Chapter 23: Recommendation Systems

Introduction: Why Recommendations Matter

Recommendations drive engagement and revenue.

```
Netflix: 80% of watched content from recommendations
Amazon: 35% of revenue from recommendations
YouTube: 70% of watch time from recommendations

Without Recommendations:
User sees random content → Low engagement → Leaves

With Recommendations:
User sees personalized content → High engagement → Stays
```

---

1. Collaborative Filtering

User-Based Collaborative Filtering

"Users who are similar to you liked..."

```
Concept: Find similar users, recommend what they liked

User Matrix:
        Movie A  Movie B  Movie C  Movie D
User 1:    5        3        ?        4
User 2:    4        ?        2        5
User 3:    ?        5        3        4
User 4:    5        4        3        ?

Goal: Predict User 1's rating for Movie C

Step 1: Find similar users
User 1 rated: A=5, B=3, D=4
User 2 rated: A=4, D=5 (similar to User 1!)
User 3 rated: B=5, C=3, D=4
User 4 rated: A=5, B=4, C=3

Calculate similarity (cosine similarity):
Sim(User1, User2) = 0.85 (very similar!)
Sim(User1, User3) = 0.45
Sim(User1, User4) = 0.92 (most similar!)

Step 2: Weighted average of similar users' ratings
User 2 rated Movie C: 2 (weight: 0.85)
User 4 rated Movie C: 3 (weight: 0.92)

Predicted rating = (2 × 0.85 + 3 × 0.92) / (0.85 + 0.92)
                 = (1.7 + 2.76) / 1.77
                 = 2.52

Recommend Movie C with predicted rating: 2.5
```

Implementation:

```python
```

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

class UserBasedCF:
    def __init__(self):
        self.ratings = {}  # user_id -> {item_id: rating}
        self.user_similarity = {}

    def add_rating(self, user_id, item_id, rating):
        if user_id not in self.ratings:
            self.ratings[user_id] = {}

        self.ratings[user_id][item_id] = rating

    def calculate_similarity(self, user1_id, user2_id):
        """Calculate cosine similarity between two users"""
        user1_ratings = self.ratings.get(user1_id, {})
        user2_ratings = self.ratings.get(user2_id, {})

        # Find common items
        common_items = set(user1_ratings.keys()) & set(user2_ratings.keys())

        if len(common_items) == 0:
            return 0

        # Create vectors
        vector1 = [user1_ratings[item] for item in common_items]
        vector2 = [user2_ratings[item] for item in common_items]

        # Cosine similarity
        similarity = cosine_similarity([vector1], [vector2])[0][0]

        return similarity

    def find_similar_users(self, user_id, n=10):
        """Find n most similar users"""
        similarities = []

        for other_user_id in self.ratings:
            if other_user_id != user_id:
                sim = self.calculate_similarity(user_id, other_user_id)
                if sim > 0:
                    similarities.append((other_user_id, sim))

        # Sort by similarity
        similarities.sort(key=lambda x: x[1], reverse=True)
```

```
        return similarities[:n]

    def recommend(self, user_id, n=10):
        """Recommend n items for user"""
        user_ratings = self.ratings.get(user_id, {})

        # Find similar users
        similar_users = self.find_similar_users(user_id, 20)

        # Aggregate recommendations
        predictions = {}

        for similar_user_id, similarity in similar_users:
            similar_user_ratings = self.ratings[similar_user_id]

            # Items this user hasn't rated yet
            for item_id, rating in similar_user_ratings.items():
                if item_id not in user_ratings:
                    if item_id not in predictions:
                        predictions[item_id] = {
                            'total_weight': 0,
                            'weighted_sum': 0
                        }

                    predictions[item_id]['weighted_sum'] += rating * similarity
                    predictions[item_id]['total_weight'] += similarity

        # Calculate predicted ratings
        recommendations = []
        for item_id, data in predictions.items():
            predicted_rating = data['weighted_sum'] / data['total_weight']
            recommendations.append((item_id, predicted_rating))

        # Sort by predicted rating
        recommendations.sort(key=lambda x: x[1], reverse=True)

        return recommendations[:n]

