Université Tunis-Dauphine

Master Big Data et Intelligence Artificielle

Projet Machine Learning

Étudier les systèmes de recommandation

Réalisé par : Fatma Chahed, Aziz Dhif, Oubeid Allah Jemli et Ghassen Rhouma
Année universitaire : 2024–2025

3 juin 2025

Partie 1

Remarque

Les **questions demandées** dans le projet sont réécrites dans une couleur bleue spécifique afin de les distinguer clairement.

Les **réponses** sont rédigées en noir pour une meilleure lisibilité.

*Tous les **fichiers** contenant les codes sources sont joints en pièces jointes dans l'e-mail envoyé avec ce rapport.*

a) C'est quoi le filtrage collaboratif ?

Le filtrage collaboratif est une technique largement utilisée dans les systèmes de recommandation personnalisés, qui prédit les préférences d'un utilisateur à partir des préférences des autres. Contrairement aux systèmes non personnalisés qui recommandent les mêmes éléments à tout le monde, le filtrage collaboratif fournit des suggestions adaptées en exploitant les similitudes soit entre les utilisateurs, soit entre les éléments. L'idée principale est que les utilisateurs ayant des goûts similaires auront tendance à aimer des choses similaires. Cette méthode utilise généralement une matrice d'utilité et des fonctions de similarité pour générer des recommandations personnalisées.

b) Combien de types de filtrage collaboratif peut on utiliser ?

Il existe deux types de filtrage collaboratif :

1. **Filtrage Collaboratif Utilisateur-à-Utilisateur (user to user) :**
   - Identifie les utilisateurs ayant des schémas de notation similaires.
   - Recommande les éléments appréciés par ces utilisateurs similaires.
   - Suppose que des utilisateurs similaires aimeront les mêmes éléments.
2. **Filtrage Collaboratif Élément-à-Élément(item to item) :**
   - Identifie les éléments ayant reçu des évaluations similaires de la part des utilisateurs.
   - Recommande des éléments similaires à ceux que l'utilisateur a déjà appréciés.
   - Plus stable lorsque les préférences des utilisateurs sont rares ou abstraites.

c) Quelle fonction de similarité a été utilisée ?

La fonction de similarité utilisée est la **similarité cosinus**. Elle permet de mesurer à quel point deux utilisateurs (ou deux films) se ressemblent dans leurs notes. Plus le résultat est proche de 1, plus ils ont des goûts similaires.

d) Pourquoi, dans le code de la fonction `find_neighbor()`, commence-t-on par 1 et non par 0 ?

— Analyse de ce code :

```
closest_neighbor = user_removed_mean_rating.index[sorted_idx
    [1:k + 1]].tolist()
```

L'indice 0 dans `sorted_idx` correspond à l'utilisateur cible, car sa similarité avec lui-même est la plus élevée (toujours égale à 1). Il faut donc l'exclure de la liste de ses voisins, car le garder introduirait un biais.

Expliquer pourquoi ce code ne permet pas d'obtenir le résultat voulu par l'auteur et comment peut on l'améliorer ?

— Le code suivant :

```
user_mean = data.mean(axis=0)
user_removed_mean_rating = (data - user_mean).fillna(0)
```

centre les notes par film, en soustrayant la note moyenne de chaque film. Cela suit une logique basée sur les éléments (item-based).

— Or, puisque l'algorithme effectue un filtrage collaboratif utilisateur-utilisateur, la similarité entre utilisateurs doit être basée sur leur écart par rapport à leur propre moyenne, et non celle des films.

— Pour améliorer cela, on peut remplacer la ligne précédente par un centrage par utilisateur :

```
user_mean = data.mean(axis=1)
user_removed_mean_rating = (data.T - user_mean).T.fillna(0)
```

e) Que veut dire `recommande_seen = True` ?

Le paramètre `recommande_seen=True` signifie :

"Inclure dans les recommandations les éléments que l'utilisateur a déjà vus ou consultés."

Si ce paramètre est défini sur `False`, la fonction exclut les éléments déjà vus, ce qui est souvent plus utile dans les systèmes réels qui cherchent à suggérer du contenu nouveau.

f) Comment peut on ajuster la fonction de similarité ou citer d'autres fonctions de similarité ?

— Avant de changer la fonction de similarité, il est essentiel de bien centrer les données.
— Dans le filtrage basé sur les utilisateurs, cela signifie centrer les notes ligne par ligne (par utilisateur), et non colonne par colonne.
— Une fois les données centrées, on peut adapter la fonction de similarité.

