Finding Similar Items

Why? *Given millions of documents, find all similar pairs. Brute-force is $O(n^2)$—infeasible. Solution: a 3-stage pipeline converting the problem from* representation → compression → fast search.

1. Shingling (Representation)

- Convert document to **set of k-grams** (k consecutive tokens). Captures local word order: "dog bites man" ≠ "man bites dog". Using individual words fails because two different articles about same topic share vocabulary.
- k=5 for short docs (emails), k=10 for long docs. Too small k ⇒ every doc shares same common shingles.
- Hash each shingle to integer for compactness. Doc becomes a set of integers instead of strings.

2. MinHash (Compression)

- Goal: compress each set into a short **signature** while preserving Jaccard similarity. Cannot compare full shingle sets—too large.
- Key property: for a random permutation π, the MinHash value $h(S) = \min_{x \in S} \pi(x)$. Then:

$$\Pr[h(A) = h(B)] = \frac{|A \cap B|}{|A \cup B|} = J(A, B)$$

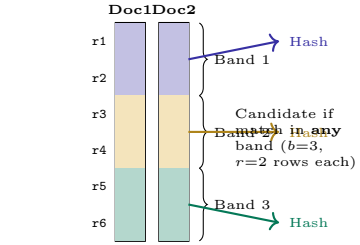
- **Intuition**: Permute all rows randomly. MinHash = index of first row with a 1. For two columns to agree, the first non-$(0,0)$ row must be $(1,1)$. Probability of this = $\frac{|\text{intersection}|}{|\text{union}|}$ = Jaccard.
- Use n hash functions ⇒ signature of length n. **Fraction of matching entries ≈ Jaccard similarity** (unbiased estimator). More hash functions = better estimate.
- **Practical implementation**: never actually permute rows (too expensive). Instead simulate with hash functions $h(x)=(ax+b) \bmod p$. Initialize all signature values to ∞. Scan each row; for each 1-entry, compute all hash values and keep minimum per column.

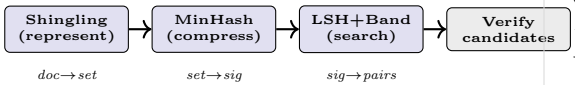
3. LSH + Banding (Efficient Search)

- Even with short signatures, comparing all $O(n^2)$ pairs is too slow for millions of documents.
- **Banding trick**: divide each signature into b bands of r rows. Hash each band of each column into a bucket. Two documents are a **candidate pair** if they match on **all** r rows in **at least one** band.

$$P(\texttt{candidate}) = 1 - (1 - s^r)^b$$

- Creates an **S-curve**. Below a threshold similarity, probability of becoming candidates ≈ 0. Above threshold, ≈ 1. Threshold ≈ $(1/b)^{1/r}$.
- ↑ b (more bands, fewer rows each) ⇒ ↓ threshold: easier to match in at least one band ⇒ more candidates, more FP, fewer FN.
- ↑ r (fewer bands, more rows each) ⇒ ↑ threshold: harder to match all rows ⇒ fewer candidates, fewer FP, more FN.
- Total hash functions $n = b \times r$ is fixed budget. Choosing b determines r—you're choosing the **FP/FN tradeoff**.
- LSH has **both FP and FN**: FP because low-similarity pairs can hash together by luck ($s = 0.2$ still has 20% chance per hash); FN because high-similarity pairs can differ in all bands by bad luck.
- After LSH: **verify** candidate pairs with actual similarity computation to eliminate FP.





Full Pipeline & Key Formulas

- Shingling (representation) → MinHash (compression) → LSH+Banding (fast search) → Verify candidates.
- **Jaccard similarity**: $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$. Range $[0,1]$. J=1 iff identical sets. J=0 iff disjoint.
- Applications: plagiarism detection (rearranged text), near-duplicate web pages, content-based image search.

Frequent Itemsets & Association Rules

Why? *Discover items that frequently co-occur in transactions (market-basket). Applications: shelf management, cross-selling (diapers → beer), recommendations. Core challenge: exponential # of possible itemsets (2^d for d items), memory limitations for pair counting.*

Definitions

- **Support**(I) = # baskets containing *all* items in I.
- **Frequent itemset**: support ≥ threshold s.
- **Association rule** $I \to j$: if a basket contains I, how likely does it also contain j?

$$\texttt{Confidence}(I \to j) = \frac{\texttt{support}(I \cup \{j\})}{\texttt{support}(I)}$$

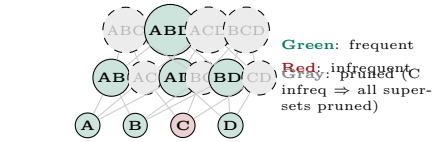
- Confidence ∈ $[0,1]$. Max confidence = 1 means j *always* co-occurs with I. support$(I \cup \{j\})$ ≤ support(I) always (numerator ≤ denominator).
- **Not causation**—only co-occurrence. Diapers don't *cause* beer purchases.
- # possible rules from k-itemset: $\binom{k}{1} + \cdots + \binom{k}{k-1} = 2^k - 2$ (every non-empty proper subset as LHS).

Two-Step Mining Approach

- **Step 1**: Find all frequent itemsets (the hard, computationally expensive part). This is where Apriori/PCY/SON come in.
- **Step 2**: Generate rules from frequent itemsets (easy—just divide known support values and check ≥ minconf). No extra data scanning needed—all supports already computed.

Anti-Monotone Property of Support (Bottom-Up)

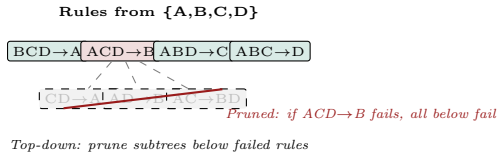
- If itemset S is **infrequent**, every **superset** of S is also infrequent. (A basket containing $\{A,B,C\}$ must contain $\{A,B\}$, but not vice versa. So support can only decrease as set grows.)
- Enables **bottom-up pruning**: if $\{A\}$ is infrequent, skip *all* pairs/triples/etc. containing A. Prunes entire branches of the lattice.



