Graph Search and Ranking Cheatsheet

$\pi(u) = \frac{\deg(u)}{2|E|}$ — Stationary distribution for undirected graphs, probability of being at node u after infinite steps

$R(i) = \frac{1}{\pi(i)}$ — Return time, expected steps to return to node i after leaving it

$H(i, j)$ — Hitting time, expected steps to reach j starting from i

$H(i, k) = k^2 - i^2$ — Hitting time for chain graph

$H(k-1, k) = 2k - 1$ — One step on chain takes $2k - 1$ expected steps due to oscillation

$\mathrm{CT}(i, j) = H(i, j) + H(j, i)$ — Commute time, round trip expected time

$v(x) = \frac{v(x-1)+v(x+1)}{2}$ — Harmonic function for electrical network, voltage at x is average of neighbors

$\forall i : \sum_{j \in N(i)} g_{ji}(v(j) - v(i)) = 0$ — Kirchhoff's current law, no charge accumulation at node i, g_{ji} is conductance

$v(i) = \frac{\sum_{j \in N(i)} g_{ji} v(j)}{\sum_{j \in N(i)} g_{ji}}$ — Voltage as conductance-weighted average of neighbors

Minimize $\sum_{(i,j) \in E} g_{ij}(v(i) - v(j))^2$ — Laplacian minimization for harmonic function

$M(i, j) = i$ links to j — Adjacency matrix, 1 if edge exists

$C(j, i) = \frac{M(i,j)}{d(i)}$ — Transition probability matrix, $d(i)$ is degree of i

$p_t = C^\top p_{t-1}$ — Probability distribution evolves by transpose of C

$p_1 = p_0 \iff p_0$ is stationary — Stationary distribution definition

$p = \alpha C p + (1 - \alpha) r$ — PageRank equation with teleport

$p = (1 - \alpha)(I - \alpha C)^{-1} r$ — Closed form PageRank solution

$p = (1 - \alpha) \sum_{k \geq 0} \alpha^k C^k r$ — Series expansion of PageRank

$x^{(k)} = A x^{(k-1)}$ — Power iteration for eigenvector computation

$x^{(k)} = c_1 u_1 + \lambda_2^k c_2 u_2 + \lambda_3^k c_3 u_3 + \cdots \to c_1 u_1$ — Convergence to dominant eigenvector

$x^{(k)} \leftarrow x^{(k)} - \frac{(x^{(k)} - x^{(k-1)})^2}{x^{(k)} - 2x^{(k-1)} + x^{(k-2)}}$ — Aitken acceleration for faster convergence

$\tilde{C} = \alpha C + (1 - \alpha) \frac{\mathbf{1}_{|V| \times |V|}}{|V|}$ — PageRank with uniform teleport to all nodes

$$\hat{C} = \begin{bmatrix} \alpha C_{|V| \times |V|} & (1 - \alpha)\mathbf{1}_{|V| \times 1}/|V| \\ \mathbf{1}_{1 \times |V|} & 0 \end{bmatrix}$$

— Sparse graph representation with dummy node d

$p_r = \alpha C p_r + (1 - \alpha) r$ — Personalized PageRank with custom teleport r

$p_{ar} = a p_r$ — Linearity in teleport distribution

$p_{r_1 + r_2} = p_{r_1} + p_{r_2}$ — Additivity in teleport distribution

$\mathrm{PPV}_u = p_{\delta_u}$ — Personalized PageRank Vector for node u with impulse teleport

$p_r = \sum_u r(u) \mathrm{PPV}_u$ — Arbitrary teleport as linear combination of impulse responses

$p_r = p_{\alpha C r} + (1 - \alpha) r$ — Pushdown property, αC pushed into subscript

$\mathrm{PPV}_u = \alpha \sum_{(u,v) \in E} C(v, u) \mathrm{PPV}_v + (1 - \alpha)\delta_u$ — Hub decomposition, PPV from out-neighbors

$\|p - \tilde{p}\|_1 \leq 2 \sum_{u \in P} \frac{p_u}{1 - \alpha}$ — PageRank score stability bound for perturbed nodes P

$Q(u, v) = \frac{A(u,v)}{D(u,u)}$ — Markovian transition probability, $D(u, u)$ is row sum

$$L = I - \frac{\Pi^{1/2} Q \Pi^{-1/2} + \Pi^{-1/2} Q^\top \Pi^{1/2}}{2}$$

— Directed graph Laplacian

$$x^\top L x = \sum_{(u,v)\in E} \pi(u) Q(u,v) \left(\frac{x_u}{\sqrt{\pi(u)}} - \frac{x_v}{\sqrt{\pi(v)}} \right)^2$$

— Smoothness penalty for directed graphs

$a(v) = \sum_{u \to v} h(u)$ — Authority score from hub in-neighbors
$h(u) = \sum_{u \to v} a(v)$ — Hub score from authority out-neighbors
$a = E^\top h$ — Authority vector from hub vector
$h = Ea$ — Hub vector from authority vector
$h = EE^\top h$ — Hub eigenvector equation
$a = E^\top E a$ — Authority eigenvector equation
$E = U \Sigma V^\top$ — Singular Value Decomposition of edge matrix
$EE^\top = U \Sigma^2 U^\top$ — Hub-hub similarity matrix
$E^\top E = V \Sigma^2 V^\top$ — Authority-authority similarity matrix
$\|S - \tilde{S}\|_2 = 2\delta$ where $\delta = \lambda_1 - \lambda_2$ — HITS instability, small perturbation flips eigenvalues
$\|a - \tilde{a}\|_2 = \sqrt{2}$ — HITS authority vectors can be perpendicular after perturbation
$p(v \to w) = \sum_{u:(u,v),(u,w)\in E} \frac{1}{\mathrm{InDeg}(v)} \frac{1}{\mathrm{OutDeg}(u)}$ — SALSA transition probability
$\pi(v) \propto \mathrm{InDeg}(v)$ — SALSA stationary distribution proportional to in-degree
$a = (1-\beta) E^\top h + \beta r_{\text{auth}}$ — PHITS authority with teleport
$h = (1-\beta) Ea + \beta r_{\text{hub}}$ — PHITS hub with teleport
$\mathrm{sim}(u,u) = 1$ — SimRank self-similarity

$$\mathrm{sim}(u,v) = \frac{\beta}{|I(u)||I(v)|} \sum_{a \in I(u), b \in I(v)} \mathrm{sim}(a,b)$$

— SimRank recursive definition

$$s(a,b) = \sum_t \Pr(\text{path } t : (a,b) \to (x,x))\, \beta^{\text{length}(t)}$$

— Expected f-meeting distance

$r(q,v) = \mathrm{PPV}_q(v)$ — Individual proximity from query node q to v
$r(Q,v;|Q|) = \prod_{q\in Q} r(q,v)$ — AND query
$r(Q,v;1) = 1 - \prod_{q\in Q}(1 - r(q,v))$ — OR query

$$r(Q,v;k) = r(Q \setminus \{q_1\}, v; k-1)\, r(q_1,v) + r(Q \setminus \{q_1\}, v; k)(1 - r(q_1,v))$$

— K-out-of-n meeting probability

$g(H) = \sum_{v \in V(H)} r(Q,v;k)$ — Centerpiece subgraph goodness
$L_G = NN^\top = D - A$ — Undirected graph Laplacian

$$N(v,e) = \begin{cases} -\sqrt{A(e)} & e = (v,\cdot) \\ \sqrt{A(e)} & e = (\cdot,v) \\ 0 & \text{otherwise} \end{cases}$$

— Node-edge incidence for weighted graphs
$x^\top Lx = \sum_{(u,v)\in E} A(u,v)(x_u - x_v)^2$ — Laplacian quadratic form penalizes rough scores

$$\min_x \; x^\top Lx + B \sum_{u \prec v} \max\{0, 1 + x_u - x_v\}$$

— Ranking optimization with Laplacian smoothing
$x_u \propto \sqrt{\pi(u)}$ — Optimal score without training data is proportional to $\sqrt{\text{PageRank}}$
$\text{loss}(y) = \max\{0, y\}$ — Hinge loss

$$\text{huber}(y) = \begin{cases} 0 & y \le 0 \\ \frac{y^2}{2W} & 0 < y \le W \\ y - W/2 & y > W \end{cases}$$

— Huber loss smooth approximation
$q_{uv} = \pi(u)Q(u,v)$ — Reference circulation from PageRank
$p_{uv} \ge 0,\ \sum_{u,v} p_{uv} = 1$ — Valid circulation constraints
$\sum_u p_{uv} = \sum_w p_{vw}$ — Flow balance at node v

$$\min - \sum_{u,v} p_{uv} \log p_{uv}$$

— Maximum entropy flow objective

$$\min \sum_{u,v} p_{uv} \log \frac{p_{uv}}{q_{uv}}$$

— Minimum KL divergence from reference flow
$p_{uv} \propto q_{uv} e^{\beta_v - \beta_u}$ — Unconstrained maximum entropy flow solution

$$p_{dv} \propto q_{dv}\, e^{\beta_v - \beta_d + \text{bias}(v)}$$

— Flow with teleports and preferences

$$p_{vd} \propto q_{vd}\, e^{\beta_d - \beta_v + \alpha\tau_v}$$

— Outgoing teleport flow

$$p_{uv} \propto q_{uv} e^{\beta_v - \beta_u - (1-\alpha)\tau_u + \text{bias}(v)}$$

— Regular edge flow with bias

$$\text{bias}(v) = \sum_{r \prec v} \pi_{rv} - \sum_{v \prec s} \pi_{vs}$$

— Bias from preference constraints

$$1 + \sum_{(w,u) \in \hat{E}} p_{wu} \leq \sum_{(w,v) \in \hat{E}} p_{wv} + s_{uv}$$

— Margin constraint with slack

$$\min \text{KL}(p \| q) + C \sum_{u \prec v} s_{uv} + C_1 F^2 \quad \text{s.t.} \ \sum p_{uv} = F$$

— Objective with scaled margin

$$C(j,i) = \begin{cases} \alpha \frac{\beta(t(i,j))}{\sum_k \beta(t(i,k))} & i \neq d, j \neq d \\ 1 - \alpha & i \neq d, j = d \\ r_j & i = d, j \neq d \end{cases}$$

— Typed edge conductance

$$\frac{\partial C(i,j)}{\partial \beta(\tau)}$$

depends on whether $\tau = t(i,j)$ — Gradient for typed conductance learning

$$p \approx C^H p_0$$

— Finite horizon approximation for PageRank

$$\frac{\partial}{\partial \beta(t)} (C^h p_0)_i = \sum_j \Big(\frac{\partial C(i,j)}{\partial \beta(t)} (C^{h-1} p_0)_j + C(i,j) \, \frac{\partial}{\partial \beta(t)} (C^{h-1} p_0)_j \Big)$$

— Gradient via chain rule

$\text{score}(x,y) = |\Gamma(x) \cap \Gamma(y)|$ — Common neighbors
$\text{score}(x,y) = |\Gamma(x)| \, |\Gamma(y)|$ — Preferential attachment
$\text{score}(x,y) = \frac{|\Gamma(x) \cap \Gamma(y)|}{|\Gamma(x) \cup \Gamma(y)|}$ — Jaccard coefficient

$$\text{score}(x,y) = \sum_{z \in \Gamma(x) \cap \Gamma(y)} \frac{1}{\log |\Gamma(z)|}$$

— Adamic–Adar

$$\text{score}(x,y) = \sum_{\ell \geq 1} \beta^\ell \, \text{paths}_\ell(x,y)$$

— Katz score

PageRank models random surfer with probability α following links and probability $1-\alpha$ teleporting, ensures irreducibility and aperiodicity for unique stationary distribution

Personalized PageRank allows query-specific teleport distribution enabling entity search and recommendations with sparse teleport via hub decomposition reducing $O(|V|^2)$ to manageable size

HITS computes hub and authority scores via power iteration on $EE^\top$ and $E^\top E$ but suffers from topology sensitivity and lacks stabilizing teleport parameter unlike PageRank

SALSA makes HITS probabilistic via bipartite random walk yielding stationary distribution proportional to in-degree more stable than original HITS against clique attacks

Graph Laplacian approach penalizes score roughness $x^\top Lx$ assuming neighbors should have similar scores works for undirected and directed graphs via modified Laplacian

Network circulation optimizes flows $\{p_{uv}\}$ directly with flow balance and KL divergence from reference yielding more constrained hypothesis space and better generalization than Laplacian

Typed edge conductance learns weights β_t for edge types via gradient descent on finite-horizon PageRank approximation useful for multi-relational graphs

Link prediction uses common neighbors, Jaccard, Adamic–Adar, Katz, PageRank variants; Katz and Adamic–Adar often best, combining topology with attributes improves results

Key stability results: PageRank score-stable with ℓ_1 bound but rank-unstable under adversarial perturbations; HITS extremely unstable; eigengap determines sensitivity; teleport $1-\alpha$ controls PageRank stability

Hub decomposition theorem $\mathrm{PPV}_u = \alpha \sum_{(u,v)} C(v,u)\mathrm{PPV}_v + (1-\alpha)\delta_u$ enables precomputing hub PPVs and reconstructing arbitrary PPV via backward propagation

Asynchronous message passing processes nodes by largest residual, stops at precomputed hubs, enabling anytime algorithm with bounded error for web-scale personalized search

Topic-sensitive PageRank computes multiple PageRanks per topic, teleports within topic pages, query blended PageRanks gives personalized results

SimRank measures structural similarity recursively, requires $O(|V|^2)$, interpreted as meeting probability of paired random walkers

Centerpiece subgraphs use meeting probabilities of multiple random walkers to find subgraphs containing query nodes; used in social networks and image captioning

DOM tree segmentation prevents topic drift in HITS by separating sidebar links from content links

Electrical network analogy relates random walk probabilities to voltages; hitting time corresponds to voltage difference; effective resistance algorithms apply

Return time $R(i) = 1/\pi(i)$ explains high-degree nodes have short return times

Chain graph hitting time $H(i,k) = k^2 - i^2$ grows quadratically; adjacent nodes $H(k-1,k) = 2k-1$ shows oscillation effects

Power iteration convergence rate set by λ_2; teleport ensures $\lambda_2 \leq \alpha$; Aitken acceleration speeds convergence

Cover time $\Theta(n^2)$ for chain graph; depends on diameter and mixing time

Commute time relates to effective resistance; $CT(i,j) = 2|E|R_{\text{eff}}(i,j)$

Maximum entropy flow and minimum-KL flow yield exponential family solutions

Preference constraints encoded as linear inequalities; dual variables give exponential form

Margin scaling F and regularization C_1F^2 balance fit vs complexity

Typed conductance nonconvex but trainable; chain-rule gradient with power iteration

Sampling in graphs: forward walk, inverse PageRank, bidirectional regularized walk

Page staleness detection via recursive definition with escape probability

PHITS stabilizes HITS via teleport but too much teleport collapses to uniform

Laplacian limitations: assumes smoothness everywhere; L^+ dense; generalization bound scale with $L^+(u, u)$

Circulation advantages: respects Markov balance, sparse, reduced variance

Typed conductance: scaling invariance requires $\beta \geq 1$; finite horizon H

Link prediction challenges: sparsity, class imbalance, temporal mismatch

Best predictors: Katz, Adamic–Adar, common neighbors, Jaccard; hitting time poor baseline

Enhancing link prediction with attributes and community structure

Experiments: PageRank robust to random deletion; HITS unstable; adversarial examples possible

DOM segmentation improves authority scoring by filtering irrelevant hubs

$\alpha \approx 0.85$ balances locality vs stability

Hub selection based on high PageRank; precompute PPVs to amortize cost

Bidirectional walk with self-loops for uniform sampling

Clique attacks break HITS; SALSA mitigates

Eigengap δ determines HITS sensitivity

Series expansion interpretation of PageRank as weighted paths

Sparse teleport algorithm identifies active nodes and hubs

Flow balance generalizes Kirchhoff's law

Dummy node d implements teleport in sparse graph

Dual variables interpret in statistical mechanics

Huber loss ensures differentiability

Model cost regularizes β_t

Convergence criteria for power iteration and message passing

Graph diameter influences cover and mixing time

Harmonic functions minimize Laplacian

Effective resistance metric aids clustering

Personalized importance fuses global and local structure

Irreducibility and aperiodicity guaranteed by teleport

Row-stochastic vs column-stochastic conventions

Eigenvector centrality family

Preferential attachment as baseline

Jaccard normalizes by union

Adamic–Adar discounts hubs

Katz sums exponentially decayed paths

Hitting time asymmetry makes it poor proximity measure
Score vs rank stability distinction
Coupled walks technique for stability bounds
Adversarial perturbations cause $\Theta(n^2)$ ranking swaps
Topic drift in HITS solved with DOM segmentation
Maximum entropy as principled uniformity
KL divergence non-negative and convex
Dual variables yield exponential-form flows
Primal–dual optimization for flows
Feature engineering for link prediction
ML classifiers for link prediction
Evaluation metrics AUC, MAP, Precision@k
Cold start problem for new nodes
Temporal dynamics and concept drift
Multi-relational graphs and typed edges
Community structure affects link prediction
Homophily important for edges
Scalability: sparse matrices, distributed computing
Index–query tradeoff
Reproducibility: seeds, splits, error bars
Interpretability: feature importance, saliency
Fairness: demographic parity, equalized odds
Robustness: adversarial attacks and defenses
Theoretical guarantees: convergence, bounds
Practical guidelines: start simple, validate, iterate
Open problems: dynamic graphs, attributed graphs, GNNs, robustness

Graph Search and Learning — Complete Cheat Sheet

1 Three Principles of Node Relatedness

1. **Multiple Paths**: More paths between nodes → stronger relatedness
2. **Shorter Paths**: Shorter paths contribute more than longer paths
3. **Degree Penalty**: Paths through high-degree nodes should matter less

Why shortest path fails: ignores multiple paths and alternative routes
Why max flow fails: doesn't sufficiently penalize long paths

2 Random Walk Fundamentals

2.1 State Probability Vector $p(t)$

$p(t) = [p_1(t), p_2(t), \ldots, p_n(t)]$ where $p_i(t)$ = probability of being at node i at time t

2.2 Transition Matrix P

$P[i, j]$ = probability of moving from node i to node j
For undirected unweighted graphs:

$$P[i,j] = \begin{cases} 1/\text{degree}(i), & \text{if edge } (i,j) \text{ exists} \\ 0, & \text{otherwise} \end{cases}$$

2.3 One-Step Update Rule

$$p(t+1) = P^T p(t)$$

2.4 Steady State Distribution π

$$\pi = P^T \pi$$

For undirected graphs:

$$\pi_i = \frac{\text{degree}(i)}{2m}$$

2.5 Expected Return Time

$$E[\text{Return\_Time}_i] = \frac{1}{\pi_i}$$

3 Hitting Time and Cover Time

3.1 Hitting Time

$H(i, j)$ = expected number of steps to reach node j from i
Asymmetric: $H(i, j) \neq H(j, i)$

3.2 Commute Time

$$CT(i, j) = H(i, j) + H(j, i)$$

3.3 Cover Time

Expected number of steps to visit all nodes at least once.

4 Absorbing Random Walks

4.1 Node Types

Transient nodes T Absorbing nodes A

4.2 Transition Matrix Structure

Block matrix:

$$\begin{bmatrix} Q & R \\ 0 & I \end{bmatrix}$$

4.3 Escape Probability Equation

$$x = Qx + R\mathbf{1}$$

4.4 Solution

$$x = (I - Q)^{-1}R\mathbf{1}$$

Fundamental matrix:

$$N = (I - Q)^{-1}$$

5 Graph Matrices

5.1 Degree Matrix D

$D[i, i] = \text{degree}(i)$

5.2 Adjacency Matrix A

$A[i, j] =$ edge weight

5.3 Graph Laplacian L

$$L = D - A$$

5.4 Normalized Laplacian

$$L_{\text{norm}} = D^{-1/2} L D^{-1/2}$$

6 Harmonic Functions and Voltage

6.1 Harmonic Property

$$h(i) = \sum_j \frac{c_{ij}}{d_i} h(j)$$

6.2 Voltage Equation

$$Lv = b$$

7 Electrical Network Analogy

7.1 Conductance

$c_{ij} = 1/r_{ij}$

7.2 Series and Parallel Rules

Series:

$$R_{\text{total}} = r_1 + r_2 + \cdots$$

Parallel:

$$C_{\text{total}} = c_1 + c_2 + \cdots$$

7.3 Effective Conductance

$$CT(s, t) = \frac{2m}{C_{\text{eff}}(s, t)}$$

8 PageRank

8.1 Recurrence

$$PR(i) = \frac{1-\alpha}{N} + \alpha \sum_{j \to i} \frac{PR(j)}{\text{out\_degree}(j)}$$

8.2 Matrix Form

$$\pi = (1-\alpha)d + \alpha P^T \pi$$

8.3 Homogeneous Form

$$\pi = M\pi$$

where $M = (1-\alpha)d\mathbf{1}^T + \alpha P^T$

8.4 Power Iteration

$$\pi_{k+1} = (1-\alpha)d + \alpha P^T \pi_k$$

9 Personalized PageRank (PPR)

9.1 Teleport Vector

Impulse teleport:

$$d = [0, \ldots, 1, \ldots, 0]$$

9.2 Linearity

$$\alpha d_1 + \beta d_2 \to \alpha \pi_1 + \beta \pi_2$$

9.3 Backward Propagation

$$\text{credit}[j] += \alpha \frac{\text{credit}[i]}{\text{in\_degree}(i)}$$

10 SimRank

$$\text{SimRank}(a,b) = \gamma \cdot \frac{1}{|I(a)|\,|I(b)|} \sum_{i \in I(a)} \sum_{j \in I(b)} \text{SimRank}(i,j)$$

Base case:

$$\text{SimRank}(a,a) = 1$$

11 HITS

Authority update:

$$a(i) = \sum_{j \to i} h(j)$$

Hub update:

$$h(i) = \sum_{i \to j} a(j)$$

Matrix form:

$$a = E^T h, \quad h = Ea$$

Combined:

$$a = E^T E a, \quad h = E E^T h$$

12 Pseudo-Relevance Feedback

Build document graph, use absorbing walk:

$$x = (I - Q)^{-1} R\mathbf{1}$$

Rank documents by escape probability.

13 Learning to Rank in Graphs

Feature engineering + GNNs + joint optimization.

Learn teleport weights w_i via gradient descent.

14 Key Formulas Summary

$$\begin{aligned}
\pi &= P^T \pi \\
\pi_i &= \frac{\deg(i)}{2m} \\
CT(i,j) &= H(i,j) + H(j,i) \\
CT(s,t) &= \frac{2m}{C_{\text{eff}}(s,t)} \\
x &= (I - Q)^{-1} R\mathbf{1} \\
L &= D - A \\
PR(i) &= \frac{1-\alpha}{N} + \alpha \sum_{j \to i} \frac{PR(j)}{\text{out}(j)}
\end{aligned}$$

15 Convergence and Complexity

Power iteration: $O(m)$ per iteration.

16 Random Walk $\leftrightarrow$ Electrical Network

Commute time $\leftrightarrow$ effective resistance.

Escape probability $\leftrightarrow$ voltage.

17 Relatedness Measures Comparison

Shortest path, max flow, effective conductance, commute time, hitting time, PageRank, SimRank, HITS.

18 Practical Applications

IR, recommendation, social networks, knowledge graphs.

19 Implementation Tips

Sparse matrices, parallelism, approximations, caching, push vs pull strategies.

Vector Similarity, Losses, and Retrieval Cheatsheet

1 Vector Similarities

1.1 Dot Product

$$\text{sim}(q, d) = \sum_i q_i \times d_i$$

Use: Basic similarity for dense vectors, unnormalized.

1.2 Cosine Similarity

$$\text{sim}(q, d) = \frac{q \cdot d}{\|q\| \cdot \|d\|}$$

Use: Normalized similarity, range $[-1, 1]$, length-invariant.

1.3 Euclidean Distance

$$\text{dist}(q, d) = \sqrt{\sum_i (q_i - d_i)^2}$$

Use: L2 distance, smaller = more similar.

1.4 Manhattan Distance

$$\text{dist}(q, d) = \sum_i |q_i - d_i|$$

Use: L1 distance, faster computation than Euclidean.

1.5 MaxSim (ColBERT)

$$\text{sim}(Q, D) = \sum_j \max_k (q_j \cdot d_k)$$

Use: Multi-vector retrieval; for each query word, find best doc word match and sum.

2 Loss Functions

2.1 Pointwise Loss

$$L = -\log \sigma(s_+) - \log(1 - \sigma(s_-))$$

Use: Binary classification, s_+ relevant score, s_- irrelevant score.

2.2 Pairwise Ranking Loss

$$L = -\log \sigma(s_+ - s_-)$$

Use: Ensure relevant doc scores higher than irrelevant.

2.3 Triplet Loss

$$L = \max(0, \text{margin} + \text{sim}(q, d_-) - \text{sim}(q, d_+))$$

Use: Enforce margin between positive and negative pairs.

2.4 Listwise Softmax Loss

$$L = -\log \frac{\exp(s_+)}{\sum_i \exp(s_i)}$$

Use: Rank entire list; treats as classification over all docs.

2.5 Contrastive Loss

$$L = -\log \frac{\exp(\text{sim}(q, d_+)/\tau)}{\sum_i \exp(\text{sim}(q, d_i)/\tau)}$$

Use: In-batch negatives, τ temperature parameter.

3 Regularization Terms

3.1 L1 Regularization (Lasso)

$$R = \lambda \sum_i |v_i|$$

Use: Encourage sparsity, creates exact zeros.

3.2 L2 Regularization (Ridge)

$$R = \lambda \sum_i v_i^2$$

Use: Prevent large weights, smooth solutions.

3.3 FLOPS Regularization

$$R = \lambda \sum_i (\text{ReLU}(v_i))^2$$

Use: Sparse retrieval, penalizes only positive values.

3.4 SPLADE Loss

$$L = -\log P(d_+|q) + \alpha \cdot \text{FLOPS}(q) + \beta \cdot \text{FLOPS}(d)$$

Use: Balance relevance and sparsity for query and document.

