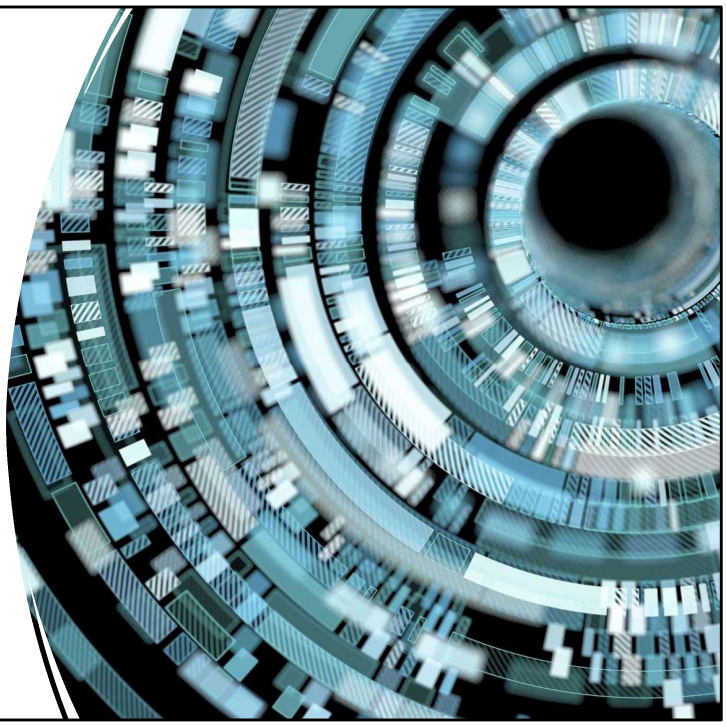
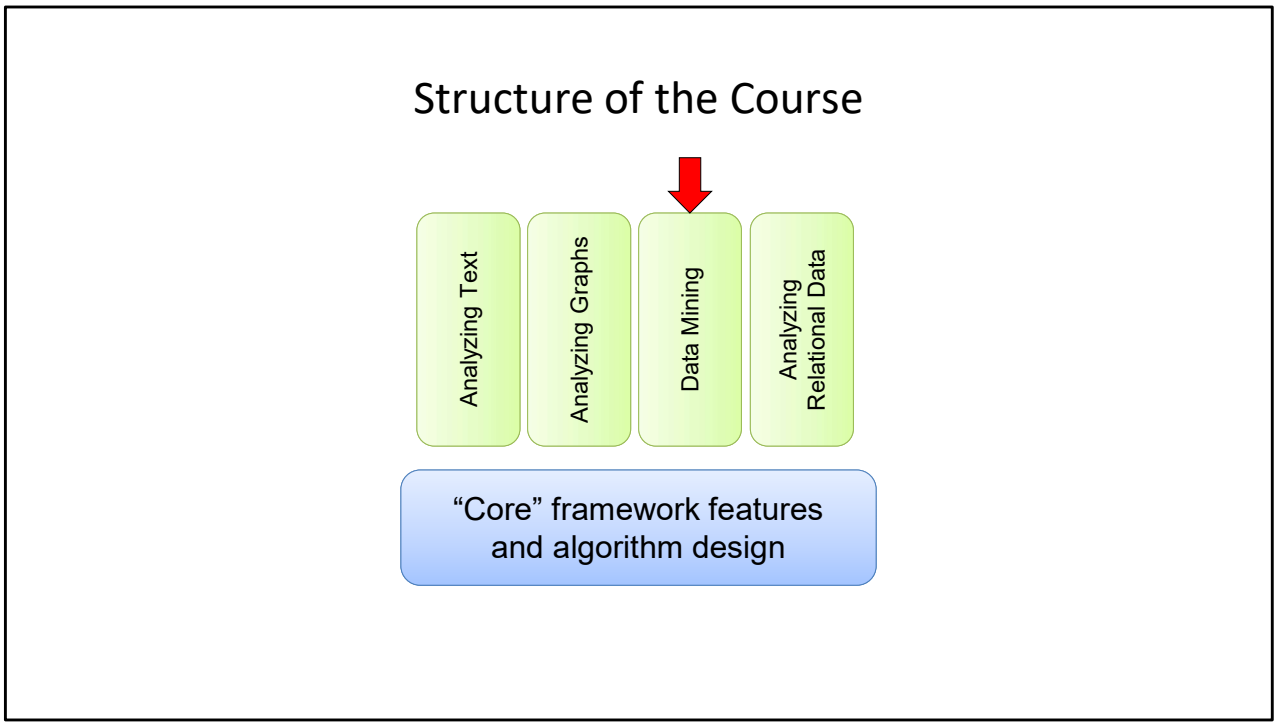
Data-Intensive Distributed Computing CS431/451/631/651

Module 6 – Data Mining / Machine Learning

Part 2 – LSH, Min-Hashing, and Clustering





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

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.

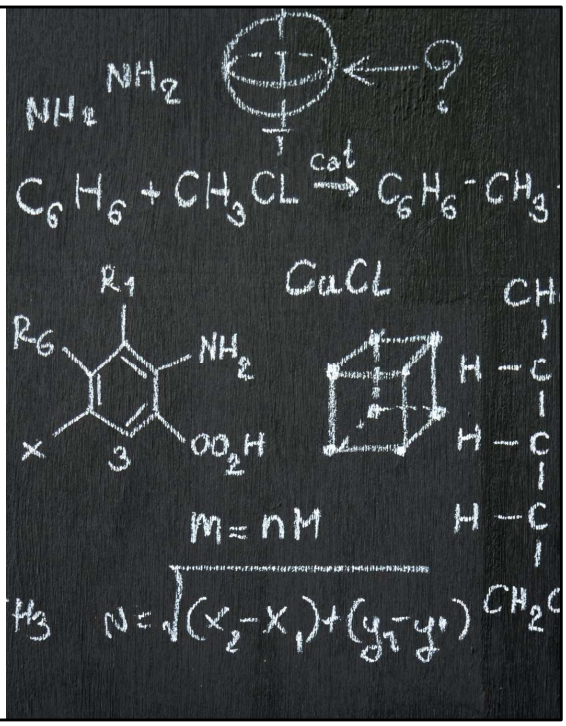
Reducing Dimensions

To reduce the dimensionality of a set of n-grams:

Hash them modulo some large prime (but much smaller than the original number of dimensions)

On the assignment: (mod 1,000,009)

1M << 4B : collisions are rare enough to ignore



Some of the following diagrams are borrowed

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

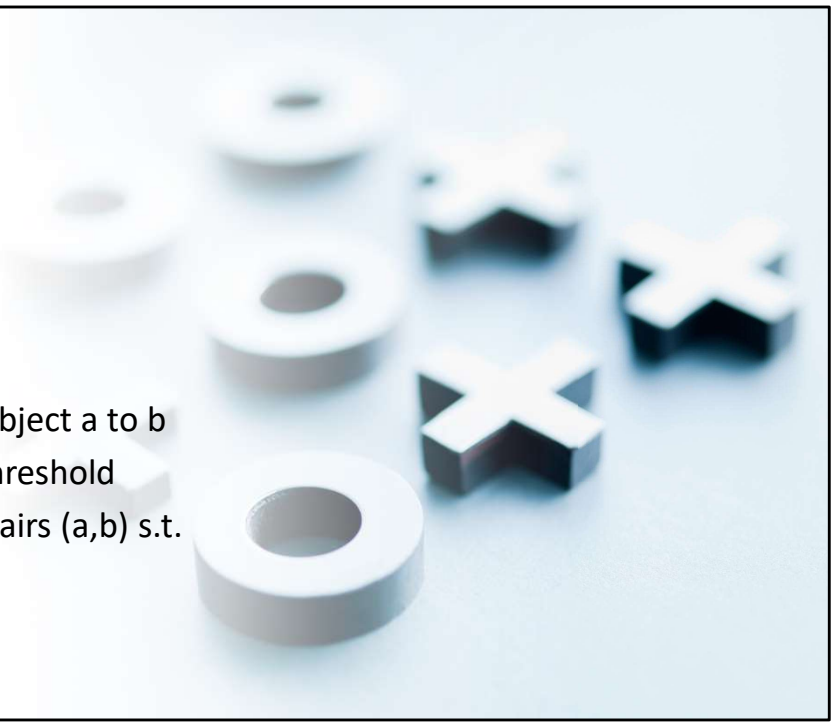
- If a slide says that at the bottom:
  - I've borrowed the whole slide, or
  - I've borrowed the diagrams and put my own words on them

