DSA5101 Introduction to Big Data for Industry

AY2025/26 Sem1 By Zhao Peiduo

Lecture 1

Frequent Itemsets and Association Rules

Association Rule Discovery Market-basket model:

- **Goal:** Identify items that are bought together by many supermarket customers.
- **Approach:** Process sales data of each customer to find dependencies among items.

The Market-Basket Model

- A large set of **items** — e.g., things sold in a supermarket.
- A large set of **baskets** — each basket is a small subset of items (what a customer buys).
- Want to discover **association rules** — e.g., people who bought $\{x, y, z\}$ tend to buy $\{v, w\}$.
- A many-to-many mapping between two kinds of things — connections among **items**, not baskets.
- Example:

Input:

TID	Items
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

Output: Rules Discovered: $\{\text{Milk} \to \text{Coke}\}, \{\text{Diaper, Milk} \to \text{Beer}\}$

Frequent Itemsets

- **Simplest question:** Find sets of items that appear together *frequently* in baskets.
- **Support** for itemset I: Number of baskets containing all items in I (often as a fraction of total baskets).
- Given a **support threshold** s: sets of items appearing in at least s baskets are **frequent itemsets**.
- Example:Support of {Beer, Bread} = 2 baskets.

Association Rules

- $\{i_1, \ldots, i_k\} \to j$ means: "if a basket contains all i's, it's *likely* to contain j".
- **Confidence:** Probability of j given $I = \{i_1, \ldots, i_k\}$

$$\text{conf}(I \to j) = \frac{\text{support}(I \cup \{j\})}{\text{support}(I)}$$

Interesting Association Rules

- Not all high-confidence rules are interesting — e.g., $X \to$ milk might be high just because milk is common.
- **Interest:** Difference between confidence and the fraction of baskets containing j

$$\text{Interest}(I \to j) = |\text{conf}(I \to j) - \Pr[j]|$$

- Interesting rules have high positive or negative interest values (usually above 0.5).

Finding Association Rules Problem: Find all rules with support $\geq s$ and confidence $\geq c$.

Mining Association Rules

- Find all frequent itemsets I.
- **Rule generation:** For every subset A of I, generate a rule $A \to I \setminus A$ with:

$$\text{conf}(A \to I \setminus A) = \frac{\text{support}(I)}{\text{support}(A)}$$

- **Observation 1:** Single pass over subsets of I to compute confidence.
- **Observation 2:** Monotonicity — if $B \subset A \subset I$, then

$$\text{conf}(B \to I \setminus B) \leq \text{conf}(A \to I \setminus A)$$

- Use monotonicity to prune rules below confidence threshold.

Itemsets: Computation Model

- Data is typically kept in flat files:
  - Stored on disk, too large to fit in main memory.
  - Stored basket-by-basket (e.g., $20, 52, 38, -1, 40, 22, -1, 20, 22, -1, \ldots$)
- **Major cost:** Time taken to read baskets from disk to memory.
- Assume baskets are small — a block of baskets can be expanded in main memory to generate all subsets of size k via k nested loops.
- Large subsets can often be ruled out using **monotonicity**.

Communication Cost is Key

- Running time $\propto$ #passes through data $\times$ data size.
- A pass = reading all baskets sequentially.
- Since data size is fixed, measure speed by number of passes.

Main-Memory Bottleneck

- For many algorithms, main memory is the limiting factor.
- While reading baskets, we need to count occurrences (e.g., pairs).
- The number of distinct items we can count is limited by memory.

Finding Frequent Pairs

- Goal: Find frequent pairs $\{i_1, i_2\}$.
- Frequent pairs are common; frequent triples are rare.
- Probability of being frequent drops exponentially with set size.
- **Approach:**
  - First focus on pairs, then extend to larger sets.
  - Generate all itemsets, but keep only those likely to be frequent.

Counting Pairs in Memory

- **Approach 1:** Use a matrix to count all pairs.
- **Approach 2:** Store triples $[i, j, c]$ meaning count of pair $\{i, j\}$ is c.
- Memory usage:
  - Approach 1: 4 bytes per pair.
  - Approach 2: 12 bytes per pair with count > 0.

Comparing the Two Approaches

- Triangular matrix: Count only if $i < j$, needs $2n^2$ bytes total.
- Approach 2 wins if less than 1/3 of possible pairs occur.

If Memory Fits All Pairs:

1. For each basket, double loop to generate all pairs.
2. Increment count for each generated pair.

Apriori Algorithm

- A two-pass (and beyond) algorithm that limits memory usage using **monotonicity (downward closure)**.
- **Key idea:** If I is frequent, then every subset $J \subset I$ is also frequent.
- **Contrapositive for pairs:** If i is infrequent, then no pair containing i can be frequent.

Steps (all itemset sizes)

- **Pass 1 (for $k = 1$):** Count each item; items with count $\geq s$ form L_1 (frequent items).
- **For $k \geq 2$:**
  - **Candidate generation:** From L_{k-1}, form C_k by joining two $(k-1)$-itemsets that share exactly $k-2$ items; take their union (size k).
  - **Pruning:** Remove any $I \in C_k$ if some $(k-1)$-subset of I is not in L_{k-1} (by monotonicity).
  - **Counting:** Make one pass; count supports of C_k.
  - **Filtering:** $L_k = \{I \in C_k \mid \text{support}(I) \geq s\}$.

Rule Generation (from frequent itemsets)

- For each frequent I and each nonempty $A \subset I$, form the rule $A \to I \setminus A$ with

$$\text{conf}(A \to I \setminus A) = \frac{\text{support}(I)}{\text{support}(A)}.$$

- **Observation (confidence monotonicity):** If $B \subset A \subset I$, then

$$\text{conf}(B \to I \setminus B) \leq \text{conf}(A \to I \setminus A).$$

- Use this to prune rules below the confidence threshold c.

Memory Requirement

- Pass 1: Memory $\propto$ number of (distinct) items.
- Pass k ($k \geq 2$): Memory $\propto$ number of candidates in C_k (for market-basket data and reasonable s, $k = 2$ is often the heaviest).
- **Trick:** Re-number frequent items $1, 2, \ldots$ and keep a map to original item IDs for compact indexing.

Hash Functions

- **Definition:** A hash function h takes a *hash-key value* and produces a *bucket number* $\in [0, B-1]$.
- Should randomize hash-keys roughly uniformly into buckets.

Indexing using Hash Functions

- Used for indexing to enable fast search/retrieval.
- **Example:** Hash the name to the ordinal position of the first letter, use as bucket index.

