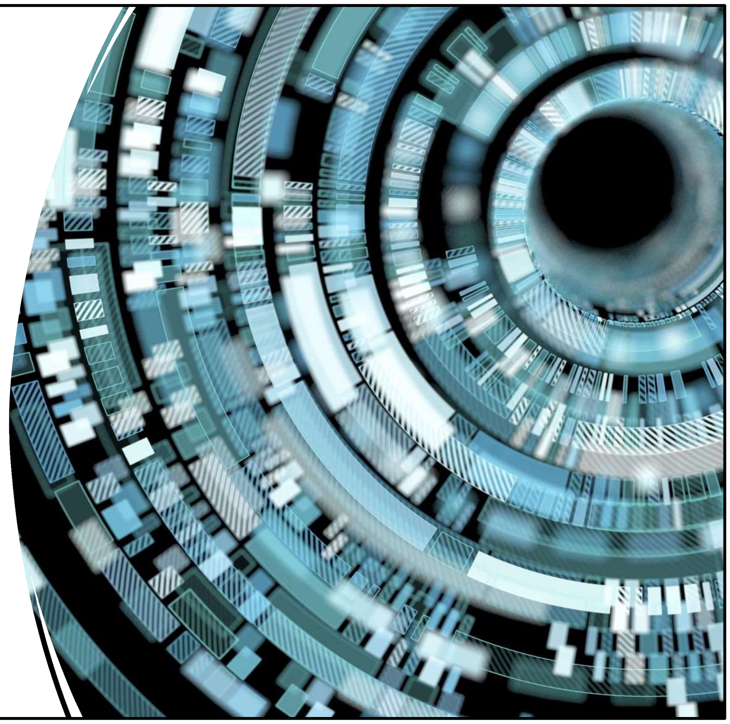
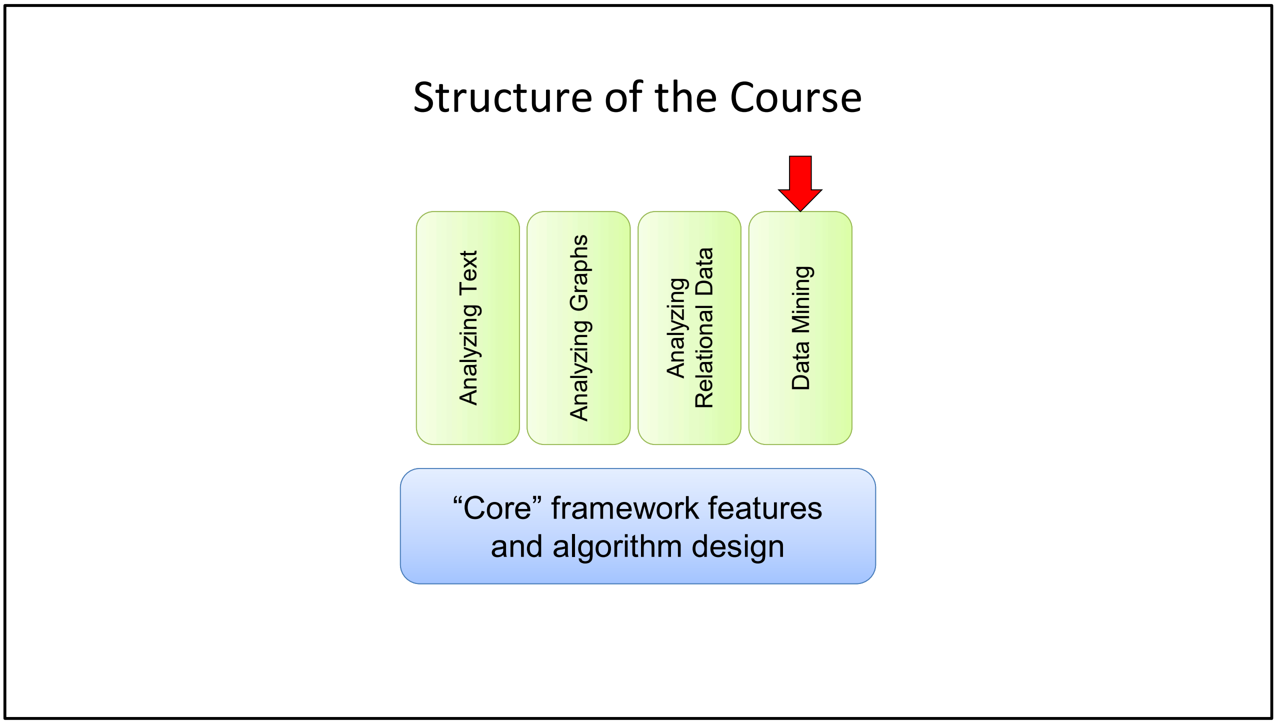
Data-Intensive Distributed Computing CS431/451/631/651

