Lecture 12: Locality-Sensitive Hashing and MinHash

Sam McCauley

October 17, 2025

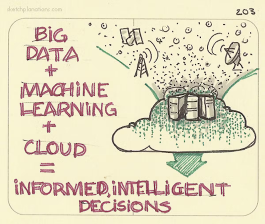
Williams College

Admin

- Midterm a week from today in class
- Practice midterm out in the next 24 hours; solutions early next week
- I created both the midterm and the practice midterm at the same time; they should be extremely similar in terms of structure and very similar in terms of topics
- No writing code on the midterm
- Review session Tuesday; bring questions
- Assignment 4 out tonight (I'm adding some extra starter code because I know you're also studying)
- I will grade everything before the midterm

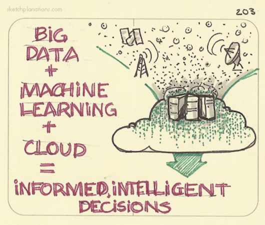
Finding Similar Items

Back to Normal Inputs



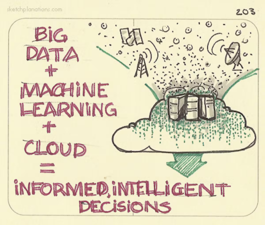
- Today: no more streaming! Have all data available to us.

Back to Normal Inputs



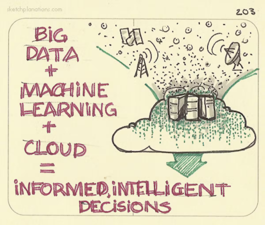
- Today: no more streaming! Have all data available to us.
- But data is still big!

Back to Normal Inputs



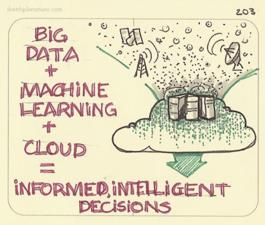
- Today: no more streaming! Have all data available to us.
- But data is still big!
    - In particular: high-dimensional

Back to Normal Inputs



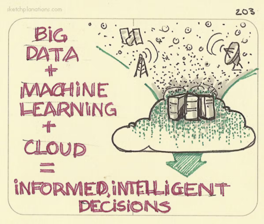
- Today: no more streaming! Have all data available to us.
- But data is still big!
    - In particular: high-dimensional
    - Table with many columns

Back to Normal Inputs



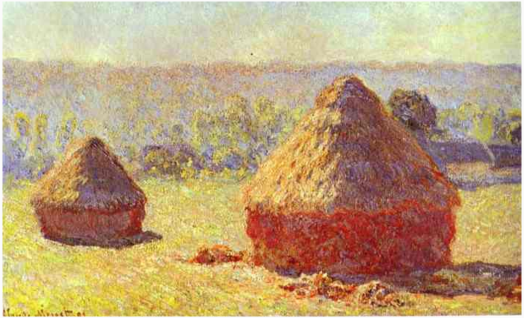
- Today: no more streaming! Have all data available to us.
- But data is still big!
  - In particular: high-dimensional
  - Table with many columns
  - For each Netflix user, what movies have they seen

Back to Normal Inputs



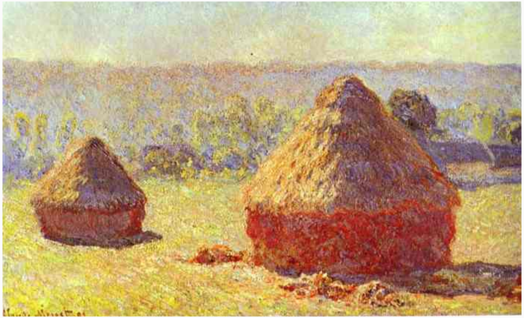
- Today: no more streaming! Have all data available to us.
- But data is still big!
    - In particular: high-dimensional
    - Table with many columns
    - For each Netflix user, what movies have they seen
- Goal: solve a computationally difficult, but important, problem. If you've taken an ML course it's reasonably likely that you've seen some variant of this problem.

Finding Similar Pair



- Given a set of objects

Finding Similar Pair



- Given a set of objects
- Find the most similar pair of objects in the set

Why Find Similar Objects?

- Find similar news articles for user suggestions.

Why Find Similar Objects?

- Find similar news articles for user suggestions.
- Similar music: Spotify suggests music by finding similar users, and selecting what they listen to

Why Find Similar Objects?

- Find similar news articles for user suggestions.
- Similar music: Spotify suggests music by finding similar users, and selecting what they listen to
- Machine learning in general (training, evaluation, actual algorithms, etc.)

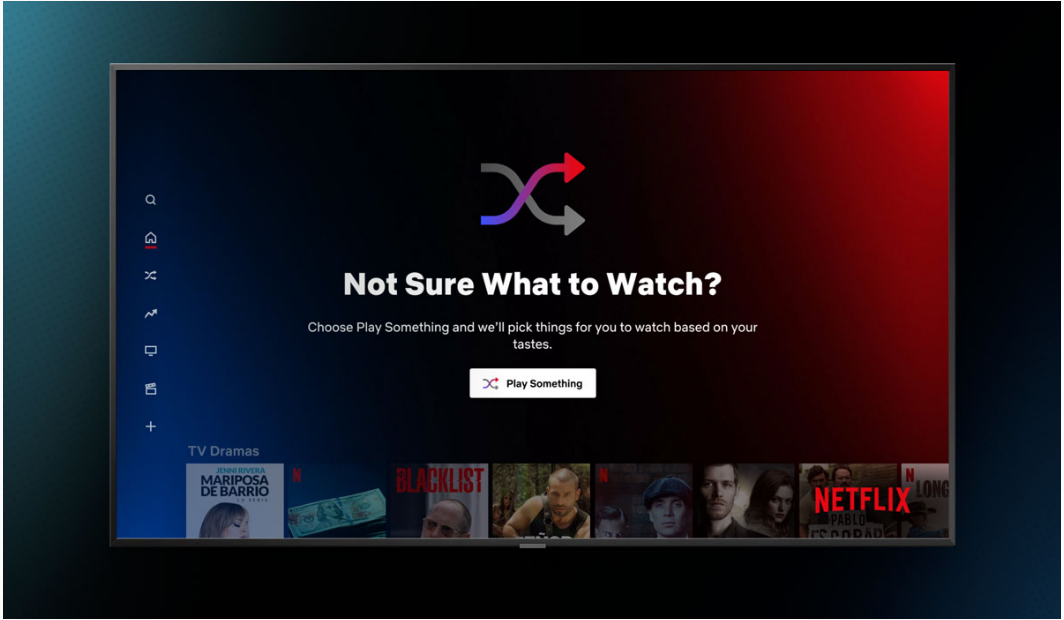
Why Find Similar Objects?

- Find similar news articles for user suggestions.
- Similar music: Spotify suggests music by finding similar users, and selecting what they listen to
- Machine learning in general (training, evaluation, actual algorithms, etc.)
- Data deduplication, etc.

Why Find Similar Objects?

- Find similar news articles for user suggestions.
- Similar music: Spotify suggests music by finding similar users, and selecting what they listen to
- Machine learning in general (training, evaluation, actual algorithms, etc.)
- Data deduplication, etc.
- “Give me a similar pair in this dataset” is a common query!

Similarity Search Example



Strategies for Similarity Search

First attempt: 1-dimensional data

92
44
7
65
60
23
80
67

- Given a list of numbers

First attempt: 1-dimensional data

92
44
7
65
60
23
80
67

- Given a list of numbers
- “Similarity" is the absolute value of the difference between them

First attempt: 1-dimensional data

92
44
7
65
60
23
80
67

- Given a list of numbers
- “Similarity" is the absolute value of the difference between them
- How can we find the closest numbers (i.e. ones with smallest difference)?

First attempt: 1-dimensional data

- How efficiently can we do this?

First attempt: 1-dimensional data

7
23
44
60
65
67
80
92

- How efficiently can we do this?
- Step 1: Sort!

First attempt: 1-dimensional data

7
23
44
60
65
67
80
92

- How efficiently can we do this?
- Step 1: Sort!
- Step 2: Scan through list, find most similar adjacent elements.

First attempt: 1-dimensional data

7
23
44
60
65
67
80
92

- How efficiently can we do this?
- Step 1: Sort!
- Step 2: Scan through list, find most similar adjacent elements.
- $O(n \log n)$ time

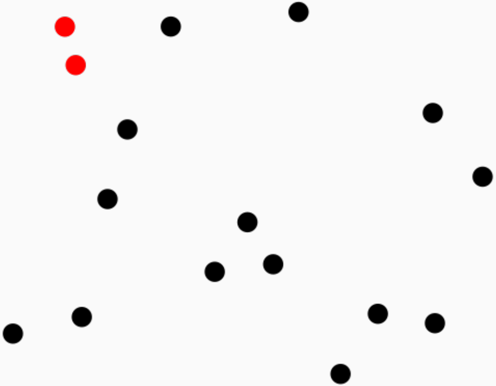
First attempt: 1-dimensional data

Aside: can we do better? Yes, there's a clever $O(n)$ algorithm based on sampling.

44
60
65
67
80
92

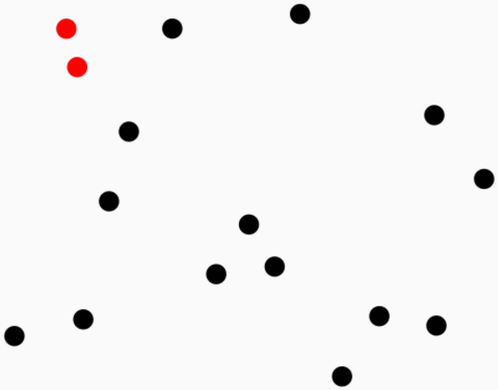
- How efficiently can we do this?
- Step 1: Sort!
- Step 2: Scan through list, find most similar adjacent elements.
- $O(n \log n)$ time

Two-dimensional Data?



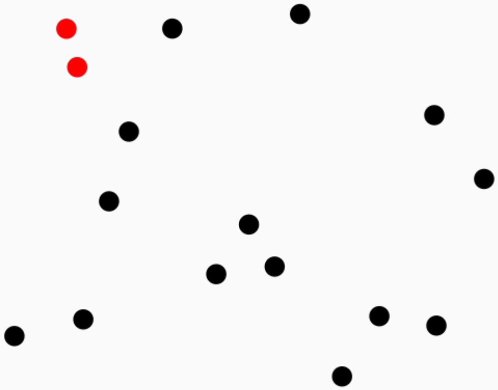
- You might have seen this in CS 256. (Not with me ☹)

Two-dimensional Data?