Bottom-up: prune supersets of infrequent items

Anti-Monotone Property of Confidence (Top-Down)

- For rules derived from the **same frequent itemset**: confidence **decreases** as LHS shrinks. Why? All rules share the same numerator (support of the full itemset), but smaller LHS has larger support (denominator) ⇒ smaller fraction.
- conf$(ABC \to D)$ ≥ conf$(AB \to CD)$ ≥ conf$(A \to BCD)$
- Enables **top-down pruning**: if $ABC \to D$ fails minconf, *all* rules with D on RHS and any subset of $\{A,B,C\}$ on LHS will also fail. Prune entire subtree downward.
- **Key contrast**: Itemset discovery = **bottom-**



Naive Pair Counting & Memory Problem

- Naive: scan baskets, count every pair. With 100K items ⇒ ~5B pairs × 4 bytes = 20GB. Doesn't fit in 2GB RAM.
- **Triangular matrix**: pre-allocate slot for *every* possible pair $\{i,j\}$ where $i < j$. $n(n-1)/2$ entries, 4 bytes each. $O(1)$ lookup via index formula. Wastes space if most pairs never occur.
- **Triples**: store only observed pairs as (i,j,count). Each entry 12 bytes. Saves space when sparse. Breakeven at ∼ 1/3 density—above that, triangular wins.
- Both fail when total pairs exceed memory. Need smarter approach.

Apriori Algorithm

- **Bottom-up, level-by-level**:
  - Pass 1: scan all baskets, count individual items (1D array, very cheap). Keep frequent items as L_1. **Discard counts**—only keep the set L_1.
  - Pass 2: scan all baskets again. For each basket, generate only pairs where **both items** $\in L_1$. Count these candidates. Keep frequent pairs as L_2.
  - Pass k: generate candidates C_k from L_{k-1} (only if **all** $(k-1)$-subsets are in L_{k-1}).

One full pass to count. Keep as L_k.
  - Stop when $L_k = \emptyset$.
- Key insight: Apriori **solves the memory problem** not by storing pairs more efficiently, but by **drastically reducing the number of pairs** to count. If 100K items but only 1K frequent ⇒ only ∼500K candidate pairs instead of 5B.
- One pass per level. Finding pairs is the bottleneck (most numerous).

PCY (Park-Chen-Yu) Algorithm

- Key idea: **use idle memory during Pass 1** to build a cheap approximate filter for pairs.
- **Pass 1**: while counting individual items, *also* hash every pair from each basket into a bucket. Increment bucket count by 1. Buckets are **shared**—multiple pairs map to same bucket. Stores bucket counts (cheap: 1 counter per bucket) not individual pair counts.
- After Pass 1: convert bucket counts to **bitmap** (1 bit per bucket). Bucket $\geq s \Rightarrow$ bit=1, else bit=0. Bitmap is 1/32 the size of the hash table (1 bit vs 4 bytes).
- **Pass 2**: pair (i, j) is candidate only if: (1) both i, j are frequent, **AND** (2) (i, j) hashes to bucket with bit=1. **Two filters** instead of one.
- **What PCY eliminates**: pairs of frequent items that *don't actually co-occur frequently.* In Apriori, if both A and B are frequent, $\{A, B\}$ is always a candidate even if they never appear together. PCY catches this if their bucket is infrequent.
- **Both algorithms do 2 full passes**. PCY does more work in Pass 1 (hashing pairs) but saves memory in Pass 2 (fewer candidates need individual counters). Hashing is computationally cheap; memory is the bottleneck.
- If frequent pair exists, its bucket is *definitely* frequent (no FN). But infrequent pairs might land in frequent buckets due to collisions (FP). Even imperfect filtering helps.

Random Sampling

- Run Apriori on random sample (fraction p) with threshold $p \cdot s$. Single pass over sample.
- **False positives**: frequent in sample, not in full data. Eliminated by verification pass.
- **False negatives**: frequent in full data, missed in sample. *Cannot* be recovered—never identified as candidate. Mitigate by lowering threshold further.

SON Algorithm

- **Exact answer, no false negatives**. Divide data into memory-sized chunks.
- For each chunk (fraction p): run full algorithm with threshold $p \cdot s$. Find locally frequent itemsets.
- Candidate = frequent in ≥ 1 chunk. Pass 2: count all candidates across full data.
- **Correctness**: if globally frequent (support $> s$), total support $> s$, so support in at least one chunk must exceed $p \cdot s$ (by pigeonhole). Hence found as candidate. No FN.
- Parallelizable: chunks processed independently (like MapReduce).

Clustering

Why? *Group similar data points* without labels *(unsupervised). Goal: ↓ intra-cluster distance, ↑ inter-cluster distance. Different algorithms make different tradeoffs: scalability vs. shape flexibility vs. parameter sensitivity vs. cluster assumptions.*

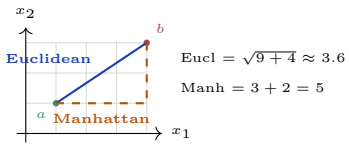
Distance Measures

- The choice of distance function **fundamentally shapes** what clusters look like. Different data types need different distances.

L_p-Norm (Minkowski Distance):

$$d_p(a, b) = \left(\sum_{i=1}^{d} |a_i - b_i|^p\right)^{1/p}$$

- p=1: **Manhattan distance** (city-block). Sum of absolute differences per dimension. Moves along axes only—like navigating a grid.
- p=2: **Euclidean distance** (straight-line). Most common default. Sensitive to large deviations in any single dimension (due to squaring).
- $p \to \infty$: **Chebyshev distance**. Max absolute difference across all dimensions: $d_\infty(a, b) = \max_i |a_i - b_i|$.
- Key difference: Manhattan treats all per-dimension deviations equally (additive). Euclidean penalizes large deviations more (squared then rooted). E.g. $(3, 0)$ vs $(0, 0)$: Manhattan = 3, Euclidean = 3. But $(2, 2)$ vs $(0, 0)$: Manhattan = 4, Euclidean = $2\sqrt{2} \approx 2.83$.



