COMPSCI 514: ALGORITHMS FOR DATA SCIENCE

Cameron Musco

University of Massachusetts Amherst. Fall 2020.

Lecture 7

LOGISTICS

- Solutions for Problem Set 1 have been posted.
- Problem Set 2 will be released in the next day or two.
- The grading for ’select all that apply’ questions on quizzes has been broken – will be fixed going forward.
- Quiz 3 Feedback:
  - People have mixed feelings on breakout rooms. Many think they are too small/too short.
  - A number of people suggested polls during class and summaries of the material at the end of class. I’ll try to implement these.

SUMMARY

Last Class:

- Wrap up Bloom Filters: how to set $k = \#$ hash functions to minimize false positive rate.
- Space usage of $O(n)$ bits vs. $O(n \cdot \text{item size})$ for hash tables.
- Start on streaming algorithms: the distinct items problem.
- Estimating distinct item count via MinHashing.

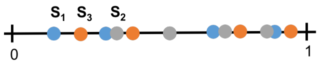
This Class:

- Finish up distinct items: median trick to boost success probability. Distinct items in practice.
- Application of MinHash to estimating the Jaccard similarity.
- Start on fast similarity search and locality sensitive hashing.

LAST TIME: MINHASHING

Hashing for Distinct Elements:

- Let $\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_k : U \rightarrow [0, 1]$ be random hash functions
- $\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_k := 1$
- For $i = 1, \ldots, n$
    - For j=1,..., k, $\mathbf{s}_j := \min(\mathbf{s}_j, \mathbf{h}_j(x_i))$
- $\mathbf{s} := \frac{1}{k} \sum_{j=1}^{k} \mathbf{s}_j$
- Return $\widehat{\mathbf{d}} = \frac{1}{\mathbf{s}} - 1$



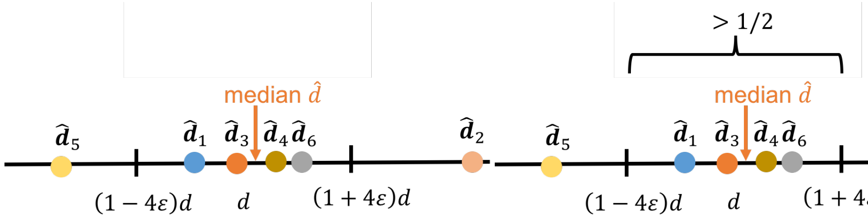
- Setting $k = \frac{1}{\epsilon^2 \cdot \delta}$, algorithm returns $\widehat{\mathbf{d}}$ with $|d - \widehat{\mathbf{d}}| \leq 4\epsilon \cdot d$ with probability at least $1 - \delta$.
- Space complexity is $k = \frac{1}{\epsilon^2 \cdot \delta}$ real numbers $s_1, \ldots, s_k$.
- $\delta = 5\%$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

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 $\widehat{\mathrm{d}}_1, \ldots, \widehat{\mathrm{d}}_t$ be the outcomes of the t trials, return $\widehat{\mathrm{d}} = median(\widehat{\mathrm{d}}_1, \ldots, \widehat{\mathrm{d}}_t)$.



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

THE MEDIAN TRICK

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

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

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

$$\Pr\left(\widehat{\mathbf{d}} \notin [(1-4\epsilon)d, (1+4\epsilon)d]\right) \leq \Pr\left(\mathbf{X} < \frac{2}{3} \cdot t \frac{5}{6} \cdot \mathbb{E}[\mathbf{X}]\right) \leq \Pr\left(|\mathbf{X} - \mathbb{E}[\mathbf{X}]| \geq \frac{1}{6}\mathbb{E}[\mathbf{X}]\right.$$

Apply Chernoff bound:

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

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

MEDIAN TRICK

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

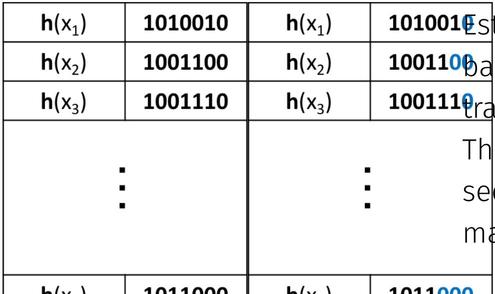
No dependence on the number of distinct elements d or the number of items in the stream n! Both of these numbers are typically very large.

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).

