Recommender Systems Deep Dive

Interview Preparation & Research Foundations

Contents

1 Introduction

This document provides a comprehensive overview of recommender systems, covering fundamental algorithms, deep learning approaches, evaluation metrics, and industry best practices. Whether preparing for research positions, ML engineering roles at companies like Netflix/Spotify, or deepening your understanding of personalization systems, this guide synthesizes key concepts from the RecSys community.

1.1 What This Document Covers

- **Classical Methods**: Collaborative filtering (user-based, item-based, matrix factorization)
- **Deep Learning**: Neural collaborative filtering, sequential models, transformers
- **Hybrid Approaches**: Combining content-based and collaborative methods
- **Evaluation**: Offline metrics, A/B testing, beyond-accuracy objectives
- **Production Systems**: Candidate generation, ranking, re-ranking pipelines
- **Research Trends**: Multi-stakeholder optimization, fairness, explainability

1.2 Target Audience

This guide is designed for:

- ML engineers interviewing at Netflix, Spotify, YouTube, Amazon
- Researchers preparing for RecSys conference submissions
- Industry practitioners building production recommendation systems
- Students wanting comprehensive RecSys foundations

2 Recommended Reading List

2.1 Priority 1: Essential Textbooks

1. Recommender Systems Handbook (3rd Edition, 2022)

Authors: Francesco Ricci, Lior Rokach, Bracha Shapira
Publisher: Springer
Pages: 1060
Why Read: The *definitive* reference for recommender systems. Written by top researchers in the field.
Structure (5 parts):

1. **General Techniques**: Collaborative filtering, content-based, knowledge-based, hybrid methods
2. **Advanced Techniques**: Session-based, adversarial ML, group recommendations, cross-domain
3. **Value & Impact**: Business metrics, user experience, long-term value
4. **Human-Computer Interaction**: Explanations, trust, control, transparency
5. **Applications**: E-commerce, music, video, social media, news

Key Topics Covered:

- Neural networks and context-aware methods
- Reciprocal recommender systems (two-way matching)
- Natural language techniques for RecSys
- Explainable AI for recommendations

- Ethical and societal implications

Interview Prep Value:
Comprehensive coverage of everything from basics to cutting-edge research. Essential for senior roles.

Reading Strategy:

- Week 1-2: Part 1 (General Techniques) - Chapters 1-10
- Week 3: Part 2 (Advanced Techniques) - Focus on neural methods
- Week 4: Part 3 (Evaluation) + Part 4 (HCI) - Critical for interviews
- Reference: Part 5 (Applications) - Read chapters relevant to target company

2. Recommender Systems: The Textbook (2016)

Author: Charu C. Aggarwal
Publisher: Springer
Pages: 498
Why Read: Aggarwal is a master educator. Clearest explanations of fundamental algorithms.

Structure (3 categories):

1. **Algorithms & Evaluation**: Collaborative filtering, content-based, knowledge-based, ensemble methods, evaluation
2. **Domain-Specific Applications**: Location-based, social tagging, time series recommendations
3. **Advanced Topics**: Privacy, attack-resistant systems, context-aware recommendations

Strengths:

- Mathematical rigor with intuitive explanations
- Detailed coverage of matrix factorization (SVD, SVD++, NMF)
- Excellent treatment of neighborhood methods
- Strong focus on evaluation methodologies

Interview Prep Value:
Best for understanding *how algorithms work*. Perfect for coding interview preparation.

Reading Strategy:

- Essential: Chapters 2-3 (Collaborative Filtering), Chapter 6 (Evaluation)
- Important: Chapters 4-5 (Content-based, Knowledge-based), Chapter 7 (Ensembles)
- Advanced: Chapters 8-11 (Context-aware, Time-aware, Social)

2.2 Priority 2: Deep Learning & Modern Methods

3. Deep Learning for Recommender Systems (Survey Papers)

Since there's no single "Deep Learning RecSys" textbook yet, focus on these comprehensive surveys:

Zhang et al. (2019): "Deep Learning Based Recommender System: A Survey and New Perspectives"
ArXiv: `https://arxiv.org/abs/1707.07435`

Covers:

- Neural Collaborative Filtering (NCF)
- Autoencoders for collaborative filtering
- CNNs for feature extraction from images/text
- RNNs for sequential recommendations

- Deep reinforcement learning for RecSys
- Adversarial learning and GANs

2024 Update: "In-depth Survey: Deep Learning in Recommender Systems"
Springer Neural Computing and Applications
New Topics:

- Transformers for recommendations (BERT4Rec, SASRec)
- Graph neural networks (LightGCN, PinSage)
- Variational autoencoders (VAE) for collaborative filtering
- Cross-domain transfer learning
- Multi-task learning architectures

Interview Prep Value:
Essential for understanding Netflix/YouTube-style systems. Most interview questions come from these topics.