4 Attention Mechanisms

4.1 Self-Attention Score

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Use: Transformer attention, d_k = dimension of keys.

4.2 Cross-Attention

$$\text{CrossAttn}(Q_{\text{query}}, K_{\text{doc}}, V_{\text{doc}}) = \text{softmax}\left(\frac{Q_{\text{query}}K_{\text{doc}}^T}{\sqrt{d_k}}\right)V_{\text{doc}}$$

Use: Query attends to document in cross-encoder.

4.3 Multi-Head Attention

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

where

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Use: Multiple attention perspectives in parallel.

5 Hashing Functions

5.1 LSH Hash Function

$$h(v) = \text{sign}(v \cdot r)$$

Use: Random hyperplane projection, r random vector.

5.2 Multi-Probe LSH

$$\text{hash}(v) = [h_1(v), h_2(v), \ldots, h_L(v)]$$

Use: L hash functions create bucket signature.

5.3 LSH Probability

$$P(h(v_1) = h(v_2)) = 1 - \frac{\theta}{\pi}$$

Use: Probability two vectors hash same; θ angle between them.

5.4 Min-Hash

$$h(S) = \min\{\pi(x) : x \in S\}$$

Use: Set similarity, π random permutation.

6 Evaluation Metrics

6.1 Mean Reciprocal Rank (MRR)

$$\text{MRR} = \frac{1}{|Q|} \sum_{q} \frac{1}{\text{rank}_q}$$

Use: Average reciprocal ranks of first relevant doc.

6.2 MRR@k

$$\text{MRR@k} = \frac{1}{|Q|} \sum_{q} \frac{1}{\text{rank}_q} \text{ if } \text{rank}_q \leq k \text{ else } 0$$

Use: MRR considering only top k results.

6.3 Recall@k

$$\text{Recall@k} = \frac{|\{\text{relevant docs in top k}\}|}{|\{\text{all relevant docs}\}|}$$

Use: Fraction of relevant docs retrieved in top k.

6.4 Precision@k

$$\text{Precision@k} = \frac{|\{\text{relevant docs in top k}\}|}{k}$$

Use: Fraction of top k that are relevant.

6.5 NDCG@k

$$\text{DCG@k} = \sum_{i=1}^{k} \frac{2^{\text{rel}_i} - 1}{\log_2(i+1)}, \quad \text{NDCG@k} = \frac{\text{DCG@k}}{\text{IDCG@k}}$$

Use: Rank-aware metric; higher positions weighted more.

6.6 Mean Average Precision (MAP)

$$\text{AP} = \frac{1}{|R|} \sum_{i=1}^{n} P(i) \cdot \text{rel}(i), \quad \text{MAP} = \frac{1}{|Q|} \sum_{q} \text{AP}_q$$

Use: Average precision across all relevant docs.

7 BM25 Formula

$$\text{BM25}(q, d) = \sum_{i} \text{IDF}(q_i) \frac{f(q_i, d)(k_1 + 1)}{f(q_i, d) + k_1(1 - b + b \cdot |d|/\text{avgdl})}$$

$$\text{IDF}(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5} + 1$$

Use: Sparse baseline; term frequency with diminishing returns and length normalization.

8 TF-IDF

$$\text{TF}(t,d) = \frac{f(t,d)}{|d|}, \quad \text{IDF}(t) = \log \frac{N}{n(t)}, \quad \text{TF-IDF}(t,d) = \text{TF}(t,d) \cdot \text{IDF}(t)$$

Use: Classic sparse retrieval; term importance weighted by rarity.

9 Embedding Aggregations

9.1 Mean Pooling

$$v_{\text{doc}} = \frac{1}{n} \sum_{i=1}^{n} v_i$$

Use: Average word embeddings.

9.2 Max Pooling

$$v_{\text{doc}}[j] = \max_i v_i[j]$$

Use: Max per dimension across all word vectors.

9.3 Weighted Average

$$v_{\text{doc}} = \frac{\sum_i w_i v_i}{\sum_i w_i}$$

Use: Weight by IDF or attention scores.

9.4 CLS Token (BERT)

$$v_{\text{doc}} = v_{[\text{CLS}]}$$

Use: Use special [CLS] token embedding as doc representation.

10 Distance-Based Hashing

10.1 Random Projection

$$y = \text{sgn}(Wx)$$

Use: Project to lower dimension then sign.

10.2 Product Quantization

Split vector into m subvectors, quantize each independently:

$$v = [v_1, v_2, \ldots, v_m] \rightarrow [c_1, c_2, \ldots, c_m]$$

Use: Compress vectors; each subvector mapped to centroid.

10.3 Scalar Quantization

$$v_{\text{quantized}} = \frac{\text{round}(v \cdot \text{scale})}{\text{scale}}$$

Use: Reduce precision.

11 Score Normalization

11.1 Min-Max Normalization

$$s_{\text{norm}} = \frac{s - s_{\min}}{s_{\max} - s_{\min}}$$

Use: Scale scores to $[0, 1]$.

11.2 Z-Score Normalization

$$s_{\text{norm}} = \frac{s - \mu}{\sigma}$$

Use: Zero mean, unit variance.

11.3 Softmax

$$p_i = \frac{\exp(s_i/\tau)}{\sum_j \exp(s_j/\tau)}$$

Use: Convert scores to probabilities.

12 Negative Sampling

12.1 Random Negative

Sample random doc not marked relevant. Use: Simple but easy negatives don't help much.

12.2 Hard Negative (BM25)

Sample high-scoring docs from BM25 not relevant. Use: Challenging negatives.

12.3 Hard Negative (In-batch)

Use other queries' relevant docs as negatives. Use: Efficient.

12.4 Hard Negative (Model-based)

Sample docs model ranks high but aren't relevant. Use: Train on model's mistakes.

13 Approximate Nearest Neighbor Search

13.1 HNSW Search Complexity

Construction: $O(N \log N \cdot M)$, Search: $O(\log N)$ Use: Hierarchical graph, very fast in practice.

13.2 LSH Query Time

$$O\Big(L \cdot \frac{n}{2^k} \cdot d\Big)$$

Use: Probabilistic guarantees, may need multiple tables.

13.3 IVF (Inverted File Index)

Cluster docs into n_{clusters}, search nearest clusters. Search: $O(n_{\text{probe}} \cdot (N/n_{\text{clusters}}) \cdot d)$

14 Quantization Formulas

14.1 8-bit Quantization

$$v_{\text{quant}} = \text{clip}\Big(\text{round}(v \cdot 127/v_{\text{max}}), -128, 127\Big), \quad v_{\text{dequant}} = v_{\text{quant}} \cdot v_{\text{max}}/127$$

Use: Reduce memory 4x, slight accuracy loss.

14.2 Binary Quantization

$$b_i = \begin{cases} 1, & v_i > 0 \\ 0, & \text{else} \end{cases}$$

Use: Extreme compression, fast Hamming distance.

14.3 Hamming Distance

$$\text{dist}(b_1, b_2) = \sum_i \text{XOR}(b_1[i], b_2[i])$$

Use: Distance for binary vectors.

15 Model Distillation

15.1 Soft Label Distillation

$$L_{\text{distill}} = \text{KL}(p_{\text{student}} \| p_{\text{teacher}})$$

where $p = \text{softmax}(\text{logits}/T)$, T temperature. Use: Student mimics teacher's output.

15.2 Hard Label Distillation

$$L_{\text{distill}} = \text{CrossEntropy}(y_{\text{pred}}, \arg\max(p_{\text{teacher}}))$$

Use: Train student on teacher's predictions.

15.3 Combined Loss

$$L = \alpha \cdot L_{\text{task}} + (1 - \alpha) \cdot L_{\text{distill}}$$

Use: Balance task loss and distillation, α weight parameter.

16 Generative Retrieval (DSI)

16.1 Document Indexing Loss

$$L_{\text{index}} = -\log P(\text{docid} \mid \text{doc\_tokens})$$

Use: Learn to generate doc ID from content.

16.2 Query-to-DocID Loss

$$L_{\text{retrieval}} = -\log P(\text{docid} \mid \text{query})$$

Use: Generate doc ID directly from query.

16.3 Roundtrip Loss

$$L_{\text{round}} = -\log P(\text{doc\_tokens} \mid \text{docid}) \cdot P(\text{docid} \mid \text{doc\_tokens})$$

Use: Enforce bidirectional consistency.

16.4 Beam Search for ID Generation

At each step, keep top k candidates:

$$\text{score(id)} = \prod_t P(\text{token}_t \mid \text{query}, \text{token}_{<t})$$

Use: Generate structured doc IDs incrementally.

17 Key Architecture Comparisons

17.1 Bi-encoder (Late Interaction)

$$\text{enc}_q(\text{query}) \to v_q, \quad \text{enc}_d(\text{doc}) \to v_d, \quad \text{score} = v_q \cdot v_d$$

Pro: Fast, pre-compute docs, ANN search.
Con: Information bottleneck, no interaction.

17.2 Cross-encoder (Early Interaction)

$$\text{score} = \text{enc}([\text{query}, \text{doc}])$$

Pro: Full interaction, very accurate.
Con: Very slow, no pre-compute, no ANN.

17.3 ColBERT (Multi-vector Late)

$$\text{enc}_q(\text{query}) \to [v_1, v_2, \dots], \quad \text{enc}_d(\text{doc}) \to [u_1, u_2, \dots]$$

$$\text{score} = \sum_j \max_k (v_j \cdot u_k)$$

Pro: Word-level detail, ANN possible.
Con: Multiple probes, more storage.

18 PLAID Stages

18.1 Stage 1: Centroid Collection

For each query vector, find top-r centroids. Take union across all query vectors.
Use: Coarse filtering with cluster centroids.

18.2 Stage 2: Centroid Filtering

Keep centroid c if:

$$\max_j (q_j \cdot c) > \text{threshold}$$

Use: Remove unsupported centroids.

18.3 Stage 3: Document Scoring

$$\text{score\_approx}(q, d) \text{ using only centroids}$$

Keep top-k documents.
Use: Approximate scoring for pruning.

18.4 Stage 4: Exact Scoring

Decompress:

$$v = \text{centroid} + \text{residual}, \quad \text{score\_exact}(q, d) = \sum_j \max_k (q_j \cdot v_k)$$

Use: Full MaxSim on filtered set.

19 Training Techniques

19.1 Learning Rate Warmup

$$\text{lr}(t) = \text{lr}_{\max} \cdot \min(1, t/t_{\text{warmup}})$$

19.2 Linear Decay

$$\text{lr}(t) = \text{lr}_{\max} \cdot \max(0, (t_{\max} - t)/t_{\max})$$

19.3 Gradient Clipping

$$\text{if } \|\nabla\| > \text{threshold: } \nabla \leftarrow \nabla \cdot \frac{\text{threshold}}{\|\nabla\|}$$

19.4 Batch Size Effect on Learning

$$\text{Effective batch} = \text{batch\_size} \times \text{num\_accumulation\_steps}$$

20 Sparse Retrieval Specifics

20.1 SPLADE Encoding

$$w = \text{ReLU}(\text{MLP}(\text{BERT}(\text{text}))), \quad \text{sparse\_vector}[i] = \begin{cases} w[i], & w[i] > 0 \\ 0, & \text{otherwise} \end{cases}$$

20.2 SPLADE Scoring

$$\text{score}(q, d) = \sum_i q[i] \cdot d[i]$$

Only non-zero dimensions contribute. Use: Efficient inverted index lookup.

20.3 Vocabulary Expansion

Token "cat" $\rightarrow$ high weights at dimensions for "cat", "feline", "kitten".
Use: Automatic synonym matching.

20.4 Inverted Index Structure

For term t:

$$\text{posting\_list} = [(doc\_id_1, w_1), (doc\_id_2, w_2), \dots]$$

Only store docs with non-zero weight. Use: Classical IR structure for learned sparse vectors.

Similarity Search & LSH - Complete Cheatsheet

Course: Indexing and Retrieving Text and Graphs
Student: brycisliova
Date: 2025-11-18

1 Problem Setup

- **Point Query:** Given q, find K most similar documents from N docs ($O(N)$ naive)
- **All-Pairs:** For each doc, find K most similar ($O(N^2)$ naive)
- **Range Query:** Find all docs within distance r from q ($O(N)$ naive)

Goal: Achieve $o(N)$ or $\tilde{O}(N)$ time using LSH.

2 Distance & Similarity Measures

2.1 Hamming Distance (bit vectors)

$$\|x, y\|_H = \sum_{i=1}^{N} \mathbf{1}_{x_i \neq y_i} \in [0, N]$$

2.2 L1 Distance (Manhattan)

$$\|a - b\|_1 = \sum_j |a_j - b_j|$$

2.3 L2 Distance (Euclidean)

$$\|a - b\|_2 = \sqrt{\sum_j (a_j - b_j)^2}$$

2.4 Cosine Similarity

$$\text{sim}(a, b) = \frac{a \cdot b}{\|a\|_2 \|b\|_2} \in [-1, 1], \quad \theta = \arccos(\text{sim}(a, b))$$

2.5 Dot Product Similarity

$$\text{sim}(a, b) = a \cdot b$$

2.6 Jaccard Similarity (sets)

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \in [0,1], \quad d(A,B) = 1 - J(A,B)$$

3 Traditional Hashing (Exact Matching)

3.1 Hash Function Families

- Weakly Universal: $\Pr(h(x_1) = h(x_2)) \leq 1/M$ for all $x_1 \neq x_2$
- Strongly Universal: $\Pr(h(x_1) = y_1 \wedge h(x_2) = y_2) = 1/M^2$ for all $x_1 \neq x_2$, $y_1 \neq y_2$

3.2 Perfect Hashing (O(1) worst-case lookup)

- Level 1: Hash n items into n buckets using f, retry until $\sum_i b_i^2 \leq 4n$
- Level 2: For bucket i with b_i items, use g_i into $2b_i^2$ slots, retry until zero collisions

Space: $O(n)$, **Preprocessing:** $O(n)$ expected, **Lookup:** $O(1)$ worst-case

3.3 Analysis Tools

1. Linearity of Expectation: $E[X+Y] = E[X] + E[Y]$
2. Union Bound: $\Pr(A \cup B) \leq \Pr(A) + \Pr(B)$
3. Markov's Inequality: $\Pr(X \geq a) \leq E[X]/a$
4. Geometric Distribution: $E[\text{trials to success}] = 1/p$

4 Locality-Sensitive Hashing (LSH)

4.1 Definition

A hash family $\mathcal{F}$ is (d_1, d_2, p_1, p_2)-sensitive if:

$$\begin{cases} \text{distance}(x,y) \leq d_1 \implies \Pr[h(x) = h(y)] \geq p_1 & (\text{close} \to \text{high collision}) \\ \text{distance}(x,y) \geq d_2 \implies \Pr[h(x) = h(y)] \leq p_2 & (\text{far} \to \text{low collision}) \end{cases}$$

with $d_1 < d_2$ and $p_1 > p_2$.

4.2 Amplification

- **AND Construction (k functions):** p_1^k (close), p_2^k (far)
- **OR Construction (L tables):** $1 - (1 - p_1^k)^L$ (close), $1 - (1 - p_2^k)^L$ (far)

4.3 Parameter Selection

$$k = \frac{\log n}{\log(1/p_2)}, \quad L = n^\rho, \quad \rho = \frac{\log(1/p_1)}{\log(1/p_2)} < 1$$

5 LSH for Hamming Distance

5.1 Hash Family

$$h_i(x) = x_i$$

5.2 Collision Probability

$$\Pr[h_i(x) = h_i(y)] = 1 - \frac{\|x, y\|_H}{N}$$

5.3 K-Bit Sketch (AND Construction)

$$h_{\text{k-bits}}(x) = (x_{i_1}, x_{i_2}, \ldots, x_{i_k})$$

5.4 Python Implementation

```python
def hamming_distance(x, y):
    return sum(xi != yi for xi, yi in zip(x, y))

def hamming_lsh_k_bits(x, bit_positions):
    return tuple(x[i] for i in bit_positions)

def build_hamming_lsh(data, k, L):
    tables = []
    for l in range(L):
        bits = random.sample(range(N), k)
        table = defaultdict(list)
        for x in data:
            hash_val = hamming_lsh_k_bits(x, bits)
            table[hash_val].append(x)
        tables.append((bits, table))
    return tables

def query_hamming_lsh(q, tables, r):
    candidates = set()
    for bits, table in tables:
        hash_val = hamming_lsh_k_bits(q, bits)
        for x in table[hash_val]:
            if hamming_distance(q, x) <= r:
                yield x
            candidates.add(x)
            if len(candidates) > 2*L:
                return
```

6 Problem Setup

- **Query Types:**
  - Point Query: Given q, find K most similar docs from N docs $\rightarrow O(N)$ naive
  - All-Pairs: For each doc, find K most similar $\rightarrow O(N^2)$ naive
  - Range Query: Find all docs within distance r from $q \rightarrow O(N)$ naive

- **Goal:** Achieve $o(N)$ or $\tilde{O}(N)$ time using LSH

7 Distance & Similarity Measures

7.1 Hamming Distance (bit vectors)

$$\|x - y\|_H = \sum_i \mathbf{1}_{x_i \neq y_i}, \quad \text{metric } \in [0, N]$$

```
def hamming_distance(x, y):
    return sum(xi != yi for xi, yi in zip(x, y))
```

7.2 L1 Distance (Manhattan)

$$\|a - b\|_1 = \sum_j |a_j - b_j|$$

```
def l1_distance(a, b):
    return sum(abs(ai - bi) for ai, bi in zip(a, b))
```

7.3 L2 Distance (Euclidean)

$$\|a - b\|_2 = \sqrt{\sum_j (a_j - b_j)^2}$$

```
def l2_distance(a, b):
    return sqrt(sum((ai - bi)**2 for ai, bi in zip(a, b)))
```

7.4 Cosine Similarity

$$\text{sim}(a, b) = \frac{a \cdot b}{\|a\|_2 \|b\|_2} \in [-1, 1], \quad \theta = \arccos(\text{sim}(a, b))$$

```
def cosine_similarity(a, b):
    return dot(a, b) / (norm(a) * norm(b))
```

7.5 Dot Product Similarity

$$\text{sim}(a, b) = a \cdot b \in \mathbb{R}$$

```
def dot_product_similarity(a, b):
    return dot(a, b)
```

7.6 Jaccard Similarity (sets)

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \in [0, 1], \quad d_J = 1 - J(A, B)$$

```
def jaccard_similarity(A, B):
    return len(A & B) / len(A | B)
```

8 Traditional Hashing (Exact Matching)

- Weakly Universal: $\Pr[h(x_1) = h(x_2)] \leq 1/M$
- Strongly Universal: $\Pr[h(x_1) = y_1 \wedge h(x_2) = y_2] = 1/M^2$

Perfect Hashing:

- Level 1: Hash n items into n buckets until $\sum_i b_i^2 \leq 4n$
- Level 2: For bucket i with b_i items, use g_i into $2b_i^2$ slots

9 Locality-Sensitive Hashing (LSH) - Core Concept

Definition: Hash family F is (d_1, d_2, p_1, p_2)-sensitive if:

$$\begin{cases} \text{distance}(x, y) \leq d_1 \implies \Pr[h(x) = h(y)] \geq p_1 \\ \text{distance}(x, y) \geq d_2 \implies \Pr[h(x) = h(y)] \leq p_2 \end{cases}$$

9.1 Amplification

- AND Construction (k functions): p_1^k, p_2^k
- OR Construction (L tables): $1 - (1 - p_1^k)^L$, $1 - (1 - p_2^k)^L$

9.2 Parameter Selection

$$k = \frac{\log n}{\log(1/p_2)}, \quad L = n^\rho, \quad \rho = \frac{\log(1/p_1)}{\log(1/p_2)} < 1$$

10 LSH for Hamming Distance

```
def hamming_lsh_single(x, i):
    return x[i]

def hamming_lsh_k_bits(x, bit_positions):
    return tuple(x[i] for i in bit_positions)
```

```
def build_hamming_lsh(data, k, L):
    tables = []
    for     in range(L):
        bits = random.sample(range(N), k)
        table = defaultdict(list)
        for x in data:
            hash_val = hamming_lsh_k_bits(x, bits)
            table[hash_val].append(x)
        tables.append((bits, table))
    return tables
```

11 LSH for L1 Distance

```
def l1_to_hamming(x, C):
    bits = []
    for val in x:
        bits.extend([1]*val + [0]*(C-val))
    return bits
```

12 LSH for Cosine Similarity

```
def cosine_lsh_single(x, u):
    return 1 if dot(u, x) >= 0 else -1

def cosine_lsh_sketch(x, random_vectors):
    return tuple(cosine_lsh_single(x, u) for u in random_vectors)
```

```
def random_unit_vector(d):
    v = [random.gauss(0,1) for _ in range(d)]
    norm = sqrt(sum(x**2 for x in v))
    return [x/norm for x in v]
```

13 LSH for Dot Product, L2, Jaccard, Shingling

```
# Dot Product -> Cosine
def dot_to_cosine_transform(x, max_norm):
    x_scaled = [xi * max_norm / norm(x) for xi in x]
    padding = sqrt(1 - sum(xi**2 for xi in x_scaled))
    return x_scaled + [padding]

# L2
def l2_lsh_single(x, direction, offset, a):
    projection = dot(x, direction) + offset
    return int(projection / a)
```

```
# Min-hash
def min_hash_linear(S, a, b, p):
    return min((a*s + b) % p for s in S)
```

```
# Shingling
def word_shingles(text, w=4):
    words = text.split()
    return {tuple(words[i:i+w]) for i in range(len(words)-w+1)}

def char_shingles(text, w=8):
    return {text[i:i+w] for i in range(len(text)-w+1)}
```

14 Complete LSH Data Structure

```
class LSH:
    def __init__(self, hash_family, k, L):
        self.k = k
```

```
        self.L = L
        self.tables = []
        self.hash_params = []

        for _ in range(L):
            params = [hash_family.generate_params() for _ in range(k)]
            self.hash_params.append(params)
            self.tables.append(defaultdict(list))

    def _hash(self, x, params_list):
        return tuple(hash_family(x, p) for p in params_list)

    def insert(self, x, data=None):
        for i in range(self.L):
            hash_val = self._hash(x, self.hash_params[i])
            self.tables[i][hash_val].append(data or x)

    def query(self, q, max_candidates=None):
        candidates = set()
        for i in range(self.L):
            hash_val = self._hash(q, self.hash_params[i])
            for x in self.tables[i][hash_val]:
                candidates.add(x)
                if max_candidates and len(candidates) >= max_candidates
   :
                    return candidates
        return candidates
```

13. QUERY ALGORITHMS

RANGE QUERY (find all within distance r)

```
def range_query_lsh(q, lsh, distance_func, r):
    """
    Returns all points x with distance(q, x)  r
    """
    candidates = lsh.query(q, max_candidates=2*lsh.L)
    results = []
    for x in candidates:
        if distance_func(q, x) <= r:
            results.append(x)
    return results
```

K-NEAREST NEIGHBORS (approximate)

```
def knn_query_lsh(q, lsh, distance_func, k):
    """
    Returns approximately k nearest neighbors
    Strategy: Try increasing radii until k neighbors found
    """
    r = initial_radius_guess()
    for _ in range(max_iterations):
```

```
        results = range_query_lsh(q, lsh, distance_func, r)
        if len(results) >= k:
            # Sort and return top k
            results.sort(key=lambda x: distance_func(q, x))
            return results[:k]
        r *= 2  # Increase radius
    return results
```

c-APPROXIMATE RANGE QUERY

If $\exists p$ with $d(q,p) \leq r$, return some p' with $d(q,p') \leq cr$.
Guarantees (with high probability):

- Near point ($d \leq r$) collides in at least one table
- At most $2L$ far points ($d \geq cr$) collide

14. ALL-PAIRS SIMILARITY

Problem: For each of n docs, find K most similar docs. Goal: $o(n^2)$ time

All Pairs LSH

```
def all_pairs_lsh(documents, num_permutations=100, threshold=50):
    """
    Find similar pairs using min-hash LSH
    threshold: minimum number of hash collisions to verify
    """
    # Phase 1: Compute min-hash signatures
    signatures = []
    for doc in documents:
        sig = []
        for  in permutations:
            sig.append(min_hash(doc, ))
        signatures.append(sig)

    # Phase 2: Find candidate pairs
    candidate_counts = defaultdict(int)
    for perm_idx in range(num_permutations):
        # Group by hash value for this permutation
        buckets = defaultdict(list)
        for doc_id, sig in enumerate(signatures):
            buckets[sig[perm_idx]].append(doc_id)

        # All pairs in same bucket are candidates
        for bucket in buckets.values():
            for i in range(len(bucket)):
                for j in range(i+1, len(bucket)):
```

```
                    pair = (bucket[i], bucket[j])
                    candidate_counts[pair] += 1

    # Phase 3: Verify candidates with enough collisions
    results = defaultdict(list)
    for (i, j), count in candidate_counts.items():
        if count >= threshold:
            sim = compute_actual_similarity(documents[i], documents[j])
            results[i].append((j, sim))
            results[j].append((i, sim))

    # Phase 4: Return top K for each doc
    for doc_id in results:
        results[doc_id].sort(key=lambda x: x[1], reverse=True)
        results[doc_id] = results[doc_id][:K]

    return results
```

External Sort Approach (very large datasets)

- For each permutation π, create file f_π
- For each document d: write $(\min(\pi(d)), d.id)$ to f_π
- Sort f_π by min-hash value
- Create file g_π; for each contiguous block in f_π, write all pairs $(d1, d2)$ to g_π
- Merge all g_π files, counting co-occurrences; verify pairs with count $\geq$ threshold

15. GRAPH CONTRACTION (Web Crawling Application)

Motivation: Detect and merge mirror/duplicate web pages

Detect Mirrors

```
def detect_mirrors(pages, similarity_threshold=0.9):
    """
    Returns groups of mirror pages
    """
    shingle_sets = {p: word_shingles(p.content) for p in pages}
    lsh = build_min_hash_lsh(shingle_sets, num_perms=100)
    mirrors = defaultdict(set)
    for p1 in pages:
        candidates = lsh.query(shingle_sets[p1])
        for p2 in candidates:
            if p1 != p2:
```

```
                sim = jaccard_similarity(shingle_sets[p1], shingle_sets[p2])
                if sim >= similarity_threshold:
                    mirrors[p1].add(p2)
    return merge_equivalence_classes(mirrors)
```

Contract Web Graph

```
def contract_web_graph(graph, mirror_groups):
    canonical = {}
    for group in mirror_groups:
        rep = choose_representative(group)
        for page in group:
            canonical[page] = rep
    new_graph = defaultdict(set)
    for u, v in graph.edges():
        u_canon = canonical.get(u, u)
        v_canon = canonical.get(v, v)
        if u_canon != v_canon:
            new_graph[u_canon].add(v_canon)
    return new_graph
```

Cascading Effect: After contracting $v1, v2 \to v$:

- $u1 \to v, u2 \to v$
- If $u1, u2$ now have very similar outlinks, they may be detected as mirrors in next iteration
- Iterate until convergence (no new mirrors)

NEURAL TEXT ANALYSIS & EMBEDDINGS - COMPLETE CHEATSHEET

Course: Indexing and Retrieving Text and Graphs
Student: brycisliova

2025-11-18

1 WORD EMBEDDINGS FOUNDATIONS

1.1 Traditional One-Hot Encoding

- bank = [1, 0, 0, 0, ..., 0] vocab_size dimensions
- river = [0, 1, 0, 0, ..., 0]
- Problem: No semantic relationship, sparse, high-dimensional

1.2 Neural Dense Embeddings

- bank = [0.23, -0.45, 0.67, 0.12, ...] 50-300 dimensions
- river = [0.19, -0.41, 0.70, 0.15, ...]
- Benefit: Semantic similarity, compact, continuous
- Key property: $\cos(v_{\text{bank}}, v_{\text{river}}) > \cos(v_{\text{bank}}, v_{\text{xylophone}})$

1.3 Word2Vec CBOW

$$P(w_t \mid w_{t-k}, ..., w_{t-1}, w_{t+1}, ..., w_{t+k})$$

Predict word from context.

