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: {Milk → Coke}, {Diaper, Milk → 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)

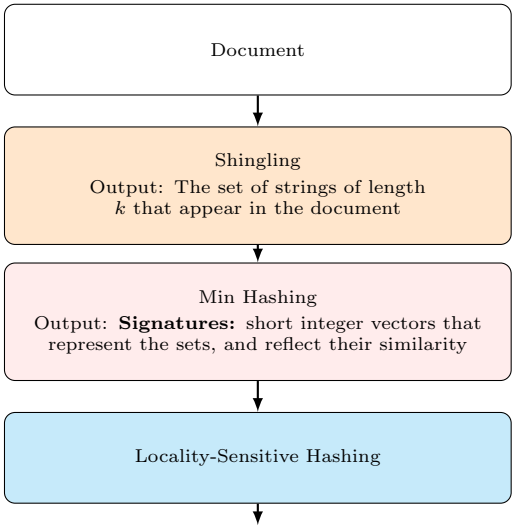
- Apriori: Keep candidate pairs instead of all.
- LSH: Hash documents so *similar ones fall in the same bucket.*
- **Pass 1:** Hash to buckets.
- **Pass 2:** Compare only within buckets.
- Benefit: Cuts comparisons from $O(N^2)$ to $O(N)$.

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.



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 ⇒ high resemblance.

Choosing k

- Small k: many chance matches (noisy). Large k: few shingles, little overlap.
- Heuristic: shorter texts ⇒ smaller k; longer docs ⇒ 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 ⇒ GBs).
- Sparse set size: $\approx 32 \cdot |S(D)|$ bits (if 32-bit IDs). For $L = 10^4$, $k = 5$, $|S(D)| \approx 10^4$ ⇒ ∼40 KB instead of many GBs.
- Rule of thumb: replace an intractable $|\mathcal{U}|$-bit column by a few thousand 32-bit IDs ⇒ *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 ≈ similarity of signatures.*
- **Compression:** Signature vectors are tiny (e.g., ≈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 ⇒ $h_\pi(D_1) = 3$ (since 4 is third in order).
- First 1 for D_2 under π: row 3 ⇒ $h_\pi(D_2) = 1$.
- Thus $h_\pi(D_1) \neq h_\pi(D_2)$. With many permutations, the fraction of matches ≈ 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:* $g(x) = 2x+1 \bmod 5$ ⇒ $[g(1), g(2), g(3), g(4), g(5)] = [3,0,2,4,1]$ (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., $N = 10^6$ ⇒∼ 5×10^{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 ⇒ $O(N^2)$.

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 bucket 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 ⇒ $O(bN) \ll O(N^2)$.

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" ≡ "identical in that band".

LSH Parameters

- b **(bands):** larger b ⇒ more lenient (more false positives).
- r **(rows per band):** larger r ⇒ 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

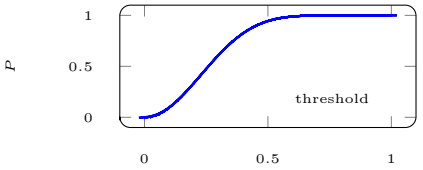
- Probability two sets hash to the same bucket = $t = \mathrm{sim}(C_1, C_2)$.
- *False negatives*: pairs with high t that miss.
- *False positives*: pairs with low t that collide.

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

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

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 → 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 $Z \in \mathbb{R}^{K \times d}$
while stopping criterion not reached **do**
  for $i = 1$ to N **do**
    $k^\star \leftarrow \arg\min_j \|x_i - z_j\|^2$
    $r_{ik^\star} \leftarrow 1; \quad r_{ij} \leftarrow 0$ for $j \neq k^\star$
  end for
  for $k = 1$ to K **do**
    $z_k \leftarrow \frac{\sum_{i=1}^N r_{ik}\, x_i}{\sum_{i=1}^N r_{ik}}$
  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

$\text{rank}(A)$ = 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!

CUR Decomposition

Sparsity Motivation

- Data matrix A is often sparse.
- However, SVD factors U and V are not sparse.
- CUR solves this by sampling rows and columns of A.

Definition and Goal

- Express A as

$$A \approx CUR$$

such that

$$\|A - CUR\|_F \text{ is small.}$$

- C: sample of columns of A.
- R: sample of rows of A.
- U: pseudoinverse adjustment.

Computing the Pseudoinverse U

- Let W = intersection of sampled C and R.
- W_{ij} = entry of A at j-th col of C, i-th row of R.
- Compute SVD:

$$W = XZY^T$$

- Then:

$$U = W^+ = YZ^+X^T$$

- Z^+: reciprocals of nonzero singular values.
- W^+ = pseudoinverse of W.

Rough Intuition

- CUR tends to pick points far from the origin.
- Assuming smoothness/no outliers, these capture directions of maximum variation.

Algorithm 2 CUR: Column Sampling Algorithm (similarly for rows)

Require: Matrix $A \in \mathbb{R}^{m \times n}$, sample size c

Ensure: $C_d \in \mathbb{R}^{m \times c}$

1: **for** $x = 1 : n$ **do** [column distribution]

2: $\quad P(x) = \dfrac{\sum_i A(i,x)^2}{\sum_{i,j} A(i,j)^2}$

3: **end for**

4: **for** $i = 1 : c$ **do** [sample columns]

5: $\quad$ Pick $j \in \{1, \ldots, n\}$ based on distribution $P(x)$

6: $\quad$ Compute $C_d(:, i) = \dfrac{A(:, j)}{\sqrt{cP(j)}}$

7: **end for**

CUR: Pros and Cons Pros

- Easy interpretation: basis vectors are actual rows/columns.
- Sparse basis: retains sparsity of original A.

Cons

- Duplicate columns/rows: large-norm columns sampled multiple times.

Solution to Duplicates

- If we want to get rid of the duplicates:
  - Throw them away.
  - Scale (multiply) the columns/rows by the square root of the number of duplicates.

SVD vs. CUR

SVD: $A = U\Sigma V^T$

- A: Huge but sparse.
- U: Big and dense.
- Σ: Sparse and small.
- V^T: Big and dense.

CUR: $A = CUR$

- A: Huge but sparse.
- C: Big but sparse.
- U: Dense but small.
- R: Big but sparse.

Empirical Performance (DBLP data)

- Dataset: DBLP bibliographic data.
- A_{ij} = number of papers published by author i at conference j.
- 428K authors (rows), 3659 conferences (columns).
- Very sparse matrix.

Goal: Dimensionality Reduction

- How much time does it take?
- What is the reconstruction error?
- How much memory is required?

Results: DBLP – Big Sparse Matrix

- **Accuracy:** 1− relative sum squared errors.
- **Space ratio:** #output matrix entries / #input matrix entries.
- **CPU time:** Performance compared across SVD, CUR, and CUR (no duplicates).

Lecture 5

Recommendation Systems

- Recommendation systems are an extensive class of web applications.
- Goal: Predict a user's responses to options.

Types of Recommendations

- Hand curated: List of best movies.
- Simple aggregates: Videos with the highest views in the past month.
- Personalized: Examples: YouTube, Lazada. (Our focus.)

Formal Model

- X = set of customers.
- S = set of items.
- Utility function: $u : X \times S \to R$.
- R = set of ratings.
- R is a totally ordered set (e.g., a rating could be in $[0, 1]$ or 1–5 stars).

Utility Matrix

- The utility function can be represented by a utility matrix.
- Typically sparse.

Example matrix:

	Avatar	LOTR	Matrix	Pirates
Alice	1		0.2	
Bob		0.5		0.3
Carol	0.2		1	
David				0.4

Key Problems

1. Data Collection: Gathering known ratings. How do we get the entries in the initial utility matrix?
2. Matrix Imputation: Extrapolating unknown ratings from known ones. May focus on high unknown ratings. Only care about what you like, ignore what you do not like.
3. Evaluating extrapolation methods: How to measure the performance of recommendation methods?

Gathering Ratings

- Explicit methods:
  - Ask users to rate items (not scalable, biased).
  - Crowdsourcing: pay people to label items.
- Implicit methods:
  - Infer ratings from user actions (e.g., purchase implies high rating).
- In practice, a combination of methods is used.

Extrapolating Utilities

- Key problem: Sparse utility matrix. Most customers rate only a few items.
- Cold start:
  - New items are not rated.
  - New customers have no rating history.
- Approaches:
  1. Content-based: Recommend items similar to previous purchases.
  2. Collaborative: Recommend items that peers bought.
  3. Latent factor based: Compare items/customers in latent space.

Content-based Recommendations

Goal

- Recommend items that are similar to what the customer has already rated highly.

Plan of Action

- Build item profiles from item features.
- From items a user likes, build a user profile.
- Match user profile to item profiles.
- Recommend the closest items.

Item Profile

- Create an item profile matrix P.
- Each row corresponds to an item; each column to a feature.

Examples of features

- Movies: actor, director, genre, etc.
- Text: set of important words.

Challenge

- How to pick important features?

Document Features via TF–IDF

- f_{ij}: frequency of term i in document j.
- $TF_{ij} = \frac{f_{ij}}{\max_k f_{kj}}$.
- n_i: number of documents that contain term i.
- N: total number of documents.
- $IDF_i = \log \frac{N}{n_i}$.
- TF–IDF$(i, j) = TF_{ij} \cdot IDF_i$.

Doc profile

- Set of important terms with top TF–IDF weights.

Recommendation Outline

- **Inputs:** item profile matrix P, utility matrix U.
- **Output:** items unrated by x to recommend to x.

1. Compute user profile $v(x)$.
2. For each unrated item i with profile g_i, score similarity between $v(x)$ and g_i and recommend best.

Step 1: User Profiles (Method 1)

- Weighted average of rated item profiles.

Example: ratings $1, 2, 2, 3$

$$v(x) = \tfrac{1}{4}\big[1 \cdot (1,0,1) + 2 \cdot (0,1,0) + 2 \cdot (1,1,0) + 3 \cdot (1,1,1)\big] = \big(\tfrac{6}{4}, \tfrac{7}{4}, \tfrac{4}{4}\big).$$

Step 1: User Profiles (Method 2)

- Subtract mean rating before averaging.

Example: mean $= 2$

$$v(x) = \tfrac{1}{4}\big[(1-2)(1,0,1) + (2-2)(0,1,0) + (2-2)(1,1,0) + (3-2)(1,1,1)\big] = (0, \tfrac{1}{4}, 0).$$

Step 2: Predictions

$$\mathrm{Sim}(v(x), g_j) = \frac{v(x) \cdot g_j}{\|v(x)\| \, \|g_j\|} = \cos\theta$$

- For each unrated item j, compute similarity.
- Rank and recommend top candidates.

Step 2: Acceleration with LSH

- Hash item profiles with random hyperplanes.
- Hash $v(x)$ into a bucket with same scheme.
- Only compare against items in the same bucket.
- Angle-based similarity $\equiv$ cosine similarity (order preserved).

Pros of Content-based Approach

- **Independence from other users**
  - No cold-start problem with other users.
- **Able to recommend to users with unique tastes**
- **Able to recommend new and unpopular items**
  - No first-rate problem assuming we have the item profile of the new item.
- **Explainable**
  - Can explain recommendations by looking at the user profile content features.

Cons of Content-based Approach

- **Finding appropriate features is non-trivial**
  - E.g. Movies, images…
- **Difficult to recommend for new users**
  - User profile is built from users ratings.
- **Overspecialization**
  - Does not recommend items outside of user's content profile.
  - People can have multiple interests.
  - **Unable to make use of quality judgement of peers.**

Collaborative Filtering (user-based)

- **Goal:** Identify similar users and recommend things that similar users liked.
- Given a user x:
  1. Find a set N of other users whose ratings are similar to x's ratings.
  2. Estimate x's ratings based on ratings of other users in N.

Step 1: Finding Similar Users

- Let r_x be the vector of user x's ratings.
- Jaccard-similarity: Considers r_x as a subset of items, but ignores values of ratings.
- Cosine-similarity: Treats missing ratings as 0 (negative). To normalize the rows, we need to subtract row mean.

Step 2: Predicting Ratings

- Let N = set of k most similar users to x who rated item i.
- Prediction for rating of item i by x:

$$r_{xi} = \frac{1}{k} \sum_{y \in N} r_{yi}$$

- Weighted average:

$$r_{xi} = \frac{\sum_{y \in N} sim(x,y) \cdot r_{yi}}{\sum_{y \in N} sim(x,y)}$$

Item-based Collaborative Filtering

- Step 1: Given an item i, find similar items.
- Step 2: Estimate rating for i based on ratings for similar items.
- Uses similarity metrics between items.

Item-based CF (formulas)

- Let r_{xj} = rating of item j by user x.
- Let $sim(i, j)$ = similarity between i and j.
- Let $N(i; x)$ = set of items rated by x similar to i.

$$r_{xi} = \frac{\sum_{j \in N(i;x)} sim(i,j) \cdot r_{xj}}{\sum_{j \in N(i;x)} sim(i,j)}$$

Item-based vs User-based CF

- Duals in theory, but item-based usually outperforms.
- Intuition: Easier to find similar movies (same genre) than similar users with only single-genre preferences.

Pros and Cons of CF

- + Works for any kind of item: Directly uses utility matrix, no feature engineering.
- - Cold Start: Needs critical mass of users/items, cannot recommend unrated/new items.
- - Sparsity: Hard to find overlaps in sparse matrices.
- - Popularity Bias: Tends to recommend popular items, struggles with unique tastes.

Hybrid Methods

- **Ensembling:** Combine multiple recommenders.
- **Content + CF:**
  - Build item profiles for new items.
  - Use user demographic for new users.

Evaluation

- Ratings matrix split into known and test data set.
- Known ratings used for training, unknown (test set) used for evaluation.

