Recommendation Systems Deep Dive

Expert-Level Guide for Staff/Principal Interviews

Overview

This guide provides **expert-level** coverage of recommendation systems for engineers with search background who need to refresh on recommendation-specific concepts. It covers modern deep learning approaches, production architecture patterns, and interview preparation for Staff/Principal roles.

Key Differences: Search vs Recommendations

- **Search**: User expresses intent via query → match documents
- **Recommendations**: Predict user preferences without explicit query → discovery
- **Search**: Relevance is primary objective
- **Recommendations**: Balance relevance, diversity, novelty, serendipity
- **Search**: Query-dependent features dominate
- **Recommendations**: User history & collaborative signals dominate

1 Modern Recommendation Architecture

1.1 Industry Standard: Three-Stage Funnel

Stage 1: Candidate Generation (Retrieval)

- **Goal**: Reduce 100M items → 10K candidates (99.99% reduction)
- **Latency budget**: 10-50ms
- **Methods**: Multiple retrieval sources, each contributing candidates

Stage 2: Ranking

- **Goal**: Score 10K candidates → Top 500 items
- **Latency budget**: 50-100ms
- **Model**: Complex ML model (neural network, gradient boosting)

Stage 3: Re-ranking

- **Goal**: Final ordering with diversity, business rules
- **Latency budget**: 10-20ms
- **Methods**: Rule-based, lightweight model, or optimization

1.2 Candidate Generation (Retrieval) in Depth

Unlike search (where query provides strong signal), recommendations need **multiple retrieval sources**:

1. Collaborative Filtering Retrievals

A. User-Based CF

```
Given user u:
1. Find K similar users {u1, u2, ..., uK}
2. Recommend items those users liked
3. Similarity: Cosine(user_vector_u, user_vector_v)
```

B. Item-Based CF (More Stable)

```
Given user u who liked items {i1, i2, ..., im}:
1. For each liked item ij, find K similar items
2. Aggregate and rank candidate items
3. Similarity: Cosine(item_vector_i, item_vector_j)
```

Why Item-Based ¿ User-Based in Production:

- Item similarities change slower (precompute daily)
- User preferences change constantly (require real-time)
- Fewer items than users (smaller index)
- Better cold start for new users

C. Matrix Factorization (SVD, ALS)

Problem formulation:

$$
\begin{aligned}
R &\approx U \times V^T \\
\text{where } R &\in \mathbb{R}^{n\times m} \text{ (user-item interaction matrix)} \\
U &\in \mathbb{R}^{n\times k} \text{ (user embeddings)} \\
V &\in \mathbb{R}^{m\times k} \text{ (item embeddings)}
\end{aligned}
$$

Loss function (explicit feedback):

$$\min_{U,V} \sum_{(u,i)\in\text{observed}} (r_{ui} - u_u^T v_i)^2 + \lambda(||u_u||^2 + ||v_i||^2) \tag{1}$$

Loss function (implicit feedback - BPR):

$$\min_{U,V} \sum_{(u,i,j)} -\log\sigma(u_u^T(v_i - v_j)) + \lambda(||u_u||^2 + ||v_i||^2 + ||v_j||^2) \tag{2}$$

where i is positive item, j is negative item for user u

Training (ALS - Alternating Least Squares):

```
# Alternate between fixing U and optimizing V, and vice versa
for epoch in range(num_epochs):
    # Fix U, optimize V
    for item in items:
        users_who_liked = get_users(item)
        # Closed-form solution for item embedding
        V[item] = solve(U[users_who_liked], R[:, item])

    # Fix V, optimize U
    for user in users:
        items_user_liked = get_items(user)
        # Closed-form solution for user embedding
        U[user] = solve(V[items_user_liked], R[user, :])
```

At serving time:

```
# Retrieve candidates using ANN search
user_embedding = U[user_id]  # (k,)
candidates = ann_index.search(user_embedding, top_k=1000)
# Returns item IDs with highest dot product scores
```

2. Deep Learning Retrievals

A. Two-Tower Neural Network (Google YouTube DNN)

Architecture:

```
User Features              Item Features
(age, gender,              (title, category,
 watch history,             upload time,
 search history)            tags, popularity)
     ↓                           ↓
User Tower (DNN)           Item Tower (DNN)
(3-4 layers, ReLU)         (3-4 layers, ReLU)
     ↓                           ↓
User Embedding             Item Embedding
   (128-dim)                  (128-dim)
         ↓                  ↓
          Dot Product → Score
```

Training:

```
import torch
import torch.nn as nn

class TwoTowerModel(nn.Module):
    def __init__(self, user_features_dim, item_features_dim, embedding_dim):
        super().__init__()

        # User tower
        self.user_tower = nn.Sequential(
            nn.Linear(user_features_dim, 256),
            nn.ReLU(),
            nn.BatchNorm1d(256),
            nn.Dropout(0.3),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, embedding_dim)
        )

        # Item tower
        self.item_tower = nn.Sequential(
            nn.Linear(item_features_dim, 256),
            nn.ReLU(),
            nn.BatchNorm1d(256),
            nn.Dropout(0.3),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, embedding_dim)
        )

    def forward(self, user_features, item_features):
        user_emb = self.user_tower(user_features)  # (batch, embedding_dim)
        item_emb = self.item_tower(item_features)  # (batch, embedding_dim)

        # Normalize embeddings (important for dot product!)
        user_emb = F.normalize(user_emb, p=2, dim=1)
        item_emb = F.normalize(item_emb, p=2, dim=1)
```

```
        # Dot product score
        score = torch.sum(user_emb * item_emb, dim=1)
        return score, user_emb, item_emb

# Training with sampled softmax
def sampled_softmax_loss(model, user_features, pos_item_features,
                         neg_item_features_list):
    """
    user_features: (batch, user_feat_dim)
    pos_item_features: (batch, item_feat_dim) - positive items
    neg_item_features_list: list of (batch, item_feat_dim) - negative samples
    """
    # Positive score
    pos_score, user_emb, pos_item_emb = model(user_features, pos_item_features)

    # Negative scores
    neg_scores = []
    for neg_item_features in neg_item_features_list:
        neg_score, _, _ = model(user_features, neg_item_features)
        neg_scores.append(neg_score)

    # Concatenate: [pos_score, neg_score1, neg_score2, ...]
    all_scores = torch.stack([pos_score] + neg_scores, dim=1)  # (batch, 1+num_neg)

    # Labels: positive is at index 0
    labels = torch.zeros(all_scores.size(0), dtype=torch.long).to(all_scores.device)

    # Softmax cross-entropy
    loss = F.cross_entropy(all_scores, labels)
    return loss
```

Negative Sampling Strategies:

1. **Random sampling**: Sample uniformly from item catalog
2. **Popularity-based**: Sample proportional to popularity$^{0.75}$ (word2vec trick)
3. **Hard negative mining**: Sample items user didn't click but were shown
4. **Batch negative**: Use other positives in batch as negatives (efficient!)