2.3 Priority 3: Industry Best Practices

4. Tech Company Blog Posts & Papers

Must-Read Industry Papers:

1. **YouTube** (2016): "Deep Neural Networks for YouTube Recommendations" - Two-stage architecture (candidate generation + ranking)
2. **Netflix** (2012): "Netflix Recommender System" - Matrix factorization at scale
3. **Spotify** (2020): "The Rise of the Recommendation Era" - Sequential models and explore-exploit
4. **Amazon** (2003): "Item-to-Item Collaborative Filtering" - Classic item-based CF
5. **Pinterest** (2018): "PinSage: Graph Convolutional Neural Networks for Web-Scale Recommender Systems"
6. **Meta** (2019): "Deep Learning Recommendation Model (DLRM)"

Why Read Industry Papers:

- Understand production constraints (latency, scalability, freshness)
- Learn real-world architectures (two-tower models, multi-stage funnels)
- See how metrics connect to business goals
- Prepare for system design interviews

3 Core Recommender System Concepts

3.1 The Recommendation Problem

Given:

- Set of users $U = \{u_1, u_2, \ldots, u_m\}$
- Set of items $I = \{i_1, i_2, \ldots, i_n\}$
- User-item interaction matrix $R \in \mathbb{R}^{m \times n}$ (typically 99%+ sparse)

Goal: Predict relevance score $\hat{r}_{ui}$ for unobserved user-item pairs and recommend top-k items.

3.2 Types of Feedback

Explicit Feedback:

- Star ratings (1-5 stars)
- Thumbs up/down
- Like/dislike

Implicit Feedback:

- Clicks, views, watches
- Purchase history
- Time spent on page
- Skip behavior (negative signal)

Key Difference: Implicit feedback is abundant but noisy. No explicit negative feedback (absence doesn't mean dislike).

4 Collaborative Filtering

4.1 User-Based Collaborative Filtering

Core Idea: Recommend items liked by similar users.

Algorithm:

1. Compute user similarity: $\text{sim}(u, v) = \cos(\vec{r}_u, \vec{r}_v)$
2. Find k nearest neighbors: $N_k(u) = \{v \mid \text{sim}(u, v) \text{ highest}\}$
3. Predict rating: $\hat{r}_{ui} = \bar{r}_u + \frac{\sum_{v \in N_k(u)} \text{sim}(u,v) \cdot (r_{vi} - \bar{r}_v)}{\sum_{v \in N_k(u)} |\text{sim}(u,v)|}$

Similarity Metrics:

- **Cosine**: $\text{sim}(u, v) = \frac{\vec{r}_u \cdot \vec{r}_v}{\|\vec{r}_u\| \|\vec{r}_v\|}$
- **Pearson Correlation**: Centered cosine (subtracts user mean)
- **Jaccard**: For binary feedback (sets of items)

Python Implementation:

```
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# R: user-item matrix (m x n)
# users in rows, items in columns
user_sim = cosine_similarity(R)

def predict_user_based(user_id, item_id, k=20):
    # Find k nearest neighbors who rated item_id
    neighbors = user_sim[user_id].argsort()[::-1][1:k+1]
    neighbors = [n for n in neighbors if R[n, item_id] > 0]

    if not neighbors:
        return R[user_id].mean() # Fallback to user average

    # Weighted average of neighbor ratings
    sims = user_sim[user_id, neighbors]
    ratings = R[neighbors, item_id]
    return np.dot(sims, ratings) / sims.sum()
```

Pros:

- Simple, interpretable
- No domain knowledge needed
- Works well for niche items

Cons:

- Cold start for new users
- Scalability: $O(m^2)$ similarity computation
- Sparsity: Hard to find similar users

4.2 Item-Based Collaborative Filtering

Core Idea: Recommend items similar to what user already likes.

Algorithm:

1. Compute item similarity: $\text{sim}(i, j)$ based on users who rated both
2. For target item i, find k nearest neighbors: $N_k(i)$
3. Predict: $\hat{r}_{ui} = \frac{\sum_{j \in N_k(i)} \text{sim}(i,j) \cdot r_{uj}}{\sum_{j \in N_k(i)} |\text{sim}(i,j)|}$

Why Item-Based Often Outperforms User-Based:

- Items are more stable than users (user tastes change)
- Fewer items than users in many domains ($n \ll m$)
- Item similarities can be precomputed offline
- Better scalability in production

Amazon's Approach:

```
# Amazon's item-to-item CF (simplified)
def recommend_items(user_id, purchased_items, k=10):
    scores = {}
    for item_i in purchased_items:
        # Get k most similar items to item_i
        similar_items = item_sim[item_i].argsort()[::-1][1:k+1]
        for item_j in similar_items:
            if item_j not in purchased_items:
                scores[item_j] = scores.get(item_j, 0) + item_sim[item_i, item_j]

    # Return top-N items by aggregated similarity
    return sorted(scores.items(), key=lambda x: x[1], reverse=True)[:k]
```

Production Optimizations:

- Precompute top-k similar items for each item
- Store in key-value store (Redis) for fast lookup
- Update similarities incrementally (online learning)

4.3 Matrix Factorization

Core Idea: Decompose sparse user-item matrix into low-rank user and item latent factors.

Model:

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

$$\hat{r}_{ui} = \vec{p}_u^T \vec{q}_i = \sum_{f=1}^{k} p_{uf} \cdot q_{if}$$

Where:

- $P \in \mathbb{R}^{m \times k}$: User latent factors
- $Q \in \mathbb{R}^{n \times k}$: Item latent factors
- k: Number of latent dimensions (typically 50-200)

Optimization (Alternating Least Squares - ALS):

```
import numpy as np

def matrix_factorization_als(R, k=50, lambda_reg=0.1, iterations=20):
    m, n = R.shape
    P = np.random.rand(m, k)
    Q = np.random.rand(n, k)

    # Mask for observed ratings
    mask = (R > 0).astype(float)

    for iteration in range(iterations):
        # Fix Q, solve for P
        for u in range(m):
            Q_u = Q[mask[u] > 0] # Items rated by user u
            r_u = R[u, mask[u] > 0]
            P[u] = np.linalg.solve(Q_u.T @ Q_u + lambda_reg * np.eye(k),
                                   Q_u.T @ r_u)

        # Fix P, solve for Q
        for i in range(n):
            P_i = P[mask[:, i] > 0] # Users who rated item i
            r_i = R[mask[:, i] > 0, i]
            Q[i] = np.linalg.solve(P_i.T @ P_i + lambda_reg * np.eye(k),
                                   P_i.T @ r_i)

    return P, Q
```