Other Important Distances:

- **Cosine similarity**: $\cos(a, b) = \frac{a \cdot b}{\|a\| \|b\|}$. Measures angle between vectors, ignores magnitude. Range $[-1, 1]$; cosine distance $= 1 - \cos(a, b)$. Good for text (TF-IDF vectors) where document length shouldn't matter.
- **Jaccard distance**: $d_J(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$. For sets/binary vectors. Used in k-FreqItem, MinHash/LSH.
- **Mahalanobis distance**: $d(x, c) = \sqrt{\sum_i \left(\frac{x_i - c_i}{\sigma_i}\right)^2}$. Normalizes each dimension by cluster's std dev. Used in BFR to handle elongated clusters (see BFR section).

Normalization / Scaling:

- Without normalization, features with large ranges (salary in thousands) dominate features with small ranges (age in decades). **Must normalize** before using Euclidean/Manhattan.
- Mean absolute deviation: $z_i[j] = \frac{x_i[j] - m_j}{s_j}$ where m_j is mean, s_j is mean absolute deviation of dimension j. Puts all dimensions on comparable scales.
- **Decision trees are scale-invariant** (only care about order). **k-NN and k-Means are not**—must normalize.

Two Clustering Philosophies

- **Hierarchical (Agglomerative)**: start with every point as its own cluster, repeatedly merge closest pair. Builds a **tree** (dendrogram) of nested clusterings at all granularities. Don't need k upfront.
- **Point Assignment (Partitioning)**: e.g. k-means. Start with k centers, assign each point to nearest, update centers, repeat. Gives a single **flat partition**—just k groups, no nesting, no internal hierarchy.

Cluster Representation

- **Centroid** (Euclidean spaces): the **mean** of all points' coordinates. An artificial computed point that may not correspond to any real data point. Natural for Euclidean data.
- **Clustroid** (non-Euclidean spaces): the actual data point that is most "central"—minimizes $\sum d(x, c)^2$ over all points x in the cluster. Needed when you **can't compute an average** (e.g. sets with Jaccard, strings with edit distance—what's the "mean" of two sets?).

Inter-Cluster Distance (Linkage)

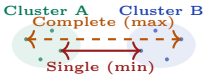
How do we decide which two clusters to merge? Several options, each giving different clustering behavior:

- **Single-link** (minimum): distance between the two **closest** points, one from each cluster. Tends to create **long chain-like** clusters—a single nearby point can bridge two groups. Sensitive to noise.
- **Complete-link** (maximum): distance between the two **farthest** points. Produces **compact, round** clusters. Conservative—won't merge unless even the farthest pair is close.
- **Average-link**: average of all pairwise distances between points in the two clusters. **Compromise** between single and complete.
- **Centroid/Clustroid distance**: distance between representative points. Simple and fast.
- **Ward's method**: merge the pair that minimizes the increase in total within-cluster variance. Often produces the most balanced clusters.

Cohesion-based merging: instead of distance between clusters, ask "how cohesive is the merged result?"

- **Diameter**: max distance between any two points in merged cluster. Lower = more cohesive.
- **Average pairwise distance**: average of all distances in merged cluster.
- **Density**: diameter (or avg distance) divided by # points. **Lower ratio = higher density = better cluster** (many points packed into small space). We want to merge the pair

whose union has **lowest** density ratio.



Hierarchical Clustering (Agglomerative)

- **Bottom-up**: start with each point as its own cluster, repeatedly merge closest pair. Produces **dendrogram**—cut at desired level for k clusters.
- Don't need k upfront—choose after seeing dendrogram. Cut high → few big clusters. Cut low → many small clusters.
- Stopping criteria: pre-set k, or stop when next merge exceeds a quality threshold (diameter, density).
- Complexity: naive $O(n^3)$, with priority queue $O(n^2 \log n)$—**doesn't scale** to large datasets. Motivates k-means, BFR, etc.

K-Means

- Iterative: (1) assign each point to **nearest centroid**, (2) recompute centroids as **mean** of assigned points, (3) repeat until stable (no point changes cluster).
- Must specify k upfront. **Elbow method**: plot avg distance vs k, find "knee" where improvement rate drops sharply.
- Smart init: pick first centroid randomly, then each subsequent as far as possible from existing centroids (avoids all starting in same region).
- Each iteration **guaranteed to reduce** total squared distance ⇒ convergence guaranteed (but to local minimum, not global). Usually converges in few iterations.
- Assumes **convex/spherical** clusters. Sensitive to initialization and outliers. Complexity: $O(n \cdot k \cdot d)$ per iteration.

BFR (Bradley-Fayyad-Reina)

- **Scales k-means to data too large for memory**. Processes data in disk-sized chunks.
- **Key assumption**: clusters are Gaussian (axis-aligned ellipses). Strong assumption but enables very compact representation.
- **Sufficient statistics** per cluster (only $2d+1$ numbers):
  - **N**: number of points.
  - **SUM**: d-dim vector, sum of all coordinates.
  - **SUMSQ**: d-dim vector, sum of squares.
- From these: centroid = SUM/N, variance = SUMSQ/N − (SUM/N)^2. Can **discard actual points** and keep only stats.
- **Three point sets**:
  - **DS** (Discard Set): points close to a centroid, summarized into stats, *discarded*.
  - **CS** (Compression Set): groups of nearby points not yet assigned to any cluster, summarized with same stats. "Proto-clusters."
  - **RS** (Retained Set): isolated points that don't fit anywhere, kept as actual points.
- **Mahalanobis distance** (used to decide "close enough"):