1.4 Word2Vec Skip-Gram

$$P(\text{context} \mid w_t)$$

Predict context from word.

1.5 GloVe (Global Vectors)

$$w_i \cdot w_j + b_i + b_j \approx \log(X_{ij})$$

where X_{ij} = co-occurrence count of words i and j.

1.6 Contextualized Embeddings

- BERT, ELMo: Different vectors for the same word in different contexts
- bank in river bank $\neq$ bank in savings bank

1.7 Word Embedding Analogies

$$v_{\text{king}} - v_{\text{man}} + v_{\text{woman}} \approx v_{\text{queen}}$$

$$v_{\text{Paris}} - v_{\text{France}} + v_{\text{Germany}} \approx v_{\text{Berlin}}$$

1.8 Similarity Metric

$$\text{sim}(w_1, w_2) = \cos(v_{w_1}, v_{w_2}) = \frac{v_{w_1} \cdot v_{w_2}}{\|v_{w_1}\|\|v_{w_2}\|}$$

2 CONVOLUTIONAL NEURAL NETWORKS FOR TEXT

2.1 CNN Architecture Pipeline

1. Input: $[w_1, w_2, ..., w_n]$ word sequence
2. Embedding: $[e_1, e_2, ..., e_n], \quad e_i \in \mathbb{R}^d$
3. Convolution: Apply filters of width k (typically $k = 3, 4, 5$)
4. Pooling: Max or Average pooling $\rightarrow$ fixed-size vector
5. Dense: Fully connected layers
6. Output: Softmax class probabilities

2.2 Convolution Operation Formula

$$h_i = f(W \cdot [e_i; e_{i+1}; ...; e_{i+k-1}] + b)$$

where

$$W \in \mathbb{R}^{m \times (k \cdot d)}, \quad k = \text{filter width}, \quad f = \text{activation function (ReLU or tanh)}$$

2.3 Pooling

- Max pooling: $c = \max(h_1, h_2, ..., h_n)$
- Average pooling: $c = \frac{1}{n} \sum_{i=1}^{n} h_i$

2.4 Applications

Document Classification, Sentiment Analysis, Topic Classification, Query Classification

2.5 Pros and Cons

- Pros: Fast, captures local patterns, no vanishing gradients
- Cons: Fixed receptive field, loses positional information, limited context

3 RECURRENT NEURAL NETWORKS

3.1 Basic RNN (Elman Network)

$$h_t = \sigma(x_t W_h + h_{t-1} U_h + b_h), \quad y_t = \text{softmax}(h_t W_y + b_y)$$

3.2 Jordan Network Alternative

$$h_t = \sigma(x_t W_h + y_{t-1} U_h + b_h)$$

3.3 RNN Properties

- Turing complete, sequential processing, weight sharing
- Vanishing and exploding gradient problems

4 GATED RECURRENT UNIT (GRU)

4.1 GRU Equations

$$r_t = \sigma(x_t W_r + h_{t-1} U_r + b_r)$$
$$c_t = \tanh(x_t W_h + (r_t \odot h_{t-1}) U_h + b_h)$$
$$z_t = \sigma(x_t W_z + h_{t-1} U_z + b_z)$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot c_t$$

4.2 Gate Explanations

- Reset gate $r_t \in [0, 1]$: how much past state to forget
- Update gate $z_t \in [0, 1]$: how to blend old state vs new candidate

5 LONG SHORT-TERM MEMORY (LSTM)

5.1 LSTM Equations

$$i_t = \sigma(x_t U_i + h_{t-1} W_i + b_i)$$
$$f_t = \sigma(x_t U_f + h_{t-1} W_f + b_f)$$
$$o_t = \sigma(x_t U_o + h_{t-1} W_o + b_o)$$
$$g_t = \tanh(x_t U_g + h_{t-1} W_g + b_g)$$
$$c_t = c_{t-1} \odot f_t + g_t \odot i_t, \quad h_t = \tanh(c_t) \odot o_t$$

5.2 Bidirectional LSTM

$$\overrightarrow{h_t} \text{ (left context)}, \quad \overleftarrow{h_t} \text{ (right context)}, \quad h_t = [\overrightarrow{h_t}; \overleftarrow{h_t}]$$

6 USING LSTM STATES FOR APPLICATIONS

- Document classification: $\bar{h} = \frac{1}{T} \sum_{t=1}^{T} h_t$
- Sentence pair similarity: encode each sentence with LSTM, compare with cosine similarity
- Textual entailment: encode both sentences separately or together

7 RNNs AS LANGUAGE MODELS

$$P(\text{sentence}) = \prod_{t=1}^{T} P(x_t \mid x_1, ..., x_{t-1})$$

8 SEQUENCE TO SEQUENCE TRANSLATION

- Encoder-Decoder Architecture
- Fixed-size bottleneck problem $\rightarrow$ solution: attention mechanisms

9 ATTENTION MECHANISMS

$$e_{ij} = \text{score}(g_{i-1}, h_j), \quad a_{ij} = \frac{\exp(e_{ij})}{\sum_j \exp(e_{ij})}, \quad \hat{h_i} = \sum_j a_{ij} \cdot h_j$$

10 SELF-ATTENTION AND TRANSFORMERS

$$k_t = W_K x_t + b_K, \quad q_t = W_Q x_t + b_Q, \quad v_t = W_V x_t + b_V$$
$$e_{t,\tau} = q_t \cdot k_\tau, \quad a_{t,\tau} = \text{softmax}_\tau(e_{t,\tau}), \quad o_t = \sum_\tau a_{t,\tau} v_\tau$$

10.1 Positional Encoding

$$PE(pos, 2i) = \sin(pos/10000^{2i/d}), \quad PE(pos, 2i+1) = \cos(pos/10000^{2i/d})$$

Transformers, Attention, RNNs, and Neural Retrieval

11 Self-Attention and Transformers

11.1 Multi-Head Attention

- Runs h attention heads in parallel.
- Each head has separate W_Q, W_K, W_V matrices.
- Allows focus on different aspects (syntactic vs semantic).

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W_O$$

11.2 Transformer Architecture

- Stacked self-attention layers.
- Each layer:
  - Multi-head attention
  - Feed-forward network
  - Layer normalization & residual connections
- Positional encodings added to input embeddings.

12 Masked Language Modeling (MLM)

- Input sentence with [MASK] tokens.
- Task: Predict masked words using bidirectional context.

Example:

Input: "The [MASK] is giving low interest [MASK]" Target: "bank", "rates"

- Self-supervised pretraining on large corpora.
- Learns rich, context-dependent embeddings.

13 Popular Transformer Models

- **BERT:** Bidirectional encoder, MLM, pretrained on Wikipedia & BookCorpus.
- Variants:
  - RoBERTa, ALBERT, DistilBERT, SpanBERT
  - Domain-specific: BioBERT, SciBERT, CamemBERT
- Other families: GPT (autoregressive), T5 (Text-to-Text), XLNet (permutation-based LM)

Common theme: self-attention, pretraining, fine-tuning, transfer learning.

14 RNNs and Seq2Seq Models

14.1 Abstract RNN

$$h_t = \text{RNN}(h_{t-1}, x_{t-1})$$

14.2 LSTM

- Dual states $[c_t, h_t]$.
- Only h_t used for output.

14.3 GRU

- Single state h_t.
- Simplified version of LSTM.

14.4 Seq2Seq Without Attention

$$\begin{aligned}
&\text{Encoder: } h_n = \text{RNN}_e(h_{n-1}, x_{n-1}), \quad n = 1, \ldots, N \\
&\text{Decoder: } g_0 = h_N, \quad g_m = \text{RNN}_d(g_{m-1}, y_{m-1}), \quad m = 1, \ldots, M \\
&y_m \sim \text{OUT}(g_m)
\end{aligned}$$

14.5 Seq2Seq With Attention

$$
\begin{aligned}
e_{mn} &= g_{m-1} \cdot h_n \\
a_{mn} &= \text{softmax}_n(e_{mn}) \\
\hat{h}_m &= \sum_n a_{mn} h_n \\
g_m &= \text{RNN}_d(g_{m-1}, \hat{h}_m, y_{m-1}) \\
y_m &\sim \text{OUT}(g_m)
\end{aligned}
$$

15 Retrieval & Semantic Search

15.1 Document & Query Representation

- Traditional: TF-IDF, sparse vectors.
- Neural: Dense embeddings (LSTM/BERT) capture semantic meaning.

15.2 Relevance Scoring

- Traditional: cosine similarity / BM25.
- Neural: embedding similarity, cross-encoder, bi-encoder, poly-encoder.

15.3 Dense Retrieval

- Encode documents offline, store in ANN index (FAISS, Annoy).
- Encode query at runtime $\rightarrow$ retrieve top-k nearest.
- Hybrid: BM25 first, neural re-ranking.

15.4 Training

- Siamese networks: query/document embeddings $\rightarrow$ dot product or MLP.
- Losses:
  - Cross-Entropy
  - Margin loss (ranking)
  - Contrastive loss (embedding similarity)

16 Training & Optimization

16.1 Optimizers

SGD, Adam, AdamW, RMSprop, AdaGrad

16.2 Learning Rate Scheduling

Constant, step decay, exponential decay, cosine annealing, warm-up

16.3 Gradient & Regularization

- Gradient clipping: $\text{clip}(g, \text{threshold})$
- Dropout, weight decay, early stopping, layer normalization

16.4 Batching & Padding

- Pad sequences to max length per batch.
- Use attention mask to ignore padding in attention computations.

17 Classification Metrics

- **Accuracy:** Fraction of correct predictions

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Precision:** Fraction of positive predictions that are correct

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Recall:** Fraction of actual positives that are predicted

$$\text{Recall} = \frac{TP}{TP + FN}$$

- **F1-Score:** Harmonic mean of precision and recall

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

18 Retrieval Metrics

- **Mean Average Precision (MAP):**

$$AP = \frac{1}{R} \sum_{k=1}^{n} P(k) \cdot \text{rel}(k)$$

where $R =$ total relevant documents, $P(k) =$ precision at k, $\text{rel}(k) =$ relevance indicator

- **Normalized Discounted Cumulative Gain (NDCG):**

$$DCG = \sum_{i=1}^{p} \frac{2^{\text{rel}_i} - 1}{\log_2(i+1)}, \quad NDCG = \frac{DCG}{IDCG}$$

- **Mean Reciprocal Rank (MRR):**

$$MRR = \frac{1}{Q} \sum_{q=1}^{Q} \frac{1}{\text{rank}_q}$$

- **Precision at K (P@K):**

$$P@K = \frac{\text{relevant in top-K}}{K}$$

19 Sequence Generation Metrics

- **BLEU:** Measures n-gram overlap

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

  where p_n = modified n-gram precision, BP = brevity penalty

- **ROUGE-N:** Measures recall of n-grams

$$ROUGE\text{-}N = \frac{\sum \text{overlapping n-grams}}{\sum \text{reference n-grams}}$$

- **Perplexity (PPL):**

$$PPL = \exp\left(-\frac{1}{N} \sum_{i=1}^{N} \log P(w_i | w_{<i})\right)$$

20 Embedding Quality Metrics

- Word similarity: correlation with human judgments
- Analogy accuracy: fraction of correct analogies

$$\text{king} - \text{man} + \text{woman} \approx \text{queen}$$

- Clustering purity: embeddings cluster by category
- Downstream task performance: accuracy on classification, retrieval, etc.

21 Practical Implementation Tips

21.1 Hardware Considerations

- GPU essential; CUDA for NVIDIA GPUs
- Batch size limited by GPU memory
- Mixed precision training: float16 + float32

21.2 Debugging Strategies

- Start with small model/dataset; overfit on single batch
- Check gradient flow; watch for vanishing/exploding gradients
- Visualize attention weights; monitor training curves

21.3 Hyperparameter Tuning

- Learning rate: 1e−3 to 1e−4
- Batch size: 16, 32, 64, 128
- Hidden size: 128, 256, 512, 1024
- Layers: 1–12 for transformers
- Dropout: 0.1–0.3, Weight decay: 1e−5 to 1e−3

21.4 Pre-training & Fine-tuning

- Pre-train on large unlabeled corpus
- Fine-tune on task-specific data; optionally freeze early layers

21.5 Transfer Learning Workflow

- Load pre-trained model (BERT, RoBERTa)
- Add task-specific head
- Fine-tune on downstream task

21.6 Data Augmentation for Text

- Back-translation, synonym replacement
- Random insertion/deletion
- Contextualized augmentation via language model

22 Common Pitfalls & Solutions

- Overfitting: more data, regularization, early stopping
- Underfitting: larger model, longer training, better features
- Vanishing gradients: use LSTM/GRU, gradient clipping
- Exploding gradients: gradient clipping, lower learning rate
- Slow training: increase batch size, mixed precision
- OOM errors: reduce batch/model size, gradient accumulation/checkpointing

23 Advanced Topics

23.1 Sparse Attention & Efficient Transformers

- Longformer, BigBird, Linformer, Performer
- Distillation, pruning, quantization

23.2 Memory-Augmented Networks

- Neural Turing Machine, Memory Networks, Transformer-XL

23.3 Multi-Task & Few-Shot Learning

- Shared encoder, task-specific decoders
- Meta-learning, prompt-based learning, in-context learning

23.4 Interpretability

- Attention visualization
- Gradient-based saliency
- Probing tasks, LIME, SHAP

24 Mathematical Foundations

24.1 Activation Functions

- **Sigmoid:** $\sigma(x) = \frac{1}{1+e^{-x}}$, gradient: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
- **Tanh:** $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, gradient: $1 - \tanh^2(x)$

- **ReLU:** $\max(0, x)$, gradient: 1 if $x > 0$, else 0
- **Leaky ReLU:** $\max(\alpha x, x)$
- **GELU:** $x \cdot \Phi(x)$ where Φ is the CDF of standard normal

Neural Text Retrieval - Master Cheatsheet

CS6101: Indexing and Retrieving Text and Graphs

Contents

1 Fundamental Concepts

1.1 Sparse vs Dense Retrieval

Sparse (Traditional - BM25)

- Vector dims = vocabulary size (∼100K+)
- Most values are zero
- Exact word matching only
- Problem: "car" ≠ "automobile"

Dense (Neural)

- Vector dims = 100-1000
- All values non-zero
- Semantic matching
- Solution: "car" ≈ "automobile"

Distributional Hypothesis: "You shall know a word by the company it keeps"

2 Word Embeddings

2.1 Word2Vec (Mikolov et al., 2013)

Skip-gram: Predict context from center word

$$P(\text{context} \mid \text{center}) = \frac{\exp(u_{\text{context}} \cdot v_{\text{center}})}{\sum_w \exp(u_w \cdot v_{\text{center}})}$$

Negative Sampling:

$$L = \log \sigma(u_{\text{pos}} \cdot v_{\text{center}}) + \sum_{i=1}^{k} \log \sigma(-u_{\text{neg}_i} \cdot v_{\text{center}})$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$, $k = 5 - 20$ negative samples.

2.2 GloVe (Pennington et al., 2014)

$$\text{Loss} = \sum_{i,j} f(X_{ij}) \left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij} \right)^2$$

where

$$f(x) = \begin{cases} (x/x_{\max})^\alpha & x < x_{\max} \\ 1 & \text{otherwise} \end{cases}$$

2.3 Word Vector Properties

Semantic Arithmetic:

$$v(\text{king}) - v(\text{man}) + v(\text{woman}) \approx v(\text{queen})$$

Similarity Metrics:

$$\text{Cosine: } \cos(u,v) = \frac{u \cdot v}{\|u\|\|v\|}, \quad \text{Euclidean: } d(u,v) = \|u - v\| = \sqrt{\sum_i (u_i - v_i)^2}$$

3 Sentence & Document Representations

3.1 Simple Aggregation

$$\text{doc\_vector} = \frac{1}{n} \sum_{i=1}^{n} w_i \cdot \text{word\_vector}_i$$

3.2 CNNs for Text

$$c_i = f(W \cdot [v_i, v_{i+1}, \ldots, v_{i+h-1}] + b), \quad \hat{c} = \max(c_1, c_2, \ldots, c_{n-h+1})$$

3.3 RNNs, LSTMs, GRUs

Basic RNN:

$$h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t + b_h), \quad y_t = W_{hy} h_t + b_y$$

LSTM:

$$\begin{aligned}
f_t &= \sigma(W_f [h_{t-1}, x_t] + b_f) \\
i_t &= \sigma(W_i [h_{t-1}, x_t] + b_i) \\
\tilde{C}_t &= \tanh(W_C [h_{t-1}, x_t] + b_C) \\
C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \\
o_t &= \sigma(W_o [h_{t-1}, x_t] + b_o) \\
h_t &= o_t \odot \tanh(C_t)
\end{aligned}$$

GRU:

$$\begin{aligned}
r_t &= \sigma(W_r [h_{t-1}, x_t]) \\
z_t &= \sigma(W_z [h_{t-1}, x_t]) \\
\tilde{h}_t &= \tanh(W [r_t \odot h_{t-1}, x_t]) \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t
\end{aligned}$$

4 Deep Retrieval Architectures

4.1 Architecture Comparison

Architecture	Accuracy	Speed	Storage	Use Case
Single-Vector (Bi-encoder)	Good	Fastest	Minimal	Initial retrieval, millions of docs
ColBERT (Multi-vector)	Better	Fast	Moderate	Medium corpus, better accuracy
Cross-Encoder	Best	Very Slow	Minimal	Reranking ¡100 docs

4.2 Bi-Encoder

$$q = \text{Encoder}_Q(\text{query}), \quad d = \text{Encoder}_D(\text{doc}), \quad s = q \cdot d$$

4.3 Cross-Encoder

Input: [CLS] query tokens [SEP] doc tokens [SEP]

$$\text{Attention: } A_{ij} = \text{softmax}\left(\frac{Q_i K_j^T}{\sqrt{d_k}}\right)$$

4.4 ColBERT (Multi-Vector)

$$S(Q, D) = \sum_{i=1}^{m} \max_{j=1}^{n} (q_i \cdot d_j)$$

5 Training Dense Retrievers

5.1 Contrastive Learning (InfoNCE)

$$L = -\log \frac{\exp(\text{sim}(q, d^+))}{\exp(\text{sim}(q, d^+)) + \sum_i \exp(\text{sim}(q, d_i^-))}$$

5.2 Negative Sampling

- Random negatives
- BM25 hard negatives
- In-batch negatives
- Model-based hard negatives

6 Approximate Nearest Neighbor (ANN)

6.1 FAISS: IVF + PQ

- IVF: cluster into k centroids
- PQ: split vectors, quantize sub-vectors

6.2 HNSW: Graph-based search

- Build hierarchical graph
- Navigate greedily from top

7 Evaluation Metrics

7.1 Ranking Metrics

$$\text{Recall@k} = \frac{|\text{Relevant} \cap \text{Top-}k|}{|\text{Relevant}|}, \quad \text{Precision@k} = \frac{|\text{Relevant} \cap \text{Top-}k|}{k}$$

$$\text{MRR} = \frac{1}{Q} \sum_{q=1}^{Q} \frac{1}{\text{rank of first relevant doc}}$$

$$\text{NDCG@k} = \frac{\text{DCG@k}}{\text{IDCG@k}}, \quad \text{DCG@k} = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

8 Production Pipelines

8.1 Two-Stage Retrieval

1. Stage 1: Fast retrieval (Bi-encoder / BM25)
2. Stage 2: Rerank top candidates (Cross-encoder / ColBERT)

8.2 Hybrid Retrieval

$$\text{Score} = \alpha \cdot \text{BM25} + (1 - \alpha) \cdot \text{Dense}$$

9 Advanced Topics

9.1 Query Expansion

$$q_{\text{expanded}} = q + \beta \cdot \frac{1}{k} \sum_{i=1}^{k} d_i$$

9.2 Distillation

- Teacher: Cross-encoder
- Student: Bi-encoder
- Loss: KL divergence

DSA5046 – Information Retrieval Study Guide

Contents

1 Information Retrieval Basics

1.1 The IR Problem

- Represent documents and queries.
- Define a similarity function.
- Retrieve the top-k documents for a query.

2 Text Representation

2.1 Bag of Words (BoW)

- Represent each document as a vector of word counts.
- Order, grammar, and semantics are ignored.
- Dimensionality = vocabulary size.

3 TF–IDF

3.1 Term Frequency

$$tf_{t,d} = \frac{\text{count}(t, d)}{\sum_{t'} \text{count}(t', d)}$$

3.2 Document Frequency

$$df_t = |\{d : t \in d\}|$$

3.3 Inverse Document Frequency

$$idf_t = \log \frac{N}{df_t}$$

3.4 TF-IDF Weight

$$w_{t,d} = tf_{t,d} \cdot idf_t$$

4 Vector Space Model

4.1 Document Vector

$$\mathbf{d}_i = (w_{1,i}, w_{2,i}, \ldots, w_{V,i})$$

4.2 Query Vector

Same form as document vector.

4.3 Cosine Similarity

$$\cos(\theta) = \frac{\mathbf{d} \cdot \mathbf{q}}{\|\mathbf{d}\|\|\mathbf{q}\|}$$

5 Latent Semantic Indexing (LSI)

5.1 SVD Decomposition

Given term-document matrix A:

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

5.2 Low-Rank Approximation

$$A_k = U_k \Sigma_k V_k^T$$

5.3 Query Projection

$$q_k = q^T U_k \Sigma_k^{-1}$$

6 Nonnegative Matrix Factorization (NMF)

6.1 Objective

$$A \approx WH, \quad W \geq 0,\ H \geq 0$$

6.2 Optimization

$$\min_{W,H \geq 0} \|A - WH\|_F^2$$

6.3 Multiplicative Updates

$$H \leftarrow H \circ \frac{W^T A}{W^T W H} \qquad W \leftarrow W \circ \frac{A H^T}{W H H^T}$$

7 PLSA

7.1 Model

$$P(d, w) = P(d) \sum_z P(z|d) P(w|z)$$

7.2 E-step

$$P(z|d, w) = \frac{P(w|z)P(z|d)}{\sum_{z'} P(w|z')P(z'|d)}$$

7.3 M-step

$$P(w|z) = \frac{\sum_d n(d,w)P(z|d,w)}{\sum_{w'}\sum_d n(d,w')P(z|d,w)}$$

$$P(z|d) = \frac{\sum_w n(d,w)P(z|d,w)}{\sum_{z'}\sum_w n(d,w)P(z'|d,w)}$$

8 Latent Dirichlet Allocation (LDA)

8.1 Generative Process

1. Draw topic distribution $\theta_d \sim Dir(\alpha)$.
2. For each word:
   - Choose topic $z \sim \theta_d$.
   - Choose word $w \sim \phi_z$.