Stochastic Gradient Descent (SGD) Alternative:

```
def matrix_factorization_sgd(R, k=50, lr=0.001, lambda_reg=0.1, epochs=100):
    m, n = R.shape
    P = np.random.rand(m, k) * 0.1
    Q = np.random.rand(n, k) * 0.1

    for epoch in range(epochs):
        for u, i in zip(*np.where(R > 0)):
            # Compute error
            error = R[u, i] - P[u].dot(Q[i])

            # Update with gradient descent
            P[u] += lr * (error * Q[i] - lambda_reg * P[u])
            Q[i] += lr * (error * P[u] - lambda_reg * Q[i])

    return P, Q
```

Extensions:

- **SVD++**: Incorporate implicit feedback
- **Biased MF**: Add user/item bias terms: $\hat{r}_{ui} = \mu + b_u + b_i + \vec{p}_u^T \vec{q}_i$
- **Temporal MF**: Time-aware factors
- **NMF (Non-negative MF)**: Constrain $P, Q \geq 0$ for interpretability

Netflix Prize Winning Approach:

- Ensemble of 100+ models
- Blend of matrix factorization variants
- RBMs (Restricted Boltzmann Machines)
- Temporal dynamics modeling
- Final RMSE: 0.8567 (10% improvement over baseline)

5 Content-Based Filtering

Core Idea: Recommend items similar to what user has liked before, based on item features.

5.1 Feature Extraction

Text Features (movies, articles, products):

- TF-IDF vectors from descriptions
- Word embeddings (Word2Vec, GloVe, BERT)
- Topic models (LDA)

Categorical Features:

- Genre (one-hot encoding)
- Director, actors (multi-hot encoding)
- Tags, keywords

Numerical Features:

- Price, popularity, release year
- Average rating

5.2 Building Content-Based Recommender

Algorithm:

1. Build item profile: Feature vector $\vec{f}_i$ for each item
2. Build user profile: Aggregate features of items user liked
3. Compute similarity: sim(user_profile, $\vec{f}_i$)
4. Rank items by similarity

Python Implementation:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

# Example: Movie recommendations based on plot descriptions
movies = ['Movie A', 'Movie B', 'Movie C']
descriptions = [
    'Action thriller with car chases',
    'Romantic comedy set in Paris',
    'Action adventure with explosions'
]

# Extract TF-IDF features
tfidf = TfidfVectorizer(stop_words='english')
item_features = tfidf.fit_transform(descriptions)

# User has watched Movie A and liked it
user_profile = item_features[0]

# Find most similar movies
similarities = cosine_similarity(user_profile, item_features).flatten()
recommendations = sorted(enumerate(similarities), key=lambda x: x[1], reverse=True)

# Top recommendations (excluding Movie A)
for idx, score in recommendations[1:]:
    print(f"{movies[idx]}: {score:.3f}")
```

Pros:

- No cold start for items (can recommend new items immediately)
- No need for user data from other users
- Transparent recommendations (can explain via features)

Cons:

- Limited serendipity (filter bubble)
- Requires rich item metadata
- Cold start for new users
- Over-specialization

6 Neural Collaborative Filtering (NCF)

Motivation: Matrix factorization assumes linear interaction between user and item latent factors. Neural networks can model non-linear interactions.

6.1 NCF Architecture

Paper: He et al. (2017) - "Neural Collaborative Filtering"

Model:

1. **Input Layer**: User ID and Item ID (one-hot encoded)
2. **Embedding Layer**: Map to dense vectors
3. **Neural CF Layers**: MLP to learn interaction function
4. **Output Layer**: Predicted rating or probability

PyTorch Implementation:

```
import torch
import torch.nn as nn

class NCF(nn.Module):
    def __init__(self, num_users, num_items, embedding_dim=64, hidden_layers=[128, 64, 32]):
        super(NCF, self).__init__()

        # Embeddings
        self.user_embedding = nn.Embedding(num_users, embedding_dim)
        self.item_embedding = nn.Embedding(num_items, embedding_dim)

        # MLP layers
        layers = []
        input_dim = embedding_dim * 2
        for hidden_dim in hidden_layers:
            layers.append(nn.Linear(input_dim, hidden_dim))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(0.2))
            input_dim = hidden_dim

        self.mlp = nn.Sequential(*layers)
        self.output = nn.Linear(hidden_layers[-1], 1)

    def forward(self, user_ids, item_ids):
        user_embed = self.user_embedding(user_ids)
        item_embed = self.item_embedding(item_ids)

        # Concatenate embeddings
        x = torch.cat([user_embed, item_embed], dim=1)

        # Pass through MLP
        x = self.mlp(x)
        rating = self.output(x)

        return rating.squeeze()

# Training
model = NCF(num_users=10000, num_items=5000)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.MSELoss()

for epoch in range(epochs):
    for user_batch, item_batch, rating_batch in dataloader:
        optimizer.zero_grad()
        predictions = model(user_batch, item_batch)
        loss = criterion(predictions, rating_batch)
        loss.backward()
        optimizer.step()
```

6.2 Generalized Matrix Factorization (GMF)

NCF generalizes MF by using element-wise product instead of dot product:

$$\begin{aligned} \text{MF}: &\quad \hat{y}_{ui} = \vec{p}_u^T \vec{q}_i \\ \text{GMF}: &\quad \hat{y}_{ui} = \sigma(\vec{h}^T(\vec{p}_u \odot \vec{q}_i)) \end{aligned}$$

6.3 NeuMF: Neural Matrix Factorization

Combine GMF and MLP paths:

```
class NeuMF(nn.Module):
    def __init__(self, num_users, num_items, gmf_dim=64, mlp_dim=64):
        super(NeuMF, self).__init__()

        # GMF path
        self.gmf_user_embed = nn.Embedding(num_users, gmf_dim)
        self.gmf_item_embed = nn.Embedding(num_items, gmf_dim)

        # MLP path
        self.mlp_user_embed = nn.Embedding(num_users, mlp_dim)
        self.mlp_item_embed = nn.Embedding(num_items, mlp_dim)
        self.mlp = nn.Sequential(
            nn.Linear(mlp_dim * 2, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU()
        )

        # Combine GMF and MLP
        self.output = nn.Linear(gmf_dim + 64, 1)

    def forward(self, user_ids, item_ids):
        # GMF path
        gmf_u = self.gmf_user_embed(user_ids)
        gmf_i = self.gmf_item_embed(item_ids)
        gmf_output = gmf_u * gmf_i # Element-wise product

        # MLP path
        mlp_u = self.mlp_user_embed(user_ids)
        mlp_i = self.mlp_item_embed(item_ids)
        mlp_input = torch.cat([mlp_u, mlp_i], dim=1)
        mlp_output = self.mlp(mlp_input)

        # Concatenate and predict
        combined = torch.cat([gmf_output, mlp_output], dim=1)
        rating = self.output(combined)

        return rating.squeeze()
```

7 Sequential Recommendations

Problem: Traditional CF ignores order of user interactions. Sequential models capture temporal dynamics.

7.1 RNN-Based Models

GRU4Rec (Hidasi et al., 2016):

```
import torch.nn as nn

class GRU4Rec(nn.Module):
    def __init__(self, num_items, embedding_dim=100, hidden_dim=100):
        super(GRU4Rec, self).__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)
        embeds = self.item_embedding(item_sequence)
        gru_out, _ = self.gru(embeds)
```

```
        # Use last hidden state
        last_hidden = gru_out[:, -1, :]
        logits = self.fc(last_hidden)

        return logits
```

7.2 Transformer-Based Models

SASRec (Self-Attentive Sequential Recommendation):

```
class SASRec(nn.Module):
    def __init__(self, num_items, max_len=50, embedding_dim=64, num_heads=2, num_layers=2):
        super(SASRec, self).__init__()
        self.item_embedding = nn.Embedding(num_items + 1, embedding_dim, padding_idx=0)
        self.pos_embedding = nn.Embedding(max_len, embedding_dim)

        encoder_layer = nn.TransformerEncoderLayer(
            d_model=embedding_dim,
            nhead=num_heads,
            dim_feedforward=embedding_dim * 4,
            dropout=0.2
        )
        self.transformer = nn.TransformerEncoder(encoder_layer, num_layers=num_layers)
        self.fc = nn.Linear(embedding_dim, num_items)

    def forward(self, item_sequence):
        # item_sequence: (batch, seq_len)
        seq_len = item_sequence.size(1)
        positions = torch.arange(seq_len).unsqueeze(0).to(item_sequence.device)

        # Embeddings
        item_embeds = self.item_embedding(item_sequence)
        pos_embeds = self.pos_embedding(positions)
        embeds = item_embeds + pos_embeds

        # Transformer
        embeds = embeds.transpose(0, 1) # (seq_len, batch, dim)
        mask = self.generate_square_subsequent_mask(seq_len).to(item_sequence.device)
        transformer_out = self.transformer(embeds, mask=mask)
        transformer_out = transformer_out.transpose(0, 1) # (batch, seq_len, dim)

        # Predict next item
        logits = self.fc(transformer_out[:, -1, :])
        return logits

    def generate_square_subsequent_mask(self, sz):
        mask = torch.triu(torch.ones(sz, sz), diagonal=1).bool()
        return mask
```

BERT4Rec (Bidirectional Encoder):

- Masks random items in sequence
- Predicts masked items using bidirectional context
- Pre-training + fine-tuning approach

8 Two-Stage Recommendation Architecture

Problem: Can't score millions of items for every user in real-time.
Solution: Two-stage funnel (Candidate Generation → Ranking)

8.1 Stage 1: Candidate Generation (Retrieval)

Goal: Narrow down millions of items to hundreds of candidates (fast, approximate).
Methods:

1. **Collaborative Filtering**: User's nearest neighbors' items
2. **ANN Search**: Find items with similar embeddings (FAISS, Annoy)
3. **Two-Tower Model**: Encode user and items separately, dot product similarity

Two-Tower Architecture (YouTube):

```
class TwoTowerModel(nn.Module):
    def __init__(self, num_users, num_items, embedding_dim=128):
        super(TwoTowerModel, self).__init__()

        # User tower
        self.user_embedding = nn.Embedding(num_users, embedding_dim)
        self.user_mlp = nn.Sequential(
            nn.Linear(embedding_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 128)
        )