Evaluating Predictions

- Compare predictions with known ratings in test set:
  - Root-Mean Square Error (RMSE):

$$\sqrt{\tfrac{1}{N} \sum_{x,i} (r_{xi} - r^*_{xi})^2}$$

    where r_{xi} is predicted and r^*_{xi} is true rating.
  - Precision at top 10: % of top 10 recommendations that are relevant.
  - Rank correlation: Spearman's correlation between system and user rankings.

Evaluating Predictions without Ratings

- Sometimes only know if an item was watched or not.
- 0/1 model:
  - Coverage: number of items/users system can make predictions for.
  - Precision: accuracy of predictions.
  - ROC: tradeoff curve between false positives and false negatives.

Other Considerations

- Prediction accuracy may not be all that matters.
- Prediction diversity.
- Prediction context.

Complexity of Collaborative Filtering

- Most expensive step: finding the k most similar items.
  - Can be made efficient with LSH.
  - Often pre-computed.
- Clustering helps make utility matrix less sparse, easier to find similar items/users.

BellKor Recommender System

- Winner of the Netflix Challenge.
- Multi-scale modelling of the data:
  - **Global**: Overall deviations of users/movies.
  - **Factorization**: 'Regional' effects.
  - **Collaborative filtering**: Local patterns.

RMSE Evaluation

$$RMSE = \sqrt{\tfrac{1}{|R|} \sum_{(i,x) \in R} (\hat{r}_{xi} - r_{xi})^2}$$

Local and Global Effects

- Global:
  - Overall mean rating = 3.7
  - Dark Knight avg = 4.2 (+0.5 above avg)
  - Joe's avg = 3.5 (−0.2 below avg)
  - Baseline estimate: $3.7 + 0.5 - 0.2 = 4$
- Local neighbourhood (CF):
  - Joe disliked Joker (similar to Dark Knight).
  - Final estimate: 3.8

Modelling Local and Global Effects

$$r_{xi} = \frac{\sum_{j \in N(i;x)} sim(i,j) \cdot r_{xj}}{\sum_{j \in N(i;x)} sim(i,j)}$$

Item-based CF with baseline:

$$r_{xi} = b_{xi} + \frac{\sum_{j \in N(i;x)} sim(i,j) \cdot (r_{xj} - b_{xj})}{\sum_{j \in N(i;x)} sim(i,j)}$$

where $b_{xi} = \mu + b_x + b_i$

- μ = overall mean rating.
- b_x = user x deviation.
- b_i = item i deviation.

Problems with Current Model

1. Similarity measures arbitrary.
2. Pairwise similarities neglect interdependencies.
3. Weighted average restrictive.
   - Solution: Replace $sim(i, j)$ with weights w_{ij}.

Interpolation Weights w_{ij}

$$\hat{r}_{xi} = b_{xi} + \sum_{j \in N(i;x)} w_{ij} (r_{xj} - b_{xj})$$

Remarks:

- Still need similarity measure to form $N(i; x)$.
- $w_{ij} \in \mathbb{R}$ are interpolation weights.
- $\sum w_{ij} \neq 1$ allowed.
- w_{ij} models interactions between movies (not user-dependent).

Finding w_{ij}

- Minimize RMSE:

$$\sqrt{\frac{1}{|R|}\sum_{(i,x)\in R}(\hat{r}_{xi}-r_{xi})^2}$$

- Equivalent to minimizing SSE.
- Approach: minimize SSE on training data.

Optimization Problem

$$J(w)=\sum_{x,i}\Big(b_{xi}+\sum_{j\in N(i;x)}w_{ij}(r_{xj}-b_{xj})-r_{xi}\Big)^2$$

- Convex function.
- Solve with gradient descent:

$$\nabla_w J=\frac{\partial J(w)}{\partial w_{ij}}$$

Gradient Descent Algorithm

- Hyperparameters: K iterations, η learning rate.
- Initialize $\theta_0\in\mathbb{R}^p$.
- For $k=0,\dots,K-1$:

$$\theta_{k+1}=\theta_k-\eta\nabla\Phi(\theta_k)$$

- Return θ_K.

Algorithm 3 Gradient Descent

1: **Input:** Number of iterations K, learning rate η
2: **Initialize:** $\theta_0\in\mathbb{R}^p$
3: **for** $k=0,1,\dots,K-1$ **do**
4: $\quad\theta_{k+1}\leftarrow\theta_k-\eta\nabla\Phi(\theta_k)$
5: **end for**
6: **return** θ_K

Interpolation Weights

$$\hat{r}_{xi}=b_{xi}+\sum_{j\in N(i;x)}w_{ij}(r_{xj}-b_{xj})$$

- Weights w_{ij} derived explicitly to minimize RMSE/SSE.

Latent Factor Models

$$R\approx Q\cdot P^T$$

- Apply "SVD" on Netflix data.
- R has missing entries (ignored initially).
- Reconstruction error minimized on known ratings.

Ratings as Products of Factors

$$\hat{r}_{xi}=q_i\cdot p_x=\sum_f q_{if}p_{xf}$$

- q_i = row i of Q.
- p_x = column x of P^T.

SVD Recap

- $A=U\Sigma V^T$.
- U: left singular vectors.
- V: right singular vectors.
- Σ: singular values.

On Netflix challenge:

$$R\approx Q\cdot P^T,\quad Q=U,\quad P^T=\Sigma V^T$$

SVD Objective

$$\min_{U,V,\Sigma}\sum_{ij\in A}(A_{ij}-[U\Sigma V^T]_{ij})^2$$

- Equivalent to minimizing RMSE.
- **Problem:** Missing entries treated as zeros.

Supervised Matrix Factorization

$$\min_{P,Q}\sum_{(i,x)\in R}(r_{xi}-q_i\cdot p_x)^2$$

- Optimize only over known ratings.
- Solve with gradient descent or SGD.
- Initialize P,Q using SVD.
- P,Q map users/movies to latent space (not orthonormal).

Regularization

$$\min_{P,Q}\sum_{training}(r_{xi}-q_ip_x)^2+\Big[\lambda_1\sum_x\|p_x\|^2+\lambda_2\sum_i\|q_i\|^2\Big]$$

- λ_1,λ_2: regularization parameters.
- Shrinks model when data is scarce.
- Allows complexity with sufficient data.

Effect of Regularization

- Optimization is defined over training set, but goal is generalization.
- Apply L2 regularization:

$$\min_{P,Q}\sum_{(x,i)\in training}(r_{xi}-q_ip_x)^2+\lambda_1\sum_x\|p_x\|^2+\lambda_2\sum_i\|q_i\|^2$$

- Shrinks/simplifies model when data is scarce.
- Still allows complexity when enough data is available.

Latent Factor Model with Biases

$$r_{xi}=\mu+b_x+b_i+q_i\cdot p_x$$

- μ: Overall mean rating.
- b_x: User bias (Joe's avg -0.2).
- b_i: Item bias (Dark Knight $+0.5$).
- Baseline estimate: $3.7+0.5-0.2=4$.
- Add contribution from latent factors.

Making b_x, b_i Learnable

$$\min_{Q,P}\sum_{(x,i)\in R}\Big(r_{xi}-(\mu+b_x+b_i+q_ip_x)\Big)^2+\lambda_1\sum_i\|q_i\|^2+\lambda_2\sum_x\|p_x\|^2+\lambda_3\sum_x\|b_x\|^2+\lambda_4\sum_i\|b_i\|^2$$

- b_x, b_i are trainable parameters.
- λ tuned by grid search on validation set.
- Solved with (stochastic) gradient descent.

Temporal Bias

- Rise in avg movie rating (2004).
- Reasons:
  - Netflix improvements.
  - Meaning of ratings changed.
- Movie age: Older movies usually better.

Modeling Temporal Biases & Factors

$$r_{xi}=\mu+b_x+b_i+q_i\cdot p_x$$
$$r_{xi}(t)=\mu+b_x(t)+b_i(t)+q_i\cdot p_x$$

- $b_i(t)=b_i+b_{i,\mathrm{Bin}(t)}$ (bin = 10 weeks).
- Add time dependence to $p_x(t)$, $q_i(t)$.
- Models user preference dynamics.

BellKor's Pragmatic Chaos (Big Picture)

- Combined ~500 predictors from CF models.
- Probe blending → linear blend.
- Latent user and movie features integrated.
- Achieved 10.09% RMSE improvement (Netflix Prize).

Lecture 6

Basic Steps of Search Engines

1. Crawl the web and locate all web pages (with public access).
2. **Rate the importance of each page in the database.**
3. Given a search query, decide which pages to display in the search result, taking importance into account.

Remark:

- Actual algorithms are secret, but conceptually:
- Text matching first produces candidate websites.
- Then results are **ranked by importance score from step 2**.
- Focus is on step 2.

Web Search Challenges

1. Which web page to trust? Idea: **Trustworthy websites may point to each other**.
2. Some questions are ambiguous: Example: query "newspaper"
   - A newspaper website may not use the word "newspaper" often.
   - Idea: **Pages that actually know about newspapers may point to many newspapers**.

Link Analysis Algorithms

- PageRank
- Hubs and Authorities (HITS)
- Topic-Specific (Personalized) PageRank
- Web Spam Detection Algorithms

PageRank: The "Flow" Formulation Links as Votes

- Idea: Treat links as votes.
- Page importance increases with number of links (incoming/outgoing).
- **Not all in-links equal:**
  - Links from important pages should count more.
  - Recursive definition.

Intuition

- Webpages are more important if many visit them.
- Hard to track visits → Assume people follow links randomly.
- **Random Surfer Model:**
  - Start at random page, follow an out-link at random, repeat.
  - PageRank = limiting probability of being at a page.
- Importance of a page = its share of the importance of each predecessor page.
- Eventually: formulate importance as the **principal eigenvector** of the transition matrix of the Web.

Simple Recursive Formulation

- Each link's vote is proportional to the **importance** of its source page.
- If a page j with importance r_j has n out-links, each out-link gets $\frac{r_j}{n}$ votes.
- The importance of page j is the sum of the votes on its in-links.
  - Example: $r_j=\frac{r_i}{3}+\frac{r_k}{4}$

PageRank: Flow Formulation

- A vote from an important page is worth more.
- A page is important if it has many in-links from important pages.
- Define a **crank** r_j for page j:

$$r_j = \sum_{i \to j} \frac{r_i}{d_i}$$

- d_i = out-degree of node i

Solving the Flow Equations

- 3 equations, 3 unknowns, no constants
  - No unique solution
  - All solutions are equivalent up to scale
- Additional constant for uniqueness:

$$r_y + r_a + r_m = 1$$

- **Solution:** $r_y = \frac{2}{5},\ r_a = \frac{2}{5},\ r_m = \frac{1}{5}$
- Gaussian elimination works for small examples. Another formulation is used for large graphs.

PageRank: Matrix Formulation

- **Stochastic adjacency matrix** M
  - Suppose that page i has d_i out-links
  - If $i \to j$, then $M_{ji} = \frac{1}{d_i}$ else $M_{ji} = 0$
  - M is a column stochastic matrix (columns sum to 1)
- **Rank vector** r**:** vector with an entry per page/node
  - r_i is the importance score of page i
  - $\sum_i r_i = 1$
- Flow equations can be written:

$$r = M \cdot r$$

Eigenvector Formulation

- Flow equations in matrix form: $r = M \cdot r$
- So the rank vector r is an **eigenvector** of the stochastic web matrix M
- In fact, it is the **principal eigenvector**, with corresponding eigenvalue 1
  - Largest eigenvalue of M is 1 since M is column stochastic (non-negative entries)
- As in SVD, we can find r via **power iteration** on M

Power Iteration Method

- **Power iteration:**
  - Suppose there are N web pages
  - Initialize: $r^{(0)} = \left[\frac{1}{N}, \ldots, \frac{1}{N}\right]^T$
  - Iterate: $r^{(t+1)} = M \cdot r^{(t)}$
  - Stop when $|r^{(t+1)} - r^{(t)}|_1 < \varepsilon$
- $|x|_1 = \sum_{1 \le i \le N} |x_i|$ is the L_1 norm (other norms possible)

Random Walk Interpretation

- Imagine a random web surfer:
  - At any time t, surfer is on some page i
  - At time $t+1$, surfer follows an out-link from i uniformly at random
  - Ends up on some page j linked from i
  - Process repeats indefinitely
- Let $p(t)$ be the vector whose ith coordinate is the probability that the surfer is at page i at time t
- So, $p(t)$ is a probability distribution over pages/nodes

The Stationary Distribution

- Where is the surfer at time $t+1$?

$$p(t+1) = M \cdot p(t)$$

- Suppose the random walk reaches a stationary state:

$$p(t+1) = M \cdot p(t) = p(t)$$

- Our original rank vector r satisfies $r = M \cdot r$
- So, r is a **stationary distribution** for the random walk

Existence and Uniqueness

- A central result from the theory of Markov chains:
- For graphs that satisfy **certain conditions**, the stationary distribution is **unique** and eventually will be reached no matter what the initial probability distribution is at time $t = 0$

PageRank: Three Questions

$$r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{d_i} \quad \text{or equivalently} \quad r = Mr$$

- **Does this converge?**
- **Does it converge to what we want?**

PageRank: Problems

1. **Dead ends (nodes with no out-links):**
   - Random walk has nowhere to go.
   - Importance "leaks out."
2. **Spider traps (group of nodes with no links pointing out):**
   - Random walker gets stuck.
   - Eventually absorbs all importance.

Solution: Teleports

- At each time step, the random surfer:
  1. With probability β, follow a random out-link.
  2. With probability $1 - \beta$, jump to a random page.
- Typical β in range 0.8–0.9.
- Surfer will teleport out of traps within a few steps.

Problem: Dead Ends

- No out-links cause rank leakage.

Solution: Always Teleport

- Follow teleport links with probability 1.0 from dead-ends.
- Adjust transition matrix accordingly.