8.2 Collapsed Gibbs Sampling

$$P(z_i = k|\mathbf{z}_{-i}, \mathbf{w}) \propto (n_{d,k}^{-i} + \alpha_k) \cdot \frac{n_{k,w_i}^{-i} + \beta_{w_i}}{n_{k,\cdot}^{-i} + \sum_w \beta_w}$$

9 HITS Algorithm

9.1 Update Rules

Authority:

$$a = A^T h$$

Hub:

$$h = Aa$$

Normalize:

$$a \leftarrow \frac{a}{\|a\|}, \quad h \leftarrow \frac{h}{\|h\|}$$

10 PageRank

10.1 Transition Matrix

$$P = \alpha S + (1-\alpha)\frac{1}{n}\mathbf{1}\mathbf{1}^T$$

10.2 PageRank Equation

$$\pi = P^T\pi$$

11 Language Models for IR

11.1 Query Likelihood

$$P(q|d) = \prod_{w \in q} P(w|d)$$

11.2 Smoothing

$$P(w|d) = \frac{tf_{w,d} + \mu P(w|C)}{|d| + \mu}$$

12 BM25

12.1 BM25 Formula

$$BM25(d,q) = \sum_{w \in q} IDF(w) \cdot \frac{f_{w,d}(k_1+1)}{f_{w,d} + k_1 \left(1 - b + b\frac{|d|}{avgdl}\right)}$$

12.2 IDF

$$IDF(w) = \log \frac{N - df_w + 0.5}{df_w + 0.5}$$

13 Evaluation Metrics

13.1 Precision

$$P = \frac{TP}{TP + FP}$$

13.2 Recall

$$R = \frac{TP}{TP + FN}$$

13.3 F1 Score

$$F_1 = \frac{2PR}{P + R}$$

13.4 MAP

$$AP = \frac{1}{R} \sum_{k=1}^{N} P(k) \cdot rel(k)$$

$$MAP = \frac{1}{|Q|} \sum_{q} AP(q)$$

13.5 NDCG

$$DCG@k = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

$$NDCG = \frac{DCG}{IDCG}$$

14 Clustering

14.1 K-Means Objective

$$\min \sum_i \|\mathbf{x}_i - c_{z_i}\|^2$$

14.2 Centroid Update

$$c_k = \frac{1}{|C_k|} \sum_{x \in C_k} x$$

14.3 Hierarchical Clustering

- Agglomerative
- Divisive

Linkage:

- Single linkage
- Complete linkage
- Average linkage

15 Classification

15.1 Naive Bayes

$$P(c|d) \propto P(c) \prod_{w \in d} P(w|c)$$

15.2 Logistic Regression

$$P(c = 1|x) = \sigma(w^T x)$$

15.3 Support Vector Machine

$$\min \frac{1}{2}\|w\|^2 + C \sum_i \xi_i$$

16 Word Embeddings

16.1 Skip-Gram (word2vec)

$$\max \sum_{(w,c)} \log P(c|w)$$

16.2 GloVe

$$J = \sum_{i,j} f(X_{ij})(w_i^T w_j + b_i + b_j - \log X_{ij})^2$$

17 Neural IR

- Siamese networks
- Bi-encoders
- Cross-encoders
- BERT-based rerankers

18 Recommender Systems

18.1 Matrix Factorization

$$R_{ij} \approx p_i^T q_j$$

18.2 Objective

$$\min_{p,q} \sum_{(i,j)\in\Omega} (R_{ij} - p_i^T q_j)^2 + \lambda(\|p_i\|^2 + \|q_j\|^2)$$

19 Graph-Based IR

- Personalized PageRank
- TextRank for summarization
- Graph embeddings

20 IR System Pipeline

1. Preprocessing
2. Tokenization
3. Stopword removal
4. Stemming/Lemmatization

5. Vectorization (TF-IDF or embeddings)
6. Indexing
7. Ranking
8. Evaluation

🎯 DIVERSE & FAIR RANKING CHEATSHEET

CS-6001: Indexing and Retrieving Text and Graphs

📊 PART 1: FOUNDATIONAL CONCEPTS

What is Ranking?

```
Ranking = Ordering items by relevance/usefulness to a query
Process: Find candidates → Score → Order → Present top-k
```

Relevance Score

```
# Basic concept
Relevance_Score = f(query, document)

# Properties:
- Ordinal (order matters more than absolute values)
- Query-dependent (same doc, different queries → different scores)
- Context-sensitive (location, time, user history)
```

🔢 PART 2: CLASSICAL RELEVANCE SCORING

TF-IDF (Term Frequency - Inverse Document Frequency)

```
# Term Frequency
TF(term, doc) = count(term in doc) / total_words_in_doc

# Inverse Document Frequency
IDF(term) = log(total_documents / documents_containing_term)

# Combined Score
TF-IDF(term, doc) = TF(term, doc) × IDF(term)

# Multi-term query
Score(query, doc) = Σ TF-IDF(term, doc) for each term in query
```

Intuition:

- TF: More mentions → more relevant
- IDF: Rare terms → more informative
- Common words (the, is, a) → low IDF → low impact

- Specific terms (eigenvalue, mitochondria) → high IDF → high impact

Example:

```
Query: "machine learning"
Doc A: ML paper (mentions both 50 times each)
Doc B: News (mentions "machine" once in "machine shop")

Doc A: High TF × High IDF → HIGH SCORE
Doc B: Low TF × Medium IDF → LOW SCORE
```

BM25 (Best Matching 25)

```
# Full Formula
BM25(Q, D) = Σ IDF(qᵢ) × [f(qᵢ,D) × (k₁ + 1)] / [f(qᵢ,D) + k₁ × (1 - b + b ×
|D|/avgdl)]

# Where:
# Q = query
# D = document
# qᵢ = query term i
# f(qᵢ,D) = frequency of qᵢ in D
# |D| = length of document D
# avgdl = average document length
# k₁ = term frequency saturation parameter (typical: 1.2-2.0)
# b = length normalization parameter (typical: 0.75)

# Simplified components:
TF_component = (f × (k₁ + 1)) / (f + k₁ × doc_length_factor)
doc_length_factor = (1 - b + b × |D|/avgdl)
```

Key Improvements over TF-IDF:

1. **Saturation**: Diminishing returns (100 mentions ≠ 100x better than 1)
2. **Length Normalization**: Penalizes long documents appropriately

Parameter Guide:

```
k₁ = 1.2   (lower = more saturation, higher = less saturation)
k₁ = 2.0   (closer to TF-IDF behavior)

b = 0.75   (standard - moderate length normalization)
b = 1.0    (full length penalty)
b = 0.0    (no length penalty)
```

PART 3: LEARNING TO RANK (LTR)

Three Approaches

```
# 1. POINTWISE - Predict absolute relevance
def pointwise_approach():
    """
    Treats as regression/classification
    Loss: MSE or Cross-Entropy
    """
    for (query, doc) in training_data:
        predicted_score = model(features(query, doc))
        loss = (predicted_score - true_relevance)²
    return model

# 2. PAIRWISE - Compare document pairs
def pairwise_approach():
    """
    Learns relative order
    Algorithm: RankNet, RankSVM
    """
    for (doc_A, doc_B) where relevance(A) > relevance(B):
        score_A = model(features(query, doc_A))
        score_B = model(features(query, doc_B))
        loss = max(0, 1 - (score_A - score_B))  # Hinge loss
    return model

# 3. LISTWISE - Optimize entire ranking
def listwise_approach():
    """
    Optimizes full ranking list
    Algorithm: ListNet, ListMLE
    """
    for query with documents D:
        predicted_ranking = model(all_docs)
        true_ranking = ground_truth_order
        loss = CrossEntropy(predicted_ranking, true_ranking)
    return model
```

Comparison Table

Approach	Granularity	Complexity	Performance	Algorithm Examples
Pointwise	Single doc	Low	Good	Linear Regression, SVR
Pairwise	Doc pairs	Medium	Better	RankNet, RankSVM, LambdaRank
Listwise	Full list	High	Best	ListNet, ListMLE, LambdaMART

Common Features for LTR

```
features = {
    # Query-Document features
    'tf_idf_score': 0.85,
    'bm25_score': 12.3,
    'cosine_similarity': 0.72,
    'query_term_coverage': 0.8,  # % of query terms in doc

    # Document features
    'pagerank': 0.65,
    'document_length': 2500,
    'is_recent': True,
    'domain_authority': 0.88,

    # User features
    'past_clicks': 15,
    'dwell_time_avg': 45.2,  # seconds
    'location_match': True,

    # Statistical features
    'query_frequency': 10000,  # how often query appears
    'doc_click_rate': 0.12,
    'query_doc_click_count': 50
}
```

🚨 PART 4: PROBLEMS WITH PURE RELEVANCE

The Three Major Issues

```
1. FILTER BUBBLE
   Problem: Users only see what they already agree with
   Example: Political search → only see your viewpoint
   Consequence: Echo chambers, polarization

2. REDUNDANCY
   Problem: Top results all say the same thing
   Example: "flu symptoms" → 10 identical result sets
   Consequence: Wasted time, no new information

3. UNFAIRNESS
   Problem: Systematic discrimination against groups
   Example: "CEO" image search → 95% male
   Consequence: Reinforces stereotypes, compounds over time
```

Position Bias - The Exposure Problem

```
# Position determines visibility
position_exposure = {
```

```
    1:  100%,  # Everyone sees
    2:   85%,
    3:   60%,
    5:   40%,
    10:  10%,
    20:   1%   # Almost no one sees
}

# Small ranking difference → HUGE exposure difference
# Position 1 vs Position 10 = 10x visibility difference
```

PART 5: DIVERSITY IN RANKING

Why Diversity?

```
Ambiguous Query Example: "jaguar"
- Animal (biology)
- Car brand (automotive)
- Operating system (technology)
- Sports team (NFL)

Without diversity: All results about ONE interpretation
With diversity: Results covering MULTIPLE interpretations
```

Maximal Marginal Relevance (MMR)

```
def mmr_algorithm(query, documents, k, lambda_param=0.7):
    """
    Balance relevance and novelty

    Args:
        query: user query
        documents: candidate documents
        k: number of results to return
        lambda_param: 0-1, controls relevance vs diversity tradeoff
                    1.0 = pure relevance
                    0.0 = maximum diversity
                    0.7 = balanced (common default)
    """
    result_set = []
    remaining = documents.copy()

    while len(result_set) < k:
        best_score = -infinity
        best_doc = None

        for doc in remaining:
            # Relevance to query
```

```
            relevance = similarity(query, doc)

            # Redundancy with existing results
            if result_set:
                redundancy = max(similarity(doc, r) for r in result_set)
            else:
                redundancy = 0

            # MMR Score
            mmr_score = lambda_param * relevance - (1 - lambda_param) * redundancy

            if mmr_score > best_score:
                best_score = mmr_score
                best_doc = doc

        result_set.append(best_doc)
        remaining.remove(best_doc)

    return result_set
```

MMR Example Walkthrough:

```
Query: "apple"
Documents:
  D1: iPhone review (relevance: 0.9)
  D2: iPhone specs (relevance: 0.85)
  D3: Apple fruit nutrition (relevance: 0.8)
  D4: Apple company history (relevance: 0.75)

Lambda = 0.7 (70% relevance, 30% diversity)

STEP 1: result_set = []
  All docs scored on pure relevance
  Select D1 (0.9)
  result_set = [D1]

STEP 2: result_set = [D1]
  D2: MMR = 0.7×0.85 - 0.3×sim(D2,D1) = 0.7×0.85 - 0.3×0.95 = 0.31
  D3: MMR = 0.7×0.8 - 0.3×sim(D3,D1) = 0.7×0.8 - 0.3×0.1 = 0.53 ← BEST
  D4: MMR = 0.7×0.75 - 0.3×sim(D4,D1) = 0.7×0.75 - 0.3×0.6 = 0.35

  Select D3 (different topic despite lower relevance!)
  result_set = [D1, D3]

Result: Diverse ranking covers both interpretations
```

Intent-Aware Diversification

```
def intent_aware_ranking(query, documents, intents):
    """
    Explicitly model and cover query intents
    """
    # Step 1: Detect intents and probabilities
    intents = {
        'programming': 0.7,  # "python" → programming
        'animal': 0.2,       # "python" → snake
        'comedy': 0.1        # "python" → Monty Python
    }

    # Step 2: Intent coverage goal
    top_k = 10
    target_coverage = {
        'programming': 7,  # 70% of results
        'animal': 2,       # 20% of results
        'comedy': 1        # 10% of results
    }

    # Step 3: Select documents to maximize coverage
    result = select_docs_covering_intents(documents, target_coverage)

    return result

# Optimization objective
score = sum(P(intent_i) * coverage(intent_i, results) for each intent)
```

⚖️ PART 6: FAIRNESS IN RANKING

Key Concepts

```
# Protected Attributes
protected_attributes = [
    'gender',
    'race',
    'age',
    'religion',
    'sexual_orientation',
    'disability_status',
    'national_origin'
]

# Group Fairness vs Individual Fairness
"""
GROUP FAIRNESS:
- Groups should have proportional representation
- Example: 40% women in pool → 40% women in top-k

INDIVIDUAL FAIRNESS:
```

```
- Similar individuals → similar rankings
- Example: Two identical candidates (except gender) → similar positions
"""
```

Fairness Metrics

```
# 1. DEMOGRAPHIC PARITY
def demographic_parity(ranking, protected_group):
    """
    Groups should have equal probability of appearing in top-k
    """
    return P(in_top_k | group_A) == P(in_top_k | group_B)

# 2. EQUAL OPPORTUNITY
def equal_opportunity(ranking, protected_group):
    """
    Among qualified items, equal chance for each group
    """
    return P(top_k | qualified, group_A) == P(top_k | qualified, group_B)

# 3. EXPOSURE FAIRNESS
def exposure_fairness(ranking):
    """
    Exposure proportional to merit
    """
    ratio_A = exposure(group_A) / merit(group_A)
    ratio_B = exposure(group_B) / merit(group_B)
    return ratio_A == ratio_B

# 4. DISPARATE IMPACT
def disparate_impact(ranking):
    """
    80% rule from employment law
    """
    impact_ratio = selection_rate(protected) / selection_rate(non_protected)
    return impact_ratio >= 0.8  # If < 0.8, significant disparate impact
```

🛠 PART 7: FAIR RANKING ALGORITHMS

FA*IR Algorithm (Fair Top-k Ranking)

```
def fair_topk_ranking(documents, k, min_proportion_protected=0.3):
    """
    FA*IR: Fairness-Aware Top-k Ranking
    Ensures minimum proportion of protected group in every prefix

    Args:
        documents: list of (doc, relevance, is_protected)
```

```
        k: number of results
        min_proportion_protected: minimum % of protected items (e.g., 0.3 = 30%)
    """
    # Sort by relevance
    documents.sort(key=lambda x: x['relevance'], reverse=True)

    ranking = []
    protected_docs = [d for d in documents if d['is_protected']]
    non_protected_docs = [d for d in documents if not d['is_protected']]

    for position in range(1, k+1):
        # Calculate required minimum protected items up to this position
        required_protected = floor(position * min_proportion_protected)
        current_protected = sum(1 for d in ranking if d['is_protected'])

        # Check if fairness constraint is violated
        if current_protected < required_protected:
            # Must select from protected group
            if protected_docs:
                next_doc = protected_docs.pop(0)  # Highest relevance protected
            else:
                break  # No more protected docs available
        else:
            # Can select highest relevance overall
            if protected_docs and non_protected_docs:
                if protected_docs[0]['relevance'] > non_protected_docs[0]
['relevance']:
                    next_doc = protected_docs.pop(0)
                else:
                    next_doc = non_protected_docs.pop(0)
            elif protected_docs:
                next_doc = protected_docs.pop(0)
            elif non_protected_docs:
                next_doc = non_protected_docs.pop(0)
            else:
                break

        ranking.append(next_doc)

    return ranking

# Example usage
documents = [
    {'id': 1, 'relevance': 0.95, 'is_protected': False},
    {'id': 2, 'relevance': 0.93, 'is_protected': False},
    {'id': 3, 'relevance': 0.90, 'is_protected': True},
    {'id': 4, 'relevance': 0.88, 'is_protected': False},
    {'id': 5, 'relevance': 0.85, 'is_protected': True},
]

fair_ranking = fair_topk_ranking(documents, k=5, min_proportion_protected=0.4)

# Result ensures at least 40% protected in every prefix:
```

```
# Position 1-5: at least 2 protected items (40% of 5)
# Position 1-10: at least 4 protected items (40% of 10)
```

Fairness Constraint Visualization

```
Position  | Required Protected (30% rule) | Cumulative
----------|-------------------------------|------------
    1     |        0                      |     0
    2     |        0                      |     0
    3     |        0                      |     0
    4     |        1                      |    ≥1
    5     |        1                      |    ≥1
    6     |        1                      |    ≥1
    7     |        2                      |    ≥2
    8     |        2                      |    ≥2
    9     |        2                      |    ≥2
   10     |        3                      |    ≥3
```

Calibrated Fairness

```python
def calibrated_fair_ranking(documents, k, protected_group):
    """
    Within each relevance tier, maintain proportional group representation
    """
    # Step 1: Partition into relevance buckets
    high_rel = [d for d in documents if d['relevance'] >= 0.8]
    medium_rel = [d for d in documents if 0.5 <= d['relevance'] < 0.8]
    low_rel = [d for d in documents if d['relevance'] < 0.5]

    def get_group_proportions(bucket):
        protected = sum(1 for d in bucket if d['is_protected'])
        total = len(bucket)
        return protected / total if total > 0 else 0

    # Step 2: Compute proportions in each bucket
    proportions = {
        'high': get_group_proportions(high_rel),
        'medium': get_group_proportions(medium_rel),
        'low': get_group_proportions(low_rel)
    }

    # Step 3: Fill ranking maintaining proportions
    ranking = []
    for position in range(k):
        # Determine bucket for this position (top positions from high bucket)
        if position < len(high_rel):
            bucket = high_rel
            target_proportion = proportions['high']
```

```
        elif position < len(high_rel) + len(medium_rel):
            bucket = medium_rel
            target_proportion = proportions['medium']
        else:
            bucket = low_rel
            target_proportion = proportions['low']

        # Sample from bucket maintaining group proportion
        current_protected_ratio = sum(1 for d in ranking if d['is_protected']) /
len(ranking) if ranking else 0

        if current_protected_ratio < target_proportion:
            # Select protected item
            candidates = [d for d in bucket if d['is_protected'] and d not in
ranking]
        else:
            # Select non-protected item
            candidates = [d for d in bucket if not d['is_protected'] and d not in
ranking]

        if candidates:
            next_doc = max(candidates, key=lambda x: x['relevance'])
            ranking.append(next_doc)

    return ranking
```

📐 PART 8: EVALUATION METRICS

Relevance Metrics

NDCG (Normalized Discounted Cumulative Gain)

```
import math

def dcg_at_k(relevances, k):
    """
    Discounted Cumulative Gain

    Args:
        relevances: list of relevance scores at each position
        k: cutoff position

    Returns:
        DCG@k score
    """
    dcg = 0.0
    for i in range(min(k, len(relevances))):
        # Position i+1 (1-indexed), relevance at position i (0-indexed)
        rel = relevances[i]
        position = i + 1
```

```
        discount = math.log2(position + 1)
        gain = (2 ** rel) - 1
        dcg += gain / discount
    return dcg

def ndcg_at_k(predicted_relevances, ideal_relevances, k):
    """
    Normalized DCG

    Args:
        predicted_relevances: relevances in predicted ranking order
        ideal_relevances: relevances in ideal (best possible) order
        k: cutoff position

    Returns:
        NDCG@k score (0-1, where 1 is perfect)
    """
    dcg = dcg_at_k(predicted_relevances, k)
    idcg = dcg_at_k(ideal_relevances, k)

    if idcg == 0:
        return 0.0

    return dcg / idcg

# Example
predicted = [3, 2, 3, 0, 1, 2]  # Relevances at positions 1,2,3,4,5,6
ideal = [3, 3, 2, 2, 1, 0]      # Best possible order

ndcg_score = ndcg_at_k(predicted, ideal, k=6)
print(f"NDCG@6 = {ndcg_score:.3f}")  # Output: NDCG@6 = 0.923

# DCG Calculation Breakdown:
# DCG = (2³-1)/log₂(2) + (2²-1)/log₂(3) + (2³-1)/log₂(4) + ...
#     = 7/1.0 + 3/1.58 + 7/2.0 + 0/2.32 + 1/2.58 + 3/2.81
#     = 7.0 + 1.90 + 3.5 + 0 + 0.39 + 1.07
#     = 13.86
# IDCG = 15.02
# NDCG = 13.86 / 15.02 = 0.923
```

Mean Average Precision (MAP)

```
def average_precision(predicted, relevant_items):
    """
    Average Precision for single query

    Args:
        predicted: list of predicted items in rank order
        relevant_items: set of truly relevant items

    Returns:
```

```
        AP score
    """
    if not relevant_items:
        return 0.0

    score = 0.0
    num_relevant = 0

    for k, item in enumerate(predicted, 1):
        if item in relevant_items:
            num_relevant += 1
            precision_at_k = num_relevant / k
            score += precision_at_k

    return score / len(relevant_items)

def mean_average_precision(queries_predictions, queries_relevant):
    """
    MAP across multiple queries

    Args:
        queries_predictions: dict {query_id: [predicted_items]}
        queries_relevant: dict {query_id: set(relevant_items)}

    Returns:
        MAP score
    """
    aps = []
    for query_id in queries_predictions:
        predicted = queries_predictions[query_id]
        relevant = queries_relevant[query_id]
        ap = average_precision(predicted, relevant)
        aps.append(ap)

    return sum(aps) / len(aps) if aps else 0.0

# Example
predicted = ['doc1', 'doc5', 'doc3', 'doc8', 'doc2']
relevant = {'doc1', 'doc3', 'doc2'}

# AP calculation:
# Position 1: doc1 is relevant, P@1 = 1/1 = 1.0
# Position 2: doc5 not relevant
# Position 3: doc3 is relevant, P@3 = 2/3 = 0.667
# Position 4: doc8 not relevant
# Position 5: doc2 is relevant, P@5 = 3/5 = 0.6
# AP = (1.0 + 0.667 + 0.6) / 3 = 0.756
```

Diversity Metrics

α-NDCG (Intent-Aware NDCG)

```
def alpha_ndcg(ranking, intents, alpha=0.5):
    """
    α-NDCG: Penalizes redundant coverage of same intent

    Args:
        ranking: list of (doc, intents_covered) tuples
        intents: total number of intents
        alpha: diversity parameter (0=pure diversity, 1=pure relevance)

    Returns:
        α-NDCG score
    """
    score = 0.0
    intent_coverage = {i: 0 for i in range(intents)}

    for position, (doc, doc_intents) in enumerate(ranking, 1):
        position_discount = math.log2(position + 1)

        for intent in doc_intents:
            # Gain for covering this intent
            gain = (1 - alpha) ** intent_coverage[intent]
            score += gain / position_discount
            intent_coverage[intent] += 1

    return score

# Example
ranking = [
    ('doc1', [0, 1]),    # Covers intents 0 and 1
    ('doc2', [0]),       # Covers intent 0 (redundant!)
    ('doc3', [2]),       # Covers intent 2 (new!)
]

score = alpha_ndcg(ranking, intents=3, alpha=0.5)
# Intent 0: covered at positions 1 and 2 (second coverage discounted)
# Intent 1: covered at position 1
# Intent 2: covered at position 3
```

Subtopic Recall

```
def subtopic_recall(ranking, all_intents):
    """
    Simple: What fraction of intents are covered?

    Args:
        ranking: list of documents
        all_intents: set of all possible intents for query

    Returns:
        Subtopic-Recall@k score (0-1)
```

```
    """
    covered_intents = set()
    for doc in ranking:
        covered_intents.update(doc['intents'])

    return len(covered_intents) / len(all_intents)

# Example
all_intents = {0, 1, 2, 3, 4}  # 5 possible intents
ranking = [
    {'id': 'doc1', 'intents': {0, 1}},
    {'id': 'doc2', 'intents': {0}},
    {'id': 'doc3', 'intents': {2}},
]

recall = subtopic_recall(ranking, all_intents)
# Covered: {0, 1, 2} = 3 intents
# Total: 5 intents
# Recall = 3/5 = 0.6
```

Fairness Metrics

Exposure Metrics

```
def exposure_ratio(ranking, position_exposure_weights=None):
    """
    Compare exposure between protected and non-protected groups

    Args:
        ranking: list of documents with 'is_protected' attribute
        position_exposure_weights: dict {position: exposure_weight}

    Returns:
        Exposure ratio (ideally close to 1.0)
    """
    if position_exposure_weights is None:
        # Default: exponential decay
        position_exposure_weights = {
            i: 1.0 / math.log2(i + 1) for i in range(1, len(ranking) + 1)
        }

    protected_exposure = 0.0
    non_protected_exposure = 0.0

    for position, doc in enumerate(ranking, 1):
        exposure = position_exposure_weights[position]
        if doc['is_protected']:
            protected_exposure += exposure
        else:
            non_protected_exposure += exposure
```

```
    if non_protected_exposure == 0:
        return float('inf')

    return protected_exposure / non_protected_exposure

# Example
ranking = [
    {'id': 1, 'is_protected': False},  # Position 1
    {'id': 2, 'is_protected': True},   # Position 2
    {'id': 3, 'is_protected': False},  # Position 3
    {'id': 4, 'is_protected': True},   # Position 4
    {'id': 5, 'is_protected': False},  # Position 5
]

ratio = exposure_ratio(ranking)
# Protected: 1/log(3) + 1/log(5) = 0.63 + 0.43 = 1.06
# Non-protected: 1/log(2) + 1/log(4) + 1/log(6) = 1.0 + 0.5 + 0.39 = 1.89
# Ratio = 1.06 / 1.89 = 0.56 (unfair, should be closer to 1.0)
```

Disparate Impact

```
def disparate_impact_ratio(ranking, top_k):
    """
    80% rule from employment law

    Args:
        ranking: list of documents
        top_k: cutoff position

    Returns:
        Impact ratio (should be ≥ 0.8 for fairness)
    """
    top_k_docs = ranking[:top_k]

    protected_in_topk = sum(1 for d in top_k_docs if d['is_protected'])
    non_protected_in_topk = sum(1 for d in top_k_docs if not d['is_protected'])

    total_protected = sum(1 for d in ranking if d['is_protected'])
    total_non_protected = sum(1 for d in ranking if not d['is_protected'])

    selection_rate_protected = protected_in_topk / total_protected if
total_protected > 0 else 0
    selection_rate_non_protected = non_protected_in_topk / total_non_protected if
total_non_protected > 0 else 0

    if selection_rate_non_protected == 0:
        return float('inf')

    impact_ratio = selection_rate_protected / selection_rate_non_protected
```