Batch Negative Example:

```
# In-batch negatives (used by Pinterest, Alibaba)
def batch_negative_loss(user_emb, item_emb):
    """
    user_emb: (batch_size, dim)
    item_emb: (batch_size, dim) - each is positive for corresponding user
    """
    # Score matrix: (batch_size, batch_size)
    scores = torch.matmul(user_emb, item_emb.T)

    # Diagonal elements are positive pairs
    # Off-diagonal are negatives
    labels = torch.arange(scores.size(0)).to(scores.device)

    loss = F.cross_entropy(scores, labels)
    return loss
```

At Serving Time:

```
# Build ANN index for all items (offline, daily)
item_embeddings = []
for item_batch in all_items:
    item_features = get_item_features(item_batch)
```

```
    _, item_emb = item_tower(item_features)
    item_embeddings.append(item_emb)

item_embeddings = torch.cat(item_embeddings)  # (num_items, embedding_dim)

# Store in FAISS
import faiss
index = faiss.IndexFlatIP(embedding_dim)  # Inner product (dot product)
index.add(item_embeddings.cpu().numpy())

# At serving time (real-time)
user_features = get_user_features(user_id)
user_emb, _ = user_tower(user_features)

# ANN search
D, I = index.search(user_emb.cpu().numpy(), k=1000)
# I contains item IDs, D contains scores
```

B. Sequential Models (GRU4Rec, SASRec, BERT4Rec)

Use case: Capture sequential patterns in user behavior

GRU4Rec (Session-based):

```
class GRU4Rec(nn.Module):
    def __init__(self, num_items, embedding_dim, hidden_dim):
        super().__init__()
        self.item_embedding = nn.Embedding(num_items, embedding_dim)
        self.gru = nn.GRU(embedding_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, num_items)

    def forward(self, item_sequence):
        """
        item_sequence: (batch, seq_len) - item IDs in session
        """
        # Embed items
        embedded = self.item_embedding(item_sequence)  # (batch, seq_len, emb_dim)

        # GRU
        output, hidden = self.gru(embedded)  # output: (batch, seq_len, hidden_dim)

        # Predict next item from last timestep
        logits = self.fc(output[:, -1, :])  # (batch, num_items)

        return logits

# Training
model = GRU4Rec(num_items=100000, embedding_dim=128, hidden_dim=256)

for session in sessions:
    # session = [item1, item2, item3, item4, item5]
    input_seq = session[:-1]  # [item1, item2, item3, item4]
    target = session[-1]       # item5

    logits = model(input_seq)
    loss = F.cross_entropy(logits, target)
    loss.backward()
    optimizer.step()
```

SASRec (Self-Attention for Sequential Recommendation):

```
class SASRec(nn.Module):
    def __init__(self, num_items, embedding_dim, num_heads, num_layers):
        super().__init__()
        self.item_embedding = nn.Embedding(num_items, embedding_dim)
        self.pos_embedding = nn.Embedding(200, embedding_dim)  # Max seq length
```

```
        # Transformer encoder
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=embedding_dim,
            nhead=num_heads,
            dim_feedforward=embedding_dim*4
        )
        self.transformer = nn.TransformerEncoder(encoder_layer, num_layers)

        self.fc = nn.Linear(embedding_dim, num_items)

    def forward(self, item_sequence):
        """
        item_sequence: (batch, seq_len)
        """
        batch_size, seq_len = item_sequence.size()

        # Item embeddings
        item_emb = self.item_embedding(item_sequence)  # (batch, seq_len, dim)

        # Positional embeddings
        positions = torch.arange(seq_len).unsqueeze(0).expand(batch_size, -1).to(
    item_sequence.device)
        pos_emb = self.pos_embedding(positions)  # (batch, seq_len, dim)

        # Combine
        x = item_emb + pos_emb

        # Transformer (with causal masking for autoregressive)
        mask = nn.Transformer.generate_square_subsequent_mask(seq_len).to(item_sequence.
    device)
        x = x.transpose(0, 1)  # (seq_len, batch, dim)
        output = self.transformer(x, mask=mask)
        output = output.transpose(0, 1)  # (batch, seq_len, dim)

        # Predict next item
        logits = self.fc(output[:, -1, :])  # (batch, num_items)

        return logits
```

3. Content-Based Retrieval
Good for cold start (new items):

```
# For new item (no collaborative signal yet)
# 1. Extract content features
item_features = {
    'title_embedding': bert_encode(item.title),  # (768,)
    'category': item.category,
    'tags': item.tags,
    'description_embedding': bert_encode(item.description)
}

# 2. Find similar items in catalog
similar_items = faiss_index.search(item_features['title_embedding'], k=100)

# 3. Retrieve users who liked those similar items
candidate_users = get_users_who_liked(similar_items)
```

4. Graph-Based Retrieval (Pinterest, Alibaba)
Item2Vec / Node2Vec:

```
# Build item-item co-occurrence graph
# Edge weight = # of users who interacted with both items
```

```python
import networkx as nx
from node2vec import Node2Vec

# Create graph
G = nx.Graph()
for user in users:
    items = user.interacted_items
    # Add edges between co-occurring items
    for i in range(len(items)):
        for j in range(i+1, len(items)):
            if G.has_edge(items[i], items[j]):
                G[items[i]][items[j]]['weight'] += 1
            else:
                G.add_edge(items[i], items[j], weight=1)

# Node2Vec random walks
node2vec = Node2Vec(G, dimensions=128, walk_length=80, num_walks=10)
model = node2vec.fit(window=10, min_count=1)

# Get item embeddings
item_embedding = model.wv[item_id]

# Retrieve similar items
similar_items = model.wv.most_similar(item_id, topn=100)
```

5. **Blending Multiple Retrieval Sources**

```python
def retrieve_candidates(user_id, top_k=1000):
    """Blend multiple retrieval sources"""

    # Source 1: Collaborative filtering (user-based)
    cf_candidates = cf_retrieval(user_id, top_k=300)

    # Source 2: Two-tower model
    two_tower_candidates = two_tower_retrieval(user_id, top_k=300)

    # Source 3: Sequential model (if user has recent session)
    if has_recent_session(user_id):
        seq_candidates = sequential_retrieval(user_id, top_k=200)
    else:
        seq_candidates = []

    # Source 4: Trending items (exploration)
    trending_candidates = get_trending_items(top_k=100)

    # Source 5: Content-based (for diversity)
    content_candidates = content_based_retrieval(user_id, top_k=100)

    # Combine and deduplicate
    all_candidates = deduplicate([
        cf_candidates,
        two_tower_candidates,
        seq_candidates,
        trending_candidates,
        content_candidates
    ])

    # Return top K by combined score
    return rank_and_select(all_candidates, top_k=top_k)
```

1.3 Ranking Stage

Goal: Precisely score candidates using rich features & complex models

Modern Ranking Architectures:

1. Deep & Wide (Google Play)

Architecture:

```
Wide Component                   Deep Component
(Cross features,                (Embeddings + DNN,
 manual features)                learned features)
      ↓                                ↓
  Linear Model                      Deep NN
      ↓                                ↓
         Combined (concatenate) → Sigmoid → CTR
```

Implementation:

```
class DeepAndWide(nn.Module):
    def __init__(self, wide_dim, deep_dim, embedding_dims):
        super().__init__()

        # Wide component (linear)
        self.wide = nn.Linear(wide_dim, 1)

        # Deep component (DNN)
        # Embeddings for categorical features
        self.embeddings = nn.ModuleList([
            nn.Embedding(num_categories, emb_dim)
            for num_categories, emb_dim in embedding_dims
        ])

        # Deep network
        total_emb_dim = sum([emb_dim for _, emb_dim in embedding_dims])
        self.deep = nn.Sequential(
            nn.Linear(total_emb_dim + deep_dim, 256),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 1)
        )

    def forward(self, wide_features, categorical_features, deep_features):
        # Wide path
        wide_out = self.wide(wide_features)  # (batch, 1)

        # Embed categorical features
        embeddings = [emb(cat_feat) for emb, cat_feat in
                      zip(self.embeddings, categorical_features)]
        embeddings = torch.cat(embeddings, dim=1)  # (batch, total_emb_dim)

        # Deep path
        deep_input = torch.cat([embeddings, deep_features], dim=1)
        deep_out = self.deep(deep_input)  # (batch, 1)

        # Combine
        logits = wide_out + deep_out
        output = torch.sigmoid(logits)

        return output
```