# Usage
cf = UserBasedCF()

# Add ratings
cf.add_rating('user1', 'movie_a', 5)
cf.add_rating('user1', 'movie_b', 3)
cf.add_rating('user1', 'movie_d', 4)
```

```python
cf.add_rating('user2', 'movie_a', 4)
cf.add_rating('user2', 'movie_c', 2)
cf.add_rating('user2', 'movie_d', 5)

cf.add_rating('user3', 'movie_b', 5)
cf.add_rating('user3', 'movie_c', 3)
cf.add_rating('user3', 'movie_d', 4)

cf.add_rating('user4', 'movie_a', 5)
cf.add_rating('user4', 'movie_b', 4)
cf.add_rating('user4', 'movie_c', 3)

# Get recommendations for user1
recommendations = cf.recommend('user1', n=5)

print("Recommendations for user1:")
for item_id, predicted_rating in recommendations:
    print(f"  {item_id}: {predicted_rating:.2f}")

# Output:
# Recommendations for user1:
#   movie_c: 2.52
```

Item-Based Collaborative Filtering

"Users who liked this also liked..."

```
Concept: Find similar items, recommend similar items to what user liked

Item Similarity Matrix:
          Movie A  Movie B  Movie C  Movie D
Movie A:    1.0      0.3      0.1      0.8
Movie B:    0.3      1.0      0.7      0.2
Movie C:    0.1      0.7      1.0      0.3
Movie D:    0.8      0.2      0.3      1.0

User 1 liked Movie A (rating: 5)
Most similar to Movie A: Movie D (similarity: 0.8)
→ Recommend Movie D

User 1 liked Movie B (rating: 3)
Most similar to Movie B: Movie C (similarity: 0.7)
→ Recommend Movie C

Advantages over User-Based:
✓ Items change less than user preferences
✓ Can precompute item similarities
✓ Faster at query time
✓ Better for large user bases
```

Implementation:

```python
```

```python
class ItemBasedCF:
    def __init__(self):
        self.ratings = {}  # user_id -> {item_id: rating}
        self.item_similarity = {}  # Precomputed

    def add_rating(self, user_id, item_id, rating):
        if user_id not in self.ratings:
            self.ratings[user_id] = {}
        self.ratings[user_id][item_id] = rating

    def build_item_similarity_matrix(self):
        """Precompute item-to-item similarities"""
        # Get all items
        all_items = set()
        for user_ratings in self.ratings.values():
            all_items.update(user_ratings.keys())

        all_items = list(all_items)

        # Build item vectors (users who rated each item)
        item_vectors = {}

        for item in all_items:
            vector = []
            for user_id in self.ratings:
                rating = self.ratings[user_id].get(item, 0)
                vector.append(rating)
            item_vectors[item] = vector

        # Calculate similarities
        for i, item1 in enumerate(all_items):
            self.item_similarity[item1] = {}

            for j, item2 in enumerate(all_items):
                if item1 == item2:
                    self.item_similarity[item1][item2] = 1.0
                else:
                    sim = cosine_similarity(
                        [item_vectors[item1]],
                        [item_vectors[item2]]
                    )[0][0]

                    self.item_similarity[item1][item2] = sim

        print(f"Built similarity matrix for {len(all_items)} items")
```

```python
    def recommend(self, user_id, n=10):
        """Recommend items based on item similarity"""
        user_ratings = self.ratings.get(user_id, {})

        if not user_ratings:
            return []

        predictions = {}

        # For each item user hasn't rated
        all_items = set()
        for ratings in self.ratings.values():
            all_items.update(ratings.keys())

        unrated_items = all_items - set(user_ratings.keys())

        for candidate_item in unrated_items:
            weighted_sum = 0
            similarity_sum = 0

            # Look at items user has rated
            for rated_item, rating in user_ratings.items():
                if rated_item in self.item_similarity and \
                   candidate_item in self.item_similarity[rated_item]:

                    similarity = self.item_similarity[rated_item][candidate_item]

                    if similarity > 0:
                        weighted_sum += similarity * rating
                        similarity_sum += similarity

            if similarity_sum > 0:
                predicted_rating = weighted_sum / similarity_sum
                predictions[candidate_item] = predicted_rating