        # Item tower
        self.item_embedding = nn.Embedding(num_items, embedding_dim)
        self.item_mlp = nn.Sequential(
            nn.Linear(embedding_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 128)
        )

    def encode_user(self, user_ids):
        u = self.user_embedding(user_ids)
        return self.user_mlp(u)

    def encode_item(self, item_ids):
        i = self.item_embedding(item_ids)
        return self.item_mlp(i)

    def forward(self, user_ids, item_ids):
        user_embeds = self.encode_user(user_ids)
        item_embeds = self.encode_item(item_ids)

        # Dot product similarity
        scores = (user_embeds * item_embeds).sum(dim=1)
        return scores

# At serving time:
# 1. Encode user once
# 2. Use ANN (FAISS) to find top-k items by dot product
# 3. Pass candidates to ranking model
```

8.2 Stage 2: Ranking

Goal: Precisely rank candidates using rich features (slower, accurate).
Features:

- User features: Demographics, historical behavior
- Item features: Metadata, popularity, freshness

- Context features: Time, device, location
- Cross features: User-item interactions

Deep & Cross Network (DCN):

```
class DCN(nn.Module):
    def __init__(self, input_dim, cross_layers=3, deep_layers=[512, 256, 128]):
        super(DCN, self).__init__()

        # Cross Network
        self.cross_layers = nn.ModuleList([
            nn.Linear(input_dim, input_dim) for _ in range(cross_layers)
        ])

        # Deep Network
        deep = []
        for hidden_dim in deep_layers:
            deep.append(nn.Linear(input_dim, hidden_dim))
            deep.append(nn.ReLU())
            deep.append(nn.Dropout(0.2))
            input_dim = hidden_dim
        self.deep = nn.Sequential(*deep)

        # Combine
        self.output = nn.Linear(input_dim + input_dim, 1)

    def forward(self, x):
        # Cross network
        x_cross = x
        for cross_layer in self.cross_layers:
            x_cross = x * cross_layer(x_cross) + x_cross

        # Deep network
        x_deep = self.deep(x)

        # Combine and predict
        combined = torch.cat([x_cross, x_deep], dim=1)
        return self.output(combined).squeeze()
```

9 Evaluation Metrics

9.1 Rating Prediction Metrics

For explicit feedback (1-5 star ratings):

RMSE (Root Mean Squared Error):

$$\text{RMSE} = \sqrt{\frac{1}{|T|} \sum_{(u,i) \in T} (r_{ui} - \hat{r}_{ui})^2}$$

MAE (Mean Absolute Error):

$$\text{MAE} = \frac{1}{|T|} \sum_{(u,i) \in T} |r_{ui} - \hat{r}_{ui}|$$

9.2 Ranking Metrics

For top-N recommendations:

Precision@K:

$$\text{Precision@K} = \frac{\#\text{ relevant items in top-K}}{K}$$

Recall@K:

$$\text{Recall@K} = \frac{\text{\# relevant items in top-K}}{\text{Total \# relevant items}}$$

NDCG@K (Normalized Discounted Cumulative Gain):

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

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

MRR (Mean Reciprocal Rank):

$$\text{MRR} = \frac{1}{|U|} \sum_{u=1}^{|U|} \frac{1}{\text{rank of first relevant item}}$$

MAP (Mean Average Precision):

$$\text{MAP} = \frac{1}{|U|} \sum_{u=1}^{|U|} \frac{1}{|R_u|} \sum_{k=1}^{K} \text{Precision@k} \cdot \text{rel}_k$$

9.3 Beyond-Accuracy Metrics

Coverage:

$$\text{Coverage} = \frac{|\{i \mid i \text{ recommended to at least 1 user}\}|}{|I|}$$

Diversity (Intra-List Diversity):

$$\text{ILD} = \frac{2}{|L|(|L|-1)} \sum_{i \in L} \sum_{j \in L, j > i} (1 - \text{sim}(i, j))$$

Novelty:

$$\text{Novelty}(i) = -\log_2 \frac{\text{popularity}(i)}{|U|}$$

Serendipity: Unexpected but relevant recommendations.

9.4 Python Evaluation Example

```
import numpy as np

def ndcg_at_k(relevance_scores, k):
    """
␣␣␣␣relevance_scores:␣list␣of␣relevance␣(1␣for␣relevant,␣0␣for␣not)
␣␣␣␣k:␣cutoff
␣␣␣␣"""
    relevance_scores = np.array(relevance_scores)[:k]

    # DCG
    dcg = np.sum((2**relevance_scores - 1) / np.log2(np.arange(2, len(relevance_scores) + 2)))

    # IDCG (ideal DCG - all relevant items first)
    ideal_relevance = sorted(relevance_scores, reverse=True)
    idcg = np.sum((2**ideal_relevance - 1) / np.log2(np.arange(2, len(ideal_relevance) + 2)))

    return dcg / idcg if idcg > 0 else 0.0

def precision_at_k(recommended, relevant, k):
    recommended_k = set(recommended[:k])
```

```
    return len(recommended_k & relevant) / k

def recall_at_k(recommended, relevant, k):
    recommended_k = set(recommended[:k])
    return len(recommended_k & relevant) / len(relevant) if relevant else 0.0

# Example usage
recommended_items = [101, 205, 304, 112, 501] # Top-5 recommendations
relevant_items = {101, 304, 501, 999} # Ground truth relevant

print(f"Precision@5: {precision_at_k(recommended_items, relevant_items, 5):.3f}")
print(f"Recall@5: {recall_at_k(recommended_items, relevant_items, 5):.3f}")