Fonctions de similarité courantes :

- **Similarité de Jaccard :** utilisée pour des données binaires (j'aime / je n'aime pas). Basée sur l'intersection divisée par l'union des éléments notés.
- **Similarité cosinus ajustée :** variation de la similarité cosinus, avec des notes centrées par moyenne utilisateur (ou élément).
- **Distance Euclidienne :** mesure la distance directe entre les vecteurs. Moins adaptée car elle ne normalise pas.

Améliorations possibles :

— Imposer un nombre minimum de notes communes (ex. au moins 3 en commun).
— Pondérer la similarité selon le nombre d'éléments co-notés.
— Normaliser les vecteurs pour éviter les biais liés aux utilisateurs très actifs.

g) Test du code

— Nous avons suivi les étapes suivantes :

1. Télécharger les données liées à Amazon.
2. Copier le code présent dans l'article dans un notebook.
3. Exécuter le code.
4. Apporter les modifications nécessaires pour assurer le bon fonctionnement du code.
5. Tester différents scénarios de recommandations pour l'utilisateur ayant l'identifiant `A1CV1WROP5KTTW`.

NB

Le code de cette partie, avec les modifications apportées, sera disponible en pièce jointe sous le nom "Code_Part1.ipynb".

— Voici quelques modifications apportées au code afin de pouvoir tester les recommandations des utilisateurs en utilisant le dataset d'Amazon :

1. Nous avons stocké une version des données sans la colonne `user_id` dans une nouvelle variable `rating_data`, afin de pouvoir effectuer des calculs comme la moyenne par film :

```
# S parer user_id et les donn es de notation
rating_data = data.drop(columns=["user_id"])
```

```
#  tape  1 : Calculer la moyenne des notes pour chaque
   film
user_mean = rating_data.mean(axis=0)
```

2. Nous avons réassigné `user_id` comme index afin de pouvoir accéder aux lignes par identifiant utilisateur — ce qui est nécessaire pour le système de recommandation basé sur `data.loc[userid]` :

```
#  tape  2 : Soustraire la moyenne des films des notes de
    chaque utilisateur
user_removed_mean_rating = (rating_data - user_mean).
   fillna(0)

# R assigner user_id comme index pour aligner les
   utilisateurs
user_removed_mean_rating.index = data["user_id"]
```

3. Nous avons utilisé `user_id` comme index dans le dataframe principal `data` pour pouvoir faire les recommandations ultérieurement :

```
# D finir user_id comme index du dataframe principal
data.set_index("user_id", inplace=True)
```

Vous pouvez voir la version finale de notre code dans le fichier envoyé.

— Résultats obtenus sur l'utilisateur `A1CV1WROP5KTTW` :

- Avec **50 films recommandés** et 5 voisins (neighbors) :
  Ici, on a fait varier le nombre de films de 5 à 50.

```
# la recommandation pour l'utilisateur A1CV1WROP5KTTW
recommendations = recommend_items(data, "A1CV1WROP5KTTW",
    n_neighbor=5, n_items=50)
recommandations
```

. Voici ce que nous avons obtenu comme résultat :

Les notes des films ont été triées par ordre décroissant afin de proposer les meilleures recommandations. Avec 50 films recommandés, les notes varient entre 3,82 et 5,00.

- Avec 50 films recommandés et **50 voisins (neighbors)**
  Ici, on a fait varier le nombre de voisins de 5 à 100.

```
# la recommandation pour l'utilisateur A1CV1WROP5KTTW
recommendations1 = recommend_items(data, "A1CV1WROP5KTTW"
   , n_neighbor=100, n_items=50)
recommandations1
```

.

— Comparaison des résultats :

Visuellement, on obtient le même résultat de recommandations avec 5 et 100 voisins dans ce cas. Nous allons vérifier l'égalité des résultats à l'aide du

	movieId	predicted_ratings
2	Movie3	5.000000
57	Movie59	5.000000
67	Movie69	5.000000
65	Movie67	5.000000
62	Movie64	5.000000
71	Movie73	5.000000
58	Movie60	5.000000
60	Movie62	5.000000
56	Movie58	5.000000
43	Movie45	5.000000
24	Movie26	5.000000
18	Movie20	5.000000
15	Movie17	5.000000
51	Movie53	5.000000
142	Movie144	5.000000
169	Movie171	5.000000
157	Movie159	5.000000
201	Movie203	5.000000
152	Movie154	5.000000
88	Movie90	5.000000
81	Movie83	5.000000
93	Movie95	4.823467
26	Movie28	4.823467
50	Movie52	4.686212
...		
125	Movie127	4.044824
139	Movie141	4.013943
119	Movie121	3.906800
200	Movie202	3.823467