Fun Problem – Near Neighbours

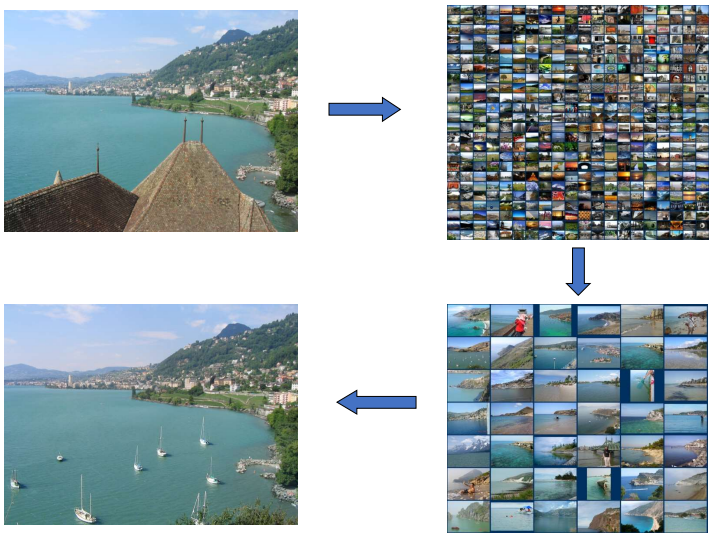
Problem:

- S – Set of Objects
- D(a,b) – Distance from object a to b
- t – maximum distance threshold

Goal: Find all unordered pairs (a,b) s.t. $D(a,b) \leq t$

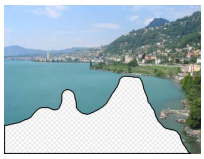


Scene Completion Problem



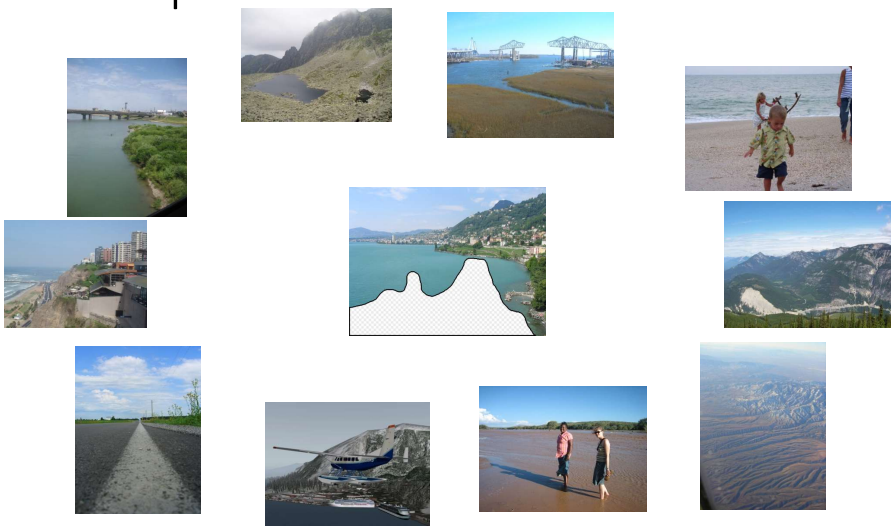
J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org

Scene Completion Problem



10 nearest neighbors, 20,000 image database

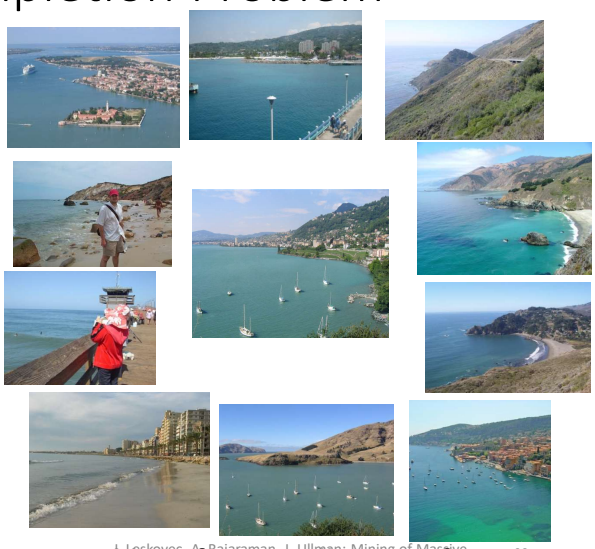
Scene Completion Problem



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

There are lots of ways to measure image similarity

Similar Visually

Similar Subject

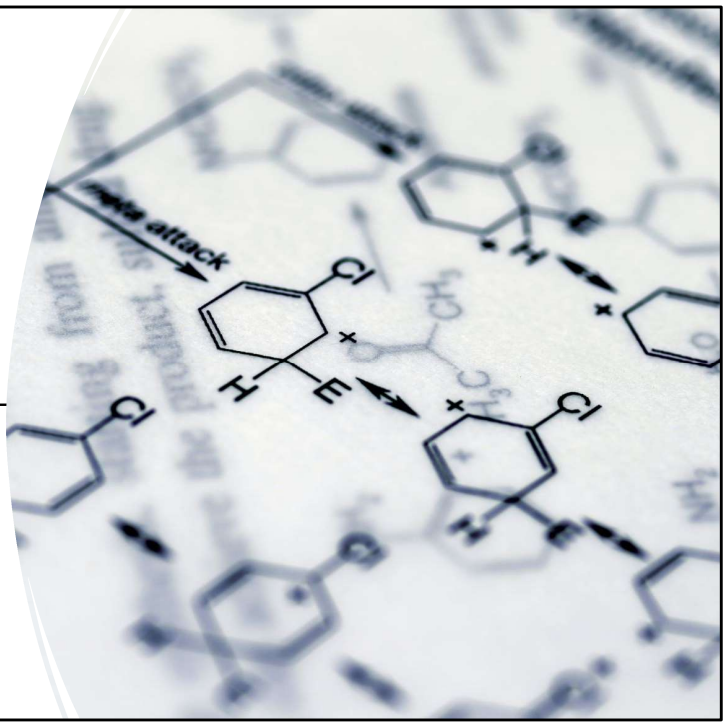
Visual Similarity is tricky – You need a model of human vision + cognition. Or just accept it's only going to be OK. Or use a neural network (they're essentially magic)

Similar Subject – Well…I guess you need the same things. A way to extract "what's in the image" and then a regular inverted index to find other images with the same "caption" – I know of one such thing, BLIP. It's a neural network again.

Template Based Protein Folding

Given a sequence, find proteins in database with a similar sequence.