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

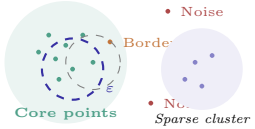
- Normalizes deviation in each dimension by cluster's std dev in that dimension. 10 units off in a tight dimension (σ=1) is much worse than 10 units off in a spread dimension (σ=100). Mahalanobis distance of 2–3 is typical threshold.
- **Processing each chunk** (step by step):
  - For each new point: if close enough to an existing DS centroid (Mahalanobis < threshold) → add to that **DS** (just update N/SUM/SUMSQ, discard point).
  - Remaining new points + old RS points → cluster together using any in-memory algorithm → groups become new **CS**, isolated points become new **RS**.
  - Consider merging **CS with CS**: if two CS clusters are close (combined variance below threshold), merge them. *CS never merges with DS during intermediate chunks*—only CS-to-CS.
  - **Last chunk only**: force all remaining CS into their nearest DS. Force all RS into nearest DS. Everything must be assigned by the end.

CURE (Clustering Using Representatives)

- Handles **arbitrary cluster shapes** (crescents, S-shapes, interleaving spirals)—unlike k-means/BFR which assume convex/elliptical.
- Instead of single centroid, represent cluster by **multiple dispersed points** spread around its boundary. Then **shrink** these representatives toward the centroid by a fraction (e.g. 20%).
- Shrinkage serves dual purpose: makes representatives robust to outliers and prevents clusters from being dominated by extreme boundary points.
- Pass 1: take random sample that fits in memory → cluster hierarchically → pick well-dispersed representatives → shrink them. Pass 2: scan entire dataset, assign each point to nearest representative's cluster.
- Memory-efficient in Pass 2: only representative points needed in memory, not entire dataset.

DBSCAN (Density-Based)

- Fundamentally different approach: **cluster = dense region of points separated by sparse regions**. Not centroid-based at all.
- Two parameters: ε (radius) and MinPts (minimum neighbors).
  - **Core point**: has ≥ MinPts neighbors within ε.
  - **Border point**: within ε of a core point but not itself core.
  - **Noise point**: neither core nor border.
- Clusters grow by connecting core points that are within ε of each other (**density-reachable**).
- **Strengths**: finds clusters of **any shape** (not just convex), naturally identifies **noise/outliers**, **doesn't need** k.
- **Limitations**: (1) Single global ε fails for **varying-density clusters**—too small fragments sparse cluster, too large merges dense cluster with noise. (2) Border point assignment is **non-deterministic**: a border point reachable from two clusters goes to whichever is processed first. Core point assignments are always deterministic.
- Tiny change in ε can produce completely different clustering.



Anomaly Detection

Why? *Find data points significantly different from the majority. Critical for fraud detection, equipment failure prediction, network intrusion, AI system monitoring. Context matters: freezing temp is anomalous in Singapore but normal in Antarctica. Anomalies can be important signals (equipment failure) or noise (data entry error).*

1. Statistical Methods

- Fit a distribution (e.g. Gaussian), flag points with low probability (e.g. $> 3\sigma$ from mean).
- Rigorous when assumptions hold. Fails when data doesn't match assumed distribution.

2. Distance-Based (k-NN Distance)

- Anomaly score = distance to k-th nearest neighbor. Large distance ⇒ likely anomaly.
- Uses **single global threshold**. **Problem with varying density**: in a sparse cluster, even *normal* points have large k-NN distances (neighbors are far). Near a dense cluster, a *true outlier* might have small k-NN distance (dense cluster is close by). Global threshold misranks: flags normal sparse-cluster points over true outliers near dense clusters.

3. Density-Based (LOF)

- **Local Outlier Factor**: compares each point's **local reachability density** to the average density of its k neighbors.
- LOF $\gg 1$ ⇒ point is in a sparser region than its neighbors ⇒ anomaly. LOF ≈ 1 ⇒ consistent with local neighborhood ⇒ normal.
- **Fixes varying-density problem**: comparison is always *local*. A point in a sparse cluster has sparse neighbors too, so LOF ≈ 1. A true outlier between clusters has denser neighbors than itself, so LOF $\gg 1$.
- Drawback: $O(n^2)$ distance computations.

4. Clustering-Based

- Run clustering (e.g. DBSCAN), flag noise points as anomalies. Or use k-means and flag points far from centroids.
- Can use **relative distance** (point's distance / cluster's average distance) to handle clusters of different sizes.
- Simple, leverages existing algorithms. But: (1) sensitive to algorithm/parameter choice, (2) how many clusters? (3) outliers can distort the clustering itself (circular problem with k-means).

5. Reconstruction-Based (Autoencoders/PCA)

- Assumption: normal data has **structure** capturable in lower dimensions. Anomalies don't fit this structure.
- Compress data to lower dims (PCA or autoencoder's bottleneck), then reconstruct.
- **Normal** data: reconstructs well (low error). **Anomalies**: reconstruct poorly (high error—compression was optimized for normal patterns).

```
Anomaly score = ||x − x̂||    (reconstruction error)
```

- Flexible, works with deep learning. But struggles in very high dimensions.

Summary of Tradeoffs

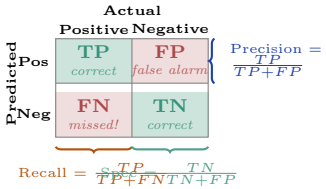
- Statistical: rigorous but assumes distribution. Distance: intuitive but global threshold. LOF: handles varying density but $O(n^2)$. Clustering: leverages existing tools but circular dependency. Reconstruction: flexible but high-dim issues.
- No single method universally best—choose based on data characteristics.