Why Teleports Solve the Problem

- Spider-traps: importance gets stuck.
  - **Solution:** teleport out in finite steps.
- Dead-ends: importance leaks out.
  - **Solution:** always teleport, matrix stays stochastic.

Theory (Markov Chains)

- If M is column-stochastic, irreducible, and aperiodic:
  - Power iteration converges.
  - Unique positive stationary distribution exists.

Periodicity

- Markov chain periodic $\Rightarrow$ return only at multiples of $k > 1$.
- **Solution:** Add self-loops (ensures non-zero prob. of repeating in 1 step).

Irreducibility

- Chain irreducible $\iff$ can reach any state from any other.
- **Solution:** Add random teleports.

Solution: Random Teleports

- At each step:
  1. With probability β, follow a random out-link.
  2. With probability $1 - \beta$, teleport to a random page.
- PageRank equation:

$$r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1 - \beta) \frac{1}{N}$$

The Google Matrix

- **PageRank equation [Brin-Page, '98]**

$$r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1 - \beta) \frac{1}{N}$$

- **The Google Matrix** A**:**

$$A = \beta M + (1 - \beta) \left[\frac{1}{N}\right]_{N \times N}$$

  where all entries are $1/N$.
- A is stochastic, aperiodic, and irreducible. Power method:

$$r^{(t+1)} = A \cdot r^{(t)}$$

  converges to the desired solution.
- What is β? In practice, $\beta = 0.8 \sim 0.9$

Computing Page Rank

- **Power Iteration** $r^{new} = A \cdot r^{old}$
- Easy if enough memory to hold A, r^{old}, r^{new}
- Example: $N = 1$ billion pages
  - Assume 4 bytes per entry.
  - 2 billion entries $\approx$ 8GB.
  - Matrix A has N^2 entries $\Rightarrow$ too big.

Matrix Formulation

- Suppose N pages.
- Page j has d_j out-links.
- $M_{ij} = 1/d_i$ when $j \leftarrow i$, 0 otherwise.
- Random teleport is equivalent to:
  - Add teleport link from j to all pages with prob $(1 - \beta)/N$.
  - Reduce out-link prob from $1/d_j$ to β/d_j.
- Equivalent: Tax each page a fraction $(1 - \beta)$ of score, redistribute evenly.

Rearranging the Equation

$$r = A \cdot r, \quad A_{ji} = \beta M_{ji} + \frac{1 - \beta}{N}$$

$$\begin{aligned} r_j &= \sum_{i=1}^{N} A_{ji} r_i \\ &= \sum_{i=1}^{N} \left[\beta M_{ji} + \tfrac{1-\beta}{N}\right] r_i \\ &= \sum_{i=1}^{N} \beta M_{ji} r_i + \tfrac{1-\beta}{N} \sum_{i=1}^{N} r_i \\ &= \sum_{i=1}^{N} \beta M_{ji} r_i + \tfrac{1-\beta}{N} \quad \text{since } \textstyle\sum r_i = 1 \end{aligned}$$

$$r = \beta M \cdot r + \left[\tfrac{1-\beta}{N}\right]_N$$

Note: assume M has no dead ends.

Sparse Matrix Formulation

- Rearranged PageRank equation:

$$r = \beta M \cdot r + \left[\tfrac{1-\beta}{N}\right]_N$$

- M is a sparse matrix (with no dead ends). E.g., 10 links/node $\approx 10N$ entries.
- Per iteration:
  - Compute $r^{new} = \beta M \cdot r^{old}$.
  - Add constant $(1 - \beta)/N$ to each entry.
- Note: if M has dead ends, must renormalize r^{new} to sum to 1.

PageRank: The Complete Algorithm

- **Input:** Graph G and parameter β
  - Directed graph G (can have spider traps and dead ends)
  - Parameter β
- **Output:** PageRank vector r^{new}
- Set: $r_j^{old} = \frac{1}{N}$
- Repeat until convergence: $\sum_j |r_j^{new} - r_j^{old}| > \varepsilon$
- For all j: $r_j^{new} = \sum_{i \to j} \beta \frac{r_i^{old}}{d_i}$
- $r_j^{new} = 0$ if in-degree of j is 0
- Now re-insert the leaked PageRank: $r_j^{new} = r_j^{new} + \frac{1-S}{N}$ where $S = \sum_j r_j^{new}$
- $r^{old} = r^{new}$

Remark: If the graph has dead-ends, the leaked PageRank is redistributed explicitly.

Sparse Matrix Encoding

- Encode sparse matrix using only nonzero entries.
- Store M: one source node at a time, listing out-degree and destination nodes. Example:

source node	degree	destination nodes
0	3	1, 5, 7
1	5	17, 64, 113, 117, 245

- Space requirement: #edges + #nodes integers.
- Example: $N = 10^9$ nodes, 10 edges per node → 44 GB. Too large → store on disk, not memory.

Basic Algorithm: Update Step

- Assume enough RAM to fit r^{new}.
- Store r^{old} and M on disk.
- One power iteration step:
  - Initialize all entries of $r^{new} = \frac{1-\beta}{N}$.
  - For each page i (with out-degree d_i):
    - Read $i, d_i, dest_1, \ldots, dest_{d_i}, r^{old}(i)$.
    - For $j = 1 \ldots d_i$: $r^{new}(dest_j) += \beta \frac{r^{old}(i)}{d_i}$.

Analysis

- Store r^{old} and M on disk.
- Each iteration requires:
  - Read r^{old} and M.
  - Write r^{new} back to disk.
- Communication cost per iteration: $2|r| + |M|$.
- $|r|$ = #nodes, $|M|$ = #edges + #nodes.

Analysis of Block Update

- Break r^{new} into k blocks that fit in memory.
- Scan M and r^{old} once for each block.
- Total cost:
  - k scans of M and r^{old}.
  - Communication cost per iteration: $k(|M| + |r|) + |r| = k|M| + (k+1)|r|$.
- Challenge: M is much larger than r (10–20x). Avoid scanning M k times per iteration.

Block-Stripe Analysis

- Break M into stripes
  - Each stripe contains only destination nodes in the corresponding block of r^{new}.
- Some additional overhead per stripe (but usually worth it).
- Cost per iteration of Power method:

$$|M|(1+\epsilon) + (k+1)|r|$$

  where $|M|\epsilon$ is for repeated storage of out-degrees.
- Each out-degree is repeatedly stored up to $\min(k,$ corresponding out-degree) times.

Some Problems with PageRank

- Measures generic popularity of a page → **Solution:** Topic-Specific PageRank.
- Uses a single measure of importance → **Solution:** Hubs-and-Authorities.
- Susceptible to link spam (artificial topographies). → **Solution:** TrustRank.

Topic-Specific PageRank

- Goal: Measure popularity within a topic.
- Evaluate web pages by **proximity to a topic**.
- Allows queries to be answered by user interests.

Matrix Formulation

$$A_{ij} = \begin{cases} \beta M_{ij} + \frac{1-\beta}{|S|} & \text{if } i \in S \\ \beta M_{ij} & \text{otherwise} \end{cases}$$

- A is stochastic.
- Computation: multiply by M, then add vector (sparsity preserved).

Discovering the Topic Vector S

- Different PageRanks for different topics (e.g., DMOZ categories).
- Choosing ranking:
  - Advanced search → user selects topic.
  - Classify query into a topic.
  - Use context: e.g., history of queries.
  - User context: bookmarks, preferences.

Application to Measuring Proximity in Graphs

- Random Walk with Restarts: S is a singleton.
- Proximity should account for:
  - Multiple connections.
  - Quality (direct/indirect, length, degree, weight).

SimRank: Idea

- Random walks from a fixed node on k-partite graphs.
- Setting: k-partite graph (e.g., authors, conferences, tags).
- Topic-Specific PageRank: teleport set $S = \{u\}$.
- Scores → similarity to node u.
- Problem: must run once per node u (scalable only sub-Web scale).

PageRank: Summary

- Normal PageRank: teleport uniformly to any node. Example: $S = [0.1, 0.1, 0.1, \ldots]$.
- Topic-Specific PageRank: teleport to topic set. Example: $S = [0.1, 0, 0, 0.2, \ldots]$.
- Random Walk with Restarts: teleport always to same node. Example: $S = [0, 0, 0, 1, 0, 0, 0]$.

Spam on the Web

- **Spamming:** Any deliberate action to boost a web page's position in search engine results.
- **Spam:** Web pages that are the result of spamming.
- **This is a broad definition:** SEO industry might disagree.
- Approximately 10–15% of web pages are spam.

Web Search

1. Crawl the Web.
2. Index pages by the words they contained.
3. Respond to search queries with pages containing those words.

Early page ranking:

- Order pages matching a query by "importance".
- First search engines considered:
  1. Number of times query words appeared.
  2. Prominence of word position (title, header).

First Spammers: Term Spam

- Add the word "movie" 1000 times to a page, hide with background color.
- Or, run query "movie" in a search engine, copy links to popular results, paste them invisibly.
- These techniques are called **term spam**.

Google's Solution to Term Spam

- Disregard self-citation; pay attention to how other pages describe a page.
- Use **anchor text** and its surrounding context.
- Use **PageRank** to measure importance of Web pages.

Why It Works?

- Anchor text: Shirt-seller cannot fool system if others don't say he is about movies.
- PageRank: Page not very important → won't rank high.

What he could try instead:

- Create 1000 pages linking to his website with "movie" in anchor text.
- These pages have no in-links → low PageRank.
- Could still gain unfair advantage, but cannot beat IMDB, etc.

Spam Farms and Link Spam

- **Spam farms:** concentrate PageRank on a single page.
- **Link spam:** creating link structures that boost PageRank artificially.

Link Spamming: Spammer's Perspective

- **Inaccessible pages.**
- **Accessible pages:** e.g., blogs where comments/links can be posted.
- **Owned pages:** controlled by spammer, may span multiple domains.

Link Farms

- **Goal:** maximize PageRank of target page t.
- **Technique:** get links from accessible pages, construct "link farm" for multiplier effect.

Analysis

- x: PageRank contributed by accessible pages.
- y: PageRank of target page t.
- Rank of each "farm" page = $\frac{\beta y}{M} + \frac{1-\beta}{N}$.

$$y = x + \beta M \left[\frac{\beta y}{M} + \frac{1-\beta}{N} \right] + \frac{1-\beta}{N}$$

$$y = x + \beta^2 y + \frac{\beta(1-\beta)M}{N} + \frac{1-\beta}{N}$$

$$y = \frac{x}{1-\beta^2} + c\frac{M}{N}, \quad \text{where } c = \frac{\beta}{1+\beta}$$

- Example: for $\beta = 0.85$, $\frac{1}{1-\beta^2} = 3.6$.
- **Multiplier effect for acquired PageRank:** By making M large, we can make y large.

HITS: Hubs and Authorities

- **HITS (Hypertext-Induced Topic Selection)**
  - A measure of importance of pages or documents, similar to PageRank.
  - Proposed around the same time as PageRank (1998).
- **Goal:** Find good newspapers or "experts" who link in a coordinated way to good newspapers.
- **Idea:** Links act as votes. A page is more important if it has more links.

Finding newspapers

- Each page has 2 scores:
  - **Hub:** Quality as an expert (sum of authority votes pointed to).
  - **Authority:** Quality as content (sum of votes from experts).
- Repeated improvement updates scores.

Classes of Pages

1. **Authorities:** Pages containing useful information
   - Newspaper home pages
   - Course home pages
   - Home pages of car manufacturers
2. **Hubs:** Pages that link to authorities
   - List of newspapers
   - Course bulletin
   - List of car manufacturers

Counting in-links: Authority

- Authorities collect votes from hubs.
- Example: NYT authority score = sum of hub scores pointing to it.
- Initialize each page with hub score = 1.

Mutually Recursive Definition

- A good hub links to many good authorities.
- A good authority is linked from many good hubs.
- Each node has two scores: hub score and authority score, represented as vectors h and a.

Convergence and Matrix Form

- HITS converges to a single stable point.
- Notation:
  - $a = (a_1, \ldots, a_n)$, $h = (h_1, \ldots, h_n)$
  - Adjacency matrix $A \in \mathbb{R}^{N \times N}$, $A_{ij} = 1$ if $i \to j$, else 0
- Hub: $h = A \cdot a$
- Authority: $a = A^T \cdot h$

HITS Algorithm

- Each page i has:
  - Authority score: a_i
  - Hub score: h_i
- Initialize: $a_j^{(0)} = 1/\sqrt{N}, h_j^{(0)} = 1/\sqrt{N}$
- Iterate until convergence:
  - Authority: $a_i^{(t+1)} = \sum_{j \to i} h_j^{(t)}$
  - Hub: $h_i^{(t+1)} = \sum_{i \to j} a_j^{(t)}$
  - Normalize: $\sum_i (a_i^{(t+1)})^2 = 1, \quad \sum_j (h_j^{(t+1)})^2 = 1$

HITS algorithm in vector notation

- Initialize: $a_i = h_i = \frac{1}{\sqrt{n}}$
- Repeat until convergence:
  - $h = A \cdot a$
  - $a = A^T \cdot h$
  - Normalize a and h
- Then: $a = A^T \cdot (A \cdot a)$
  - a is updated in 2 steps: $a = A^T(Aa) = (A^T A)a$
  - h is updated in 2 steps: $h = A(A^T h) = (AA^T)h$
- Convergence criterion:

$$\sum_i (h_i^{(t)} - h_i^{(t-1)})^2 < \epsilon, \quad \sum_i (a_i^{(t)} - a_i^{(t-1)})^2 < \epsilon$$

Existence and Uniqueness

- $A^T A$ and AA^T are real, symmetric ⇒ eigenvectors linearly independent.
- HITS converges to h^* and a^*:
  - h^* is principal eigenvector of AA^T
  - a^* is principal eigenvector of $A^T A$

PageRank vs HITS

- Both answer: *What is the value of an in-link from $u \to v$?*
- PageRank: value depends on links **into** u.
- HITS: value depends on links **out of** u as well.
- Post-1998: PageRank and HITS evolved differently.

