# Cours Machine Learning

Chapitre 14

## Systèmes de Recommandation

**Objectifs d'apprentissage :**

- Comprendre les types de systèmes de recommandation (content-based, collaborative filtering, hybrid)
- Maîtriser le collaborative filtering (user-based, item-based, matrix factorization)
- Implémenter des recommenders avec deep learning (NCF, autoencoders, two-tower models)
- Évaluer les systèmes avec les métriques appropriées (RMSE, Precision@K, NDCG)
- Résoudre les problèmes pratiques (cold start, sparsity, diversity)

**Prérequis :** Chapitres 01, 06, 14 (Fondamentaux Math, Réseaux de Neurones, Best Practices)

**Durée estimée :** 6-8 heures

**Notebooks :** `14_demo_*.ipynb`, `14_exercices.ipynb`

Cours ML - Sandbox-ML
Version 1.0 - 2026

# Table des matières

# 1 Introduction aux Systèmes de Recommandation

## 1.1 Motivation

Les systèmes de recommandation sont omniprésents dans notre quotidien numérique : Netflix suggère des films, Spotify recommande de la musique, Amazon propose des produits, YouTube suggère des vidéos. Ces systèmes sont cruciaux pour l'expérience utilisateur et génèrent une valeur économique considérable.

**Exemple : Impact business des recommandations**

- **Netflix** : 80% du contenu visionné provient de recommandations
- **Amazon** : 35% des ventes proviennent de recommandations de produits
- **YouTube** : 70% du temps de visionnage provient de recommandations
- **Spotify** : Les playlists personnalisées augmentent l'engagement de 40%

## 1.2 Problématique

**Définition : Système de Recommandation**

Un système de recommandation est un algorithme qui prédit la préférence ou le rating qu'un utilisateur donnerait à un item, dans le but de suggérer les items les plus pertinents. Formellement : étant donné un ensemble d'utilisateurs $U$, un ensemble d'items $I$, et une fonction de rating partielle $r : U \times I \to \mathbb{R}$, prédire $\hat{r}(u, i)$ pour les paires $(u, i)$ non observées.

## 1.3 Types de Systèmes de Recommandation

TABLE 1 – Taxonomie des systèmes de recommandation

| Type | Description |
|---|---|
| **Content-Based** | Recommande des items similaires à ceux appréciés par l'utilisateur (basé sur features des items) |
| **Collaborative Filtering** | Recommande des items aimés par des utilisateurs similaires (basé sur interactions user-item) |
| **Hybrid** | Combine content-based et collaborative filtering pour bénéficier des deux approches |
| **Knowledge-Based** | Utilise des règles explicites et connaissances du domaine |
| **Deep Learning** | Utilise des réseaux de neurones pour apprendre des représentations complexes |

# 2 Collaborative Filtering (Filtrage Collaboratif)

Le collaborative filtering est l'approche la plus populaire et repose sur l'hypothèse que les utilisateurs qui ont aimé les mêmes items dans le passé auront des goûts similaires dans le futur.

## 2.1 User-Item Matrix

**Définition : User-Item Matrix**

La matrice utilisateur-item $\mathbf{R} \in \mathbb{R}^{m \times n}$ contient les ratings de $m$ utilisateurs pour $n$ items :

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ r_{21} & r_{22} & \cdots & r_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ r_{m1} & r_{m2} & \cdots & r_{mn} \end{bmatrix}$$

où $r_{ui}$ est le rating de l'utilisateur $u$ pour l'item $i$. La plupart des entrées sont manquantes (matrice sparse).

**Exemple : Matrice User-Item (MovieLens)**

| | Titanic | Matrix | Inception | Avatar | Interstellar |
|---|---|---|---|---|---|
| Alice | 5 | ? | 4 | ? | 5 |
| Bob | 1 | 5 | ? | 3 | ? |
| Carol | ? | 4 | 5 | 4 | ? |
| Dave | 5 | 1 | ? | 5 | 4 |

Objectif : prédire les ratings manquants (?) pour recommander des films.

## 2.2 Mesures de Similarité

### 2.2.1 Similarité Cosine

**Définition : Similarité Cosine**

La similarité cosine entre deux vecteurs $\mathbf{u}$ et $\mathbf{v}$ est :

$$\text{sim}_{\cos}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|} = \frac{\sum_{i=1}^{n} u_i v_i}{\sqrt{\sum_{i=1}^{n} u_i^2}\sqrt{\sum_{i=1}^{n} v_i^2}}$$

Valeur entre -1 (opposés) et 1 (identiques), 0 signifie orthogonaux.

### 2.2.2 Corrélation de Pearson

**Définition : Corrélation de Pearson**

La corrélation de Pearson mesure la corrélation linéaire entre deux vecteurs :

$$\text{sim}_{\text{pearson}}(\mathbf{u}, \mathbf{v}) = \frac{\sum_i (u_i - \bar{u})(v_i - \bar{v})}{\sqrt{\sum_i (u_i - \bar{u})^2}\sqrt{\sum_i (v_i - \bar{v})^2}}$$

où $\bar{u}$ et $\bar{v}$ sont les moyennes. Capture mieux les préférences relatives.

### 2.2.3 Similarité Jaccard

Pour les données binaires (vu/non vu, acheté/non acheté) :

$$\text{sim}_{\text{jaccard}}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

## 2.3 User-Based Collaborative Filtering

**Définition : User-Based CF**

Prédire le rating de l'utilisateur $u$ pour l'item $i$ en agrégeant les ratings des $k$ utilisateurs les plus similaires à $u$ :

$$\hat{r}_{ui} = \bar{r}_u + \frac{\sum_{v \in N_k(u)} \text{sim}(u, v) \cdot (r_{vi} - \bar{r}_v)}{\sum_{v \in N_k(u)} |\text{sim}(u, v)|}$$

où :

- $N_k(u)$ : ensemble des $k$ utilisateurs les plus similaires à $u$ ayant raté l'item $i$
- $\bar{r}_u$ : rating moyen de l'utilisateur $u$
- $\text{sim}(u, v)$ : similarité entre utilisateurs $u$ et $v$

**Algorithm 1** User-Based Collaborative Filtering

**Require :** Matrice de ratings $\mathbf{R}$, utilisateur $u$, item $i$, nombre de voisins $k$

**Ensure :** Rating prédit $\hat{r}_{ui}$

1 : Calculer les similarités entre $u$ et tous les autres utilisateurs

2 : Identifier les $k$ utilisateurs les plus similaires ayant raté $i$ : $N_k(u)$