Classification & Regression I: Evaluation

Why? *Given features X, predict label y (classification) or continuous value (regression). Must evaluate models properly—wrong metrics give misleading results, especially on imbalanced data. Must split data correctly to prevent information leakage.*

Confusion Matrix

- 2×2 table: rows = predicted, columns = actual. Four cells:
- **TP** (predicted +, actually +), **TN** (predicted −, actually −), **FP** (predicted +, actually −, "false alarm"), **FN** (predicted −, actually +, "missed").
- Mnemonic: 2nd word = what model *predicted* (Positive/Negative). 1st word = whether prediction was *correct* (True/False).

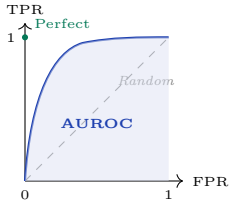


Core Metrics

- **Accuracy** $= \frac{TP+TN}{\text{all}}$. **Misleading on imbalanced data!** With 1000 healthy, 20 sick: "always predict healthy" gets 1000/1020 = 98% accuracy, 0% recall. **Never use accuracy alone on imbalanced data.**
- **Precision** $= \frac{TP}{TP+FP}$: "Of all I predicted positive, how many are truly positive?" Matters when **false alarms are costly** (spam filter sending real email to spam).
- **Recall** (= **Sensitivity**) $= \frac{TP}{TP+FN}$: "Of all actual positives, how many did I catch?" Matters when **missing positives is costly** (disease screening).
- **Specificity** $= \frac{TN}{TN+FP}$: accuracy among actual negatives. "How good at correctly clearing healthy people?"
- **F1** $= \frac{2 \cdot P \cdot R}{P+R}$: **harmonic mean** of precision and recall. Always closer to whichever is lower—can't game by maximizing one. P=1, R=0.1 ⇒ F1 = 0.18 (not 0.55).
- **Precision-Recall tradeoff**: lowering classification threshold ⇒ more predicted positive ⇒ ↑ recall, ↓ precision. Raising threshold ⇒ opposite.
- Key distinction: Sensitivity/Specificity condition on **ground truth** (actual class). Precision conditions on **prediction** (what model said).

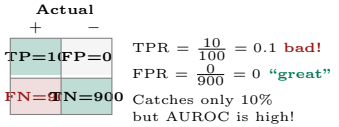
ROC / AUROC / AUPRC

- Many classifiers output **scores** (probabilities), not hard labels. Need threshold to convert. Different thresholds ⇒ different TP/FP tradeoffs.
- **ROC curve**: plot TPR (y-axis) vs FPR (x-axis) at *all* thresholds. Threshold = ∞: point $(0,0)$. Threshold = $-\infty$: point $(1,1)$. Good classifier bows toward **top-left**.
- **AUROC**: area under ROC curve. 1=perfect, 0.5=random (diagonal). Probabilistic interpretation: P(model scores random positive > random negative). Threshold-independent.
- **Problem with AUROC on imbalanced data**: massive TN count makes FPR look great even when recall is terrible. AUROC can be high while model misses 90% of positives.
- **AUPRC**: Precision (y) vs Recall (x) curve. **Neither axis uses TN**, so not inflated by class imbalance. Preferred metric for imbalanced datasets. Perfect = top-right corner.



Why AUROC misleads on imbalanced data:

Example: 900 negatives, 100 positives.



$\text{FPR} = \frac{FP}{FP+\underbrace{TN}_{\text{huge!}}} \Rightarrow$ `always near 0 when TN dominates`

ROC curve hugs left axis (low FPR) regardless of how many positives are missed. Even adding FP barely moves FPR when TN=900. **AUPRC fixes this**: neither Precision nor Recall uses TN, so not inflated by imbalance.

Multi-class Evaluation

- With C classes, confusion matrix is $C \times C$. Decompose into C binary problems using **one-vs-rest**.
- **Micro-averaging**: sum TP/FP/FN across *all* classes first, then compute P/R/F1. Effectively **favors larger classes** (they contribute more to the sum).
- **Macro-averaging**: compute P/R/F1 *per class*, then average. Treats all classes **equally regardless of size**. Better if you care about rare classes.

Data Splitting & Cross-Validation

- **Train / Validation / Test** split (e.g. 60/20/20). Training set trains model. Validation set tunes hyperparameters / selects model. Test set is **only used once at the very end** for final evaluation. Peeking at test set during development ⇒ optimistic bias.
- **K-fold Cross-Validation**: split data into k folds. In round i, fold i is validation, rest is training. Train a **completely new model from scratch each round**—no memory between rounds. Average k validation scores ⇒ robust performance estimate.
- Purpose of CV: answer "how well will this *type* of algorithm perform?" not "produce one final model." After CV selects hyperparameters, retrain on all training data.
- **Information leakage**: if you normalize data (subtract mean, divide by std) **before** splitting, test set's values influenced the normalizer. Correct: fit normalizer on training data only, apply same transform to test.

k-Nearest Neighbors (k-NN)

Why? *Simplest classifier: no training, no model parameters. Just memorize data and look at what's nearby. Non-parametric—zero assumptions about data distribution. But expensive at prediction time and degrades in high dimensions.*

Algorithm

- For test point x: compute distance to **every** training point. Find k nearest neighbors. **Majority vote** among neighbors' labels (classification) or **average** of neighbors' values (regression).
- "Lazy learner": no training phase at all. All computation happens at prediction time: $O(n \cdot d)$ per query.

Choosing k

- k=1: decision boundary is very noisy—every training point creates its own pocket. Overfits.
- Large k: smoother boundary, more robust to noise. But may underfit—too much averaging dilutes signal. Extreme: k=n always predicts overall majority class.
- Use cross-validation to find optimal k.

Key Properties

- **Sensitive to feature scaling**: if income is in thousands and age is in decades, income dominates distance. **Must** normalize/standardize features before using k-NN.
- **Curse of dimensionality**: in high-d space, all pairwise distances converge—"nearest" neighbor is barely closer than the farthest. k-NN loses discriminative power.
- Non-parametric: decision boundary can take any shape (adapts to data). Can capture complex nonlinear patterns.
- Unlike decision trees, **not** scale-invariant. Unlike neural nets, no training overhead.