Module 6 – Data Mining / Machine Learning

Part 2 – LHS, Min-Hashing, and Random Hashing



Woah, trippy



What the, we skipped relational data?
(Yes, the assignments flow more easily this way…maybe I should change the graphic…)

Some of the following diagrams are borrowed

Thanks to Jure Leskovec, Anand Rajaraman, Jeff Ullman (Stanford University)

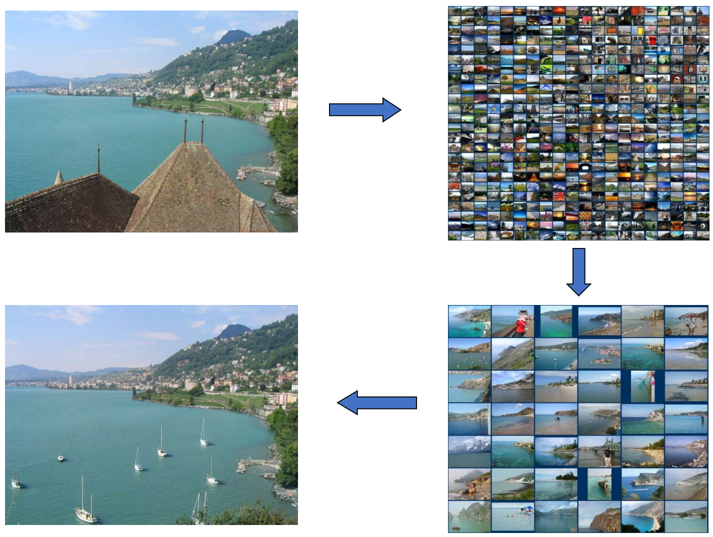
- If a slide says that at the bottom:
    - I've borrowed the whole slide, or
    - I've borrowed the diagrams and put my own words on them

Detour – Embeddings and Distance

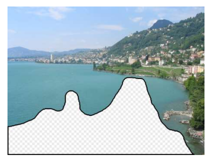
- Sometimes the feature vector is still too high-dimensional to work with!
- Example - text message with 140 characters: 256^{140} = really big & sparse
- Possible Embedding – character 4-grams – 256^{4} = still really big, still sparse
- Some models DO have 2 billion parameters. A spam filter shouldn't

Here each feature would be 0 or 1, meaning "does not contain this 4-gram" or "does". You could also have them be counts instead of strictly 0 or 1. 0 or 1 is easier.