2. DeepFM (Huawei, Criteo)

Key innovation: Replace manual wide features with FM (Factorization Machine)

Architecture:

```
Features → Embeddings → FM Component (2-way interactions)
                         ↓
                         DNN Component (high-order interactions)
                         ↓
                Combined → Sigmoid → CTR
```

FM Component:

$$y_{FM} = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n}\sum_{j=i+1}^{n} \langle v_i, v_j \rangle x_i x_j \tag{3}$$

Implementation:

```
class DeepFM(nn.Module):
    def __init__(self, feature_sizes, embedding_dim=16):
        super().__init__()

        # Embeddings (shared between FM and DNN)
        self.embeddings = nn.ModuleList([
            nn.Embedding(feat_size, embedding_dim)
            for feat_size in feature_sizes
        ])

        # FM: first-order weights
        self.fm_first_order = nn.ModuleList([
            nn.Embedding(feat_size, 1)
            for feat_size in feature_sizes
        ])

        # DNN
        total_embedding_dim = len(feature_sizes) * embedding_dim
        self.dnn = nn.Sequential(
            nn.Linear(total_embedding_dim, 256),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 1)
        )

        self.bias = nn.Parameter(torch.zeros(1))

    def forward(self, categorical_features):
        """
        categorical_features: list of (batch,) tensors, one per feature field
        """
        # Get embeddings
        embeddings = [emb(feat) for emb, feat in
                      zip(self.embeddings, categorical_features)]
        # Each: (batch, embedding_dim)

        # FM first-order
        first_order = sum([
            emb(feat).squeeze(1) for emb, feat in
            zip(self.fm_first_order, categorical_features)
        ])  # (batch,)

        # FM second-order (pairwise interactions)
        # Efficient computation: sum of squares - square of sum
        sum_of_embeddings = sum(embeddings)  # (batch, embedding_dim)
        square_of_sum = torch.pow(sum_of_embeddings, 2)  # (batch, embedding_dim)
```

```
        sum_of_squares = sum([torch.pow(emb, 2) for emb in embeddings])  # (batch,
    embedding_dim)

        second_order = 0.5 * torch.sum(square_of_sum - sum_of_squares, dim=1)  # (batch,)

        # DNN
        dnn_input = torch.cat(embeddings, dim=1)  # (batch, total_embedding_dim)
        dnn_out = self.dnn(dnn_input).squeeze(1)  # (batch,)

        # Combine
        logits = self.bias + first_order + second_order + dnn_out
        output = torch.sigmoid(logits)

        return output
```

3. Multi-Task Learning (YouTube, Facebook)
Motivation: Optimize for multiple objectives simultaneously
Example objectives:

- Click probability (engagement)
- Watch time (quality engagement)
- Like probability
- Share probability
- Conversion probability (purchase, subscribe)

Architecture:

```
class MultiTaskRanking(nn.Module):
    def __init__(self, input_dim):
        super().__init__()

        # Shared bottom layers
        self.shared = nn.Sequential(
            nn.Linear(input_dim, 256),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(256, 128),
            nn.ReLU()
        )

        # Task-specific towers
        self.click_tower = nn.Sequential(
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 1),
            nn.Sigmoid()
        )

        self.watch_time_tower = nn.Sequential(
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 1)  # Regression, no sigmoid
        )

        self.like_tower = nn.Sequential(
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 1),
            nn.Sigmoid()
        )
```

```python
    def forward(self, features):
        shared_repr = self.shared(features)  # (batch, 128)

        p_click = self.click_tower(shared_repr)  # (batch, 1)
        watch_time = self.watch_time_tower(shared_repr)  # (batch, 1)
        p_like = self.like_tower(shared_repr)  # (batch, 1)

        return {
            'p_click': p_click,
            'watch_time': watch_time,
            'p_like': p_like
        }

# Training
def multi_task_loss(outputs, labels, weights={'click': 1.0, 'watch_time': 0.5, 'like':
    0.3}):
    loss_click = F.binary_cross_entropy(outputs['p_click'], labels['click'])
    loss_watch_time = F.mse_loss(outputs['watch_time'], labels['watch_time'])
    loss_like = F.binary_cross_entropy(outputs['p_like'], labels['like'])

    total_loss = (weights['click'] * loss_click +
                  weights['watch_time'] * loss_watch_time +
                  weights['like'] * loss_like)

    return total_loss

# Final ranking score
def compute_score(outputs):
    # Weighted combination
    score = (0.4 * outputs['p_click'] +
             0.4 * outputs['watch_time'] / 300.0 +  # Normalize watch time
             0.2 * outputs['p_like'])
    return score
```

1.4 Re-ranking Stage

Goals:

1. **Diversity**: Avoid showing too many similar items
2. **Freshness**: Boost recent content
3. **Exploration**: Include some random items for discovery
4. **Business rules**: Position constraints, deduplication

1. Maximal Marginal Relevance (MMR)
Formula:

$$\text{MMR} = \arg\max_{d_i \in R \setminus S} [\lambda \cdot \text{Relevance}(d_i) - (1 - \lambda) \cdot \max_{d_j \in S} \text{Similarity}(d_i, d_j)] \tag{4}$$

Implementation:

```python
def mmr_rerank(ranked_items, scores, similarity_matrix, lambda_param=0.7, top_k=50):
    """
    ranked_items: List of item IDs sorted by score
    scores: Dict mapping item_id -> score
    similarity_matrix: item_i, item_j -> similarity [0, 1]
    lambda_param: Trade-off between relevance and diversity
    """
    selected = []
    remaining = ranked_items.copy()
```

```
    # First item: highest score
    first_item = remaining.pop(0)
    selected.append(first_item)

    while len(selected) < top_k and remaining:
        mmr_scores = []

        for item in remaining:
            # Relevance component
            relevance = scores[item]

            # Diversity component (similarity to already selected)
            max_similarity = max([
                similarity_matrix[item][selected_item]
                for selected_item in selected
            ])

            # MMR score
            mmr = lambda_param * relevance - (1 - lambda_param) * max_similarity
            mmr_scores.append((item, mmr))

        # Select item with highest MMR
        best_item = max(mmr_scores, key=lambda x: x[1])[0]
        selected.append(best_item)
        remaining.remove(best_item)

    return selected
```

2. Determinantal Point Process (DPP)
Used by Google, Twitter for diversity-aware ranking.
Idea: Model subset selection with diversity kernel

```
import numpy as np
from dppy.finite_dpps import FiniteDPP

def dpp_rerank(items, scores, similarity_matrix, k=50):
    """
    Use DPP to select diverse subset
    """
    n = len(items)

    # Build DPP kernel: L = quality * diversity
    # Quality diagonal matrix
    quality = np.diag([scores[item] for item in items])

    # Diversity matrix (inverse of similarity)
    diversity = 1 - similarity_matrix

    # DPP kernel
    L = quality @ diversity @ quality

    # Sample from DPP
    dpp = FiniteDPP('likelihood', **{'L': L})
    dpp.sample_exact()

    selected_indices = dpp.list_of_samples[-1]
    selected_items = [items[i] for i in selected_indices[:k]]

    return selected_items
```

3. Exploration (Multi-Armed Bandits)
Epsilon-Greedy:

```
def epsilon_greedy_rerank(ranked_items, epsilon=0.1, top_k=50):
```

```python
    """
    With probability epsilon, inject random items
    """
    import random

    result = []
    for i in range(top_k):
        if random.random() < epsilon:
            # Explore: random item from catalog
            result.append(random.choice(item_catalog))
        else:
            # Exploit: use ranked item
            if i < len(ranked_items):
                result.append(ranked_items[i])

    return result
```

Thompson Sampling (Contextual Bandit):

```python
# Used by Netflix for personalized thumbnails
class ThompsonSampling:
    def __init__(self, num_items):
        # Beta distribution parameters for each item
        self.alpha = np.ones(num_items)  # Successes (clicks)
        self.beta = np.ones(num_items)   # Failures (no clicks)

    def select_item(self, candidate_items):
        # Sample from Beta distribution for each candidate
        samples = [
            np.random.beta(self.alpha[item], self.beta[item])
            for item in candidate_items
        ]

        # Select item with highest sample
        best_idx = np.argmax(samples)
        return candidate_items[best_idx]

    def update(self, item, clicked):
        # Update distribution based on feedback
        if clicked:
            self.alpha[item] += 1
        else:
            self.beta[item] += 1
```

1.5 Multi-Objective Optimization

At Staff/Principal level, you're expected to balance multiple business objectives, not just maximize a single metric.