- You might have seen this in CS 256. (Not with me ☹)
- Divide and conquer, $O(n \log n)$ time.

Two-dimensional Data?



- You might have seen this in CS 256. (Not with me ☹)
- Divide and conquer, $O(n \log n)$ time.
- (Again, possible in $O(n)$ with sampling.)

What About Higher Dimensions?

- We want VERY high dimensions (tens, hundreds, or even millions)

What About Higher Dimensions?

- We want VERY high dimensions (tens, hundreds, or even millions)
- Songs listened to, movies watched, image tags, etc.

What About Higher Dimensions?

- We want VERY high dimensions (tens, hundreds, or even millions)
- Songs listened to, movies watched, image tags, etc.
- Words that appear in a book, k-grams that appear in a DNA sequence

What About Higher Dimensions?

- We want VERY high dimensions (tens, hundreds, or even millions)
- Songs listened to, movies watched, image tags, etc.
- Words that appear in a book, k-grams that appear in a DNA sequence
- In ML applications, often want to search in the embedded space

What I mean by “dimension”


Josefine S. (Protected by ...

+ Follow

PRO

(k) The 38 movies I saw in 2010 :)

Only counting movies I saw for the first time. Faves: 4, atm. Best: "Harry Potter and the Deathly Hallows: Part I". :D
WARNING: LIST MAY CONTAIN **SPOILERS!**

9-Jan-2010: 1. **Fish tank**

17-Jan-2010: 2. **The cove**
Fave! People = shit. I saw it with Miss C, who was rendered speechless with rage.

29-Jan-2010: 3. **The sound of insects: Record of a mummy**

1-Feb-2010: 4. **Dreamland**
Icelandic festival docu.

3-Feb-2010: 5. **A mother’s courage: Talking back to autism**

5-Feb-2010: 6. **Creation**
Charles Darwin biopic.

- I mean that each datapoint is a long vector.

What I mean by “dimension”

Josefine S. (Protected by ... + Follow

PRO

(k) The 38 movies I saw in 2010 :)

Only counting movies I saw for the first time. Faves: 4, atm. Best: "Harry Potter and the Deathly Hallows: Part I". :D
WARNING: LIST MAY CONTAIN **SPOILERS!**

9-Jan-2010: 1. **Fish tank**

17-Jan-2010: 2. **The cove**
Fave! People = shit. I saw it with Miss C, who was rendered speechless with rage.

29-Jan-2010: 3. **The sound of insects: Record of a mummy**

1-Feb-2010: 4. **Dreamland**
Icelandic festival docu.

3-Feb-2010: 5. **A mother’s courage: Talking back to autism**

5-Feb-2010: 6. **Creation**
Charles Darwin biopic.

- I mean that each datapoint is a long vector.
- “The songs this user has listened to are: [. . .]”

What I mean by “dimension”

Josefine S. (Protected by ... + Follow

PRO

(k) The 38 movies I saw in 2010 :)

Only counting movies I saw for the first time. Faves: 4, atm. Best: "Harry Potter and the Deathly Hallows: Part I". :D
WARNING: LIST MAY CONTAIN **SPOILERS!**

9-Jan-2010: 1. **Fish tank**

17-Jan-2010: 2. **The cove**
Fave! People = shit. I saw it with Miss C, who was rendered speechless with rage.

29-Jan-2010: 3. **The sound of insects: Record of a mummy**

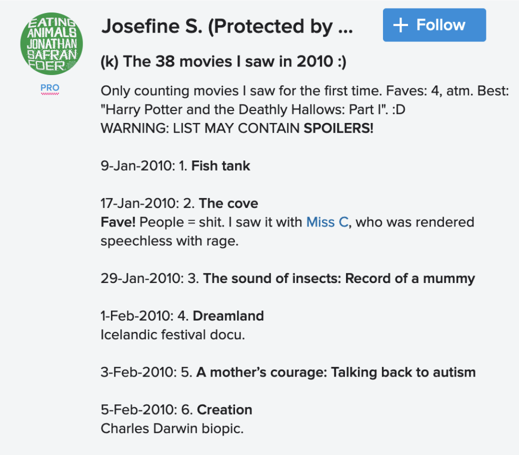
1-Feb-2010: 4. **Dreamland**
Icelandic festival docu.

3-Feb-2010: 5. **A mother’s courage: Talking back to autism**

5-Feb-2010: 6. **Creation**
Charles Darwin biopic.

- I mean that each datapoint is a long vector.
- “The songs this user has listened to are: [. . .]”
- “The movies this user has watched are: [. . .]”

What I mean by “dimension”


EATING ANIMALS JONATHAN SAFRAN FOER
PRO

Josefine S. (Protected by ... + Follow

(k) The 38 movies I saw in 2010 :)

Only counting movies I saw for the first time. Faves: 4, atm. Best: "Harry Potter and the Deathly Hallows: Part I". :D
WARNING: LIST MAY CONTAIN **SPOILERS!**

9-Jan-2010: 1. **Fish tank**

17-Jan-2010: 2. **The cove**
Fave! People = shit. I saw it with Miss C, who was rendered speechless with rage.

29-Jan-2010: 3. **The sound of insects: Record of a mummy**

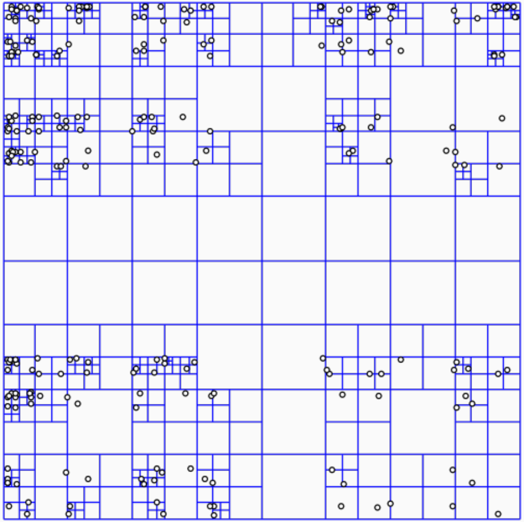
1-Feb-2010: 4. **Dreamland**
Icelandic festival docu.

3-Feb-2010: 5. **A mother’s courage: Talking back to autism**

5-Feb-2010: 6. **Creation**
Charles Darwin biopic.

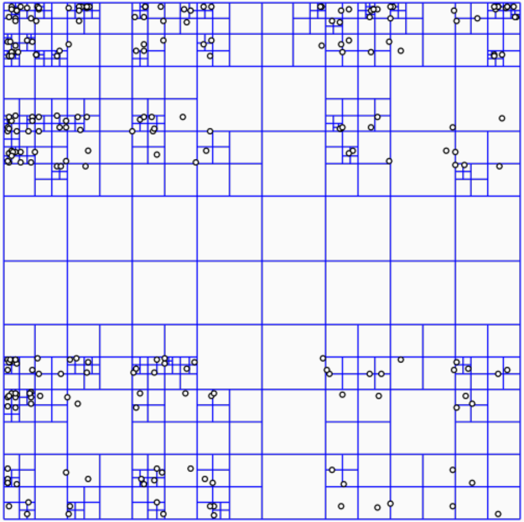
- I mean that each datapoint is a long vector.
- “The songs this user has listened to are: [. . .]”
- “The movies this user has watched are: [. . .]”
- “The tags generated for this image are: [. . .]”

How Efficient are High-dimensional Algorithms?



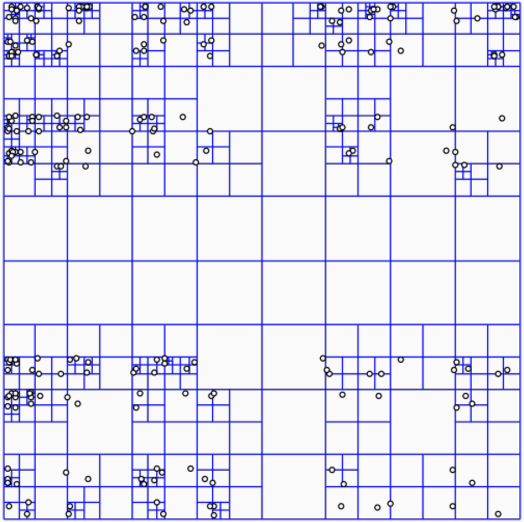
- I mentioned: $O(n \log n)$ for 1 or 2 dimensions
- In fact, can get $O(n \log n)$ for constant dimensions

How Efficient are High-dimensional Algorithms?



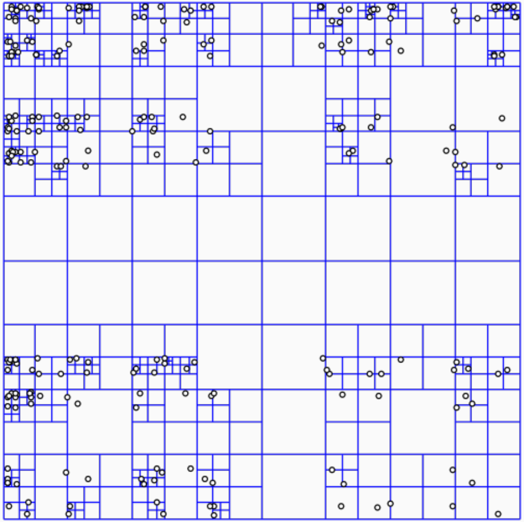
- I mentioned: $O(n \log n)$ for 1 or 2 dimensions
- In fact, can get $O(n \log n)$ for constant dimensions
- But: *exponential* time in the dimension!

How Efficient are High-dimensional Algorithms?



- I mentioned: $O(n \log n)$ for 1 or 2 dimensions
- In fact, can get $O(n \log n)$ for constant dimensions
- But: *exponential* time in the dimension!
- Something like $O(2^d \cdot n \log n)$

How Efficient are High-dimensional Algorithms?



- I mentioned: $O(n \log n)$ for 1 or 2 dimensions
- In fact, can get $O(n \log n)$ for constant dimensions
- But: *exponential* time in the dimension!
- Something like $O(2^d \cdot n \log n)$
- Worse than trying all pairs if $> \log n$ dimensions

Curse of Dimensionality

- Curse of dimensionality: Many problems have running time exponential in the dimension of the objects.