Decision Trees

Why? *Highly interpretable—a flowchart of if/else rules that a human can follow and explain. Handles mixed data types natively.* ***Scale-invariant****: only cares about value ordering, not magnitude. But single trees are* ***unstable****—small data change can produce completely different tree (high variance).*

Training (Greedy Top-Down Induction)

- Start at root with all data. At each node: try **all features and all possible split points**. Pick the split that **maximally reduces impurity** in children (weighted by child sizes).
- Recursively split until stopping condition (pure node, max depth, min samples).
- Greedy: locally optimal split at each step, not globally optimal tree structure.

Impurity Measures

$\texttt{Gini} = 1 - \sum_i p_i^2$ $\qquad$ $\texttt{Entropy} = -\sum_i p_i \log_2 p_i$

- Both equal 0 when node is pure (all one class). Maximized when classes equally distributed.
- **Gini index**: probability that two randomly chosen samples from the node have different class labels.

- **Entropy**: measures information content / uncertainty.
- **Information Gain**: IG = Entropy(parent) $-$ $\sum \frac{|child|}{|parent|}$ · Entropy(*child*). Pick feature/split maximizing IG.
- In practice, Gini and Entropy produce very similar trees.

Pruning (Prevent Overfitting)

- Unpruned tree perfectly fits training data ⇒ memorization ⇒ poor generalization.
- **Pre-pruning** (early stopping): set max depth, min samples per leaf, min impurity decrease. Simple but may stop too early.
- **Post-pruning**: grow full tree first, then remove subtrees whose removal improves *validation* performance. More principled.
- **Cost-complexity pruning**: $\text{Cost}(T) = \text{Error}(T) + \alpha \cdot |\text{leaves}|$. Higher α ⇒ simpler tree. Tune α via cross-validation.

Key Properties

- **Scale-invariant**: any monotone transformation (USD→SGD, standardization, log transform) produces the *exact same tree*. Only the ordering of values matters, not magnitudes.
- Handles both categorical and numerical features natively (no encoding needed).
- Interpretable: can draw and explain the decision path.
- But **unstable**: removing a few data points can completely change the tree ⇒ high variance. This motivates ensemble methods.

Ensemble Methods

Why? *Single trees: unstable, high variance. "Wisdom of crowds": if 1000 people guess gumballs in a jar, individual guesses are wild but the* ***average is surprisingly accurate****. Combining many models cancels individual errors. Two strategies:* ***Bagging*** *(parallel, ↓ variance) and* ***Boosting*** *(sequential, ↓ bias).*

Bagging (Bootstrap Aggregating)

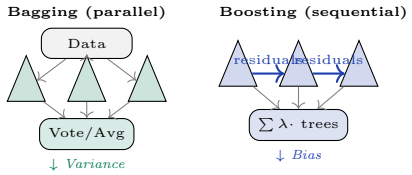
- Create N **bootstrap samples**: sample **with replacement** from original data, same size. Some points duplicated, some omitted. Each sample ≈ 63.2% unique points.
- Train one tree per bootstrap sample. Aggregate predictions: **majority vote** (classification) or **average** (regression).
- Trees trained **independently** ⇒ embarrassingly parallel.
- Each tree sees slightly different data ⇒ makes slightly different errors ⇒ errors **cancel when aggregated** ⇒ reduces variance.

Random Forest = Bagging + Feature Sampling

- Key addition over bagging: at each split, consider only a **random subset of features** ($\sqrt{p}$ for classification, $p/3$ for regression). Not all features.
- Why? Without this, if one feature is very dominant, *all* bagged trees will use it first ⇒ trees are correlated ⇒ averaging doesn't help much (like groupthink—independent opinions asked first, aggregated later, vs. everyone discussing then voting).
- Feature sampling forces trees to use **different features**, making them **decorrelated** ⇒ averaging is much more effective.
- **Variable importance**: for each feature, measure total Gini decrease across all splits in all trees. Features with large total decrease are more important. Useful for feature selection.
- Hyperparameters: # trees (more = better, diminishing returns), max features/split, max depth.
- ↓ overfitting: more trees, fewer features per split. ↑ overfitting: deeper individual trees, more features per split.
- **Pros**: near state-of-art accuracy, robust, parallelizable, minimal tuning needed. **Con**: loses single-tree interpretability (can't draw one tree and show client).

Gradient Boosted Decision Trees (GBDT)

- Fundamentally different from bagging: **sequential, not parallel**. Each new tree specifically targets what previous trees got wrong.
- **Algorithm**:
  - Step 1: start with initial prediction (e.g. mean of y). Current residuals $r = y - \hat{y}$.
  - Step 2: fit a (shallow) decision tree to predict r—the residuals, not the original targets.
  - Step 3: update predictions: $\hat{y}_{\text{new}} = \hat{y}_{\text{old}} + \lambda \cdot \text{tree}(r)$. Learning rate λ (e.g. 0.1) **shrinks** each tree's contribution.
  - Step 4: recompute residuals, repeat.
- **Learning rate** λ: small λ means each tree contributes a small correction ⇒ needs more trees but generalizes better (less overfitting). Like golf: don't swing full force every shot—take many gentle corrections.
- **Bagging vs Boosting summary**:
  - **Bagging**: parallel, independent trees, reduces **variance** (averaging). Works because trees see different data.
  - **Boosting**: sequential, dependent trees, reduces **bias** (error correction). Works because each tree fixes predecessors' mistakes.