The Real-World Problem:

Recommendation systems have **conflicting objectives**:

- **User engagement**: Click-through rate, watch time
- **Business revenue**: Ad clicks, subscription conversions
- **Content diversity**: Don't show only viral content
- **Creator satisfaction**: Fair distribution of impressions
- **Platform health**: Reduce misinformation, toxic content

Naive approach: Single weighted objective $L = w_1 \cdot \text{CTR} + w_2 \cdot \text{Revenue}$

Problem: Hard to tune weights, doesn't show trade-offs

1. Pareto Frontier & Scalarization

Pareto optimality: A solution where you can't improve one objective without hurting another

Weighted sum approach:

$$L = \sum_{i=1}^{n} w_i \cdot f_i(x), \quad \sum w_i = 1 \tag{5}$$

Implementation:

```python
import numpy as np

class MultiObjectiveRanker:
    def __init__(self, objectives, weights):
        """
        objectives: List of objective functions (higher is better)
        weights: List of weights for each objective
        """
        self.objectives = objectives
        self.weights = np.array(weights)
        assert abs(self.weights.sum() - 1.0) < 1e-6  # Must sum to 1

    def score(self, item, user_context):
        """Compute weighted score for item"""
        scores = np.array([
            obj(item, user_context) for obj in self.objectives
        ])
        # Normalize to [0, 1] before combining
        scores_normalized = (scores - scores.min()) / (scores.max() - scores.min() + 1e
   -8)
        return np.dot(self.weights, scores_normalized)

    def rank(self, items, user_context):
        """Rank items by multi-objective score"""
        scores = [(item, self.score(item, user_context)) for item in items]
        ranked = sorted(scores, key=lambda x: x[1], reverse=True)
        return [item for item, score in ranked]

# Example: Balance CTR and revenue
def ctr_objective(item, user_context):
    # Predict click probability
    return model_ctr.predict_proba(item, user_context)

def revenue_objective(item, user_context):
    # Predict expected revenue
    return item.price * model_conversion.predict_proba(item, user_context)

def diversity_objective(item, user_context):
    # Penalize similarity to recently shown items
    recent_items = user_context['recent_items']
    avg_similarity = np.mean([similarity(item, rec) for rec in recent_items])
    return 1 - avg_similarity  # Higher score for diverse items

# Create multi-objective ranker
ranker = MultiObjectiveRanker(
    objectives=[ctr_objective, revenue_objective, diversity_objective],
    weights=[0.5, 0.3, 0.2]  # 50% CTR, 30% revenue, 20% diversity
)

ranked_items = ranker.rank(candidate_items, user_context)
```

Interview talking point: "I'd start with simple weighted sum, then A/B test different weight combinations to find Pareto optimal trade-off"

2. Pareto Frontier Exploration

Instead of picking one weight vector, explore the Pareto frontier:

```
def compute_pareto_frontier(items, objectives, num_points=10):
    """
    Compute Pareto frontier by varying weights
    Returns: List of (weights, pareto_optimal_items, objective_values)
    """
    pareto_frontier = []

    # Grid search over weight space
    for alpha in np.linspace(0, 1, num_points):
        weights = [alpha, 1 - alpha]  # For 2 objectives

        # Rank items with these weights
        ranker = MultiObjectiveRanker(objectives, weights)
        ranked = ranker.rank(items, user_context)

        # Evaluate objectives on top-k
        top_k = ranked[:50]
        obj_values = [
            np.mean([obj(item, user_context) for item in top_k])
            for obj in objectives
        ]

        pareto_frontier.append({
            'weights': weights,
            'items': top_k,
            'ctr': obj_values[0],
            'revenue': obj_values[1]
        })

    return pareto_frontier

# Visualize trade-off
import matplotlib.pyplot as plt

frontier = compute_pareto_frontier(items, [ctr_objective, revenue_objective])

ctrs = [p['ctr'] for p in frontier]
revenues = [p['revenue'] for p in frontier]

plt.plot(ctrs, revenues, 'o-')
plt.xlabel('CTR')
plt.ylabel('Revenue')
plt.title('Pareto Frontier: CTR vs Revenue Trade-off')
plt.show()
```

Use in interviews: Show you understand trade-offs, not just maximize single metric

3. Multi-Task Learning Approach

Instead of post-hoc weighted combination, train a single model to predict multiple objectives:

```
import torch
import torch.nn as nn

class MultiTaskRankingModel(nn.Module):
    """
    Shared bottom + task-specific towers
    Used by YouTube, Alibaba for multi-objective ranking
    """
    def __init__(self, input_dim, shared_dim, task_dims):
        super(MultiTaskRankingModel, self).__init__()

        # Shared bottom layers (feature extraction)
        self.shared = nn.Sequential(
            nn.Linear(input_dim, 512),
```

```
            nn.ReLU(),
            nn.Linear(512, shared_dim),
            nn.ReLU()
        )

        # Task-specific towers
        self.ctr_tower = nn.Sequential(
            nn.Linear(shared_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 1),
            nn.Sigmoid()  # CTR prediction
        )

        self.revenue_tower = nn.Sequential(
            nn.Linear(shared_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 1)  # Revenue prediction (regression)
        )

        self.engagement_tower = nn.Sequential(
            nn.Linear(shared_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 1)  # Watch time prediction
        )

    def forward(self, x):
        # Shared representation
        shared_repr = self.shared(x)

        # Task-specific predictions
        ctr = self.ctr_tower(shared_repr)
        revenue = self.revenue_tower(shared_repr)
        engagement = self.engagement_tower(shared_repr)

        return {
            'ctr': ctr,
            'revenue': revenue,
            'engagement': engagement
        }

# Training with multi-task loss
def multi_task_loss(predictions, labels, weights):
    """
    predictions: Dict of task predictions
    labels: Dict of task labels
    weights: Dict of task weights
    """
    ctr_loss = nn.BCELoss()(predictions['ctr'], labels['ctr'])
    revenue_loss = nn.MSELoss()(predictions['revenue'], labels['revenue'])
    engagement_loss = nn.MSELoss()(predictions['engagement'], labels['engagement'])

    total_loss = (
        weights['ctr'] * ctr_loss +
        weights['revenue'] * revenue_loss +
        weights['engagement'] * engagement_loss
    )

    return total_loss, {
        'ctr_loss': ctr_loss.item(),
        'revenue_loss': revenue_loss.item(),
        'engagement_loss': engagement_loss.item()
    }
```

```python
# At serving time, combine predictions
def multi_objective_score(predictions, weights):
    """Combine multi-task predictions into final score"""
    score = (
        weights['ctr'] * predictions['ctr'] +
        weights['revenue'] * predictions['revenue'] / 100 +  # Normalize
        weights['engagement'] * predictions['engagement'] / 3600  # Normalize
    )
    return score
```