3 : Calculer le rating moyen de chaque utilisateur : $\bar{r}_u$, $\bar{r}_v$

4 : Calculer la prédiction pondérée :

$$\hat{r}_{ui} = \bar{r}_u + \frac{\sum_{v \in N_k(u)} \text{sim}(u, v) \cdot (r_{vi} - \bar{r}_v)}{\sum_{v \in N_k(u)} |\text{sim}(u, v)|}$$

5 : **return** $\hat{r}_{ui}$

### 2.3.1 Complexité User-Based CF

- **Temps (calcul similarités)** : $O(m^2 n)$ où $m$ = nb users, $n$ = nb items
- **Temps (prédiction)** : $O(mk)$ pour trouver $k$ voisins
- **Espace** : $O(m^2)$ pour stocker la matrice de similarité
- **Problème** : ne passe pas à l'échelle pour des millions d'utilisateurs

## 2.4 Item-Based Collaborative Filtering

**Définition : Item-Based CF**

Au lieu de trouver des utilisateurs similaires, on trouve des items similaires. Prédiction :

$$\hat{r}_{ui} = \frac{\sum_{j \in N_k(i)} \text{sim}(i, j) \cdot r_{uj}}{\sum_{j \in N_k(i)} |\text{sim}(i, j)|}$$

où $N_k(i)$ est l'ensemble des $k$ items les plus similaires à $i$ que l'utilisateur $u$ a ratés.

**Astuce**

**Item-Based vs User-Based :**

- **Item-Based** est souvent préféré car :
  - Les items changent moins que les utilisateurs (similarités stables)
  - Plus facile à pré-calculer et mettre en cache
  - Meilleure scalabilité pour beaucoup d'utilisateurs
- **User-Based** peut être meilleur si :
  - Il y a beaucoup plus d'items que d'utilisateurs
  - Les préférences utilisateurs sont très diversifiées

## 2.5 Matrix Factorization (Factorisation de Matrice)

L'approche de factorisation de matrice est plus avancée et performante que les approches basées sur les voisins.

**Définition : Matrix Factorization**

L'idée est de décomposer la matrice user-item $\mathbf{R} \in \mathbb{R}^{m \times n}$ en deux matrices de faible dimension :

$$\mathbf{R} \approx \mathbf{U}\mathbf{V}^T$$

où :

- $\mathbf{U} \in \mathbb{R}^{m \times k}$ : matrice des features latentes des utilisateurs
- $\mathbf{V} \in \mathbb{R}^{n \times k}$ : matrice des features latentes des items
- $k \ll \min(m, n)$ : nombre de facteurs latents (typiquement 10-200)

Chaque ligne $\mathbf{u}_i$ de $\mathbf{U}$ représente l'utilisateur $i$ dans l'espace latent.
Chaque ligne $\mathbf{v}_j$ de $\mathbf{V}$ représente l'item $j$ dans l'espace latent.
Prédiction : $\hat{r}_{ij} = \mathbf{u}_i \cdot \mathbf{v}_j$

**Exemple : Interprétation des facteurs latents (films)**

Avec $k = 2$ facteurs pour des films :

- **Facteur 1** : "Sérieux vs Léger" (drame vs comédie)
- **Facteur 2** : "Orienté action vs Romance"

Un utilisateur avec $\mathbf{u} = [0.9, -0.3]$ préfère les films sérieux sans action.
Un film avec $\mathbf{v} = [0.8, 0.6]$ est un drame d'action (ex : Dark Knight).
Rating prédit : $\hat{r} = 0.9 \times 0.8 + (-0.3) \times 0.6 = 0.72 - 0.18 = 0.54$

### 2.5.1 Optimisation : Singular Value Decomposition (SVD)

**Théorème : SVD**

Toute matrice $\mathbf{R} \in \mathbb{R}^{m \times n}$ peut être décomposée :

$$\mathbf{R} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

où :
- $\mathbf{U} \in \mathbb{R}^{m \times m}$ : vecteurs propres gauches (orthogonaux)
- $\mathbf{\Sigma} \in \mathbb{R}^{m \times n}$ : valeurs singulières diagonales ($\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$)
- $\mathbf{V} \in \mathbb{R}^{n \times n}$ : vecteurs propres droits (orthogonaux)

Approximation de rang $k$ : $\mathbf{R}_k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^T$ minimise l'erreur de Frobenius.

**Attention**

Le SVD classique ne fonctionne pas directement sur les matrices sparse (avec valeurs manquantes). On doit utiliser des variantes adaptées ou des algorithmes itératifs.

### 2.5.2 Alternating Least Squares (ALS)

ALS est un algorithme itératif pour la factorisation de matrice avec valeurs manquantes.

**Algorithm 2** Alternating Least Squares (ALS)

**Require :** Matrice sparse $\mathbf{R}$, nombre de facteurs $k$, régularisation $\lambda$, nb itérations $T$
**Ensure :** Matrices $\mathbf{U}$ et $\mathbf{V}$

Initialiser $\mathbf{U}$ et $\mathbf{V}$ aléatoirement
**for** $t = 1$ **to** $T$ **do**
  **Fix V, optimize U :**
  **for** chaque utilisateur $i$ **do**
    $\mathbf{u}_i \leftarrow \text{argmin}_{\mathbf{u}_i} \sum_{j:r_{ij} \text{ observed}} (r_{ij} - \mathbf{u}_i \cdot \mathbf{v}_j)^2 + \lambda \|\mathbf{u}_i\|^2$
    Solution : $\mathbf{u}_i = (\mathbf{V}_i^T \mathbf{V}_i + \lambda \mathbf{I})^{-1} \mathbf{V}_i^T \mathbf{r}_i$
  **end for**
  **Fix U, optimize V :**
  **for** chaque item $j$ **do**
    $\mathbf{v}_j \leftarrow \text{argmin}_{\mathbf{v}_j} \sum_{i:r_{ij} \text{ observed}} (r_{ij} - \mathbf{u}_i \cdot \mathbf{v}_j)^2 + \lambda \|\mathbf{v}_j\|^2$
    Solution : $\mathbf{v}_j = (\mathbf{U}_j^T \mathbf{U}_j + \lambda \mathbf{I})^{-1} \mathbf{U}_j^T \mathbf{r}_j$
  **end for**
**end for**
**return** $\mathbf{U}$, $\mathbf{V}$

où $\mathbf{V}_i$ contient les vecteurs des items ratés par l'utilisateur $i$, et $\mathbf{r}_i$ leurs ratings.

