsilon^2} = \frac{\epsilon^2 \cdot \delta}{\epsilon^2} = \delta.$$

How should we set k if we want $4\epsilon \cdot d$ error with probability $\geq 1 - \delta$?

$$k = \frac{1}{\epsilon^2 \cdot \delta}.$$

\mathbf{s}_j : minimum of d distinct hashes chosen randomly over $[0, 1]$. $\mathbf{s} = \frac{1}{k} \sum_{j=1}^k \mathbf{s}_j$.
 $\widehat{\mathbf{d}} = \frac{1}{\mathbf{s}} - 1$: estimate of # distinct elements d .

Hashing for Distinct Elements:

- Let $h_1, h_2, \dots, h_k : U \rightarrow [0, 1]$ be random hash functions
- $s_1, s_2, \dots, s_k := 1$
- For $i = 1, \dots, n$
 - For $j=1, \dots, k$, $s_j := \min(s_j, h_j(x_i))$
- $s := \frac{1}{k} \sum_{j=1}^k s_j$
- Return $\hat{d} = \frac{1}{s} - 1$



- Setting $k = \frac{1}{\epsilon^2 \cdot \delta}$, algorithm returns \hat{d} with $|d - \hat{d}| \leq 4\epsilon \cdot d$ with probability at least $1 - \delta$.
- Space complexity is $k = \frac{1}{\epsilon^2 \cdot \delta}$ real numbers s_1, \dots, s_k .
- $\delta = 5\%$ failure rate gives a factor 20 overhead in space complexity.

How can we decrease the cost of a small failure rate δ ?

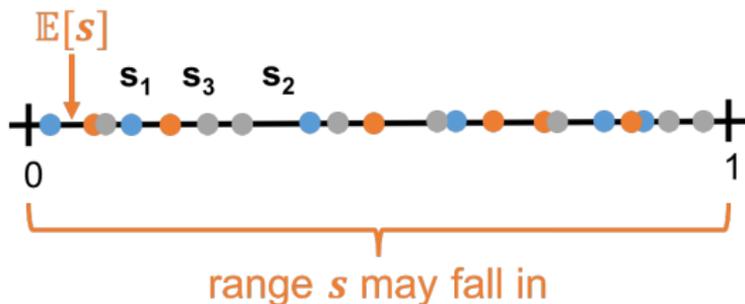
One Thought: Apply stronger concentration bounds. E.g., replace Chebyshev with Bernstein. This won't work. *Why?*

Bernstein Inequality (applied to mean): Consider independent random variables X_1, \dots, X_k all falling in $[-M, M]$ and let $X = \frac{1}{k} \sum_{i=1}^k X_i$. Let $\mu = \mathbb{E}[X]$ and $\sigma^2 = \text{Var}[X]$. For any $t \geq 0$:

$$\Pr(|X - \mu| \geq t) \leq 2 \exp\left(-\frac{t^2}{2\sigma^2 + \frac{4Mt}{3k}}\right).$$

For us, $t^2 = O\left(\frac{\epsilon}{d^2}\right)$ and $\frac{4Mt}{3k} = O\left(\frac{\epsilon}{dk}\right)$ so if $k \ll d$ exponent has small magnitude (i.e., bound is bad).

Exponential tail bounds are weak for random variables with very large ranges compared to their expectation.

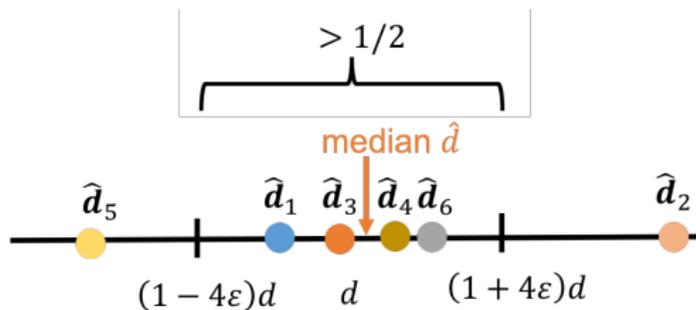


IMPROVED FAILURE RATE

How can we improve our dependence on the failure rate δ ?