Curse of Dimensionality

- Curse of dimensionality: Many problems have running time exponential in the dimension of the objects.
- Well-known phenomenon

Curse of Dimensionality

- Curse of dimensionality: Many problems have running time exponential in the dimension of the objects.

- Well-known phenomenon

- Applies to similarity search, machine learning, combinatorics
  - Approximation techniques, like those we learn about today, are underused to a slightly shocking extent—even in ML people sometimes keep dimensionality low to avoid this issue, affecting the quality of results

Avoiding the Curse of Dimensionality

- Today we're talking about how to get efficient algorithms for arbitrarily large dimensions.

Avoiding the Curse of Dimensionality

- Today we’re talking about how to get efficient algorithms for arbitrarily large dimensions.
- Linear cost in terms of dimension (but expensive in terms of the problem size).

Avoiding the Curse of Dimensionality

- Today we're talking about how to get efficient algorithms for arbitrarily large dimensions.
- Linear cost in terms of dimension (but expensive in terms of the problem size).
- Two tools to get us there:

Avoiding the Curse of Dimensionality

- Today we're talking about how to get efficient algorithms for arbitrarily large dimensions.
- Linear cost in terms of dimension (but expensive in terms of the problem size).
- Two tools to get us there:
  - Assume that the close pair is much closer than any other (*approximate* closest pair)

Avoiding the Curse of Dimensionality

- Today we're talking about how to get efficient algorithms for arbitrarily large dimensions.
- Linear cost in terms of dimension (but expensive in terms of the problem size).
- Two tools to get us there:
  - Assume that the close pair is much closer than any other (*approximate* closest pair)

We'll come back to this later

Avoiding the Curse of Dimensionality

- Today we're talking about how to get efficient algorithms for arbitrarily large dimensions.
- Linear cost in terms of dimension (but expensive in terms of the problem size).
- Two tools to get us there:
  - Assume that the close pair is much closer than any other (*approximate* closest pair)
  - Use hashing! …A special kind of hashing

We'll come back to this later

Avoiding the Curse of Dimensionality

- Today we're talking about how to get efficient algorithms for arbitrarily large dimensions.
- Linear cost in terms of dimension (but expensive in terms of the problem size).
- Two tools to get us there:

  We'll come back to this later

  - Assume that the close pair is much closer than any other (*approximate* closest pair)
  - Use hashing! ...A special kind of hashing
- For many of these problems, random inputs are worst-case inputs
  - Worst case behavior actually occurs for many common use cases; guarantees (even approximate) can be very valuable

Locality-Sensitive Hashing

Locality-Sensitive Hashing

- Normally, hashing spreads out elements.

Locality-Sensitive Hashing

- Normally, hashing spreads out elements.
- This is key to hashing: no matter how clustered my data begins, I wind up with a nicely-distributed hash table

Locality-Sensitive Hashing

- Normally, hashing spreads out elements.
- This is key to hashing: no matter how clustered my data begins, I wind up with a nicely-distributed hash table
- Locality-sensitive hashing tries to act like a normal hash for items that are dissimilar, but *wants* collisions for similar elements

Locality-Sensitive Hashing

- Normally, hashing spreads out elements.
- This is key to hashing: no matter how clustered my data begins, I wind up with a nicely-distributed hash table
- Locality-sensitive hashing tries to act like a normal hash for items that are dissimilar, but *wants* collisions for similar elements
- Similar items are likely to wind up in the same bucket. But dissimilar items are not

Locality-Sensitive Hashing: Formal Definition

- Needs a similarity threshold r, an approximation factor $c < 1$

Locality-Sensitive Hashing: Formal Definition

- Needs a similarity threshold r, an approximation factor $c < 1$
- Two guarantees:

Locality-Sensitive Hashing: Formal Definition

- Needs a similarity threshold r, an approximation factor $c < 1$
- Two guarantees:
  - If two items x and y have similarity $\geq r$, $h(x) = h(y)$ with probability at least p_1.

Locality-Sensitive Hashing: Formal Definition

- Needs a similarity threshold r, an approximation factor $c < 1$
- Two guarantees:
  - If two items x and y have similarity $\geq r$, $h(x) = h(y)$ with probability at least p_1.
  - If two items x and y have similarity $\leq cr$, $h(x) = h(y)$ with probability at most p_2.

Locality-Sensitive Hashing: Formal Definition

- Needs a similarity threshold r, an approximation factor $c < 1$
- Two guarantees:
  - If two items x and y have similarity $\geq r$, $h(x) = h(y)$ with probability at least p_1.
  - If two items x and y have similarity $\leq cr$, $h(x) = h(y)$ with probability at most p_2.
- High level idea: close items are likely to collide. Far items are unlikely to collide.

Locality-Sensitive Hashing: Formal Definition

- Needs a similarity threshold r, an approximation factor $c < 1$
- Two guarantees:
  - If two items x and y have similarity $\geq r$, $h(x) = h(y)$ with probability at least p_1.
  - If two items x and y have similarity $\leq cr$, $h(x) = h(y)$ with probability at most p_2.
- High level idea: close items are likely to collide. Far items are unlikely to collide.
- Generally want p_2 to be about $1/n$; then we get a normal hash table for far (i.e. similarity $\leq cr$) elements.

Why Locality-Sensitive Hashing Helps

	(101, 37, 65) (103, 37, 64)	(91, 84, 3)		(100, 18, 79)
0	1	2	3	4

Ideally, close items hash to the same bucket.

Issue: Low probability of success!

- If we have $p_2 = 1/n$, then p_1 is usually very small.

Issue: Low probability of success!

We'll put numbers on this later

- If we have $p_2 = 1/n$, then p_1 is usually very small.
  - Think something like $1/\sqrt{n}$: a lot bigger than $1/n$, but nowhere near 1.

Issue: Low probability of success!

- If we have $p_2 = 1/n$, then p_1 is usually very small.
  - Think something like $1/\sqrt{n}$: a lot bigger than $1/n$, but nowhere near 1.
- How can we increase this probability?

Issue: Low probability of success!

- If we have $p_2 = 1/n$, then p_1 is usually very small.
  - Think something like $1/\sqrt{n}$: a lot bigger than $1/n$, but nowhere near 1.
- How can we increase this probability?
- Repetitions! Maintain many hash tables, each with a different locality-sensitive hash function, and try all of them.

LSH with Repetitions

0	1	2	3	4
(101, 37, 65)	(103,37,64)	(91,84,3)	(100,18,79)	

0	1	2	3	4
	(101,37,65) (103,37,64)	(91,84,3)		(100,18,79)

0	1	2	3	4
(101, 37, 65)		(103,37,64)		(91,84,3) (100,18,79)

Similarity

What Do We Mean by “Similar”?

- How can we measure the similarity of objects?

What Do We Mean by “Similar”?

- How can we measure the similarity of objects?
- Images in machine learning: often Euclidean distance (the distance we’re familiar with on a day-to-day basis)

What Do We Mean by “Similar”?

- How can we measure the similarity of objects?
- Images in machine learning: often Euclidean distance (the distance we’re familiar with on a day-to-day basis)
- What about sets?

What Do We Mean by “Similar”?

- How can we measure the similarity of objects?
- Images in machine learning: often Euclidean distance (the distance we’re familiar with on a day-to-day basis)
- What about sets?
    - Songs listened to by a user

What Do We Mean by “Similar”?

- How can we measure the similarity of objects?
- Images in machine learning: often Euclidean distance (the distance we’re familiar with on a day-to-day basis)
- What about sets?
    - Songs listened to by a user
    - Movies watched by a user

What Do We Mean by “Similar”?

- How can we measure the similarity of objects?
- Images in machine learning: often Euclidean distance (the distance we’re familiar with on a day-to-day basis)
- What about sets?
  - Songs listened to by a user
  - Movies watched by a user
  - Human-generated tags given to an image

What Do We Mean by “Similar”?

- How can we measure the similarity of objects?
- Images in machine learning: often Euclidean distance (the distance we’re familiar with on a day-to-day basis)
- What about sets?
  - Songs listened to by a user
  - Movies watched by a user
  - Human-generated tags given to an image
  - Words that appear in a document

What Do We Mean by “Similar”?

- How can we measure the similarity of objects?
- Images in machine learning: often Euclidean distance (the distance we’re familiar with on a day-to-day basis)
- What about sets?
  - Songs listened to by a user
  - Movies watched by a user
  - Human-generated tags given to an image
  - Words that appear in a document
- Need a way to measure *set similarity*

Set Similarity (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	Khalid
Khalid	Drake
Drake	Travis Scott
Travis Scott	

- When are two sets similar?

Set Similarity (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	Khalid
Khalid	Drake
Drake	Travis Scott
Travis Scott	

- When are two sets similar?
- Let's look at our two sets. Similar if they have a lot of *overlap*

Set Similarity (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	Khalid
Khalid	Drake
Drake	Travis Scott
Travis Scott	

- When are two sets similar?
- Let's look at our two sets. Similar if they have a lot of *overlap*
- I.e. lots of artists in common, compared to total artists in either list

Set Similarity (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	Khalid
Khalid	Drake
Drake	Travis Scott
Travis Scott	

Very similar!

- When are two sets similar?
- Let's look at our two sets. Similar if they have a lot of *overlap*
- I.e. lots of artists in common, compared to total artists in either list

Set Similarity (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	
Khalid	
Drake	
Travis Scott	

- When are two sets similar?
- Let's look at our two sets. Similar if they have a lot of *overlap*
- I.e. lots of artists in common, compared to total artists in either list

Set Similarity (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	
Khalid	
Drake	
Travis Scott	

Not very similar!

- When are two sets similar?
- Let's look at our two sets. Similar if they have a lot of *overlap*
- I.e. lots of artists in common, compared to total artists in either list

Set Similarity (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	Ed Sheerhan
Khalid	Drake
Drake	Travis Scott
Travis Scott	Taylor Swift

- When are two sets similar?
- Let's look at our two sets. Similar if they have a lot of *overlap*
- I.e. : lots of artists in common, compared to total artists in either list

Set Similarity (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	Ed Sheerhan
Khalid	Drake
Drake	Travis Scott
Travis Scott	Taylor Swift

Moderatly similar!