FIGURE 1 – Recommandation de 50 films pour l'utilisateur A1CV1WROP5KTTW

	movieId	predicted_ratings
157	Movie159	5.000000
24	Movie26	5.000000
67	Movie69	5.000000
43	Movie45	5.000000
18	Movie20	5.000000
62	Movie64	5.000000
71	Movie73	5.000000
15	Movie17	5.000000
60	Movie62	5.000000
58	Movie60	5.000000
81	Movie83	5.000000
57	Movie59	5.000000
56	Movie58	5.000000
142	Movie144	5.000000
88	Movie90	5.000000
201	Movie203	5.000000
51	Movie53	5.000000
169	Movie171	5.000000
152	Movie154	5.000000
2	Movie3	5.000000
65	Movie67	5.000000
93	Movie95	4.823467
26	Movie28	4.823467
50	Movie52	4.686212
...		
125	Movie127	4.044824
139	Movie141	4.013943
119	Movie121	3.906800
199	Movie201	3.823467

FIGURE 2 – Recommandation avec 100 voisins pour l'utilisateur A1CV1WROP5KTTW

code suivant :

```
recommendations.equals(recommendations1)
✓ 0.0s
True
```

— **Interprétation des résultats :**

- Dans notre cas, les résultats obtenus avec 5 voisins et avec 100 voisins sont **identiques**. Cela peut s'expliquer par le fait que les voisins supplémentaires (du 6^{e} au 100^{e}) ont des profils très proches des 5 premiers, ou bien qu'ils ont eu un poids trop faible pour influencer les recommandations finales.
- Cependant, de manière générale, **augmenter le nombre de voisins peut modifier les recommandations**. En effet, lorsqu'on passe de 5 à 100 voisins, on intègre des utilisateurs aux profils peut-être moins similaires. Cela ajoute de la diversité aux préférences prises en compte et peut ainsi changer la moyenne pondérée des notes prévues pour les films non encore évalués par l'utilisateur.

h) Donner la définition de la matrice de similarité en collaborative filtering

L'article **"Collaborative Filtering Similarity Calculations"** présente plusieurs éléments essentiels liés à la construction des matrices de similarité, notamment :

- La **construction d'une matrice de similarité** entre les articles ;
- Le **choix des voisins les plus similaires** (méthode *Top-N* ou seuil) ;
- La **prédiction d'une note** à partir des évaluations d'autres articles similaires ;
- Certaines **limites du système**, telles que le *problème du démarrage à froid*.

Définition de la matrice de similarité

La **matrice de similarité** dans le filtrage collaboratif est une matrice carrée qui quantifie la similarité entre les éléments (ou utilisateurs) sur la base de leurs notes ou interactions. Chaque cellule représente un score de similarité entre deux éléments, calculé à l'aide de mesures telles que la similarité cosinus ajustée (basée sur des notes normalisées).

- Proche de 1 : forte similarité ;
- Proche de -1 : forte dissimilarité ;
- Proche de 0 : faible corrélation ou absence de lien significatif.

Cette matrice constitue la base du système de recommandation, en identifiant les éléments similaires à ceux déjà appréciés par l'utilisateur. Elle permet ainsi de prédire les préférences et de générer des suggestions personnalisées.

Dans le **filtrage collaboratif basé sur les items**, la **similarité** mesure à quel point deux articles (par exemple, deux films) sont perçus comme similaires, en fonction des évaluations fournies par les utilisateurs. L'idée sous-jacente est que si plusieurs utilisateurs évaluent deux articles de manière comparable, alors ces articles peuvent être considérés comme similaires.

Partie 2

a. Quel est le jeu de données utilisé dans ce notebook ? A-t-il été utilisé en totalité ?

Le jeu de données utilisé est **MovieLens** et seules **100 000 lignes** ont été utilisées sur un total de **25 millions de lignes** disponibles.

b. De quel type de données s'agit-il dans ce modèle d'apprentissage automatique ?

Il s'agit de **données d'interactions** entre utilisateurs et éléments (films), représentées sous forme de **notes** attribuées par les utilisateurs aux films. Ces données ont été importées à partir d'un fichier `.tsv` et contiennent 4 colonnes :

— une pour l'utilisateur,
— une pour le film,
— une pour la note, et
— une colonne `Timestamp` indiquant la date à laquelle la note a été attribuée.

c. Quel problème le filtrage collaboratif permet-il de résoudre ?