PCY (Park–Chen–Yu) Algorithm

- **Observation:** In Apriori's Pass 1, most memory is idle (only item counts stored). Use spare memory to *hash pairs into buckets* and prune candidate pairs before Pass 2.

Pass 1 of PCY

- Maintain a hash table with as many buckets as memory allows.
- For each basket:
  - Count each item's frequency (to get frequent items L_1).
  - For each unordered pair $\{i, j\}$ in the basket: compute bucket $b = h(\{i, j\})$ and increment that bucket's count (cap counts at s if desired).
- After the pass, replace bucket counts with a **bit vector**: bit $b = 1$ iff bucket count $\geq s$, else 0 (bitmap uses about 1/32 the memory of 32-bit integer counts).

Using Hash Buckets to Prune Candidate Pairs

- If a bucket's count $< s$, then no pair hashing to that bucket can be frequent — prune them.
- If a pair is truly frequent, its bucket must be frequent (so it will not be pruned).

PCY – Pass 2

- Count only pairs $\{i, j\}$ that satisfy **both**:
  1. i and j are frequent items (i.e., in L_1), and
  2. The pair hashes to a bucket whose bit is 1 in the bitmap (a "frequent" bucket).
- Note: On this pass, a table of (item, item, count) triples is essential (triangular matrices don't align with hash pruning).
- For PCY to beat Apriori, the hash table should eliminate roughly $\geq 2/3$ of the candidate pairs.

Refinement: Multistage Algorithm (3 passes)

- After Pass 1 of PCY, *rehash only* the pairs that would be considered in PCY's Pass 2 (i.e., both items frequent and first-hash bucket frequent) using an **independent** hash function to a second bucket table.
- Replace the second bucket counts with a second bit vector (slightly smaller, e.g., $\frac{31}{32}$ size).

Multistage – Pass 3

- Count only pairs $\{i, j\}$ that satisfy all:
  1. i and j are frequent,
  2. $\{i, j\}$ hashes to a frequent bucket in the *first* bitmap, and
  3. $\{i, j\}$ hashes to a frequent bucket in the *second* bitmap.
- Effect: Fewer candidate pairs than plain PCY, with combined bitmaps using about $\frac{1}{16}$ of the memory of integer bucket counts.

Important Points

1. The two hash functions must be independent.
2. Both hashes must be checked on the final counting pass.
3. More stages are possible for additional pruning, but each stage needs another bitmap; eventually memory runs out.

Refinement: Multihash (2 passes)

- Use several independent hash tables in *the first pass* and create multiple bitmaps (same total bitmap space as PCY if divided).
- **Pass 2:** Count only pairs whose buckets are frequent in *all* bitmaps (analogous to Pass 3 of multistage).
- *Trade-off:* Halving buckets doubles average bucket count; ensure many buckets still fall below s to retain pruning power.

Main-Memory Details

- We do not need to count a bucket past s.
- On the second pass, a triple table is required (cannot use a triangular matrix).

Adding More Hash Functions

- Either multistage or multihash can use more than two hash functions.
- In multistage, diminishing returns as bit-vectors consume memory.
- In multihash, bit-vectors use same space as one PCY bitmap; too many hash functions cause most buckets to become frequent.

Random Sampling

- If data is too large to fit in main memory, but a random sample fits in memory, then:
  - Run an in-memory frequent-itemset algorithm (e.g., A-Priori) on the sample.
  - Scale down support threshold proportionally.
- Challenge: Itemsets that are frequent in the whole dataset may not appear frequent in the sample.
- Result: May miss some frequent itemsets (false negatives).
- Advantage: Very fast and memory efficient.

SON Algorithm (Savasere, Omiecinski, Navathe)

- Works for distributed or map-reduce environment.
- Divide dataset into k chunks.
- Each chunk fits in memory.
- For each chunk:
  1. Find candidate frequent itemsets in the chunk (local frequent).
  2. Collect all candidates from all chunks.
- Run a second pass over the whole dataset to count the candidates' true support.
- Guarantee: Any itemset that is globally frequent must appear as frequent in at least one chunk.

SON Algorithm – Pass 1

- Each mapper runs A-Priori (or similar) on its chunk.
- Outputs locally frequent itemsets.
- Reducers aggregate all candidates across chunks.
- Result: A superset of globally frequent itemsets.

SON Algorithm – Pass 2

- Each mapper:
  - Counts support of the candidate itemsets from Pass 1 in its chunk.
- Reducers sum counts across mappers.
- Output: Globally frequent itemsets.

Why SON Works

- Suppose itemset I is frequent in the whole dataset.
- Then I must be frequent in at least one chunk.
- Thus I will be found in Pass 1.
- No false negatives.
- May have false positives (candidates that are not globally frequent).

Toivonen's Algorithm

- Another sampling-based algorithm.
- Steps:
  1. Take a random sample of the dataset that fits in memory.
  2. Run A-Priori (or similar) on the sample with a lowered support threshold.
  3. Result: Candidate itemsets (may contain false positives, but hopefully no false negatives).
  4. Run a second pass over the whole dataset to count supports of candidates.
  5. If no "frequent" itemsets are missed, done.
  6. Otherwise, repeat with a larger sample.

Toivonen's Algorithm – Key Idea

- Reduce risk of false negatives by lowering the support threshold in the sample.
- False positives are okay (they will be eliminated in the second pass).
- If a false negative occurs, restart with larger sample.

Toivonen's Algorithm – Example

- True support threshold: 5%.
- Sample size: 10% of dataset.
- Adjusted threshold: 0.5%.
- Find itemsets frequent in sample $\geq$ 0.5%.
- Verify on full dataset.

Toivonen's Algorithm – Advantages and Disadvantages

- Advantage:
  - Typically needs only 2 passes (sample + full dataset).
  - Efficient for large datasets.
- Disadvantage:
  - Risk of false negatives (forces restart).
  - Sample size and threshold adjustment critical.

Comparison: SON vs. Toivonen

- **SON**:
  - Always correct (no false negatives).
  - Requires 2 full passes of dataset.
  - Well-suited for distributed/MapReduce.
- **Toivonen**:
  - May fail and require restart, but usually only 2 passes.
  - Efficient when sample fits in memory.
  - Risk of wasted work if sample is not representative.