Scene Completion Problem

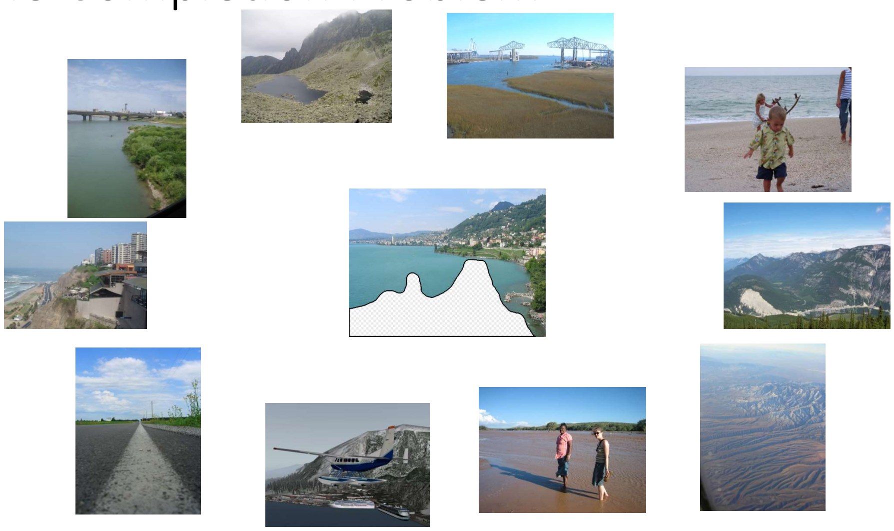


Scene Completion Problem



10 nearest neighbors, 20,000 image database

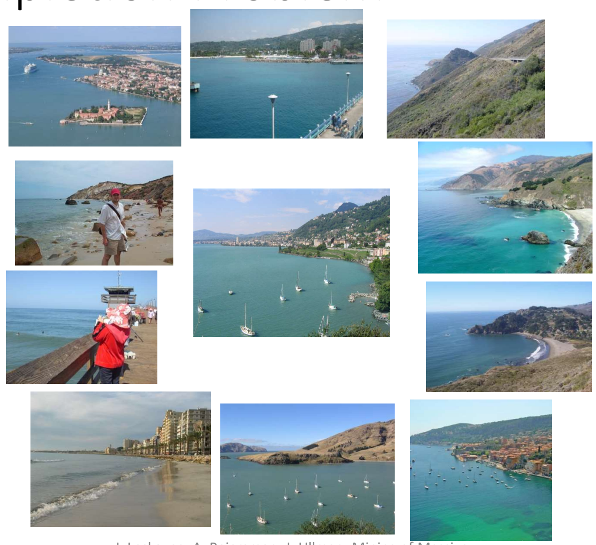
Scene Completion Problem



10 nearest neighbors from a collection of 20,000 images

10 nearest neighbors, 20,000 image database

Scene Completion Problem

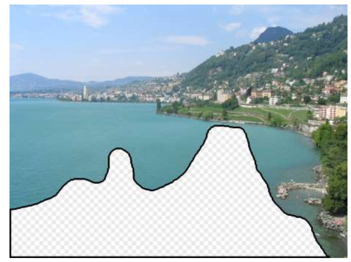


10 nearest neighbors from a collection of 2 million images

10 nearest neighbors, 2.3 million image database

A Common Metaphor

- **Many problems can be expressed as finding "similar" sets:**
  - **Find near-neighbors in high-dimensional space**
- **Examples:**
  - **Pages with similar words**
    - For duplicate detection, classification by topic
  - **Customers who purchased similar products**
    - Products with similar customer sets
  - **Images with similar features**
    - Users who visited similar websites



Today's Objective

Given high-dimensional datapoints x_1, $x_{2,}$ … x_n and some distance function $d(x_1, x_2)$:

Find **all pairs of datapoints x_i, x_j** s.t. $d(x_i,x_j) < s$

The naïve approach: just compute $d(x_i, x_j)$ for all i,j

$O(n^2)$ – not very big data

Magic: $O(n)$ – normal to want, and possible to achieve.

Why do we care? **The core technique applies to ML!**

The fingerprint is because we're going to be creating signatures (fingerprints)

Hey! "Magic" isn't an explanation!

Sure, but it caught your attention.

Q: How can you find all **identical** items in a collection of n?

A: Hash table – insert is O(1) – only need to compare collisions, not all pairs!

This is O(n) (assuming a low collision rate)

Locality Sensitive Hashing (LHS)

Normal Hash Function: If you know X and h(X):
No idea what h(X+c) is

LHS Hash Function: If you know X and h(X):
$E[h(X+c)] = h(x) \pm c$

Translation: items that are "close" have hash codes that are "close" (on average)

LHS to find Near Neighbours

1. Make a Hash Table
2. Use buckets that are **w** wide (and overlapped, so items go in multiple buckets)
3. Most values x_i, x_j s.t. $d(x_i, x_j) < c$ will have at least 1 bucket in common

Brilliant, so all we need is a LHS function?

Yup, we "**just**" need a LHS function.

We now spend the rest of the lecture "just" constructing such a creature.

"just" strikes again!

What’s Distance, Anyway?

Measuring distance between text documents:

> Remember, one goal is detecting duplicate news stories

- Diff: Too sensitive to rearrangement
- Word Count: not sensitive enough to rearrangement
- Edit Distance: too difficult to compute
- N-Grams: Just right

Jaccard

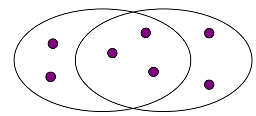
How do n-grams give us a distance measure? Jaccard distance! This is used for sets.

Do we have sets?

Yes: A document embedding is the set of n-grams it contains.

$$sim(C_1, C_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}$$

$$\mathrm{d}(\mathrm{C}_1, \mathrm{C}_2) = 1 - \mathrm{sim}(\mathrm{C}_1, \mathrm{C}_2)$$



3 in intersection
7 in union
Jaccard similarity= 3/7
Jaccard distance = 4/7

What if you can't make sets? Well, you'll need a different LHS technique. Sorry, this is only for sets!

Reminder: We want this for embeddings!

An embedding of binary (0 or 1) features is a set

If we have embeddings with dense floating point features we'll just have to do something else!

Fortunately, n-gram embeddings are pretty good

What *n* should I use?

Depends on if you're doing byte-level or word-level n-grams
Depends on what size of documents

For byte-level:

- 4-5 for short documents (tweets, sms, emails)
- 8-10 for long documents (webpages, books, papers)

For word-level:

- 2-3 for short documents
- 3-5 for long documents

The sentiment analysis paper used byte-level 4-grams, and so does the spam filter assignment!