Advantages over weighted sum:

- Shared representation learns features useful for all tasks
- Can adjust weights at serving time without retraining
- Captures task correlations (positive transfer learning)

4. Constraint-Based Optimization
Instead of soft weighting, enforce hard constraints:
Example: Maximize engagement subject to revenue constraint

$$\begin{aligned} \max_{x} \quad & \text{Engagement}(x) \\ \text{s.t.} \quad & \text{Revenue}(x) \geq R_{\min} \\ & \text{Diversity}(x) \geq D_{\min} \end{aligned} \tag{6}$$

Implementation (greedy with constraints):

```python
def constrained_ranking(items, user_context, min_revenue=100, min_diversity=0.5, k=50):
    """
    Maximize engagement subject to revenue and diversity constraints
    """
    ranked = []
    remaining = items.copy()

    current_revenue = 0
    shown_categories = set()

    while len(ranked) < k and remaining:
        # Find best item that satisfies constraints
        best_item = None
        best_engagement = -float('inf')

        for item in remaining:
            # Check revenue constraint
            item_revenue = revenue_objective(item, user_context)
            projected_avg_revenue = (current_revenue + item_revenue) / (len(ranked) + 1)

            if projected_avg_revenue < min_revenue:
                continue  # Violates revenue constraint

            # Check diversity constraint
            shown_categories.add(item.category)
            diversity_score = len(shown_categories) / len(ranked + 1)

            if diversity_score < min_diversity:
                shown_categories.remove(item.category)  # Rollback
                continue  # Violates diversity constraint

            # If constraints satisfied, check engagement
            engagement = engagement_objective(item, user_context)
            if engagement > best_engagement:
```

```
                best_engagement = engagement
                best_item = item

        if best_item is None:
            break  # No item satisfies constraints

        ranked.append(best_item)
        remaining.remove(best_item)
        current_revenue += revenue_objective(best_item, user_context)

    return ranked
```

5. Business Trade-off Discussion (Interview Gold)
Scenario: "YouTube wants to maximize watch time, but advertisers want more ad impressions"
Staff/Principal answer:

1. **Define metrics**:
    - User objective: Total watch time (hours/day/user)
    - Business objective: Ad revenue ($/user/day)
    - Constraint: User retention rate must not drop
2. **Current state**: Measure baseline (e.g., 30 min watch time, $0.50 revenue/user)
3. **Pareto analysis**:
    - Test 5 weight combinations: $(w_{\text{watch}}, w_{\text{revenue}}) = (1.0, 0.0), (0.7, 0.3), (0.5, 0.5), (0.3, 0.7), (0.0, 1.0)$
    - Plot Pareto frontier
    - Identify knee of curve (best trade-off)
4. **A/B test**:
    - Implement top 3 weight configurations
    - Run for 2 weeks with 1% traffic each
    - Monitor: Watch time, revenue, retention, user complaints
5. **Long-term monitoring**:
    - Track Pareto frontier over time (shifts with user behavior)
    - Quarterly re-optimization
    - Detect if we've moved off optimal frontier
6. **Stakeholder communication**:
    - Show trade-off curve to product team
    - Quantify: "10% more revenue costs 5% watch time"
    - Let business decide acceptable trade-off

Key Interview Talking Points:
When discussing recommendations, always mention multi-objective optimization:

1. **"Let's think about conflicting objectives..."** → Shows strategic thinking
2. **"I'd compute the Pareto frontier..."** → Demonstrates you understand trade-offs
3. **"Use multi-task learning with shared bottom..."** → Modern ML approach (YouTube, Alibaba)
4. **"Monitor multiple metrics, not just optimize one"** → Production mindset
5. **"Present trade-off curve to stakeholders"** → Staff/Principal communication skill

Common Multi-Objective Scenarios:

System	Objective 1	Objective 2
YouTube	Watch time	Ad revenue
LinkedIn	Engagement	Job applications
Spotify	Listening time	Subscription conversions
TikTok	User engagement	Creator satisfaction
Amazon	Click-through rate	Purchase revenue

This shows you've worked on real-world systems, not just textbook single-objective optimization.

2 Feature Engineering for Recommendations

Feature categories:

1. User Features

- **Demographics**: Age, gender, location, language
- **Historical behavior**:
  - # clicks/watches in last 1h, 24h, 7d, 30d
  - # purchases, avg basket size
  - Favorite categories (top 3-5)
- **Temporal patterns**:
  - Active hours (morning/afternoon/evening)
  - Weekday vs weekend behavior
- **Session context**: Device, platform, current session length

2. Item Features

- **Content**: Title, description, category, tags, metadata
- **Popularity**:
  - View count (1h, 24h, 7d, all-time)
  - CTR, conversion rate
  - Trending score
- **Quality**: User ratings, reviews, likes/dislikes ratio
- **Freshness**: Upload time, time since publish

3. User-Item Cross Features

- **Historical interaction**:
  - Has user viewed this item before?
  - Has user viewed similar items?
  - User's affinity to item category (CTR for category)
- **Similarity scores**:
  - Cosine similarity (user embedding, item embedding)
  - Category match score

4. Context Features

- **Time**: Hour of day, day of week, holiday/weekend
- **Device**: Mobile/desktop, OS, browser

- **Location**: City, country, timezone
- **Session**: Pages viewed in session, time spent

Feature Engineering Code:

```
import pandas as pd
import numpy as np

def engineer_features(user_id, item_id, context):
    features = {}

    # User features
    user_history = get_user_history(user_id, lookback_days=30)
    features['user_click_count_1h'] = count_clicks(user_history, hours=1)
    features['user_click_count_24h'] = count_clicks(user_history, hours=24)
    features['user_click_count_7d'] = count_clicks(user_history, days=7)
    features['user_avg_session_time'] = user_history['session_time'].mean()
    features['user_favorite_categories'] = get_top_categories(user_history, top_k=3)

    # Item features
    item_stats = get_item_stats(item_id)
    features['item_view_count_24h'] = item_stats['views_24h']
    features['item_ctr'] = item_stats['clicks'] / max(item_stats['impressions'], 1)
    features['item_age_hours'] = (datetime.now() - item_stats['publish_time']).
   total_seconds() / 3600
    features['item_category'] = item_stats['category']

    # Cross features
    features['user_item_category_affinity'] = get_category_affinity(user_id, item_stats['
   category'])
    features['user_viewed_before'] = item_id in user_history['item_ids']

    # User-item embedding similarity
    user_emb = get_user_embedding(user_id)
    item_emb = get_item_embedding(item_id)
    features['embedding_similarity'] = cosine_similarity(user_emb, item_emb)

    # Context features
    features['hour_of_day'] = context['timestamp'].hour
    features['is_weekend'] = context['timestamp'].weekday() >= 5
    features['device_type'] = context['device']

    return features
```

3 Cold Start Solutions

Problem: New users/items have no collaborative signals

1. New User Cold Start

Approach A: Onboarding Survey

- Ask user to select interests/preferences
- Show popular items from selected categories
- Netflix, Spotify use this approach

Approach B: Demographic-Based

- Use age, gender, location to find similar users
- Recommend what similar users like