### 2.5.3 Gradient Descent pour Matrix Factorization

On peut aussi optimiser avec la descente de gradient (SGD) :

$$\text{Loss} = \sum_{(i,j) \in \text{observed}} (r_{ij} - \mathbf{u}_i \cdot \mathbf{v}_j)^2 + \lambda(\|\mathbf{u}_i\|^2 + \|\mathbf{v}_j\|^2) \tag{1}$$

$$\mathbf{u}_i \leftarrow \mathbf{u}_i + \alpha \cdot [2(r_{ij} - \hat{r}_{ij})\mathbf{v}_j - 2\lambda \mathbf{u}_i] \tag{2}$$

$$\mathbf{v}_j \leftarrow \mathbf{v}_j + \alpha \cdot [2(r_{ij} - \hat{r}_{ij})\mathbf{u}_i - 2\lambda \mathbf{v}_j] \tag{3}$$

### 2.5.4 SVD++ : Incorporating Implicit Feedback

SVD++ étend la factorisation en incorporant les feedbacks implicites (items vus mais non ratés) :

$$\hat{r}_{ui} = \mu + b_u + b_i + \left( \mathbf{p}_u + |I_u|^{-1/2} \sum_{j \in I_u} \mathbf{y}_j \right)^T \mathbf{q}_i$$

où :

- $\mu$ : rating moyen global
- $b_u, b_i$ : biais utilisateur et item
- $\mathbf{p}_u, \mathbf{q}_i$ : facteurs latents utilisateur/item
- $I_u$ : ensemble des items pour lesquels l'utilisateur $u$ a fourni un feedback implicite
- $\mathbf{y}_j$ : facteurs implicites pour l'item $j$

# 3 Deep Learning pour Systèmes de Recommandation

Les approches de deep learning permettent de capturer des interactions non-linéaires complexes entre utilisateurs et items.

## 3.1 Neural Collaborative Filtering (NCF)

**Définition : Neural Collaborative Filtering**

NCF remplace le produit scalaire de la factorisation de matrice par un réseau de neurones pour modéliser l'interaction entre utilisateurs et items :

$$\hat{r}_{ui} = f(\mathbf{u}_i, \mathbf{v}_j; \theta)$$

où $f$ est un MLP (Multi-Layer Perceptron) et $\theta$ ses paramètres.

### 3.1.1 Architecture NCF

```python
import torch
import torch.nn as nn

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

        # Embeddings
        self.user_embedding = nn.Embedding(n_users, embedding_dim)
        self.item_embedding = nn.Embedding(n_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):
        # Embeddings
        user_emb = self.user_embedding(user_ids)  # (batch, embedding_dim)
        item_emb = self.item_embedding(item_ids)  # (batch, embedding_dim)

        # Concatenate
        x = torch.cat([user_emb, item_emb], dim=-1)  # (batch, 2*embedding_dim)

        # MLP
        x = self.mlp(x)

        # Output
        rating = self.output(x).squeeze()  # (batch,)
        return rating
```

Listing 1 – Architecture NCF avec PyTorch

### 3.1.2 Variante : Generalized Matrix Factorization (GMF)

GMF utilise le produit élément par élément des embeddings :

$$\hat{r}_{ui} = \mathbf{h}^T(\mathbf{p}_u \odot \mathbf{q}_i)$$

où $\odot$ est le produit d'Hadamard (element-wise), et $\mathbf{h}$ est un vecteur de poids appris.

### 3.1.3 NeuMF : Fusion de GMF et MLP

**Définition : Neural Matrix Factorization (NeuMF)**

NeuMF combine GMF (linéaire) et MLP (non-linéaire) :

$$\hat{r}_{ui} = \sigma(\mathbf{h}^T[\text{GMF}(\mathbf{p}_u, \mathbf{q}_i) \oplus \text{MLP}(\mathbf{u}, \mathbf{v})]) \tag{4}$$
$$= \sigma(\mathbf{h}^T[(\mathbf{p}_u \odot \mathbf{q}_i) \oplus \phi(\mathbf{u}, \mathbf{v})]) \tag{5}$$

où $\oplus$ est la concaténation et $\sigma$ est sigmoid.

## 3.2 Autoencoders pour Collaborative Filtering

Les autoencoders peuvent apprendre des représentations compactes des préférences utilisateurs.

### 3.2.1 AutoRec

**Définition : AutoRec**

AutoRec est un autoencoder qui prend en entrée le vecteur de ratings d'un utilisateur (ou item) et reconstruit ce vecteur :

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{r} + \mathbf{b}) \quad \text{(encoder)} \tag{6}$$

$$\hat{\mathbf{r}} = \sigma(\mathbf{W}'\mathbf{h} + \mathbf{b}') \quad \text{(decoder)} \tag{7}$$

Loss : $L = \sum_{i \in \text{observed}} (\hat{r}_i - r_i)^2 + \lambda(\|\mathbf{W}\|^2 + \|\mathbf{W}'\|^2)$

```python
class AutoRec(nn.Module):
    def __init__(self, n_items, hidden_dim=128):
        super(AutoRec, self).__init__()

        self.encoder = nn.Sequential(
            nn.Linear(n_items, hidden_dim),
            nn.Sigmoid()
        )

        self.decoder = nn.Sequential(
            nn.Linear(hidden_dim, n_items),
            nn.Sigmoid()
        )

    def forward(self, ratings):
        # ratings: (batch, n_items) avec 0 pour items non rates
        h = self.encoder(ratings)  # (batch, hidden_dim)
        reconstructed = self.decoder(h)  # (batch, n_items)
        return reconstructed

    def loss(self, ratings, reconstructed, mask):
        # mask: 1 pour items rates, 0 sinon
        mse = torch.sum((ratings - reconstructed) ** 2 * mask) / torch.sum(mask)
        return mse
```

Listing 2 – AutoRec avec PyTorch

## 3.3 Two-Tower Model

Le modèle Two-Tower est utilisé pour la recommandation à grande échelle (ex : YouTube, Google).