Sets and Vectors

Reminder: A Set can be represented as a bit vector

1. Assign natural numbers 0…n to the elements of the Universe set
2. Bit vector at index *i* is 1 if the set contains element *I*
3. Benefit: union/intersection are bitwise or / bitwise and

I already made the opposite argument when I said our feature vector is a set!
But a little repetition never hurts

Collection as a Matrix

- Row = Elements of Set (i.e. n-gram)
- Column = Set (i.e. document)
- 1 in (*i*,*j*) -> n-gram *i* is in document *j*

Next Objective: Compute a signature for each column (document) s.t |sig| ≪ |col|

Ideally, column similarity = signature similarity

Documents

N-Grams

1	1	1	0
1	1	0	1
0	1	0	1
0	0	0	1
1	0	0	1
1	1	1	0
1	0	1	0

Reminder: LHS

A Hash function h(☺) such that:

- If C1 and C2 are highly similar, then with high probability:
  - $h(C_1) = h(C_2)$
- If C1 and C2 are highly dissimilar, then with high probability:
  - $h(C_1) \neq h(C_2)$
- Different approach needed for each definition of “similarity”

For Jaccard distance: **Min-Hashing**!

h(☺) - My Physics 12 teacher did this with all function definitions to avoid h(x) and confusing people with other X’s we’d already been talking about.
That or he was weird. Maybe both.

Min-Hashing

Imagine you have a random permutation π

LHS function $h_\pi(C) = min\ \pi(C)$

In other words: After permuting the rows of C by π, it's the index of the first 1

Not good enough as a signature, but what if we had 100 of them?

Min-Hashing Example

2nd element of the permutation is the first to map to a 1

Permutation π

2	4	3
3	2	4
7	1	7
6	3	2
1	6	6
5	7	1
4	5	5

Input matrix (N-Grams x Documents)

1	0	1	0
1	0	0	1
0	1	0	1
0	1	0	1
0	1	0	1
1	0	1	0
1	0	1	0

Signature matrix *M*

2	1	2	1
2	1	4	1
1	2	1	2

4th element of the permutation is the first to map to a 1

J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org

The Min-Hash Property

Claim: $\Pr[h(C_1) = h(C_2)] = sim(C_1,C_2)$
Proof:

Let $y = \min(\pi(C_1 \cup C_2))$
It must be the case that $y = \min(\pi(C_1))$ or $y = \min(\pi(C_2))$ – Why?

Is it in both though? There are $|C_1 \cap C_2|$ things in common.
And $|C_1 \cup C_2|$ possible y.
So the probability it's in both $= \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}$ = sim(C1, C_2)

Why? The permutation is random. So, every single element y in the union has the same chance of being placed first by the permutation. So it's a uniform random choice!

Signatures

Let K be the number of permutations (say 100)

Sig(C)[i] = the index of the first row that has a 1 in column C, after applying permutation *I*

The signature is K x 4 bytes (400 if K=100)

Much smaller than column C!

Similarity of Signatures

We know: $\Pr[h(C_1) = h(C_2)] = \mathrm{sim}(C_1,C_2)$

The signature is 100 such hash functions.

What's $\mathrm{sim}(\mathrm{Sig}_1, \mathrm{Sig}_2)$?

$E[\mathrm{sim}(\mathrm{Sig}_1,\mathrm{Sig}_2)] = \mathrm{sim}(C_1, C_2)$

(This is true no matter how many entries the sig has, but the more it has, the lower the variance)

Min-Hashing Example

Permutation π

2	4	3
3	2	4
7	1	7
6	3	2
1	6	6
5	7	1
4	5	5

Input matrix (N-Grams x Documents)

1	0	1	0
1	0	0	1
0	1	0	1
0	1	0	1
0	1	0	1
1	0	1	0
1	0	1	0

Signature matrix *M*

2	1	2	1
2	1	4	1
1	2	1	2

Similarities:

	1-3	2-4	1-2	3-4
Col/Col	0.75	0.75	0	0
Sig/Sig	0.67	1.00	0	0

Implementation

```
sig[C][i] = ∞ for all C, I
for each row index j in each column C:
  if C[j]:
    for each hash function index i:
      sig[C][i] = min(sig[C][i], h_i(j))
```

Problem: computing h_i is prohibitive!

In fact even writing down h_i is prohibitive

Why so Prohibitive?

How many n-length permutations are there? n!

(That's not an excited answer, it's n factorial)

How many bits needed to distinguish each possible permutation?

$$\Omega(\lg(n!)) = \Omega(n \lg n)$$

That's too many bits! If doing byte-level 4-grams,

2^{32} x 32 = 137,438,953,472 = 16GB

Alternative?