Le **filtrage collaboratif** permet de résoudre le problème de la **recommandation personnalisée**. Il permet de prédire la note qu'un utilisateur pourrait attribuer à un élément (comme un film ou un produit), en se basant sur les préférences d'autres utilisateurs similaires. Cette prédiction permet de recommander des articles alignés avec les goûts d'un individu.

d. Comment le filtrage collaboratif résout-il ce problème ?

Il fonctionne en analysant les **modèles d'interaction** entre utilisateurs et éléments. En identifiant des utilisateurs ayant des goûts similaires, le système peut recommander des éléments qu'un utilisateur n'a pas encore vus, mais qui ont été appréciés par d'autres utilisateurs similaires.

En pratique, on construit une **matrice** avec les utilisateurs en lignes et les articles en colonnes. Chaque cellule (i, j) de cette matrice contient la note donnée par l'utilisateur i à l'article j. Comme cette matrice contient de nombreuses cellules vides (notes non attribuées), l'objectif est de les **prédire**.

Pour cela, on suppose que :

— chaque utilisateur i est représenté par un vecteur de préférences $U_i \in \mathbb{R}^k$,
— chaque article j est représenté par un vecteur de caractéristiques $V_j \in \mathbb{R}^k$,

où k est le nombre de **facteurs latents**.

Initialisation : Les vecteurs U_i et V_j sont initialisés aléatoirement dans l'intervalle $[-1, 1]$:

$$U_i = [u_{i1}, u_{i2}, \ldots, u_{ik}]^\top \in [-1, 1]^k, \quad V_j = [v_{j1}, v_{j2}, \ldots, v_{jk}]^\top \in [-1, 1]^k$$

Prédiction d'une note : La note prédite de l'utilisateur i pour l'article j est donnée par le calcul de ce produit scalaire qui mesure le degré d'adéquation entre les préférences de l'utilisateur et les caractéristiques de l'article :

$$\hat{R}_{ij} = U_i^\top V_j = \sum_{l=1}^{k} u_{il} \cdot v_{jl}$$

Fonction de coût à minimiser :

$$\mathcal{L}(U, V) = \sum_{(i,j)\in\mathcal{K}} \left(R_{ij} - \hat{R}_{ij}\right)^2 + \lambda \left(\|U_i\|^2 + \|V_j\|^2\right)$$

où :

— $\mathcal{K}$ est l'ensemble des couples (i, j) pour lesquels la note réelle R_{ij} est connue,
— λ est un paramètre de régularisation visant à éviter le surapprentissage.

Optimisation : On ajuste les vecteurs U_i et V_j à l'aide de la **descente de gradient** de manière à **minimiser l'erreur** entre la note prédite et la note réelle :

— calcul de l'erreur entre la note réelle et la note prédite,
— mise à jour des vecteurs U_i et V_j pour réduire cette erreur,
— répétition du processus jusqu'à convergence.

e. Pourquoi un modèle de filtrage collaboratif peut-il échouer à fournir un système de recommandation efficace ?

Un modèle de filtrage collaboratif peut échouer dans les cas suivants :

— En cas de **pénurie de données** (nouveaux utilisateurs ou nouveaux articles), ce qui empêche la détection de similarités pertinentes ;
— En présence de **biais de popularité** : certains articles populaires sont sur-recommandés ;
— À cause des **boucles de rétroaction** : les recommandations influencent les comportements futurs, renforçant les biais du système.

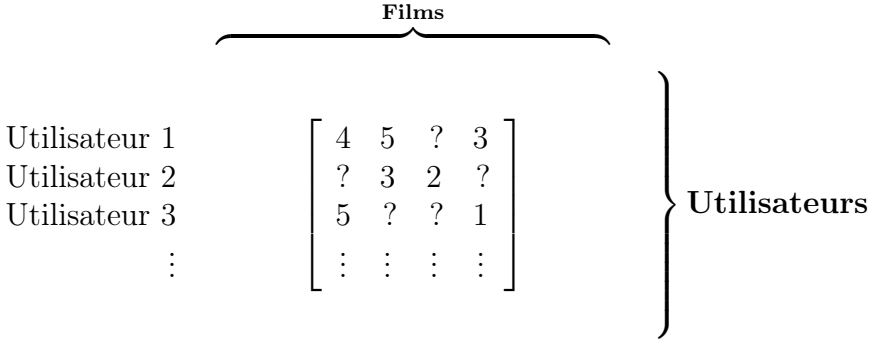
f. À quoi ressemble une représentation en tableau croisé des données de filtrage collaboratif ?