Comparison of Frequent Itemset Algorithms

Feature	Apriori	PCY	Random Sampling	SON	Toivonen's
Number of passes	One for each itemset size	One for each itemset size	< 2	2	< 2
False positives	No	No	No	No	No
False negatives	No	No	Yes	No	No
Memory usage	High	Lower than Apriori for pairs	Lower due to on-the-fly filter	Lower due to on-the-fly filter	Need to store negative border
Scalable to big data	Poor	Slightly better	Very good	Very good	Good
Candidate generation	Explicit, bottoms up	Same as Apriori, except for pairs	Sample-based heuristics	Same as Apriori, per chunk	Sample + negative border

Lecture 2

Finding Similar Items: Locality Sensitive Hashing (LSH)

- Given high-dimensional data points $x_1, x_2, \ldots$ and distance function $d(x_i, x_j)$.
- Goal: Find all pairs (x_i, x_j) such that $d(x_i, x_j) \leq s$.
- Naïve solution: $O(N^2)$ comparisons.
- Desired: $O(N)$ or close.

Motivation (From Apriori to LSH)

- Apriori: Keep candidate pairs instead of all.
- LSH: Hash documents so *similar ones fall in the same bucket*.
- **Pass 1:** Hash to buckets.
- **Pass 2:** Compare only within buckets.
- Benefit: Cuts comparisons from $O(N^2)$ to $O(N)$.

Power of LSH

- Use multiple hash functions.
- Only examine pairs that collide in at least one bucket.
- **Pros:** Tiny fraction of pairs examined.
- **Cons:** False negatives — some true pairs missed.

LSH: Essential Steps

1. **Shingling:** Represent documents as Boolean vectors.
2. **Min-Hashing:** Compress sets into signatures that preserve similarity.
3. **Locality-Sensitive Hashing:** Threshold on signatures to find candidate pairs.

Document

↓

Shingling
Output: The set of strings of length k that appear in the document

↓

Min Hashing
Output: **Signatures:** short integer vectors that represent the sets, and reflect their similarity

↓

Locality-Sensitive Hashing

↓

Output: Candidate pairs:
those pairs of signatures that we need to test for similarity

Step 1: Shingling

- Represent each document D as the set $S(D)$ of contiguous substrings of fixed length k (k-*shingles*).
- Example ($k = 2$): string `abcab` $\Rightarrow$ shingles {`ab`, `bc`, `ca`}.

Set → Boolean Matrix (incidence)

- Universe $\mathcal{U}$: all k-shingles seen in the corpus; docs $D_1, \ldots, D_m$.
- Build $M \in \{0,1\}^{|\mathcal{U}| \times m}$ where $M[i,j] = 1$ iff shingle $i \in S(D_j)$; each column = doc's Boolean vector.
- Tiny example ($k = 2$): $D_1 =$ `abc`, $D_2 =$ `bca`; $\mathcal{U} = \{$`ab`, `bc`, `ca`$\}$

Shingle	D_1	D_2
ab	1	0
bc	1	1
ca	0	1

Similarity (Jaccard on shingle sets)

- $J(A,B) = \dfrac{|A \cap B|}{|A \cup B|}$; high overlap in shingles $\Rightarrow$ high resemblance.

Choosing k

- Small k: many chance matches (noisy). Large k: few shingles, little overlap.
- Heuristic: shorter texts $\Rightarrow$ smaller k; longer docs $\Rightarrow$ larger k.

Properties of shingles

- *Order-aware:* capture local ordering (unlike bag-of-words).
- *Local change affects few shingles:* editing one character changes at most k shingles around the edit.
- *Length vs count:* a length-L doc has $\approx L-k+1$ (not-necessarily-unique) k-grams; unique set $|S(D)| \le L-k+1$.
- *Normalization matters:* case-folding, punctuation/whitespace handling, tokenization (char- vs word-grams) affect $S(D)$ and resemblance.

Practical representation (sparse) M is huge and sparse; store only $S(D)$ (IDs of shingles present), often via hashing each shingle to a 32-bit integer.

Amount of compression (vs Boolean vector)

- Boolean vector size: $|\mathcal{U}|$ bits per document (can be billions $\Rightarrow$ GBs).
- Sparse set size: $\approx 32 \cdot |S(D)|$ bits (if 32-bit IDs). For $L = 10^4$, $k = 5$, $|S(D)| \approx 10^4 \Rightarrow \sim$40 KB instead of many GBs.
- Rule of thumb: replace an intractable $|\mathcal{U}|$-bit column by a few thousand 32-bit IDs $\Rightarrow$ *orders-of-magnitude* smaller.

Min-Hashing (Step 2) — From Boolean Columns to Short Signatures

- **Goal:** Find similar columns (documents) by computing small integer *signatures.*
- **Key idea:** *Similarity of columns $\approx$ similarity of signatures.*
- **Compression:** Signature vectors are tiny (e.g., $\approx$100 integers) compared to huge Boolean columns.

Similarity-Preserving Hash

- Hash each column C (set of shingles) to a small signature $h(C)$ so that $h(C_1)$ and $h(C_2)$ preserve the similarity of C_1 and C_2.
- **Goal for** h: if $\mathrm{sim}(C_1, C_2)$ is high, then with high probability $h(C_1) = h(C_2)$; if low, then with high probability $h(C_1) \neq h(C_2)$.

Min-Hashing (for Jaccard)

- Suitable similarity-preserving hash for **Jaccard similarity**.
- Concept relies on random row permutations of the characteristic (Boolean) matrix.

Operational Overview

- Choose a random permutation π of the rows (shingles).
- Define $h_\pi(C)$ as the **index of the first row** (under π) where column C has value 1.
- Repeat with K independent permutations to build a length-K **signature** (vector of integers) for each column.
- **Signature matrix:** columns = documents (sets), rows = min-hash values for each permutation.

Example:

- Suppose document D_1 has shingles in rows $\{2,4\}$ and D_2 in rows $\{3,4\}$.
- Take permutation $\pi = [3,1,4,2]$ (ordering of rows).
- First 1 for D_1 under π: row 4 $\Rightarrow$ $h_\pi(D_1) = 3$ (since 4 is third in order).
- First 1 for D_2 under π: row 3 $\Rightarrow$ $h_\pi(D_2) = 1$.
- Thus $h_\pi(D_1) \neq h_\pi(D_2)$. With many permutations, the fraction of matches $\approx$ Jaccard similarity.

Min-Hash Property (Fundamental)

- For a random permutation π,

$$\Pr\left[h_\pi(C_1) = h_\pi(C_2)\right] \;=\; \mathrm{sim}_{\mathrm{Jaccard}}(C_1, C_2) \;=\; \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}.$$

- Intuition (row types): X-rows $(1,1)$ contribute $|C_1 \cap C_2|$, Y-rows $(1,0)$ or $(0,1)$ contribute the remainder of the union; collision occurs when the topmost (under π) 1 appears in an X-row.