Let h_i be a k-universal hash function

```
for each hash function index i:
  sig[C][i] = min(sig[C][i], hi(j))
```

Conceptually: let π_i be the “permutation” we get if we sort using h_i as the key function

K-Universal Hash Function

Pick K constants – c1 … ck

$h(x) = (c1 + c_2x + c_3x^2 + \ldots c_kx^{k-1}) \bmod p$ p being a large prime

K-Universal means $h(x_1)$, $h(x_2)$, …. $h(x_k)$ will not correlate (but after that they might)

Is that good enough? Maybe

$Pr[h_k(y)=min(h_k(X))] = (1 / |X|)(1 \pm e^{-k})$

Only need a 4-universal function to be within 2% of the Jaccard distance

(at this point I ran out of time)

Remaining slides are

Thanks to Jure Leskovec, Anand Rajaraman, Jeff Ullman
(Stanford University)

2	1	4	1
1	2	1	2
2	1	2	1

LSH: First Cut

- **Goal:** Find documents with Jaccard similarity at least ***s*** (for some similarity threshold, e.g., ***s***=0.8)

- **LSH – General idea:** Use a function ***f(x,y)*** that tells whether ***x*** and ***y*** is a ***candidate pair***: a pair of elements whose similarity must be evaluated

- **For Min-Hash matrices:**
  - Hash columns of signature matrix ***M*** to many buckets
  - Each pair of documents that hashes into the same bucket is a **candidate pair**

Candidates from Min-Hash

2	1	4	1
1	2	1	2
2	1	2	1

- **Pick a similarity threshold *s* $(0 < s < 1)$**

- Columns ***x*** and ***y*** of ***M*** are a **candidate pair** if their signatures agree on at least fraction ***s*** of their rows: ***M*** **(*i, x*)** **=** ***M*** **(*i, y*)** for at least frac. ***s*** values of ***i***
  - We expect documents ***x*** and ***y*** to have the same (Jaccard) similarity as their signatures

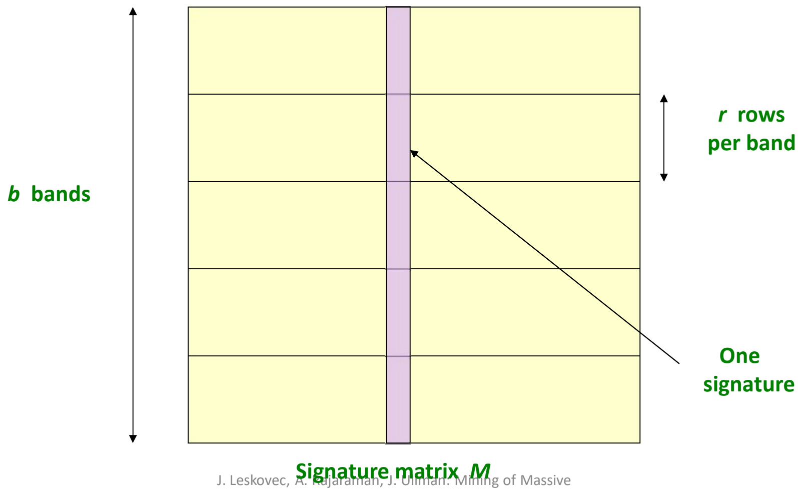
LSH for Min-Hash

2	1	4	1
1	2	1	2
2	1	2	1

- **Big idea: Hash columns of signature matrix *M* several times**
- Arrange that (only) **similar columns** are likely to **hash to the same bucket**, with high probability
- **Candidate pairs are those that hash to the same bucket**

Partition *M* into *b* Bands

2	1	4	1
1	2	1	2
2	1	2	1



Partition M into Bands

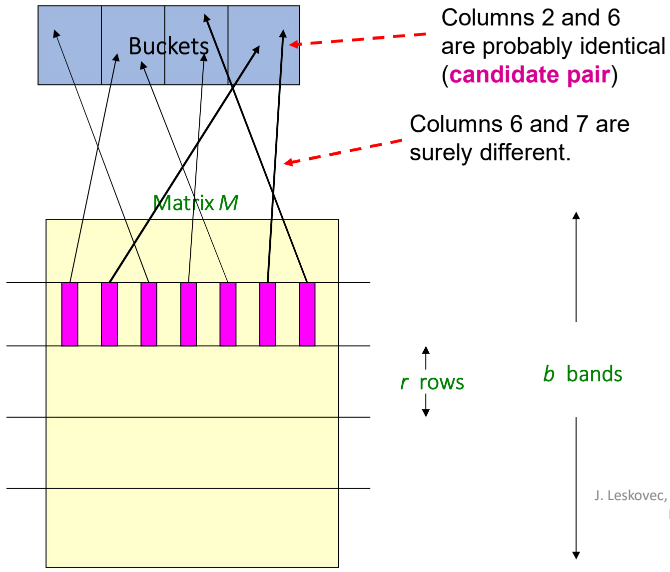
- Divide matrix ***M*** into ***b*** bands of ***r*** rows

- For each band, hash its portion of each column to a hash table with ***k*** buckets
  - Make ***k*** as large as possible

- ***Candidate*** column pairs are those that hash to the same bucket for ≥ **1** band

- Tune ***b*** and ***r*** to catch most similar pairs, but few non-similar pairs

Hashing Bands



J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org

38

Simplifying Assumption

- There are **enough buckets** that columns are unlikely to hash to the same bucket unless they are **identical** in a particular band
- Hereafter, we assume that "**same bucket**" means "**identical in that band**"
- Assumption needed only to simplify analysis, not for correctness of algorithm

