COMPSCI 514: ALGORITHMS FOR DATA SCIENCE

Cameron Musco

University of Massachusetts Amherst. Fall 2021.

Lecture 8

Last Class:

$h(1), h(2), h(5), h(6), h(3) \sim [0,1]$

- MinHashing for distinct elements
- The median trick for boosting success probability.

1 2 1 1 5 6 3 1

$X = \min\{h(1), h(2), h(1) \ldots h(3), h(1)\}$

$\mathbb{E}X = \frac{1}{d+1} = \frac{1}{6}$ $\quad \frac{1}{3+1} = \frac{1}{4}$

Last Class:

- MinHashing for distinct elements
- The median trick for boosting success probability.

This Class:

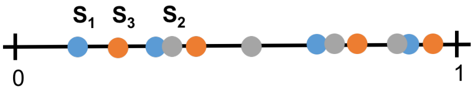
- Finish up the distinct elements problem by sketching the ideas behind practical algorithms similar to MinHashing
- Start on fast similarity search. MinHashing to estimate the Jaccard similarity between two sets.

Hashing for Distinct Elements:

- Let $\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_k : U \to [0, 1]$ 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 $\widehat{\mathbf{d}} = \frac{1}{\mathbf{s}} - 1$

$\mathbb{E} s_j = \frac{1}{d+1}$ $Var(s_j) = \frac{1}{(d+1)^2}$

$\mathbb{E} s = \frac{1}{d+1}$ $Var(s) = \frac{1}{k} \cdot \frac{1}{(d+1)^2}$

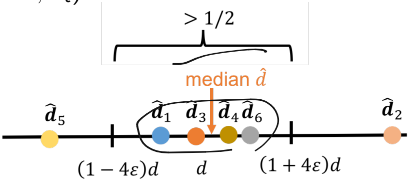


- If $k = \frac{1}{\epsilon^2 \cdot \delta}$, returns $\widehat{\mathbf{d}}$ with $|d - \widehat{\mathbf{d}}| \leq 4\epsilon \cdot d$ with probability at least $1 - \delta$ (analysis via linearity of expectation + linearity of variance + Chebyshev's inequality.)

Want to improve dependence on the failure rate δ from $1/\delta$ to $\log(1/\delta)$.

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 $\widehat{\mathrm{d}} = median(\widehat{\mathrm{d}}_1, \ldots, \widehat{\mathrm{d}}_t)$.



- We expect $\geq 4/5$ for the trials to fall in $[(1-4\epsilon)d, (1+4\epsilon)d]$.
- If $> 1/2$ fall in this range, then the median will. We can show this will occur with high probability via a Chernoff bound.

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 $[(1-4\epsilon)d, (1+4\epsilon)d]$. $\mathbb{E}[\mathbf{X}] = \frac{4}{5} \cdot t$.

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

Apply Chernoff bound:

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

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

MEDIAN TRICK

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

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

DISTINCT ELEMENTS IN PRACTICE

Our algorithm uses continuous valued fully random hash functions.

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

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.

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.

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_2)	**1001100**
h(x_3)	**1001110**
⋮	
h(x_n)	**1011000**

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_2)	**1001100**
h(x_3)	**1001110**
⋮	
h(x_n)	**1011000**

3

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

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_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**.

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?

$m = \log_2 d$

$\frac{1}{2^m} \approx \frac{1}{d}$

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

$\Pr(h(x_i) \text{ has } m \text{ trailing } 0\text{s}) = \frac{1}{2^m}$

hash d different items

$\Pr(\max \# \text{ trailing } 0\text{s} = m) \approx \frac{d}{2^m} \approx \frac{1}{2}$

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?

$$\Pr(\mathbf{h}(x_i) \text{ has } x \text{ trailing zeros}) =$$

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?

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

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?

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

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?

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

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?

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

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

Expect $\mathbf{m} \approx \log d$. $\sim d \approx 2^m$

$E s = \frac{1}{d+1}$ $\quad d = \frac{1}{s} - 1$

LOGLOG COUNTING OF DISTINCT ELEMENTS

Flajolet-Martin (LogLog) algorithm and HyperLogLog.

$\mathbf{h}(x_1)$	**1010010**
$\mathbf{h}(x_2)$	**1001100**
$\mathbf{h}(x_3)$	**1001110**
⋮	
$\mathbf{h}(x_n)$	**1011000**

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

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

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

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.

$\log_2 \log_2 d$

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**.

$\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}$ $\frac{1}{2^m}$

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

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

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

Expect $\mathbf{m} \approx \log d$. **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.

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?

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

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

Expect $\mathbf{m} \approx \log d$. **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.