The median trick: Run $t = O(\log 1/\delta)$ trials each with failure probability $\delta' = 1/5$ – each using $k = \frac{1}{\delta'\epsilon^2} = \frac{5}{\epsilon^2}$ hash functions.

- Letting $\hat{d}_1, \dots, \hat{d}_t$ be the outcomes of the t trials, return $\hat{d} = \text{median}(\hat{d}_1, \dots, \hat{d}_t)$.



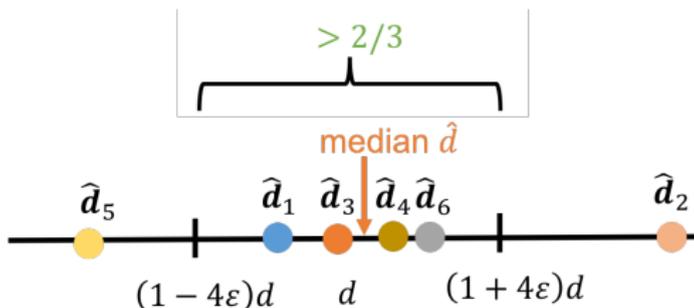
- If $> 1/2$ of trials fall in $[(1 - 4\epsilon)d, (1 + 4\epsilon)d]$, then the median will.
- Have $< 1/2$ of trials on both the left and right.

IMPROVED FAILURE RATE

How can we improve our dependence on the failure rate δ ?

The median trick: Run $t = O(\log 1/\delta)$ trials each with failure probability $\delta' = 1/5$ – each using $k = \frac{1}{\delta'^2} = \frac{5}{\epsilon^2}$ hash functions.

- Letting $\hat{d}_1, \dots, \hat{d}_t$ be the outcomes of the t trials, return $\hat{d} = \text{median}(\hat{d}_1, \dots, \hat{d}_t)$.



- If $> 2/3$ of trials fall in $[(1 - 4\epsilon)d, (1 + 4\epsilon)d]$, then the median will.
- Have $< 1/3$ of trials on both the left and right.

THE MEDIAN TRICK

- $\hat{\mathbf{d}}_1, \dots, \hat{\mathbf{d}}_t$ are the outcomes of the t trials, each falling in $[(1 - 4\epsilon)d, (1 + 4\epsilon)d]$ with probability at least $4/5$.
- $\hat{\mathbf{d}} = \text{median}(\hat{\mathbf{d}}_1, \dots, \hat{\mathbf{d}}_t)$.

What is the probability that the median $\hat{\mathbf{d}}$ falls in $[(1 - 4\epsilon)d, (1 + 4\epsilon)d]$?

- Let \mathbf{X} be the # of trials falling in $[(1 - 4\epsilon)d, (1 + 4\epsilon)d]$. $\mathbb{E}[\mathbf{X}] = \frac{4}{5} \cdot t$.

$$\Pr(\hat{\mathbf{d}} \notin [(1 - 4\epsilon)d, (1 + 4\epsilon)d]) \leq \Pr\left(\mathbf{X} < \frac{5}{6} \cdot \mathbb{E}[\mathbf{X}]\right) \leq \Pr\left(|\mathbf{X} - \mathbb{E}[\mathbf{X}]| \geq \frac{1}{6} \mathbb{E}[\mathbf{X}]\right)$$

Apply Chernoff bound:

$$\Pr\left(|\mathbf{X} - \mathbb{E}[\mathbf{X}]| \geq \frac{1}{6} \mathbb{E}[\mathbf{X}]\right) \leq 2 \exp\left(-\frac{\frac{1}{6} \cdot \frac{4}{5} t}{2 + 1/6}\right) = O\left(e^{-O(t)}\right).$$

- Setting $t = O(\log(1/\delta))$ gives failure probability $e^{-\log(1/\delta)} = \delta$.

Upshot: The median of $t = O(\log(1/\delta))$ independent runs of the hashing algorithm for distinct elements returns $\hat{d} \in [(1 - 4\epsilon)d, (1 + 4\epsilon)d]$ with probability at least $1 - \delta$.