Combating Spam

- **Term spam:** detect via statistical text analysis (like email spam).
- **Link spam:** detect/blacklist spam farm structures.
- TrustRank = PageRank with teleport to trusted pages (e.g. .edu, .gov, .org).

TrustRank: Idea

- Principle: **Approximate isolation** — rare for a good page → bad page.
- Sample **seed pages**.
- Oracle (human) labels good vs spam.
- Expensive ⇒ seed set should be small.

Trust Propagation

- Trusted pages = labeled good seeds.
- Perform topic-sensitive PageRank with teleport = trusted pages.
- Propagate trust along links.
- **Solution 1:** Threshold — pages below trust threshold ⇒ spam.

Why good idea?

- **Trust attenuation:** trust decreases with distance in graph.
- **Trust splitting:** many out-links ⇒ less scrutiny, trust split.

Picking Seed Set

- (1) **PageRank:** pick top-k pages by PR; bad pages assumed not high PR.
- (2) **Trusted domains:** .edu, .gov where membership is controlled.

Spam Mass

- In the **TrustRank** model: start with good pages and propagate trust.
- **Complementary view:** What fraction of a page's PageRank comes from spam pages?
- In practice: we don't know all spam pages ⇒ need to estimate.

Spam Mass Estimation (Solution 2)

- r_p = PageRank of page p with teleport set = all pages.
- r_p^+ = PageRank of p with teleport set = trusted pages only.
- Then: fraction of p's PageRank from spam pages:

$$r_p^- = r_p - r_p^+$$

- **Spam mass of p:**

$$\frac{r_p^-}{r_p}$$

- Pages with high spam mass are declared spam.

Lecture 7

Community Detection: The Setting

- We will work with **undirected (unweighted)** networks.
- Graph is large, assume:
  1. Graph fits in main memory.
  2. Graph is too big ⇒ only run linear time algorithms.
- We will look at a **PageRank-based algorithm** for finding dense clusters.
- Runtime proportional to the cluster size (not the graph size).

Idea: Seed Nodes

- Discover clusters based on seed nodes.
- **Given:** seed node s.
- Compute (approximate) **Personalized PageRank (PPR)** around s (teleport set = $\{s\}$).
- Intuition: If s belongs to a nice cluster, the random walk gets trapped inside the cluster.

Algorithm Outline

- Pick a seed node s of interest.
- Run PPR with teleport set = $\{s\}$.
- Sort nodes by decreasing PPR score.
- Sweep over nodes and find good clusters.

What makes a good cluster?

- Maximize within-cluster connections.
- Minimize between-cluster connections.

Graph Cuts

- Cluster quality = function of edge cut.
- Cut-set: edges with only one node in cluster.
- Cut-score: sum of edge weights in cut-set.
- Formula:

$$cut(A) = \sum_{i \in A, j \notin A} w_{ij}$$

Cut Score

- Partition quality: Cut Score.
- Degenerate case: "minimum cut" may isolate single nodes.
- **Problem:**
  - Only considers external connections.
  - Ignores internal cluster connectivity.

Graph Partitioning Criteria

- Criterion: **Conductance (unweighted edges)**.
- Measures connectivity to outside relative to density inside.

$$\phi(A) = \frac{|\{(i,j) \in E : i \in A, j \notin A\}|}{\min(vol(A), 2m - vol(A))}$$

- $vol(A) = \sum_{i \in A} d_i$ (total weight of edges incident to A).
- Balanced partitions tend to result.

Algorithm Outline: Sweep

- Pick seed node s, run PPR with teleport = $\{s\}$.
- Sort nodes by decreasing PPR score.
- Sweep over nodes and find good clusters.
- For each prefix set $A_i = \{u_1, ..., u_i\}$ compute $\phi(A_i)$.
- Local minima of $\phi(A_i)$ correspond to good clusters.

Computing the Sweep

- Sweep curve can be computed in **linear time**.
- Loop over nodes; maintain $A_i = \{u_1, ..., u_i\}$.
- Update:

$$\phi(A_{i+1}) = \frac{cut(A_{i+1})}{vol(A_{i+1})}$$

- Recurrences:

$$vol(A_{i+1}) = vol(A_i) + d_{i+1}$$

$$cut(A_{i+1}) = cut(A_i) + d_{i+1} - 2\#(\text{edges of } u_{i+1} \text{ to } A_i)$$

Computing PPR

- **Challenge:** How to compute PPR without touching the whole graph?
- Power method won't work: each iteration accesses all nodes.

$$r^{(t+1)} = \beta M \cdot r^{(t)} + (1-\beta)a$$

- a: teleport vector, $a = [0 \ldots 010 \ldots 0]^T$ (seed node s).
- r: personalized PageRank vector.
- **Approximate PageRank** (s, β, ϵ)
  - s: seed node
  - β: teleportation parameter
  - ϵ: approximation error parameter

Approximate PPR: Overview

- **Lazy random walk:** with prob. 1/2 stay put, with prob. 1/2 walk to random neighbor.

$$r_u^{(t+1)} = \tfrac{1}{2} r_u^{(t)} + \tfrac{1}{2} \sum_{i \to u} \frac{1}{d_i} r_i^{(t)}, \quad d_i = \deg(i)$$

- Track **residual PPR score** q_u.
  - $r_u^{(t)}$: PPR estimate of node u at time t.
  - q_u: residual = underestimated PPR at time t.
  - If $\frac{q_u}{d_u} \geq \epsilon$, push walk further (distribute residual to neighbors).

Push Operation

- r: approximate PPR, q: residual PPR.
- Init: all residual at seed node ($r = 0$, $q = a$).
- Iteratively **push** PageRank from q to r until q small.
- Push = one step lazy random walk from node u.

$$\text{Push}(u, r, q): \quad r'_u = r_u + (1-\beta)q_u, \quad q'_u = \tfrac{1}{2}\beta q_u, \quad q'_v = q_v + \tfrac{1}{2}\beta \frac{q_u}{d_u} \ \forall v : u \to v$$

Intuition Behind Push

- Large q_u: underestimated r_u.
- Transfer $(1-\beta)q_u$ from q to r.
- Keep $\frac{1}{2}\beta q_u$, spread rest to neighbors:

$$q_v \leftarrow q_v + \tfrac{1}{2}\beta \frac{q_u}{d_u}$$

- Each node keeps distributing residual until all residual small.

ApproxPageRank Algorithm

- Init: $r = 0$, $q = [0 \dots 010 \dots 0]$ (seed index S).
- While $\max_{u \in V} \frac{q_u}{d_u} \geq \epsilon$:
  - Pick u with $\frac{q_u}{d_u} \geq \epsilon$.
  - Apply **Push**(u, r, q).
- Return r.

Observations

- Runtime: terminates in at most $\frac{1}{\epsilon(1-\beta)}$ iterations, independent of $|V|$.
- Guarantee: if there exists cut with conductance ϕ and volume k, finds cut with conductance $O\left(\sqrt{\frac{\phi}{\log k}}\right)$.
- Smaller $\epsilon \Rightarrow$ farther random walk will spread.

Network Communities

- **Communities:** sets of tightly connected nodes.
- **Modularity:** measure of how well a network is partitioned into communities.
- Given partitioning of network into groups $s \in S$:

$$Q \propto \sum_{s \in S} [\#\text{edges within } s - \mathbb{E}[\#\text{edges within } s]]$$

- Requires a **null model**.

Null Model: Configuration Model

- Given real graph G with n nodes, m edges, construct rewired network G'.
- Same degree distribution but random connections.
- G' is a multigraph.
- Expected # edges between i, j of degrees k_i, k_j:

$$\frac{k_i k_j}{2m}$$

- Exercise: $\frac{1}{2} \sum_i \sum_j \frac{k_i k_j}{2m} = m$.

Modularity Formula

- Partitioning S of G:

$$Q(G, S) = \frac{1}{2m} \sum_{s \in S} \sum_{i \in s} \sum_{j \in s} \left(A_{ij} - \frac{k_i k_j}{2m}\right)$$

- Range: $-1 \leq Q \leq 1$.
- $Q > 0$ if groups have more edges than null model.
- $Q > 0.3 - 0.7 \implies$ significant community structure.
- Communities identified by maximizing Q.

Louvain Algorithm (Overview)

- Greedy algorithm for community detection.
- Runtime: $O(n \log n)$.
- Supports weighted graphs.
- Provides hierarchical partitions.
- Works well for large graphs: fast, converges quickly, outputs high modularity.

Louvain Algorithm: High Level

- Greedily maximizes modularity in 2 phases:
  1. **Phase 1:** Optimize modularity via local node moves.
  2. **Phase 2:** Aggregate communities into super-nodes.
- Repeat until no increase in modularity.

Louvain Phase 1 (Partitioning)

- Init: each node is own community.
- Sweep nodes in random order:
  1. Compute modularity gain ΔQ for moving i into neighbor j's community.
  2. Move i to community with largest ΔQ.
- Repeat sweeps until no changes occur.
- Order of sweep affects result $\Rightarrow$ choose new random order each pass.

Louvain: Modularity Gain

$$\Delta Q(i \to C) = \frac{1}{2m}\left[\Sigma_{in}^{(C)} + 2k_{i,in}^{(C)} - \frac{(\Sigma_{tot}^{(C)} + k_i)^2}{2m}\right] - \frac{1}{2m}\left[\Sigma_{in}^{(C)} - \frac{(\Sigma_{tot}^{(C)})^2}{2m} + \frac{k_i^2}{2m}\right]$$

- $\Sigma_{in}^{(C)}$: $2\times$ sum of internal link weights.
- $\Sigma_{tot}^{(C)}$: $2\times$ sum of all link weights incident to C.
- $k_{i,in}$: sum of link weights between i and C.
- k_i: degree (sum of all link weights of i).
- $\Delta Q = \Delta Q(i \to C) + \Delta Q(D \to i)$ if removing from old community D.

Louvain Phase 2 (Restructuring)

- Contract communities into super-nodes.
- Weighted network: edge weight = sum of weights across partitions.
- Repeat until community configuration stabilizes.

Graph Representation Learning

Machine Learning Lifecycle

- Supervised Machine Learning requires feature engineering for every task.

Feature Learning in Graphs

- Goal: Efficient task-independent feature learning for machine learning in networks.

$$f : u \mapsto \mathbb{R}^d$$

- Node $\to$ vector representation (embedding).

Why Network Embedding?

- Task: Map each node in a network to a point in a low-dimensional space.
- Distributed representation for nodes.
- Similarity of embeddings indicates similarity in the network.
- Encode network information into node representations.

Why is it Hard?

- Deep learning toolbox is designed for simple sequences or grids: CNNs for fixed-sized images/grids and RNNs or word2vec for text/sequences.
- Networks are more complex:
  - Complex topological structure (no spatial locality like grids).
  - No fixed ordering or reference point.
  - Often dynamic and multimodal.

Embedding Nodes: Setup

- Graph $G = (V, A)$. V: vertex set, A: adjacency matrix (binary).
- No node features or extra information used.

Embedding Nodes: Goal

- Encode nodes so that similarity in embedding space (e.g., dot product) approximates similarity in the original network.
- Need to define similarity in the original network.

Learning Node Embeddings

1. Define an encoder (mapping nodes to embeddings).
2. Define a node similarity function (measure of similarity in original network).
3. Optimize the encoder parameters so that:

$$\text{similarity}(u, v) \approx z_v^\top z_u$$

Two Key Components

- **Encoder:** maps each node v to a d-dimensional vector embedding

$$\text{ENC}(v) = z_v$$

- **Similarity function:** specifies how vector space relations map to network relations

$$\text{similarity}(u, v) \approx z_v^\top z_u$$

"Shallow" Encoding

$$\text{ENC}(v) = Zv$$

$$Z \in \mathbb{R}^{d \times |V|}, \quad v \in \mathbb{I}^{|V|}$$

- Z: matrix, each column is a d-dimensional node embedding.
- v: indicator vector (all zero except a 1 for node v).

Each node has a unique embedding vector (methods: DeepWalk, node2vec, LINE).

How to Define Node Similarity?

- Should nodes be similar if they:
  - Are connected?
  - Share neighbors?
  - Have similar "structural roles"?

Random Walk Approaches

- **Random Walk Embeddings:**

$$z_u^\top z_v \approx \Pr[u, v \text{ co-occur on random walk}]$$

- Estimate probability of visiting v from u under random walk R: $P_R(v|u)$.
- Optimize embeddings so that cosine similarity reflects $P_R(v|u)$.

Why Random Walks?

1. **Expressivity:** captures both local and higher-order neighborhood information.
2. **Efficiency:** only consider pairs that co-occur on random walks.

Unsupervised Feature Learning

- **Intuition:** embed nodes in d-dim space so similarity is preserved.
- **Idea:** nearby nodes in network $\to$ close in embedding space.
- **Given node u:** define nearby nodes via neighborhood $N_R(u)$ from random walk strategy R.

Feature Learning as Optimization

- Given graph $G = (V, E)$.
- Goal: Learn a mapping $z : u \to \mathbb{R}^d$.
- Log-likelihood objective:

$$\max_z \sum_{u \in V} \log P(N_R(u) \mid z_u)$$

- $N_R(u)$ = neighborhood of u.

Key Idea: Given node u, learn feature representations predictive of nodes in $N_R(u)$.

Random Walk Optimization

1. Run short fixed-length random walks from each node using strategy R.
2. For each node u, collect $N_R(u)$ = multiset of nodes visited on walks.
3. Optimize embeddings to predict $N_R(u)$ from u.

$$\max_z \sum_{u \in V} \log P(N_R(u) \mid z_u)$$

$N_R(u)$ may have repeat elements.