- When are two sets similar?
- Let's look at our two sets. Similar if they have a lot of *overlap*
- I.e. : lots of artists in common, compared to total artists in either list

Jaccard Similarity

Jaccard Similarity

- Similarity measure for sets A and B

Jaccard Similarity

- Similarity measure for sets A and B
- Defined as:

$$\frac{|A \cap B|}{|A \cup B|}$$

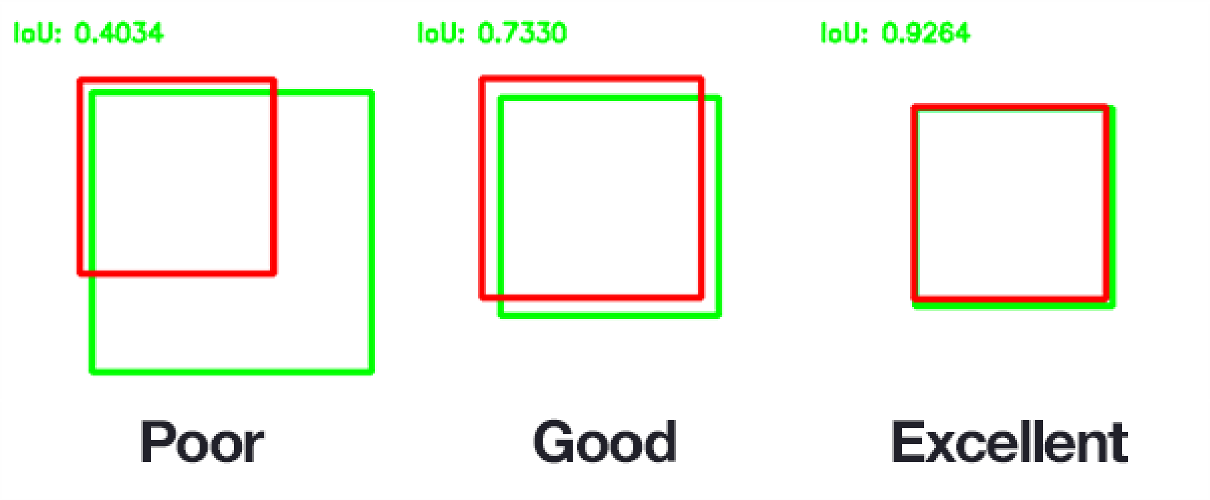
Jaccard Similarity

- Similarity measure for sets A and B
- Defined as:

$$\frac{|A \cap B|}{|A \cup B|}$$

- Intuitively: Jaccard similarity says what fraction of two sets overlaps.

Jaccard Similarity Intuition 1



Jaccard Similarity Intuition 2

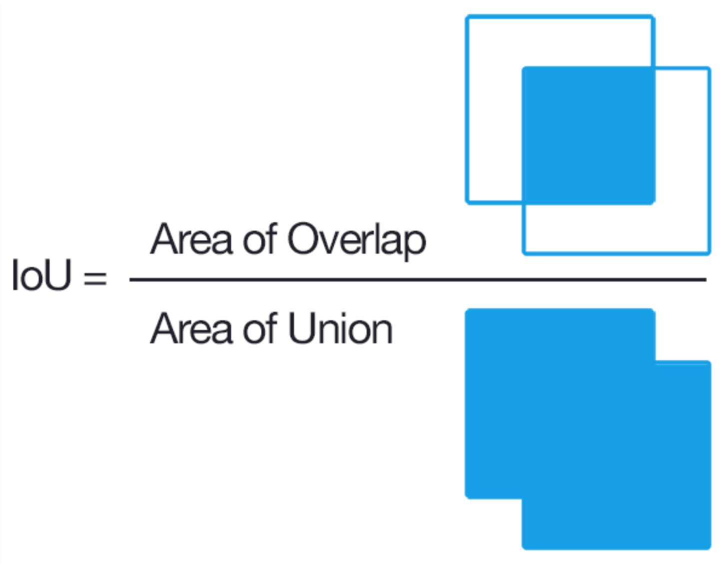
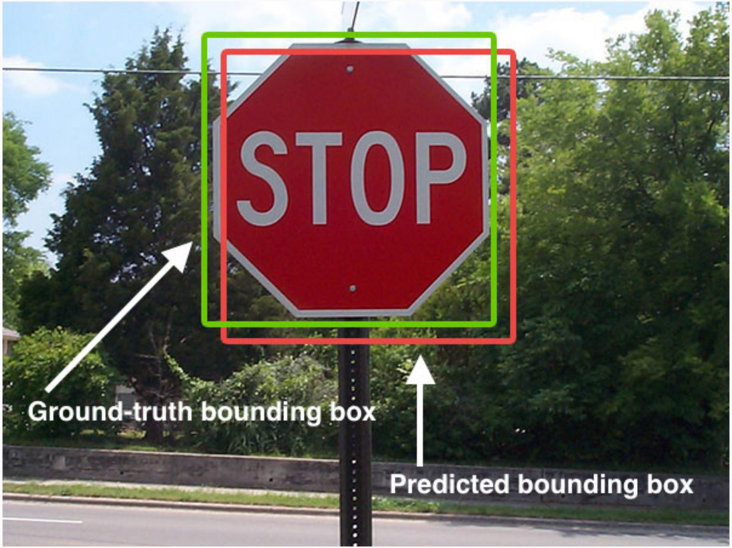


Image Search Example



Jaccard Example 1 (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	Khalid
Khalid	Drake
Drake	Travis Scott
Travis Scott	

- Similarity: $|A \cap B| / |A \cup B|$.

Jaccard Example 1 (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	Khalid
Khalid	Drake
Drake	Travis Scott
Travis Scott	

- Similarity: $|A \cap B| / |A \cup B|$.
- $|A \cap B| = 4$

Jaccard Example 1 (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	Khalid
Khalid	Drake
Drake	Travis Scott
Travis Scott	

- Similarity: $|A \cap B| / |A \cup B|$.
- $|A \cap B| = 4$
- $|A \cup B| = 5$

Jaccard Example 1 (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	Khalid
Khalid	Drake
Drake	Travis Scott
Travis Scott	

- Similarity: $|A \cap B|/|A \cup B|$.
- $|A \cap B| = 4$
- $|A \cup B| = 5$
- Jaccard Similarity: $4/5 = .8$

Set Similarity (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	
Khalid	
Drake	
Travis Scott	

- Similarity: $|A \cap B| / |A \cup B|$.

Set Similarity (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	
Khalid	
Drake	
Travis Scott	

- Similarity: $|A \cap B| / |A \cup B|$.
- $|A \cap B| = 1$

Set Similarity (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	
Khalid	
Drake	
Travis Scott	

- Similarity: $|A \cap B| / |A \cup B|$.
- $|A \cap B| = 1$
- $|A \cup B| = 5$

Set Similarity (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	
Khalid	
Drake	
Travis Scott	

- Similarity: $|A \cap B|/|A \cup B|$.
- $|A \cap B| = 1$
- $|A \cup B| = 5$
- Jaccard Similarity: $1/5 = .2$

Set Similarity (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	Ed Sheerhan
Khalid	Drake
Drake	Travis Scott
Travis Scott	Taylor Swift

- Similarity: $|A \cap B| / |A \cup B|$.

Set Similarity (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	Ed Sheerhan
Khalid	Drake
Drake	Travis Scott
Travis Scott	Taylor Swift

- Similarity: $|A \cap B| / |A \cup B|$.
- $|A \cap B| = 3$

Set Similarity (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	Ed Sheerhan
Khalid	Drake
Drake	Travis Scott
Travis Scott	Taylor Swift

- Similarity: $|A \cap B| / |A \cup B|$.
- $|A \cap B| = 3$
- $|A \cup B| = 7$

Set Similarity (Example Created in 2020)

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	Ed Sheerhan
Khalid	Drake
Drake	Travis Scott
Travis Scott	Taylor Swift

- Similarity: $|A \cap B| / |A \cup B|$.
- $|A \cap B| = 3$
- $|A \cup B| = 7$
- Jaccard Similarity: $3/7 = 0.428$

Jaccard Similarity: Properties

- Works on sets (each dimension is *binary*—an item is in the set, or not in the set)

Jaccard Similarity: Properties

- Works on sets (each dimension is *binary*—an item is in the set, or not in the set)
- Always gives a number between 0 and 1

Jaccard Similarity: Properties

- Works on sets (each dimension is *binary*—an item is in the set, or not in the set)
- Always gives a number between 0 and 1
- 1 means identical, 0 means no items in common

Jaccard Similarity: Properties

- Works on sets (each dimension is *binary*—an item is in the set, or not in the set)
- Always gives a number between 0 and 1
- 1 means identical, 0 means no items in common
- Jaccard similarity ignores items not in either set. So we learn nothing if neither of us like an artist. (Is this good?)

Jaccard Similarity: Properties

- Works on sets (each dimension is *binary*—an item is in the set, or not in the set)
- Always gives a number between 0 and 1
- 1 means identical, 0 means no items in common
- Jaccard similarity ignores items not in either set. So we learn nothing if neither of us like an artist. (Is this good?)
- Still works if one list is much longer than the other. (Generally, they'll have small similarity)

Locality-Sensitive Hash for Jaccard Similarity

- Want: items with high Jaccard Similarity are likely to hash together

Locality-Sensitive Hash for Jaccard Similarity

- Want: items with high Jaccard Similarity are likely to hash together
- Items with low Jaccard Similarity are *un*likely to hash together

Locality-Sensitive Hash for Jaccard Similarity

- Want: items with high Jaccard Similarity are likely to hash together
- Items with low Jaccard Similarity are *un*likely to hash together
- Classic method: MinHash

MinHash

MinHash

- Developed by Andrei Broder in 1997 while working at AltaVista

MinHash

- Developed by Andrei Broder in 1997 while working at AltaVista
- (AltaVista was probably the most popular search engine before Google, they wanted to detect similar web pages to eliminate them from search results)

MinHash

- Developed by Andrei Broder in 1997 while working at AltaVista
- (AltaVista was probably the most popular search engine before Google, they wanted to detect similar web pages to eliminate them from search results)
- Now used for similarity search, database joins, clustering—LOTS of things.

AltaVista in 2001

Bit Vectors

- Can represent any set as a vector of bits

Bit Vectors

- Can represent any set as a vector of bits

- Each bit is an item. “1” means that that item is in the set, “0” means it’s not

Bit Vectors

- Can represent any set as a vector of bits
- Each bit is an item. “1” means that that item is in the set, “0” means it’s not
- So if I’m keeping track of different people’s favorite colors, my universe may be {red, yellow, blue, green, purple, orange}

Bit Vectors

- Can represent any set as a vector of bits
- Each bit is an item. “1” means that that item is in the set, “0” means it’s not
- So if I’m keeping track of different people’s favorite colors, my universe may be {red, yellow, blue, green, purple, orange}
- If someone likes red and blue, we can store that information as 101000.

