COMPSCI 514: ALGORITHMS FOR DATA SCIENCE

Cameron Musco

University of Massachusetts Amherst. Spring 2020.

Lecture 7

- Problem Set 1 is due tomorrow at 8pm in Gradescope.
- No class next Tuesday (it's a Monday at UMass).
- **Talk Today:** Vatsal Sharan at 4pm in CS 151. *Modern Perspectives on Classical Learning Problems: Role of Memory and Data Amplification.*

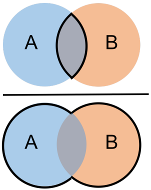
Last Class: Hashing for Jaccard Similarity

- MinHash for estimating the Jaccard similarity.
- Locality sensitive hashing (LSH).
- Application to fast similarity search.

This Class:

- Finish up MinHash and LSH.
- The Frequent Elements (heavy-hitters) problem.
- Misra-Gries summaries.

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



Two Common Use Cases:

- **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. Naively $\Omega(n)$ time.
- **All-pairs Similarity Search:** Have n different sets/bit strings. Want to find all pairs with high similarity. Naively $\Omega(n^2)$ time.

MINHASHING

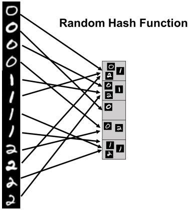
$MinHash(A) = \min_{a \in A} \mathbf{h}(a)$ where $\mathbf{h} : U \to [0,1]$ is a random hash.

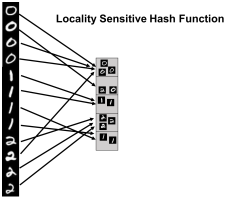
Locality Sensitivity: $\Pr[MinHash(A) = MinHash(B)] = J(A,B)$.

Represents a set with a single number that captures Jaccard similarity information!

Given a collision free hash function $\mathbf{g} : [0,1] \to [m]$,

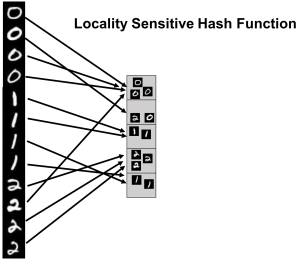
$$\Pr\left[\mathbf{g}(MinHash(A)) = \mathbf{g}(MinHash(B))\right] = J(A,B).$$





What is $\Pr\left[\mathbf{g}(MinHash(A)) = \mathbf{g}(MinHash(B))\right]$ if $\mathbf{g}$ is not collision free?
Will be a bit larger than $J(A,B)$.

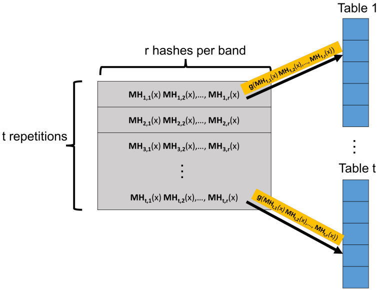
When searching for similar items only search for matches that land in the same hash bucket.



- **False Negative:** A similar pair doesn't appear in the same bucket.
- **False Positive:** A dissimilar pair is hashed to the same bucket.

Need to balance a small probability of false negatives (a high hit rate) with a small probability of false positives (a small query time.)

Balancing False Negatives/Positives with MinHash via repetition.



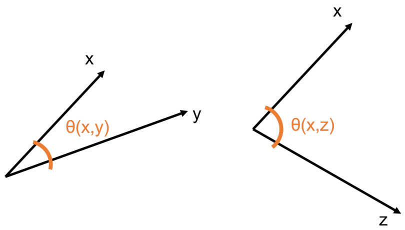
Create t hash tables. Each is indexed into not with a single MinHash value, but with r values, appended together. A length r signature:

$$\text{MH}_{i,1}(x), \text{MH}_{i,2}(x), \ldots, \text{MH}_{i,r}(x).$$

Hit Rate: Given by the s-curve: $1 - (1 - s^r)^t$.

Repetition and s-curve tuning can be used for fast similarity search with any similarity metric, given a locality sensitive hash function for that metric.

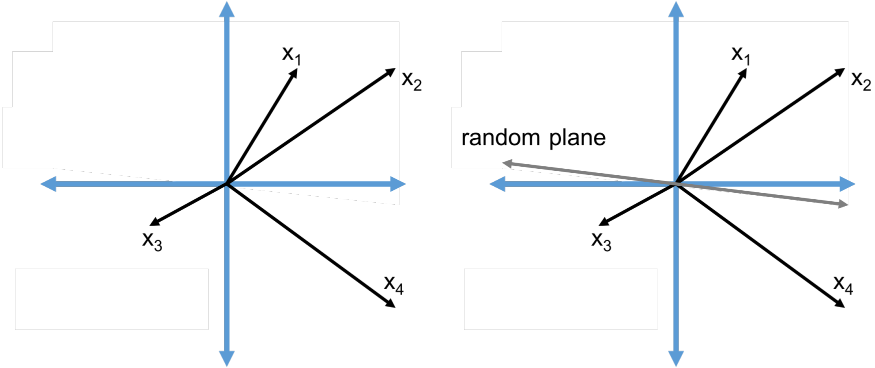
- LSH schemes exist for many similarity/distance measures: hamming distance, cosine similarity, etc.