Likelihood Factorization

$$\log P(N_R(u) \mid z_u) = \sum_{v \in N_R(u)} \log P(z_v \mid z_u)$$

Softmax parametrization:

$$P(z_v \mid z_u) = \frac{\exp(z_v \cdot z_u)}{\sum_{n \in V} \exp(z_n \cdot z_u)}$$

Random Walk Loss Function

$$\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log \frac{\exp(z_u^\top z_v)}{\sum_{n \in V} \exp(z_u^\top z_n)}$$

- Nested sums give $O(|V|^2)$ complexity.
- Normalization term from softmax is expensive.

Negative Sampling Approximation

$$\log \frac{\exp(z_u^\top z_v)}{\sum_{n\in V}\exp(z_u^\top z_n)} \approx \log(\sigma(z_u^\top z_v)) - \sum_{i=1}^{k}\log(\sigma(z_u^\top z_{n_i}))$$

with $n_i \sim P_V$.

- σ: sigmoid.
- Normalize against k random negative samples n_i.
- Sample negatives proportional to degree.
- Typical k: 5–20.

Random Walks: Summary

1. Run short fixed-length random walks from each node.
2. Collect $N_R(u)$ from walks.
3. Optimize embeddings with SGD:

$$\mathcal{L} = \sum_{u\in V}\sum_{v\in N_R(u)} -\log P(z_v \mid z_u)$$

4. Efficiently approximated via negative sampling.

How Should We Walk?

- So far: Optimize embeddings given random walk statistics.
- Key question: What strategies for random walks?
- Simplest: Fixed-length, unbiased random walks (DeepWalk, Perozzi et al. 2013).
- Limitation: Too constrained a notion of similarity.
- Need: Generalization of walk strategies.

Overview of node2vec

- **Goal:** Embed nodes with similar network neighbourhoods close in the feature space.
- Frame this as prediction-task independent maximum likelihood optimization.
- **Key observation:** Flexible notion of network neighbourhood $N_R(u)$ of node $u \rightarrow$ rich embeddings.
- Develop biased 2nd order random walk R to generate $N_R(u)$.

node2vec: Biased Walks

- Idea: Flexible biased random walks trade off between local and global views of the network.
- Two strategies to define $N_R(u)$:
  - BFS $\rightarrow$ local microscopic view.
  - DFS $\rightarrow$ global macroscopic view.

Interpolating BFS and DFS

- Biased fixed-length random walk R generates $N_R(u)$.
- Two parameters:
  - Return parameter p: return to previous node.
  - In-out parameter q: DFS (outward) vs BFS (inward).

Biased Random Walks

- 2nd-order biased walks explore neighbourhoods.
- Walker at node w after traversing (s_1, w).
- Next move depends on:
  - Back to s_1: probability $\frac{1}{p}$.
  - Same distance as s_1: probability 1.
  - Further away: probability $\frac{1}{q}$.
- p, q model transition probabilities.
- BFS-like walk: small p. DFS-like walk: small q.

Node2vec Algorithm

1. Compute random walk probabilities.
2. Simulate r random walks of length l from each node u.
3. Optimize the node2vec objective using SGD.

- Linear-time complexity.
- All steps parallelizable.

BFS vs DFS

- BFS: Micro-view of neighbourhood.
- DFS: Macro-view of neighbourhood.

Experiments: Micro vs Macro

- $p = 1, q = 2 \rightarrow$ microscopic neighbourhood.
- $p = 1, q = 0.5 \rightarrow$ macroscopic neighbourhood.

Other Random Walk Ideas

- **Different kinds of biased random walks:** Based on node attributes (Dong et al., 2017); Based on learned weights (Abu-El-Haija et al., 2017).
- **Alternative optimization schemes:**
  - Directly optimize based on 1-hop and 2-hop random walk probabilities (e.g., LINE, Tang et al. 2015).
- **Network pre-processing techniques:**
  - Run random walks on modified versions of the original network (e.g., Ribeiro et al. 2017's struct2vec, Chen et al. 2016's HARP).

How to Use Embeddings

- **Clustering/community detection:** Cluster nodes/points based on z_i.
- **Node classification:** Predict label $f(z_i)$ of node i based on z_i.
- **Link prediction:** Predict edge (i, j) based on $f(z_i, z_j)$.
  - Concatenate: $f(z_i, z_j) = g([z_i, z_j])$.
  - Hadamard: $f(z_i, z_j) = g(z_i * z_j)$.
  - Sum/Avg: $f(z_i, z_j) = g(z_i + z_j)$.
  - Distance: $f(z_i, z_j) = g(\|z_i - z_j\|_2)$.

Lecture 8

Data Streams

- In data mining, we **may not have the entire dataset in advance**.
- **Stream Management** is important when the **data input rate is not fixed**
- We can think of the data as **infinite** and **non-stationary** (distribution changes over time).
- Specialized algorithms are needed.

The Stream Model

- Input **elements** enter at a rapid rate through one or more input ports (*streams*).
- Elements of the stream are called **tuples**.
- The system cannot store the entire stream in memory.
- **Key Question:** How to make critical calculations about the stream using limited memory?

Problems on Data Streams

- **Sampling data from a stream:**
  - Construct a random sample.
- **Queries over sliding windows:**
  - Count number of items of type x in the last k elements.
- **Filtering a data stream:**
  - Select elements with property x from the stream.
- **Counting distinct elements:**
  - Number of distinct elements in the last k elements.
- **Estimating moments:**
  - Estimate average / standard deviation of last k elements.
- **Finding frequent elements.**

Sampling from a Data Stream

- Since we cannot store the entire stream, one approach is to **store a sample**.
- Two different problems:
  1. Sample a **fixed proportion** of elements in the stream.
  2. Maintain a **random sample of fixed size** over a potentially infinite stream.
- At any "time" k, we would like a random sample of s elements.
  - **Goal:** For all time steps k, each of k elements seen so far has equal probability of being sampled.

Problem 1: Sampling a Fixed Proportion

- **Scenario:** Search engine query stream.
  - Stream of tuples: (user, query, time).
  - Example question: How often did a user run the same query in a single day?
  - Enough space to store 1/10 of the query stream.
- **Naïve solution:**
  - Generate a random integer in [0, 9] for each query.
  - Store the query if the integer is 0, otherwise discard it.

Problem with Naïve Approach

- Question: What fraction of unique queries by an average user are duplicates?
- Suppose each user issues x queries once and d queries twice (total of $x + 2d$ queries).
- Correct answer: $\frac{d}{x+d}$.

Proposed Solution: Keep 10% of the Queries

- Sample will contain $\frac{x+2d}{10}$ queries.
- Expected number of duplicates:

$$\frac{1}{10}\times\frac{1}{10}\times d = \frac{d}{100}$$

- Expected number of unique queries:

$$\frac{x+2d}{10} - \frac{d}{100} = \frac{10x+19d}{100}$$

- Sample-based estimate of fraction of unique queries that are duplicates:

$$\frac{d}{100} \div \frac{10x+19d}{100} = \frac{d}{10x+19d}$$

- Very different from correct answer $\frac{d}{x+d}$.

Solution: Sample Users

- **Pick 1/10 of users** and take all their queries in the sample.

Generalized Solution

- Stream of tuples with keys:
  - Key is some subset of each tuple's components (e.g. user, search, time).
  - Example: key = user.
- To get a sample of a/b fraction of the stream:
  - Hash each tuple's key uniformly into b buckets.
  - Pick the tuple if its hash value $\leq a$.
- Example: For 30% sample, hash into $b = 10$ buckets and take tuples in the first 3 buckets.

Problem 2: Maintaining a Fixed-Size Sample

- Suppose we maintain a random sample S of size exactly s tuples.
- Non-trivial since the length of the stream is not known in advance.
- Suppose after time n, we have seen n samples.
- **Goal:** Each item to be in S with equal probability s/n.

Solution: Fixed Size Sample (Reservoir Sampling, size = s)

- Store first s elements of the stream in S.
- For each new element (the n^{th}, where $n > s$):
  - With probability s/n, keep the n^{th} element, else discard it.
  - If kept, replace one of the existing s elements in S uniformly at random.
- **Claim:** This algorithm maintains the desired property:
  - After n elements, each element seen so far is in S with probability s/n.

Proof by Induction

- Assume after n elements, each element is in S with probability s/n.
- Need to show: after element $n+1$, the property still holds — i.e., each element is in S with probability $s/(n+1)$.

Base Case

- For $n = s$, each of the s elements is in the sample with probability $s/s = 1$.

Inductive Step

- For elements already in S, probability it stays after $(n+1)^{th}$ element:

$$\left(1 - \frac{s}{n+1}\right) + \frac{s}{n+1}\times\frac{s-1}{s} = \frac{n}{n+1}$$

- Hence, probability tuple is in S at time $n+1$:

$$\frac{s}{n}\times\frac{n}{n+1} = \frac{s}{n+1}$$

Sliding Windows

- **Goal:** Answer queries about a **window of length** N – the N most recent elements received.
- **Interesting case:** N is so large that the data **cannot be stored in memory or on disk**.
- Alternatively, there may be many streams such that windows for all cannot be stored.

Counting Bits

- **Problem:** Given a stream of 0s and 1s.
- Want to answer queries such as: *How many 1s are in the last k bits?*, for any $k \leq N$.
- **Obvious solution:** Store the most recent N bits. When a new bit arrives, discard the $(N+1)^{st}$ bit.
- However, you cannot get an exact answer without storing the entire window.
- What if we cannot afford to store N bits? The number of streams or the size of N may be too large.
- **Goal:** Find an **approximation**.

First Simple Attempt

- Question: *How many 1s in the last N bits?*
- A simple solution that assumes **uniformity** (not always true).
- Maintain two counters:
  - S: number of 1s from the beginning of the stream.
  - Z: number of 0s from the beginning of the stream.
- Estimate number of 1s in last N bits:

$$N \cdot \frac{S}{S+Z}$$

- **Problem:** What if the stream is non-uniform (distribution changes over time)?

DGIM Method

- **DGIM solution** does not assume uniformity.
- Stores $O(\log^2 N)$ bits per stream.
- Provides an approximate answer with at most 50% error.
  - Example: If there are 10 ones, 50% error means between 5 and 15.
  - Error factor can be reduced with more complex algorithms and additional stored bits.

Idea: Exponential Windows

- Summarize **exponentially increasing** regions of the stream, looking backward.
- Drop small regions if they begin at the same point as a larger region.
- Example:
  - Window of width 16 has 6 ones.
  - Some regions overlap; we can reconstruct counts of last N bits, except uncertain about the last block.
- **Observation:** We can reconstruct count approximately, except for some uncertainty in the last region.

Benefits

- Stores only $O(\log^2 N)$ bits.
  - $O(\log N)$ counts of $O(\log_2 N)$ bits each.
- Easy update as new bits arrive.
- Error in count is no greater than the number of 1s in the "unknown" area.

Disadvantages

- If 1s are evenly distributed, the unknown-region error is small ($\leq 50\%$).
- However, if all 1s are in the unknown region, the error is **unbounded**.

Fixup: DGIM Method (Datar, Gionis, Indyk, Motwani)

- Instead of summarizing fixed-length blocks, summarize blocks with a fixed number of 1s.
- Block sizes (number of 1s) increase exponentially.
- When there are few 1s in the window, block sizes remain small, so the approximation error is small.

DGIM Method: Approximate Counting over Sliding Windows

- Goal: Estimate the number of 1s in the last N bits of a data stream.
- Challenge: Cannot store all N bits when N is huge.
- Idea: Maintain an **approximate summary** using **buckets**.

Timestamps

- Each bit in the stream has a **timestamp** (1, 2, 3, ...).
- Record timestamps **modulo** N (the window size), so any relevant timestamp can be represented in $O(\log_2 N)$ bits.

Buckets

- A **bucket** stores:
  1. The timestamp of its end [$O(\log N)$ bits]
  2. The number of 1s it represents [$O(\log \log N)$ bits]
- **Constraint:** The number of 1s in a bucket must be a **power of 2**.

Representing a Stream by Buckets

- Properties maintained:
  - Either **one or two** buckets with the same power-of-2 number of 1s.
  - Buckets **do not overlap** in timestamps.
  - Buckets are **sorted by size**: earlier buckets are not smaller than later buckets.
  - Buckets disappear when their end-time is $> N$ time units in the past.

Example: Bucketized Stream

- Example stream maintains:
  - One or two buckets with same power-of-2 count.
  - Non-overlapping timestamps.
  - Buckets sorted by size.

Updating Buckets

- When a new bit arrives, drop the oldest bucket if its end-time is older than N time units.
- Two cases depending on the new bit:
  - If current bit is **0:** No other changes.
- If current bit is **1:**
  1. Create a new bucket of size 1 (timestamp = current time).
  2. If there are now 3 buckets of the same size, **merge the oldest two** into one bucket of double the size.
  3. Repeat merging upward (size $1 \to 2 \to 4$, etc.).

Querying the DGIM Summary

- To estimate number of 1s in the most recent N bits:
  1. Sum sizes of all buckets except the last.
  2. Add **half the size of the last bucket**.
- Rationale: We don't know how many of the last bucket's bits fall within the window.

Error Bound: Proof

- Claim: **Error $\leq$ 50%**.
- Suppose the last bucket has size 2^r.
- Assuming half (2^{r-1}) of its 1s are within window, we err by at most 2^{r-1}.
- Since there is at least one bucket of each smaller size:

$$1 + 2 + 4 + \cdots + 2^{r-1} = 2^r - 1$$

- Thus, relative error $\leq 50\%$.

Reducing the Error Further

- Maintain $r-1$ or r buckets per size ($r > 2$) instead of 1 or 2.
- Error bound becomes at most a fraction $\frac{1}{r-1}$.
- Trade-off: larger r means smaller error but more storage.