- **NOT** as easy as hamming distance – not all mutations are equal.
  - Changing Leucine to Isoleucine (nearly identical)
  - Changing Leucine to Lysine (most properties different)
  - Also insertions / deletions



Why – Proteins with similar sequences tend to have similar shapes (folds). Well, they might be different at the places that differ, but it's at least a good starting point (if you start from scratch instead of a template, it's called "ab initio" which is Latin for ~~"from scratch"~~ "from the start")

Document Similarity

Near-Duplicate Detection is important for IR

- Lots of places that mirror usenet content
- Many news site post the same articles (newswire, freelance writers, etc.)
  - But not IDENTICAL – editor edits, website has own header/footer/"related article" links

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)

We assume that collisions due to identical hash codes will occur so rarely that we can ignore them. So collision is only due to the size of the table.
With a sufficiently large table, you can make the number of expected colissions sub-linear i.e. o(n) -- (the little o is on purpose)

Locality Sensitive Hashing (LSH)

For X, X' such that d(X,X') = c

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

LSH Hash Function: If you know X and h(X): $E[d_{hash}(h(X), h(X')] = c$

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

Things to note:

1. It's only the expected value. So it's OK if sometimes the distance is more than c, and sometimes less. (IDEALLY we want this to be a normal distribution)
2. The distance metric is d_{hash} not d, because the hash function probably does NOT return the same type as X)

(Scalar) LSH 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
   1. Most values > c will NOT share a common bucket

This works if you have a scalar hash code, but not if we have a VECTOR code (or "signature vector" to its friends)
We'll get to that later.

Brilliant, so all we need is a LSH function?

Yup, we "**just**" need a LSH 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 / Bag of Words: not sensitive **enough** to rearrangement
- Edit Distance: too difficult to compute
- doc2vec cosine distance: (too advanced)
- N-Grams: Just right

Too advanced for this course I mean…

Jaccard

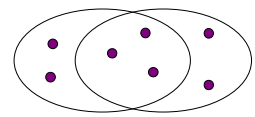
How do n-grams give us a distance measure? Jaccard distance! This is used for sets.

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 LSH 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!

Jaccard for Multisets

Just adjust the definitions of union and intersection

Union: If C = A U B then C[k] = max(A[k], B[k])

Intersection: If C = A ∩ B then C[k] = min(A[k], B[k])

|A| = the sum of the counts for all keys in A.

With that being said, n-gram embeddings are usually regular sets, not multisets. The counts for each n-gram don't (usually) make enough of a difference to justify the large increase in the size of the embedding.

Also the LSH we're about to see doesn't work for multi-sets so you'd need to come up with a different function.

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

Reminder: LSH

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. Either way I'm carrying on the tradition.

Min-Hashing – Two Views

Let C be a set of integers (each N-gram has been numbered)

Imagine you have a random permutation π

$h(C; \pi) = min_{i \in C}\ \pi(i)$

Let C be a set of n-grams (n-tuples of strings)

Enumeration of **all** n-grams, not just C

Imagine you have a random enumeration function π

$h(C; \pi) = min_{i \in C}\ \pi(i)$

Q: What on Earth is an enumeration function?
A1: It's a bijection that maps some set S onto the range [1, |S|]
(And if it's been a while, a bijection is a function that's one-to-one and onto)
A2: It's a function that enumerates a set of objects

Q: And enumerate means? Asking for a friend…
A: It means "count".

Are we Permutating or Enumerating?

Yes.

Permutating assumes the n-grams have already been enumerated and represented as a bit-vector

Enumerating assumes they are still a set of string tuples and we are assigning each n-gram an integer.

It's much easier to create a random permutation than it is to create a random n-gram enumerator. Being mathematically equivalent isn't the same thing as being computationally equivalent.

Collection as a Matrix

- Row = Elements of Set (i.e. n-gram)
- Column = one 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

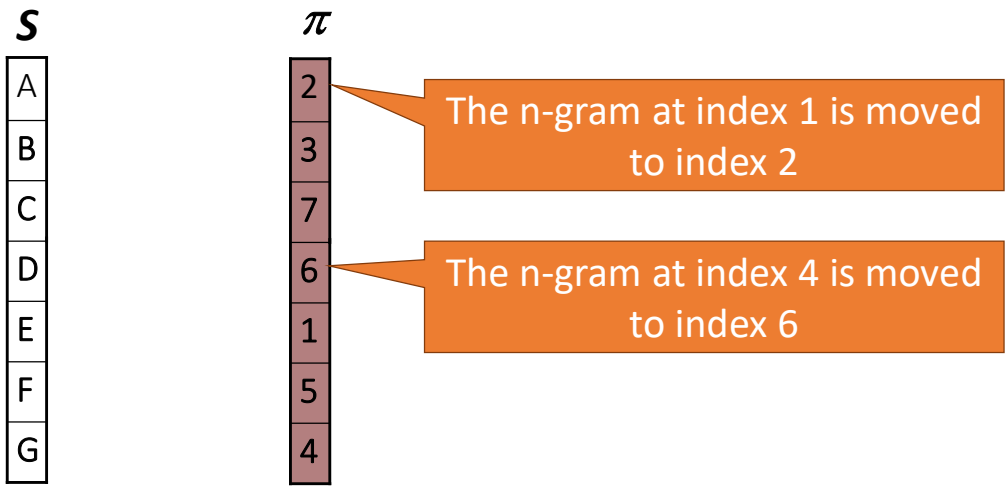
Documents

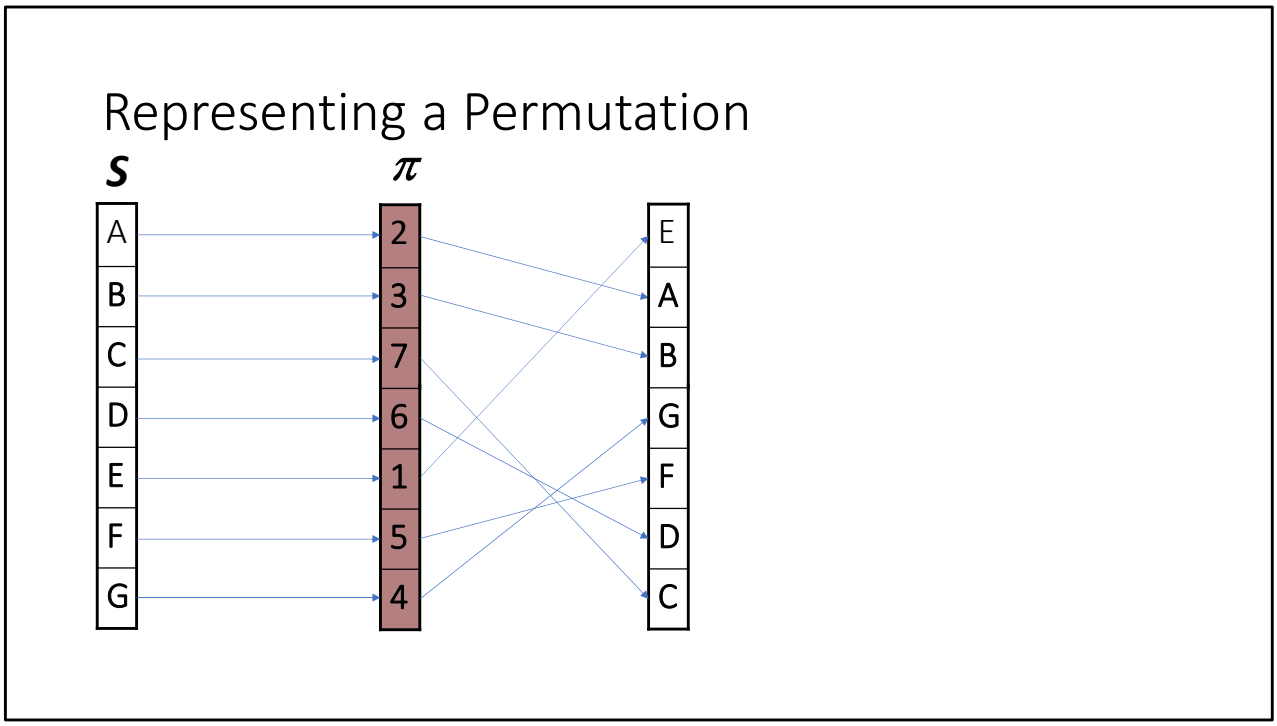
N-Grams

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

Repetition for Emphasis: row 1 is n-gram #1. It's NOT the string "1", but whichever n-gram string has been assigned (arbitrarily) as the "first" n-gram.