Similarity from Signatures

- Define signature-similarity as the **fraction of positions** (hashes) in which two signatures agree.
- **Unbiased estimator:** $\mathbb{E}$[sig-sim] = Jaccard similarity of the original columns.
- **Accuracy:** Variance decreases with K; standard error $O(1/\sqrt{K})$.

Example (compute similarity between signatures):

perm	1	2	3	4	5	6
$s(D_1)$	2	1	4	7	5	3
$s(D_2)$	2	3	4	9	6	3
match?	✓		✓			✓

$$\widehat{\mathrm{Jaccard}}(D_1, D_2) = \frac{\#\mathrm{matches}}{K} = \frac{3}{6} = 0.5.$$

Simulating Permutations in Practice

- True permutations are expensive; simulate using hash families

$$h_{a,b}(x) = (ax+b) \bmod p, \quad p > N \text{ (prime)}, \; a, b \text{ random},$$

  where x indexes rows (N rows). Each (a,b) acts like an independent permutation.
- *Example:* $g(x) = 2x + 1 \bmod 5 \Rightarrow [g(1), g(2), g(3), g(4), g(5)] = [3,0,2,4,1]$ (treat 0 as 5).

Why Min-Hashing?

- **Space:** Replace massive sparse Boolean columns by $\sim K$ small integers.
- **Speed:** Compare signatures instead of full columns; enables LSH in the next step to find candidate pairs efficiently.

Implementation Technique — One-Pass Min-Hash

- Pick K independent hash functions h_i (e.g., $K = 100$).
- For each column c and hash h_i, reserve a slot $M(i,c)$; initialize $M(i,c) = \infty$.
- **Scan rows** (shingles): when row j has a 1 in column c, for every h_i:

$$\text{if } h_i(j) < M(i,c) \text{ then } M(i,c) \leftarrow h_i(j).$$

- After the scan, the *signature matrix* M has K rows (hashes) and one column per document.

Implementation (pseudocode)

- `for each row r:`
  - `for each hash` h_i`: compute` $h_i(r)$
  - `for each column c with 1 in row r:`
    - `if` $h_i(r) < M(i,c)$ `then` $M(i,c) := h_i(r)$

Example (building M with two hashes)

- Rows 1..5; two hashes: $h(x) = x \bmod 5$, $g(x) = (2x+1) \bmod 5$.
- Columns C_1, C_2 have 1's per the input matrix (slide).
- After scanning rows, final signatures (row 5 shown boxed on slide):

$$M(:, C_1) = (1,2), \quad M(:, C_2) = (0,0).$$

So Far

- Min-Hash compresses long Boolean columns to short signatures.
- Estimate Jaccard similarity via *signature agreement fraction.*
- All-pairs on N columns is still $O(N^2)$ (e.g., $N = 10^6 \Rightarrow \sim 5 \times 10^{11}$ pairs).
- Motivation for LSH: restrict comparisons to likely-similar pairs.

Candidates from Min-Hash (thresholding, not scalable)

- Goal: find columns with Jaccard $\ge s$ (e.g., $s = 0.8$).
- With $K = 100$ rows, require ≥ 80 signature matches.
- Statistically valid but still compares *every* pair $\Rightarrow O(N^2)$.

LSH Overview

- Split signature matrix M into multiple *bands*; hash each band of each column to a bucket.
- **Candidate pairs:** columns that land in the same bucket in at least one band.
- Tunable so that only *similar* columns collide with high probability.

Partition M into Bands

- Choose b bands of r rows ($K = b \cdot r$ total rows).
- For each band, hash its r-tuple to a table with k buckets (random collision $\approx 1/k$).
- Candidate if two columns share a bucket in ≥ 1 band.
- Complexity: hash b bands for each of N columns $\Rightarrow O(bN) \ll O(N^2)$.

Simplifying Assumption for Analysis

- Sufficient buckets so that columns hash to the *same* bucket iff their band vectors are *identical.*
- Ensure identical vectors in different bands use different bucket arrays (separate tables / dictionary).
- For analysis, "same bucket" $\equiv$ "identical in that band".

LSH Parameters

- b **(bands):** larger $b \Rightarrow$ more lenient (more false positives).
- r **(rows per band):** larger $r \Rightarrow$ stricter (more false negatives).
- Required Min-Hash rows $K = b \cdot r$.

textbfLSH – What We Want

- Probability of sharing a bucket = 1 if similarity $t > s$.
- Probability = 0 (no chance) if $t < s$.

What 1 Band of 1 Row Gives You

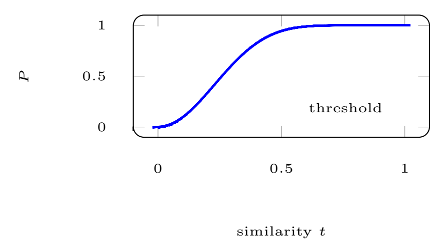
- Probability two sets hash to the same bucket $= t = \mathrm{sim}(C_1, C_2)$.
- *False negatives*: pairs with high t that miss.
- *False positives*: pairs with low t that collide.

General Case: b Bands, r Rows/Band Let similarity of two columns be t.

- Probability *one* band is identical: t^r.
- Probability *no* band identical: $(1 - t^r)^b$.
- Probability *at least one* identical band (hence candidate):

$$P(\text{candidate}) = 1 - (1 - t^r)^b.$$

S-Curve with b Bands of r Rows



Where is the Threshold? The steepest point of the S-curve (where $P = \frac{1}{2}$) satisfies

$$1 - (1 - t^r)^b = \tfrac{1}{2}.$$

For large b the approximation is

$$t \approx \Big(\frac{\ln 2}{b}\Big)^{1/r}.$$

Picking r and b via the S-Curve

- Blue zone (false negatives): sim > 0.6 but no band match.
- Green zone (false positives): sim < 0.6 but at least one match.

LSH Summary

- Tune K, b, r to *catch* most similar pairs while *discarding* most dissimilar ones.
- First filter: in-memory check that signatures really agree.
- Optional second filter: re-scan data to verify document-level similarity.

Summary: 3 Steps

1. **Shingling**: convert docs $\to$ sets of shingles.
2. **Min-Hash**: compress sets into short signatures preserving similarity.
3. **Locality-Sensitive Hashing**: hash bands of signatures to buckets to obtain candidate pairs with similarity $\ge s$.

Theory of LSH
- Used LSH to find similar documents.
- More generally, similar columns in large sparse matrices with high Jaccard similarity.
- **LSH for other distances**: e.g., Euclidean, cosine, Hamming, edit distance for strings.