Approach C: Explore Popular Items

- Show trending/popular items
- Collect feedback quickly to build profile

```
def recommend_for_new_user(user_id, user_demographics):
    """Cold start recommendation for new user"""

    if has_onboarding_preferences(user_id):
        # Use stated preferences
        prefs = get_onboarding_preferences(user_id)
        candidates = get_items_by_categories(prefs['categories'], top_k=500)
    else:
        # Use demographics to find similar users
        similar_users = find_similar_users_by_demographics(user_demographics, top_k=100)

        # Aggregate their liked items
        candidates = aggregate_user_likes(similar_users, top_k=500)

        # Also add trending items (exploration)
        trending = get_trending_items(top_k=100)
        candidates.extend(trending)

    # Rank by popularity (no personalization signal yet)
    ranked = rank_by_popularity(candidates)

    return ranked[:50]
```

2. New Item Cold Start
Approach A: Content-Based Bootstrap

- Use item content (title, description, category)
- Find similar existing items
- Recommend to users who liked similar items

```
def recommend_new_item(item_id):
    """Bootstrap recommendations for new item"""

    # Get item content features
    item_features = get_item_content(item_id)

    # Find similar items using content
    similar_items = find_similar_items_by_content(
        item_features,
        top_k=100
    )

    # Get users who liked similar items
    target_users = []
    for similar_item in similar_items:
        users = get_users_who_liked(similar_item)
        target_users.extend(users)

    # Deduplicate and rank by engagement level
    target_users = deduplicate_and_rank(target_users)

    return target_users[:1000]  # Show to these users
```

Approach B: Creator-Based

- If from known creator, show to creator's followers

- YouTube, TikTok use this

Approach C: Controlled Exploration

- Show to small random sample (5-10%)
- Collect engagement data
- Bootstrap collaborative signal

4 Evaluation Metrics

4.1 Offline Metrics

1. Ranking Metrics
NDCG@K (Normalized Discounted Cumulative Gain):

$$\text{DCG@K} = \sum_{i=1}^{K} \frac{2^{rel_i} - 1}{\log_2(i+1)} \tag{7}$$

$$\text{NDCG@K} = \frac{\text{DCG@K}}{\text{IDCG@K}} \tag{8}$$

```
def ndcg_at_k(relevance_scores, k):
    """
    relevance_scores: List of relevance scores in ranked order
    Higher is better (e.g., [5, 3, 2, 0, 1, ...])
    """
    def dcg_at_k(scores, k):
        scores = np.array(scores)[:k]
        gains = 2**scores - 1
        discounts = np.log2(np.arange(2, len(scores) + 2))
        return np.sum(gains / discounts)

    dcg = dcg_at_k(relevance_scores, k)

    # Ideal DCG (best possible ordering)
    ideal_scores = sorted(relevance_scores, reverse=True)
    idcg = dcg_at_k(ideal_scores, k)

    if idcg == 0:
        return 0.0

    return dcg / idcg

# Example
relevance = [3, 2, 3, 0, 1, 2]  # User clicked items at positions 0, 1, 2, 5
print(f"NDCG@5: {ndcg_at_k(relevance, 5):.4f}")
```

2. Diversity Metrics
Intra-List Similarity (ILS):

$$\text{ILS} = \frac{2}{K(K-1)} \sum_{i=1}^{K} \sum_{j=i+1}^{K} \text{sim}(item_i, item_j) \tag{9}$$

Lower ILS = More diverse list

```
def intra_list_similarity(recommended_items, similarity_matrix):
    """
    Measure diversity of recommendation list
    """
    k = len(recommended_items)
```

```
    if k < 2:
        return 0.0

    total_similarity = 0
    count = 0

    for i in range(k):
        for j in range(i+1, k):
            total_similarity += similarity_matrix[recommended_items[i]][recommended_items
[j]]
            count += 1

    return total_similarity / count
```

3. Coverage Metrics
Catalog Coverage:

$$\text{Coverage} = \frac{\#\text{ unique items recommended}}{\text{Total }\#\text{ items in catalog}} \tag{10}$$

```
def catalog_coverage(all_recommendations, catalog_size):
    """
    all_recommendations: List of lists, recommendations for each user
    """
    unique_items = set()
    for user_recs in all_recommendations:
        unique_items.update(user_recs)

    return len(unique_items) / catalog_size
```

4.2 Online Metrics

Primary Business Metrics:

- **CTR**: Clicks / Impressions
- **Engagement time**: Total watch time, time on site
- **Conversion rate**: Purchases / Clicks
- **Revenue per user**: Total revenue / Active users

User Retention:

- **DAU / MAU**: Daily Active Users / Monthly Active Users
- **Session frequency**: Sessions per user per week
- **Churn rate**: % users who stop using

Guardrail Metrics:

- **Latency**: p95, p99 recommendation latency
- **Error rate**: % failed recommendations
- **Diversity**: Avg unique categories per user
- **Freshness**: % items published in last 24h

5 A/B Testing for Recommendations

Setup:

```python
import hashlib

def assign_experiment_group(user_id, experiment_name, num_groups=2):
    """Deterministic assignment to experiment groups"""
    hash_input = f"{experiment_name}:{user_id}"
    hash_value = int(hashlib.md5(hash_input.encode()).hexdigest(), 16)
    return hash_value % num_groups

# Usage
user_id = "12345"
group = assign_experiment_group(user_id, "two_tower_vs_cf_2024_01")

if group == 0:
    # Control: Collaborative filtering
    recommendations = cf_recommend(user_id)
else:
    # Treatment: Two-tower model
    recommendations = two_tower_recommend(user_id)
```

Statistical Significance:

```python
import scipy.stats as stats

def check_significance(control_clicks, control_impressions,
                       treatment_clicks, treatment_impressions,
                       alpha=0.05):
    """
    Two-proportion z-test for CTR
    """
    # CTR for each group
    p_control = control_clicks / control_impressions
    p_treatment = treatment_clicks / treatment_impressions

    # Pooled proportion
    p_pooled = (control_clicks + treatment_clicks) / (control_impressions +
    treatment_impressions)

    # Standard error
    se = np.sqrt(p_pooled * (1 - p_pooled) * (1/control_impressions + 1/
    treatment_impressions))

    # Z-score
    z = (p_treatment - p_control) / se

    # P-value (two-tailed)
    p_value = 2 * (1 - stats.norm.cdf(abs(z)))

    # Results
    is_significant = p_value < alpha
    lift = (p_treatment - p_control) / p_control * 100

    print(f"Control CTR: {p_control:.4f}")
    print(f"Treatment CTR: {p_treatment:.4f}")
    print(f"Lift: {lift:+.2f}%")
    print(f"P-value: {p_value:.4f}")
    print(f"Significant: {is_significant}")

    return is_significant, lift, p_value

# Example
```

```
check_significance(
    control_clicks=1000,
    control_impressions=50000,
    treatment_clicks=1100,
    treatment_impressions=50000
)
```

6 Production Architecture

End-to-End System:

```
                 Data Collection
User Events (clicks, views) → Kafka → Feature Store

                          ↓

                Offline Training
Daily Spark Job → Train Models → Model Registry

                          ↓

              Candidate Generation

     CF          Two-Tower       Sequential
  Retrieval      Retrieval      Retrieval

       ↓                ↓                  ↓
            10K Candidates

                          ↓

                   Ranking
DeepFM Model Server (TensorFlow Serving)
→ Top 500 Items

                          ↓

                Re-ranking
Diversity (MMR) + Business Rules
→ Top 50 Items

                          ↓
                  Return to User
```

Latency Budget:

- Candidate generation: 50ms (parallel retrieval from multiple sources)
- Ranking: 80ms (batch inference on GPU)
- Re-ranking: 20ms (CPU-based MMR)
- Total: 150ms (acceptable for most use cases)