Cosine Similarity: $\cos(\theta(x,y)) = \frac{\langle x,y\rangle}{\|x\|_2 \cdot \|y\|_2}$.

- $\cos(\theta(x,y)) = 1$ when $\theta(x,y) = 0°$ and $\cos(\theta(x,y)) = 0$ when $\theta(x,y) = 90°$, and $\cos(\theta(x,y)) = -1$ when $\theta(x,y) = 180°$

SimHash Algorithm: LSH for cosine similarity.

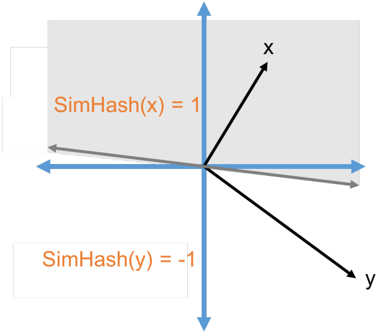


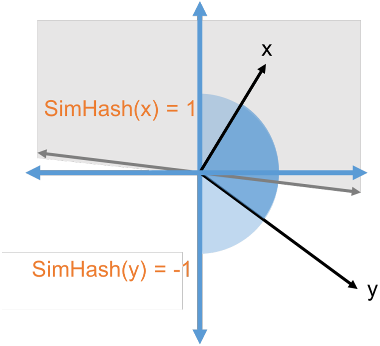
$SimHash(x) = \text{sign}(\langle x, t \rangle)$ for a random vector t.

What is $\Pr[SimHash(x) = SimHash(y)]$?

What is $\Pr[SimHash(x) = SimHash(y)]$?

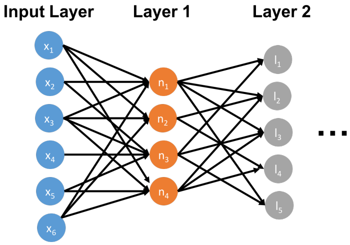
$SimHash(x) \neq SimHash(y)$ when the plane separates x from y.

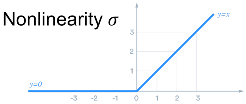




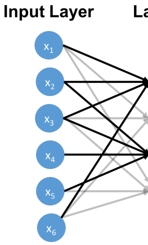
- $\Pr[SimHash(x) \neq SimHash(y)] = \frac{\theta(x,y)}{\pi}$
- $\Pr[SimHash(x) = SimHash(y)] = 1 - \frac{\theta(x,y)}{\pi} \approx \frac{\cos(\theta(x,y))+1}{2}$

Many applications outside traditional similarity search. E.g., approximate neural net computation (Anshumali Shrivastava).



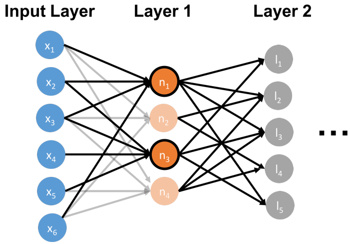


$$n_i = \sigma\left(\sum_{j=1}^{m} w(x_j, n_i)\cdot x_j\right) = \sigma(\langle w_i, x\rangle)$$

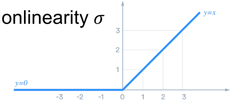


- Evaluating $\mathcal{N}(x)$ requires $|x|\cdot|\text{layer } 1| + |\text{layer } 1|\cdot|\text{layer } 2| + \ldots$ multiplications if fully connected.
- Can be expensive, especially on constrained devices like cellphones, cameras, etc.
- For approximate evaluation, suffices to identify the neurons in each layer with high activation when x is presented.

HASHING FOR NEURAL NETWORKS



Nonlinearity σ



$$n_i = \sigma\left(\sum_{j=1}^{m} w(x_j, n_i) \cdot x_j\right) = \sigma(\langle w_i, x \rangle)$$

- Important neurons have high activation $\sigma(\langle w_i, x\rangle)$.
- Since σ is typically monotonic, this means large $\langle w_i, x\rangle$.
- $\cos(\theta(w_i, x)) = \frac{\langle w_i, x\rangle}{\|w_i\|\|x\|}$. Thus these neurons can be found very quickly using LSH for cosine similarity search.
- Store each weight vector w_i (corresponding to each node) in a set of hash tables and check inputs x for similarity to these stored vectors.

Questions on MinHash and Locality Sensitive Hashing?