Distance Measures
- A real-valued function $d(\cdot,\cdot)$ is a distance measure if:
  1. $d(x,y) \geq 0$ and $d(x,y)=0 \iff x=y$ (positivity)
  2. $d(x,y)=d(y,x)$ (symmetry)
  3. $d(x,y) \leq d(x,z)+d(z,y)$ (triangle inequality)
- Examples:
  - Jaccard distance: 1− Jaccard similarity
  - Cosine distance: angle between vectors
  - Euclidean distance: $L1$, $L2$ norms on vectors

Families of Hash Functions
- A hash function allows us to test if two elements have something in common.
- $h(x)=h(y) \implies x,y$ share a property.
- A *family* of hash functions = set of functions where we can randomly generate one efficiently.
- Example: Min-Hashing signatures, where each random row permutation defines a Min-Hash function.

Locality-Sensitive (LS) Families
- Suppose we have space S of points with distance $d(x,y)$.
- A family H is (d_1,d_2,p_1,p_2)**-sensitive** if:
  1. If $d(x,y) \leq d_1$, then $\Pr[h(x)=h(y)] \geq p_1 \ \forall h \in H$
  2. If $d(x,y) \geq d_2$, then $\Pr[h(x)=h(y)] \leq p_2 \ \forall h \in H$

Min-Hash as an LS Family
- S = space of sets, d = Jaccard distance, H = Min-Hash family.
- For any $h \in H$: $\Pr[h(x)=h(y)] = 1-d(x,y)$.
- Hence $(d_1,d_2,1-d_1,1-d_2)$-sensitive family:
  - If $d(x,y) \leq d_1$, then $\Pr[h(x)=h(y)] \geq 1-d_1$.
  - If $d(x,y) \geq d_2$, then $\Pr[h(x)=h(y)] \leq 1-d_2$.

Amplifying an LS Family
- Bands-and-rows technique creates S-curves for Min-Hashing.
- Works for any (d_1,d_2,p_1,p_2)-sensitive family.
- Effect can be stacked.
- Two constructions:
  - AND → rows in a band
  - OR → many bands

AND Construction
- Construct family H' of r functions from H.
- $h(x)=h(y) \iff h_i(x)=h_i(y) \ \forall i \leq r$.
- Theorem: If H is (d_1,d_2,p_1,p_2)-sensitive, then H' is (d_1,d_2,p_1^r,p_2^r)-sensitive.

OR Construction
- Construct family H' of b functions from H.
- $h(x)=h(y) \iff h_i(x)=h_i(y)$ for at least one i.
- Theorem: If H is (d_1,d_2,p_1,p_2)-sensitive, then H' is $(d_1,d_2,1-(1-p_1)^b,1-(1-p_2)^b)$-sensitive.

Effect of AND and OR Constructions
- **AND** makes all probabilities decrease, making the selection stricter.
- **OR** makes all probabilities increase, making the selection less strict.

Combine AND and OR Constructions
- By choosing b and r carefully, the lower probability approaches 0 and the higher probability approaches 1.
- For the signature matrix, sequences of alternating AND's and OR's can be combined.

Composing Constructions
- **AND-OR construction**: r-way AND followed by b-way OR.
  - **AND**: If bands match in *all r values*, hash to the same bucket.
  - **OR**: Columns that have *at least one common bucket* ⇒ Candidate.
- Suppose $\text{Prob}[h(x)=h(y)]=s$.
- Candidate pair probability:

$$1-(1-s^r)^b$$

  (This is the S-curve).

Amplifying Probabilities: Steeper S-curves
- **AND-construction**: $(d_1,d_2,p_1,p_2) \mapsto (d_1,d_2,(p_1)^r,(p_2)^r)$. Prob(candidate) decreases faster for distant pairs.
- **OR-construction**: $(d_1,d_2,p_1,p_2) \mapsto (d_1,d_2,1-(1-p_1)^b,1-(1-p_2)^b)$. Prob(candidate) increases faster for close pairs.
- Steeper S-curve achieved by increasing $n=br$.

Choosing r and b
- Example: 50 hash functions ($r=5, b=10$).
- Blue area (False Negatives): Pairs with sim $> s = 0.6$ but no band agreement ⇒ missed candidates.
- Green area (False Positives): Pairs with sim $< s = 0.6$ but incorrectly selected as candidates. Verified later ⇒ not too bad but adds time.

OR-AND construction: b-way OR followed by r-way AND.
- Suppose $\text{Prob}[h(x)=h(y)]=s$.
- Exercise: Check that OR-AND construction makes (x,y) a candidate pair with probability

$$[1-(1-s)^b]^r$$

- Recall S-curve for AND-OR: $1-(1-s^r)^b$.
- Curves related by:
  - Vertical mirroring: $s \to 1-s$,
  - Horizontal mirroring: $P \to 1-P$,
  - Swapping r and b.

Cascading Constructions
- Example: Apply the (4,4) OR-AND followed by the (4,4) AND-OR.
- A $(0.2,0.8,0.8,0.2)$-sensitive family becomes a

$$(0.2,0.8,0.9999996,0.0008715)\text{-sensitive family}$$

- Requires 256 hash functions.

Remark
- Implementation of cascades is more complex.
- Computational cost of candidate pairs remains linear in number of documents/columns.

When to use a Cascade?
- **Advantage**: Can yield steeper S-curve if tuned properly.
- **Disadvantage**: Implementation and tuning complexity.

Stick to a Single AND-OR When:
- Few pairs of medium similarity → steep S-curve not essential.
- Candidate verification is cheap → false positives acceptable.
- Number of min-hash signatures too small.
- Example: (4,4,4,4) AND-OR-AND-OR already requires 256 signatures.

Fixed Point of S-Curves
- For AND-OR S-Curve $1-(1-s^r)^b$, there exists fixed point t where

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

- Above t: high probabilities increase.
- Below t: low probabilities decrease.
- ⇒ Improved sensitivity.

LSH for Other Distance Metrics
- **Cosine distance**: Random hyperplanes.
- **Euclidean distance**.
- Design (d_1,d_2,p_1,p_2)-sensitive families depending on distance metric.
- Signatures → reflect similarity, then LSH → candidate pairs.
- Amplify using AND/OR constructions.

Cosine Distance
- Cosine distance = angle between vectors from origin to points.
- Formula:

$$d(A,B) = \theta/\pi = \arccos\left(\frac{A\cdot B}{\|A\|\|B\|}\right)/\pi$$

- Range: $[0,1]$.
- Cosine similarity: $1-d(A,B)$.
- Remark: Differs from some other cosine similarity definitions.