LOGLOG SPACE GUARANTEES

$d \approx$ $\epsilon = .02$

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

$$\text{space used } = O\left(\frac{\log\log d}{\epsilon^2} + \log d\right)$$

LOGLOG SPACE GUARANTEES

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

$$\text{space used} = O\left(\frac{\log\log d}{\epsilon^2} + \log d\right)$$
$$= \frac{1.04 \cdot \lceil \log_2 \log_2 d \rceil}{\epsilon^2} + \lceil \log_2 d \rceil \text{ bits}^1$$

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

LOGLOG SPACE GUARANTEES

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

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

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

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.

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

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 $HLL(x_1, \ldots, x_n, y_1, \ldots, y_n)$.

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

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?

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

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!

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

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

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.

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 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).

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).

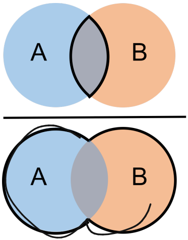
Traditional *COUNT*, *DISTINCT* SQL calls are far too slow, especially when the data is distributed across many servers.

Questions on distinct elements counting?

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

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

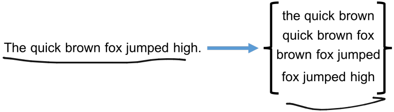
Want Fast Implementations For:

- **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.

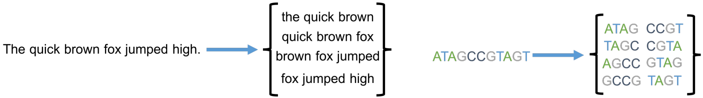
Document Similarity:

- E.g., to detect plagiarism, copyright infringement, duplicate webpages, spam.
- Use Shingling + Jaccard similarity.



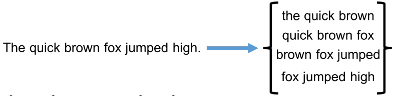
Document Similarity:

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



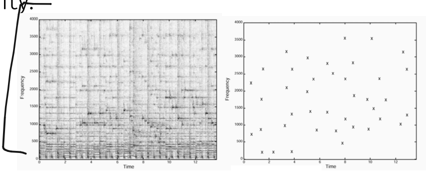
Document Similarity:

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

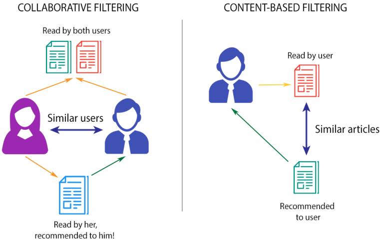


Audio Fingerprinting:

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

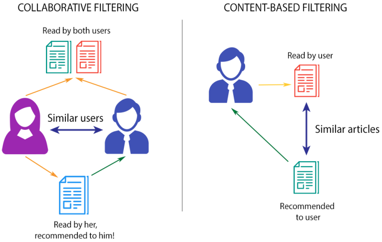


Online recommendation systems are often based on **collaborative filtering**. Simplest approach: find similar users and make recommendations based on those users.



APPLICATION: COLLABORATIVE FILTERING

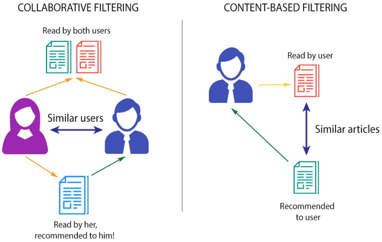
Online recommendation systems are often based on **collaborative filtering**. Simplest approach: find similar users and make recommendations based on those users.



- 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.

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.

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

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	...
...	...	...
⋮	⋮	⋮

resolve

Public Property Records

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

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.

See Section 3.8.2 of *Mining Massive Datasets* for a discussion of a real world example involving 1 million customers. Naively this would be $\binom{1000000}{2} \approx 500$ billion pairs of customers to check!

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

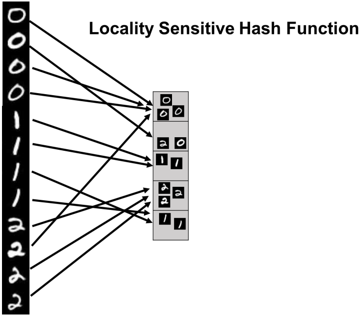
Many applications to spam/fraud detection. E.g.

- 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.

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.

How does locality sensitive hashing (LSH) help with similarity search?



- **Near Neighbor Search:** Given item x, compute $\mathbf{h}(x)$. Only search for similar items in the $\mathbf{h}(x)$ bucket of the hash table.
- **All-pairs Similarity Search:** Scan through all buckets of the hash table and look for similar pairs within each bucket.
- We will use $\mathbf{h}(x) = \mathbf{g}(MinHash(x))$ where $\mathbf{g} : [0,1] \to [n]$ is a random hash function. Why?