Example of Bands

2	1	4	1
1	2	1	2
2	1	2	1

Assume the following case:

- Suppose 100,000 columns of ***M*** (100k docs)
- Signatures of 100 integers (rows)
- Therefore, signatures take 40Mb
- Choose ***b*** = 20 bands of ***r*** = 5 integers/band
- **Goal:** Find pairs of documents that are at least ***s*** = *0.8* similar

2	1	4	1
1	2	1	2
2	1	2	1

C_1, C_2 are 80% Similar

- **Find pairs of** $\geq$ ***s***=0.8 similarity, set **b**=20, **r**=5
- **Assume:** sim(C_1, C_2) = 0.8
  - Since sim(C_1, C_2) $\geq$ **s**, we want C_1, C_2 to be a **candidate pair**: We want them to hash to at **least 1 common bucket** (at least one band is identical)
- **Probability C_1, C_2 identical in one particular band:** $(0.8)^5 = 0.328$
- Probability C_1, C_2 are ***not*** similar in all of the 20 bands: $(1\text{-}0.328)^{20} = 0.00035$
  - i.e., about 1/3000th of the 80%-similar column pairs are **false negatives** (we miss them)
  - **We would find 99.965% pairs of truly similar documents**

2	1	4	1
1	2	1	2
2	1	2	1

C_1, C_2 are 30% Similar

- **Find pairs of** $\geq$ ***s***=0.8 similarity, set **b**=20, **r**=5
- **Assume:** sim(C_1, C_2) = 0.3
  - Since sim(C_1, C_2) < **s** we want C_1, C_2 to hash to **NO common buckets** (all bands should be different)
- **Probability C_1, C_2 identical in one particular band:** $(0.3)^5 = 0.00243$
- Probability C_1, C_2 identical in at least 1 of 20 bands: $1 - (1 - 0.00243)^{20} = 0.0474$
  - In other words, approximately 4.74% pairs of docs with similarity 0.3 end up becoming **candidate pairs**
    - They are **false positives** since we will have to examine them (they are candidate pairs) but then it will turn out their similarity is below threshold **s**

2	1	4	1
1	2	1	2
2	1	2	1

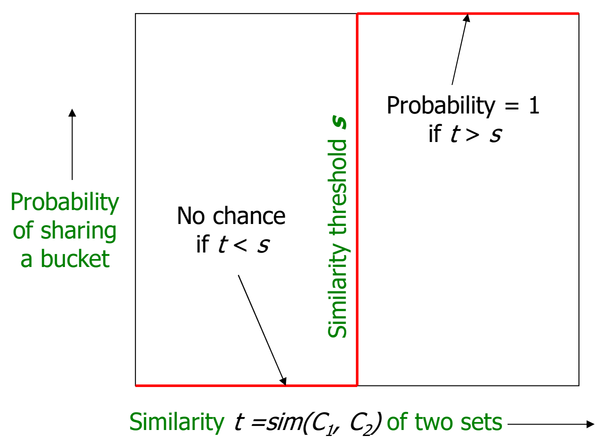
LSH Involves a Tradeoff

- **Pick:**
  - The number of Min-Hashes (rows of ***M***)
  - The number of bands ***b***, and
  - The number of rows ***r*** per band

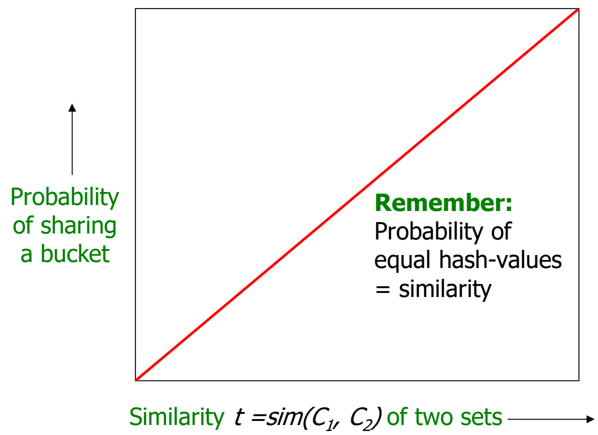
  to balance false positives/negatives

- **Example:** If we had only 15 bands of 5 rows, the number of false positives would go down, but the number of false negatives would go up