**Définition : Two-Tower Model**

Architecture avec deux réseaux séparés :

- **User Tower** : encode les features utilisateur → embedding $\mathbf{u}$
- **Item Tower** : encode les features item → embedding $\mathbf{v}$

Score de recommandation : $s(u, i) = \mathbf{u} \cdot \mathbf{v}$ (produit scalaire)

Avantage : les embeddings items peuvent être pré-calculés et indexés pour une recherche rapide (Approximate Nearest Neighbors).

```python
class TwoTowerModel(nn.Module):
    def __init__(self, user_features_dim, item_features_dim, embedding_dim=128):
        super(TwoTowerModel, self).__init__()

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

        # Item Tower
        self.item_tower = nn.Sequential(
            nn.Linear(item_features_dim, 256),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(256, 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)

        # Dot product
        scores = torch.sum(user_emb * item_emb, dim=-1)  # (batch,)
        return scores

    def get_user_embedding(self, user_features):
        return self.user_tower(user_features)

    def get_item_embedding(self, item_features):
        return self.item_tower(item_features)
```

Listing 3 – Two-Tower Model

### 3.4 Deep Interest Network (DIN)

DIN utilise un mécanisme d'attention pour capturer l'intérêt de l'utilisateur en fonction du contexte.

**Définition : Deep Interest Network**

DIN applique une attention sur l'historique de l'utilisateur en fonction de l'item candidat :

$$\mathbf{u}_i = \sum_{j \in H_u} a(i,j) \cdot \mathbf{v}_j$$

où :

- $H_u$ : historique des items de l'utilisateur $u$
- $a(i,j)$ : score d'attention entre item candidat $i$ et item historique $j$
- $\mathbf{v}_j$ : embedding de l'item $j$

Le score d'attention est calculé par un petit réseau de neurones.

## 4 Content-Based Filtering (Filtrage par Contenu)

Les systèmes content-based recommandent des items similaires à ceux appréciés par l'utilisateur, basés sur les features des items.

### 4.1 Principe

**Définition : Content-Based Filtering**

Créer un profil utilisateur basé sur les features des items qu'il a aimés, puis recommander des items similaires.

1. **Item Profile** : représenter chaque item par un vecteur de features $\mathbf{i} \in \mathbb{R}^d$
2. **User Profile** : agréger les features des items aimés : $\mathbf{u} = \frac{1}{|I_u|} \sum_{i \in I_u} \mathbf{i}$
3. **Prédiction** : $\text{score}(u,i) = \text{sim}(\mathbf{u}, \mathbf{i})$ (ex : cosine similarity)

### 4.2 Représentation TF-IDF

Pour les items textuels (articles, films avec descriptions, produits) :

**Définition : TF-IDF**

TF-IDF (Term Frequency - Inverse Document Frequency) représente chaque document par un vecteur :

$$\text{TF}(t,d) = \frac{\text{nb occurrences de } t \text{ dans } d}{\text{nb total de termes dans } d} \tag{8}$$

$$\text{IDF}(t) = \log \frac{\text{nb total de documents}}{\text{nb documents contenant } t} \tag{9}$$

$$\text{TF-IDF}(t,d) = \text{TF}(t,d) \times \text{IDF}(t) \tag{10}$$

Les termes fréquents dans un document mais rares globalement ont un poids élevé.

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

# Exemple : recommandation de films basee sur les descriptions
movie_descriptions = [
    "Action movie with explosions and car chases",
    "Romantic comedy about love and relationships",
    "Sci-fi thriller with time travel and paradoxes",
    # ...
]

# Vectorization TF-IDF
vectorizer = TfidfVectorizer(stop_words='english')
tfidf_matrix = vectorizer.fit_transform(movie_descriptions)  # (n_movies, n_features)

# Similarite entre films
similarity_matrix = cosine_similarity(tfidf_matrix)  # (n_movies, n_movies)

# Recommander des films similaires au film 0
movie_idx = 0
similar_scores = similarity_matrix[movie_idx]
top_k_indices = similar_scores.argsort()[-6:-1][::-1]  # Top 5 (excluant le film lui-meme)
```

Listing 4 – Content-Based Filtering avec TF-IDF

### 4.3 Embeddings (Word2Vec, BERT)

Pour des représentations plus riches, on peut utiliser des embeddings pré-entraînés :

- **Word2Vec / GloVe** : moyenner les embeddings des mots
- **BERT / Sentence-BERT** : embeddings de phrases/documents complets
- **Modèles multimodaux** : combiner texte, images, métadonnées

### 4.4 Avantages et Limites du Content-Based

## 5 Systèmes Hybrides

Les systèmes hybrides combinent collaborative filtering et content-based pour bénéficier des deux approches.

### 5.1 Stratégies de Combinaison

### 5.2 Exemple : Hybrid Neural Model

```python
class HybridRecommender(nn.Module):
    def __init__(self, n_users, n_items, item_features_dim, emb_dim=64):
```

TABLE 2 – Content-Based : Avantages vs Limites

| **Avantages** | **Limites** |
|---|---|
| Pas de cold start pour nouveaux items (si features dispo) | Cold start pour nouveaux utilisateurs |
| Recommandations explicables | Manque de diversité (filter bubble) |
| Pas besoin de données d'autres utilisateurs | Nécessite des features de qualité |
| Fonctionne avec peu de données utilisateur | Ne capture pas les préférences émergentes |

TABLE 3 – Stratégies de systèmes hybrides

| **Stratégie** | **Description** |
|---|---|
| **Weighted** | Combiner les scores : $s = \alpha \cdot s_{\text{CF}} + (1 - \alpha) \cdot s_{\text{CB}}$ |
| **Switching** | Choisir une approche selon le contexte (ex : CF si assez de données, sinon CB) |
| **Mixed** | Présenter des recommandations des deux approches ensemble |
| **Feature Combination** | Utiliser les features content comme entrées du CF |
| **Cascade** | Raffiner les résultats d'une approche avec l'autre |
| **Meta-level** | Utiliser le modèle d'une approche comme entrée de l'autre |

```
        super(HybridRecommender, self).__init__()

        # Collaborative Filtering part
        self.user_embedding = nn.Embedding(n_users, emb_dim)
        self.item_embedding = nn.Embedding(n_items, emb_dim)