Total Space Complexity: t trials, each using $k = \frac{1}{\epsilon^2 \delta'}$ hash functions, for $\delta' = 1/5$. Space is $\frac{5t}{\epsilon^2} = O\left(\frac{\log(1/\delta)}{\epsilon^2}\right)$ real numbers (the minimum value of each hash function).

No dependence on the number of distinct elements d or the number of items in the stream n ! Both of these numbers are typically very large.

A note on the median: The median is often used as a robust alternative to the mean, when there are outliers (e.g., heavy tailed distributions, corrupted data).

Our algorithm uses continuous valued fully random hash functions. Can't be implemented...

- The idea of using the minimum hash value of x_1, \dots, x_n to estimate the number of distinct elements naturally extends to when the hash functions map to discrete values.
- Flajolet-Martin (LogLog) algorithm and [HyperLogLog](#).

$h(x_1)$	1010010
$h(x_2)$	1001100
$h(x_3)$	1001110
	⋮
$h(x_n)$	1011000

Estimate # distinct elements based on maximum number of trailing zeros m .

The more distinct hashes we see, the higher we expect this maximum to be.

Flajolet-Martin (LogLog) algorithm and HyperLogLog.

$h(x_1)$	1010010
$h(x_2)$	1001100
$h(x_3)$	1001110
⋮	
$h(x_n)$	1011000

Estimate # distinct elements based on maximum number of trailing zeros m .

With d distinct elements what do we expect m to be?

$$\Pr(h(x_i) \text{ has } \log d \text{ trailing zeros}) = \frac{1}{2^{\log d}} = \frac{1}{d}.$$

So with d distinct hashes, expect to see 1 with $\log d$ trailing zeros. Expect $m \approx \log d$. m takes $\log \log d$ bits to store.

Total Space: $O\left(\frac{\log \log d}{\epsilon^2} + \log d\right)$ for an ϵ approximate count.

Note: Careful averaging of estimates from multiple hash functions.

Using HyperLogLog to count 1 billion distinct items with 2% accuracy:

$$\begin{aligned}
 \text{space used} &= O\left(\frac{\log \log d}{\epsilon^2} + \log d\right) \\
 &= \frac{1.04 \cdot \lceil \log_2 \log_2 d \rceil}{\epsilon^2} + \lceil \log_2 d \rceil \text{ bits} \\
 &= \frac{1.04 \cdot 5}{.02^2} + 30 = 13030 \text{ bits} \approx 1.6 \text{ kB!}
 \end{aligned}$$

Mergeable Sketch: Consider the case (essentially always in practice) that the items are processed on different machines.

- Given data structures (sketches) $HLL(x_1, \dots, x_n)$, $HLL(y_1, \dots, y_n)$ is easy to merge them to give $HLL(x_1, \dots, x_n, y_1, \dots, y_n)$. **How?**
- Set the maximum # of trailing zeros to the maximum in the two sketches.

Implementations: Google PowerDrill, Facebook Presto, Twitter Algebird, Amazon Redshift.

Use Case: Exploratory SQL-like queries on tables with 100s billions of rows. ~ 5 million count distinct queries per day. E.g.,

- **Count** number of **distinct** users in Germany that made at least one search containing the word 'auto' in the last month.
- **Count** number of **distinct** subject lines in emails sent by users that have registered in the last week, in comparison to number of emails sent overall (to estimate rates of spam accounts).

Traditional *COUNT*, *DISTINCT* SQL calls are far too slow, especially when the data is distributed across many servers.

Estimate number of search ‘sessions’ that happened in the last month (i.e., a single user making possibly many searches at one time, likely surrounding a specific topic.)