Analysis of LSH – What We Want

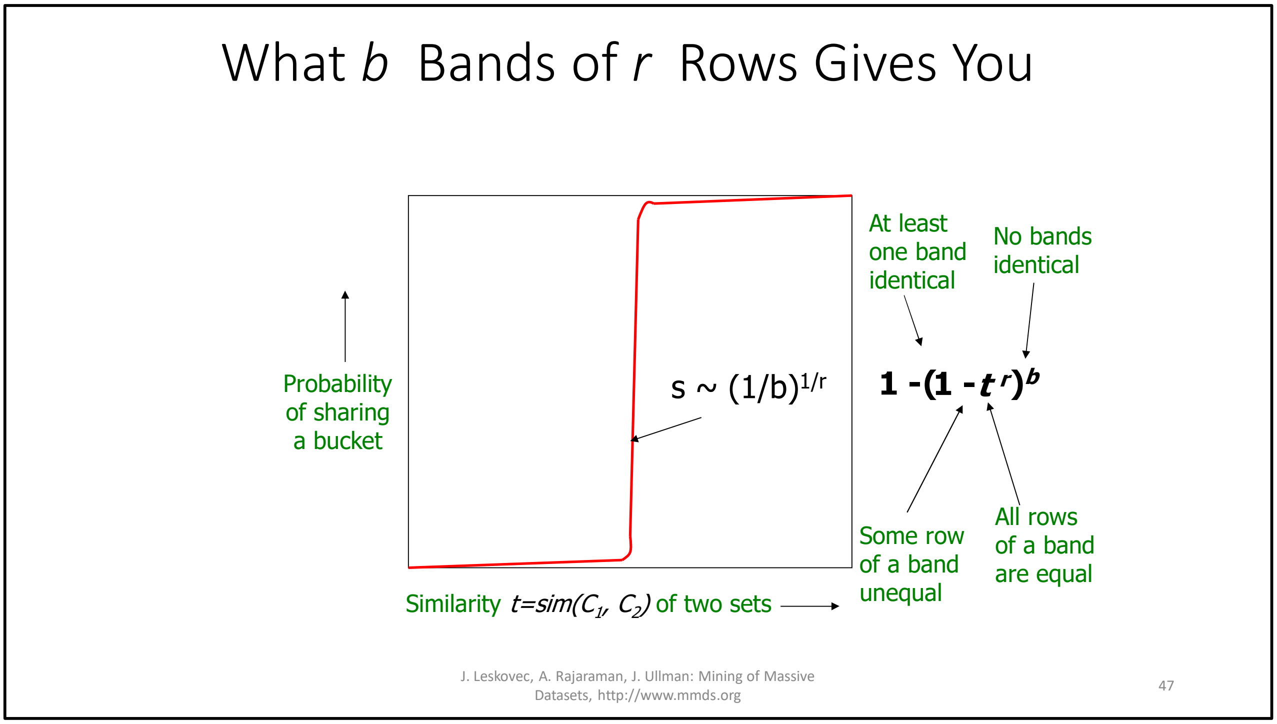


What 1 Band of 1 Row Gives You



b bands, *r* rows/band

- Columns C_1 and C_2 have similarity ***t***
- Pick any band (***r*** rows)
  - Prob. that all rows in band equal = t^r
  - Prob. that some row in band unequal = $1 - t^r$
- Prob. that no band identical = $(1 - t^r)^b$
- Prob. that at least 1 band identical = $1 - (1 - t^r)^b$



[Dan Note]: This is a hand-drawn sketch from their slides, that's why it's got a bit of a nub at the top. See next slide