Bit Vectors

- Can represent any set as a vector of bits
- Each bit is an item. “1” means that that item is in the set, “0” means it’s not
- So if I’m keeping track of different people’s favorite colors, my universe may be {red, yellow, blue, green, purple, orange}
- If someone likes red and blue, we can store that information as `101000`.
- Effective if universe is fairly small; use a list for larger universe

Bit Vectors: Jaccard Similarity

- How can we determine $A \cap B$?

Bit Vectors: Jaccard Similarity

- How can we determine $A \cap B$?
    - This is exactly A & B in C-style notation

Bit Vectors: Jaccard Similarity

- How can we determine $A \cap B$?
    - This is exactly A & B in C-style notation
- What about $A \cup B$?

Bit Vectors: Jaccard Similarity

- How can we determine $A \cap B$?
  - This is exactly A & B in C-style notation
- What about $A \cup B$?
  - This is exactly A | B in C-style notation

Bit Vectors: Jaccard Similarity

- How can we determine $A \cap B$?
    - This is exactly A & B in C-style notation
- What about $A \cup B$?
    - This is exactly A | B in C-style notation
- We want the size of these sets—need to count the number of 1s in A & B, or A | B.

MinHash

- The hash consists of a *permutation* of all possible items in the universe
  - $\{0, \ldots, 127\}$ in the assignment

MinHash

- The hash consists of a *permutation* of all possible items in the universe
  - $\{0, \ldots, 127\}$ in the assignment

- To hash a set A: find the first item in A in the order given by the permutation. That item is the hash value!

MinHash example

- Let’s stick with favorite colors, out of {red, yellow, blue, green, purple, orange}

MinHash example

- Let's stick with favorite colors, out of {red, yellow, blue, green, purple, orange}
- To hash, we randomly permute them. Let's say our current hash is given by the permutation (blue, orange, green, purple, red, yellow). Some examples:

MinHash example

- Let's stick with favorite colors, out of {red, yellow, blue, green, purple, orange}
- To hash, we randomly permute them. Let's say our current hash is given by the permutation (blue, orange, green, purple, red, yellow). Some examples:
- First set is `101000` (same as {red, blue}).

MinHash example

- Let's stick with favorite colors, out of {red, yellow, blue, green, purple, orange}
- To hash, we randomly permute them. Let's say our current hash is given by the permutation (blue, orange, green, purple, red, yellow). Some examples:
- First set is `101000` (same as {red, blue}). blue is in the set, so the hash value is blue.

MinHash example

- Let’s stick with favorite colors, out of {red, yellow, blue, green, purple, orange}
- To hash, we randomly permute them. Let’s say our current hash is given by the permutation (blue, orange, green, purple, red, yellow). Some examples:
- First set is 101000 (same as {red, blue}). blue is in the set, so the hash value is blue.
- Second set is 110010 (we could also write {red, yellow, purple}).

MinHash example

- Let's stick with favorite colors, out of {red, yellow, blue, green, purple, orange}
- To hash, we randomly permute them. Let's say our current hash is given by the permutation (blue, orange, green, purple, red, yellow). Some examples:
- First set is 101000 (same as {red, blue}). blue is in the set, so the hash value is blue.
- Second set is 110010 (we could also write {red, yellow, purple}). blue is not in the set; nor is orange; nor is green. purple is, so purple is the hash value

MinHash for Bit Vectors

- On the assignment, have bit vectors of length 128

MinHash for Bit Vectors

- On the assignment, have bit vectors of length 128
- To get a hash function, we need a random permutation of the indices of these bits. That is to say, a random permutation of $\{0, 1, 2, \ldots, 127\}$

MinHash for Bit Vectors

- On the assignment, have bit vectors of length 128
- To get a hash function, we need a random permutation of the indices of these bits. That is to say, a random permutation of $\{0, 1, 2, \ldots, 127\}$
- To hash an item x, go through the random permutation. Find the first index i in the list such that the ith bit of x is 1.

MinHash for Bit Vectors

- On the assignment, have bit vectors of length 128
- To get a hash function, we need a random permutation of the indices of these bits. That is to say, a random permutation of $\{0, 1, 2, \ldots, 127\}$
- To hash an item x, go through the random permutation. Find the first index i in the list such that the ith bit of x is 1.
- Let's say $x = 10100101$, and the permutation is $(1, 5, 2, 0, 7, 6, 4, 3)$.

MinHash for Bit Vectors

- On the assignment, have bit vectors of length 128
- To get a hash function, we need a random permutation of the indices of these bits. That is to say, a random permutation of $\{0, 1, 2, \ldots, 127\}$
- To hash an item x, go through the random permutation. Find the first index i in the list such that the ith bit of x is 1.
- Let's say $x = 10100101$, and the permutation is $(1, 5, 2, 0, 7, 6, 4, 3)$.

For the sake of space let's do 8 bits

MinHash for Bit Vectors

- On the assignment, have bit vectors of length 128
- To get a hash function, we need a random permutation of the indices of these bits. That is to say, a random permutation of $\{0, 1, 2, \ldots, 127\}$
- To hash an item x, go through the random permutation. Find the first index i in the list such that the ith bit of x is 1.
- Let's say $x = 10100101$, and the permutation is $(1, 5, 2, 0, 7, 6, 4, 3)$.
  - On your own: what is the minhash of x for this permutation?

MinHash for Bit Vectors

- On the assignment, have bit vectors of length 128

- To get a hash function, we need a random permutation of the indices of these bits. That is to say, a random permutation of $\{0, 1, 2, \ldots, 127\}$

- To hash an item x, go through the random permutation. Find the first index i in the list such that the ith bit of x is 1.

- Let's say $x = 10100101$, and the permutation is $(1, 5, 2, 0, 7, 6, 4, 3)$.
  - On your own: what is the minhash of x for this permutation?

- The minhash of x is 5.

MinHash

- A *single* MinHash: hashes each set to one of its elements (i.e. the position of one of its one bits)

MinHash

- A *single* MinHash: hashes each set to one of its elements (i.e. the position of one of its one bits)
- What happens when we store elements in buckets according to this hash table?

MinHash

- A *single* MinHash: hashes each set to one of its elements (i.e. the position of one of its one bits)
- What happens when we store elements in buckets according to this hash table?
- Not useful yet—output is too small! Almost all items will have one of the first few items in the permutation, so will hash to the first few buckets

MinHash

- A *single* MinHash: hashes each set to one of its elements (i.e. the position of one of its one bits)
- What happens when we store elements in buckets according to this hash table?
- Not useful yet—output is too small! Almost all items will have one of the first few items in the permutation, so will hash to the first few buckets
- Let's do some analysis to look at this issue in more detail

Analysis of Basic MinHash

Analysis

- What is the probability that $h(A) = h(B)$?

Analysis

- What is the probability that $h(A) = h(B)$?
- Let's look at the permutation that defines h. We can ignore any item that is not in A or B (why?)

Analysis

- What is the probability that $h(A) = h(B)$?
- Let's look at the permutation that defines h. We can ignore any item that is not in A or B (why?)
- Look at the first index in the permutation that is in A or B (i.e. it is in $A \cup B$)

Analysis

- What is the probability that $h(A) = h(B)$?
- Let's look at the permutation that defines h. We can ignore any item that is not in A or B (why?)
- Look at the first index in the permutation that is in A or B (i.e. it is in $A \cup B$)
    - If this index is in *both* A and B, then $h(A) = h(B)$

Analysis

- What is the probability that $h(A) = h(B)$?
- Let's look at the permutation that defines h. We can ignore any item that is not in A or B (why?)
- Look at the first index in the permutation that is in A or B (i.e. it is in $A \cup B$)
  - If this index is in *both* A and B, then $h(A) = h(B)$
  - If this index is in only one of A or B, then $h(A) \neq h(B)$

Analysis

- What is the probability that $h(A) = h(B)$?
- Let's look at the permutation that defines h. We can ignore any item that is not in A or B (why?)
- Look at the first index in the permutation that is in A or B (i.e. it is in $A \cup B$)
  - If this index is in *both* A and B, then $h(A) = h(B)$
  - If this index is in only one of A or B, then $h(A) \neq h(B)$
- Any index in $A \cup B$ is equally likely to be first. If the index is in $A \cap B$, they hash together; otherwise they do not

Analysis

- What is the probability that $h(A) = h(B)$?
- Let's look at the permutation that defines h. We can ignore any item that is not in A or B (why?)
- Look at the first index in the permutation that is in A or B (i.e. it is in $A \cup B$)
    - If this index is in *both* A and B, then $h(A) = h(B)$
    - If this index is in only one of A or B, then $h(A) \neq h(B)$
- Any index in $A \cup B$ is equally likely to be first. If the index is in $A \cap B$, they hash together; otherwise they do not
- Therefore: probability of hashing together is $|A \cap B|/|A \cup B|$.

MinHash as an LSH

- This means MinHash is an LSH!

MinHash as an LSH

- This means MinHash is an LSH!
- If two items have similarity at least r, they collide with probability at least $p_1 = r$

MinHash as an LSH

- This means MinHash is an LSH!
- If two items have similarity at least r, they collide with probability at least $p_1 = r$
- If two items have similarity at most cr, they collide with probability at most $p_2 = cr$

Analysis: Phrased as bit vectors

- What is the probability that $h(A) = h(B)$?
- Let's look at the permutation that defines h. We can ignore any index that is 0 in both A and B.
- Look at the first index in the permutation that is 1 in A or B
  - If this index is in *both* A and B, then $h(A) = h(B)$
  - If this index is in only one of A or B, then $h(A) \neq h(B)$
- Any index that is 1 in $A|B$ is equally likely to be first. If the index is in $A\&B$, they hash together; otherwise they do not
- Therefore: probability of hashing together is (number of 1s in $A\&B$)/(number of 1s in $A|B$).

Analysis Example

- Let's say we have $A = \{\text{red, blue, green}\}$ and $B = \{\text{red, orange, purple, green}\}$.

Analysis Example

- Let's say we have $A = \{\text{red, blue, green}\}$ and $B = \{\text{red, orange, purple, green}\}$.
- When do A and B hash together?