LSH for Cosine Distance
- Analogue of Min-Hash: **Random Hyperplanes**.
- Constructs $(d_1,d_2,1-d_1,1-d_2)$-sensitive family.

Random Hyperplanes
- Let v = normal vector to hyperplane.
- Each v defines hash function h_v with 2 buckets:

$$h_v(x) = \begin{cases} +1 & v\cdot x \geq 0 \\ -1 & v \cdot x < 0 \end{cases}$$

- LS-family H = set of all h_v.

Proof of Claim
- Prob[Red case] $= \theta/\pi = d(x,y)$.
- So, $\Pr[h(x)=h(y)] = 1-d(x,y)$.

Signatures for Cosine Distance
- Generate random vectors v, apply h_v to data points.
- Result: signature of ±1 for each point.
- Apply LSH with alternating AND/OR, as in Min-Hash.

How to Pick Random Vectors
- Expensive to generate random vector in M dimensions (need M random numbers).
- Efficient approach: use M-dimensional vectors with entries ±1.
- This covers space uniformly (unbiased).

LSH for Euclidean Distance
- Hash functions correspond to lines.
- Partition line into buckets of size a.
- Hash datapoint to bucket of its projection:
  - Signature = bucket ID for projection line.
- Nearby points → close, distant points → seldom same bucket.

Projection of Points (Cases)
- **Lucky case:** Nearby points hash to same bucket, faraway to different buckets.
- **Unlucky case:**
  - Top: unlucky quantization.
  - Bottom: unlucky projection.

Projection: Points at Distance d
- If $d \ll a$, probability points in same bucket $\geq 1-d/a$.

Projection: Large Distance
- If $d \gg a$, angle $\theta \approx 90^\circ$ needed for chance of same bucket.
- Condition: $d\cos\theta < a$.

One LS-Family for Euclidean Distance
- Let $d(x,y)$ = Euclidean distance.
- If $d(x,y) \leq a/2$, then $\Pr[h(x)=h(y)] \geq \frac{1}{2}$.
- If $d(x,y) \geq 2a$, then $\Pr[h(x)=h(y)] \leq \frac{1}{3}$.
- This yields $(a/2,2a,1/2,1/3)$-sensitive family.
- Can be amplified with AND/OR constructions.

Lecture 3

Clustering
- Partition dataset $\mathcal{D} = \{x_i\}_{i=1}^N$ into K disjoint groups.

$$\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2 \cup \cdots \cup \mathcal{D}_K$$

- Within a group: distances are small.
- Across groups: distances are large.

Typical Case
- Points in high-dimensional space.
- Similarity defined via a distance measure:
  - Euclidean, Cosine, Jaccard, Edit distance, ...

Why Clustering? Applications
- Information retrieval
- Market research
- Image compression and segmentation
- Anomaly detection:
  - Factory quality control
  - Credit card fraud detection

Which Similarity Measure?
- Vectors: Cosine similarity.
- Sets: Jaccard similarity.
- Points: Euclidean distance.
- Choice requires domain knowledge.

Problem with High Dimensions
- In 2D: clustering is easy.
- Curse of dimensionality: in high dimensions, most pairs of points are far apart.

Example: Curse of Dimensionality
- 10^6 uniform random points in $[0,1]^d$.
- Want 10 nearest neighbors from origin.
- On average, need to cover $\frac{10}{10^6} = 10^{-5}$ of volume.
- In 2D: need square with side $\sqrt{10^{-5}} = 0.0032$.
- General case: hypercube side length $(10^{-5})^{1/d}$.
- If $d = 7$: need 19.3% of range to capture 0.001% of points.

Methods of Clustering
- **Hierarchical**
  - Agglomerative (bottom-up): start with each point, merge nearest clusters.
  - Divisive (top-down): start with one cluster, recursively split.
- **Point assignment**
  - Maintain set of clusters.
  - Assign points to nearest cluster.

Hierarchical vs Point Assignment
- Point assignment works best with convex clusters.
- Hierarchical may work better for irregular shapes.
- Example: concentric circles share same centroid $\rightarrow$ use polar coordinates.

Hierarchical Clustering: Key Operation
- Repeatedly combine two nearest clusters.
- Representing a Cluster of Many Points
  - Use centroid = average of members.
  - Works in Euclidean spaces ($\mathbb{R}^n$).
  - Not valid in non-Euclidean (e.g., $\mathbb{Z}^n$).
- Distance Between Clusters
  - Define as distance between centroids of clusters.

Non-Euclidean Case
- In non-Euclidean spaces, the notion of an **average point** does not exist.
- **Approach 1:**
  - Represent each cluster by its **Clustroid** = datapoint closest to other points in the cluster.
  - Define the **distance between two clusters** as the distance between their clustroids.

Selecting the Clustroid
- Clustroid = point closest to all other points in the cluster.
- Possible definitions of "closest":
  - Smallest sum of distances to other points.
  - Smallest sum of squared distances to other points.
  - Smallest maximum distance to other points.
- Difference:
  - **Centroid:** average of all datapoints (artificial point).
  - **Clustroid:** existing datapoint closest to others.

No Cluster Representative Approaches
- Works for both Euclidean and non-Euclidean cases.
- **Approach 2:**
  - Define **Intercluster distance** = minimum distance between any two points (one from each cluster).
  - Merge clusters with the smallest intercluster distance.

Cohesion
- **Approach 3:**
  1. Pick a notion of **cohesion**.
  2. Merge clusters whose union is most cohesive.
- Cohesion could mean:
  1. **Diameter:** maximum distance between any two points in the cluster.
  2. **Average distance:** average distance among cluster points.
  3. **Density-based:** number of points divided by cluster volume.

Stopping Criteria
- If we fix the number of clusters beforehand, stopping is trivial.
- Other criteria:
  - Stop if **diameter** exceeds a threshold.
  - Stop if **density** falls below a threshold.
  - Stop if merging produces bad clusters (e.g., sudden diameter jump).

Which Approach is Best?
- Depends on cluster shape.
- **Approach 1:** merge clusters with smallest centroid/clustroid distance.
- **Approach 2:** merge clusters with smallest member-to-member distance.

Case 1: Convex Clusters
- Centroid-based merging works well.
- Closest-member merging (Approach 2) may fail.

Case 2: Concentric Circles
- Closest-member merging works best.
- Centroid-based merging fails.

K-means Clustering Algorithm
- The **K-means algorithm** is the simplest point assignment clustering algorithm.
- **Main idea:**
  - Identify some representatives of each cluster.
  - Assign each data point to a cluster by some rules.
- **Assumptions:**
  - Number of clusters K is pre-determined.
  - Euclidean case.