Representing a Permutation





In other words the rule is that if you process S and π in lockstep, the value you get from π is where the corresponding value from S gets moved to.

You can also define a permutation using the opposite rule, where the value you get from π tells you what index from S to retrieve (So you're processing π and the output in lockstep)

Remember: This kind of thing is important when you have a lot of data! You usually want to join the inputs together in this way. (Especially here, where we're not actually going to generate the complete permutation of the set…you'll see why in a second, if I haven't already blabbed about it)

It depends which is worse: random access of the output, or random access of the input!
In our situation, we don't need to compute the full π(S) , just one element at a time, so this approach is more efficient! All memory access is sequential.

The Min-Hash Property

Claim: $\Pr[h(C_1; \pi) = h(C_2; \pi)] = \text{sim}(C_1, C_2)$
Proof:

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

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

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!

Think of it this way: The only part of the permutation that matters are the entries in $C_1 \cup C_2$
The way to generate a permutation is: pick a random value, that's entry 1. Then pick another (without replacement), that's entry 2.
Since entry 1 is "smallest", a random permutation means picking the smallest element **uniformly**

Or think of it this way: If some elements are more likely to be small than others, how could you possibly call that unbiased?

So, h(C;π) is a LSH?

Yup

Let A, B be docs s.t. S(A,B) = s

h(A; π) = h(B; π) with an s% chance

$$S(\text{sig}_1, \text{sig}_2) = \begin{cases} 1 & \text{sig}_1 = \text{sig}_2 \\ 0 & \text{otherwise} \end{cases} \Rightarrow E[S] = s$$

That's the property we wanted for an LSH

However, the variance is through the roof, and although the expected value is right, two hash codes are either distance 0, or distance 1, with nothing in between. That's not good enough.

Signatures

Generate K independent permutations

Sig(C)[i] = $h(C;\pi_i)$ i.e. the min-hash defined by the ith permutation

The signature is K x 4 bytes (400 if K=100) assuming no more than 2^{32} n-grams

Much smaller than column C!

Why we want more than 1:

1. We'll see the math in a bit here
2. We wanted a normal distribution. We don't have one with one min-hash function, but as the number of trails (number of hash function) goes to +infinity, the distribution will converge on the normal distribution. (Central Limit Theorem)

Similarity of Signatures

$sim(Sig_1,Sig_2)$ = percentage of entries that are equal.

NOTE: Signature is NOT a set, cannot use Jaccard similarity!

We know: $Pr[h(C_1) = h(C_2)] = sim(C_1,C_2)$

$E[sim(Sig_1,Sig_2)] = sim(C_1, C_2)$

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

Min-Hashing Example

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

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

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

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

Why? If there are n! of them, if you have an encoding that **averages** less than $\Omega(n \lg n)$ bits, there will be fewer possible encodings than there are permutations to be encoded! That's a contradiction because

1. The concept of an encoding is that each unique object gets a unique encoding
2. Pigeonhole principle means that at least 1 encoding represents multiple objects

Those darn pigeons.

Alternative?

Let h_i be a k-universal hash function

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

Conceptually: let π_i be the "permutation" we get if we sort using h_i as the key function – break ties arbitrarily

Question for the class: Is that going to be the same as a permutation chosen uniformly at random?
(No, it's not, but it's close enough for government work)

K-Universal Hash Function

Pick K constants – $c_1 \ldots c_k$

$h(x) = (c_1 + c_2x + c_3x^2 + \ldots c_kx^{k-1}) \bmod p$ 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 for the probability to be within 2% of the Jaccard distance

Hold on, let's check the map…

Goal: Find all pairs of documents (A,B) s.t. Sim(A,B) >= s for some score s (e.g. 80%)

General Idea: LSH – A hash function where similar documents have similar hash codes

Our LSH – MinHash – If two documents have Jaccard similarity s, then E[f(A,B)] = s

OK, so…

Documents A,B are a “candidate pair” if MinHash(A,B) >= s.

Dan, my friend, does that not still require all pairwise comparisons?

LSH requires a hash table, too. Not just the codes!

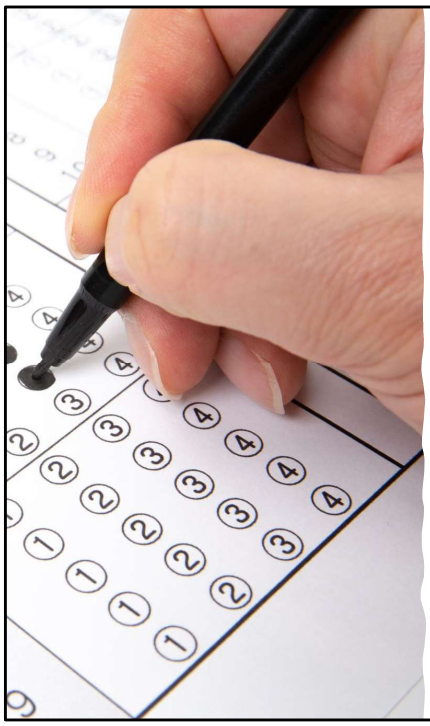
MinHash Table

A MinHash signature is an n-vector, not a scalar hash code.

How do you assign based on that?

1. An n-dimensional table (Or, equivalently, a 1-dimensional table where you use a secondary hash function on the sig vector)
2. n independent 1-dimensional tables
3. A secret third thing?

Problems
1 – Two documents with 80% similarity aren't THAT likely to have identical signatures so we'll miss a lot of them.
2 – Two documents with 10% similarity are likely to have at least 1 value in common in their signature vectors
(We COULD count how many times A,B collide but then we're back to pairwise comparisons, not practical at scale)



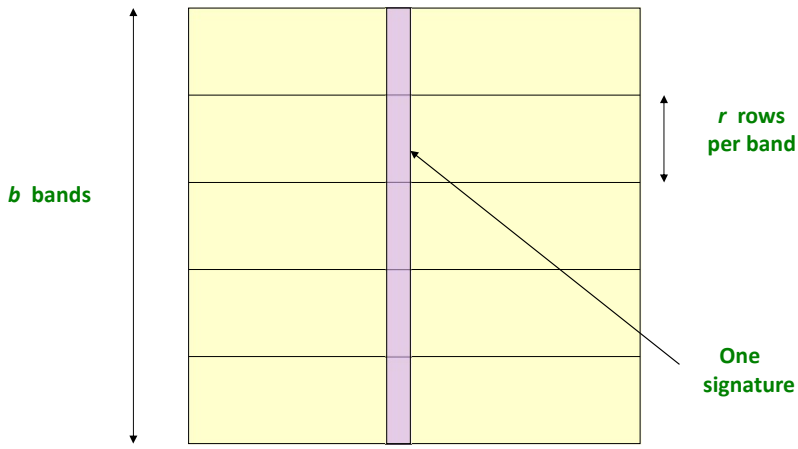
The Secret Third Thing (Middle Ground)

Slice the signature vector into b equal sized “bands” of r values each

Have b 1-dimensional tables, and consider (A,B) a candidate pair if they collide in **<u>any</u>** of the tables