Extensions

- For queries of the form "How many 1s in the last k bits?", where $k < N$:
  - Find earliest bucket B overlapping with k.
  - Answer = (sum of sizes of later buckets) + $\frac{1}{2}$ (size of B).
- The same approach generalizes to streams of integers, estimating the sum over recent windows.

Filtering Data Streams

- Each element of a data stream is a **tuple**.
- Given a list of keys S, determine which tuples of the stream are in S.
- **Obvious solution:** Use a hash table.
  - However, we **may not have enough memory** to store all of S.
  - We may need to process millions of filters simultaneously.

First Cut Solution

- Given a set of keys S to filter:
  1. Create a bit array B of n bits, initially all 0s.
  2. Choose a hash function h with range $[0, n)$.
  3. For each $s \in S$, set $B[h(s)] = 1$.
  4. For each stream element a, output a if $B[h(a)] == 1$.
- This outputs an item if it hashes to a position that was set by some $s \in S$.

First Cut Solution: Behavior

- If an item hashes to a bit set to 0, it is surely not in $S \Rightarrow$ **drop it**.
- If an item hashes to a bit set to 1, it **may be in** $S \Rightarrow$ **output it**.
- Hence, the scheme **creates false positives but no false negatives**.
  - If an item is in S, it will always be output.
  - If an item is not in S, it may still be output.

First Cut Solution: Example

- $|S| = 10^9$ email addresses, $|B| = 1$ GB $= 8 \times 10^9$ bits.
- If an address is in S, it hashes to a bit set to $1 \Rightarrow$ always passes (**no false negatives**).
- About 1/8 of bits are set to $1 \Rightarrow$ roughly 1/8 of addresses not in S pass (**false positives**).

Analysis: Throwing Darts

- More accurate analysis of false positives:
- Suppose we throw m darts into n targets uniformly at random.
- What is the probability that a target gets at least one dart?
  - **Targets** $\to$ bits/buckets.
  - **Darts** $\to$ hash values of items.
- Probability a particular target is not hit:

$$\left(1 - \frac{1}{n}\right)^m$$

- Probability a target is hit at least once:

$$1 - \left(1 - \frac{1}{n}\right)^m \approx 1 - e^{-m/n}$$

- As $n \to \infty$, this approaches $1 - 1/e$ when $m = n$.
- Fraction of 1s in bit array B = probability of false positive:

$$P(\text{bit} = 1) = 1 - e^{-m/n}$$

- Example: $m = 10^9$ darts, $n = 8 \times 10^9$ targets:

$$1 - e^{-1/8} = 0.1175$$

- Matches approximate earlier estimate of $1/8 = 0.125$.

Bloom Filter

- Generalization using multiple hash functions.
- Given $|S| = m$, $|B| = n$, and k independent hash functions $h_1, h_2, \ldots, h_k$:
  - **Initialization:**
    - Set all bits in B to 0.
    - For each $s \in S$, set $B[h_i(s)] = 1$ for all $i = 1, \ldots, k$.
  - **Run-time:**
    - When a stream element x arrives:
      - If all $B[h_i(x)] = 1$, declare $x \in S$.
      - Otherwise, discard x.

Bloom Filter Analysis

- Each insertion throws $k \times m$ darts into n targets.
- Fraction of bits set to 1:

$$1 - e^{-km/n}$$

- For a query, the false positive probability is:

$$(1 - e^{-km/n})^k$$

- Trade-offs:
  - Increasing k reduces false positives up to an optimal point.
  - More k also increases computation.
- Example: $m = 1$ billion, $n = 8$ billion
  - $k = 1$: $(1 - e^{-1/8}) = 0.1175$
  - $k = 2$: $(1 - e^{-1/4})^2 = 0.0493$
- **Optimal** value of k: $\frac{n}{m} \ln(2)$
  - In our case: $k = 8 \ln(2) = 5.54 \approx 6$
  - Error at $k = 6$: $(1 - e^{-3/4})^6 = 0.0216$

Bloom Filter – Wrap-up

- **Bloom filters guarantee no false negatives** and use limited memory.
  - Great for pre-processing before more expensive checks.
- **Suitable for hardware implementation.**
  - Hash function computations can be parallelized.
- **One big B or k small Bs?**
  - Equivalent: $(1 - e^{-km/n})^k$ vs. $(1 - e^{-m/(n/k)})^k$.
  - Keeping one big B is simpler.

Counting Distinct Elements

- **Problem:**
  - Data stream consists of elements from a universe of size N.
  - Maintain a count of the number of distinct elements seen so far.
- **Obvious approach:**
  - Maintain a hash table of all distinct elements seen so far.
  - Impractical when stream size is large.

Using Small Storage

- **Real problem:** What if we do not have enough space to store all elements?
- Estimate the count in an **unbiased way**.
- Accept small error, but limit probability that the error is large.

Flajolet-Martin Approach

- Choose a hash function h mapping each element to at least $\log_2 N$ bits.
- For each stream element a, let $r(a)$ be the number of trailing 0s in $h(a)$. $r(a)$ = position of first 1 counting from the right. For instance, $h(a) = 12 \to 1100_2 \Rightarrow r(a) = 2$.
- Record $R = \max_a r(a)$ across all seen elements.
- Estimated number of distinct elements: 2^R.

Why It Works: Intuition

- $h(a)$ hashes a uniformly at random among N values.
- $h(a)$ is a sequence of $\log_2 N$ bits.
- A fraction 2^{-r} of all as have a tail of r zeros.
  - 50% of as end with *0.
  - 25% of as end with *00.
- If we saw longest tail $r = 2$, we've likely seen about $2^r = 4$ distinct elements.
- Hence, it takes about 2^r items before we see one with a zero-suffix of length r.

Why It Works: More Formally

- We show that probability of finding a tail of r zeros: $\to 1$ if $m \gg 2^r$, $\to 0$ if $m \ll 2^r$. where m = number of distinct elements seen so far.
- Thus, 2^R will almost always be around m.
- Probability that a given $h(a)$ ends in at least r zeros: 2^{-r}.
- Probability of not seeing a tail of length r among m elements:

$$(1 - 2^{-r})^m$$

- Note: $(1 - 2^{-r})^m \approx e^{-m2^{-r}}$.
- Probability of not finding tail of length r:
  - If $m \ll 2^r$, then $e^{-m2^{-r}} \approx 1 \Rightarrow$ low chance of finding one.
  - If $m \gg 2^r$, then $e^{-m2^{-r}} \approx 0 \Rightarrow$ high chance of finding one.
- Therefore, 2^R will almost always approximate m.

Why It Doesn't Work (Variance Issue)

- $\mathbb{E}[2^R]$ is actually infinite:
  - Probability halves when $R \to R + 1$, but 2^R doubles.
- **Fix:** Use multiple hash functions h_i to get many samples of R_i.
- Combine samples R_i:
  - **Average?** Sensitive to large outliers.
  - **Median?** Better, since all estimates are powers of 2.
  - **Robust solution:**
    - Partition samples into small groups.
    - Compute mean within each group.
    - Take the median of those means.

Moments – Definition

- Suppose a stream has elements chosen from a set A of N values.
- Let m_i be the number of times value i occurs in the stream.
- The k^{th} moment is:

$$\sum_{i \in A} (m_i)^k$$

- Examples:
  - $k = 1$: Total count of all elements.
  - $k = 2$: Frequency moment (used in AMS algorithm).
  - $k = 3$: Skewness — asymmetry of the distribution.
  - $k = 4$: Kurtosis — "peakedness" or tail heaviness.

AMS Method (Alon, Matias, Szegedy)

- Gives an **unbiased estimate**.
- Works for all moments k.
- Focus on the **2nd moment** $S = \sum_i m_i^2$.
- Maintain several random variables X:
  - Each X stores $(X.el, X.val)$.
  - $X.el$ = item i at random time t.
  - $X.val$ = count m_i of item i starting from t.
- Goal: compute $S = \sum_i m_i^2$.

One Random Variable X

- Assume stream length n (later relaxed).
- Pick a random time $t < n$ (uniformly).
- Let stream element at time t be item i:
  - Set $X.el = i$.
  - Maintain $c = X.val$ = number of is seen in the stream from t onwards.
- Then estimate of the 2nd moment:

$$S = f(X) = n(2c - 1)$$

- With multiple Xs: $S = \frac{1}{k} \sum_{j=1}^{k} f(X_j)$.

Expectation Analysis

- 2nd moment: $S = \sum_i m_i^2$.
- Let c_t = number of times item i appears from time t onwards.
- $\mathbb{E}[f(X)] = \frac{1}{n} \sum_t n(2c_t - 1)$.
- Grouping by item i:

$$\mathbb{E}[f(X)] = \frac{1}{n} \sum_i n(1 + 3 + 5 + \cdots + 2m_i - 1)$$

- Simplify:

$$(1 + 3 + 5 + \cdots + 2m_i - 1) = m_i^2$$

- Thus:

$$\mathbb{E}[f(X)] = \sum_i (m_i)^2 = S$$

- So $f(X)$ is an **unbiased estimator** of the 2nd moment.

Higher-Order Moments

- Same algorithm, different formula:

$$f(X) = n(c^k - (c - 1)^k)$$

- Examples:
  - For $k = 2$: $n(2c - 1)$.
  - For $k = 3$: $n(3c^2 - 3c + 1)$.
- Derived from telescoping sum logic:

$$\sum_{c=1}^{m} (c^k - (c - 1)^k) = m^k$$

Combining Samples in Practice

- Compute $f(X) = n(2c - 1)$ for as many Xs as memory allows.
- Average within groups.
- Take the median of group averages (reduces variance).

Handling Infinite Streams (Fixups)

1. Separate the n factor — store only counts in X, maintain n externally.
2. If we can store only k variables:
   - Each starting time t is selected with probability k/n.
   - Use **fixed-size reservoir sampling**:
     - Keep the first k items.
     - For $n > k$, choose new element with probability k/n.
     - If chosen, replace one existing stored variable uniformly at random.

Exponentially Decaying Windows

Counting Items

- **New Problem:** Given a stream, which items appear more than s times in the window?
- **Possible solution:** Treat the stream as one binary stream per item:
  - Each basket $\to$ bit vector (1 = item present, 0 = absent).
  - Use DGIM to estimate count of 1's for each item.

Limitations

- Only approximate.
- Need one bit stream per item $\to$ infeasible for large domains (e.g., e-commerce).

Exponentially Decaying Windows

- Goal: find **currently** most popular items.
- Instead of last N items $\to$ compute **smooth aggregation** over all time.
- If stream is $a_1, a_2, \ldots$, define at time t:

$$\text{Sum}_t = \sum_{i=1}^{t} a_i (1 - c)^{t-i}$$

  where c is small (10^{-6}–10^{-9}).
- **Update rule:** on a_{t+1} arrival, multiply current sum by $(1 - c)$ and add a_{t+1}.

Sliding vs Decaying Windows

- Weight of past items decays geometrically: $(1 - c)^t$.
- Total effective weight:

$$\sum_t (1 - c)^t = \frac{1}{1 - (1 - c)} = \frac{1}{c}$$

- Thus, exponential decay mimics a sliding window of width $\approx 1/c$.

Counting / Scoring Items

- Choose threshold $t < 1$.
- On new item arrival:
  1. Multiply all existing scores by $(1 - c)$.
  2. If item already has a score, add 1; else create score = 1.
  3. Remove items whose score $< t$.
- Since $\sum$ scores $= 1/c$, maintain at most $1/(ct)$ active items.

Lecture 9

Online Algorithms
- **Classic model of algorithms:**
  - You get to see the entire input, then compute some function of it.
  - In this context, called an *offline algorithm.*
- **Online Algorithms:**
  - You see the input one piece at a time, and must make irrevocable decisions along the way.
  - Similar to the *data stream model.*

Online Bipartite Matching
- **Perfect matching:** all vertices of the graph are matched.
- **Maximum matching:** a matching that contains the largest possible number of matches.

Matching Algorithm
- **Problem:** Find a maximum matching for a given bipartite graph.
- There is a polynomial-time offline algorithm based on augmenting paths (Hopcroft–Karp 1973).
- **But what if we do not know the entire graph upfront?**

Online Graph Matching Problem
- Initially, we are given the set of *boys.*
- In each round, one *girl*'s choices are revealed (her edges).
- At that time, we have to decide to:
  1. Pair the girl with a boy.
  2. Or not pair the girl with any boy.
- **Example of application:** assigning tasks to servers.

Greedy Algorithm
- Greedy algorithm for the online graph matching problem:
  - Pair the new girl with *any* eligible boy.
  - If there is none, do not pair the girl.
- **Question:** How good is the algorithm?

Competitive Ratio
- For input I, suppose greedy produces matching M_{greedy}, and an optimal matching is M_{opt}.
- **Competitive ratio:**

$$\min_I \left(\frac{|M_{\text{greedy}}|}{|M_{\text{opt}}|} \right)$$

- This measures greedy's *worst-case performance over all possible inputs.*

Analyzing the Greedy Algorithm (1)
- Consider the case $M_{\text{greedy}} \neq M_{\text{opt}}$.
- Let G = set of girls matched in M_{opt} but not in M_{greedy}.
- Then:

$$|M_{\text{opt}}| = |M_{\text{greedy}}| + |G|$$

- Define B = set of boys linked to girls in G.
  - Boys in B are already matched in M_{greedy}.
  - Otherwise, greedy would have matched them.
- Hence $|M_{\text{greedy}}| \geq |B|$.
- Optimal matching pairs all girls in G with boys in B:

$$(3) \quad |G| \leq |B|$$

- Combining (2) and (3):

$$(4) \quad |G| \leq |B| \leq |M_{\text{greedy}}|$$

- So we have:

$$(1)\ |M_{\text{opt}}| = |M_{\text{greedy}}| + |G|, \quad (4)\ |G| \leq |B| \leq |M_{\text{greedy}}|$$

- Combining (1) and (4):
  - Worst case when $|G| = |B| = |M_{\text{greedy}}|$.
  - Then $|M_{\text{opt}}| \leq 2|M_{\text{greedy}}|$.
- Hence:

$$\frac{|M_{\text{greedy}}|}{|M_{\text{opt}}|} \geq \frac{1}{2}$$

Worst-case Scenario
- Illustrates the situation where the greedy algorithm performs the worst.
- Example matching: $(1, a), (2, b)$.
- Graph shows minimal overlap between optimal and greedy matches.