Cluster Representatives
- Consider a dataset $\mathcal{D} = \{x_i\}_{i=1}^N$ with $x_i \in \mathbb{R}^d$.
- Represent the "centers" of these K clusters by $z_1, \ldots, z_K \in \mathbb{R}^d$.

Assignment Matrix
- Associate each datapoint x_i to some cluster representative z_k:

$$r_{ik} = \begin{cases} 1 & \text{if } x_i \text{ is assigned to cluster } k \\ 0 & \text{otherwise} \end{cases}$$

- The $(N \times K)$ matrix $R = \{r_{ik}\}$ is the **assignment matrix**.
- Example:

$$R = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \quad \Rightarrow \quad x_1 \in \text{Cluster } 3,\ x_2 \in \text{Cluster } 1$$

The Loss Function
- Define loss to improve clustering: **within-cluster mean square loss**.

$$J(R, Z) = \frac{1}{2N} \sum_{k=1}^{K} \sum_{x \in C_k} \|x - z_k\|^2$$

- Measures average distance between datapoints and their cluster centers.

The K-means Algorithm
- Tries to minimize $J(R, Z) = \frac{1}{2N} \sum_{i=1}^N \sum_{k=1}^K r_{ik} \|x_i - z_k\|^2$.
- Hard to solve directly (NP-hard).
- Solution: optimize **step-wise**.

Step I: Given Z, find R
- Fix Z, then minimize distortion by assigning each x_i to closest center.

$$r_{ik} = \begin{cases} 1 & k = \arg\min_\ell \|x_i - z_\ell\|^2 \\ 0 & \text{otherwise} \end{cases}$$

Step II: Given R, find Z
- Fix R, then update centers:

$$\frac{\partial J}{\partial z_k} = 0 \quad \Rightarrow \quad z_k = \frac{\sum_i r_{ik} x_i}{\sum_i r_{ik}} = \frac{\sum_{x \in C_k} x}{|C_k|}$$

- This is the **mean** of points in cluster k.
- Same as centroid definition in hierarchical clustering.

Algorithm 1 K-means Clustering Algorithm

Require: Dataset $\mathcal{D} = \{x_i\}_{i=1}^N$, $x_i \in \mathbb{R}^d$; K; stopping criterion
Ensure: Centers Z, assignments R

```
Initialize Z ∈ R^{K×d}
while stopping criterion not reached do
    for i = 1 to N do
        k* ← arg min_j ||x_i − z_j||^2
        r_{ik*} ← 1;   r_{ij} ← 0 for j ≠ k*
    end for
    for k = 1 to K do
        z_k ← (Σ_{i=1}^N r_{ik} x_i) / (Σ_{i=1}^N r_{ik})
    end for
end while
return Z, R
```

Convergence
- Next clustering only depends on current clustering.
- Loss decreases $\Leftrightarrow$ new clustering assignment.
- Only a finite number of assignments $\Rightarrow$ K-means converges in finite steps.

However
- Loss J converges, R, Z stop updating.
- Does not imply global optimum.
- Depends on initial conditions.

Initializations for K-means

- **Approach 1: Pick points to be far away from each other**
  - Let z be a random point in our dataset.
  - Pick z_{k+1} to be a datapoint that is the furthest from all existing centres $z_1, z_2, \ldots, z_k$.
  - This works well assuming that there are no outliers.
- **Approach 2:** Take a small sample and cluster it; use the centroids as seed: K-means ++. Sample size $\propto K \log N$.

Sampling rule

- First point chosen uniformly at random.
- Subsequent point p added with probability $\propto D(p)^2$, where $D(p)$ = distance to nearest point already sampled.

How to choose K?

- Without domain knowledge, use the **elbow method**.
- Try different K; plot **within-cluster mean-square loss** vs. K.
- Best K is at the "elbow" of the curve.

BFR Algorithm

- Variant of K-means for datasets that do *not* fit in main memory.
- Assumptions: Clusters are normally distributed around centroids in Euclidean space.
- Covariance is diagonal (clusters axis-aligned).
- Memory usage $O(\#\text{clusters})$ instead of $O(\#\text{data})$.
- When high-confidence assignments exist, summarize clusters and discard points from RAM.

Summary statistics per cluster

- N : number of points.
- SUM : vector sum of all points.
- SUMSQ: vector whose i-th entry is $\sum x_i^2$ for that cluster.
- Centroid kept; all constituent points discarded from RAM.
- Each cluster summarized by $2d + 1$ scalars instead of storing all points (d = dimension).

Derived quantities

- Mean in dimension i: $\dfrac{\text{SUM}_i}{N}$.
- Variance in dimension i: $\dfrac{\text{SUMSQ}_i}{N} - \left(\dfrac{\text{SUM}_i}{N}\right)^2$.

BFR Overview

1. Initialize K clusters/centroids.
2. Load a small chunk of points into memory.
3. Assign high-confidence points to existing clusters, then summarize & discard.
4. Run hierarchical clustering on remaining points to create extra mini-clusters.
5. Attempt to merge new mini-clusters with existing clusters.

Three Classes of Points

- **Discard set (DS)**: points close enough to existing centroids to be summarized and discarded.
- **Compression set (CS)**: mini-clusters close to each other but *not* to any centroid; retained as summaries.
- **Retained set (RS)**: isolated points not yet summarized.

BFR Algorithm in Detail

Step 1: Initialize K Centroids

- Read points in memory-sized chunks; apply K-means++ (or similar) on first chunk to get initial K centroids.

Step 3: Assign & Summarize High-Confidence Points

- For each point in the current chunk compute Mahalanobis distance to every centroid.
- If the minimum distance is below threshold τ, assign to that cluster (add to **Discard Set, DS**) and update its N, SUM, SUMSQ; then discard the point from RAM.

Step 4: Cluster Remaining Points

- Run an in-memory algorithm (e.g. hierarchical clustering) on the leftover points plus any previous **Retained Set (RS)**.
- Compact mini-clusters become **Compression Set (CS)**; isolated singletons form the new **RS**.

Step 5: Merge & Finalize

- Attempt to merge CS mini-clusters if their combined variance (computed via N, SUM, SUMSQ) stays below a variance threshold.
- On the final pass: either merge each remaining CS mini-cluster or RS point to its closest centroid or treat them as outliers.

"Sufficiently Close" via Mahalanobis Distance

$$d(x, c) = \sqrt{\sum_{i=1}^{d} \left(\frac{x_i - c_i}{\sigma_i}\right)^2}$$

- Normalized Euclidean distance accounting for per-dimension spread.
- Lower $d(x, c) \Rightarrow$ higher probability the point belongs to the cluster.
- Threshold: choose $\tau \propto \sqrt{d}$; e.g. $\tau = 3\sqrt{d}$ keeps ≈99.7% of normally distributed points.