Partition *M* into *b* Bands

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



Signature matrix *M*

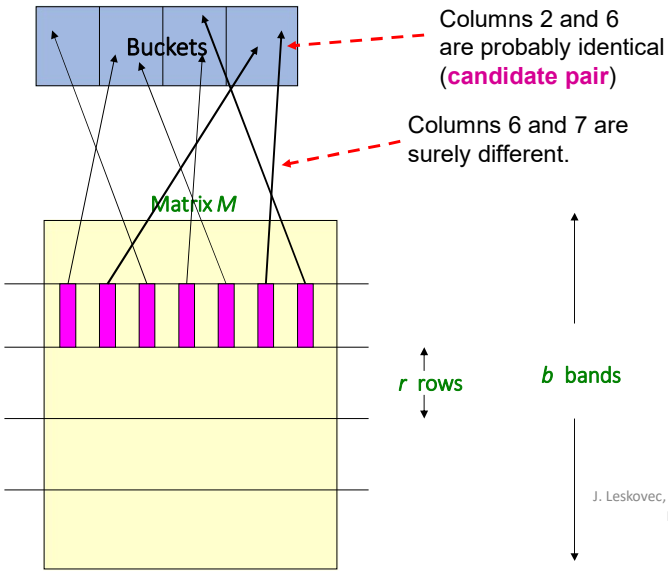
Big Assumption

Our hash tables will have a lot of buckets

(Or we'll use secondary hashing – Either way, we're going to ignore collisions)

This assumption **simplifies the math** (but doesn't change the correctness)

Hashing Bands



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

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 ≥ *s*=0.8** similarity, set **b**=20, **r**=5
- **Assume:** sim(C_1, C_2) = 0.8
  - Since sim(C_1, C_2) ≥ **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: $(1-0.328)^{20} = 0.00035$
  - 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 ≥ *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

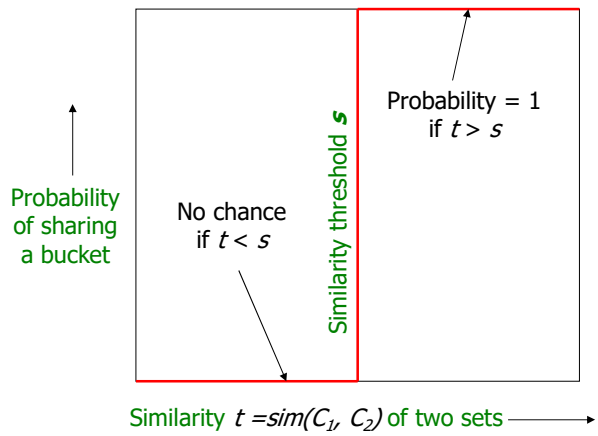
LSH Involves a Tradeoff

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

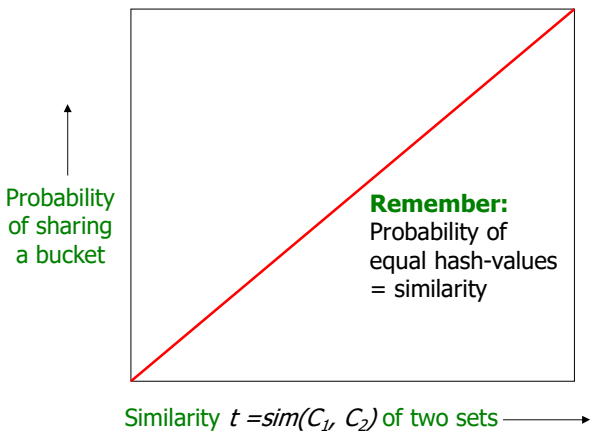
  to balance false positives/negatives

- **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

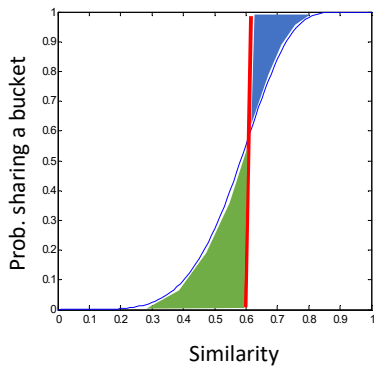


b bands, *r* rows/band

- 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 = $1 - (1 - t^r)^b$

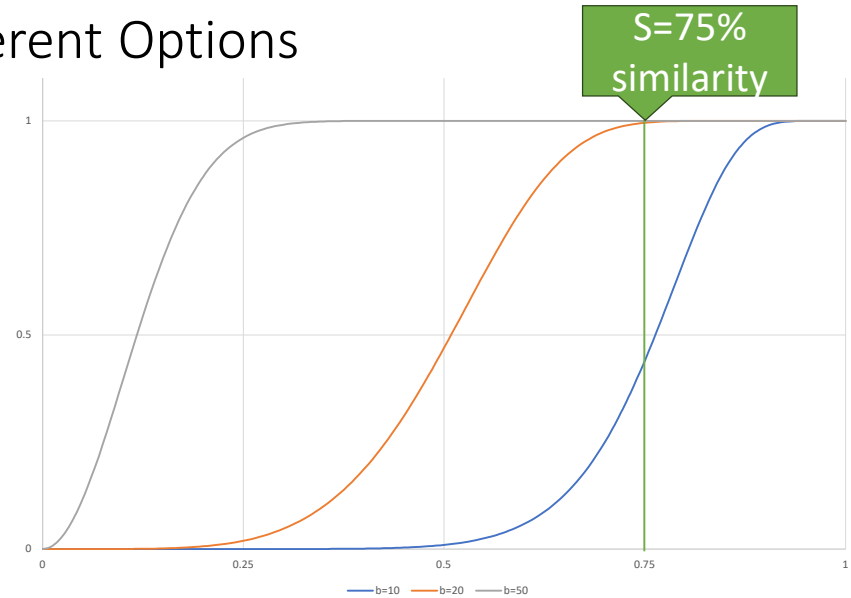
Picking *r* and *b*: The S-curve

- **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 Options



56

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

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

Summary

- Step 1 – convert each document into n-grams
  - Step 1.1 – convert each unique n-gram into an integer
- Step 2 – Generate a set of universal hash functions
- Step 3 – For each document, compute the short signature vector
- Step 4 – Pick values of R, B to tune to the false-positive and/or false negative rates you want
- Step 5 – Hash each of the B bands for each document to find candidate pairs
- Step 6 (technically optional, but absurd to skip) – Confirm the signatures are similar
- Step 7 (more optional) – Confirm the documents are similar

Why is step 6 recommended, but 7 less so? 6 is easy – the signatures are short! 7 is not. The sets are potentially quite large!

Isn’t this still quadratic?

Yes.

If the false positive rate is 5% you’d still need to compare 5% n(n-1)/2 candidate pairs only to reject them.

BUT: It’s a much faster comparison than Jaccard. And 0.05 is a small constant, isn’t it?

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**

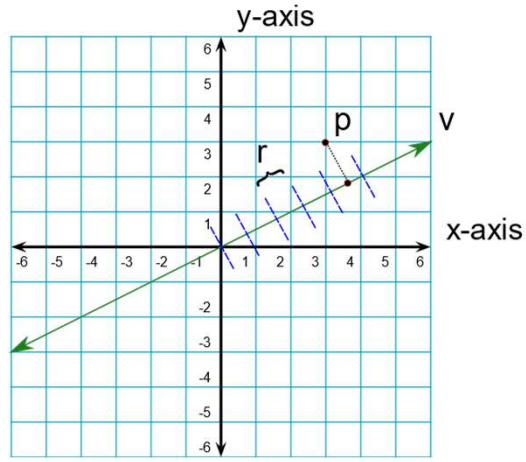
OR: `import org.apache.spark.ml.feature.MinHashLSH`

It's also another thing that's already in the Spark ML package.

What if I have n-dimensional coordinates and am using Euclidian Distance?