7 Interview Preparation

7.1 Sample Questions

1. Design YouTube Video Recommendations

- Clarify: 2B users, 500M videos, homepage recommendations
- Discuss: Two-tower vs matrix factorization
- Address: Cold start for new videos, diversity, watch time optimization
- Scale: ANN index sharding, embedding table size

2. Design TikTok "For You" Feed

- Emphasize: Sequential patterns (session-based)
- Discuss: Real-time vs batch, freshness importance
- Address: Virality detection, creator boost

3. Design Amazon "Customers Who Bought This Also Bought"

- Focus: Item-item CF, co-purchase matrix
- Discuss: Real-time updates, complementary vs similar items

7.2 Key Talking Points

Always mention:

- **Multiple retrieval sources** (not just one algorithm)
- **Trade-offs**: Accuracy vs diversity vs freshness
- **Cold start solution** for new users/items
- **Evaluation**: Both offline (NDCG) and online (CTR, engagement)
- **A/B testing** strategy
- **Scaling**: ANN search, embedding sharding

Go deep on one area (based on interviewer interest):

- Model architecture (two-tower, DeepFM, sequential)
- Feature engineering (user/item/cross features)
- Candidate generation (ANN search, blending)
- Re-ranking (diversity, MMR, DPP)

7.3 Recommended Resources

Papers:

1. **Deep Neural Networks for YouTube Recommendations** (Covington et al., 2016)
2. **Wide & Deep Learning for Recommender Systems** (Cheng et al., 2016)
3. **DeepFM** (Guo et al., 2017)
4. **GRU4Rec** (Hidasi et al., 2016)
5. **Self-Attentive Sequential Recommendation** (Kang & McAuley, 2018)
6. **Deep Learning Recommendation Model (DLRM)** - Facebook (Naumov et al., 2019)

Blog Posts:

- Netflix: Personalized Recommendations
- Spotify: Discover Weekly
- Pinterest: Homefeed Recommendations
- Instagram: Explore Recommendations
- TikTok: For You Algorithm

8 The Future of Recommendations & Strategic Bets

At Principal level, you're not just executing current best practices—you're **guiding the company's long-term technical strategy**. This section covers emerging trends and how to position your organization for the future.

8.1 LLMs in Recommendations: The Paradigm Shift

Large Language Models are fundamentally changing how we think about recommendations. Understanding this evolution is critical for Principal-level interviews in 2024-2025.

The Traditional Paradigm:

- User embeddings + Item embeddings → Dot product → Top-K recommendations
- Models trained on implicit feedback (clicks, watches)
- No natural language understanding

The LLM Paradigm:

- Rich text understanding → Deep semantic representations
- Ability to explain recommendations in natural language
- Zero-shot generalization to new items/categories

Approach 1: LLMs as Feature Extractors (Pragmatic, Happening Now)
Use case: Replace hand-crafted text features with LLM embeddings
Architecture:

```
import torch
from transformers import AutoModel, AutoTokenizer

class LLMFeatureExtractor:
    """Use LLM to create rich item embeddings from text"""
    def __init__(self, model_name='sentence-transformers/all-MiniLM-L6-v2'):
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.model = AutoModel.from_pretrained(model_name)
        self.model.eval()

    def extract_item_embedding(self, title, description, tags):
        """
        Convert unstructured text into dense embedding
        Input: "Product: iPhone 15. Description: Latest Apple..."
        Output: 384-dim embedding
        """
        # Combine all text fields
        text = f"Title: {title}. Description: {description}. Tags: {tags}"

        # Tokenize and encode
        inputs = self.tokenizer(text, return_tensors='pt',
                               truncation=True, max_length=512)

        with torch.no_grad():
            outputs = self.model(**inputs)
            # Use [CLS] token embedding
            embedding = outputs.last_hidden_state[:, 0, :].squeeze()

        return embedding.numpy()

# Integration with existing recommendation system
class HybridRecommender:
    """Combine LLM features with traditional CF features"""
    def __init__(self):
        self.llm_extractor = LLMFeatureExtractor()
```

```
        self.cf_model = TwoTowerModel()  # Traditional collaborative filtering

    def get_item_features(self, item):
        # LLM features from text
        llm_emb = self.llm_extractor.extract_item_embedding(
            item.title, item.description, item.tags
        )

        # Traditional CF features
        cf_emb = self.cf_model.item_tower(item.id)

        # Concatenate or weighted sum
        combined = torch.cat([llm_emb, cf_emb])
        return combined
```

Real-world examples:

- **Netflix**: Uses sentence transformers to embed plot summaries, combine with CF signals
- **Spotify**: LLM embeddings for podcast descriptions, music reviews
- **Pinterest**: CLIP (vision-language model) for image-text understanding

Benefits:

- **Cold start**: New items with good descriptions get meaningful embeddings immediately
- **Semantic understanding**: "wireless earbuds" matches "Bluetooth headphones"
- **Cross-lingual**: Multilingual models work across languages

Costs:

- **Inference cost**: \$0.001-\$0.01 per item (vs \$0.00001 for traditional lookup)
- **Latency**: 50-200ms for LLM forward pass (can batch + cache)
- **Complexity**: Need GPU infrastructure, model updates

Interview talking point: "I'd use a lightweight sentence transformer (384-dim) to embed item descriptions offline, then cache in feature store. This gives us semantic understanding for \$500/month in compute, vs building a custom NLP pipeline."

Approach 2: LLMs as the Recommender (Futuristic, Research-y)

Use case: Frame recommendation as a text generation task

Prompt-based recommendation:

```
Input prompt:
"User has watched:
1. Inception (sci-fi thriller, 2010)
2. Interstellar (sci-fi drama, 2014)
3. The Prestige (mystery thriller, 2006)

Based on this history, recommend 5 movies the user would enjoy.
For each recommendation, explain why."

LLM Output:
1. Shutter Island (2010) - Psychological thriller with complex plot twists,
   similar to The Prestige's mind-bending narrative.
2. Arrival (2016) - Intelligent sci-fi like Interstellar, explores deep
   concepts through human lens.
3. ...
```

Implementation sketch:

```python
from openai import OpenAI

class LLMRecommender:
    def __init__(self):
        self.client = OpenAI()

    def recommend(self, user_history, k=5):
        # Build prompt from user history
        history_text = self._format_history(user_history)

        prompt = f"""User has watched:
{history_text}

Recommend {k} movies the user would enjoy. For each, explain why."""

        response = self.client.chat.completions.create(
            model="gpt-4",
            messages=[{"role": "user", "content": prompt}],
            temperature=0.7
        )

        recommendations = self._parse_response(response.choices[0].message.content)
        return recommendations

    def _format_history(self, items):
        return "\n".join([f"{i+1}. {item.title} ({item.genre}, {item.year})"
                         for i, item in enumerate(items)])
```

Benefits:

- **Natural language explanations**: "Recommended because you liked Inception's plot twists"
- **Zero-shot generalization**: Works on new categories without retraining
- **Contextual reasoning**: Can incorporate time-of-day, mood, recent events
- **Conversational refinement**: User can say "No thrillers, only comedies" → instant adaptation

Costs (Why This Isn't Production-Ready Yet):

- **Extreme inference cost**: \$0.01-\$0.10 per recommendation (100-1000x more expensive)
- **Latency**: 1-5 seconds (vs 10ms for traditional ranker)
- **Lack of control**: Can't enforce business rules, diversity, or fairness constraints easily
- **Hallucinations**: Might recommend non-existent movies
- **No explicit optimization**: Can't directly optimize for click-through rate or watch time

When LLM-as-recommender makes sense:

- **Low-frequency, high-value decisions**: Luxury purchases, major life decisions
- **Explainability critical**: Medical recommendations, financial advice
- **Small catalog**: 1000s of items, not millions
- **Budget allows**: Can afford \$0.10 per recommendation

Interview framework - The Hybrid Future:
"For YouTube-scale recommendations (billions of impressions/day), I'd use:

1. **Retrieval**: Traditional ANN search on collaborative filtering embeddings (¡ 1ms, \$0.00001/request)
2. **Ranking**: Two-tower model enhanced with LLM-generated item embeddings (10ms, \$0.0001/request)

3. **Re-ranking**: For top-10 results, use small fine-tuned LLM to generate explanation snippets (50ms, $0.001/request)

This balances cost ($0.0011 total vs $0.10 for pure LLM) and quality (semantic understanding + personalization)."