Analysis Example

- Let's say we have $A = \{\text{red, blue, green}\}$ and $B = \{\text{red, orange, purple, green}\}$.
- When do A and B hash together?
- If red or green appears before blue, orange, and purple then they hash together

Analysis Example

- Let's say we have $A = \{\text{red, blue, green}\}$ and $B = \{\text{red, orange, purple, green}\}$.
- When do A and B hash together?
- If red or green appears before blue, orange, and purple then they hash together
- If blue or orange or purple appear before red and green, then they don't hash together

Analysis Example

- Let's say we have $A = \{\text{red, blue, green}\}$ and $B = \{\text{red, orange, purple, green}\}$.
- When do A and B hash together?
- If red or green appears before blue, orange, and purple then they hash together
- If blue or orange or purple appear before red and green, then they don't hash together
- Probability that red or green is first out of {red, blue, green, orange, purple} is $2/5$.

Analysis Example

- Let's say we have $A = \{\text{red, blue, green}\}$ and $B = \{\text{red, orange, purple, green}\}$.
- When do A and B hash together?
- If red or green appears before blue, orange, and purple then they hash together
- If blue or orange or purple appear before red and green, then they don't hash together
- Probability that red or green is first out of {red, blue, green, orange, purple} is $2/5$.
- Therefore, A and B hash together with probability $2/5$.

Making Sure We Find the Close Pair

- To find the close pair, compare all pairs of items that hash to the same value

Making Sure We Find the Close Pair

- To find the close pair, compare all pairs of items that hash to the same value
    - (We’ll talk about how to do this in a moment)

Making Sure We Find the Close Pair

- To find the close pair, compare all pairs of items that hash to the same value
    - (We’ll talk about how to do this in a moment)
- Let’s say our close pair has similarity .5. How many times do we need to repeat?

Making Sure We Find the Close Pair

- To find the close pair, compare all pairs of items that hash to the same value
  - (We'll talk about how to do this in a moment)
- Let's say our close pair has similarity .5. How many times do we need to repeat?
- Each repetition has the close pair in the same bucket with probability .5. So need 2 repetitions in expectation.

An Aside on Expectation

Lemma

If a random process succeeds with probability p, then in expectation it takes $1/p$ iterations of the process before success.

Examples:

An Aside on Expectation

Lemma

If a random process succeeds with probability p, then in expectation it takes $1/p$ iterations of the process before success.

Examples:

- It takes two coin flips in expectation before we see a heads

An Aside on Expectation

Lemma

If a random process succeeds with probability p, then in expectation it takes $1/p$ iterations of the process before success.

Examples:

- It takes two coin flips in expectation before we see a heads
- We need to roll a 6-sided die 6 times before we see (say) a three.

An Aside on Expectation

Lemma

If a random process succeeds with probability p, then in expectation it takes $1/p$ iterations of the process before success.

Examples:

- It takes two coin flips in expectation before we see a heads
- We need to roll a 6-sided die 6 times before we see (say) a three.

Proof: the expectation is

$$\sum_{i=1}^{\infty} ip(1-p)^{i-1} = p\sum_{i=1}^{\infty} i(1-p)^{i-1} = p\frac{1}{(1-(1-p))^2} = \frac{1}{p}.$$

Concatenations and Repetitions

Problems with this Approach

- Buckets are really big!! (After all, lots of items are pretty likely to have a given bit set.)

Problems with this Approach

- Buckets are really big!! (After all, lots of items are pretty likely to have a given bit set.)
- How can we decrease the probability that items hash together?

Problems with this Approach

- Buckets are really big!! (After all, lots of items are pretty likely to have a given bit set.)
- How can we decrease the probability that items hash together?
- Answer: concatenate multiple hashes together.

Concatenating Hashes

- Rather than one hash h, concatenate k independent hashes $h_1, h_2, \ldots h_k$, each with its own permutation $P_1, P_2, \ldots P_k$.

Concatenating Hashes

- Rather than one hash h, concatenate k independent hashes $h_1, h_2, \ldots h_k$, each with its own permutation $P_1, P_2, \ldots P_k$.
- To hash an item: repeat the process of searching through the permutation for each hash. Then concatenate the results together (can just use string concatenation)

Concatenating Hashes

- Rather than one hash h, concatenate k independent hashes $h_1, h_2, \ldots h_k$, each with its own permutation $P_1, P_2, \ldots P_k$.
- To hash an item: repeat the process of searching through the permutation for each hash. Then concatenate the results together (can just use string concatenation)
- How does this affect the probability for sets A and B?

Concatenating Hashes

- Rather than one hash h, concatenate k independent hashes $h_1, h_2, \ldots h_k$, each with its own permutation $P_1, P_2, \ldots P_k$.
- To hash an item: repeat the process of searching through the permutation for each hash. Then concatenate the results together (can just use string concatenation)
- How does this affect the probability for sets A and B?
    - For each of the k independent hashes, A and B collide with probability $|A \cap B|/|A \cup B|$.

Concatenating Hashes

- Rather than one hash h, concatenate k independent hashes $h_1, h_2, \ldots h_k$, each with its own permutation $P_1, P_2, \ldots P_k$.
- To hash an item: repeat the process of searching through the permutation for each hash. Then concatenate the results together (can just use string concatenation)
- How does this affect the probability for sets A and B?
  - For each of the k independent hashes, A and B collide with probability $|A \cap B|/|A \cup B|$.
  - We only obtain the same concatenated hashes if *all* of the hashes are the same.

Concatenating Hashes

- Rather than one hash h, concatenate k independent hashes $h_1, h_2, \ldots h_k$, each with its own permutation $P_1, P_2, \ldots P_k$.
- To hash an item: repeat the process of searching through the permutation for each hash. Then concatenate the results together (can just use string concatenation)
- How does this affect the probability for sets A and B?
  - For each of the k independent hashes, A and B collide with probability $|A \cap B|/|A \cup B|$.
  - We only obtain the same concatenated hashes if *all* of the hashes are the same.
  - They are independent, so we can multiply to obtain probability $(|A \cap B|/|A \cup B|)^k$ of A and B colliding.

Concatenation Example

- Let's say we have $A = \{\text{red}, \text{blue}\}$ and $B = \{\text{red}, \text{orange}\}$, and $k = 3$.

Concatenation Example

- Let's say we have $A = \{\text{red, blue}\}$ and $B = \{\text{red, orange}\}$, and $k = 3$.
- $P_1 = \{\text{red, green, blue, orange}\}$,$P_2 = \{\text{orange, green, blue, red}\}$,$P_3 = \{\text{red, green, blue, orange}\}$

Concatenation Example

- Let's say we have $A = \{\text{red, blue}\}$ and $B = \{\text{red, orange}\}$, and $k = 3$.
- $P_1 = \{\text{red, green, blue, orange}\}$,$P_2 = \{\text{orange, green, blue, red}\}$,$P_3 = \{\text{red, green, blue, orange}\}$
- Let's hash A.

Concatenation Example

- Let's say we have $A = \{\text{red, blue}\}$ and $B = \{\text{red, orange}\}$, and $k = 3$.
- $P_1 = \{\text{red, green, blue, orange}\}$,$P_2 = \{\text{orange, green, blue, red}\}$,$P_3 = \{\text{red, green, blue, orange}\}$
- Let's hash A.
    - First hash: red is in A.

Concatenation Example

- Let's say we have $A = \{\text{red, blue}\}$ and $B = \{\text{red, orange}\}$, and $k = 3$.
- $P_1 = \{\text{red, green, blue, orange}\}$,$P_2 = \{\text{orange, green, blue, red}\}$,$P_3 = \{\text{red, green, blue, orange}\}$
- Let's hash A.
  - First hash: red is in A.
  - Second hash: orange not in A, nor is green. Blue is in A.

Concatenation Example

- Let's say we have $A = \{\text{red, blue}\}$ and $B = \{\text{red, orange}\}$, and $k = 3$.
- $P_1 = \{\text{red, green, blue, orange}\}$,$P_2 = \{\text{orange, green, blue, red}\}$,$P_3 = \{\text{red, green, blue, orange}\}$
- Let's hash A.
  - First hash: red is in A.
  - Second hash: orange not in A, nor is green. Blue is in A.
  - Third hash: red is in A.

Concatenation Example

- Let's say we have $A = \{\text{red, blue}\}$ and $B = \{\text{red, orange}\}$, and $k = 3$.
- $P_1 = \{\text{red, green, blue, orange}\}$,$P_2 = \{\text{orange, green, blue, red}\}$,$P_3 = \{\text{red, green, blue, orange}\}$
- Let's hash A.
  - First hash: red is in A.
  - Second hash: orange not in A, nor is green. Blue is in A.
  - Third hash: red is in A.
- Concatenating, we have $h(A) = \text{redbluered}$

Concatenation Example

- Let's say we have $A = \{\text{red}, \text{blue}\}$ and $B = \{\text{red}, \text{orange}\}$, and $k = 3$.

Concatenation Example

- Let's say we have $A = \{\text{red, blue}\}$ and $B = \{\text{red, orange}\}$, and $k = 3$.
- $P_1 = \{\text{red, green, blue, orange}\}$,$P_2 = \{\text{orange, green, blue, red}\}$,$P_3 = \{\text{red, green, blue, orange}\}$
- Let's hash B.

Concatenation Example

- Let's say we have $A = \{\text{red, blue}\}$ and $B = \{\text{red, orange}\}$, and $k = 3$.
- $P_1 = \{\text{red, green, blue, orange}\}$,$P_2 = \{\text{orange, green, blue, red}\}$,$P_3 = \{\text{red, green, blue, orange}\}$
- Let's hash B.
    - First hash: red is in B.

Concatenation Example

- Let's say we have $A = \{\text{red, blue}\}$ and $B = \{\text{red, orange}\}$, and $k = 3$.
- $P_1 = \{\text{red, green, blue, orange}\}$,$P_2 = \{\text{orange, green, blue, red}\}$,$P_3 = \{\text{red, green, blue, orange}\}$
- Let's hash B.
  - First hash: red is in B.
  - Second hash: orange is in B.

Concatenation Example

- Let's say we have $A = \{\text{red, blue}\}$ and $B = \{\text{red, orange}\}$, and $k = 3$.
- $P_1 = \{\text{red, green, blue, orange}\}$,$P_2 = \{\text{orange, green, blue, red}\}$,$P_3 = \{\text{red, green, blue, orange}\}$
- Let's hash B.
    - First hash: red is in B.
    - Second hash: orange is in B.
    - Third hash: red is in B.