Bucketed Random Projection

1. Create a random unit n-vector v "axis of projection"
2. For each point p, dot with v to get a single value.
   The distance along v where p is projected
3. Pick "radius" r. Create buckets with width r (i.e. intervals along the axis of projection v)
4. Only need to do pairwise comparisons for documents in each bucket



The Spark ML Package has this one, too.

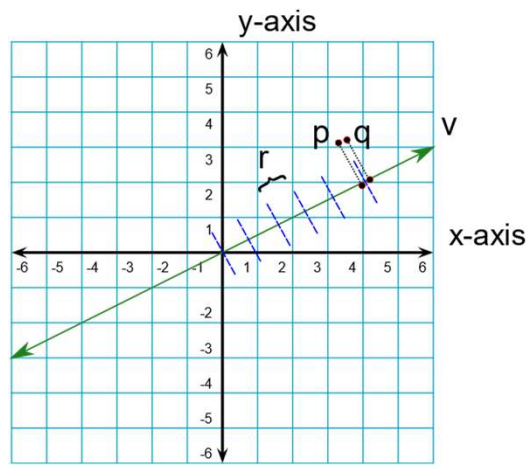
This is not on the exam, I just wanted to show you a second LSH real quick-like.

Bucketed Random Projection – Problem 1

p and q are very close, but in different buckets!

Solution(?): put the document into neighboring buckets, too.

But wait for problem 2.



If two points are t apart –

Their projected value might be t apart (if the vector pq is parallel to v)
Their projected value might be 0 (if pq is orthogonal to v)

Bucketed Random Projection – Problem 2

p and q are far apart, but project to the same value (no matter r, they are in the same bucket)

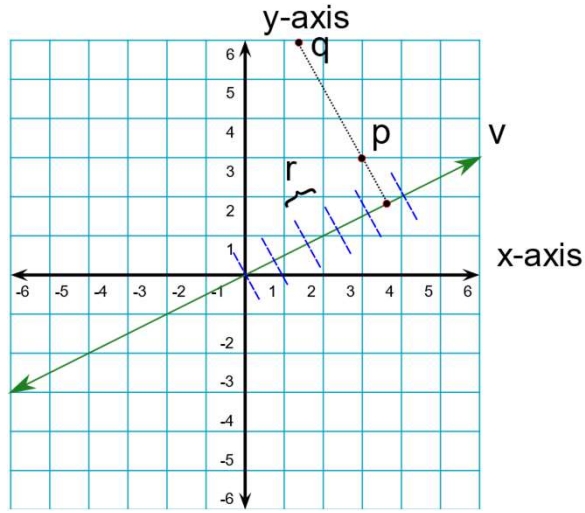
But that's gotta be rare…right?

WRONG.

The curse of dimensionality.

For two random vectors v_1 v_2,

$E[v_1 \bullet v_2]$ -> 0 as d -> infinity



Uh oh…

If the math isn't obvious from looking, it means our random projection axis v, statistically speaking, is going to be nearly orthogonal to just about any vector pq (where p and q are arbitrary data points from our collection)

In other words, in 2D the "points colinear" case seems rare, but for 768 dimensions? It's nearly every case!

The Curse of Dimensionality

Okay… so almost all points will be in bucket 0. Uh oh?

Uh oh indeed!

Solution:

- more than 1 projection (just like MinHashing)
- Non-uniform distribution (Gaussian)
- Ignore bucket 0

The “random vectors” assumes they’re uniformly random, so we can play with the distributions. With enough vectors we can break the curse.

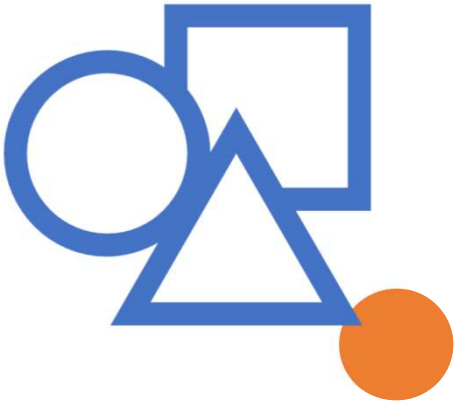
Ignoring bucket 0 is important as “most” pairs will be pairs in bucket 0. So being in bucket 0 is NOT a strong indicator of similarity. Being together in any other bucket is, though.

Bucketed Random Projection

Pick 3 random vectors v_1, v_2, v_3

For a datapoint p, obtain a 3-vector $(v_1 \cdot p, v_2 \cdot p, v_3 \cdot p)$

Playing with the size of buckets and number of random vectors lets you tune the false-positive probability (probability two random points p q will hash into the same bucket



The Blessing of Dimensionality

Remember "Dimensionality Reduction?"

One way to do that to Euclidian / Vector embeddings is a random projection matrix (from D_{high} to D_{low})

Computing a true projection matrix is expensive.

But the *Curse of Dimensionality* means if you pick independent random vectors, they're basically all orthogonal!

If you just pick D_{low} random unit vectors for your rows, your matrix is **basically** a projection matrix. Close enough for government work. (I say that a lot don't I?)

More Cursing

Another implication of the Curse of Dimensionality.

Take the unit hypercube and the unit hypersphere.

As d goes to infinity, Volume(Hypersphere) / Volume(Hypercube) -> 0

So? Well, it means we'll have problems with clustering…

Plus side is hyper dimensional brownies are almost entirely corner pieces!

What if I have n-dimensional unit vectors and am using cosine distance?

Cosine LSH

1. Generate k random hyperplanes
2. Signature: length k bit-vector
   1. 1 = above plane, 0 = below plane
3. Pick radius r – candidate pairs differ by no more than r bits

E.g. k = 8, r = 1, h(v) = 10010101

Put into buckets:

10010101

10010100

10010111

…

00010101

I usually skip this in class – definitely not on the exam.

Clustering

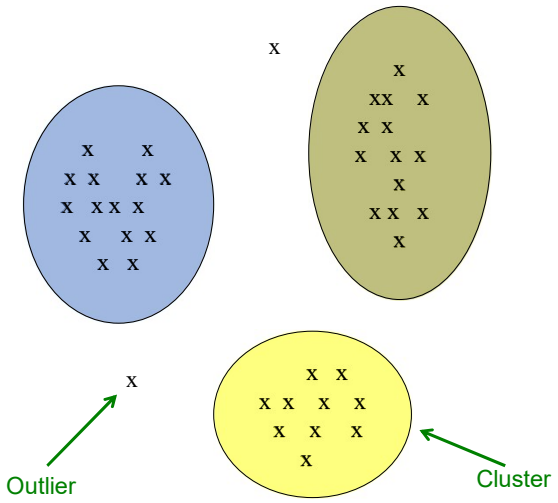
- **Given a cloud of data points we want to understand its structure**

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

The Problem of Clustering

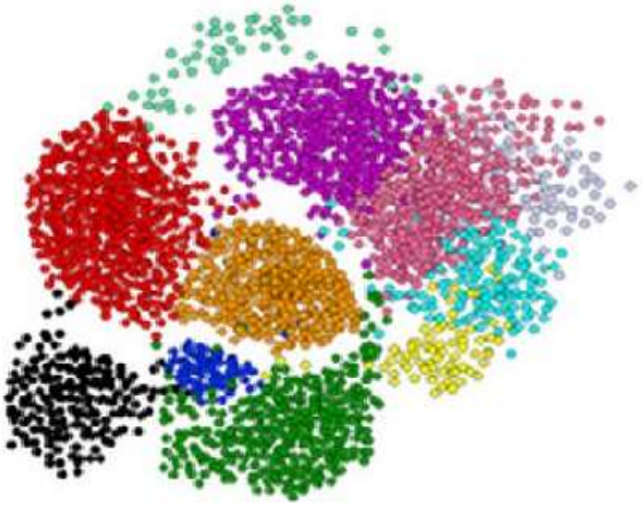
- Given a **set of points**, with a notion of **distance** between points, **group the points** into some number of ***clusters***, so that
  - Members of a cluster are close/similar to each other
  - Members of different clusters are dissimilar
- **Usually:**
  - Points are in a high-dimensional space
  - Similarity is defined using a distance measure
    - Euclidean, Cosine, Jaccard, edit distance, …

Example: Clusters & Outliers



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

Clustering is a hard problem!