# NDCG example
relevance = [1, 0, 1, 0, 1] # 1 if relevant, 0 if not (for top-5)
print(f"NDCG@5: {ndcg_at_k(relevance, 5):.3f}")
```

10 Production Challenges & Solutions

10.1 Cold Start Problem

New User Cold Start:

- **Popular items**: Show trending/top-rated items
- **Onboarding**: Ask user to rate a few items or select preferences
- **Content-based**: Use demographics, device, location
- **Meta-learning**: Learn from other users with similar signup behavior

New Item Cold Start:

- **Content-based**: Use item metadata (genre, tags, price)
- **Explore-exploit**: Show new items to some users (Thompson Sampling, UCB)
- **Transfer learning**: Use embeddings from similar items

10.2 Scalability

Challenges:

- 100M+ users, 10M+ items
- Real-time inference (¡ 100ms)
- Model retraining daily/hourly

Solutions:

1. **Approximate Nearest Neighbors (ANN)**: FAISS, Annoy, ScaNN
2. **Distributed Training**: Spark MLlib, Horovod, Ray
3. **Model Serving**: TensorFlow Serving, TorchServe, Triton
4. **Caching**: Redis for precomputed recommendations
5. **Sharding**: Partition users/items across servers

FAISS Example:

```
import faiss
import numpy as np

# Item embeddings (10M items, 128-dim)
item_embeddings = np.random.randn(10_000_000, 128).astype('float32')

# Build index
index = faiss.IndexFlatIP(128) # Inner product (for dot product similarity)
index.add(item_embeddings)

# Query: Find top-100 items for user
user_embedding = np.random.randn(1, 128).astype('float32')
k = 100
scores, item_ids = index.search(user_embedding, k)

print(f"Top-{k} items: {item_ids[0]}")
print(f"Scores: {scores[0]}")
```

10.3 Freshness vs. Accuracy Trade-off

Problem: User preferences change, items come and go. Stale recommendations hurt engagement.
Solutions:

- **Online learning**: Update models with streaming data (online SGD)
- **Recency weighting**: Exponential decay of old interactions
- **Time-aware features**: Hour of day, day of week, seasonality
- **A/B testing**: Continuously validate model performance

10.4 Diversity & Filter Bubble

Problem: Pure accuracy optimization leads to echo chambers.
Solutions:

- **MMR (Maximal Marginal Relevance)**: Balance relevance and diversity
- **DPP (Determinantal Point Process)**: Probabilistic diverse subset selection
- **Multi-objective optimization**: Accuracy + diversity + novelty
- **Exploration**: ϵ-greedy, Thompson Sampling

MMR Algorithm:

```
def maximal_marginal_relevance(candidates, user_profile, lambda_param=0.5, k=10):
    """
    candidates: list of item embeddings
    user_profile: user embedding
    lambda_param: trade-off between relevance and diversity (0-1)
    """
    selected = []
    remaining = list(range(len(candidates)))

    for _ in range(k):
        scores = []
        for idx in remaining:
            # Relevance to user
            relevance = cosine_similarity(candidates[idx], user_profile)

            # Max similarity to already selected items
```

```
            if selected:
                diversity = max(cosine_similarity(candidates[idx], candidates[s])
                                for s in selected)
            else:
                diversity = 0

            # MMR score
            mmr = lambda_param * relevance - (1 - lambda_param) * diversity
            scores.append((idx, mmr))

        # Select item with max MMR
        best_idx, _ = max(scores, key=lambda x: x[1])
        selected.append(best_idx)
        remaining.remove(best_idx)

    return selected
```

11 Advanced Topics

11.1 Multi-Task Learning

Goal: Jointly optimize for multiple objectives (clicks, watch time, likes, shares).

YouTube Ranking Model:

```
class MultiTaskModel(nn.Module):
    def __init__(self, input_dim, shared_layers=[256, 128], task_layers=[64, 32]):
        super(MultiTaskModel, self).__init__()

        # Shared bottom layers
        shared = []
        for hidden_dim in shared_layers:
            shared.append(nn.Linear(input_dim, hidden_dim))
            shared.append(nn.ReLU())
            input_dim = hidden_dim
        self.shared = nn.Sequential(*shared)

        # Task-specific towers
        self.click_tower = self.build_tower(input_dim, task_layers)
        self.watch_time_tower = self.build_tower(input_dim, task_layers)
        self.like_tower = self.build_tower(input_dim, task_layers)

        # Output heads
        self.click_head = nn.Linear(task_layers[-1], 1) # Binary classification
        self.watch_time_head = nn.Linear(task_layers[-1], 1) # Regression
        self.like_head = nn.Linear(task_layers[-1], 1) # Binary classification

    def build_tower(self, input_dim, layers):
        tower = []
        for hidden_dim in layers:
            tower.append(nn.Linear(input_dim, hidden_dim))
            tower.append(nn.ReLU())
            input_dim = hidden_dim
        return nn.Sequential(*tower)

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

        # Task-specific predictions
        click_repr = self.click_tower(shared_repr)
        watch_time_repr = self.watch_time_tower(shared_repr)
        like_repr = self.like_tower(shared_repr)
```

```
        click_prob = torch.sigmoid(self.click_head(click_repr))
        watch_time = self.watch_time_head(watch_time_repr)
        like_prob = torch.sigmoid(self.like_head(like_repr))

        return click_prob, watch_time, like_prob

# Training with multi-task loss
def multi_task_loss(predictions, targets, weights=[1.0, 0.5, 0.3]):
    click_pred, watch_time_pred, like_pred = predictions
    click_target, watch_time_target, like_target = targets

    click_loss = F.binary_cross_entropy(click_pred, click_target)
    watch_time_loss = F.mse_loss(watch_time_pred, watch_time_target)
    like_loss = F.binary_cross_entropy(like_pred, like_target)

    return (weights[0] * click_loss +
            weights[1] * watch_time_loss +
            weights[2] * like_loss)
```

11.2 Graph Neural Networks for RecSys

Motivation: User-item interactions form a bipartite graph. GNNs can propagate information through the graph.

LightGCN (He et al., 2020):