Performance-based Advertising
- Introduced by **Overture** around 2000.
- **Advertisers bid on search keywords**.
- When someone searches for that keyword, *the highest bidder's ad is shown.*
- **Advertiser is charged only if the ad is clicked on** (CPC — Cost per click).
- Similar model adopted by Google around 2002, called **AdWords**.

AdWords Problem
- A stream of queries arrives at the search engine: $q_1, q_2, \ldots$
- Several advertisers bid on each query.
- When query q_i arrives, the search engine must pick a subset of advertisers to show their ads.
- **Goal:** Maximize search engine's revenue.
- **Simple solution:** Instead of raw bids, use expected revenue = Bid × CTR (Click Through Rate).
- Clearly, we need an **online algorithm!**

Limitations of Simple Algorithm
- Sort advertisers by expected revenue instead of raw bid.
- **Challenges:**
  - CTR of an ad is unknown.
  - Advertisers have limited budgets and bid on multiple queries.

Complications: Budget
- Two complications:
  - Budget
  - CTR of an ad is unknown
- (1) **Budget:** Each advertiser has a limited budget.
- Search engine guarantees that advertiser will not be charged more than their daily budget.

Complications: CTR
- (2) **CTR (Click-Through Rate):** Each ad-query pair has a different likelihood of being clicked.
  - Advertiser 1 bids \$2, click probability = 0.1.
  - Advertiser 2 bids \$1, click probability = 0.5.
- CTR is predicted or measured historically, averaged over time.
- **Hard problem:** Exploration vs. exploitation.
  - **Exploit:** Keep showing an ad with good estimated CTR.
  - **Explore:** Show new ads to estimate CTR more accurately.
- CTR is position-dependent (e.g., Ad #1 clicked more than Ad #2).

AdWords Problem (Formal Definition)
- **Given:**
  1. A set of bids by advertisers for search queries.
  2. A click-through rate (CTR) for each advertiser-query pair.
  3. A budget for each advertiser.
  4. A limit on the number of ads displayed per query.
- **Goal:** Respond to each search query with a set of advertisers such that:
  1. Set size $\leq$ limit on ads per query.
  2. Each advertiser has bid on that query.
  3. Each advertiser has enough budget left to pay for a click.

Greedy Algorithm (for AdWords Problem)
- **Simplified environment:**
  - One ad shown per query.
  - All advertisers have same budget B.
  - All ads equally likely to be clicked.
  - Value of each ad is 1.
- **Algorithm:**
  - For a query, pick any advertiser who has bid 1 for that query.
- **Competitive ratio of greedy:** $\frac{1}{2}$.

Bad Scenario for Greedy
- Two advertisers A and B.
  - A bids on query x, B bids on x and y.
  - Both have budgets of \$4.
- **Query stream:** x, x, x, x, y, y, y, y
  - Optimal: A, A, A, A, B, B, B, B
  - Worst-case greedy: $B, B, B, B, -, -, -, -$
- Competitive ratio = 1/2.
- This is the worst case (greedy is deterministic — tie-breaking fixed).

BALANCE Algorithm [MSVV]
- Algorithm by Mehta, Saberi, Vazirani, and Vazirani.
- For each query, pick the advertiser with the **largest unspent budget**.
- Break ties arbitrarily (but deterministically).

Analyzing BALANCE (BAL)
- Advertisers A_1 and A_2 have equal budgets B each.
- OPT assigns every query to an advertiser. Otherwise, removing unassigned queries does not affect revenue(OPT) and only reduces revenue(BAL).
- Simple case: OPT exhausts the budget of both advertisers.
- We show that in this setting, revenue(BAL) $\geq \frac{3}{4} \times$ revenue(OPT)
- Observation: BAL must exhaust at least one advertiser's budget.
- Otherwise, there will be unassigned queries while both advertisers still have non-zero budgets — contradiction.
- Without loss of generality, suppose BAL exhausts the budget of A_2.
- Let x be the amount of budget unused by BAL. The corresponding queries must only be bid by A_2.
- Let $y = B - x$, so the revenue of BAL is $y + B$.
- To show that revenue(BAL) $\geq \frac{3}{4} \times$ revenue(OPT), it suffices to show that $y \geq x$.
- Queries allocated to A_1 and A_2 in the optimal solution.
- Optimal revenue $= 2B$.
- Balance revenue $= 2B - x = B + y$.
- The x queries unassigned by BAL must only be linked to A_2.
- We claim $y \geq x$.
- As a consequence:
  - Balance revenue is minimum for $x = y = B/2$.
  - Minimum Balance revenue $= 3B/2$.
  - Competitive ratio $\geq 3/4$.
- Consider the B queries that OPT assigned to A_1.
- BAL must fully assign these B queries.
- Case 1: At least $B/2$ of these were assigned to A_1.

$$\Rightarrow y \geq B/2.$$

- Case 2: More than $B/2$ of these were assigned to A_2.
  - Let q be the last of these queries assigned to A_2.
  - Just before assigning q, the unspent budget of A_1 must be at most that of A_2, i.e., $\leq B/2$.
  - Hence $x \leq B/2$, and we are done.

BALANCE: General Result
- In the general case, the worst competitive ratio of BALANCE is:

$$1 - \frac{1}{e} \approx 0.63$$

- No online algorithm achieves a better competitive ratio.
- The following example illustrates this worst case.

Worst Case for BALANCE
- N advertisers: $A_1, A_2, \ldots, A_N$, each with budget $B > N$.
- $N \cdot B$ queries appear in N rounds of B queries each.
- Bidding pattern:
  - Round 1: bidders $A_1, A_2, \ldots, A_N$.
  - Round 2: bidders $A_2, A_3, \ldots, A_N$.
  - Round i: bidders $A_i, \ldots, A_N$.
- Optimal allocation: In round i, allocate all B queries to A_i.
- Optimum revenue $= N \cdot B$.

BALANCE Allocation

- Round i: Queries shared among $N - (i-1)$ advertisers.
- After k rounds, the sum of allocations to each of $A_k, \ldots, A_N$ is:

$$S_k = S_{k+1} = \cdots = S_N = \sum_{i=1}^{k-1} \frac{B}{N-(i-1)}.$$

- Find the smallest k such that $S_k \geq B$.
- After k rounds, no more queries can be allocated to any advertiser.

BALANCE: Analysis

- Fact: $H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln(n)$ for large n (Euler's result).
- $S_k = 1$ implies:

$$H_{N-k} = \ln(N) - 1 = \ln\left(\frac{N}{e}\right)$$

- Also, $H_{N-k} = \ln(N-k)$, which means $N - k = \frac{N}{e}$.
- Hence:

$$k = N\left(1 - \frac{1}{e}\right)$$

BALANCE: Final Result

- After the first $k = N(1-1/e)$ rounds, no more queries can be allocated.
- Revenue $= B \cdot N(1-1/e)$.
- Competitive ratio $= 1 - 1/e$.

Example: Web Advertising

- Google's goal: maximize revenue.
- Traditional way: Pay by impression (CPM)
  - Best strategy: go with the highest bidder.
  - But this ignores the effectiveness of an ad.
- Modern way: Pay per click (CPC)
  - Best strategy: go with expected revenue.
  - Expected revenue of ad a for query q:

$$E[\text{revenue}_{a,q}] = P(\text{click}_a \mid q) \times \text{amount}_{a,q}$$

  - $P(\text{click}_a \mid q)$: probability that a user clicks on ad a given query q (unknown).
  - $\text{amount}_{a,q}$: bid amount for ad a on query q (known).

k-Armed Bandit

- Each arm a:
  - Wins (reward = 1) with fixed, unknown probability μ_a.
  - Loses (reward = 0) with probability $1 - \mu_a$.
- All draws are independent given $\mu_1, \mu_2, \ldots, \mu_k$.
- Goal: maximize total reward by deciding which arms to pull.

Mapping to Web Advertising

- Each query is a bandit.
- Each ad is an arm.
- We want to estimate each arm's probability of winning μ_a (the ad's click-through rate).
- Each time we pull an arm, we perform an experiment and gain information.

Stochastic k-Armed Bandit Setting

- Set of k choices (arms).
- Each arm a has an unknown distribution P_a supported on $[0,1]$.
- Game lasts T rounds.
- In each round t: Pick an arm a and obtain a random sample X_t from P_a.
- Rewards are independent of previous draws.
- Goal: maximize $\sum_{t=1}^{T} X_t$.
- We do not know μ_a, but each pull gives us information about it.

Online Optimization

- Online optimization with limited feedback.
- At each time step, a choice must be made.
- We only receive information about the chosen action.
- Similar to online algorithms: we make sequential decisions with partial information.

Solving the Bandit Problem

- A policy is a rule that, in each iteration, selects which arm to pull.
- Policy ideally depends on history of rewards.
- Objective: minimize a measure called regret.

Performance Metric: Regret

- Let μ_a be the mean of distribution P_a.
- Best arm reward: $\mu^* = \max_a \mu_a$.
- Let a_t be the arm chosen at time t.
- Instantaneous regret: $r_t = \mu^* - \mu_{a_t}$.
- Total regret:

$$R_T = \sum_{t=1}^{T} r_t$$

- Goal: choose a policy that ensures $\frac{R_T}{T} \to 0$ as $T \to \infty$.

Allocation Strategies

- If payoffs were known, pick $\arg\max_a \mu_a$.
- For estimation, pick each of the k arms equally often, $\frac{T}{k}$ times.
- Estimate:

$$\hat{\mu}_a = \frac{k}{T} \sum_{j=1}^{T/k} X_{a,j}$$

- Regret:

$$R_T = \frac{T}{k} \sum_{a=1}^{k} (\mu^* - \hat{\mu}_a)$$

Bandit Algorithm: First Try

- Regret is defined in terms of average reward.
- If we can estimate average reward accurately, we can minimize regret.
- Algorithm: Greedy
  - Take the action with the highest estimated average reward.
  - Example:
    - A_1: reward 1 with probability 0.3 (unknown).
    - A_2: reward 1 with probability 0.7 (unknown).
    - Suppose A_1 is played once and gives reward 1.
    - Suppose A_2 is played once and gives reward 0.
    - Then A_1 appears better and will always be chosen — never exploring A_2.

Exploration vs Exploitation

- The example shows the decision trade-off:
  - Exploration: gather data about less-known arms.
  - Exploitation: act on current knowledge to maximize reward.
- The Greedy algorithm does not explore enough.
- We need strategies balancing exploration (trying new arms) and exploitation (using best-known arms).

Optimism and Limitations of Greedy

- The Greedy algorithm is overconfident in its estimate of μ_a.
- A single reward of 0 should not imply that the average reward is 0.
- As a result, Greedy can converge to suboptimal solutions.

Algorithm: ε-Greedy

- For $t = 1, \ldots, T$:
  - Set $\varepsilon_t = O(1/t)$ (decays over time).
  - With probability ε_t: explore by picking an arm uniformly at random.
  - With probability $1 - \varepsilon_t$: exploit by picking the arm with the highest empirical mean payoff.
- **Theorem** (Auer et al., 2002): for a suitable ε_t, the total regret satisfies

$$R_T = O(k \log T) \quad \Rightarrow \quad \frac{R_T}{T} = O\left(\frac{k \log T}{T}\right) \to 0.$$

Issues with ε-Greedy

- The algorithm explicitly separates exploration and exploitation — not elegant.
- Exploration makes suboptimal choices because it picks any arm with equal probability.
- Idea: when exploring/exploiting, we should compare arms using additional information.

Comparing Arms

- Suppose experiments give:

Arm 1: 1 0 0 1 1 0 0 1 0 1

Arm 2: 1

Arm 3: 1 1 0 1 1 1 0 1 1 1

- Mean values: Arm 1 = 5/10, Arm 2 = 1, Arm 3 = 8/10.
- Question: which arm to pick next?
- Idea: do not rely only on the mean payoff — consider confidence as well.

Confidence Intervals

- A confidence interval is a range where the true mean μ_a lies with high probability.
- Example: $\mu_a \in [0.2, 0.5]$ with probability 0.95.
- The fewer times an arm is tried, the larger the interval (less accurate estimate).
- The interval shrinks as we gather more data.
- If confidence intervals are known:
  - Instead of picking the arm with the highest mean,
  - Pick the arm with the highest *upper bound* of its confidence interval.
- This approach is called an *optimistic policy*: we assume each arm is as good as possible within current evidence.

Calculating Confidence Bounds

- Fix an arm a.
- Let $Y_{a,1}, \ldots, Y_{a,m}$ be payoffs in m trials, i.i.d. in $[0,1]$.
- Mean payoff: $\mu_a = E[Y_a]$.
- Estimate: $\hat{\mu}_{a,m} = \frac{1}{m} \sum_{l=1}^{m} Y_{a,l}$.
- Find b such that with high probability $|\mu_a - \hat{\mu}_{a,m}| \leq b$.
- Goal: lower bound for $P(|\mu_a - \hat{\mu}_{a,m}| \leq b)$.