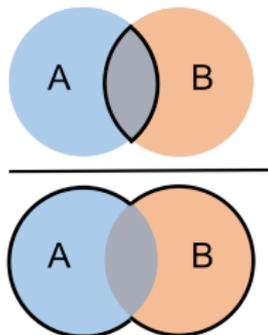
IP Address	Time Stamp	Query
127.65.48.31	10:10:26	“bloom filter”
62.54.16.001	10:10:28	“united airlines”
16.578.32.12	10:13:34	“china news”
192.68.001.1	10:14:05	“bmw”
174.15.254.1	10:14:54	“khalid tour”
127.65.48.31	10:15:45	“hashing”
192.68.001.1	10:16:18	“car loans”
⋮	⋮	⋮

- Count distinct keys where key is $(IP, Hr, Min \bmod 10)$.
- Using HyperLogLog, cost is roughly that of a (distributed) linear scan (to stream through all items in table).

Questions on distinct elements counting?

Jaccard Index: A similarity measure between two sets.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{\# \text{ shared elements}}{\# \text{ total elements}}.$$



Natural measure for similarity between bit strings – interpret an n bit string as a set, containing the elements corresponding the positions of its ones. $J(x, y) = \frac{\# \text{ shared ones}}{\text{total ones}}$.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{\# \text{ shared elements}}{\# \text{ total elements}}.$$

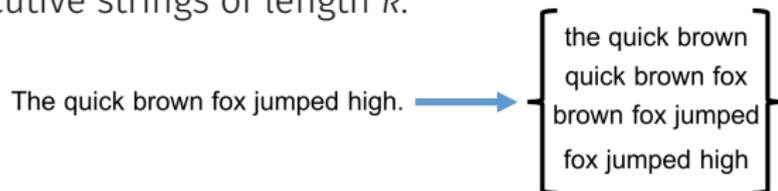
- Computing exactly requires roughly linear time in $|A| + |B|$ (using a hash table or binary search). Not bad.
- **Near Neighbor Search:** Have a database of n sets/bit strings and given a set A , want to find if it has high similarity to anything in the database. $O(n \cdot \text{average set size})$ time.
- **All-pairs Similarity Search:** Have n different sets/bit strings and want to find all pairs with high similarity. $O(n^2 \cdot \text{average set size})$ time.

Prohibitively expensive when n is very large. We'll see how to significantly improve on these runtimes with random hashing.

How should you measure similarity between two-documents?

E.g., to detect plagiarism and copyright infringement, to see if an email message is similar to previously seen spam, to detect duplicate webpages in search results, etc.

- If the documents are not identical, doing a word-by-word comparison typically gives nothing. Can compute edit distance, but this is very expensive if you are comparing many documents.
- **Shingling + Jaccard Similarity:** Represent a document as the set of all consecutive strings of length k .



- Measure similarity as Jaccard similarity between shingle sets.
- Also used to measure word similarity. E.g., in spell checkers.

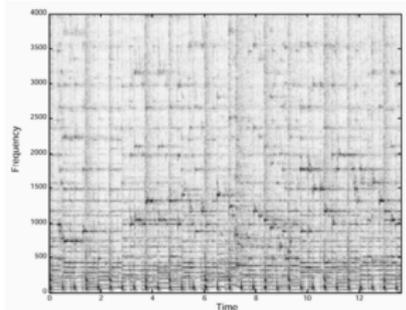
APPLICATION: AUDIO FINGERPRINTING

How should you measure similarity between two audio clips?

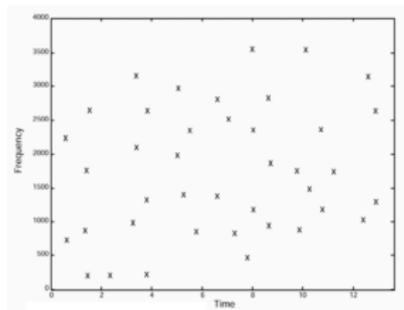
E.g. in audio search engines like Shazam, for detecting copyright infringement, for search in sound effect libraries, etc.

Audio Fingerprinting + Jaccard Similarity:

Step 1: Compute the spectrogram: representation of frequency intensity over time.



Step 2: Threshold the spectrogram to a binary matrix representing the sound clip.



Compare thresholded spectrograms with Jaccard similarity.

Questions?