***k*-Frequent Items (Heavy-Hitters) Problem**: Consider a stream of n items $x_1, \ldots, x_n$ (with possible duplicates). Return any item at appears at least $\frac{n}{k}$ times.

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_1
5	12	3	3	4	5	5	10	3	5

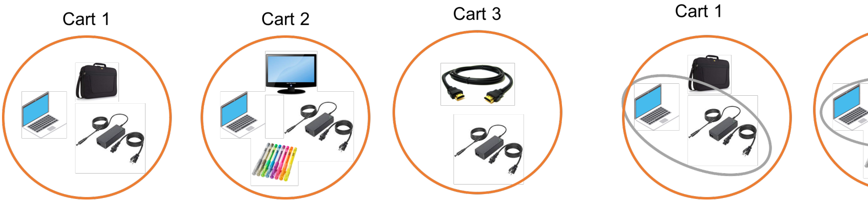
- What is the maximum number of items that must be returned? At most k items with frequency $\geq \frac{n}{k}$.
- Trivial with $O(n)$ space – store the count for each item and return the one that appears $\geq n/k$ times.
- Can we do it with less space? I.e., without storing all n items?
- Similar challenge as with the distinct elements problem.

Applications of Frequent Items:

- Finding top/viral items (i.e., products on Amazon, videos watched on Youtube, Google searches, etc.)
- Finding very frequent IP addresses sending requests (to detect DoS attacks/network anomalies).
- 'Iceberg queries' for all items in a database with frequency above some threshold.

Generally want very fast detection, without having to scan through database/logs. I.e., want to maintain a running list of frequent items that appear in a stream.

Association rule learning: A very common task in data mining is to identify common associations between different events.



- Identified via frequent itemset counting. Find all sets of k items that appear many times in the same basket.
- Frequency of an itemset is known as its support.
- A single basket includes many different itemsets, and with many different baskets an efficient approach is critical. E.g., baskets are Twitter users and itemsets are subsets of who they follow.

Majority: Consider a stream of n items $x_1, \ldots, x_n$, where a single item appears a majority of the time. Return this item.

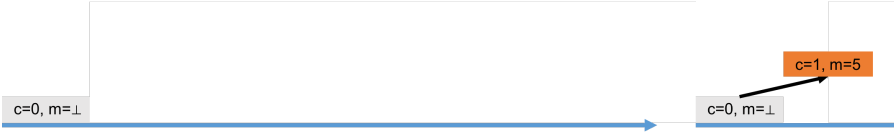
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
5	12	3	5	4	5	5	10	5	5

- Basically k-Frequent items for $k = 2$ (and assume a single item has a strict majority.)

Boyer-Moore Voting Algorithm: (our first *deterministic algorithm*)

- Initialize count $c := 0$, majority element $m := \perp$
- For $i = 1, \ldots, n$
  - If $c = 0$, set $m := x_i$ and $c := 1$.
  - Else if $m = x_i$, set $c := c + 1$.
  - Else if $m \neq x_i$, set $c := c - 1$.

Just requires $O(\log n)$ bits to store c and space to store m.



x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
5	12	3	5	4	5	5	10	5	5

x_1	x_2
5	12

Boyer-Moore Voting Algorithm:

- Initialize count $c := 0$, majority element $m := \perp$
- For $i = 1, \ldots, n$
  - If $c = 0$, set $m := x_i$ and $c := 1$.
  - Else if $m = x_i$, set $c := c + 1$.
  - Else if $m \neq x_i$, set $c := c - 1$.

Claim: The Boyer-Moore algorithm always outputs the majority element, regardless of what order the stream is presented in (if it is a strict majority).

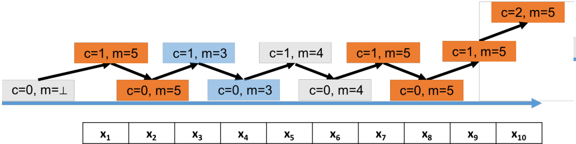
CORRECTNESS OF BOYER-MOORE

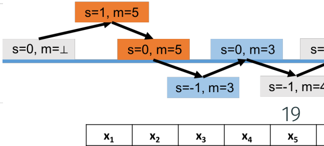
Boyer-Moore Voting Algorithm:

- Initialize count $c := 0$, majority element $m := \perp$
- For $i = 1, \ldots, n$
  - If $c = 0$, set $m := x_i$ and $c := 1$.
  - Else if $m = x_i$, set $c := c + 1$.
  - Else if $m \neq x_i$, set $c := c - 1$.

Claim: The Boyer-Moore algorithm always outputs the majority element, regardless of what order the stream is presented in.

Proof: Let M be the true majority element. Let $s = c$ when $m = M$ and $s = -c$ otherwise (s is a 'helper' variable).





Next Time: Will see a variant on the Boyer-Moore algorithm – the Misra-Greis summary.

- Stores k top items at once and solves the Frequent Items problem.