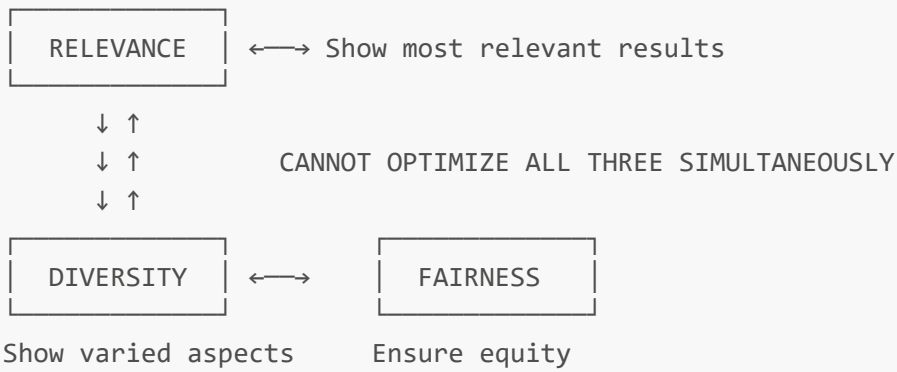
```
    return impact_ratio

# Example
ranking = [
    {'id': 1, 'is_protected': False},
    {'id': 2, 'is_protected': False},
    {'id': 3, 'is_protected': False},
    {'id': 4, 'is_protected': False},
    {'id': 5, 'is_protected': True},   # Only 1 protected in top 5
    {'id': 6, 'is_protected': True},
    {'id': 7, 'is_protected': True},
]

ratio = disparate_impact_ratio(ranking, top_k=5)
# Protected in top-5: 1 out of 3 total = 33.3%
# Non-protected in top-5: 4 out of 4 total = 100%
# Impact ratio = 0.333 / 1.0 = 0.33 < 0.8 → UNFAIR
```

⚡ PART 9: THE TRADEOFF - RELEVANCE vs DIVERSITY vs FAIRNESS

The Trilemma



Multi-Objective Optimization

```
def combined_ranking_score(ranking, query, alpha=0.5, beta=0.3, gamma=0.2):
    """
    Balance three objectives

    Args:
        ranking: candidate ranking
        query: user query
        alpha: relevance weight
        beta: diversity weight
        gamma: fairness weight
        (alpha + beta + gamma = 1.0)

    Returns:
```

```
        Combined score
    """
    # Compute individual scores
    relevance_score = ndcg_at_k(ranking, k=len(ranking))
    diversity_score = subtopic_recall(ranking, query['intents'])
    fairness_score = 1.0 - abs(0.5 - exposure_ratio(ranking))  # Penalize
deviation from 1.0

    # Weighted combination
    total_score = (
        alpha * relevance_score +
        beta * diversity_score +
        gamma * fairness_score
    )

    return total_score

# Different contexts require different weights:

# E-commerce (relevance-focused)
weights_ecommerce = {'alpha': 0.7, 'beta': 0.2, 'gamma': 0.1}

# News/Search (diversity-focused)
weights_news = {'alpha': 0.4, 'beta': 0.4, 'gamma': 0.2}

# Hiring/Lending (fairness-focused)
weights_hiring = {'alpha': 0.3, 'beta': 0.2, 'gamma': 0.5}
```

Practical Example: Job Search

```
# Scenario: Ranking job candidates

# RELEVANCE-ONLY
relevance_only = [
    {'name': 'A', 'score': 0.95, 'gender': 'M', 'education': 'Elite'},
    {'name': 'B', 'score': 0.93, 'gender': 'M', 'education': 'Elite'},
    {'name': 'C', 'score': 0.92, 'gender': 'M', 'education': 'Elite'},
    {'name': 'D', 'score': 0.90, 'gender': 'M', 'education': 'Elite'},
    {'name': 'E', 'score': 0.89, 'gender': 'M', 'education': 'Elite'},
]
# Issues: No diversity, no fairness, may reflect biased training data

# FAIR (30% women in top-5)
fair_ranking = [
    {'name': 'A', 'score': 0.95, 'gender': 'M', 'education': 'Elite'},
    {'name': 'F', 'score': 0.82, 'gender': 'F', 'education': 'State'},
    {'name': 'B', 'score': 0.93, 'gender': 'M', 'education': 'Elite'},
    {'name': 'C', 'score': 0.92, 'gender': 'M', 'education': 'Elite'},
    {'name': 'G', 'score': 0.80, 'gender': 'F', 'education': 'Regional'},
]
# Issues: Reduced avg relevance (0.884 vs 0.918), still not diverse
```

```
# DIVERSE + FAIR
diverse_fair = [
    {'name': 'A', 'score': 0.95, 'gender': 'M', 'education': 'Elite'},
    {'name': 'F', 'score': 0.82, 'gender': 'F', 'education': 'Bootcamp'},
    {'name': 'H', 'score': 0.88, 'gender': 'M', 'education': 'Self-taught'},
    {'name': 'I', 'score': 0.85, 'gender': 'F', 'education': 'Industry'},
    {'name': 'J', 'score': 0.78, 'gender': 'NB', 'education': 'Online'},
]
# Benefits: Gender diversity, education diversity, fair representation
# Cost: Lower avg relevance (0.856)

# THE TRADEOFF IS REAL AND UNAVOIDABLE
```

🔍 PART 10: ADVANCED TOPICS

Feedback Loops & Dynamic Fairness

```
"""
THE FEEDBACK LOOP PROBLEM:

Time T=0:
  System ranks women slightly lower (due to biased training data)

Time T=1:
  Women get less exposure (due to lower rankings)
  Women get fewer clicks (due to position bias, not relevance)

Time T=2:
  System learns "women items get fewer clicks"
  System ranks women even lower

Time T=3:
  Cycle continues and amplifies
  Gap widens over time

SOLUTION: Intervention strategies
"""

def feedback_aware_ranking(current_ranking, historical_data, fairness_constraint):
    """
    Account for feedback loop effects
    """
    # Debiased click model: separate position bias from true relevance
    for doc in current_ranking:
        observed_clicks = historical_data[doc['id']]['clicks']
        position_bias = historical_data[doc['id']]['avg_position_bias']

        # Debiased relevance
        true_relevance = observed_clicks / position_bias
```

```
        doc['debiased_score'] = true_relevance

    # Re-rank with debiased scores
    fair_ranking = apply_fairness_constraint(
        current_ranking,
        fairness_constraint,
        score_key='debiased_score'
    )

    return fair_ranking
```

Contextual Fairness

```
def context_specific_fairness(query, documents, context):
    """
    Fairness definition depends on context

    Different contexts require different fairness notions
    """
    if context == 'hiring':
        # Legal requirements: no discrimination
        fairness_constraint = {
            'type': 'demographic_parity',
            'protected_attributes': ['gender', 'race', 'age'],
            'min_proportion': 0.3
        }

    elif context == 'ecommerce':
        # Business fairness: small sellers get exposure
        fairness_constraint = {
            'type': 'exposure_fairness',
            'protected_group': 'small_sellers',
            'min_exposure_ratio': 0.2
        }

    elif context == 'news':
        # Information diversity: multiple viewpoints
        fairness_constraint = {
            'type': 'viewpoint_diversity',
            'political_spectrum': ['left', 'center', 'right'],
            'min_coverage': [0.25, 0.5, 0.25]
        }

    elif context == 'search':
        # Source diversity: different domains
        fairness_constraint = {
            'type': 'source_diversity',
            'max_same_domain': 3,  # At most 3 results from same domain
        }
```

```
    return apply_constraint(documents, fairness_constraint)
```

PART 11: QUICK REFERENCE TABLES

Algorithm Comparison

Algorithm	Optimizes	Complexity	Best For
TF-IDF	Relevance	O(n)	Simple text matching
BM25	Relevance	O(n)	Better relevance, standard baseline
MMR	Relevance + Diversity	$O(nk^2)$	Query ambiguity, redundancy reduction
FA*IR	Relevance + Fairness	O(nk)	Group fairness requirements
Intent-Aware	Relevance + Diversity	O(nki)	Multiple query interpretations
LTR (RankNet)	Relevance	$O(n^2)$ training	Large-scale, feature-rich

Metric Quick Reference

Metric	Measures	Range	Interpretation
NDCG@k	Relevance quality	[0, 1]	1.0 = perfect, higher better
MAP	Relevance (binary)	[0, 1]	Average precision, higher better
α-NDCG	Diversity + Relevance	[0, ∞)	Intent coverage, higher better
Subtopic Recall	Diversity	[0, 1]	Fraction of intents covered
Exposure Ratio	Fairness	[0, ∞)	1.0 = fair, deviation = unfair
Disparate Impact	Fairness	[0, ∞)	≥0.8 fair, <0.8 unfair
NDKL	Fairness	[0, ∞)	0 = fair, higher = more unfair

Parameter Tuning Guide

```
# BM25 Parameters
bm25_params = {
    'k1': {
        'default': 1.2,
        'range': [0.5, 3.0],
        'effect': 'Higher = less saturation (more like TF-IDF)'
    },
    'b': {
        'default': 0.75,
        'range': [0.0, 1.0],
        'effect': 'Higher = more length normalization'
```

```
    }
}

# MMR Parameters
mmr_params = {
    'lambda': {
        'default': 0.7,
        'range': [0.0, 1.0],
        'effect': '1.0 = pure relevance, 0.0 = pure diversity'
    }
}

# FA*IR Parameters
fair_params = {
    'min_proportion': {
        'default': 0.3,
        'range': [0.0, 1.0],
        'effect': 'Minimum fraction of protected group in results'
    }
}

# Multi-Objective Weights
weights = {
    'alpha': {  # Relevance
        'ecommerce': 0.7,
        'news': 0.4,
        'hiring': 0.3
    },
    'beta': {   # Diversity
        'ecommerce': 0.2,
        'news': 0.4,
        'hiring': 0.2
    },
    'gamma': {  # Fairness
        'ecommerce': 0.1,
        'news': 0.2,
        'hiring': 0.5
    }
}
```

🎓 PART 12: KEY TAKEAWAYS & EXAM TIPS

Core Concepts You MUST Know

```
1. RELEVANCE SCORING
   ✓ TF-IDF formula and intuition
   ✓ BM25 improvements (saturation, length normalization)
   ✓ Learning-to-Rank approaches (pointwise, pairwise, listwise)

2. DIVERSITY
```

```
   ✓ Why needed (filter bubbles, redundancy, query ambiguity)
   ✓ MMR algorithm (can you implement it?)
   ✓ Intent-aware diversification

3. FAIRNESS
   ✓ Group vs individual fairness
   ✓ Position bias and exposure
   ✓ FA*IR algorithm
   ✓ Fairness metrics (demographic parity, equal opportunity)

4. EVALUATION
   ✓ NDCG calculation (step-by-step)
   ✓ MAP for binary relevance
   ✓ α-NDCG for diversity
   ✓ Exposure ratio for fairness

5. TRADEOFFS
   ✓ Cannot optimize all three simultaneously
   ✓ Context determines weights
   ✓ Pareto frontier concept
```

Common Exam Questions

```
Q1: Calculate NDCG@k given ranking and relevances
    → Know the DCG formula
    → Don't forget to normalize by IDCG
    → Watch out for log base 2

Q2: Implement MMR algorithm
    → Remember: relevance - redundancy
    → Greedy selection (one at a time)
    → Lambda parameter controls tradeoff

Q3: Explain why pure relevance ranking is insufficient
    → Filter bubbles
    → Redundancy
    → Unfairness and discrimination
    → Position bias

Q4: Design fair ranking for specific scenario
    → Identify protected group
    → Choose fairness definition
    → Apply appropriate algorithm (FA*IR, calibrated, etc.)
    → Discuss tradeoffs

Q5: Compare TF-IDF vs BM25
    → Saturation (BM25 has it, TF-IDF doesn't)
    → Length normalization (BM25 better)
    → Parameters (BM25 has k1, b)

Q6: What is disparate impact?
```

```
    → 80% rule
    → Selection rate comparison
    → When to consider unfair
```

Formula Cheat Sheet

```
# MUST MEMORIZE THESE

# TF-IDF
TF-IDF(t,d) = (count(t,d) / |d|) × log(N / df(t))

# BM25
BM25(q,d) = Σ IDF(qᵢ) × [f(qᵢ,d)×(k₁+1)] / [f(qᵢ,d) + k₁×(1-b+b×|d|/avgdl)]

# DCG
DCG@k = Σᵢ₌₁ᵏ (2^rel(i) - 1) / log₂(i+1)

# NDCG
NDCG@k = DCG@k / IDCG@k

# MMR
MMR = λ × Relevance(q,d) - (1-λ) × max(Similarity(d,r) for r in Results)

# Exposure Ratio
Exposure_Ratio = Exposure(protected) / Exposure(non-protected)

# Disparate Impact
DI = min(Rate_A/Rate_B, Rate_B/Rate_A) ≥ 0.8
```

🚀 PART 13: PRACTICAL IMPLEMENTATION TIPS

Complete End-to-End System

```
class DiverseFairRankingSystem:
    """
    Production-ready ranking system
    """

    def __init__(self, config):
        self.relevance_model = self.load_model(config['relevance_model'])
        self.diversity_config = config['diversity']
        self.fairness_config = config['fairness']
        self.weights = config['weights']  # alpha, beta, gamma

    def rank(self, query, documents, k=10):
        """
        Main ranking pipeline
        """
```

```
        # Stage 1: Candidate Generation (fast, broad recall)
        candidates = self.retrieve_candidates(query, documents, top_n=1000)

        # Stage 2: Relevance Scoring (detailed, expensive)
        for doc in candidates:
            doc['relevance_score'] = self.relevance_model.score(query, doc)

        # Stage 3: Intent Detection (for diversity)
        intents = self.detect_intents(query)

        # Stage 4: Multi-Objective Optimization
        ranking = self.optimize_ranking(
            candidates=candidates,
            query=query,
            intents=intents,
            k=k,
            relevance_weight=self.weights['alpha'],
            diversity_weight=self.weights['beta'],
            fairness_weight=self.weights['gamma']
        )

        # Stage 5: Post-Processing (optional)
        ranking = self.apply_business_rules(ranking)

        return ranking[:k]

    def optimize_ranking(self, candidates, query, intents, k,
                         relevance_weight, diversity_weight, fairness_weight):
        """
        Greedy multi-objective optimization
        """
        ranking = []
        remaining = candidates.copy()

        # Track coverage for diversity
        intent_coverage = {intent: 0 for intent in intents}

        # Track fairness constraint
        protected_count = 0

        for position in range(1, k + 1):
            best_score = -float('inf')
            best_doc = None

            for doc in remaining:
                # Component 1: Relevance
                rel_score = doc['relevance_score']

                # Component 2: Diversity (MMR-style)
                if ranking:
                    redundancy = max(
                        self.similarity(doc, r) for r in ranking
                    )
                    # Intent coverage bonus
```

```
                    intent_bonus = sum(
                        1.0 / (1 + intent_coverage[i])
                        for i in doc['intents']
                    )
                    div_score = -redundancy + intent_bonus
                else:
                    div_score = 0

                # Component 3: Fairness
                required_protected = floor(position *
self.fairness_config['min_proportion'])
                if protected_count < required_protected and doc['is_protected']:
                    fair_score = 1.0  # Boost protected if needed
                elif protected_count >= required_protected and not
doc['is_protected']:
                    fair_score = 1.0  # Boost non-protected otherwise
                else:
                    fair_score = 0.5

                # Combined score
                total_score = (
                    relevance_weight * rel_score +
                    diversity_weight * div_score +
                    fairness_weight * fair_score
                )

                if total_score > best_score:
                    best_score = total_score
                    best_doc = doc

            # Add to ranking
            ranking.append(best_doc)
            remaining.remove(best_doc)

            # Update tracking
            for intent in best_doc['intents']:
                intent_coverage[intent] += 1
            if best_doc['is_protected']:
                protected_count += 1

        return ranking

    def evaluate(self, ranking, ground_truth, query):
        """
        Comprehensive evaluation
        """
        metrics = {}

        # Relevance
        metrics['ndcg@10'] = self.compute_ndcg(ranking, ground_truth, k=10)
        metrics['map'] = self.compute_map(ranking, ground_truth)

        # Diversity
        metrics['subtopic_recall'] = self.compute_subtopic_recall(
```

```
            ranking, query['intents']
        )
        metrics['alpha_ndcg'] = self.compute_alpha_ndcg(
            ranking, query['intents']
        )

        # Fairness
        metrics['exposure_ratio'] = self.compute_exposure_ratio(ranking)
        metrics['disparate_impact'] = self.compute_disparate_impact(ranking)

        return metrics
```

🎯 PART 14: REAL-WORLD SCENARIOS

Scenario-Based Problem Solving

```
# SCENARIO 1: Search Engine (Google, Bing)
search_config = {
    'weights': {'alpha': 0.5, 'beta': 0.3, 'gamma': 0.2},
    'diversity': {
        'method': 'intent_aware',
        'min_intents_covered': 3,
        'mmr_lambda': 0.7
    },
    'fairness': {
        'type': 'source_diversity',
        'max_same_domain': 2,
        'promote_small_sites': True
    }
}

# SCENARIO 2: E-commerce (Amazon, eBay)
ecommerce_config = {
    'weights': {'alpha': 0.7, 'beta': 0.2, 'gamma': 0.1},
    'diversity': {
        'method': 'category_diversity',
        'max_same_category': 3
    },
    'fairness': {
        'type': 'seller_fairness',
        'min_small_seller_proportion': 0.2
    }
}

# SCENARIO 3: Job Board (LinkedIn, Indeed)
hiring_config = {
    'weights': {'alpha': 0.3, 'beta': 0.2, 'gamma': 0.5},
    'diversity': {
        'method': 'skill_diversity',
        'balance_experience_levels': True
```

```
    },
    'fairness': {
        'type': 'demographic_parity',
        'protected_attributes': ['gender', 'race'],
        'min_proportion': 0.3,
        'audit_frequency': 'daily'
    }
}

# SCENARIO 4: News Feed (Twitter, Facebook)
social_config = {
    'weights': {'alpha': 0.4, 'beta': 0.4, 'gamma': 0.2},
    'diversity': {
        'method': 'viewpoint_diversity',
        'political_balance': [0.33, 0.34, 0.33],  # left, center, right
        'temporal_diversity': True
    },
    'fairness': {
        'type': 'creator_fairness',
        'exposure_variance_limit': 0.3,
        'small_creator_boost': 1.2
    }
}
```

PART 15: DEBUGGING & COMMON MISTAKES

Common Implementation Bugs

```
# MISTAKE 1: Forgetting to normalize scores
def wrong_combine():
    relevance = 0.9  # Range: [0, 1]
    diversity = 5.2  # Range: [0, 10]
    fairness = 0.3   # Range: [0, 1]

    # This is WRONG - scales are different!
    score = relevance + diversity + fairness

def correct_combine():
    # Normalize to same scale first
    relevance_norm = relevance / 1.0
    diversity_norm = diversity / 10.0
    fairness_norm = fairness / 1.0

    score = alpha * relevance_norm + beta * diversity_norm + gamma * fairness_norm

# MISTAKE 2: Not handling empty cases
def wrong_mmr(docs, results):
    redundancy = max(sim(d, r) for r in results)  # Crashes if results empty!

def correct_mmr(docs, results):
```

```
    redundancy = max(sim(d, r) for r in results) if results else 0.0

# MISTAKE 3: Off-by-one errors in NDCG
def wrong_dcg(relevances, k):
    dcg = 0
    for i in range(k):  # i is 0-indexed
        dcg += (2**relevances[i] - 1) / log2(i)  # WRONG: i=0 → log2(0) =
undefined!

def correct_dcg(relevances, k):
    dcg = 0
    for i in range(k):
        position = i + 1  # Convert to 1-indexed
        dcg += (2**relevances[i] - 1) / log2(position + 1)

# MISTAKE 4: Not considering fairness in every prefix
def wrong_fair_ranking():
    # Check fairness only at position k
    if count_protected(ranking[:k]) < min_protected:
        fix_fairness()

def correct_fair_ranking():
    # Check fairness at EVERY position
    for i in range(1, k+1):
        required = floor(i * min_proportion)
        actual = count_protected(ranking[:i])
        if actual < required:
            promote_protected_item()

# MISTAKE 5: Ignoring computational complexity
def wrong_diversity(docs, k):
    # Recompute all similarities every iteration - O(n²k)
    for position in range(k):
        for doc in docs:
            for result in results:
                sim = compute_similarity(doc, result)  # Expensive!

def correct_diversity(docs, k):
    # Precompute similarity matrix - O(n²) once
    sim_matrix = precompute_similarities(docs)
    for position in range(k):
        for doc in docs:
            redundancy = max(sim_matrix[doc][r] for r in results)
```

🏆 FINAL SUMMARY - THE ESSENTIALS

If You Remember Only 5 Things

```
1️⃣  RELEVANCE IS NOT ENOUGH
    Pure relevance → filter bubbles, redundancy, unfairness
```

```
    Modern systems need diversity + fairness

2️⃣  POSITION MATTERS ENORMOUSLY
    Position 1 vs 10 = 10x visibility difference
    Small ranking changes → huge exposure differences
    This is why fairness is critical

3️⃣  TRADEOFFS ARE UNAVOIDABLE
    Relevance ↔ Diversity ↔ Fairness
    Cannot maximize all three
    Context determines weights

4️⃣  ALGORITHMS TO KNOW
    • BM25 for relevance
    • MMR for diversity
    • FA*IR for fairness
    • NDCG for evaluation

5️⃣  FAIRNESS DEFINITIONS VARY
    Context matters: hiring ≠ e-commerce ≠ search
    No universal definition
    Must consider stakeholders
```

One-Liner Summaries

```
TF-IDF:      "Frequent in doc, rare overall = important"
BM25:        "TF-IDF with saturation and length normalization"
MMR:         "Relevance minus redundancy = diversity"
FA*IR:       "Fairness constraint in every prefix"
NDCG:        "Relevance quality with position discount"
α-NDCG:      "NDCG with penalty for redundant intent coverage"
LTR:         "Learn ranking function from data"
Exposure:    "Visibility determined by position"
Fairness:    "Equal opportunity or proportional representation"
Tradeoff:    "More of one means less of others"
```

🎓 GOOD LUCK ON YOUR EXAM!

Key Study Strategy:

1. ☑ Understand concepts deeply (don't just memorize)
2. ☑ Be able to implement algorithms (MMR, FA*IR, NDCG)
3. ☑ Calculate metrics by hand (NDCG, DCG, MAP)
4. ☑ Explain tradeoffs (when to use what)
5. ☑ Apply to scenarios (given context, design system)

Last-Minute Review:

- BM25 formula (with k_1 and b parameters)

- DCG/NDCG calculation (with $\log_2(i+1)$ discount)
- MMR algorithm (λ relevance - (1-λ) redundancy)
- FA*IR fairness constraint (floor(i × min_proportion))
- Why pure relevance fails (filter bubble, redundancy, unfairness)

Remember: This is an evolving field with no perfect answers. Understanding the principles matters more than memorizing formulas. Think critically about tradeoffs and real-world implications.

END OF CHEATSHEET *Created for CS-6001: Indexing and Retrieving Text and Graphs Topic: Diverse and Fair Ranking*

Learning to Rank (LTR) - Complete Cheatsheet

Course: CS-6001 - Indexing and Retrieving Text and Graphs
Topic: Learning to Rank for Information Retrieval
Author: Study Guide
Date: 2025-11-18

Table of Contents

Core Concepts

What is Learning to Rank?

Definition: Machine learning approach to automatically learn optimal ranking functions from training data.

Goal: Given query q and documents D = {d_1, d_2, ..., d_n}, produce optimal ordering.

Key Idea:

```
Traditional: Hand-craft ranking formula (BM25, TF-IDF)
LTR: Learn ranking function f(q, d) from data
```

Why LTR?

- [x] Combines hundreds of features automatically
- [x] Learns from user behavior (clicks, dwell time)
- [x] Optimizes directly for ranking metrics (NDCG, MAP)
- [x] Adapts to different domains
- [x] Continuously improves with more data

Problem Formulation

```
Input:  Query q, Documents D = {d₁, d₂, ..., dₙ}
        Features: x(q, dᵢ) ∈ ℝᵐ
        Labels: yᵢ ∈ {0, 1, 2, 3, 4} (relevance grades)
```

```
Output: Ranking function f: (q, d) → score
        Ranked list: π = [d_{π(1)}, d_{π(2)}, ..., d_{π(n)}]
```

🔀 Three Approaches

1️⃣ POINTWISE Approach

Philosophy: Treat as regression/classification for individual documents

Process:

```
Training:
  For each (query, doc) pair:
    Predict relevance score: ŷᵢ = f(x(q, dᵢ))
    Minimize: L(y, ŷ) = (y - ŷ)²

Testing:
  Score each document independently
  Sort by predicted scores
```

Algorithms: Linear Regression, Logistic Regression, Neural Networks

Pros ☑:

- Simple implementation
- Many existing ML algorithms work
- Fast training
- Easy to interpret

Cons ❌:

- Ignores relative ordering
- Treats documents independently
- Loss ≠ ranking metric
- Doesn't optimize for position

Example:

```
Query: "machine learning"
Doc A: score = 3.8 → Rank 2
Doc B: score = 2.1 → Rank 3
Doc C: score = 4.2 → Rank 1
Final: [C, A, B]
```

2️⃣ PAIRWISE Approach

Philosophy: Learn to compare pairs - which document is better?