Concatenation Example

- Let's say we have $A = \{\text{red, blue}\}$ and $B = \{\text{red, orange}\}$, and $k = 3$.
- $P_1 = \{\text{red, green, blue, orange}\}$,$P_2 = \{\text{orange, green, blue, red}\}$,$P_3 = \{\text{red, green, blue, orange}\}$
- Let's hash B.
  - First hash: red is in B.
  - Second hash: orange is in B.
  - Third hash: red is in B.
- Concatenating, we have $h(B) = \text{redorangered}$

Putting it all Together

- For each hash table, we concatenate k hashes to obtain a *signature*

Putting it all Together

- For each hash table, we concatenate k hashes to obtain a *signature*
- Hash the signature of each item to obtain a bucket to place the item in

Putting it all Together

- For each hash table, we concatenate k hashes to obtain a *signature*
- Hash the signature of each item to obtain a bucket to place the item in
- Check every pair of items in each bucket and see if it's the closest

Putting it all Together

- For each hash table, we concatenate k hashes to obtain a *signature*
- Hash the signature of each item to obtain a bucket to place the item in
- Check every pair of items in each bucket and see if it's the closest
- Quite often we'll get unlucky and the close pair won't be in the same bucket. What can we do?

Putting it all Together

- For each hash table, we concatenate k hashes to obtain a *signature*
- Hash the signature of each item to obtain a bucket to place the item in
- Check every pair of items in each bucket and see if it's the closest
- Quite often we'll get unlucky and the close pair won't be in the same bucket. What can we do?
- Need to repeat all of that multiple times until we find the close pair (let's say we repeat R times)

Putting it all Together

- For each hash table, we concatenate k hashes to obtain a *signature*
- Hash the signature of each item to obtain a bucket to place the item in
- Check every pair of items in each bucket and see if it's the closest
- Quite often we'll get unlucky and the close pair won't be in the same bucket. What can we do?
- Need to repeat all of that multiple times until we find the close pair (let's say we repeat R times)
- So: overall need kR permutations

Putting it all Together

- For each hash table, we concatenate k hashes to obtain a *signature*
- Hash the signature of each item to obtain a bucket to place the item in
- Check every pair of items in each bucket and see if it's the closest
- Quite often we'll get unlucky and the close pair won't be in the same bucket. What can we do?
- Need to repeat all of that multiple times until we find the close pair (let's say we repeat R times)
- So: overall need kR permutations
- What kind of values work for k and R?

Putting it Together: Analysis

- Let's say we have a set of n items $x_1, \ldots, x_n$

Putting it Together: Analysis

- Let's say we have a set of n items $x_1, \ldots, x_n$
- The close pair of items has Jaccard similarity $3/4$

Putting it Together: Analysis

- Let's say we have a set of n items $x_1, \ldots, x_n$
- The close pair of items has Jaccard similarity $3/4$
- Every other pair of items has similarity $1/3$

Putting it Together: Analysis

- Let's say we have a set of n items $x_1, \ldots, x_n$
- The close pair of items has Jaccard similarity $3/4$
- Every other pair of items has similarity $1/3$
- How should we set k? How many repetitions R is it likely to take?

Putting it Together: Analysis (Finding k)

- Non-similar pairs have similarity $1/3$

Putting it Together: Analysis (Finding k)

- Non-similar pairs have similarity $1/3$
- We want buckets to be small (have $O(1)$ size)

Putting it Together: Analysis (Finding k)

- Non-similar pairs have similarity $1/3$
- We want buckets to be small (have $O(1)$ size)
- Look at an element x_i. What is the expected size of its bucket?

Putting it Together: Analysis (Finding k)

- Non-similar pairs have similarity $1/3$
- We want buckets to be small (have $O(1)$ size)
- Look at an element x_i. What is the expected size of its bucket?
- $\sum_{j \neq i} (1/3)^k$ (since x_i and any x_j with $j \neq i$ share a hash value with probability $1/3^k$)

Putting it Together: Analysis (Finding k)

- Non-similar pairs have similarity $1/3$
- We want buckets to be small (have $O(1)$ size)
- Look at an element x_i. What is the expected size of its bucket?
- $\sum_{j \neq i} (1/3)^k$ (since x_i and any x_j with $j \neq i$ share a hash value with probability $1/3^k$)
- We can then solve $(n-1)(1/3)^k = 1$ to get $k = \log_3(n-1)$.

Putting it Together: Analysis (Predicting R)

- The similar pair has Jaccard similarity .75

Putting it Together: Analysis (Predicting R)

- The similar pair has Jaccard similarity .75
- So they are in the same bucket with probability $(.75)^k$

Putting it Together: Analysis (Predicting R)

- The similar pair has Jaccard similarity .75
- So they are in the same bucket with probability $(.75)^k$
- We have $k = (\log_3 n - 1)$. So....we need to do some algebra. (Let's assume that k is already an integer)

Putting it Together: Analysis (Predicting R)

- $(.75)^{\log_3(n-1)}$

Putting it Together: Analysis (Predicting R)

- $(.75)^{\log_3(n-1)}$
- $(.75)^{\log_3(n-1)} = 2^{\log_2(n-1)\log_2(3/4)/\log(3)}$

Putting it Together: Analysis (Predicting R)

- $(.75)^{\log_3(n-1)}$
- $(.75)^{\log_3(n-1)} = 2^{\log_2(n-1)\log_2(3/4)/\log(3)}$
- $2^{\log_2(n-1)\log_2(3/4)/\log(3)} = (n-1)^{\log(3/4)/\log(3)}$

When dealing with tricky log/exponent expressions, try putting everything as a power of 2.

Putting it Together: Analysis (Predicting R)

- $(.75)^{\log_3(n-1)}$
- $(.75)^{\log_3(n-1)} = 2^{\log_2(n-1)\log_2(3/4)/\log(3)}$
- $2^{\log_2(n-1)\log_2(3/4)/\log(3)} = (n-1)^{\log(3/4)/\log(3)}$
- $\approx n^{\log(3/4)/\log(3)} = 1/n^{.26}$

Putting it Together: Analysis (Predicting R)

- The similar pair has Jaccard similarity .75
- So they are in the same bucket with probability $(.75)^k$
- We have $k = (\log_3 n - 1)$. So....we need to do some algebra. (Let's assume that k is already an integer)
- $(.75)^{\log_3(n-1)} = 2^{\log_2(n-1)\log_2(3/4)/\log(3)} = (n-1)^{\log(3/4)/\log(3)} \approx 1/n^{.26}$

Putting it Together: Analysis (Predicting R)

- The similar pair has Jaccard similarity .75
- So they are in the same bucket with probability $(.75)^k$
- We have $k = (\log_3 n - 1)$. So....we need to do some algebra. (Let's assume that k is already an integer)
- $(.75)^{\log_3(n-1)} = 2^{\log_2(n-1)\log_2(3/4)/\log(3)} = (n-1)^{\log(3/4)/\log(3)} \approx 1/n^{.26}$
- So we expect about $R = n^{.26}$ repetitions. That's a lot!

Putting it Together: Analysis (Predicting R)

- The similar pair has Jaccard similarity .75
- So they are in the same bucket with probability $(.75)^k$
- We have $k = (\log_3 n - 1)$. So....we need to do some algebra. (Let's assume that k is already an integer)
- $(.75)^{\log_3(n-1)} = 2^{\log_2(n-1)\log_2(3/4)/\log(3)} = (n-1)^{\log(3/4)/\log(3)} \approx 1/n^{.26}$
- So we expect about $R = n^{.26}$ repetitions. That's a lot!
- But it's not far from the state of the art.

Putting it Together: Analysis (Predicting R)

- The similar pair has Jaccard similarity .75
- So they are in the same bucket with probability $(.75)^k$
- We have $k = (\log_3 n - 1)$. So....we need to do some algebra. (Let's assume that k is already an integer)
- $(.75)^{\log_3(n-1)} = 2^{\log_2(n-1)\log_2(3/4)/\log(3)} = (n-1)^{\log(3/4)/\log(3)} \approx 1/n^{.26}$
- So we expect about $R = n^{.26}$ repetitions. That's a lot!
- But it's not far from the state of the art.
- And way better than brute force!

Finding R and k in general

Let's say we have n points where the close pairs have similarity j_1, and all other pairs have similarity at most j_2

- First, set k so that each bucket has size $O(1)$: $k = \log_{1/j_2} n$.

Finding R and k in general

Let's say we have n points where the close pairs have similarity j_1, and all other pairs have similarity at most j_2

- First, set k so that each bucket has size $O(1)$: $k = \log_{1/j_2} n$.
  - Doable at home: show that this is the optimal value for k using the below analysis.

Finding R and k in general

Let's say we have n points where the close pairs have similarity j_1, and all other pairs have similarity at most j_2

- First, set k so that each bucket has size $O(1)$: $k = \log_{1/j_2} n$.
  - Doable at home: show that this is the optimal value for k using the below analysis.
- Then, number of R we need in expectation is:

$$\left(\frac{1}{j_1}\right)^k = \left(\frac{1}{j_1}\right)^{\log_{1/j_2} n} = n^{\log_{(1/j_2)}(1/j_1)}.$$

Plan for Assignment 2

- Until we find the close pair of items:

Plan for Assignment 2

- Until we find the close pair of items:
- Hash all n items using MinHash

Plan for Assignment 2

- Until we find the close pair of items:
- Hash all n items using MinHash
- For each bucket, compare all pair of items in the bucket to see if they are close. If a close pair is found, return return it

Plan for Assignment 2

- Until we find the close pair of items:
- Hash all n items using MinHash
- For each bucket, compare all pair of items in the bucket to see if they are close. If a close pair is found, return return it
- (Our analysis shows that we'll need to hash all n items $n^{\log_{1/j_2}(1/j_1)}$ times in expectation)

Practical MinHash Considerations

So many Permutations!

- OK, so kR repetitions is a LOT of preprocessing, and a lot of random number generation

So many Permutations!

- OK, so kR repetitions is a LOT of preprocessing, and a lot of random number generation
- And most of this won't ever be used! Most of the time, when we hash, we don't make it more than a few indices into the permutation.

So many Permutations!

- OK, so kR repetitions is a LOT of preprocessing, and a lot of random number generation
- And most of this won't ever be used! Most of the time, when we hash, we don't make it more than a few indices into the permutation.
- Idea: Instead of taking just the first hash item that appears in the permutation, take the first (say) 3. Concatenate them together. Then we just need $k/3$ permutations per hash table to get similar bounds.

So many Permutations!