Logistic Regression

Model

- $P(Y{=}1|X) = \sigma(Xw + b) = \frac{1}{1+e^{-(Xw+b)}}$
- Sigmoid function squashes any real number to $(0, 1)$ ⇒ interpretable as probability.
- Decision boundary: the hyperplane where $Xw + b = 0$. **Linear** in feature space.
- Loss function: **binary cross-entropy** (log loss). Trained iteratively via gradient descent—compute gradient of loss w.r.t. parameters, update in opposite direction.

Gradient Descent Variants

- **Full-batch GD**: use **all** n data points to compute gradient, make 1 update. Most accurate gradient estimate per step, but only 1 parameter update per epoch (slow).
- **Mini-batch GD**: use subset (e.g. 32–256 samples) per update. Multiple updates per epoch. Good balance of accuracy and speed.
- **SGD** (Stochastic): use **1 sample** per update. n updates per epoch. Noisiest gradient but fastest exploration of loss landscape.
- Key terminology: 1 **iteration/step** = 1 gradient computation + 1 parameter update. 1 **epoch** = 1 complete pass through all data. For full-batch: 1 iter = 1 epoch. For mini-batch of size B: 1 epoch = n/B iters. For SGD: 1 epoch = n iters.

Neural Networks / Deep Learning

Why? *Logistic regression = single linear boundary. But many real problems need nonlinear boundaries. Stack multiple layers of "logistic-regression-like" units with nonlinear activations ⇒ can approximate any continuous function (****universal approximation theorem****).*

- **Architecture**: input layer → hidden layers → output layer. Each neuron: $z = Xw + b$, $a = g(z)$ where g is activation function. Hidden layers learn increasingly abstract features.
- **Activation functions** (introduce nonlinearity):
  - **Sigmoid**: maps to $(0, 1)$. Problem: **vanishing gradient** in deep nets (extreme inputs ⇒ gradient ≈ 0).
  - **Tanh**: maps to $(-1, 1)$, centered at 0. Same vanishing gradient issue.
  - **ReLU** $= \max(0, z)$: default choice in modern deep learning. Simple, fast, largely solves vanishing gradient. "Dead neuron" issue: if $z < 0$ always, gradient is always 0.
  - **Parametric ReLU**: small slope for $z < 0$ instead of flat 0. Prevents dead neurons.
- **Why nonlinearity is essential**: without activation functions, n stacked linear layers = single linear transformation (matrix multiplication is associative). No gain in expressiveness. Nonlinear activations break this, enabling deep nets to learn complex patterns.
- **Training**: **backpropagation** computes gradients layer by layer using chain rule + gradient descent updates all weights simultaneously.
- **Parameter counting**: fully connected layer from m inputs to n outputs: $m \times n$ weights + n biases. Example: $3 \to 2 \to 1$ network: weights = $3{\times}2 + 2{\times}1 = 8$, biases = $2 + 1 = 3$, total = 11. Modern networks: millions/billions of parameters.
- **Tricks to speed up learning**: Xavier/He weight initialization, **Adam optimizer** (adaptive learning rates per parameter), **batch normalization** (normalize layer inputs).
- **Tricks to reduce overfitting**: **Dropout** (randomly zero out neurons during training—forces redundancy), L1/L2 regularization, **early stopping** (stop when

validation loss starts increasing), **data augmentation** (rotations, flips, etc. to create more training data).

Clustering II: k-FreqItem & SILK

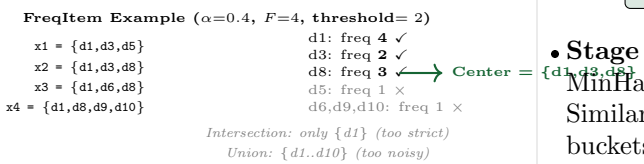
Why? *Standard k-means fails on* ***ultra-high-dimensional sparse binary data*** *(millions of dimensions, but each point has only a few non-zero entries). Mean is meaningless for binary vectors (0.3 in a binary dim?). Euclidean distance uninformative. Embedding loses information. k-FreqItem clusters directly using Jaccard distance + frequent item computation. Combines MinHash, LSH, frequent itemsets, and clustering into a single* ***"meta analytics"*** *pipeline.*

Why K-Means Fails on Sparse Binary Data

- K-means centroid = mean of points. For binary vectors, mean produces dense real-valued vectors (e.g. $[0.3, 0.7, 0.1, \ldots]$). This is **not binary**, so you can't use Jaccard distance between a point and its "center." Representation mismatch.
- Euclidean distance treats a 0-0 match the same as a 1-1 match. For ultra-sparse data (most entries are 0), almost all pairs have similar Euclidean distance ⇒ no discriminative power.
- Embedding (SVD, etc.) to lower dims loses the interpretability of which features define each cluster.

FreqItem Center Computation

- Instead of mean, the "center" is the set of **most frequent items** across cluster members.
- For cluster with N points, count how often each dimension has value 1. Keep dimensions with frequency $\geq \max(1, \lceil \alpha \cdot F \rceil)$, where F = max frequency, $\alpha \in (0, 1]$ is a parameter.
- **Not intersection** (α=1, too strict—one missing feature drops it). **Not union** (α=0, too loose—includes noise). FreqItem is the **middle ground**.



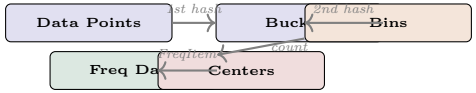
k-FreqItem Algorithm

- Same iterative template as k-means:
  - (1) Assign each point to center with **max Jaccard similarity** (min Jaccard distance).
  - (2) Update each center via **FreqItem computation** from assigned points.
  - (3) Repeat until stable.
- Distance: $J(x, c) = \frac{|x \cap c|}{|x \cup c|}$; objective = maximize total similarity.
- Center is **interpretable**: a set of features characterizing the cluster (e.g. {machine, learning, neural, network} for a topic cluster).