```
import torch
import torch.nn as nn
import torch_geometric
from torch_geometric.nn import MessagePassing

class LightGCN(nn.Module):
    def __init__(self, num_users, num_items, embedding_dim=64, num_layers=3):
        super(LightGCN, self).__init__()
        self.num_users = num_users
        self.num_items = num_items
        self.num_layers = num_layers

        # Embeddings (users and items share embedding space)
        self.embedding = nn.Embedding(num_users + num_items, embedding_dim)
        nn.init.normal_(self.embedding.weight, std=0.1)

    def forward(self, edge_index):
        # edge_index: user-item edges
        embeddings = self.embedding.weight
        all_embeddings = [embeddings]

        # Graph convolution layers
        for layer in range(self.num_layers):
            embeddings = self.propagate(edge_index, embeddings)
            all_embeddings.append(embeddings)

        # Average embeddings across layers
        final_embeddings = torch.stack(all_embeddings, dim=1).mean(dim=1)

        user_embeddings = final_embeddings[:self.num_users]
        item_embeddings = final_embeddings[self.num_users:]

        return user_embeddings, item_embeddings

    def propagate(self, edge_index, embeddings):
        # Normalize by degree (symmetric normalization)
```

```
        row, col = edge_index
        deg = torch_geometric.utils.degree(row, embeddings.size(0))
        deg_inv_sqrt = deg.pow(-0.5)
        deg_inv_sqrt[deg_inv_sqrt == float('inf')] = 0
        norm = deg_inv_sqrt[row] * deg_inv_sqrt[col]

        # Message passing
        return torch_geometric.utils.scatter(embeddings[col] * norm.view(-1, 1), row,
                                             dim=0, dim_size=embeddings.size(0), reduce='sum')