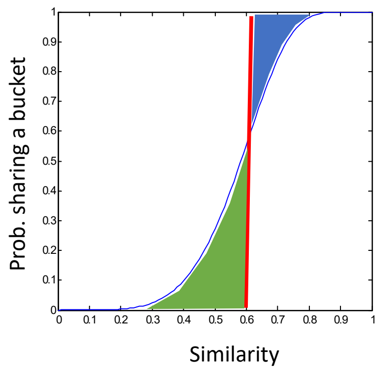
Example: $b = 20$; $r = 5$

- **Similarity threshold s**
- **Prob. that at least 1 band is identical:**

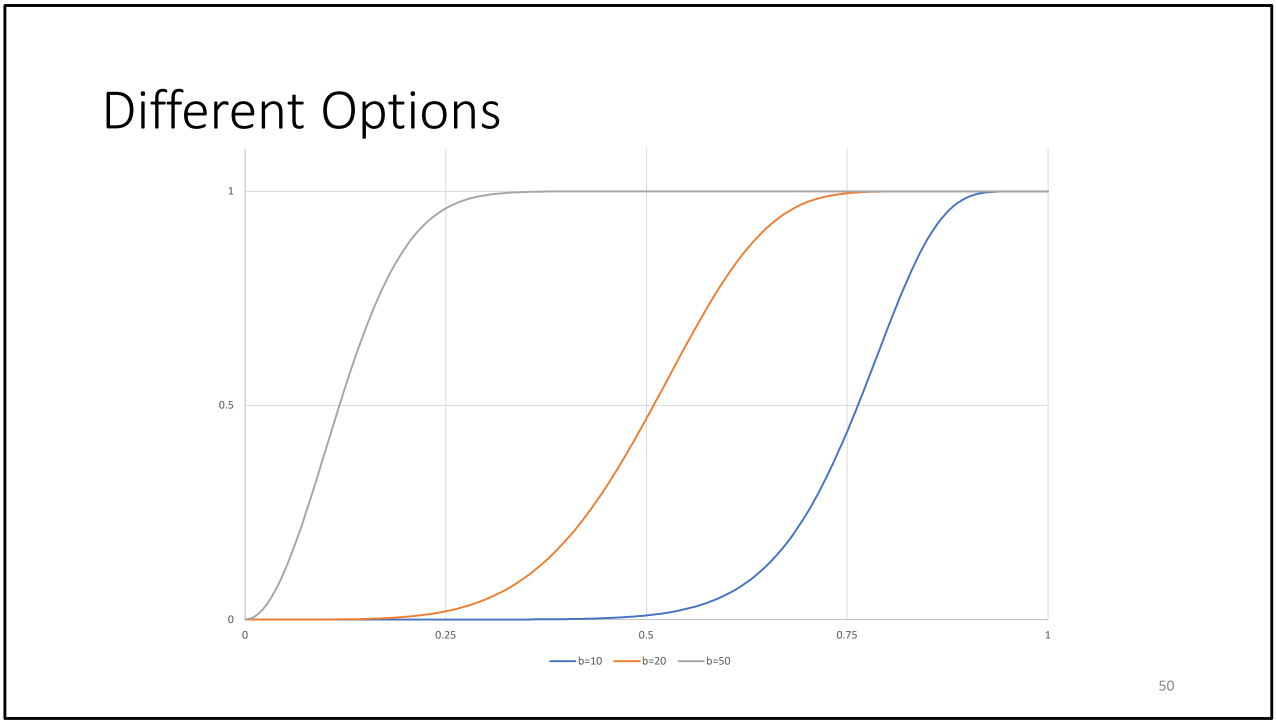
s	$1-(1-s^r)^b$
.2	.006
.3	.047
.4	.186
.5	.470
.6	.802
.7	.975
.8	.9996

Picking *r* and *b*: The S-curve

- **Picking *r* and *b* to get the best S-curve**
  - 50 hash-functions (r=5, b=10)



Blue area: False Negative rate
Green area: False Positive rate



Different values for b, given 100 elements in the signatures. (b * r = 100)

This is one of Dan’s slides again

LSH Summary

- Tune ***M, b, r*** to get almost all pairs with similar signatures, but eliminate most pairs that do not have similar signatures

- Check in main memory that **candidate pairs** really do have **similar signatures**

- **Optional:** In another pass through data, check that the remaining candidate pairs really represent similar documents

Summary: 3 Steps

- **N-Grams:** Convert documents to sets
  - We used hashing to assign each N-Gram an ID
- **Min-Hashing:** Convert large sets to short signatures, while preserving similarity
  - We used **similarity preserving hashing** to generate signatures with property $\mathbf{Pr}[h_\pi(C_1) = h_\pi(C_2)] = sim(C_1, C_2)$
  - We used hashing to get around generating random permutations
- **Locality-Sensitive Hashing:** Focus on pairs of signatures likely to be from similar documents
  - We used hashing to find **candidate pairs** of similarity $\geq$ **s**

Do it with Spark

1. Generate Signatures : **map**
2. Split each signature into bands: **flatMap**
3. Ship each band somewhere: **groupByKey** with custom partitioner
4. Find collisions within each band: **mapPartitions**
    1. Remove (some) false positives by double checking signatures are similar before emitting
5. Merge results: **union** -> **distinct**
6. [optional] remove remaining false positives by checking sets are similar (expensive): **filter**

What's this doing in the course???

Remember Sentiment Analysis???

"Features – sliding window byte-level 4-grams"

Length of bit-vector: 2^{32} (4 billion parameters, big model)

Length of example signature: 100

(but each value is an unsigned 32-bit integer)

What if we want bit-vectors still?

"Random Projection" – If you have N-length embeddings, and want M length – Create a random projection matrix (M x N)

- First row is a random unit vector
- Second row is a random unit vector orthogonal to first row
- Third row is a random unit vector orthogonal to first two
- …

Isn't that a Real Vector?

Yes – it will be a vector from $\mathbb{R}^k$

But – clamp like this: If the value is positive, it's 1, else it's 0

Tada! Bit vector!

Problem: It's expensive!

Solution: Have you tried not caring?

With high probability a set of random unit vectors is **approximately orthogonal**

Approximately orthogonal means that while the inner product is sufficiently small
The argument here is the same as

Can we go even faster?

Yes, through the power of hashing!
Define two hash functions: h, σ
h: [n] -> [m]   σ : [n] -> {-1,1}

For column i of the random projection matrix:
- entry at index h(i) is set to σ(i)
- all other entries are 0

Sparse: fast to compute
All columns are unit-vectors – trivially