        # Sort by predicted rating
        recommendations = sorted(
            predictions.items(),
            key=lambda x: x[1],
            reverse=True
        )

        return recommendations[:n]

# Usage
cf = ItemBasedCF()
```

```python
# Add ratings
users_ratings = {
    'user1': {'movie_a': 5, 'movie_b': 3, 'movie_d': 4},
    'user2': {'movie_a': 4, 'movie_c': 2, 'movie_d': 5},
    'user3': {'movie_b': 5, 'movie_c': 3, 'movie_d': 4},
    'user4': {'movie_a': 5, 'movie_b': 4, 'movie_c': 3}
}

for user_id, ratings in users_ratings.items():
    for item_id, rating in ratings.items():
        cf.add_rating(user_id, item_id, rating)

# Build similarity matrix (precompute)
cf.build_item_similarity_matrix()

# Get recommendations
recommendations = cf.recommend('user1', n=5)

print("Recommendations for user1 (Item-Based):")
for item_id, predicted_rating in recommendations:
    print(f"  {item_id}: {predicted_rating:.2f}")
```

---

2. Content-Based Filtering

"Recommend items similar to what you liked."

```
Concept: Analyze item features, recommend similar items

Movie Features:
Movie A: [Action, Sci-Fi, 2020s, High-Budget]
Movie B: [Drama, Romance, 2010s, Medium-Budget]
Movie C: [Action, Adventure, 2020s, High-Budget]

User 1 liked Movie A
Most similar: Movie C (both Action, 2020s, High-Budget)
→ Recommend Movie C

Feature Vectors:
Movie A: [1, 0, 1, 0, 1, 0, 1]
    (Action, Drama, Sci-Fi, Romance, 2020s, 2010s, High-Budget)

Movie C: [1, 0, 0, 1, 1, 0, 1]
    (Action, Drama, Sci-Fi, Adventure, 2020s, 2010s, High-Budget)

Cosine Similarity: 0.75 (similar!)
```

Implementation:

```python
```

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

class ContentBasedRecommender:
    def __init__(self):
        self.items = {}  # item_id -> features
        self.vectorizer = TfidfVectorizer()
        self.item_vectors = None
        self.item_ids = []

    def add_item(self, item_id, features):
        """
        Add item with features
        features: text description of item
        """
        self.items[item_id] = features

    def build_model(self):
        """Build TF-IDF vectors for all items"""
        self.item_ids = list(self.items.keys())
        item_features = [self.items[item_id] for item_id in self.item_ids]

        # Create TF-IDF matrix
        self.item_vectors = self.vectorizer.fit_transform(item_features)

        print(f"Built model for {len(self.item_ids)} items")

    def find_similar_items(self, item_id, n=10):
        """Find items similar to given item"""
        if item_id not in self.item_ids:
            return []

        item_idx = self.item_ids.index(item_id)
        item_vector = self.item_vectors[item_idx]

        # Calculate similarity to all other items
        similarities = cosine_similarity(item_vector, self.item_vectors)[0]

        # Get top N (excluding the item itself)
        similar_indices = similarities.argsort()[::-1][1:n+1]

        results = [
            (self.item_ids[idx], similarities[idx])
            for idx in similar_indices
        ]
```

```python
        return results

    def recommend_for_user(self, user_liked_items, n=10):
        """Recommend based on user's liked items"""
        if not self.item_vectors:
            self.build_model()

        # Get indices of liked items
        liked_indices = [
            self.item_ids.index(item_id)
            for item_id in user_liked_items
            if item_id in self.item_ids
        ]

        if not liked_indices:
            return []

        # Average vector of liked items (user profile)
        user_profile = self.item_vectors[liked_indices].mean(axis=0)

        # Find items similar to user profile
        similarities = cosine_similarity(user_profile, self.item_vectors)[0]

        # Remove already liked items
        for idx in liked_indices:
            similarities[idx] = -1

        # Get top N
        top_indices = similarities.argsort()[::-1][:n]

        results = [
            (self.item_ids[idx], similarities[idx])
            for idx in top_indices
            if similarities[idx] > 0
        ]

        return results

# Usage
recommender = ContentBasedRecommender()

# Add movies with features
recommender.add_item('movie1', 'action sci-fi space adventure thrilling')
recommender.add_item('movie2', 'drama romance emotional heartwarming')
recommender.add_item('movie3', 'action adventure superhero exciting')
recommender.add_item('movie4', 'comedy funny lighthearted entertaining')
recommender.add_item('movie5', 'action sci-fi futuristic technology')
```

```
# Build model
recommender.build_model()

# Find similar to movie1
similar = recommender.find_similar_items('movie1', n=3)
print("Similar to movie1:")
for item_id, score in similar:
    print(f"  {item_id}: {score:.2f}")

# Output:
#   movie5: 0.78 (both action sci-fi)
#   movie3: 0.65 (both action)
#   movie2: 0.12 (not similar)

# Recommend for user who liked movie1 and movie3
recommendations = recommender.recommend_for_user(['movie1', 'movie3'], n=3)
print("\nRecommendations:")
for item_id, score in recommendations:
    print(f"  {item_id}: {score:.2f}")

# Output:
#   movie5: 0.85 (similar to liked movies)
#   movie4: 0.25
```