        # Content-Based part
        self.item_content_encoder = nn.Sequential(
            nn.Linear(item_features_dim, 128),
            nn.ReLU(),
            nn.Linear(128, emb_dim)
        )

        # Fusion layer
        self.fusion = nn.Sequential(
            nn.Linear(emb_dim * 3, 128),  # user_emb + item_emb +
content_emb
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 1)
        )

```

```
    def forward(self, user_ids, item_ids, item_features):
        # Collaborative embeddings
        user_emb = self.user_embedding(user_ids)  # (batch, emb_dim)
        item_emb = self.item_embedding(item_ids)  # (batch, emb_dim)

        # Content embeddings
        content_emb = self.item_content_encoder(item_features)  # (batch, emb_dim)

        # Concatenate all
        x = torch.cat([user_emb, item_emb, content_emb], dim=-1)  # (batch, 3*emb_dim)

        # Predict
        rating = self.fusion(x).squeeze()  # (batch,)
        return rating
```

Listing 5 – Modele hybride neural

# 6 Métriques d'Évaluation

## 6.1 Métriques de Prédiction de Rating

### 6.1.1 Root Mean Squared Error (RMSE)

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

où $\Omega$ est l'ensemble des ratings de test.

### 6.1.2 Mean Absolute Error (MAE)

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

MAE est moins sensible aux outliers que RMSE.

## 6.2 Métriques de Ranking (Top-K)

Pour évaluer les recommandations top-K, on utilise des métriques de ranking.

### 6.2.1 Precision@K et Recall@K

**Définition : Precision@K et Recall@K**

Pour un utilisateur $u$, soit $R_u$ l'ensemble des items pertinents (ex : ratés 4+) et $T_u$ les top-K items recommandés :

$$\text{Precision@K} = \frac{|R_u \cap T_u|}{K} \tag{11}$$

$$\text{Recall@K} = \frac{|R_u \cap T_u|}{|R_u|} \tag{12}$$

**Precision@K** : proportion d'items pertinents parmi les K recommandés.
**Recall@K** : proportion d'items pertinents qui ont été recommandés.

### 6.2.2 F1@K

Moyenne harmonique de Precision@K et Recall@K :

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

### 6.2.3 Mean Average Precision (MAP)

**Définition : Average Precision (AP)**

Pour un utilisateur, AP est la moyenne des précisions calculées à chaque position pertinente :

$$\text{AP@K} = \frac{1}{|R_u|} \sum_{k=1}^{K} \text{Precision@k} \cdot \text{rel}(k)$$

où $\text{rel}(k) = 1$ si l'item en position $k$ est pertinent, 0 sinon.
MAP est la moyenne des AP sur tous les utilisateurs :

$$\text{MAP@K} = \frac{1}{|U|} \sum_{u \in U} \text{AP@K}_u$$

### 6.2.4 Normalized Discounted Cumulative Gain (NDCG)

NDCG prend en compte l'ordre des recommandations et permet des relevances graduelles.

**Définition : NDCG@K**

$$\text{DCG@K} = \sum_{i=1}^{K} \frac{2^{\text{rel}_i} - 1}{\log_2(i+1)} \tag{13}$$

$$\text{IDCG@K} = \text{DCG@K avec classement parfait} \tag{14}$$

$$\text{NDCG@K} = \frac{\text{DCG@K}}{\text{IDCG@K}} \tag{15}$$

où $\text{rel}_i$ est la relevance de l'item en position $i$ (ex : rating réel).
NDCG est entre 0 (pire) et 1 (parfait).

## 6.3 Métriques de Diversité et Coverage

### 6.3.1 Coverage

**Définition : Catalog Coverage**

Proportion d'items recommandés au moins une fois :

$$\text{Coverage} = \frac{|\bigcup_{u \in U} T_u|}{|I|}$$

Une bonne coverage signifie que le système ne recommande pas toujours les mêmes items populaires.

### 6.3.2 Diversity

**Définition : Intra-List Diversity**

Mesure la diversité au sein des K recommandations d'un utilisateur :

$$\text{Diversity}(T_u) = \frac{2}{K(K-1)} \sum_{i,j \in T_u, i \neq j} (1 - \text{sim}(i,j))$$

Plus les items sont différents, plus la diversité est élevée.

### 6.3.3 Novelty

**Définition : Novelty**

Capacité à recommander des items peu connus :

$$\text{Novelty}(T_u) = -\frac{1}{K} \sum_{i \in T_u} \log_2 p(i)$$

où $p(i)$ est la popularité de l'item $i$ (proportion d'utilisateurs l'ayant raté).

## 6.4 Évaluation Offline vs Online

TABLE 4 – Évaluation offline vs online

| | **Offline** | **Online (A/B Test)** |
|---|---|---|
| **Données** | Historique (train/test split) | Utilisateurs réels en temps réel |
| **Métriques** | RMSE, Precision@K, NDCG | CTR, conversion, engagement, revenue |
| **Coût** | Faible | Élevé (risque business) |
| **Rapidité** | Rapide (itérations) | Lent (plusieurs semaines) |
| **Réalisme** | Limité (biais historique) | Élevé (comportement réel) |

**Astuce**

**Stratégie d'évaluation recommandée :**

1. **Offline** : tester rapidement plusieurs modèles (RMSE, NDCG)
2. **Offline advanced** : simulation avec replay ou counterfactual
3. **Online (A/B test)** : comparer les meilleurs modèles en production sur un petit % d'utilisateurs
4. **Gradual rollout** : déployer progressivement le meilleur modèle

# 7 Problèmes Pratiques et Solutions

## 7.1 Cold Start Problem

**Définition : Cold Start Problem**

Difficulté à faire des recommandations pertinentes pour :

— **Nouveaux utilisateurs** : pas d'historique de ratings
— **Nouveaux items** : pas de ratings reçus
— **Nouveau système** : peu de données globalement

### 7.1.1 Solutions pour Cold Start

Table 5 – Solutions au cold start

| Approche | Description |
|---|---|
| **Content-Based** | Utiliser les features des items pour nouveaux items, ou les features démographiques pour nouveaux users |