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

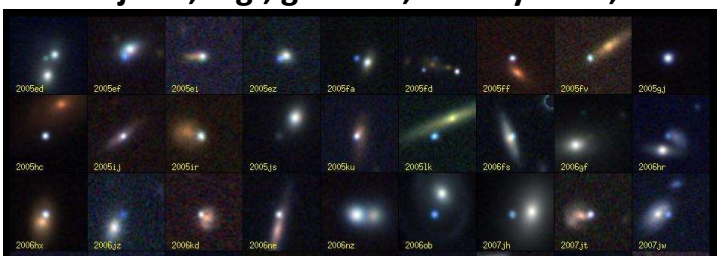
Why is it hard? The Curse of Dimensionality

- Clustering in two dimensions looks easy
- Clustering small amounts of data looks easy
- And in most cases, looks are not deceiving

- Many applications involve not 2, but 10 or 10,000 dimensions
- **High-dimensional spaces look different:** Almost all pairs of points are at about the same distance

Clustering Problem: Galaxies

- **A catalog of 2 billion "sky objects" represents objects by their radiation in 7 dimensions (frequency bands)**
- **Problem: Cluster into similar objects, e.g., galaxies, nearby stars, quasars, etc.**
- **Sloan Digital Sky Survey**



Clustering Problem: Music CDs

- **Intuitively: Music divides into categories, and customers prefer a few categories**
  - But what are categories really?
- Represent a CD by a set of customers who bought it:
- Similar CDs have similar sets of customers, and vice-versa

Clustering Problem: Music CDs

Space of all CDs:

- Think of a space with one dim. for each customer
  - Values in a dimension may be 0 or 1 only
  - A CD is a point in this space $(x_1, x_2, \ldots, x_k)$, where $x_i = 1$ iff the ith customer bought the CD
- For Amazon, the dimension is tens of millions
- **Task:** Find clusters of similar CDs

Clustering Problem: Documents

Finding topics:

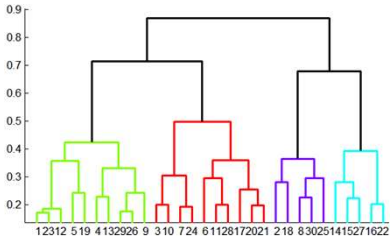
- Represent a document by a vector (x_1, x_2,…, x_k), where x_i = 1 iff the i th word (in some order) appears in the document
  - It actually doesn't matter if k is infinite; i.e., we don't limit the set of words
- **Documents with similar sets of words may be about the same topic**

Cosine, Jaccard, and Euclidean

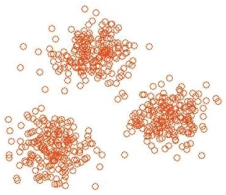
- **As with CDs we have a choice when we think of documents as sets of words or shingles:**
  - **Sets as vectors:** Measure similarity by the **cosine distance**
  - **Sets as sets:** Measure similarity by the **Jaccard distance**
  - **Sets as points:** Measure similarity by **Euclidean distance**

Overview: Methods of Clustering

- **Hierarchical:**
  - **Agglomerative** (bottom up):
    - Initially, each point is a cluster
    - Repeatedly combine the two "nearest" clusters into one
  - **Divisive** (top down):
    - Start with one cluster and recursively split it



- **Point assignment:**
  - Maintain a set of clusters
  - Points belong to "nearest" cluster

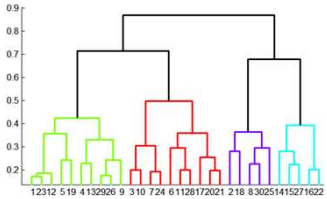


J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org

79

The top right image is called a "dendrogram" from Latin for branch. Because it's a tree. This is usually how hierarchical clusters are shown. You see this a lot in biology, it's a phylogenetic tree! (Created by clustering genomes using evolutionary edit distance as the metric).

Hierarchical Clustering



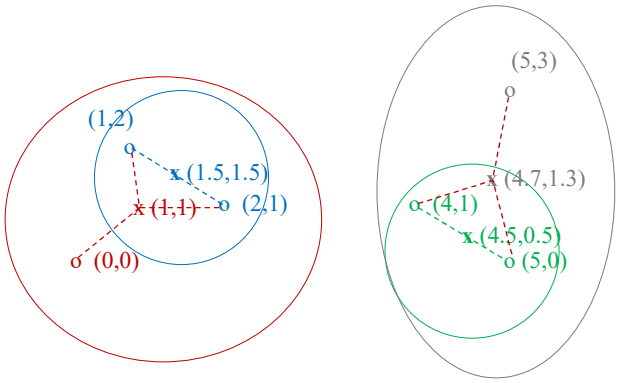
- **Key operation:**
  Repeatedly combine two nearest clusters

- **Three important questions:**
  - **1)** How do you represent a cluster of more than one point?
  - **2)** How do you determine the "nearness" of clusters?
  - **3)** When to stop combining clusters?

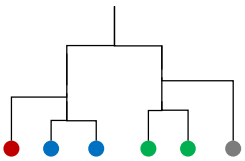
Hierarchical Clustering

- **Key operation: Repeatedly combine two nearest clusters**
- **(1) How to represent a cluster of many points?**
  - **Key problem:** As you merge clusters, how do you represent the "location" of each cluster, to tell which pair of clusters is closest?
- **Euclidean case:** each cluster has a ***centroid*** = average of its (data)points
- **(2) How to determine "nearness" of clusters?**
  - Measure cluster distances by distances of centroids

Example: Hierarchical clustering



Data:
o … data point
x … centroid



Dendrogram

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

82

And in the Non-Euclidean Case?

What about the Non-Euclidean case?

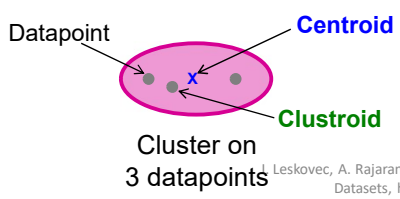
- The only "locations" we can talk about are the points themselves
  - i.e., there is no "average" of two points

- **Approach 1:**
  - **(1) How to represent a cluster of many points?**
    clustroid = (data)point "***closest***" to other points
  - **(2) How do you determine the "nearness" of clusters?** Treat clustroid as if it were centroid, when computing inter-cluster distances

"Closest" Point?

- **(1) How to represent a cluster of many points?**
  clustroid = point "***closest***" to other points
- **Possible meanings of "closest":**
  - Smallest maximum distance to other points
  - Smallest average distance to other points
  - Smallest sum of squares of distances to other points
    - For distance metric ***d*** clustroid ***c*** of cluster ***C*** is:

$$\min_{c} \sum_{x \in C} d(x,c)^2$$



Cluster on
3 datapoints

Centroid is the avg. of all (data)points in the cluster. This means centroid is an "artificial" point.
Clustroid is an **existing** (data)point that is "closest" to all other points in the cluster.