- OK, so kR repetitions is a LOT of preprocessing, and a lot of random number generation
- And most of this won't ever be used! Most of the time, when we hash, we don't make it more than a few indices into the permutation.
- Idea: Instead of taking just the first hash item that appears in the permutation, take the first (say) 3. Concatenate them together. Then we just need $k/3$ permutations per hash table to get similar bounds.
- So let's say we have $A = \{\text{black, red, green, blue, orange}\}$, and we're looking at a permutation $P = \{\text{purple, red, white, orange, yellow, blue, green, black}\}$.

So many Permutations!

- OK, so kR repetitions is a LOT of preprocessing, and a lot of random number generation
- And most of this won't ever be used! Most of the time, when we hash, we don't make it more than a few indices into the permutation.
- Idea: Instead of taking just the first hash item that appears in the permutation, take the first (say) 3. Concatenate them together. Then we just need $k/3$ permutations per hash table to get similar bounds.
- So let's say we have $A = \{$black, red, green, blue, orange$\}$, and we're looking at a permutation $P = \{$purple, red, white, orange, yellow, blue, green, black$\}$.
- Then A hashes to redorangeblue

Reducing Permutations

- If you take the $\hat{k}$ first items when hashing, rather than just taking the first one, we only need $kR/\hat{k}$ total permutations.

Reducing Permutations

- If you take the $\hat{k}$ first items when hashing, rather than just taking the first one, we only need $kR/\hat{k}$ total permutations.
- Does this affect the analysis?

Reducing Permutations

- If you take the $\hat{k}$ first items when hashing, rather than just taking the first one, we only need $kR/\hat{k}$ total permutations.
- Does this affect the analysis?
    - Yes; the k we're concatenating for each hash table are no longer independent!

Reducing Permutations

- If you take the $\hat{k}$ first items when hashing, rather than just taking the first one, we only need $kR/\hat{k}$ total permutations.
- Does this affect the analysis?
    - Yes; the k we're concatenating for each hash table are no longer independent!
    - But this works fine in practice (and is used all the time)

Reducing Permutations

- If you take the $\hat{k}$ first items when hashing, rather than just taking the first one, we only need $kR/\hat{k}$ total permutations.
- Does this affect the analysis?
  - Yes; the k we're concatenating for each hash table are no longer independent!
  - But this works fine in practice (and is used all the time)
- We will do this on the Assignment; in fact I recommend using $\hat{k} = k$. That means that each repetition has only one permutation.

Reducing Permutations

- If you take the $\hat{k}$ first items when hashing, rather than just taking the first one, we only need $kR/\hat{k}$ total permutations.
- Does this affect the analysis?
  - Yes; the k we're concatenating for each hash table are no longer independent!
  - But this works fine in practice (and is used all the time)
- We will do this on the Assignment; in fact I recommend using $\hat{k} = k$. That means that each repetition has only one permutation.
- I think it makes life very significantly easier. In the real world you want a smaller value of $\hat{k}$

Assignment Parameters

- 128 bit integers (stored as a `struct` of two unsigned 64 bit ints; called an `Item`)

Assignment Parameters

- 128 bit integers (stored as a `struct` of two unsigned 64 bit ints; called an `Item`)
- Universe: $\{0, \ldots, 127\}$. (You can pretend that these are images, each of which is labelled with a subset of 128 possible tags.)

Assignment Parameters

- 128 bit integers (stored as a `struct` of two unsigned 64 bit ints; called an `Item`)
- Universe: $\{0, \ldots, 127\}$. (You can pretend that these are images, each of which is labelled with a subset of 128 possible tags.)
- Each bit is a `0` or 1 at random

Assignment Parameters

- 128 bit integers (stored as a `struct` of two unsigned 64 bit ints; called an `Item`)
- Universe: $\{0, \ldots, 127\}$. (You can pretend that these are images, each of which is labelled with a subset of 128 possible tags.)
- Each bit is a `0` or 1 at random
- (Not realistic case, but hard case!)

What About Hashing?

- MinHash: go through each index in the permutation

What About Hashing?

- MinHash: go through each index in the permutation
- See if the corresponding bit is a 1 in the `Item` we're hashing.

What About Hashing?

- MinHash: go through each index in the permutation
- See if the corresponding bit is a 1 in the `Item` we're hashing.
- How can we do this?

What About Hashing?

- MinHash: go through each index in the permutation
- See if the corresponding bit is a 1 in the `Item` we're hashing.
- How can we do this?
- Most efficient way I know is not clever. Just go through each index, and check to see if that bit is set (say by calculating `x & (1 « index)` —but remember that these are 128 bits)

Concatenating Indices

- Each time you hash you'll get k indices

Concatenating Indices

- Each time you hash you'll get k indices
- Each is a number from 0 to 127

Concatenating Indices

- Each time you hash you'll get k indices
- Each is a number from `0` to 127
- How can these get concatenated together?

Concatenating Indices

- Each time you hash you'll get k indices
- Each is a number from `0` to 127
- How can these get concatenated together?
- Option 1: convert to strings, call `strcat`

Concatenating Indices

- Each time you hash you'll get *k* indices
- Each is a number from `0` to 127
- How can these get concatenated together?
- Option 1: convert to strings, call `strcat`
- Note: need to make sure to convert to *three-digit* strings! Otherwise hashing to 12 and then 1 will look the same as hashing to 1 and then 21. (`012` and `001` instead)

Concatenating Indices

- Each time you hash you'll get *k* indices
- Each is a number from `0` to 127
- How can these get concatenated together?
- Option 1: convert to strings, call `strcat`
- Note: need to make sure to convert to *three-digit* strings! Otherwise hashing to 12 and then 1 will look the same as hashing to 1 and then 21. (`012` and `001` instead)
- Option 2: Treat as bits. `0` to 127 can be stored in 7 bits. Store the hash as a sequence of *k* 8-bit chunks.

Getting a Good *k*

- In theory we want buckets of size 1.

Getting a Good k

- In theory we want buckets of size 1.
- In practice, we want *slightly* bigger.

Getting a Good k

- In theory we want buckets of size 1.
- In practice, we want *slightly* bigger.
- Why? Having a large number of buckets and/or repetitions leads to bad constants

Getting a Good k

- In theory we want buckets of size 1.
- In practice, we want *slightly* bigger.
- Why? Having a large number of buckets and/or repetitions leads to bad constants
- Smaller k means fewer buckets, fewer repetitions (but bigger buckets and more comparisons)

Getting a Good k

- In theory we want buckets of size 1.
- In practice, we want *slightly* bigger.
- Why? Having a large number of buckets and/or repetitions leads to bad constants
- Smaller k means fewer buckets, fewer repetitions (but bigger buckets and more comparisons)
- Start with $k \approx \log_3 n$, but experiment with slightly smaller values.

Repetitions?

- You’re guaranteed that there exists a close pair in the dataset

Repetitions?

- You’re guaranteed that there exists a close pair in the dataset
- My implementation just keeps repeating until the pair is found (no maximum number of repetitions)

Repetitions?

- You're guaranteed that there exists a close pair in the dataset
- My implementation just keeps repeating until the pair is found (no maximum number of repetitions)
- The discussion of repetitions in the lecture is for two reasons: 1. analysis, 2. give intuition for the tradeoff by varying k

How to Deal with Buckets?

- Each time we hash, (i.e. build a new “hash table”) need to figure out what hashes where so that we can compare elements with the same hash

How to Deal with Buckets?

- Each time we hash, (i.e. build a new "hash table") need to figure out what hashes where so that we can compare elements with the same hash
- Unfortunately, we're not hashing to a number from (say) 0 to $n - 1$. We're instead concatenating indices

How to Deal with Buckets?

- Each time we hash, (i.e. build a new “hash table”) need to figure out what hashes where so that we can compare elements with the same hash
- Unfortunately, we’re not hashing to a number from (say) 0 to $n-1$. We’re instead concatenating indices
- How to keep track of buckets?

How to Deal with Buckets?

- Each time we hash, (i.e. build a new "hash table") need to figure out what hashes where so that we can compare elements with the same hash
- Unfortunately, we're not hashing to a number from (say) 0 to $n - 1$. We're instead concatenating indices
- How to keep track of buckets?
- Answer: take the concatenated indices and put them into `MurmurHash`, then take modulo n.

How to Deal with Buckets?

- Each time we hash, (i.e. build a new "hash table") need to figure out what hashes where so that we can compare elements with the same hash
- Unfortunately, we're not hashing to a number from (say) 0 to $n - 1$. We're instead concatenating indices
- How to keep track of buckets?
- Answer: take the concatenated indices and put them into `MurmurHash`, then take modulo n.
- Then, we are hashing to a number from 0 to $n - 1$!

Assignment 4 plan

- Get a list of 128-bit bit vectors

Assignment 4 plan

- Get a list of 128-bit bit vectors
- First, minhash each to put it in a bucket from 0 to $n - 1$

Assignment 4 plan

- Get a list of 128-bit bit vectors
- First, minhash each to put it in a bucket from 0 to $n - 1$
- For each bucket, compare all pairs of vectors in the bucket; if a similar one is ever found return it

Assignment 4 plan

- Get a list of 128-bit bit vectors
- First, minhash each to put it in a bucket from 0 to $n - 1$
- For each bucket, compare all pairs of vectors in the bucket; if a similar one is ever found return it
- If a similar one is not found, repeat with new minhashes

Assignment 4: Minhashing a vector

We begin by generating a random permutation of the numbers from `0` to 127

We find the minhash of a 128-bit vector x as follows:

1. Find the first $k \approx \log_3 n$ numbers i in the permutation where x has a 1 in position i

Assignment 4: Minhashing a vector

We begin by generating a random permutation of the numbers from `0` to 127

We find the minhash of a 128-bit vector x as follows:

1. Find the first $k \approx \log_3 n$ numbers i in the permutation where x has a 1 in position i
2. Concatenate these numbers to form a string s

Assignment 4: Minhashing a vector

We begin by generating a random permutation of the numbers from `0` to 127

We find the minhash of a 128-bit vector x as follows:

1. Find the first $k \approx \log_3 n$ numbers i in the permutation where x has a 1 in position i
2. Concatenate these numbers to form a string s
3. Use Murmurhash on s to get a large random integer back

Assignment 4: Minhashing a vector

We begin by generating a random permutation of the numbers from 0 to 127

We find the minhash of a 128-bit vector x as follows:

1. Find the first $k \approx \log_3 n$ numbers i in the permutation where x has a 1 in position i
2. Concatenate these numbers to form a string s
3. Use Murmurhash on s to get a large random integer back
4. Take mod n to get a number from 0 to $n-1$