| **Hybrid** | Combiner CF et content-based (CF pour users établis, CB pour nouveaux) |
| **Popularité** | Recommander les items les plus populaires aux nouveaux utilisateurs |
| **Onboarding** | Demander explicitement les préférences initiales (ex : Netflix demande de rater 3 films) |
| **Transfer Learning** | Utiliser des connaissances d'un domaine similaire (ex : musique → podcasts) |
| **Meta-Learning** | Apprendre à recommander rapidement avec peu de données |

## 7.2 Sparsity (Matrice Creuse)

**Définition : Sparsity**

La matrice user-item est très sparse : la plupart des utilisateurs n'ont raté qu'une infime fraction des items.
Exemple MovieLens : 100,000 ratings pour 1,682 films et 943 users → densité $= \frac{100{,}000}{1{,}682 \times 943} \approx 6.3\%$

#### 7.2.1 Solutions

— **Matrix Factorization** : capture les patterns sous-jacents mieux que les approches basées voisins
— **Implicit Feedback** : utiliser des données implicites (vues, clics, temps passé) en plus des ratings explicites
— **Regularization** : éviter l'overfitting sur les rares ratings observés
— **Side Information** : incorporer des features utilisateurs/items (hybrid)

### 7.3 Scalability (Passage à l'Échelle)

#### 7.3.1 Défis

— **Millions d'utilisateurs** : impossible de stocker toute la matrice de similarité ($O(m^2)$)
— **Millions d'items** : recherche exhaustive trop lente pour top-K
— **Temps réel** : latence $<$ 100ms pour recommandations

#### 7.3.2 Solutions

Table 6 – Solutions pour la scalabilité

| Technique | Description |
|---|---|
| **Item-Based CF** | Similarités items plus stables que users, pré-calcul possible |
| **ALS** | Parallélisable sur Spark/Hadoop pour grandes matrices |
| **ANN (Approx NN)** | FAISS, Annoy, ScaNN pour recherche rapide de voisins ($O(\log n)$ au lieu de $O(n)$) |
| **Two-Tower** | Pré-calculer embeddings items, recherche ANN en temps réel |
| **Candidate Generation + Ranking** | Pipeline 2 étapes : générer 100-1000 candidats rapides, puis ranker finement |
| **Caching** | Mettre en cache les recommandations pour utilisateurs actifs |

### 7.4 Diversity vs Accuracy Trade-off

**Attention**

Maximiser uniquement l'accuracy (RMSE, NDCG) peut conduire à :
— **Filter bubble** : recommander toujours le même type de contenu
— **Popularité bias** : recommander principalement les items populaires
— **Manque de découverte** : utilisateurs ne découvrent pas de nouveaux contenus

#### 7.4.1 Solutions

— **Re-ranking** : après avoir généré les top-K, re-ranker pour diversifier
— **MMR (Maximal Marginal Relevance)** : maximiser la pertinence tout en minimisant la similarité intra-liste

— **Calibration** : s'assurer que les recommandations reflètent la distribution des préférences de l'utilisateur
— **Exploration** : introduire occasionnellement des items aléatoires ou peu connus (multi-armed bandit)

### 7.5 Fairness et Biais

#### 7.5.1 Types de Biais

— **Popularité bias** : les items populaires sont sur-recommandés (rich get richer)
— **Position bias** : les utilisateurs cliquent plus sur les premiers items affichés
— **Biais démographiques** : sous-représentation de certains groupes
— **Feedback loops** : recommandations → interactions → renforcement du biais

#### 7.5.2 Solutions

— **Débiasing** : pondérer les données pour corriger les biais (inverse propensity scoring)
— **Fairness constraints** : contraindre le modèle à être équitable (ex : parité de recommandations par groupe)
— **Exploration** : recommander activement des items sous-exposés
— **Transparency** : expliquer pourquoi un item est recommandé

## 8 Applications Pratiques

### 8.1 E-commerce (Amazon, Alibaba)

— **Produits recommandés** : "Customers who bought X also bought Y"
— **Session-based** : recommandations basées sur la session en cours (séquences)
— **Bundles** : recommander des groupes de produits complémentaires
— **Métriques** : CTR (click-through rate), conversion rate, revenue

### 8.2 Streaming Vidéo (Netflix, YouTube)

— **Content recommendations** : films/séries basés sur historique
— **Continue watching** : reprendre là où l'utilisateur s'est arrêté
— **Thumbnails personnalisés** : choisir la vignette selon les préférences de l'utilisateur
— **Two-stage** : candidate generation (retrieval) + ranking
— **Métriques** : watch time, retention, engagement

### 8.3 Streaming Musical (Spotify, Apple Music)

— **Discover Weekly** : playlist personnalisée hebdomadaire
— **Radio** : générer une playlist infinie basée sur une chanson/artiste
— **Audio features** : tempo, énergie, valence, danceability
— **Sequential models** : prendre en compte l'ordre des écoutes (RNN, Transformers)

### 8.4 Réseaux Sociaux (Facebook, LinkedIn, TikTok)

— **News feed ranking** : classer les posts par pertinence

— **Friend/connection suggestions** : link prediction
— **Ads targeting** : publicités personnalisées
— **Challenges** : biais de confirmation, filter bubbles, fake news

### 8.5 Publicité en Ligne (Google Ads, Facebook Ads)

— **CTR prediction** : prédire la probabilité de clic
— **Conversion prediction** : prédire l'achat
— **Bidding optimization** : optimiser les enchères en temps réel (RTB)
— **Métriques** : CTR, CPC (cost per click), ROAS (return on ad spend)

## 9 Implémentation Complète d'un Système de Recommandation

### 9.1 Pipeline End-to-End