The 2025-2026 Bet: Specialized Recommendation LLMs

Predict that we'll see:

- **Small, fast, domain-specific models**: 1B-7B parameters, fine-tuned on rec data
- **Hybrid architectures**: LLM encoder + traditional ranking head
- **Cost reduction**: 10-100x cheaper than GPT-4 via model compression
- **Controllable generation**: RLHF to optimize for business metrics

8.2 Multi-Modal & Cross-Domain Recommendations

The Principal-Level Vision:

Instead of building separate recommenders for videos, articles, products, music—can we build **one unified model**?

Why it matters:

- **Organizational leverage**: One team, one platform, serves all product lines
- **Better representations**: Learn that "users who watch cooking videos buy kitchen gadgets"
- **Cold start solved**: New video can leverage knowledge from 10 years of e-commerce data
- **Personalization across products**: Understand user holistically, not per-app

The Technical Challenge:

Different modalities have different:

- **Features**: Videos (visual, audio, subtitles), Products (images, specs, price), Articles (text, author)
- **Interaction signals**: Click, purchase, watch-time, share
- **Catalogs**: 1M videos vs 100M products vs 10K music artists
- **Objectives**: Maximize watch-time vs revenue vs engagement

Approach: Unified Embedding Space

Architecture:

```
class UnifiedRecommender(nn.Module):
    """Single model for videos, products, articles, music"""
    def __init__(self, user_dim=256, item_dim=256, unified_dim=512):
        super().__init__()

        # User tower (shared across all domains)
        self.user_tower = nn.Sequential(
            nn.Linear(user_dim, 512),
            nn.ReLU(),
            nn.Linear(512, unified_dim)
        )

        # Modality-specific item encoders
        self.video_encoder = VideoEncoder(output_dim=item_dim)
        self.product_encoder = ProductEncoder(output_dim=item_dim)
        self.article_encoder = ArticleEncoder(output_dim=item_dim)

        # Project to unified space
        self.item_projection = nn.Linear(item_dim, unified_dim)

    def forward(self, user_features, item, item_type):
```

```
        # User representation (same for all item types)
        user_emb = self.user_tower(user_features)

        # Item representation (modality-specific encoder)
        if item_type == 'video':
            item_emb = self.video_encoder(item)
        elif item_type == 'product':
            item_emb = self.product_encoder(item)
        elif item_type == 'article':
            item_emb = self.article_encoder(item)

        # Project to unified space
        item_emb = self.item_projection(item_emb)

        # Score in unified space
        score = torch.dot(user_emb, item_emb)
        return score

# Cross-domain training
def cross_domain_loss(model, batch):
    """
    Batch contains multiple item types
    Learn unified user representation that works for all
    """
    user_emb = model.user_tower(batch['user_features'])

    losses = []
    for item_type in ['video', 'product', 'article']:
        if item_type in batch:
            item_emb = model.encode_item(batch[item_type], item_type)
            score = torch.dot(user_emb, item_emb)
            label = batch[f'{item_type}_label']
            loss = nn.BCELoss()(score, label)
            losses.append(loss)

    return sum(losses) / len(losses)  # Average across domains
```

Real-world examples:

- **Google**: One user embedding for Search, YouTube, Play Store, News
- **Amazon**: Unified recommendations for products, Prime Video, Kindle books
- **Meta**: Single user model for Feed, Stories, Reels, Marketplace

Benefits:

- **Transfer learning**: Video-watching behavior informs product recommendations
- **Data efficiency**: 100M video interactions + 1B product interactions = better user understanding
- **Cold start**: New user watches 3 videos → can already recommend products
- **Organizational efficiency**: One platform team, not five domain-specific teams

Challenges:

- **Negative transfer**: Music listening patterns might not correlate with e-commerce
- **Domain imbalance**: If 90% of data is videos, model might underfit products
- **Conflicting objectives**: Watch-time vs purchase revenue—which to optimize?
- **Serving complexity**: Need to query multiple item types simultaneously

Interview Strategy - The Phased Approach:
"To build a unified recommender for Google (Search, YouTube, News, Shopping):
Phase 1 (Year 1): Build separate recommenders, but with **aligned user embeddings**

- Shared user tower, domain-specific item towers
- Prove that unified user representation helps all domains
- Metrics: Watch time (YouTube), CTR (Search), purchases (Shopping)

Phase 2 (Year 2): Add cross-domain signals

- "Users who search for 'cameras' → recommend photography YouTube channels"
- Measure lift: Does cross-domain signal improve recommendations?

Phase 3 (Year 3): Fully unified platform

- Single API: recommend(user, context) → ranked list of videos/products/articles
- Multi-objective optimization: Balance engagement across all surfaces
- Platform team owns user understanding company-wide

Success metrics:

- Time from 'new content type' to 'production recommender': 3 months → 2 weeks
- Organizational leverage: 10 engineers support 100+ product teams
- Business impact: 15% lift in engagement from cross-domain signals

"

Key Interview Talking Points:
When asked about the future of recommendations, demonstrate Principal-level vision:

1. **"LLMs will augment, not replace, traditional rec systems in the next 2-3 years"**
    - Use LLMs as feature extractors (pragmatic, cost-effective)
    - Reserve LLM-as-recommender for high-value, low-frequency use cases
    - Expect specialized small models (1B-7B params) to emerge
2. **"The winning strategy is cross-domain user understanding"**
    - Companies with multiple surfaces (video + e-commerce + social) have an advantage
    - Unified user embeddings enable transfer learning and cold start
    - But start with domain-specific models, evolve to unified platform
3. **"Multi-modal models (CLIP, Flamingo) unlock new recommendation paradigms"**
    - Recommend products based on video content ("buy the jacket from that TikTok")
    - Image-to-product search ("find me this outfit")
    - Text-to-video ("show me tutorials on...") with semantic understanding
4. **"The cost curve matters"**
    - Pure LLM recommendations cost 100-1000x more than traditional
    - But as models compress (distillation, quantization), economics shift
    - By 2026, expect 10x cost reduction → makes LLMs viable for high-QPS

This shows you're not just executing today's playbook—you're **positioning the organization for the next 5 years**.

9 Conclusion

Recommendation systems differ from search in fundamental ways:

- **No explicit query** → Must predict user preferences
- **Collaborative signals** dominate over content
- **Multiple objectives**: Relevance + Diversity + Novelty
- **Sequential patterns** matter (user journey)

Key takeaways for interviews:

1. Master the three-stage funnel (retrieval → ranking → re-ranking)
2. Know modern deep learning architectures (two-tower, DeepFM, sequential models)
3. Understand production concerns (cold start, diversity, scaling)
4. Practice calculating: embedding size, latency budget, A/B test significance

This guide should refresh your recommendations knowledge from your search background and prepare you for Staff/Principal level discussions!