Defining "Nearness" of Clusters

- **(2) How do you determine the "nearness" of clusters?**
  - **Approach 2:**
    Intercluster distance = minimum of the distances between any two points, one from each cluster
  - **Approach 3:**
    Pick a notion of "**cohesion**" of clusters, *e.g.*, maximum distance from the clustroid
    - Merge clusters whose *union* is most cohesive

Cohesion

- **Approach 3.1:** Use the **diameter** of the merged cluster = maximum distance between points in the cluster
- **Approach 3.2:** Use the **average distance** between points in the cluster
- **Approach 3.3:** Use a **density-based approach**
  - Take the diameter or avg. distance, e.g., and divide by the number of points in the cluster

Implementation

- **Naïve implementation of hierarchical clustering:**
  - At each step, compute pairwise distances between all pairs of clusters, then merge
  - $O(N^3)$
- Careful implementation using priority queue can reduce time to $O(N^2 \log N)$
  - **Still too expensive for really big datasets that do not fit in memory**

k-means clustering

k–means Algorithm(s)

- Assumes Euclidean space/distance
- Start by picking ***k***, the number of clusters
- Initialize clusters by picking one point per cluster

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

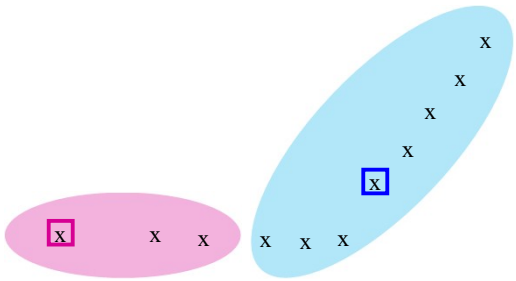
89

On picking one point: pick a random point, then for the rest of the clusters, pick something that maxiumizes the average distance between it and the existing points… kinda of expensive for large k, but usually k is small.

Populating Clusters

- **1)** For each point, place it in the cluster whose current centroid it is nearest
- **2)** After all points are assigned, update the locations of centroids of the ***k*** clusters
- **3)** Reassign all points to their closest centroid
  - Sometimes moves points between clusters
- **Repeat 2 and 3 until convergence**
  - **Convergence:** Points don't move between clusters and centroids stabilize
  - "Fix point"

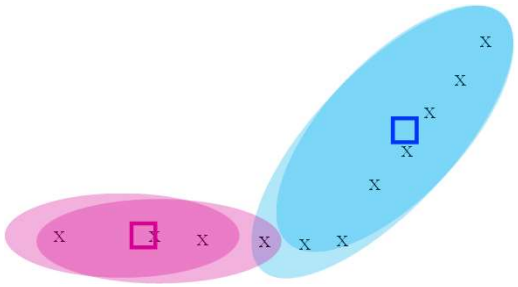
Example: Assigning Clusters



x … data point
□ … centroid

Clusters after round 1

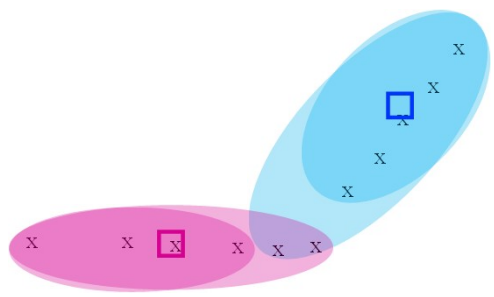
Example: Assigning Clusters



x … data point
□ … centroid

Clusters after round 2

Example: Assigning Clusters



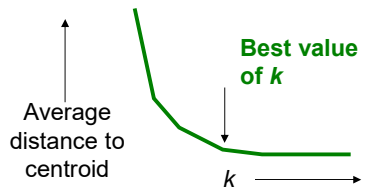
x … data point
□ … centroid

Clusters at the end

Getting the *k* right

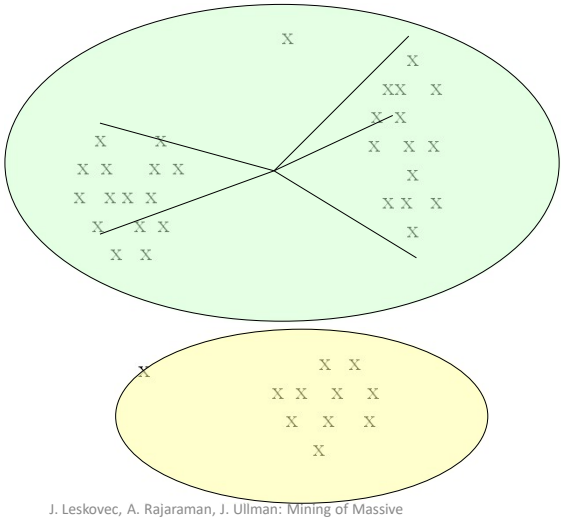
How to select *k*?

- Try different **k**, looking at the change in the average distance to centroid as **k** increases
- Average falls rapidly until right **k**, then changes little



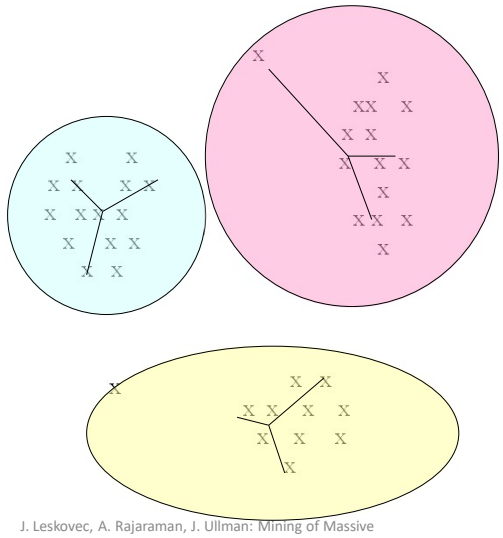
Example: Picking *k*

Too few;
many long
distances
to centroid.



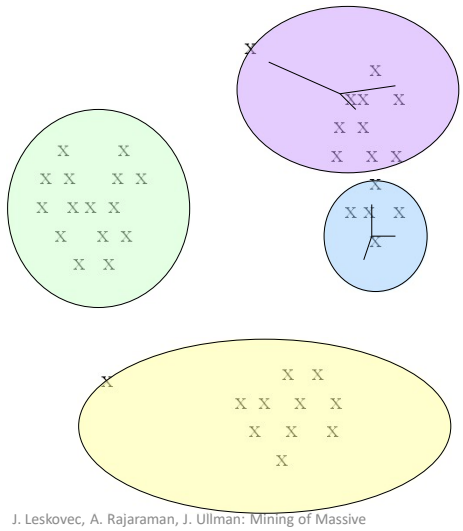
Example: Picking *k*

Just right; distances rather short.



Example: Picking *k*

Too many;
little improvement
in average
distance.



Basic MapReduce Implementation

```
class Mapper  {
  def setup() = {
    clusters = loadClusters()
  }

  def map(id: Int, vector: Vector) = {
    emit(clusters.findNearest(vector), vector)
  }
}

class Reducer {
  def reduce(clusterId: Int, values: Iterable[Vector]) = {
    for (vector <- values) {
      sum += vector
      cnt += 1
    }
    emit(clusterId, sum/cnt)
  }
}
```

Spark it up

Doing this in Spark is much better!

1. Pick random centroids
2. create pair RDD by assigning (key is cluster #) – partition by key and CACHE
3. reduceByKey + map to get new centroids
4. Create new pair RDD by reassigning using new centroids
    1. Accumulator tracks how many data points changed cluster ID
    2. Don't forget to unpersist the old assignment RDD
5. If accumulator > 0, goto step 3

Why is this so fast:

1. Datapoints held in memory
2. Because datapoints before / after reduceByKey are partitioned by cluster ID, partition change only occurs for data points that change cluster ID.
    1. So? So if their partition doesn't change, they "shuffle" to the worker node they're currently already on (so don't touch the disk, don't touch the network, just stay put)