```python
# 1. Data Loading et Preprocessing
import pandas as pd
from sklearn.model_selection import train_test_split

# Charger les donnees (ex: MovieLens)
ratings = pd.read_csv('ratings.csv')  # user_id, item_id, rating, timestamp
users = pd.read_csv('users.csv')  # user_id, age, gender, occupation
items = pd.read_csv('items.csv')  # item_id, title, genres

# Train/Test split (temporal ou random)
train, test = train_test_split(ratings, test_size=0.2, random_state=42)

# 2. Matrix Factorization avec Surprise
from surprise import SVD, Dataset, Reader
from surprise.model_selection import cross_validate

reader = Reader(rating_scale=(1, 5))
data = Dataset.load_from_df(train[['user_id', 'item_id', 'rating']], reader)

# Entrainement SVD
svd = SVD(n_factors=100, n_epochs=20, lr_all=0.005, reg_all=0.02)
cross_validate(svd, data, measures=['RMSE', 'MAE'], cv=5, verbose=True)

# Fit sur toutes les donnees train
trainset = data.build_full_trainset()
svd.fit(trainset)

# 3. Predictions
def predict_rating(user_id, item_id):
    return svd.predict(user_id, item_id).est

# 4. Top-K Recommendations
def recommend_top_k(user_id, k=10):
```

```
    # Items non rates par l'utilisateur
    rated_items = set(train[train['user_id'] == user_id]['item_id'])
    all_items = set(items['item_id'])
    candidates = all_items - rated_items

    # Predire pour tous les candidats
    predictions = [(item_id, predict_rating(user_id, item_id))
                   for item_id in candidates]

    # Trier et retourner top-K
    predictions.sort(key=lambda x: x[1], reverse=True)
    return predictions[:k]

# Exemple
top_10 = recommend_top_k(user_id=42, k=10)
print(f"Top 10 recommendations for user 42: {top_10}")

# 5. Evaluation sur test set
from sklearn.metrics import mean_squared_error, mean_absolute_error

y_true = test['rating'].values
y_pred = [predict_rating(row['user_id'], row['item_id'])
          for _, row in test.iterrows()]

rmse = mean_squared_error(y_true, y_pred, squared=False)
mae = mean_absolute_error(y_true, y_pred)
print(f"Test RMSE: {rmse:.4f}, MAE: {mae:.4f}")
```

Listing 6 – Pipeline complet de recommandation

## 9.2 Évaluation Ranking (Precision@K, NDCG)

```
import numpy as np
from sklearn.metrics import ndcg_score

def precision_at_k(recommendations, relevant_items, k):
    """
    recommendations: liste ordonnee d'item_ids recommandes
    relevant_items: set d'item_ids pertinents (ex: rating >= 4)
    """
    top_k = recommendations[:k]
    relevant_in_top_k = len(set(top_k) & relevant_items)
    return relevant_in_top_k / k

def recall_at_k(recommendations, relevant_items, k):
    top_k = recommendations[:k]
    relevant_in_top_k = len(set(top_k) & relevant_items)
    return relevant_in_top_k / len(relevant_items) if len(relevant_items
    ) > 0 else 0

def ndcg_at_k(recommendations, true_relevances, k):
```

```
    """
    recommendations: liste ordonnee d'item_ids recommandes
    true_relevances: dict {item_id: relevance_score}
    """
    # Construire le vecteur de relevances pour les top-K recommandes
    relevances = [true_relevances.get(item_id, 0) for item_id in
    recommendations[:k]]

    # Ideal ranking (trier par relevance)
    ideal_relevances = sorted(true_relevances.values(), reverse=True)[:k
    ]

    # Calculer NDCG
    if sum(ideal_relevances) == 0:
        return 0

    dcg = sum([(2**rel - 1) / np.log2(i + 2) for i, rel in enumerate(
    relevances)])
    idcg = sum([(2**rel - 1) / np.log2(i + 2) for i, rel in enumerate(
    ideal_relevances)])

    return dcg / idcg if idcg > 0 else 0

# Exemple
user_id = 42
recommendations = [item_id for item_id, score in recommend_top_k(user_id
    , k=20)]
relevant_items = set(test[(test['user_id'] == user_id) & (test['rating']
    >= 4)]['item_id'])

print(f"Precision@10: {precision_at_k(recommendations, relevant_items,
    10):.4f}")
print(f"Recall@10: {recall_at_k(recommendations, relevant_items, 10):.4f
    }")