Process:

```
Training:
  For each pair (dᵢ, dⱼ) where yᵢ > yⱼ:
    Learn: f(q, dᵢ) > f(q, dⱼ)
    Minimize: L = -log P(dᵢ > dⱼ)

Testing:
  Use learned function to compare pairs
  Sort documents by pairwise comparisons
```

Key Algorithms: RankNet, RankSVM, LambdaRank

Pros ☑:

- Models relative preferences
- Better than pointwise for ranking
- Handles inconsistent judgments
- More robust

Cons ❌:

- $O(n^2)$ pairs (quadratic complexity)
- Still doesn't directly optimize NDCG
- Assumes pairwise independence
- Can be slow for large n

Example:

```
Training pairs:
  (A, B) → A > B ✓
  (B, C) → C > B ✓
  (A, C) → A > C ✓
Learned order: A > C > B
```

---

3 LISTWISE Approach

Philosophy: Consider entire ranked list, optimize ranking metrics directly

Process:

```
Training:
  Input: Query q with all documents [d₁, ..., dₙ]
  Model probability of permutations
  Minimize: Distance between predicted and ideal ranking
```

```
  Loss: Based on NDCG, MAP, etc.

Testing:
  Score entire list jointly
  Output optimal permutation
```

Key Algorithms: ListNet, AdaRank, LambdaMART

Pros ☑:

- Directly optimizes ranking metrics
- Considers full list context
- Theoretically optimal
- State-of-the-art performance

Cons ❌:

- Computationally expensive (n! permutations)
- Complex implementation
- Requires approximations
- Slower training

Example:

```
Input: [A, B, C, D, E] with relevances [3, 1, 4, 0, 2]
ListNet learns:
  P(C,A,E,B,D) = 0.65  (ideal)
  P(A,C,E,B,D) = 0.20
  P(other orders) = 0.15
Outputs most probable ranking
```

🤖 Key Algorithms

RankNet (Pairwise)

```
Architecture: Neural Network
Loss: Cross-entropy on pairs

P(dᵢ > dⱼ) = 1 / (1 + e^(-(sᵢ - sⱼ)))

Loss = -Σ log P(dᵢ > dⱼ)

where sᵢ = f(x(q, dᵢ)) is the score from neural net
```

Key Features:

- Smooth, differentiable loss

- Backpropagation through pairs
- Can use graded relevance
- Popular in production systems

RankSVM (Pairwise)

```
Objective: Maximize margin between relevant/irrelevant

Minimize: ½||w||² + C Σ ξᵢⱼ

Subject to:
  w·(φ(q,dᵢ) - φ(q,dⱼ)) ≥ 1 - ξᵢⱼ
  ξᵢⱼ ≥ 0
  for all pairs where yᵢ > yⱼ
```

Key Features:

- SVM-based approach
- Kernel trick applicable
- Convex optimization
- Strong theoretical guarantees

ListNet (Listwise)

```
Model: Probability distribution over permutations

P(π | s) = Π[i=1 to n] φ(s_{π(i)}) / Σ[j=i to n] φ(s_{π(j)})

where φ(x) = e^x (softmax)

Loss: Cross-entropy between:
  - Distribution from ground truth
  - Distribution from model predictions

L = -Σ P_y(π) log P_s(π)
```

Key Features:

- Top-k approximation for efficiency
- Permutation probability
- End-to-end differentiable
- Production-ready

AdaRank (Listwise)

```
Boosting approach:

1. Initialize weights: w_m(q_i) = 1/M for all queries
2. For round t = 1 to T:
   a. Train weak ranker h_t on weighted data
   b. Compute performance: P(q_i) on metric (MAP/NDCG)
   c. Update weights:
      w_{t+1}(q_i) = w_t(q_i) · e^(-α·P(q_i))
      (increase weight on poor queries)
   d. α_t = ½ log((1-ε)/ε) where ε is error
3. Final: H(x) = Σ α_t·h_t(x)
```

Key Features:

- Focuses on hard queries
- Directly optimizes ranking metric
- Ensemble of weak rankers
- Adaptive weighting

LambdaMART (Listwise)

```
Combines: LambdaRank + MART (Multiple Additive Regression Trees)

Key Innovation:
  Gradient ∂C/∂s_i → λ_i (lambda)
  λ_ij = -∂C/∂s_i |Δ_NDCG|

where |Δ_NDCG| = |NDCG(swap i,j) - NDCG(current)|

Trains gradient boosted decision trees
Each tree learns to improve ranking metric
```

Key Features:

- State-of-the-art performance
- Used in Bing, Yahoo
- Handles interactions automatically
- Robust to noisy data

🎛️ Features in LTR

1. Query-Document Features

```
# Text matching features
BM25_score(q, d)
```

```
TF_IDF(q, d)
Language_Model_score(q, d)
Cosine_similarity(q, d)

# Term statistics
num_query_terms_matched
query_coverage = matched_terms / total_query_terms
term_proximity = avg_distance_between_query_terms

# Position features
first_position_of_query_term
last_position_of_query_term
span_of_query_terms
```

2. Query Features

```
query_length
query_term_frequencies
query_type: {navigational, informational, transactional}
is_popular_query
query_click_entropy
query_reformulation_rate
```

3. Document Features

```
# Web-specific
PageRank(d)
HITS_authority(d)
HITS_hub(d)

# Content
document_length
title_length
URL_length
URL_depth
num_slashes_in_URL
has_www_prefix

# Quality signals
domain_authority
is_spam_score
content_freshness
last_update_date
```

4. Query-Independent Features

```
# Popularity
click_through_rate (CTR)
conversion_rate
average_dwell_time
bounce_rate

# Link analysis
num_inbound_links
num_outbound_links
anchor_text_quality

# User behavior
bookmarked_count
shared_count
comments_count
```

5. Combined/Derived Features

```
# Ratios
BM25 / max_BM25_in_result_set
normalized_PageRank

# Interactions
BM25 * PageRank
query_length * document_length

# Statistical
percentile_rank_of_feature
z_score_normalization
```

📊 Evaluation Metrics

Precision @ K

```
P@K = (# relevant docs in top K) / K

Example:
Top 5: [Rel, NotRel, Rel, Rel, NotRel]
P@5 = 3/5 = 0.60
P@3 = 2/3 = 0.67
```

Mean Average Precision (MAP)

```
AP = (1/R) Σ P@k · rel(k)

where:
  R = total relevant docs
  P@k = precision at position k
  rel(k) = 1 if doc at k is relevant, 0 otherwise

Example:
Ranking: [R, N, R, N, R, N, R]
         ↓     ↓     ↓     ↓
AP = (1/4) · (1/1 + 2/3 + 3/5 + 4/7)
   = (1/4) · (1.0 + 0.67 + 0.60 + 0.57)
   = 0.71

MAP = Average of AP over all queries
```

Normalized Discounted Cumulative Gain (NDCG)

⭐ MOST IMPORTANT METRIC ⭐

```
DCG@K = Σ[i=1 to K] (2^rel_i - 1) / log₂(i + 1)

NDCG@K = DCG@K / IDCG@K

where IDCG@K = DCG of ideal ranking
```

Step-by-Step Example:

```
Ranking with grades:
Position | Relevance | Gain Calculation
---------|-----------|------------------
1        | 3         | (2³-1)/log₂(2) = 7/1.00 = 7.00
2        | 2         | (2²-1)/log₂(3) = 3/1.58 = 1.89
3        | 3         | (2³-1)/log₂(4) = 7/2.00 = 3.50
4        | 0         | (2⁰-1)/log₂(5) = 0/2.32 = 0.00
5        | 1         | (2¹-1)/log₂(6) = 1/2.58 = 0.39

DCG@5 = 7.00 + 1.89 + 3.50 + 0.00 + 0.39 = 12.78

Ideal ranking (3,3,2,1,0):
IDCG@5 = 7.00 + 3.50 + 1.89 + 0.39 + 0.00 = 12.78

NDCG@5 = 12.78 / 12.78 = 1.00 (perfect!)
```

Why NDCG?: ☑ Handles graded relevance (not just binary)
☑ Position-aware (top results matter most)

- [x] Normalized [0,1] for comparison
- [x] Standard in research and industry

Mean Reciprocal Rank (MRR)

```
RR = 1 / rank of first relevant document

Example:
Query 1: First relevant at position 3 → RR = 1/3
Query 2: First relevant at position 1 → RR = 1/1
Query 3: First relevant at position 2 → RR = 1/2

MRR = (1/3 + 1/1 + 1/2) / 3 = 0.61
```

Use case: Navigational queries (one correct answer)

Expected Reciprocal Rank (ERR)

```
ERR = Σ[r=1 to n] (1/r) · P(user stops at position r)

P(stop at r) = Rᵣ · Π[i=1 to r-1] (1 - Rᵢ)

where Rᵢ = (2^relᵢ - 1) / 2^max_rel

Models cascade: user scans from top, stops when satisfied
```

🧮 Mathematical Formulas

Loss Functions

Pointwise Loss (Regression)

```
L = Σ (yᵢ - f(xᵢ))²          # Mean Squared Error
L = Σ |yᵢ - f(xᵢ)|            # Mean Absolute Error
```

Pairwise Loss (RankNet)

```
L = Σ log(1 + e^(-σ(sᵢ - sⱼ)))

where σ is scaling parameter
```

Listwise Loss (ListNet)

```
L = -Σ P(π|y) log P(π|s)

Cross-entropy between label and prediction distributions
```

Optimization

Gradient Descent

```
w ← w - η · ∇L(w)

where η is learning rate
```

LambdaRank Gradient

```
λᵢ = Σⱼ (∂C/∂sᵢ) · |Δ_NDCG_ij|

Gradient scaled by change in NDCG from swapping i and j
```

Feature Normalization

```
Min-Max:     x' = (x - min) / (max - min)
Z-score:     x' = (x - μ) / σ
Log:         x' = log(1 + x)
Rank:        x' = rank(x) / n
```

🚀 Quick Reference

Approach Comparison Table

Aspect	Pointwise	Pairwise	Listwise
Granularity	Individual docs	Doc pairs	Full list
Complexity	O(n)	$O(n^2)$	O(n!)
Loss	MSE, Cross-entropy	Pairwise ranking	NDCG/MAP
Optimization	Standard ML	Pairwise preference	Direct ranking metric

Aspect	Pointwise	Pairwise	Listwise
Examples	Linear Reg, NN	RankNet, RankSVM	ListNet, LambdaMART
Speed	⚡⚡⚡ Fast	⚡⚡ Medium	⚡ Slow
Accuracy	★★ Good	★★★ Better	★★★★ Best
Use Case	Simple tasks	Medium datasets	Production systems

Algorithm Selection Guide

```
Choose Pointwise if:
  ✓ Simple baseline needed
  ✓ Fast training required
  ✓ Binary relevance (relevant/not)
  ✓ Small feature set

Choose Pairwise if:
  ✓ Need better than baseline
  ✓ Have preference judgments
  ✓ Moderate dataset size
  ✓ Care about relative ordering

Choose Listwise if:
  ✓ Need state-of-the-art
  ✓ Have graded relevance
  ✓ Large training data
  ✓ Can afford computation
  ✓ Production deployment
```

Common Pitfalls & Solutions

Problem: Overfitting

```
Symptoms:
  - Training metric ↑, Validation metric ↓
  - Perfect scores on training data

Solutions:
  ✓ Regularization (L1/L2): L = Loss + λ||w||²
  ✓ Early stopping
  ✓ Cross-validation (query-level splits)
  ✓ Feature selection
  ✓ More training data
```

Problem: Position Bias in Click Data

```
Symptoms:
  - Top positions over-represented
  - Users click top results regardless of relevance

Solutions:
  ✓ Inverse Propensity Scoring
  ✓ Randomized experiments
  ✓ Regression discontinuity
  ✓ Examination model
```

Problem: Cold Start

```
Symptoms:
  - No training data for new queries
  - Poor performance on rare queries

Solutions:
  ✓ Query expansion
  ✓ Transfer learning
  ✓ Fallback to BM25
  ✓ Meta-learning
```

Implementation Checklist

Data Preparation

```
☐ Collect query-document pairs
☐ Obtain relevance labels (0-4 scale)
☐ Extract features (query, doc, query-doc)
☐ Normalize features
☐ Split data: train/validation/test (query-level)
☐ Handle missing values
☐ Remove outliers
```

Training

```
☐ Choose approach (pointwise/pairwise/listwise)
☐ Select algorithm (RankNet/LambdaMART/etc.)
☐ Set hyperparameters
☐ Implement loss function
☐ Add regularization
☐ Monitor training/validation metrics
```

```
☐ Early stopping criterion
☐ Save best model
```

Evaluation

```
☐ Compute NDCG@k (k=1,3,5,10)
☐ Compute MAP
☐ Compute MRR
☐ Statistical significance testing
☐ Per-query analysis
☐ Error analysis
☐ Feature importance analysis
```

Deployment

```
☐ Optimize inference speed
☐ Feature caching
☐ Two-stage ranking (retrieve + rerank)
☐ A/B testing framework
☐ Online monitoring
☐ Feedback collection
☐ Model retraining pipeline
```

Advanced Topics

Deep Learning for Ranking

Neural Architectures

```
DSSM (Deep Semantic Similarity Model):
  - Learn semantic embeddings
  - Query and doc through separate NNs
  - Cosine similarity in latent space

BERT-based Rankers:
  - Cross-encoder: [CLS] query [SEP] doc [SEP]
  - Bi-encoder: Separate encodings
  - ColBERT: Late interaction

Architecture:
  Input → Embedding → Transformer → Pooling → Score
```

Advantages

- Learn from raw text
- Capture semantic similarity
- Transfer learning
- State-of-the-art results

Unbiased Learning to Rank

Position Bias Correction

```
Inverse Propensity Scoring:

Unbiased estimate:
  R̂ = Σ (oᵢ · rᵢ) / P(examine position i)

where:
  oᵢ = click indicator (0/1)
  rᵢ = true relevance
  P(examine position i) = propensity

Estimate propensities via:
  - Randomized experiments
  - Click models (PBM, UBM, DBN)
  - Intervention harvesting
```

Multi-Objective Ranking

```
Optimize multiple objectives:
  - Relevance
  - Diversity
  - Freshness
  - Personalization
  - Fairness

Approaches:
  1. Linear combination: L = α·L_rel + β·L_div + γ·L_fresh
  2. Pareto optimization
  3. Constrained optimization
  4. Multi-task learning
```

Code Templates

Python Skeleton - Pointwise

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import numpy as np

# Load data
X_train = load_features()  # Shape: (n_samples, n_features)
y_train = load_labels()    # Shape: (n_samples,)

# Train model
model = LinearRegression()
model.fit(X_train, y_train)

# Predict
y_pred = model.predict(X_test)

# Rank documents
ranked_indices = np.argsort(-y_pred)  # Descending order
```

Python Skeleton - Pairwise (RankNet)

```
import torch
import torch.nn as nn

class RankNet(nn.Module):
    def __init__(self, input_dim):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)

    def pairwise_loss(self, s_i, s_j, label):
        # label = 1 if i > j, else -1
        return torch.log(1 + torch.exp(-label * (s_i - s_j)))

# Training loop
model = RankNet(input_dim=100)
optimizer = torch.optim.Adam(model.parameters())

for epoch in range(epochs):
    for x_i, x_j, label in pair_loader:
        s_i = model(x_i)
        s_j = model(x_j)
        loss = model.pairwise_loss(s_i, s_j, label)
```

```
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Python Skeleton - Listwise (ListNet)

```
import torch
import torch.nn.functional as F

def listnet_loss(y_pred, y_true):
    """
    y_pred: predicted scores [batch_size, list_size]
    y_true: ground truth relevance [batch_size, list_size]
    """
    # Convert to probability distributions
    pred_probs = F.softmax(y_pred, dim=1)
    true_probs = F.softmax(y_true, dim=1)

    # Cross-entropy loss
    loss = -torch.sum(true_probs * torch.log(pred_probs + 1e-10))
    return loss

# Usage
model = RankingNN(input_dim=100, hidden_dim=64)
optimizer = torch.optim.Adam(model.parameters())

for epoch in range(epochs):
    for features, labels in data_loader:
        # features: [batch, docs_per_query, feature_dim]
        # labels: [batch, docs_per_query]

        scores = model(features)  # [batch, docs_per_query]
        loss = listnet_loss(scores, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

NDCG Calculation

```
import numpy as np

def dcg(relevances, k=None):
    """Discounted Cumulative Gain"""
    if k is None:
        k = len(relevances)
    relevances = np.asarray(relevances)[:k]
```

```
    gains = 2**relevances - 1
    discounts = np.log2(np.arange(2, len(relevances) + 2))
    return np.sum(gains / discounts)

def ndcg(relevances, k=None):
    """Normalized DCG"""
    dcg_score = dcg(relevances, k)
    ideal_relevances = sorted(relevances, reverse=True)
    idcg_score = dcg(ideal_relevances, k)
    return dcg_score / idcg_score if idcg_score > 0 else 0.0

# Example
ranked_relevances = [3, 2, 3, 0, 1]  # From ranking
ndcg_5 = ndcg(ranked_relevances, k=5)
print(f"NDCG@5: {ndcg_5:.4f}")
```

Feature Extraction Example

```
def extract_features(query, document):
    """Extract features for (query, document) pair"""
    features = {}

    # Query-document features
    features['bm25'] = compute_bm25(query, document)
    features['tfidf'] = compute_tfidf(query, document)
    features['query_coverage'] = len(set(query) & set(document)) / len(query)
    features['term_proximity'] = compute_proximity(query, document)

    # Document features
    features['doc_length'] = len(document)
    features['pagerank'] = document.pagerank
    features['url_length'] = len(document.url)

    # Query-independent features
    features['ctr'] = document.click_through_rate
    features['dwell_time'] = document.avg_dwell_time

    return list(features.values())

# Usage
query_tokens = tokenize("machine learning tutorial")
doc = get_document(doc_id)
feature_vector = extract_features(query_tokens, doc)
```

Exam Prep - Key Points to Remember

Must-Know Concepts

1. **Three approaches**: Pointwise, Pairwise, Listwise
2. **NDCG formula** and calculation
3. **RankNet** (pairwise NN approach)
4. **LambdaMART** (state-of-the-art)
5. **Feature types**: query, doc, query-doc, query-independent
6. **Position bias** in click data
7. **Evaluation metrics**: MAP, NDCG, MRR
8. **Trade-offs** between approaches

Common Exam Questions

Q1: Compare pointwise vs pairwise vs listwise approaches.

```
Answer Framework:
- Definition of each
- How training works
- Loss functions
- Pros/cons
- Example algorithms
- When to use each
```

Q2: Calculate NDCG@5 for given ranking.

```
Steps:
1. Apply formula: (2^rel - 1) / log₂(pos + 1)
2. Sum for positions 1-5 (DCG)
3. Sort by relevance (get ideal)
4. Calculate IDCG same way
5. NDCG = DCG / IDCG
```

Q3: Explain how RankNet works.

```
Key points:
- Pairwise neural network
- Learns P(doc_i > doc_j)
- Cross-entropy loss on pairs
- Sigmoid activation for probability
- Backpropagation through pairs
```

Q4: What features would you use for web search ranking?

```
Categories:
- Text matching (BM25, TF-IDF)
- Link analysis (PageRank)
```

```
- User behavior (CTR, dwell time)
- Quality signals (spam score)
- Query features (length, type)
```

Q5: How to handle position bias in click data?

```
Solutions:
- Inverse propensity scoring
- Randomized experiments
- Click models (examination probability)
- Interleaving experiments
- Counterfactual learning
```

Resources & Tools

Libraries & Frameworks

```
Python:
  - LightGBM: Fast gradient boosting (LambdaMART)
  - XGBoost: Gradient boosting with ranking objective
  - RankLib: Java library for LTR
  - TF-Ranking: TensorFlow library
  - PyTorch: For neural ranking models
  - scikit-learn: Basic ML algorithms

Datasets:
  - LETOR: Benchmark LTR datasets
  - Microsoft MSLR-WEB: Web search ranking
  - Yahoo! LTR Challenge: Large-scale dataset
  - Istella: Italian search engine data
```

Evaluation Tools

```
# Using sklearn for ranking metrics
from sklearn.metrics import ndcg_score

y_true = [[3, 2, 3, 0, 1]]  # True relevances
y_score = [[0.8, 0.4, 0.9, 0.1, 0.3]]  # Predicted scores

ndcg = ndcg_score(y_true, y_score, k=5)
```

Summary Cheat Sheet (1-Page)

```
┌──────────────────────────────────────────────────────────┐
│              LEARNING TO RANK CHEAT SHEET                │
├──────────────────────────────────────────────────────────┤
│ APPROACHES:                                              │
│  Pointwise → Pairwise → Listwise (increasing performance)│
│                                                          │
│ KEY ALGORITHMS:                                          │
│  RankNet, RankSVM, ListNet, AdaRank, LambdaMART          │
│                                                          │
│ MAIN METRIC:                                             │
│  NDCG@K = DCG@K / IDCG@K                                 │
│  DCG = Σ (2^rel - 1) / log₂(pos + 1)                     │
│                                                          │
│ FEATURES (4 types):                                      │
│  1. Query-Doc: BM25, TF-IDF, term proximity              │
│  2. Query: length, type, popularity                      │
│  3. Document: PageRank, length, freshness                │
│  4. Query-independent: CTR, dwell time, links            │
│                                                          │
│ EVALUATION METRICS:                                      │
│  NDCG@K, MAP, MRR, P@K, ERR                              │
│                                                          │
│ CHALLENGES:                                              │
│  Position bias, overfitting, scalability, cold start     │
│                                                          │
│ SOLUTIONS:                                               │
│  IPS, regularization, two-stage ranking, transfer learning│
└──────────────────────────────────────────────────────────┘
```

🎓 Final Tips for Success

For Exams:

- ☑ Memorize NDCG formula and practice calculations
- ☑ Know trade-offs between three approaches
- ☑ Understand RankNet and LambdaMART
- ☑ Be able to list and categorize features
- ☑ Practice explaining concepts simply

For Projects:

- ☑ Start with simple baseline (BM25)
- ☑ Try pointwise approach first
- ☑ Use LightGBM or XGBoost for pairwise/listwise
- ☑ Focus on feature engineering
- ☑ Always use query-level cross-validation
- ☑ Report NDCG@k for multiple k values

For Interviews:

- [x] Explain ranking problem clearly
- [x] Discuss production considerations (latency, scalability)
- [x] Mention A/B testing and online metrics
- [x] Know about position bias and counterfactual learning
- [x] Be familiar with neural ranking (BERT-based)

END OF CHEATSHEET

Good luck with your studies! 🚀

🎯 RANKING EVALUATION CHEATSHEET

Complete Reference for Information Retrieval Evaluation Metrics

---

📊 FUNDAMENTAL CONCEPTS

Core Terminology

Term	Definition	Example
Query	User's information need	"machine learning algorithms"
Document	Retrievable item	Web page, paper, email
Relevance	Document satisfies query	Binary (0/1) or graded (0-3)
Ranking	Ordered list of results	[doc5, doc2, doc9, ...]
Retrieved Set	Documents returned by system	All docs with score > threshold
Relevant Set	Ground truth relevant docs	Determined by human judges

Confusion Matrix for IR

```
                    Actually Relevant    Actually Not Relevant
Retrieved           True Positive (TP)   False Positive (FP)
Not Retrieved       False Negative (FN)  True Negative (TN)
```

🔢 BINARY RELEVANCE METRICS

1. PRECISION

Definition: Fraction of retrieved documents that are relevant
Formula: `Precision = TP / (TP + FP)`
Range: [0, 1] (higher is better)
When to use: When false positives are costly

Example:

```
Retrieved: 20 documents
Relevant in retrieved: 15
Precision = 15/20 = 0.75
```

Interpretation: 75% of results are relevant

---

2. RECALL

Definition: Fraction of relevant documents that are retrieved
Formula: `Recall = TP / (TP + FN)`
Range: [0, 1] (higher is better)
When to use: When missing relevant docs is costly

Example:

```
Total relevant in collection: 30
Retrieved relevant: 15
Recall = 15/30 = 0.50
```

Interpretation: Found 50% of all relevant documents

---

3. F-MEASURE (F1 SCORE)

Definition: Harmonic mean of precision and recall
Formula: `F1 = 2PR / (P + R)`
Alternative: `F1 = 2TP / (2TP + FP + FN)`
Range: [0, 1] (higher is better)

General F-Measure:

```
F_β = (1 + β²) × PR / (β²P + R)

β = 1: Equal weight to P and R (standard F1)
β > 1: Favor recall (e.g., β=2)
β < 1: Favor precision (e.g., β=0.5)
```

Example:

```
P = 0.75, R = 0.50
F1 = 2 × 0.75 × 0.50 / (0.75 + 0.50)
   = 0.75 / 1.25 = 0.60

F2 = 5 × 0.75 × 0.50 / (4×0.75 + 0.50)
   = 1.875 / 3.50 = 0.536 (closer to recall)
```

---

4. ACCURACY

Formula: `Accuracy = (TP + TN) / (TP + TN + FP + FN)`
⚠ **WARNING**: Misleading in IR! (Most documents are not relevant → high TN)

RANK-AWARE METRICS (BINARY RELEVANCE)

5. PRECISION AT K (P@K)

Definition: Precision considering only top K results
Formula: `P@K = (# relevant in top K) / K`
Common values: K = 5, 10, 20

Example:

```
Top 10 results: [R, R, N, R, N, N, R, N, R, N]
P@10 = 5/10 = 0.50
P@5 = 3/5 = 0.60
```

Why it matters: Users rarely see beyond first page

---

6. AVERAGE PRECISION (AP)

Definition: Average of precision values at each relevant document position
Formula: `AP = (1/R) × Σ[P(k) × rel(k)]`

- R = total relevant documents
- P(k) = precision at position k
- rel(k) = 1 if doc at k is relevant, else 0

Step-by-Step Calculation:

```
Ranking: [R, N, R, R, N, R, N, N, N, R]
Total relevant (R): 5

Position | Doc | Relevant? | # Rel so far | P(k) | P(k)×rel(k)
---------|-----|-----------|--------------|------|------------
    1    |  R  |     1     |      1       | 1/1  |    1.000
    2    |  N  |     0     |      1       | 1/2  |    0.000
    3    |  R  |     1     |      2       | 2/3  |    0.667
    4    |  R  |     1     |      3       | 3/4  |    0.750
    5    |  N  |     0     |      3       | 3/5  |    0.000
    6    |  R  |     1     |      4       | 4/6  |    0.667
    7    |  N  |     0     |      4       | 4/7  |    0.000
    8    |  N  |     0     |      4       | 4/8  |    0.000
    9    |  N  |     0     |      4       | 4/9  |    0.000
   10    |  R  |     1     |      5       | 5/10 |    0.500

Sum of P(k)×rel(k) = 1.000 + 0.667 + 0.750 + 0.667 + 0.500 = 3.584
AP = 3.584 / 5 = 0.717
```

Key Insight: AP rewards putting relevant docs earlier!

---

7. MEAN AVERAGE PRECISION (MAP)

Definition: Mean of AP across multiple queries
Formula: `MAP = (1/Q) × Σ[AP_q]`
Most popular metric in IR research

Example:

```
Query 1: AP = 0.80
Query 2: AP = 0.65
Query 3: AP = 0.90
Query 4: AP = 0.70

MAP = (0.80 + 0.65 + 0.90 + 0.70) / 4 = 0.7625
```

---

8. RECIPROCAL RANK (RR)

Definition: Inverse rank of first relevant document
Formula: `RR = 1 / rank_first_relevant`
Use case: Known-item search, question answering

Examples:

```
[R, N, N, ...] → RR = 1/1 = 1.000 (perfect)
[N, N, R, ...] → RR = 1/3 = 0.333
[N, N, N, N, N, R, ...] → RR = 1/6 = 0.167
No relevant found → RR = 0
```

Mean Reciprocal Rank (MRR):

```
MRR = (1/Q) × Σ[RR_q]

Query 1: First relevant at rank 1 → RR = 1.0
Query 2: First relevant at rank 3 → RR = 0.333
Query 3: First relevant at rank 2 → RR = 0.5

MRR = (1.0 + 0.333 + 0.5) / 3 = 0.611
```

---

9. R-PRECISION

Definition: Precision at R, where R = number of relevant docs for query
Formula: `R-Precision = (# relevant in top R positions) / R`

Example:

```
Query has 8 relevant documents total
Top 8 results contain 6 relevant documents
R-Precision = 6/8 = 0.75
```

🎓 GRADED RELEVANCE METRICS

10. DISCOUNTED CUMULATIVE GAIN (DCG)

Definition: Cumulative gain with logarithmic discount by rank
Use case: When relevance has multiple levels (0=not relevant, 1=somewhat, 2=relevant, 3=highly relevant)

Formula (Version 1):

```
DCG@k = Σ[i=1 to k] (rel_i / log₂(i+1))
```

Formula (Version 2 - Industry Standard):

```
DCG@k = rel₁ + Σ[i=2 to k] (rel_i / log₂(i))
```

Formula (Version 3 - Exponential Gain):

```
DCG@k = Σ[i=1 to k] ((2^rel_i - 1) / log₂(i+1))
```

Example (using Version 2):

```
Ranking with relevance scores: [3, 2, 3, 0, 1, 2]

DCG@6 = 3 + 2/log₂(2) + 3/log₂(3) + 0/log₂(4) + 1/log₂(5) + 2/log₂(6)
      = 3 + 2/1 + 3/1.585 + 0/2 + 1/2.322 + 2/2.585
      = 3 + 2.000 + 1.893 + 0 + 0.431 + 0.774
      = 8.098

Logarithm values:
log₂(2) = 1.000
log₂(3) = 1.585
log₂(4) = 2.000
log₂(5) = 2.322
log₂(6) = 2.585
log₂(7) = 2.807
log₂(8) = 3.000
log₂(9) = 3.170
log₂(10) = 3.322
```

---

11. NORMALIZED DCG (nDCG)

Definition: DCG normalized by ideal DCG
Formula: `nDCG@k = DCG@k / IDCG@k`
Range: [0, 1] (higher is better)

IDCG = DCG of perfect ranking (relevance scores sorted descending)

Complete Example:

```
Your System's Ranking: [3, 2, 3, 0, 1, 2]
DCG@6 = 8.098 (calculated above)

Ideal Ranking (sorted): [3, 3, 2, 2, 1, 0]
IDCG@6 = 3 + 3/log₂(2) + 2/log₂(3) + 2/log₂(4) + 1/log₂(5) + 0/log₂(6)
       = 3 + 3/1 + 2/1.585 + 2/2 + 1/2.322 + 0
       = 3 + 3.000 + 1.262 + 1.000 + 0.431 + 0
       = 8.693

nDCG@6 = 8.098 / 8.693 = 0.932
```

Interpretation:

- nDCG = 1.0: Perfect ranking
- nDCG = 0.0: Worst ranking
- nDCG = 0.932: Excellent! (93.2% of ideal)

Why normalize? Allows comparison across queries with different numbers of relevant documents

---

🔄 ADVANCED METRICS

12. EXPECTED RECIPROCAL RANK (ERR)

Definition: Models cascade user behavior (stops when satisfied)
Formula: `ERR = Σ[i=1 to k] (1/i × P(user stops at i))`

Probability model:

```
R_i = (2^rel_i - 1) / 2^max_grade          [satisfaction probability]
P(user reaches i) = ∏[j=1 to i-1] (1 - R_j)
P(user stops at i) = P(user reaches i) × R_i
```

Example (max_grade=3):

```
Ranking: [2, 1, 3, 0]

R₁ = (2²-1)/2³ = 3/8 = 0.375
R₂ = (2¹-1)/2³ = 1/8 = 0.125
R₃ = (2³-1)/2³ = 7/8 = 0.875
R₄ = (2⁰-1)/2³ = 0/8 = 0.000

P(stop at 1) = 0.375
P(stop at 2) = (1-0.375) × 0.125 = 0.078
P(stop at 3) = (1-0.375)(1-0.125) × 0.875 = 0.478
P(stop at 4) = (1-0.375)(1-0.125)(1-0.875) × 0 = 0

ERR = 1×0.375 + (1/2)×0.078 + (1/3)×0.478 + (1/4)×0
    = 0.375 + 0.039 + 0.159 + 0
    = 0.573
```

---

13. RANK-BIASED PRECISION (RBP)

Definition: Geometric discount based on user persistence
Formula: `RBP = (1-p) × Σ[i=1 to k] p^(i-1) × rel_i`
Parameter: p = persistence (typical: p=0.8)

Example (p=0.8, binary relevance):

```
Ranking: [R, N, R, R, N]

RBP = (1-0.8) × [0.8⁰×1 + 0.8¹×0 + 0.8²×1 + 0.8³×1 + 0.8⁴×0]
    = 0.2 × [1 + 0 + 0.64 + 0.512 + 0]
    = 0.2 × 2.152
    = 0.430
```

Interpretation of p:

- p=0.5: User has 50% chance to view next result
- p=0.8: User is quite patient (80% continue)
- p=0.95: Very persistent user

---

14. BPREF

Definition: Handles incomplete relevance judgments
Formula: `bpref = (1/R) × Σ[r∈R] (1 - |n ranked higher than r| / min(R, N))`

- R = relevant documents
- N = judged non-relevant documents

Use case: When not all documents have been judged

PRECISION-RECALL CURVES

Standard Recall Levels

Evaluate at: **0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0**

11-Point Interpolated Precision

Formula: `P_interp(r) = max{P(r') : r' ≥ r}`

Example:

```
Actual PR points: (0.1, 0.9), (0.3, 0.8), (0.5, 0.6), (0.7, 0.4), (0.9, 0.2)

Interpolated values:
P(0.0) = max{0.9, 0.8, 0.6, 0.4, 0.2} = 0.9
P(0.1) = max{0.9, 0.8, 0.6, 0.4, 0.2} = 0.9
P(0.2) = max{0.8, 0.6, 0.4, 0.2} = 0.8
P(0.3) = max{0.8, 0.6, 0.4, 0.2} = 0.8
P(0.4) = max{0.6, 0.4, 0.2} = 0.6
P(0.5) = max{0.6, 0.4, 0.2} = 0.6
P(0.6) = max{0.4, 0.2} = 0.4
P(0.7) = max{0.4, 0.2} = 0.4
P(0.8) = max{0.2} = 0.2
P(0.9) = max{0.2} = 0.2
P(1.0) = 0.2

Mean Interpolated Precision = (0.9+0.9+0.8+0.8+0.6+0.6+0.4+0.4+0.2+0.2+0.2) / 11
                            = 6.0 / 11 = 0.545
```

METRIC SELECTION GUIDE

Scenario	Recommended Metrics	Rationale
Web Search	nDCG@10, MAP, P@10	Users view ~10 results; graded relevance
Known-Item Search	MRR, Success@1	Finding one specific document
E-commerce	nDCG@20, Click-through Rate	Revenue from early results
Academic Search	MAP, nDCG@100, Recall	Researchers want comprehensive results
Question Answering	MRR, Success@1, ERR	One correct answer needed
Legal/Medical IR	Recall, F-measure	Cannot miss relevant documents
News/Social Media	P@10, nDCG@10	Recent/relevant content in top results
Incomplete Judgments	bpref, nDCG	Not all docs judged

STATISTICAL SIGNIFICANCE TESTING

Paired t-test

Use: Compare two systems across queries
Null hypothesis: No difference in mean scores

```
from scipy import stats

system_a_scores = [0.8, 0.7, 0.85, 0.6, 0.9]  # One score per query
system_b_scores = [0.75, 0.72, 0.8, 0.65, 0.88]

t_statistic, p_value = stats.ttest_rel(system_a_scores, system_b_scores)

if p_value < 0.05:
    print("Statistically significant difference!")
```

Wilcoxon Signed-Rank Test

Use: Non-parametric alternative (doesn't assume normality)

```
from scipy import stats

w_statistic, p_value = stats.wilcoxon(system_a_scores, system_b_scores)
```

Significance Levels

- p < 0.05: Significant (*)
- p < 0.01: Highly significant (**)
- p < 0.001: Very highly significant (***)

⚠ COMMON PITFALLS

1. Wrong Averaging

```
❌ WRONG: F1 = (P + R) / 2  [arithmetic mean]
✓ CORRECT: F1 = 2PR / (P + R)  [harmonic mean]
```

2. Ignoring Ranking Position

```
❌ Using only Precision/Recall
✓ Use MAP, nDCG, or RBP instead
```

3. Small Test Sets

```
❌ Testing on 5 queries
✓ Need 50+ queries for reliable results
```

4. Cherry-Picking Metrics

```
❌ Only reporting metrics where you win
✓ Report standard metrics (MAP, nDCG@10)
```

5. Treating Evaluation as Ground Truth

```
❌ Assuming relevance judgments are perfect
✓ Understand inter-annotator agreement, use multiple judges
```

📊 QUICK REFERENCE FORMULAS

```
Precision = TP / (TP + FP)
Recall = TP / (TP + FN)
F1 = 2 × P × R / (P + R)
P@K = (# relevant in top K) / K
AP = (1/R) × Σ[P(k) × rel(k)]
MAP = (1/Q) × Σ[AP_q]
MRR = (1/Q) × Σ[1/rank_first_relevant_q]
DCG@k = Σ[rel_i / log₂(i+1)]
nDCG@k = DCG@k / IDCG@k
```

🧮 WORKED EXAMPLE: COMPLETE EVALUATION

Scenario: 3 queries, binary relevance

Query 1: 5 relevant docs total
Ranking: `[R, R, N, R, N, N, R, N, R, N]`

```
P@5 = 3/5 = 0.60
P@10 = 5/10 = 0.50
Recall@10 = 5/5 = 1.00

AP calculation:
Position 1 (R): P(1) = 1/1 = 1.000
Position 2 (R): P(2) = 2/2 = 1.000
Position 4 (R): P(4) = 3/4 = 0.750
Position 7 (R): P(7) = 4/7 = 0.571
```

```
Position 9 (R): P(9) = 5/9 = 0.556
AP = (1.000 + 1.000 + 0.750 + 0.571 + 0.556) / 5 = 0.775

RR = 1/1 = 1.000 (first relevant at position 1)
```

Query 2: 3 relevant docs total
Ranking: `[N, R, N, N, R, N, R, N, N, N]`

```
P@5 = 2/5 = 0.40
P@10 = 3/10 = 0.30
Recall@10 = 3/3 = 1.00

AP calculation:
Position 2 (R): P(2) = 1/2 = 0.500
Position 5 (R): P(5) = 2/5 = 0.400
Position 7 (R): P(7) = 3/7 = 0.429
AP = (0.500 + 0.400 + 0.429) / 3 = 0.443

RR = 1/2 = 0.500
```

Query 3: 4 relevant docs total
Ranking: `[R, N, R, R, R, N, N, N, N, N]`

```
P@5 = 4/5 = 0.80
P@10 = 4/10 = 0.40
Recall@10 = 4/4 = 1.00

AP calculation:
Position 1 (R): P(1) = 1/1 = 1.000
Position 3 (R): P(3) = 2/3 = 0.667
Position 4 (R): P(4) = 3/4 = 0.750
Position 5 (R): P(5) = 4/5 = 0.800
AP = (1.000 + 0.667 + 0.750 + 0.800) / 4 = 0.804

RR = 1/1 = 1.000
```

Aggregate Metrics:

```
MAP = (0.775 + 0.443 + 0.804) / 3 = 0.674
MRR = (1.000 + 0.500 + 1.000) / 3 = 0.833
Mean P@5 = (0.60 + 0.40 + 0.80) / 3 = 0.600
Mean P@10 = (0.50 + 0.30 + 0.40) / 3 = 0.400
Mean Recall@10 = (1.00 + 1.00 + 1.00) / 3 = 1.000
```

🎓 TEST COLLECTION RESOURCES

Major IR Test Collections

Collection	Domain	Queries	Documents	URL
TREC	News, web, tweets	Varies	Millions	trec.nist.gov
MS MARCO	Web search, QA	1M+	8.8M passages	microsoft.github.io/msmarco
ClueWeb	Web pages	Various	1B+ pages	lemurproject.org/clueweb
Robust04	News articles	250	528K docs	Standard for research
NTCIR	Asian languages	Varies	Varies	research.nii.ac.jp/ntcir

Evaluation Tools

- **trec_eval**: Standard TREC evaluation tool (C)
- **pytrec_eval**: Python wrapper for trec_eval
- **ir_measures**: Modern Python IR evaluation library
- **ranx**: Fast ranking evaluation (Python)

PYTHON IMPLEMENTATION SNIPPETS

Calculate MAP

```
def average_precision(relevant_positions, total_relevant):
    """
    relevant_positions: list of ranks where relevant docs appear (1-indexed)
    total_relevant: total number of relevant documents
    """
    if total_relevant == 0:
        return 0.0

    precision_sum = 0.0
    for i, pos in enumerate(relevant_positions, start=1):
        precision_at_pos = i / pos
        precision_sum += precision_at_pos

    return precision_sum / total_relevant

# Example
relevant_positions = [1, 2, 4, 7, 9]  # ranks of relevant docs
total_relevant = 5
ap = average_precision(relevant_positions, total_relevant)
print(f"AP: {ap:.4f}")  # Output: AP: 0.7754
```

Calculate nDCG

```python
import math

def dcg_at_k(relevances, k):
    """
    relevances: list of relevance scores in ranking order
    k: cutoff position
    """
    dcg = 0.0
    for i, rel in enumerate(relevances[:k], start=1):
        if i == 1:
            dcg += rel
        else:
            dcg += rel / math.log2(i)
    return dcg

def ndcg_at_k(relevances, k):
    """
    Calculate nDCG@k
    """
    dcg = dcg_at_k(relevances, k)
    ideal_relevances = sorted(relevances, reverse=True)
    idcg = dcg_at_k(ideal_relevances, k)

    if idcg == 0:
        return 0.0
    return dcg / idcg

# Example
ranking_relevances = [3, 2, 3, 0, 1, 2]
k = 6
ndcg = ndcg_at_k(ranking_relevances, k)
print(f"nDCG@{k}: {ndcg:.4f}")  # Output: nDCG@6: 0.9315
```

Calculate MRR

```python
def reciprocal_rank(ranking, relevant_ids):
    """
    ranking: list of document IDs in ranking order
    relevant_ids: set of relevant document IDs
    """
    for i, doc_id in enumerate(ranking, start=1):
        if doc_id in relevant_ids:
            return 1.0 / i
    return 0.0

def mrr(queries_rankings, queries_relevant):
    """
    queries_rankings: list of rankings (one per query)
    queries_relevant: list of relevant doc sets (one per query)
    """
```

```
    rr_sum = sum(reciprocal_rank(ranking, relevant)
                 for ranking, relevant in zip(queries_rankings, queries_relevant))
    return rr_sum / len(queries_rankings)

# Example
rankings = [
    ['doc1', 'doc2', 'doc3'],  # Query 1: first relevant is doc1 (rank 1)
    ['doc5', 'doc6', 'doc7'],  # Query 2: first relevant is doc7 (rank 3)
    ['doc8', 'doc9', 'doc10'], # Query 3: first relevant is doc9 (rank 2)
]
relevant = [
    {'doc1', 'doc4'},
    {'doc7', 'doc8'},
    {'doc9', 'doc11'},
]
mrr_score = mrr(rankings, relevant)
print(f"MRR: {mrr_score:.4f}")  # Output: MRR: 0.6111
```

KEY TAKEAWAYS

1. **No single perfect metric** - choose based on use case
2. **Ranking position matters** - use MAP, nDCG, not just P/R
3. **Graded relevance is realistic** - use nDCG when available
4. **Report multiple metrics** - gives complete picture
5. **Test statistical significance** - don't over-interpret small differences
6. **Need enough queries** - 50+ for reliable evaluation
7. **User behavior matters** - ERR, RBP model how users interact
8. **Incomplete judgments are normal** - use pooling, bpref

FURTHER READING

Classic Papers

- Järvelin & Kekäläinen (2002): "Cumulated gain-based evaluation of IR techniques" (introduced DCG/nDCG)
- Voorhees & Harman (2005): "TREC: Experiment and Evaluation in Information Retrieval"
- Buckley & Voorhees (2004): "Retrieval evaluation with incomplete information"

Books

- Manning, Raghavan, Schütze: "Introduction to Information Retrieval" (Chapter 8)
- Croft, Metzler, Strohman: "Search Engines: Information Retrieval in Practice" (Chapter 8)

Online Resources

- TREC evaluation guidelines: https://trec.nist.gov/pubs.html
- IR-Anthology: https://ir.dcs.gla.ac.uk/wiki
- Information Retrieval evaluation survey papers on arXiv

End of Cheatsheet | Last Updated: 2025 | For CS-6001: Indexing and Retrieving Text and Graphs

RELEVANCE SCORE CHEATSHEET - COMPLETE EDITION

CS-6001: Indexing and Retrieving Text and Graphs

TABLE OF CONTENTS

BASIC CONCEPTS

Notation Guide

- t = term (word)
- d = document
- q = query
- N = total number of documents in collection
- df(t) = document frequency (number of docs containing term t)
- tf(t,d) = term frequency (number of times t appears in d)
- |d| = document length (number of terms in document)
- avgdl = average document length in collection
- K = number of top results to return

TERM FREQUENCY (TF)

Raw Term Frequency

```
tf(t, d) = count of term t in document d
```

WHAT IT DOES: Counts how many times a term appears in a document.

EXAMPLE:

- Document: "cat dog cat cat"
- tf(cat, d) = 3
- tf(dog, d) = 1

Normalized Term Frequency

```
tf_norm(t, d) = tf(t, d) / |d|
```

WHAT IT DOES: Divides term count by document length to prevent bias toward longer documents.

EXAMPLE:

- Document d1 (length=100): "machine" appears 5 times → tf_norm = 5/100 = 0.05
- Document d2 (length=50): "machine" appears 3 times → tf_norm = 3/50 = 0.06

Logarithmic Term Frequency

```
tf_log(t, d) = 1 + log10(tf(t, d))    if tf(t, d) > 0
             = 0                        if tf(t, d) = 0
```

WHAT IT DOES: Dampens the effect of very high term frequencies using logarithm.

EXAMPLE:

- tf = 1 → tf_log = 1 + log10(1) = 1.0
- tf = 10 → tf_log = 1 + log10(10) = 2.0
- tf = 100 → tf_log = 1 + log10(100) = 3.0

WHY: A document with 100 mentions is not 100x more relevant than one with 1 mention.

INVERSE DOCUMENT FREQUENCY (IDF)

IDF Formula

```
IDF(t) = log10(N / df(t))
```

WHAT IT DOES: Measures how rare/important a term is across the entire collection.

EXAMPLE (N = 10,000 documents):

- "the" appears in 9,900 docs → IDF = log10(10,000/9,900) = 0.004 (low, common word)
- "algorithm" appears in 100 docs → IDF = log10(10,000/100) = 2.0 (higher, rare word)
- "backpropagation" appears in 10 docs → IDF = log10(10,000/10) = 3.0 (highest, very rare)

INTUITION: Rare words are more informative for distinguishing documents.

Smoothed IDF (prevents division by zero)

```
IDF_smooth(t) = log10((N + 1) / (df(t) + 1))
```

WHAT IT DOES: Adds 1 to numerator and denominator to handle terms not in collection.

TF-IDF

TF-IDF Score

```
TF-IDF(t, d) = tf(t, d) × IDF(t)
```

WHAT IT DOES: Combines term frequency and inverse document frequency.

- High TF = term appears often in document (good)
- High IDF = term is rare across collection (good)

EXAMPLE:

- Query: "machine learning"
- Document: contains "machine" 3 times (tf=3), "learning" 2 times (tf=2)
- IDF(machine) = 1.5, IDF(learning) = 1.3
- TF-IDF(machine) = 3 × 1.5 = 4.5
- TF-IDF(learning) = 2 × 1.3 = 2.6
- TOTAL SCORE = 4.5 + 2.6 = 7.1

TF-IDF with Logarithmic TF

```
TF-IDF_log(t, d) = (1 + log10(tf(t, d))) × log10(N / df(t))
```

WHAT IT DOES: Uses log-scaled TF for better performance.

Query-Document TF-IDF Score

```
Score(q, d) = SUM of TF-IDF(t, d) for all terms t in query q
```

WHAT IT DOES: Sums TF-IDF scores for all query terms.

TF-IDF PITFALLS AND LIMITATIONS

Pitfall 1: Length Bias

PROBLEM: TF-IDF naturally favors longer documents because they have more terms.

EXAMPLE:

- Doc A (100 words): "machine" appears 2 times → tf = 2
- Doc B (1000 words): "machine" appears 5 times → tf = 5
- Doc B gets higher score even though density is lower (5/1000 < 2/100)

SOLUTION: Use normalized TF or BM25 with length normalization

---

Pitfall 2: No Term Saturation

PROBLEM: TF grows linearly - a document with 100 mentions scores 100x higher than one with 1 mention.

EXAMPLE:

- Doc A: "algorithm" appears 1 time → score = 1 × 2.0 = 2.0
- Doc B: "algorithm" appears 100 times (keyword stuffing!) → score = 100 × 2.0 = 200

REALITY: Doc B is probably spam, not 100x more relevant.

SOLUTION: Use logarithmic TF or BM25 with saturation parameter k1

Pitfall 3: Ignores Term Position

PROBLEM: TF-IDF treats all term occurrences equally, regardless of where they appear.

EXAMPLE:

- Doc A: "machine learning" in title and first sentence
- Doc B: "machine learning" buried in middle of long document
- Both get same score if tf and df are equal

REALITY: Terms in title/beginning are often more important.

SOLUTION: Use field weighting or position-aware scoring

Pitfall 4: No Query Term Proximity

PROBLEM: TF-IDF treats query terms independently.

EXAMPLE: Query: "machine learning"

- Doc A: "machine ... [500 words] ... learning"
- Doc B: "machine learning" (adjacent)
- Both get same score

REALITY: Adjacent terms often indicate higher relevance (it's a phrase!).

SOLUTION: Use phrase queries, proximity scoring, or neural models

---

Pitfall 5: Vocabulary Mismatch (Synonym Problem)

PROBLEM: TF-IDF requires exact word match.

EXAMPLE: Query: "car"

- Doc A: "automobile vehicle transportation"
- Doc A gets ZERO score despite being relevant

SOLUTION: Query expansion, stemming, word embeddings, or neural models

---

Pitfall 6: Collection-Dependent IDF

PROBLEM: IDF values change when collection changes.

EXAMPLE:

- Small collection (1000 docs): IDF(algorithm) = log(1000/10) = 2.0
- Add 9000 docs about algorithms: IDF(algorithm) = log(10000/5000) = 0.3

REALITY: IDF drops significantly, changing all rankings.

SOLUTION: Use fixed IDF from reference collection, or retrain regularly

---

Pitfall 7: No Document Quality Signal

PROBLEM: TF-IDF only considers text matching, not document authority/quality.

EXAMPLE:

- Doc A: High-quality research paper
- Doc B: Low-quality spam blog
- Both can get same TF-IDF score

SOLUTION: Combine with PageRank, user signals, or quality classifiers

---

Pitfall 8: Poor Performance on Short Queries

PROBLEM: 1-2 word queries don't provide enough signal for good ranking.

EXAMPLE: Query: "python"

- Could mean: programming language, snake, Monty Python
- TF-IDF has no context

SOLUTION: Use query expansion, personalization, or context-aware models

---

TF-IDF vs BM25 Comparison

ASPECT	TF-IDF	BM25
Term Frequency	Linear or log	Saturating (controlled by k1)
Length Normalization	None (unless added)	Built-in (controlled by b)
Theoretical Foundation	Heuristic	Probabilistic (BM25)
Parameter Tuning	Limited	k1 and b tunable
Performance	Good baseline	Better in most cases
Computational Cost	Fast	Slightly slower
Use Cases	Simple systems, baseline	Production systems

BM25 - BEST MATCH 25

BM25 Formula (THE INDUSTRY STANDARD)

```
BM25(q, d) = SUM over all terms t in query q of:
             IDF(t) × (tf(t,d) × (k1 + 1)) / (tf(t,d) + k1 × (1 - b + b × (|d| / avgdl)))
```

WHAT IT DOES: Advanced ranking function with:

- Term frequency saturation (k1 parameter)
- Document length normalization (b parameter)
- IDF weighting

WHY IT'S BETTER: Addresses TF-IDF pitfalls 1, 2, and 6

BM25 Components

IDF Component:

```
IDF(t) = log10((N - df(t) + 0.5) / (df(t) + 0.5))
```

TF Saturation Component:

```
tf_component = (tf(t,d) × (k1 + 1)) / (tf(t,d) + k1 × (1 - b + b × (|d| / avgdl)))
```

BM25 Parameters

k1 (Term Frequency Saturation): Controls how quickly term frequency stops helping

- TYPICAL VALUE: 1.2 to 2.0
- k1 = 0 → TF does not matter at all
- k1 = infinity → Linear TF (like TF-IDF)
- k1 = 1.5 (common) → Diminishing returns

b (Length Normalization): Controls document length penalty

- TYPICAL VALUE: 0.75
- b = 0 → No length normalization
- b = 1 → Full length normalization
- b = 0.75 (common) → Moderate normalization

BM25 Complete Example

SETUP:

- Query: "machine learning"
- Document d: length = 600 words, avgdl = 500
- tf(machine, d) = 3, tf(learning, d) = 2
- df(machine) = 100, df(learning) = 200, N = 10,000
- k1 = 1.5, b = 0.75

STEP 1: Calculate IDF:

```
IDF(machine) = log10((10,000 - 100 + 0.5) / (100 + 0.5)) = 1.99
IDF(learning) = log10((10,000 - 200 + 0.5) / (200 + 0.5)) = 1.69
```

STEP 2: Calculate length normalization:

```
length_norm = 1 - b + b × (|d| / avgdl)
            = 1 - 0.75 + 0.75 × (600 / 500)
            = 0.25 + 0.9 = 1.15
```

STEP 3: Calculate BM25 for each term:

```
BM25(machine) = 1.99 × (3 × 2.5) / (3 + 1.5 × 1.15)
              = 1.99 × 7.5 / 4.725
```

```
               = 3.16

BM25(learning) = 1.69 × (2 × 2.5) / (2 + 1.5 × 1.15)
               = 1.69 × 5.0 / 3.725
               = 2.27
```

TOTAL BM25 SCORE = 3.16 + 2.27 = 5.43

VECTOR SPACE MODEL

Document as Vector

```
Document d = [w1, w2, w3, ..., wn]
```

Where wi is the weight (TF-IDF or BM25) of term i in document d.

EXAMPLE:

- Vocabulary: [cat, dog, pet, animal, sat]
- Document: "The cat sat"
- Vector: [2.5, 0, 0, 0, 1.8] (using TF-IDF weights)

Query as Vector

```
Query q = [w1, w2, w3, ..., wn]
```

Same dimensionality as documents.

EXAMPLE:

- Query: "cat pet"
- Vector: [1.5, 0, 2.1, 0, 0]

COSINE SIMILARITY

Cosine Similarity Formula

```
cosine(q, d) = (q · d) / (||q|| × ||d||)
```

WHERE:

- q · d = dot product = SUM(qi × di)

- ||q|| = magnitude of q = SQRT(SUM qi^2)
- ||d|| = magnitude of d = SQRT(SUM di^2)

WHAT IT DOES: Measures angle between query and document vectors.

- Value = 1 → Vectors identical (maximum similarity)
- Value = 0 → Vectors perpendicular (no similarity)
- Range: [0, 1] for text (can be [-1, 1] in general)

Cosine Similarity Step-by-Step Example

Query vector q: [2.5, 2.0, 0, 0] Document vector d: [1.5, 1.2, 0.8, 0.5]

STEP 1: Dot Product:

```
q · d = (2.5 × 1.5) + (2.0 × 1.2) + (0 × 0.8) + (0 × 0.5)
      = 3.75 + 2.4 + 0 + 0
      = 6.15
```

STEP 2: Magnitude of q:

```
||q|| = SQRT(2.5^2 + 2.0^2 + 0^2 + 0^2)
      = SQRT(6.25 + 4.0)
      = SQRT(10.25)
      = 3.20
```

STEP 3: Magnitude of d:

```
||d|| = SQRT(1.5^2 + 1.2^2 + 0.8^2 + 0.5^2)
      = SQRT(2.25 + 1.44 + 0.64 + 0.25)
      = SQRT(4.58)
      = 2.14
```

STEP 4: Cosine Similarity:

```
cosine(q, d) = 6.15 / (3.20 × 2.14)
             = 6.15 / 6.85
             = 0.898
```

RESULT: High similarity (0.898 out of 1.0) → Document is very relevant!

LENGTH NORMALIZATION

Pivoted Length Normalization

```
normalized_tf = tf / (1 - b + b × (|d| / avgdl))
```

WHAT IT DOES: Adjusts term frequency based on document length.

- Short docs: denominator < 1 → TF score increases
- Long docs: denominator > 1 → TF score decreases

PARAMETERS:

- b = 0: No normalization
- b = 1: Full normalization
- b = 0.5 to 0.8: Typical values

Byte-Size Length Normalization

```
normalized_tf = tf / |d|^alpha
```

Where alpha controls the strength of normalization (typical: alpha = 0.5)

QUERY PROCESSING STRATEGIES

Overview

When processing a query against millions of documents, we need efficient strategies. Goal: Find top-K documents without scoring ALL documents.

DOCUMENT-AT-A-TIME (DAAT)

DAAT Algorithm

```
ALGORITHM: Document-at-a-Time

INPUT: Query q = [t1, t2, ..., tm], Top-K to return
OUTPUT: Top-K ranked documents

1. Initialize: heap H of size K (min-heap by score)
2. Get posting lists for all query terms
3. Find all candidate document IDs (union of posting lists)
4. FOR EACH document d in candidates:
     a. score = 0
     b. FOR EACH term t in query q:
          score += scoring_function(t, d)
```

```
      c. IF heap size < K OR score > H.min():
           H.insert(d, score)
           IF H.size() > K: H.remove_min()
5. RETURN sorted(H, reverse=True)
```

WHAT IT DOES: Process one document completely before moving to the next.

DAAT Characteristics

ADVANTAGES:

- Simple to implement
- Can compute exact scores easily
- Good cache locality (all data for one doc accessed together)
- Can apply early termination (stop if score can't beat top-K)

DISADVANTAGES:

- Must identify all candidate documents first
- Memory overhead for storing partial scores
- Difficult to skip documents efficiently

DAAT Example

Query: "machine learning" Posting lists:

- "machine": [Doc1, Doc2, Doc5, Doc7]
- "learning": [Doc1, Doc3, Doc5, Doc8]

Candidates: [Doc1, Doc2, Doc3, Doc5, Doc7, Doc8]

Processing:

```
Process Doc1:
  - score = BM25(machine, Doc1) + BM25(learning, Doc1) = 2.5 + 1.8 = 4.3
  - Insert into heap

Process Doc2:
  - score = BM25(machine, Doc2) + 0 = 1.2
  - Insert into heap

Process Doc3:
  - score = 0 + BM25(learning, Doc3) = 2.1
  - Insert into heap

... and so on
```

DAAT with Early Termination

```
threshold = H.min()  (minimum score in top-K heap)
max_remaining_score = SUM of max_possible_score(t) for remaining terms

IF current_score + max_remaining_score < threshold:
   SKIP this document (cannot make it to top-K)
```

TERM-AT-A-TIME (TAAT)

TAAT Algorithm

```
ALGORITHM: Term-at-a-Time

INPUT: Query q = [t1, t2, ..., tm], Top-K to return
OUTPUT: Top-K ranked documents

1. Initialize: accumulator array A (maps doc_id → score)
2. FOR EACH term t in query q:
     a. Fetch posting list for t
     b. FOR EACH document d in posting list:
          A[d] += scoring_function(t, d)
3. Find top-K documents from accumulator array A
4. RETURN sorted top-K documents
```

WHAT IT DOES: Process one query term completely across all documents before moving to next term.

TAAT Characteristics

ADVANTAGES:

- Better for compressed posting lists (decompress once)
- Can process posting lists in parallel
- Can skip terms with low IDF
- More opportunities for optimization

DISADVANTAGES:

- Requires accumulator array (memory overhead)
- Poor cache locality (jumping between documents)
- Score computation spread across term iterations

TAAT Example

Query: "machine learning" Posting lists:

- "machine": [Doc1(tf=2), Doc2(tf=1), Doc5(tf=3), Doc7(tf=1)]
- "learning": [Doc1(tf=1), Doc3(tf=2), Doc5(tf=1), Doc8(tf=3)]

Processing:

```
Accumulator array A initially: {}

Process term "machine":
  - A[Doc1] = 0 + BM25(machine, Doc1) = 2.5
  - A[Doc2] = 0 + BM25(machine, Doc2) = 1.2
  - A[Doc5] = 0 + BM25(machine, Doc5) = 3.1
  - A[Doc7] = 0 + BM25(machine, Doc7) = 1.1

Process term "learning":
  - A[Doc1] = 2.5 + BM25(learning, Doc1) = 2.5 + 1.8 = 4.3
  - A[Doc3] = 0 + BM25(learning, Doc3) = 2.1
  - A[Doc5] = 3.1 + BM25(learning, Doc5) = 3.1 + 1.6 = 4.7
  - A[Doc8] = 0 + BM25(learning, Doc8) = 2.8

Final scores: {Doc1: 4.3, Doc2: 1.2, Doc3: 2.1, Doc5: 4.7, Doc7: 1.1, Doc8: 2.8}
Top-3: [Doc5 (4.7), Doc1 (4.3), Doc8 (2.8)]
```

DAAT vs TAAT COMPARISON

ASPECT	DAAT	TAAT
Processing Order	Document by document	Term by term
Memory Usage	Lower (heap only)	Higher (accumulator array)
Cache Locality	Better	Worse
Early Termination	Easier	Harder
Parallelization	Harder	Easier
Posting List Access	Random access needed	Sequential access
Implementation	Simpler	More complex
Best For	Small queries, exact scores	Large queries, approximate
Used By	Many systems	Lucene, Elasticsearch

INDEX STRUCTURES

Inverted Index (Basic)

```
STRUCTURE:
term → [doc1, doc2, doc3, ...]
```

```
EXAMPLE:
"machine" → [Doc1, Doc2, Doc5, Doc7, Doc12]
"learning" → [Doc1, Doc3, Doc5, Doc8, Doc10]
```

WHAT IT DOES: Maps each term to list of documents containing it.

Inverted Index with Frequencies

```
STRUCTURE:
term → [(doc1, tf1), (doc2, tf2), (doc3, tf3), ...]

EXAMPLE:
"machine" → [(Doc1, 3), (Doc2, 1), (Doc5, 5), (Doc7, 2)]
"learning" → [(Doc1, 2), (Doc3, 4), (Doc5, 1), (Doc8, 3)]
```

WHAT IT DOES: Stores term frequency for each document.

ADVANTAGE: Can compute TF-IDF and BM25 scores directly.

Positional Index

```
STRUCTURE:
term → [(doc1, [pos1, pos2, ...]), (doc2, [pos1, pos2, ...]), ...]

EXAMPLE:
"machine" → [(Doc1, [5, 47, 89]), (Doc2, [12]), (Doc5, [3, 45, 67, 101, 203])]
"learning" → [(Doc1, [6, 48]), (Doc3, [8, 23, 45, 99])]
```

WHAT IT DOES: Stores positions where term appears in each document.

ADVANTAGES:

- Enables phrase queries: "machine learning" (positions 5,6 or 47,48)
- Enables proximity queries: "machine NEAR/5 learning"
- Better relevance scoring (position-based features)

DISADVANTAGE: Much larger index size (3-5x bigger)

Index Organization

POSTING LIST STRUCTURE (for each term):

```
[doc_id, tf, positions] → [doc_id, tf, positions] → ...
```

EXAMPLE for "machine":

```
[Doc1, 3, [5,47,89]] → [Doc2, 1, [12]] → [Doc5, 5, [3,45,67,101,203]]
```

STORED AS:

1. Dictionary: term → pointer to posting list
2. Posting lists: compressed sequences of (doc_id, tf, positions)

Index Compression Techniques

GAP ENCODING: Instead of storing absolute doc IDs, store gaps:

```
Original: [5, 12, 15, 27, 33]
Gap encoded: [5, 7, 3, 12, 6]
```

WHY: Gaps are smaller numbers → compress better

VARIABLE BYTE ENCODING: Store small numbers in fewer bytes:

```
Number < 128: 1 byte
Number < 16384: 2 bytes
Number < 2097152: 3 bytes
...
```

DELTA ENCODING FOR POSITIONS:

```
Original positions: [5, 47, 89]
Delta encoded: [5, 42, 42]
```

QUERY OPTIMIZATION TECHNIQUES

1. MaxScore Optimization

```
IDEA: Compute maximum possible contribution from each term
      Skip processing if remaining terms cannot change top-K

max_score(t) = IDF(t) × max_tf_in_collection(t) × ...
```

```
IF current_score + SUM(max_score(remaining_terms)) < K-th_score:
   SKIP remaining terms for this document
```

BENEFIT: Avoid processing terms that won't affect ranking

---

2. WAND (Weak AND)

```
ALGORITHM: WAND

Maintains upper bounds on term contributions
Efficiently skips documents that cannot be in top-K
Uses skip pointers in posting lists for fast navigation

BENEFIT: Can skip large portions of posting lists
USED BY: Major search engines
```

---

3. Block-Max WAND (BMW)

```
ENHANCEMENT of WAND:
- Divide posting lists into blocks
- Store maximum score for each block
- Skip entire blocks that cannot contribute to top-K

BENEFIT: Even faster than WAND (10-100x speedup)
```

---

4. Cascade Ranking

```
STAGE 1: Fast retrieval with simple scoring (e.g., BM25)
         → Select top 1000 candidates

STAGE 2: Re-rank with more expensive features
         → Machine learning model with 100+ features
         → Select top 100

STAGE 3 (optional): Neural re-ranking
         → BERT or other transformer model
         → Select final top 10

BENEFIT: Balance between speed and quality
```

---

5. Query Term Selection

```
IDEA: Not all query terms are equally important

STRATEGY:
- Remove stopwords: "the", "a", "is"
- Focus on high-IDF terms
- Process essential terms first

EXAMPLE:
Query: "what is the best machine learning algorithm"
Simplified: "best machine learning algorithm"
Priority: "algorithm" (highest IDF), then "machine", "learning", "best"
```

---

6. Dynamic Pruning

```
IDEA: Dynamically decide which documents/terms to skip based on scores

TECHNIQUES:
a. Score-based pruning: Skip docs with low partial scores
b. Term-based pruning: Skip low-IDF terms for some documents
c. Posting-list pruning: Only process top portion of each posting list

BENEFIT: Trade accuracy for speed
```

EVALUATION METRICS

Precision

```
Precision = TP / (TP + FP) = Relevant Retrieved / Total Retrieved
```

WHAT IT DOES: Of all documents we returned, what fraction are relevant?

EXAMPLE:

- Retrieved 20 documents
- 15 are relevant (TP = 15)
- 5 are not relevant (FP = 5)
- PRECISION = 15 / 20 = 0.75 (75%)

Recall

```
Recall = TP / (TP + FN) = Relevant Retrieved / Total Relevant
```

WHAT IT DOES: Of all relevant documents in collection, what fraction did we retrieve?

EXAMPLE:

- 30 relevant documents in collection
- Retrieved 15 of them (TP = 15)
- Missed 15 (FN = 15)
- RECALL = 15 / 30 = 0.50 (50%)

F1 Score (Harmonic Mean of Precision and Recall)

```
F1 = 2 × (Precision × Recall) / (Precision + Recall)
```

WHAT IT DOES: Balances precision and recall into single metric.

EXAMPLE:

- Precision = 0.75, Recall = 0.50
- F1 = 2 × (0.75 × 0.50) / (0.75 + 0.50)
- F1 = 2 × 0.375 / 1.25 = 0.60

Precision at K (P@K)

```
P@K = (Number of relevant docs in top K) / K
```

WHAT IT DOES: Measures precision in top K results only (users rarely look beyond top 10).

EXAMPLE (Top 10 results): Position: 1 2 3 4 5 6 7 8 9 10 Relevant: Y Y N Y N N Y N N N

- P@5 = 3/5 = 0.60
- P@10 = 4/10 = 0.40

Mean Average Precision (MAP)

```
AP(q) = (SUM of (Precision at k × rel(k))) / (Total relevant docs)

MAP = (SUM of AP(q) for all queries) / (Number of queries)
```

WHAT IT DOES: Rewards ranking relevant documents early.

EXAMPLE: Position: 1 2 3 4 5 6 7 8 9 10 Relevant: Y N Y N N Y N N N N

Relevant at positions 1, 3, 6:

- Precision@1 = 1/1 = 1.00
- Precision@3 = 2/3 = 0.67
- Precision@6 = 3/6 = 0.50

AP = (1.00 + 0.67 + 0.50) / 3 = 0.72

Discounted Cumulative Gain (DCG)

```
DCG@K = SUM of (rel_i / log2(i + 1)) for i = 1 to K
```

WHAT IT DOES: Cumulative gain with position discount (lower ranks contribute less).

EXAMPLE (Relevance scores on scale 0-3): Position: 1 2 3 4 5 Relevance: 3 2 3 0 1

DCG@5 = 3/log2(2) + 2/log2(3) + 3/log2(4) + 0/log2(5) + 1/log2(6) = 3/1.0 + 2/1.58 + 3/2.0 + 0/2.32 + 1/2.58 = 3.0 + 1.27 + 1.5 + 0 + 0.39 = 6.16

Normalized DCG (NDCG)

```
NDCG@K = DCG@K / IDCG@K
```

Where IDCG@K = DCG of the ideal ranking (best possible order)

WHAT IT DOES: Normalizes DCG to [0, 1] range, where 1 = perfect ranking.

EXAMPLE (continuing from above):

Ideal order: [3, 3, 2, 1, 0]

IDCG@5 = 3/1.0 + 3/1.58 + 2/2.0 + 1/2.32 + 0/2.58 = 3.0 + 1.90 + 1.0 + 0.43 + 0 = 6.33

NDCG@5 = 6.16 / 6.33 = 0.97

RESULT: 0.97 is excellent! Ranking is nearly perfect.

Mean Reciprocal Rank (MRR)

```
RR(q) = 1 / rank of first relevant document

MRR = (SUM of RR(q) for all queries) / (Number of queries)
```

WHAT IT DOES: Measures how quickly users find the first relevant result.

EXAMPLE:

- Query 1: First relevant at rank 1 → RR = 1/1 = 1.0
- Query 2: First relevant at rank 3 → RR = 1/3 = 0.33
- Query 3: First relevant at rank 2 → RR = 1/2 = 0.50

MRR = (1.0 + 0.33 + 0.50) / 3 = 0.61

ADVANCED TOPICS

Field-Based Scoring

```
Score = w_title × Score(query, title) +
        w_body × Score(query, body) +
        w_anchor × Score(query, anchor_text)
```

TYPICAL WEIGHTS:

- Title: 3.0-5.0
- Anchor text: 2.0-3.0
- Body: 1.0
- URL: 0.5-1.0

Phrase Queries

```
QUERY: "machine learning"

Must find documents where "machine" and "learning" are ADJACENT

ALGORITHM using Positional Index:
1. Get posting lists with positions
2. For each document containing both terms:
   a. Check if any position of "machine" is followed by "learning"
   b. Position(learning) = Position(machine) + 1
```

Proximity Queries

```
QUERY: machine NEAR/5 learning

Must find documents where "machine" and "learning" are within 5 words

ALGORITHM:
1. Get posting lists with positions
2. For each document:
   a. Check if any position of "machine" and "learning" differ by ≤ 5
```

Query Expansion

```
ORIGINAL QUERY: "car"

EXPANDED QUERY: "car OR automobile OR vehicle"

METHODS:
1. Thesaurus-based: Use synonym dictionary
2. Pseudo-relevance feedback: Add terms from top results
3. Word embeddings: Find similar word vectors
```

Relevance Feedback

```
PROCESS:
1. User issues query
2. System returns initial results
3. User marks relevant/non-relevant documents
4. System learns from feedback and adjusts ranking

ROCCHIO ALGORITHM:
new_query_vector = alpha × query + beta × avg(relevant_docs) - gamma ×
avg(non_relevant_docs)

PARAMETERS: alpha=1.0, beta=0.75, gamma=0.15 (typical)
```

QUICK REFERENCE TABLE

FORMULA	PURPOSE	TYPICAL USE
tf(t, d)	Count term occurrences	Basic frequency
IDF(t) = log(N / df(t))	Measure term rarity	Weight rare terms more
TF-IDF(t, d) = tf × IDF	Basic relevance score	Simple search systems
BM25(q, d)	Advanced relevance score	Production systems
cosine(q, d)	Vector similarity	Similarity matching
Precision	Accuracy of results	Quality metric
Recall	Coverage of results	Completeness metric
P@K	Top-K precision	User-focused metric
MAP	Average precision	Ranking quality
NDCG@K	Graded relevance ranking	Advanced evaluation

FORMULA	PURPOSE	TYPICAL USE
MRR	First result quality	Quick answer systems
DAAT	Document-at-a-time	Simple retrieval
TAAT	Term-at-a-time	Optimized retrieval

ALGORITHM COMPARISON TABLE

ALGORITHM	TIME COMPLEXITY	SPACE COMPLEXITY	ACCURACY	SPEED
Linear Scan	O(N × M)	O(1)	100%	Slow
DAAT	O(K × M × log K)	O(K)	100%	Medium
TAAT	O(M × avg_df)	O(N)	100%	Medium
MaxScore	O(varies)	O(K)	100%	Fast
WAND	O(varies)	O(K + M)	100%	Fast
Block-Max WAND	O(varies)	O(K + M)	100%	Very Fast

WHERE:

- N = number of documents
- M = number of query terms
- K = top-K results to return
- avg_df = average document frequency of query terms

PARAMETER TUNING GUIDE

BM25 Parameters

DOCUMENT TYPE	k1 VALUE	b VALUE	REASONING
Short (tweets)	1.2-1.5	0.3-0.5	Less length variation
Medium (articles)	1.5-2.0	0.75	Standard setting
Long (books)	1.2-1.5	0.8-0.9	Strong length normalization
Mixed lengths	1.5	0.75	Default balanced setting

Query Processing

COLLECTION SIZE	STRATEGY	TOP-K SIZE	PRUNING
< 10K docs	Linear scan	10-100	None
10K - 100K	DAAT or TAAT	10-100	Minimal

COLLECTION SIZE	STRATEGY	TOP-K SIZE	PRUNING
100K - 1M	TAAT with MaxScore	100-1000	Moderate
1M - 10M	WAND	100-1000	Aggressive
> 10M	Block-Max WAND	1000+	Very aggressive

TIPS AND TRICKS

When to Use What

SCORING FUNCTIONS:

- TF-IDF: Simple systems, small collections, baseline
- BM25: Production systems, general-purpose (BEST DEFAULT)
- Cosine Similarity: Document similarity, recommendations
- Learning to Rank: When you have lots of training data
- Neural Models: When semantic understanding is critical

QUERY PROCESSING:

- DAAT: Simple queries (1-3 terms), exact top-K needed
- TAAT: Complex queries (4+ terms), can tolerate approximation
- WAND: Large collections, need speed, exact top-K
- BMW: Very large collections, need maximum speed

INDEX TYPE:

- Basic inverted index: Boolean queries only
- Inverted index with tf: TF-IDF, BM25 scoring
- Positional index: Phrase queries, proximity queries

Common Mistakes to Avoid

1. Forgetting to normalize document length → Use BM25 or normalized TF
2. Not using IDF (treating all words equally) → Always use IDF
3. Ignoring stopwords → Remove or use stopword list
4. Not tuning parameters (k1, b) for your domain → Evaluate and tune
5. Evaluating only on precision → Use multiple metrics (recall, MAP, NDCG)
6. Processing all documents for every query → Use TAAT/DAAT with pruning
7. Not compressing posting lists → Use gap encoding, variable byte
8. Ignoring term position → Use positional index for better ranking
9. One-stage ranking for large collections → Use cascade ranking
10. Fixed IDF values as collection grows → Update regularly

WORKED EXAMPLE: COMPLETE RANKING WITH TAAT

Scenario

- Collection: 1,000 documents
- Query: "information retrieval"
- Using BM25 with k1=1.5, b=0.75, avgdl=500

Statistics

```
Term "information": df=500, IDF=log((1000-500+0.5)/(500+0.5))=0
Term "retrieval": df=50, IDF=log((1000-50+0.5)/(50+0.5))=1.27

Posting list for "information":
  [Doc1(tf=4, len=500), Doc2(tf=2, len=300), Doc5(tf=6, len=800), ...]

Posting list for "retrieval":
  [Doc1(tf=2, len=500), Doc3(tf=3, len=450), Doc5(tf=1, len=800), ...]
```

TAAT Processing

STEP 1: Initialize accumulator

```
A = {}
```

STEP 2: Process term "information" (IDF=0)

```
Since IDF=0, this term contributes 0 to all documents
Skip this term (optimization!)
```

STEP 3: Process term "retrieval" (IDF=1.27)

```
For Doc1 (tf=2, len=500):
  length_norm = 1 - 0.75 + 0.75 × (500/500) = 1.0
  score = 1.27 × (2 × 2.5) / (2 + 1.5 × 1.0) = 1.81
  A[Doc1] = 1.81

For Doc3 (tf=3, len=450):
  length_norm = 1 - 0.75 + 0.75 × (450/500) = 0.925
  score = 1.27 × (3 × 2.5) / (3 + 1.5 × 0.925) = 2.15
  A[Doc3] = 2.15

For Doc5 (tf=1, len=800):
  length_norm = 1 - 0.75 + 0.75 × (800/500) = 1.45
  score = 1.27 × (1 × 2.5) / (1 + 1.5 × 1.45) = 1.00
  A[Doc5] = 1.00
```

STEP 4: Extract top-K from accumulator

```
Sorted: [(Doc3, 2.15), (Doc1, 1.81), (Doc5, 1.00)]
```

FINAL RANKING:

1. Doc3 (2.15) - Highest tf for "retrieval", slightly shorter than average
2. Doc1 (1.81) - Good tf, average length
3. Doc5 (1.00) - Low tf, much longer than average

KEY TAKEAWAYS

SCORING:

1. ALWAYS use IDF - common words should count less
2. ALWAYS normalize for document length - BM25 does this well
3. BM25 is the industry standard - use it as default
4. Tune parameters (k1, b) on your specific data
5. Consider field weighting for structured documents

QUERY PROCESSING: 6. Use TAAT for most production systems 7. Implement WAND or BMW for very large collections 8. Use cascade ranking: fast retrieval → expensive re-ranking 9. Compress posting lists (gap encoding + variable byte) 10. Dynamic pruning can speed up by 10-100x

EVALUATION: 11. Use multiple metrics - not just precision 12. NDCG is best for graded relevance 13. MAP rewards early relevant results 14. P@10 is most user-focused 15. Always use held-out test set

PITFALLS TO AVOID: 16. TF-IDF without length normalization 17. Linear term frequency (use log or saturation) 18. Ignoring term position in documents 19. Processing all documents for every query 20. Not compressing your index

---

END OF COMPLETE CHEATSHEET