DISTINCT ELEMENTS IN PRACTICE

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

- The idea of using the minimum hash value of $x_1, \ldots, x_n$ 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_1)	**1010010**
h(x_2)	**1001100**	**h**(x_2)	**1001100**
h(x_3)	**1001110**	**h**(x_3)	**1001110**
⋮		⋮	
h(x_n)	**1011000**	**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.

LOGLOG COUNTING OF DISTINCT ELEMENTS

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, roughly what do we expect **m** to be?

a) $O(1)$ b) $O(\log d)$ c) $O(\sqrt{d})$ d) $O(d)$

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

So with d distinct hashes, expect to see 1 with $\log d$ trailing zeros.

Expect $\mathbf{m} \approx \log d$. **m** takes $\log\log d$ bits to store.

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

LOGLOG SPACE GUARANTEES

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

$$\begin{aligned}\text{space used } &= O\left(\frac{\log\log d}{\epsilon^2} + \log d\right)\\ &= \frac{1.04 \cdot \lceil \log_2 \log_2 d \rceil}{\epsilon^2} + \lceil \log_2 d \rceil \text{ bits}^1\\ &= \frac{1.04 \cdot 5}{.02^2} + 30 = 13030 \text{ bits} \approx 1.6\ kB!\end{aligned}$$

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

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

1. 1.04 is the constant in the HyperLogLog analysis. Not important!

HYPERLOGLOG IN PRACTICE

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

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

- Count number if 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.

IN PRACTICE

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

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

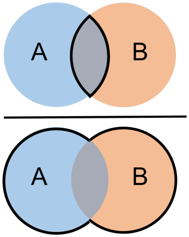
Questions on distinct elements counting?

Summary:

ANOTHER FUNDAMENTAL PROBLEM

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. $J(x, y) = \frac{\#\ \text{shared ones}}{\text{total ones}}$.

What other measures might you consider?

SEARCH WITH JACCARD SIMILARITY

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

Want Fast Implementations For:

- **Near Neighbor Search:** Have a database of n sets/bit strings and given a set A, want to find if it has high Jaccard similarity to anything in the database. $\Omega(n)$ time with a linear scan.
- **All-pairs Similarity Search:** Have n different sets/bit strings and want to find all pairs with high Jaccard similarity. $\Omega(n^2)$ time if we check all pairs explicitly.

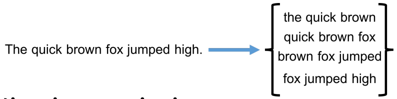
Will speed up via randomized locality sensitive hashing.

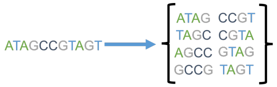
What approaches might you use here to speed up search?

APPLICATIONS

Document Similarity:

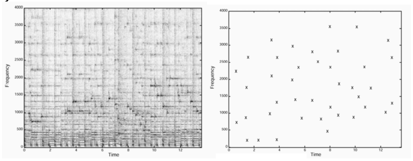
- E.g., to detect plagiarism, copyright infringement, duplicate webpages, spam.
- Use Shingling + Jaccard similarity. (n-grams, k-mers)





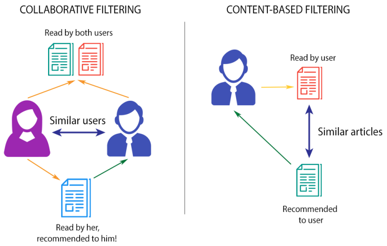
Audio Fingerprinting:

- E.g., in audio search (Shazam), Earthquake detection.
- Represent sound clip via a binary 'fingerprint' then compare with Jaccard similarity.



APPLICATION: COLLABORATIVE FILTERING

Online recommendation systems are often based on **collaborative filtering**. Simplest approach: find similar users and make recommendations based on those users.

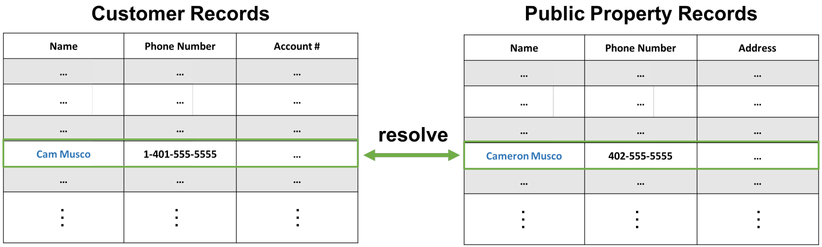


- Twitter: represent a user as the set of accounts they follow. Match similar users based on the Jaccard similarity of these sets. Recommend that you follow accounts followed by similar users.
- Netflix: look at sets of movies watched. Amazon: look at products purchased, etc.