Hoeffding's Inequality

- For i.i.d. $X_1, \ldots, X_m \in [0,1]$, let $\mu = E[X]$, $\hat{\mu}_m = \frac{1}{m} \sum_{l=1}^{m} X_l$.
- Hoeffding's inequality:

$$P(|\mu - \hat{\mu}_m| \geq b) \leq 2\exp(-2b^2 m).$$

- Define confidence level $1 - \delta$:

$$2e^{-2b^2 m} = \delta \Rightarrow b \geq \sqrt{\frac{\ln(2/\delta)}{2m}}.$$

- For a given arm a:

$$P(|\mu_a - \hat{\mu}_{a,m}| \geq b) \leq 2\exp(-2b^2 m).$$

- Let $b(a,T) = \sqrt{\frac{2\log T}{m_a}}$ where m_a is number of times arm a was played.
- Then:

$$P(|\mu_a - \hat{\mu}_{a,m}| \geq b) \leq 2T^{-4}.$$

- This probability decays quickly with T.
- If we do not play an arm, its uncertainty b increases — we never permanently rule it out.

UCB1 Algorithm (Upper Confidence Bound Sampling)

- Initialize:
$$\hat{\mu}_1 = \cdots = \hat{\mu}_k = 0, \quad m_1 = \cdots = m_k = 0.$$
- $\hat{\mu}_a$: estimated payoff (mean reward) of arm a.
- m_a: number of times arm a has been pulled so far.
- For $t = 1, \ldots, T$:
  1. For each arm a, compute:
  $$UCB(a) = \hat{\mu}_a + \alpha\sqrt{\frac{2\ln t}{m_a}}$$
  where α controls the trade-off between exploration and exploitation.
  2. Select arm
  $$j = \arg\max_a UCB(a).$$
  3. Pull arm j, observe reward y_t.
  4. Update:
  $$m_j \leftarrow m_j + 1, \quad \hat{\mu}_j \leftarrow \frac{1}{m_j}(y_t + (m_j - 1)\hat{\mu}_j).$$

Discussion of UCB1

- Confidence interval grows with $\ln t$ (total number of actions).
- It shrinks with m_a (number of times arm a was tried).
- Ensures:
  - Each arm is tried infinitely often.
  - Algorithm balances exploration and exploitation.
- Key principle: *Optimism in the face of uncertainty.*
  - The algorithm assumes unexplored arms might yield higher rewards.
  - Encourages trying less-known arms based on upper confidence bounds.

Interpretation

- The exploration term $\sqrt{\frac{2\ln t}{m_a}}$ comes from Hoeffding's inequality:
$$b \geq \sqrt{\frac{\ln(2/\delta)}{2m}}.$$
- Here, $\ln t$ corresponds to confidence level adjustment as t increases.
- Probability of large deviation decays as t grows:
$$P(|\mu - \hat{\mu}_m| \geq b) = \delta.$$

Use Case: Pinterest

- **Problem:** For new pins or ads, there is little initial information about user engagement.
- Question: How likely are users to interact with each pin/ad?
- Each pin is treated as an arm, user engagement (clicks, saves) are the rewards.
- The system must balance:
  - Exploration: showing new pins to collect data.
  - Exploitation: showing high-engagement pins to maximize user satisfaction.
- Prevents getting stuck in local optima (only showing best-known pins).

Bandit Algorithm in Operation (Round t)

1. Algorithm observes a user interacting with a set A of pins/ads.
2. Based on previous data, it selects an arm $a \in A$ and receives reward $r_{t,a}$.
3. Updates its estimates and improves future arm selection using $(a, r_{t,a})$ observations.

Lecture 10

Recommendations: Diversity

- **Redundancy leads to a bad user experience.**
  - Example headlines:
    - Obama Calls for Broad Action on Guns
    - Obama unveils 23 executive actions, calls for assault weapons ban
    - Obama seeks assault weapons ban, background checks on all gun sales
- We are uncertain about the user's information need — we should not put all eggs in one basket.
- **Question:** How do we optimize for diversity directly?

Encode Diversity as Coverage

- **Idea:** Encode diversity as a coverage problem.
- **Example:** Word cloud of news for a single day.
- Want to select articles so that most words are "covered".
- **concepts** are being covered, **documents** are doing the covering

Simple Abstract Model

- Suppose we are given a set of documents D.
  - Each document d covers a set X_d of words/topics/named entities W.
- For a set of documents $A \subseteq D$, define:
$$F(A) = \left|\bigcup_{i \in A} X_i\right|$$
- **Goal:** Maximize $F(A)$ subject to $|A| \leq k$.
$$\max_{|A| \leq k} F(A)$$
- Note: $F(A)$ is a set function $F : \text{Sets} \to \mathbb{N}$.

Maximum Coverage Problem

- Given a universe $W = \{w_1, \ldots, w_n\}$ and sets $X_1, \ldots, X_m \subseteq W$.
- **Goal:** Find k sets X_i that cover the most of W.
- More precisely, find k sets whose union size is the largest.
- **Bad news:** This is a known NP-complete problem.

Simple Greedy Heuristic

- **Greedy Algorithm:**
  - Start with $A_0 = \{\}$.
  - For $i = 1, \ldots, k$:
    - Find set d that maximizes $F(A_{i-1} \cup \{d\})$.
    - Let $A_i = A_{i-1} \cup \{d\}$.
- **Example:**
  - Evaluate $F(\{d_1\}), \ldots, F(\{d_m\})$, pick best d_1.
  - Then evaluate $F(\{d_1, d_2\}), \ldots$, pick best d_2.
  - Continue until k selections.
$$F(A) = \left|\bigcup_{d \in A} X_d\right|$$

When Greedy Heuristic Fails

- **Goal:** Maximize the size of the covered area.
- Consider the case where A is the largest set covering most part of B and C, while B and C are disjoint sets.
- Optimal: pick B and C.
- Greedy: picks A and then C — suboptimal.

Approximation Guarantee

- **Result:** Greedy produces a solution A where
$$F(A) \geq \left(1 - \frac{1}{e}\right) \times OPT$$
- Claim holds for functions $F(\cdot)$ with two properties:
  - **Monotone:** Adding more docs doesn't decrease coverage.
    If $A \subseteq B$, then $F(A) \leq F(B)$ and $F(\{\}) = 0$.
  - **Submodular:** Adding an element to a larger set gives less marginal improvement.

Submodularity: Definition

- A set function $F(\cdot)$ is called **submodular** if for all $A, B \subseteq W$:
$$F(A) + F(B) \geq F(A \cup B) + F(A \cap B)$$
- *Note:* Equality holds if F is the cardinality of finite sets A and B.

Submodularity: Equivalent Definition

- **Diminishing returns characterization**
- A set function $F(\cdot)$ is submodular if for all $A \subseteq B$:
$$F(A \cup \{d\}) - F(A) \geq F(B \cup \{d\}) - F(B)$$
- *Gain of adding d to a small set is larger than to a large set.*

Example: Set Cover

- $F(\cdot)$ is submodular: for $A \subseteq B$,
$$F(A \cup \{d\}) - F(A) \geq F(B \cup \{d\}) - F(B)$$
- **Natural example:**
  - Sets $d_1, \ldots, d_m$
  - $F(A) = \left|\bigcup_{i \in A} d_i\right|$ (size of the covered area)
  - F is clearly monotone
  - **Claim:** $F(A)$ is submodular

Submodularity: Diminishing Returns

- Submodularity is the **discrete analogue of concavity**.
- For all $A \subseteq B$:
$$F(A \cup \{d\}) - F(A) \geq F(B \cup \{d\}) - F(B)$$
- *Adding d to B helps less than adding it to A.*

Submodularity and Concavity

- **Marginal gain:**
$$\Delta_F(d|A) = F(A \cup \{d\}) - F(A)$$
- **Submodular:**
$$F(A \cup \{d\}) - F(A) \geq F(B \cup \{d\}) - F(B), \quad A \subseteq B$$
- **Concavity:**
$$f(a + d) - f(a) \geq f(b + d) - f(b), \quad a \leq b$$

Submodularity: Useful Fact

- If $F_1, \ldots, F_m$ are submodular and $\lambda_1, \ldots, \lambda_m > 0$, then
$$F(A) = \sum_{i=1}^{m} \lambda_i F_i(A)$$
is submodular.
- **Submodularity is closed under non-negative linear combinations!**
- **Useful facts:**
  - Average of submodular functions is submodular:
  $$F(A) = \sum_i P(i) \cdot F_i(A)$$
  - Multicriterion optimization:
  $$F(A) = \sum_i \lambda_i F_i(A)$$

The Set Cover Problem

- **Objective:** Pick k docs that cover most concepts.
- $F(A)$: Number of concepts covered by A.
- **Elements** = concepts, **Sets** = concepts in documents.
- $F(A)$ is submodular and monotone.
- We can use a **greedy algorithm** to optimize F.
- **The good:** Penalizes redundancy; submodular.
- **The bad:** Ignores concept importance; all-or-nothing coverage.

Probabilistic Set Cover

- **Concept Importance:** Each concept c has an importance weight w_c.
- Objective: Pick k docs that cover most concepts, weighted by importance.

Probabilistic Set Cover

- All-or-nothing could be too harsh.
- **Document coverage function:**

$$cover_d(c) = \text{probability that document } d \text{ covers concept } c$$

- Indicates how strongly document d covers concept c.
- **Set coverage function:**

$$cover_A(c) = 1 - \prod_{d \in A} (1 - cover_d(c))$$

- Represents the probability that at least one document in A covers concept c.
- **Objective:**

$$\max_{A:|A| \leq k} F(A) = \sum_c w_c \, cover_A(c)$$

- Each concept c has a weight w_c representing importance.

Optimizing $F(A)$

- Objective:

$$\max_{A:|A| \leq k} F(A) = \sum_c w_c \, cover_A(c)$$

- The objective function is **submodular**.
- Submodularity implies a *diminishing returns* property.
- A greedy algorithm achieves a $\left(1 - \frac{1}{e}\right) \approx 63\%$ approximation.

Summary: Probabilistic Set Cover

- **Goal:** pick k documents that cover the most concepts.
- Each concept c has an importance weight w_c.
- Documents partially cover concepts via $cover_d(c)$.

Lazy Optimization of Submodular Functions

- Submodular optimization often uses greedy selection.
- Greedy selection adds the document with the largest marginal gain at each step.

Submodular Functions

- **Greedy:**

$$\text{Marginal gain: } F(A \cup \{x\}) - F(A)$$

- Add the document with the highest marginal gain.
- **Greedy algorithm is slow:**
  - Each iteration re-evaluates marginal gains of all remaining documents.
  - Requires $|D|$ evaluations in round 1, $|D| - 1$ in round 2, etc.
  - Total runtime: $O(|D| \cdot K)$ for selecting K documents from D.

Speeding up Greedy

- In round i, we have $A_{i-1} = \{d_1, \ldots, d_{i-1}\}$.
- Pick:

$$d_i = \arg\max_{d \in V} [F(A_{i-1} \cup \{d\}) - F(A_{i-1})]$$

- **By submodularity:**

$$F(A_i \cup \{d\}) - F(A_i) \geq F(A_j \cup \{d\}) - F(A_j) \quad \text{for } i < j$$

- Hence, marginal benefits $\Delta_i(d)$ only decrease as i increases.

Lazy Greedy Idea

- Use Δ_i as an upper bound on Δ_j for $j > i$.
- Maintain an ordered list of marginal benefits Δ_i from the previous iteration.
- Re-evaluate Δ_i only for the top element.
- Re-sort and prune:
  - If $\Delta_i(d_2) \leq \Delta_{i+1}(d_1)$, then $\Delta_{i+1}(d_2) \leq \Delta_{i+1}(d_1)$.
- Submodularity guarantees:

$$F(A \cup \{d\}) - F(A) \geq F(B \cup \{d\}) - F(B)$$

  whenever $A \subseteq B$.

Overview of the Algorithm

1. Identify items to recommend from.
2. Identify concepts.
3. Weigh concepts by general importance.
4. Define item-concept coverage function - Probabilistic set coverage function.
5. Select items using probabilistic set cover - Lazy greedy algorithm.
6. Obtain feedback and update weights - Multiplicative weights update rule.

Lazy Greedy Algorithm

- **Initial iteration:**
  - Start with $A_0 = \emptyset$.
  - Compute $\Delta_1(d) = F(A_0 \cup \{d\}) - F(A_0)$ for all d.
  - Let $A_1 = \{\arg\max_d(\Delta_1(d))\}$.
  - Construct a priority queue for remaining $|D| - 1$ documents with $q(d) = \Delta_1(d)$.
- **Iteration i:**
  - Sort remaining $|D| - i + 1$ documents in descending order of $q(d)$.
  - Evaluate marginal gain only for the top document.
  - Let $d^* = \arg\max_d q(d)$.
  - Calculate $\Delta_i(d^*) = F(A_{i-1} \cup \{d^*\}) - F(A_{i-1})$.
  - Update $q(d^*) := \Delta_i(d^*)$.
  - Let d^{**} be the next document in the queue.
    - If $q(d^*) \geq q(d^{**})$, set $A_i = A_{i-1} \cup \{d^*\}$.
    - If $q(d^*) < q(d^{**})$, re-order and re-evaluate $q(d^{**}) := \Delta_i(d^{**})$.

Personal Concept Weights

- Each user has different preferences over concepts.

Personal Concept Weights (Mathematical Formulation)

- Assume each user u has a different preference vector $w_c^{(u)}$ over concepts c.
- The objective becomes:

$$\max_{A:|A| \leq k} F(A) = \sum_c w_c^{(u)} \, cover_A(c)$$

- **Goal:** Learn personal concept weights from user feedback.

Multiplicative Weights (MW)

- **Multiplicative Weights algorithm:**
  - Assume each concept c has weight w_c.
  - Recommend document d and receive feedback $r \in \{+1, -1\}$.
- **Update the weights:**
  - For each $c \in X_d$, set:

$$w_c \leftarrow \beta^r w_c$$

  - If concept c appears in document d and feedback $r = +1$, increase w_c by multiplying with $\beta > 1$.
  - If $r = -1$, decrease w_c (divide by β).
- Normalize weights so that:

$$\sum_c w_c = 1$$