---

3. Hybrid Approach (Netflix, Amazon)

Combine multiple techniques.

```
Hybrid Recommendation:

┌─────────────────────────────────────────┐
│ 1. Collaborative Filtering              │
│    "Users like you watched..."          │
│    Weight: 40%                          │
├─────────────────────────────────────────┤
│ 2. Content-Based                        │
│    "Similar to what you watched..."     │
│    Weight: 30%                          │
├─────────────────────────────────────────┤
│ 3. Trending/Popular                     │
│    "Everyone is watching..."            │
│    Weight: 15%                          │
├─────────────────────────────────────────┤
│ 4. Personalized Ranking (ML)            │
│    Deep learning model                  │
│    Weight: 15%                          │
└─────────────────────────────────────────┘
         ↓
  Final Score = Weighted Combination
         ↓
  Ranked List of Recommendations
```

Implementation:

```python
```

```
class HybridRecommender:
    def __init__(self):
        self.collaborative = ItemBasedCF()
        self.content_based = ContentBasedRecommender()
        self.popularity = {}

    def recommend(self, user_id, user_liked_items, n=10):
        """Hybrid recommendations"""
        recommendations = {}

        # 1. Collaborative filtering (40% weight)
        collab_recs = self.collaborative.recommend(user_id, n=20)
        for item_id, score in collab_recs:
            recommendations[item_id] = {
                'collab_score': score * 0.4,
                'content_score': 0,
                'popularity_score': 0,
                'total_score': 0
            }

        # 2. Content-based (30% weight)
        content_recs = self.content_based.recommend_for_user(user_liked_items, n=20)
        for item_id, score in content_recs:
            if item_id not in recommendations:
                recommendations[item_id] = {
                    'collab_score': 0,
                    'content_score': score * 0.3,
                    'popularity_score': 0,
                    'total_score': 0
                }
            else:
                recommendations[item_id]['content_score'] = score * 0.3

        # 3. Popularity (15% weight)
        for item_id in recommendations:
            pop_score = self.popularity.get(item_id, 0)
            recommendations[item_id]['popularity_score'] = pop_score * 0.15

        # 4. Calculate total scores
        for item_id in recommendations:
            rec = recommendations[item_id]
            rec['total_score'] = (
                rec['collab_score'] +
                rec['content_score'] +
                rec['popularity_score']
            )
```

```
        # Sort by total score
        sorted_recs = sorted(
            recommendations.items(),
            key=lambda x: x[1]['total_score'],
            reverse=True
        )

        return sorted_recs[:n]

# Usage
hybrid = HybridRecommender()

recommendations = hybrid.recommend('user123', ['movie1', 'movie3'], n=10)

print("Hybrid Recommendations:")
for item_id, scores in recommendations[:5]:
    print(f"{item_id}:")
    print(f"  Collaborative: {scores['collab_score']:.2f}")
    print(f"  Content: {scores['content_score']:.2f}")
    print(f"  Popularity: {scores['popularity_score']:.2f}")
    print(f"  Total: {scores['total_score']:.2f}")
```