APPLICATION: ENTITY RESOLUTION

Entity Resolution Problem: Want to combine records from multiple data sources that refer to the same entities.

- E.g. data on individuals from voting registrations, property records, and social media accounts. Names and addresses may not exactly match, due to typos, nicknames, moves, etc.
- Still want to match records that all refer to the same person using all pairs similarity search.



Customer Records

Name	Phone Number	Account #
...	...	...
...	...	...
...	...	...
Cam Musco	1-401-555-5555	...
...	...	...
⋮	⋮	⋮

Public Property Records

Name	Phone Number	Address
...	...	...
...	...	...
...	...	...
Cameron Musco	402-555-5555	...
...	...	...
⋮	⋮	⋮

APPLICATION: SPAM AND FRAUD DETECTION

Many applications to spam/fraud detection. E.g.

- **Fake Reviews**: Very common on websites like Amazon. Detection often looks for (near) duplicate reviews on similar products, which have been copied. 'Near duplicate' measured with shingles + Jaccard similarity.
- **Lateral phishing**: Phishing emails sent to addresses at a business coming from a legitimate email address at the same business that has been compromised.
    - One method of detection looks at the recipient list of an email and checks if it has small Jaccard similarity with any previous recipient lists. If not, the email is flagged as possible spam.

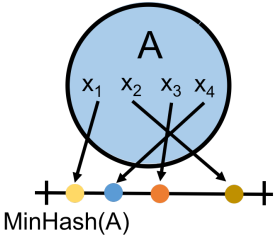
MINHASHING

Goal: Speed up Jaccard similarity search (near neighbor and all-pairs similarity search).

Strategy: Use random hashing to map each set to a very compressed representation. Jaccard similarity can be estimated from these representations.

MinHash(A): [Andrei Broder, 1997 at Altavista]

- Let $\mathbf{h} : U \to [0, 1]$ be a random hash function
- $\mathbf{s} := 1$
- For $x_1, \ldots, x_{|A|} \in A$
  - $\mathbf{s} := \min(\mathbf{s}, \mathbf{h}(x_k))$
- Return $\mathbf{s}$

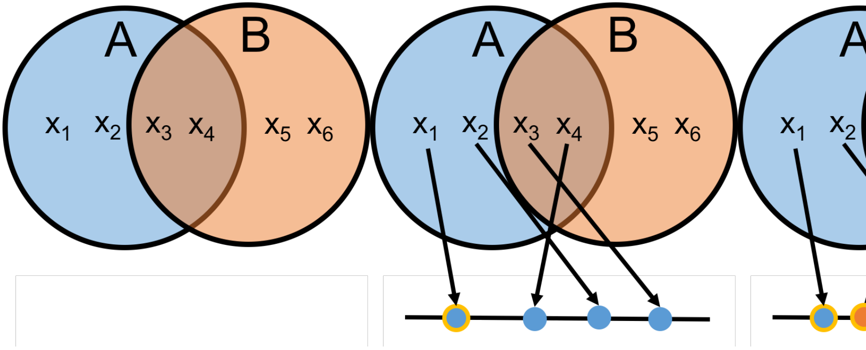


Identical to our distinct elements sketch!

MINHASH

For two sets A and B, what is $\Pr(MinHash(A) = MinHash(B))$?

- Since we are hashing into the continuous range $[0, 1]$, we will never have $\mathbf{h}(x) = \mathbf{h}(y)$ for $x \neq y$ (i.e., no spurious collisions)



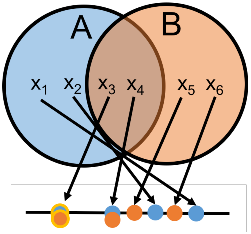
A
B
x_1 x_2 x_3 x_4 x_5 x_6
A
B
x_1 x_2 x_3 x_4 x_5 x_6
A
x_1 x_2

- $MinHash(A) = MinHash(B)$ only if an item in $A \cap B$ has the minimum hash value in both sets.

MINHASH

For two sets A and B, what is $\Pr(MinHash(A) = MinHash(B))$?

Claim: $MinHash(A) = MinHash(B)$ only if an item in $A \cap B$ has the minimum hash value in both sets.



$$\Pr(MinHash(A) = MinHash(B)) = ?\ \frac{|A \cap B|}{\text{total \# items hashed}}$$
$$= \frac{|A \cap B|}{|A \cup B|} = J(A, B).$$

Questions?