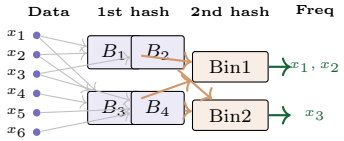
Seeding Variants

- **k-FreqItem (basic)**: random initial centers. Fast but often bad.
- **k-FreqItem++**: like k-means++. Pick first randomly, then each next proportional to distance from nearest existing center. Spreads seeds apart. But **requires** k **sequential passes**—slow when k is large (e.g. 10,000).
- **SILK**: uses two-level MinHash to find seeds in **one parallel pass**. No sequential dependency.

SILK Seeding Pipeline (4 Stages)



- **Stage 1 — Data → Buckets** (1st-level MinHash): hash data points into buckets. Similar points land in same bucket. But buckets are **impure** (FP: dissimilar points can hash together). Can't use buckets directly as seeds.
- **Stage 2 — Buckets → Bins** (2nd-level MinHash): now treat *buckets* as objects (represented by their member sets). Hash similar buckets into "bins." Bins group buckets that share many data points.



- **Stage 3 — Bins → Frequent Data**: within each bin, count how often each data point appears across the bin's buckets. Points appearing in **many** buckets within the same bin are the "core" of a cluster. Threshold: $\max(1, \lceil \beta \cdot F' \rceil)$ where F' = max frequency in the bin, β is a parameter.
- Example: Bin1 has buckets $\{B_1, B_3\}$. x_1 appears in both (freq 2), x_4 in only B_3 (freq 1). With β=0.8, threshold = 2, so x_1, x_2 are frequent data.
- **Stage 4 — Frequent Data → Centers**: apply **FreqItem computation** on the frequent data points from each bin to produce an initial center. This overseeds ($|C| > k$ centers). Then assign all n points to nearest center, compute weights (cluster sizes), and run weighted k-FreqItem++ on just the $|C|$ centers to reduce to k. Cheap since $|C| \ll n$.

Why Two Levels of Hashing?

- 1st-level buckets are **impure** due to LSH false positives: dissimilar points can hash together. Can't use buckets directly as cluster seeds.
- 2nd-level groups *similar buckets* together. Points that appear in many similar buckets (across different hash functions) are **genuinely close**—not just lucky collisions. This filters out FP noise.
- The frequent data determination further refines: only the most consistently appearing points survive.

Generic Applicability & Unary Encoding

- Any data decomposable into sub-units works: strings → n-grams, graphs → subgraphs, time series → sliding windows, DNA → subsequences, social networks → adjacency rows (community detection).
- **Unary encoding for multisets**: feature appearing n times (max m) → n ones + $(m-n)$ zeros. E.g. count 3, max 4 → "1110". Preserves ordering: higher count ⇒ more overlap under Jaccard.
- Entire pipeline is **embarrassingly parallel**. Hashing stages are independent. Sync points only at stage boundaries with lightweight data exchange.

Quick Reference: Algorithm Comparison

gray!15 Algorithm	Problem It Solves	Key Idea	Strengths	Limitations
MinHash+LSH	Find similar items in massive collections	Shingling → MinHash (compress) → LSH banding (search)	Sublinear search; probabilistic Jaccard guarantees	Approximate; FP/FN tradeoff via b, r; tuning needed
Apriori	Find frequent itemsets (bottom-up, level-wise)	Prune candidates if any subset is infrequent; one pass per level	Principled pruning; correct; conceptually simple	Memory-intensive for pairs; one full data pass per level
PCY	Reduce Apriori's candidate pairs in Pass 2	Hash pairs to shared buckets in Pass 1; bitmap filter eliminates pairs in infrequent buckets	Extra filter; uses idle memory in Pass 1	Hash collisions reduce effectiveness; still 2 passes
SON	Exact frequent itemsets on data too large for memory	Chunk data; local frequent in any chunk → global candidate; verify	Exact (no FN); parallelizable (MapReduce)	Two full passes over entire data
K-Means	Partition into k spherical clusters	Iterative assign-to-nearest-centroid + recompute mean	Simple, fast, scales well	Need k; convex only; sensitive to init/outliers
BFR	K-means on disk-resident data	Sufficient stats (N,SUM,SUMSQ) + Mahalanobis distance	Processes in chunks; $O(2d{+}1)$ memory per cluster	Assumes Gaussian/axis-aligned elliptical clusters
CURE	Arbitrary-shape clusters at scale	Multiple representative points per cluster, shrunk toward centroid	Captures non-convex shapes; robust to outliers via shrinkage	Two-pass; representative selection is heuristic
DBSCAN	Density-connected clusters of any shape	Core/border/noise classification via ε + MinPts	Arbitrary shapes; handles noise naturally; no k needed	Single ε fails for varying density; non-deterministic borders
LOF	Detect anomalies in varying-density data	Compare point's local density to its neighbors' densities	Handles varying density via local comparison	$O(n^2)$ distance computations
Decision Tree	Interpretable classification/regression	Greedy top-down splits maximizing Gini/entropy reduction	Interpretable; scale-invariant; handles mixed types	High variance; single tree overfits easily
Random Forest	Stable, accurate ensemble classification	Bagging + random feature subsets at each split (decorrelation)	Low variance; near SOTA; parallelizable; minimal tuning	Loses single-tree interpretability
GBDT	Highest accuracy via sequential error correction	Each tree predicts residuals of ensemble; learning rate controls step	Reduces bias; often best on tabular data	Sequential (not parallelizable); can overfit without tuning
k-FreqItem	Cluster ultra-high-dim sparse binary data	FreqItem centers + Jaccard distance; SILK seeding via MinHash	No embedding needed; parallelizable; generic via set decomposition	Jaccard-specific; needs sparse binary representation

Key tradeoffs: Interpretability vs Accuracy vs Scalability. Bagging = parallel, reduces variance. Boosting = sequential, reduces bias. Frequent itemsets: bottom-up support pruning (Apriori) + top-down confidence pruning (rule generation). Both exploit anti-monotonicity in opposite directions.