    def predict(self, user_ids, item_ids, user_embeddings, item_embeddings):
        user_embed = user_embeddings[user_ids]
        item_embed = item_embeddings[item_ids]
        return (user_embed * item_embed).sum(dim=1)
```

11.3 Conversational Recommenders

Goal: Interactive recommendations through dialogue.

Approaches:

- **Critiquing**: "Show me similar but cheaper options"
- **Preference elicitation**: Ask questions to narrow down options
- **Explanations**: "Why did you recommend this?"
- **Feedback loops**: Incorporate thumbs up/down in real-time

11.4 Fairness in Recommendations

Issues:

- **Popularity bias**: Rich get richer (Matthew effect)
- **Filter bubbles**: Users see narrow content
- **Demographic bias**: Gender, race, age discrimination
- **Provider fairness**: Small creators don't get exposure

Solutions:

- **Calibration**: Match recommendation distribution to user's historical distribution
- **Re-ranking**: Post-process to ensure diversity/fairness
- **Exposure control**: Guarantee minimum exposure for all items
- **Adversarial debiasing**: Remove sensitive attributes from embeddings

12 Interview Preparation Strategy

12.1 Week 1: Foundations (Ricci Handbook Part 1)

Topics:

- Collaborative filtering (user-based, item-based)
- Matrix factorization (SVD, ALS)
- Content-based filtering
- Hybrid methods

Coding Practice:

- Implement user-based CF from scratch (NumPy)
- Implement matrix factorization with SGD
- LeetCode: "Design a Recommender System" (Medium)

12.2 Week 2: Deep Learning (Aggarwal + 2024 Surveys)

Topics:

- Neural Collaborative Filtering (NCF)
- Two-tower models
- Sequential models (RNN, Transformer)
- Graph neural networks (LightGCN)

Coding Practice:

- Build NCF in PyTorch
- Implement two-tower model with FAISS retrieval
- Train GRU4Rec on MovieLens dataset

12.3 Week 3: Production Systems (Industry Papers)

Topics:

- YouTube's two-stage architecture
- Netflix's matrix factorization at scale
- Spotify's sequential models
- Pinterest's PinSage (GNN at scale)

System Design Practice:

- Design Netflix recommendation system
- Design Spotify's "Discover Weekly"
- Design Amazon's "Customers who bought this also bought..."

12.4 Week 4: Evaluation & Advanced Topics

Topics:

- Offline metrics (NDCG, MAP, Recall@K)
- A/B testing for recommendations
- Beyond-accuracy objectives (diversity, novelty, fairness)
- Multi-task learning
- Conversational recommenders

Practice:

- Implement NDCG, MAP, diversity metrics
- Design A/B test for new ranking model
- Discuss trade-offs: accuracy vs. diversity vs. serendipity

13 Common Interview Questions

13.1 Algorithmic Questions

Q1: Explain the difference between user-based and item-based collaborative filtering. When would you use each?
Answer:

- **User-based**: Find similar users, recommend what they liked. Better for users with stable preferences.
- **Item-based**: Find similar items, recommend similar to what user liked. Better when items are stable (movies vs. fashion).
- **Scalability**: Item-based often better because # items ¡ # users, item similarities can be precomputed.
- **Example**: Amazon uses item-based ("Customers who bought this also bought...").

Q2: How does matrix factorization work? What are its advantages over neighborhood methods?
Answer:

- **MF**: Decompose sparse R into $P \times Q^T$ where P is user factors, Q is item factors.
- **Advantages**: Handles sparsity better, captures latent patterns, scalable with ALS/SGD.
- **Example**: Netflix Prize winner used ensemble of MF models.

Q3: How would you handle cold start for new users and new items?
Answer:

- **New user**: Popular items, onboarding flow, content-based on demographics, meta-learning.
- **New item**: Content-based on metadata, explore-exploit strategies, transfer learning from similar items.
- **Hybrid**: Combine CF and content-based, gradually shift from content to CF as data accumulates.

13.2 System Design Questions

Q4: Design a recommendation system for YouTube videos.
Answer:

1. **Requirements**: 2B users, 800M videos, ¡ 100ms latency, personalized homepage, watch next.
2. **Architecture**:
   - **Candidate generation**: Two-tower model, retrieve top-1000 from billions using ANN (FAISS)
   - **Ranking**: Multi-task DNN predicting click, watch time, likes
   - **Re-ranking**: Diversity, freshness, fairness
3. **Features**: User watch history, video metadata, context (time, device), CTR, watch time
4. **Offline**: Batch training on TPUs, daily model updates
5. **Online**: TensorFlow Serving, Redis caching, A/B testing
6. **Metrics**: Click-through rate, watch time, engagement (likes, shares), return rate

Q5: How would you evaluate a new recommendation algorithm?
Answer:

1. **Offline evaluation**: Historical data, NDCG@K, Recall@K, coverage, diversity
2. **A/B testing**: 50/50 split, measure CTR, watch time, engagement, revenue
3. **Beyond-accuracy**: User satisfaction surveys, diversity, novelty, serendipity
4. **Long-term effects**: User retention, repeat visits, content discovery
5. **Iterate**: Start with offline → online A/B → gradual rollout → monitor long-term

13.3 Deep Learning Questions

Q6: Explain Neural Collaborative Filtering (NCF). How does it differ from matrix factorization?
Answer:

- **MF**: Linear interaction (dot product): $\hat{y}_{ui} = \vec{p}_u^T \vec{q}_i$
- **NCF**: Non-linear interaction via MLP: $\hat{y}_{ui} = \text{MLP}([\vec{p}_u; \vec{q}_i])$
- **Advantage**: Can capture complex non-linear patterns
- **NeuMF**: Combines GMF (generalized MF) and MLP paths

Q7: How would you model sequential behavior in recommendations?
Answer:

- **RNN/GRU**: GRU4Rec encodes session sequence
- **Transformer**: SASRec with self-attention captures long-range dependencies
- **BERT4Rec**: Bidirectional encoding with masked item prediction
- **Production**: Two-stage (sequence model for candidates $\rightarrow$ ranking model for final order)
- **Example**: Spotify's "Discover Weekly" uses sequential models on listening history

13.4 ML Engineering Questions

Q8: How would you scale recommendations to 100M users and 10M items?
Answer:

- **Candidate generation**: ANN with FAISS (100ms for top-1000 from 10M items)
- **Model serving**: TensorFlow Serving on GPU, batch inference
- **Caching**: Precompute recommendations for active users (Redis)
- **Sharding**: Partition users across servers
- **Training**: Distributed training on Spark, daily model updates
- **Monitoring**: Track latency, throughput, model drift

Q9: How do you handle the accuracy vs. diversity trade-off?
Answer:

- **MMR (Maximal Marginal Relevance)**: Balance relevance and diversity
- **DPP (Determinantal Point Process)**: Probabilistic diverse subset
- **Multi-objective**: Optimize weighted sum of accuracy + diversity + novelty
- **Re-ranking**: Post-process to ensure categories/genres are covered
- **Exploration**: ϵ-greedy, Thompson Sampling for new items
- **Example**: Netflix shows mix of popular + niche, recent + catalog

14 RecSys Conference & Community

14.1 ACM RecSys Conference

2024: Bari, Italy (October 14-18)
2025: Prague, Czech Republic (September 22-26)

Tracks:

- Main research track (long/short papers)
- Industry track (production systems)
- Doctoral symposium
- Workshops (fairness, explainability, session-based, etc.)
- RecSys Challenge (annual competition by companies like Spotify, Twitter)

Key Topics (2024-2025):

- Large language models for recommendations
- Multi-stakeholder optimization
- Conversational recommenders
- Fairness and bias mitigation
- Graph neural networks
- Sequential and session-based recommendations

14.2 Related Conferences

WSDM (Web Search and Data Mining): Feb/March
WWW (The Web Conference): April/May
KDD (Knowledge Discovery and Data Mining): August
CIKM (Conference on Information and Knowledge Management): October
SIGIR (Information Retrieval): July - some RecSys overlap

15 Conclusion

Recommender systems are at the intersection of machine learning, information retrieval, and user experience. Whether you're preparing for interviews at Netflix, building a production RecSys, or conducting research for RecSys 2025, this guide provides a comprehensive foundation.

Key Takeaways:

1. **Start with fundamentals**: Collaborative filtering, matrix factorization
2. **Master deep learning**: NCF, two-tower models, sequential models
3. **Study production systems**: YouTube, Netflix, Spotify architectures
4. **Practice coding**: Implement algorithms from scratch, use real datasets
5. **Think beyond accuracy**: Diversity, fairness, explainability matter
6. **Stay current**: Read RecSys papers, follow industry blogs

Final Advice: Recommender systems is a rapidly evolving field. The best way to learn is to build. Use MovieLens or Amazon Reviews datasets, implement various algorithms, compare results, and iterate. Good luck!