# Pour NDCG, on a besoin des relevances reelles
true_relevances = test[test['user_id'] == user_id].set_index('item_id')[
    'rating'].to_dict()
print(f"NDCG@10: {ndcg_at_k(recommendations, true_relevances, 10):.4f}")
```

Listing 7 – Calcul des metriques de ranking

# 10 Résumé du Chapitre

## 10.1 Points Clés

— **Collaborative Filtering** : recommander basé sur les utilisateurs/items similaires (user-based, item-based, matrix factorization)

— **Matrix Factorization** : décomposer la matrice user-item en facteurs latents (SVD, ALS), plus efficace que les approches basées voisins

— **Deep Learning** : NCF, autoencoders, two-tower models pour capturer des interactions complexes
— **Content-Based** : recommander basé sur les features des items (TF-IDF, embeddings)
— **Hybrid Systems** : combiner CF et content-based pour robustesse
— **Métriques** : RMSE/MAE pour rating prediction, Precision@K/Recall@K/NDCG pour ranking
— **Défis pratiques** : cold start, sparsity, scalability, diversity, fairness

### 10.2 Formules Essentielles

Formules à retenir

$$\text{User-Based CF :} \quad \hat{r}_{ui} = \bar{r}_u + \frac{\sum_{v \in N_k(u)} \text{sim}(u,v)(r_{vi} - \bar{r}_v)}{\sum_{v \in N_k(u)} |\text{sim}(u,v)|} \tag{16}$$

$$\text{Matrix Factorization :} \quad \mathbf{R} \approx \mathbf{U}\mathbf{V}^T, \quad \hat{r}_{ui} = \mathbf{u}_i \cdot \mathbf{v}_j \tag{17}$$

$$\text{Cosine Similarity :} \quad \text{sim}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|} \tag{18}$$

$$\text{NDCG@K :} \quad \text{NDCG} = \frac{\sum_{i=1}^{K} \frac{2^{\text{rel}_i} - 1}{\log_2(i+1)}}{\text{IDCG}} \tag{19}$$

$$\text{Precision@K :} \quad \frac{|R_u \cap T_u|}{K} \tag{20}$$

### 10.3 Comparaison des Approches

Table 7 – Comparaison des approches de recommandation

| Approche | Cold Start | Scalabilité | Accuracy | Explicabilité |
|---|---|---|---|---|
| User-Based CF | Mauvais | Faible | Moyen | Bon |
| Item-Based CF | Mauvais | Moyen | Moyen | Bon |
| Matrix Factorization | Mauvais | Bon | Élevé | Faible |
| Deep Learning | Mauvais | Bon | Très élevé | Très faible |
| Content-Based | Bon (items) | Bon | Moyen | Très bon |
| Hybrid | Bon | Bon | Très élevé | Moyen |

## 11 Exercices

### 11.1 Questions de Compréhension

1. Expliquez la différence entre user-based et item-based collaborative filtering. Dans quels contextes préférer l'un à l'autre ?
2. Pourquoi la matrice user-item est-elle sparse ? Quelles sont les conséquences pour les algorithmes de recommandation ?
3. Décrivez l'intuition derrière la matrix factorization. Que représentent les facteurs latents ?
4. Comparez les métriques RMSE et NDCG. Quand utiliser l'une plutôt que l'autre ?

5. Qu'est-ce que le cold start problem ? Proposez 3 solutions différentes.
6. Pourquoi le popularity bias est-il problématique dans les systèmes de recommandation ? Comment le réduire ?
7. Expliquez l'architecture d'un two-tower model. Quel est son principal avantage pour la scalabilité ?
8. Quelle est la différence entre évaluation offline et online (A/B testing) ? Quels sont les avantages et inconvénients de chaque approche ?

## 11.2 Exercices Pratiques

1. **Recommandation de films avec Collaborative Filtering**
   — Charger le dataset MovieLens 100K
   — Implémenter user-based CF avec similarité cosine
   — Implémenter item-based CF
   — Comparer les deux approches (RMSE, temps de calcul)
2. **Matrix Factorization avec ALS**
   — Utiliser la bibliothèque Surprise (SVD, NMF)
   — Tuner les hyperparamètres (nb facteurs, régularisation)
   — Évaluer avec cross-validation (RMSE, MAE)
   — Visualiser les embeddings avec t-SNE
3. **Système de recommandation avec Deep Learning**
   — Implémenter NCF avec PyTorch
   — Entraîner sur MovieLens
   — Évaluer avec Precision@K, Recall@K, NDCG@K
   — Comparer avec SVD classique
4. **Content-Based Filtering**
   — Créer un recommender basé sur les descriptions/genres de films
   — Utiliser TF-IDF pour vectoriser les features textuelles
   — Recommander des films similaires avec cosine similarity
   — Évaluer la diversité des recommandations
5. **Système Hybride**
   — Combiner collaborative filtering et content-based
   — Tester différentes stratégies de combinaison (weighted, switching)
   — Comparer performance vs systèmes individuels
   — Analyser les cas où le système hybride est meilleur

*Solutions complètes disponibles dans les notebooks :*

— `14_demo_collaborative_filtering.ipynb`
— `14_demo_neural_recommenders.ipynb`
— `14_exercices.ipynb`

# 12 Pour Aller Plus Loin

## 12.1 Lectures Recommandées

### 12.1.1 Articles Fondateurs

- **Matrix Factorization Techniques for Recommender Systems** (Koren et al., 2009) - IEEE Computer
- **Neural Collaborative Filtering** (He et al., 2017) - WWW 2017
- **Deep Neural Networks for YouTube Recommendations** (Covington et al., 2016) - RecSys 2016
- **Wide & Deep Learning for Recommender Systems** (Cheng et al., 2016) - DLRS 2016

### 12.1.2 Livres

- *Recommender Systems : The Textbook* - Charu C. Aggarwal (2016)
- *Practical Recommender Systems* - Kim Falk (2019)
- *Deep Learning for Search* - Tommaso Teofili (2019)

## 12.2 Bibliothèques et Frameworks

Table 8 – Bibliothèques pour systèmes de recommandation

| Bibliothèque | Description |
|---|---|
| **Surprise** | scikit pour recommandation (SVD, KNN, NMF) |
| **LightFM** | Hybrid recommenders (CF + content) |
| **Implicit** | Collaborative filtering pour implicit feedback (ALS) |
| **RecBole** | Framework PyTorch pour research en recommandation |
| **TensorFlow Recommenders** | Modèles DL pour recommandation (Google) |
| **FAISS** | Approximate nearest neighbors (Facebook AI) |
| **Annoy** | ANN pour embeddings (Spotify) |

## 12.3 Datasets

- **MovieLens** : 100K, 1M, 10M, 25M ratings de films
- **Amazon Reviews** : millions de reviews de produits
- **Netflix Prize** : 100M ratings (historique)
- **Spotify Million Playlist** : playlists et interactions
- **Yelp Dataset** : reviews de restaurants/services
- **Last.fm** : écoutes musicales
- **Book-Crossing** : ratings de livres

## 12.4 Compétitions et Challenges

- **RecSys Challenge** : compétition annuelle (Twitter, Spotify, etc.)
- **Kaggle** : diverses compétitions de recommandation

— **WSDM Cup** : recommandation et web mining

### 12.5 Sujets Avancés

1. **Sequential Recommendations** : modéliser les séquences (RNN, Transformers, SASRec)
2. **Session-Based** : recommander basé sur la session en cours (GRU4Rec)
3. **Multi-Armed Bandits** : équilibrer exploration/exploitation en ligne
4. **Contextual Recommendations** : incorporer le contexte (temps, localisation, device)
5. **Cross-Domain Recommendations** : transférer des connaissances entre domaines
6. **Conversational Recommendations** : systèmes de recommandation interactifs
7. **Explainable AI for RecSys** : rendre les recommandations interprétables
8. **Causal Inference** : débiaser les recommandations avec inférence causale

### 12.6 Prochaines Étapes

— **Chapitre 15** : Natural Language Processing (NLP)
— **Chapitre 16** : Computer Vision Avancée
— **Projets** : construire un système de recommandation end-to-end en production

## Références

1. Koren, Y., Bell, R., & Volinsky, C. (2009). *Matrix Factorization Techniques for Recommender Systems*. IEEE Computer, 42(8), 30-37.
2. He, X., Liao, L., Zhang, H., Nie, L., Hu, X., & Chua, T. S. (2017). *Neural Collaborative Filtering*. WWW 2017.
3. Covington, P., Adams, J., & Sargin, E. (2016). *Deep Neural Networks for YouTube Recommendations*. RecSys 2016.
4. Cheng, H. T., et al. (2016). *Wide & Deep Learning for Recommender Systems*. DLRS 2016.
5. Aggarwal, C. C. (2016). *Recommender Systems : The Textbook*. Springer.
6. Ricci, F., Rokach, L., & Shapira, B. (2015). *Recommender Systems Handbook* (2nd ed.). Springer.
7. Hu, Y., Koren, Y., & Volinsky, C. (2008). *Collaborative Filtering for Implicit Feedback Datasets*. ICDM 2008.
8. Zhou, G., et al. (2018). *Deep Interest Network for Click-Through Rate Prediction*. KDD 2018.