The CURE Algorithm

Motivation

- BFR assumes clusters are normally distributed and axis-aligned.
- CURE (Clustering Using REpresentatives) works in Euclidean space and allows *arbitrary* cluster shapes.
- **High-level idea**: Represent each cluster by a small set of well-scattered *representative points.*

Pass 1 — Build & summarize clusters

1. Draw a random sample that fits in memory.
2. Cluster the sample hierarchically (e.g. agglomerative with nearest-point merging).
3. For every resulting cluster
   - Pick c representatives as dispersed as possible (greedy farthest-first).
   - Move each representative 20% of the way toward the cluster centroid (shrinks boundaries). This reduces the effective boundary of large, dispersed clusters and helps nearby small, dense clusters avoid being swallowed by larger ones.
4. Discard all sample points; keep only the shrunk representatives.

Merging clusters

- Merge two clusters if any pair of their representatives is closer than threshold τ.
- Re-select scattered representatives from the merged set and shrink again.
- Continue until no more merges satisfy the closeness test.

Pass 2 — Assign every point

- Scan the full dataset once.
- For each point p, find the closest representative (Euclidean distance) and assign p to that cluster.

Evaluation with labels

- Clustering is unsupervised, but labels enable metrics such as purity, Rand-index, entropy.

Purity

$$\text{Purity}(\omega_i) = \frac{1}{n_i} \max_j n_{ij}, \quad \text{where } n_{ij} = |\{x \in \omega_i \text{ of class } j\}|,\ n_i = |\omega_i|.$$

Lecture 4

Matrix Factorization Goal

- Factor a matrix into *three* smaller matrices sharing a common low dimension r.

Capturing the Variation

- Data may lie on a lower-dimensional manifold; uncover the effective dimension.

Maximizing Variance

1. 1st direction = greatest variance.
2. 2nd direction = orthogonal to 1st and has next greatest variance, ...

Rank of a Matrix

$$\text{rank}(A) = \text{number of linearly independent columns (or rows) of } A.$$

Constrained Matrix Factorization via SVD

- SVD gives unique (up to sign) factorization obeying orthogonality and ordering constraints.
- Choose latent dimension r to minimize reconstruction error ⇔ maximize captured variance.

SVD Definition

$$A \approx U\Sigma V^T = \sum_{i=1}^{r} \sigma_i \, u_i \circ v_i$$

A: $m \times n$ input matrix (e.g. m documents, n terms).
U: $m \times r$ left singular vectors (document-to-concept).
Σ: $r \times r$ diagonal matrix of singular values $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r \geq 0$ (concept strengths).
V: $n \times r$ right singular vectors (term-to-concept).

SVD Restrictions

- U, V column-orthonormal: $U^T U = V^T V = I_r$.
- Σ diagonal with non-negative, descending singular values.

Interpretation

$$\underbrace{U}_{\text{user-to-concept}} \quad \underbrace{\Sigma}_{\text{concept strength}} \quad \underbrace{V^T}_{\text{concept-to-item}} \quad \Rightarrow \quad \text{low-rank approximation of original matrix.}$$

Dimensionality Reduction with SVD

1-D capture of variance: The first right singular vector already explains most of the variance; drop the second dimension.

Minimize reconstruction loss

$$\text{Loss} = \sum_{i,j} |x_{ij} - z_{ij}|^2 \quad \text{(old vs. new coordinates)}$$

- SVD gives the coordinates Z that minimize this loss for any chosen rank r.

How dimension reduction is done

- Set the smallest singular values to zero (truncate Σ).
- Reconstruct with the remaining factors.
- Reconstruction error = Frobenius norm $\|A - B\|_F = \sqrt{\sum_{i,j}(A_{ij} - B_{ij})^2}$ stays small.

Best low-rank approximation (Eckart–Young)

$$\text{Keep first } r \text{ factors of } A = U\Sigma V^T \Rightarrow B = \sum_{i=1}^{r} \sigma_i u_i v_i^T$$

- B is the *closest* rank-r matrix to A in Frobenius (and spectral) norm.

Choosing the number of latent factors

- Retain enough singular values to preserve 80–90% of the total "energy" $\sum \sigma_i^2$.
- Example: $\sigma = \{12.4,\ 9.5,\ 1.3\}$
  dropping $\sigma_3 = 1.3$ keeps $> 99\%$ of energy.

Finding Principal Eigenvector

- Need a method for finding the *principal eigenvalue* and the corresponding eigenvector (largest) of a symmetric matrix.
- M is symmetric if $M = M^T$.

Method

- Initialize with a random vector x_0.
- Update iteratively:

$$x_{k+1} = \frac{Mx_k}{\|Mx_k\|}$$

  for $k = 0, 1, \ldots$
- $\|\cdot\|$ denotes Frobenius norm.
- Stop when iterations stabilize.

Finding Principal Eigenvalue

- Once eigenvector x is found:

$$\lambda = x^T M x$$

- Assume $x^T x = 1$.

Finding Subsequent Eigenvectors

- Remove effect of first eigenvector x with eigenvalue λ:

$$M^* := M - \lambda x x^T$$

- Recursively find next eigenvector/eigenvalue of M^* via power iteration.

Computing the SVD

- To compute U, V in $A = U\Sigma V^T$, calculate eigenvectors of covariance matrices.
- Since Σ is diagonal:

$$A^T A = V\Sigma^2 V^T$$

- Thus, column i of V is eigenvector of $A^T A$, with eigenvalue σ_i^2.
- Similarly, eigenvectors of AA^T give U.

Complexity of SVD

- Specialized methods: $O(nm^2)$ or $O(mn^2)$ (whichever smaller).
- Less work if only singular values, first r vectors, or sparse matrix needed.
- Implementations: LINPACK, Matlab, Mathematica, SKLearn.

How to Query?

- Map query into *concept space*.
- Example: user likes "Matrix"
$$q = [5, 0, 0, 0, 0]$$
- Project into concept space:
$$q_{\text{concept}} = qV$$
- Similarly, user profile d:
$$d_{\text{concept}} = dV$$
- Observation: Users with no overlap in ratings (e.g., q: Matrix vs d: Alien, Serenity) can still be similar in concept space.

SVD Pros and Cons

Pros

- Optimal low-rank approximation in terms of Frobenius norm.

Cons

- Interpretability problem:
  - A singular vector specifies a linear combination of all input columns or rows.
- Lack of sparsity:
  - Singular vectors are dense!