---

4. Real-Time Recommendations

Update recommendations as user interacts.

```javascript
```

```
class RealTimeRecommender {
  constructor() {
    this.userProfiles = new Map();
    this.itemFeatures = new Map();
    this.recentInteractions = new Map();
  }

  async trackInteraction(userId, itemId, interactionType) {
    // Update user profile in real-time
    if (!this.userProfiles.has(userId)) {
      this.userProfiles.set(userId, {
        interests: {},
        recentItems: [],
        lastUpdate: Date.now()
      });
    }

    const profile = this.userProfiles.get(userId);

    // Get item features
    const item = this.itemFeatures.get(itemId);

    if (item) {
      // Update interest weights
      item.categories.forEach(category => {
        profile.interests[category] = (profile.interests[category] || 0) + 1;
      });

      // Track recent items
      profile.recentItems.push({
        itemId,
        timestamp: Date.now(),
        type: interactionType  // view, like, purchase
      });

      // Keep only last 50
      if (profile.recentItems.length > 50) {
        profile.recentItems.shift();
      }
    }

    // Trigger recommendation update
    await this.updateRecommendations(userId);
  }

  async updateRecommendations(userId) {
```

```
  const profile = this.userProfiles.get(userId);

  if (!profile) return;

  // Calculate scores for all items
  const scores = [];

  for (const [itemId, item] of this.itemFeatures) {
    // Skip recently interacted items
    const recentIds = profile.recentItems.map(i => i.itemId);
    if (recentIds.includes(itemId)) {
      continue;
    }

    // Calculate relevance score
    let score = 0;

    item.categories.forEach(category => {
      score += profile.interests[category] || 0;
    });

    // Recency boost (prefer newer items)
    const ageInDays = (Date.now() - item.createdAt) / (1000 * 60 * 60 * 24);
    const recencyBoost = Math.max(0, 1 - (ageInDays / 365));
    score *= (1 + recencyBoost);

    scores.push({ itemId, score });
  }

  // Sort by score
  scores.sort((a, b) => b.score - a.score);

  // Store top 100 recommendations in cache
  const topRecs = scores.slice(0, 100).map(s => s.itemId);

  await redis.setex(
    `recommendations:${userId}`,
    300, // 5 minute TTL
    JSON.stringify(topRecs)
  );

  console.log(`Updated recommendations for ${userId}`);
}

async getRecommendations(userId, n=10) {
  // Check cache
  const cached = await redis.get(`recommendations:${userId}`);
```

```javascript
    if (cached) {
      const recommendations = JSON.parse(cached);
      return recommendations.slice(0, n);
    }

    // Generate if not cached
    await this.updateRecommendations(userId);
    return await this.getRecommendations(userId, n);
  }
}

// Usage
const recommender = new RealTimeRecommender();

// User views product
await recommender.trackInteraction('user123', 'product456', 'view');
// → Updates user profile immediately
// → Regenerates recommendations

// User purchases product
await recommender.trackInteraction('user123', 'product789', 'purchase');
// → Higher weight than view
// → Recommendations updated again

// Get recommendations (real-time)
const recs = await recommender.getRecommendations('user123', 10);
console.log('Real-time recommendations:', recs);
```

Key Takeaways

1. **Collaborative Filtering:**
    - User-based: Find similar users
    - Item-based: Find similar items
    - Better for sparse data
    - Cold start problem
2. **Content-Based:**
    - Based on item features
    - No cold start for new users
    - Limited serendipity

3. **Hybrid:**
    - Combine multiple approaches
    - Better accuracy
    - Used by Netflix, Amazon
4. **Real-Time:**
    - Update as user interacts
    - Cache recommendations
    - Balance freshness vs computation

Practice Problems

1. Design Netflix recommendation system (100M users, 10K movies)
2. Design Amazon product recommendations (handle cold start)
3. Design YouTube video recommendations (billions of videos)

I've now created Chapters 20-23! Would you like me to continue with more chapters (24-30+) or would you prefer to start practicing system design problems to apply all this knowledge?