Une représentation en **tableau croisé** (crosstab) des données de filtrage collaboratif prend la forme d'une **matrice** de notes $R \in \mathbb{R}^{m\times n}$, où :

— Les **m lignes** représentent les utilisateurs ;
— Les **n colonnes** représentent les articles (ex. : films) ;
— Les **cellules** R_{ij} contiennent les notes attribuées.

Cette matrice est partiellement vide : ce sont précisément ces valeurs manquantes que l'on cherche à **prédire**.

Ci-dessous un exemple de représentation d'un tableau croisé :



g. Qu'est-ce qu'un facteur latent ? Pourquoi est-il "latent" ?

Un **facteur latent** est une caractéristique **cachée** qui influence les préférences des utilisateurs ou les attributs des éléments. Il est dit **latent** car il n'est pas directement observable mais **inféré à partir des données**.

Par exemple, un facteur latent pourrait représenter le degré auquel un film est considéré comme un film d'action, même si ce genre n'est pas explicitement indiqué. Ces facteurs sont essentiels pour modéliser les préférences implicites et pour effectuer des recommandations pertinentes.

Partie 3

a - Lister le plan suivi dans ce notebook

Voici le plan détaillé suivi dans le notebook, avec une description approfondie de chaque étape :

1. **Nettoyage des données**
   Dans cette phase, l'auteur a effectué plusieurs opérations pour préparer les données :
   (a) **Filtrage des colonnes** : seules certaines colonnes spécifiques ont été conservées, notamment les colonnes démographiques (`demographic_cols`) et les colonnes de produits (`product_cols`).
   (b) **Conversion des types de données** : les colonnes ont été converties dans des types appropriés pour permettre un traitement correct par les algorithmes.
   (c) **Traitement des valeurs manquantes** : les valeurs manquantes ont été traitées par des méthodes d'imputation adaptées au type de variable.

2. **Ingénierie des caractéristiques (Feature Engineering)**
   Cette étape vise à transformer les variables brutes en informations exploitables par le modèle. Les principales opérations incluent :
   — Conversion de certaines colonnes en format `datetime`.
   — Changement des variables continues comme l'âge, l'ancienneté, le revenu et la date d'entrée en catégories.
   — Transformation logarithmique du revenu pour normaliser sa distribution.
   — Nettoyage des valeurs incohérentes (ex. : ancienneté négative).
   Ces transformations permettent une meilleure prise en compte des similarités entre les utilisateurs dans le modèle.

3. **Sous-ensemble de données (Data Subsetting)**
   Cette étape consiste à filtrer le jeu de données pour ne conserver que les observations pertinentes et alléger la mémoire. Les opérations effectuées sont :
   — Sélection de dates spécifiques (mai 2015, juin 2015 et mai 2016) pour focaliser l'analyse sur des périodes clés.
   — Création de variables représentant la possession de produits le mois précédent afin de calculer les nouveaux achats.
   — Fusion du jeu de données avec un décalage temporel pour chaque utilisateur, permettant d'identifier les produits nouvellement acquis.

— Remplacement des valeurs négatives (indiquant des suppressions de produits) par zéro, car elles ne sont pas considérées comme des achats.
— Ajout de l'historique d'achat dans les données de test pour permettre la prédiction.

Cette étape optimise les performances en réduisant la taille des données tout en conservant l'information essentielle pour la modélisation.

4. **Suppression des doublons**

Afin d'optimiser les performances du système de recommandation basé sur la similarité utilisateur, nous avons procédé à une suppression des enregistrements dupliqués dans les données de test. Cette étape permet de réduire le temps de calcul en évitant les redondances lors de la comparaison entre utilisateurs.

— **Historique d'achat** : suppression des doublons dans les colonnes de produits récents (`new_product_col`) pour ne conserver que les combinaisons uniques d'achats.
— **Données démographiques** : transformation des variables catégorielles (`sexo`, `age`, `fecha_alta`, etc.) en format binaire, suivie d'une suppression des doublons afin d'obtenir des profils démographiques distincts.

Résultats :
— 6510 combinaisons uniques d'historique d'achat.
— 114 profils démographiques distincts.

Cette étape est cruciale pour construire efficacement les matrices de similarité dans les approches *memory-based* et *démographiques*.

5. **Construction des matrices de similarité**

Deux types de matrices de similarité ont été construites pour évaluer la probabilité d'achat de chaque produit par les clients :

— **Démographique** : basée sur des attributs clients tels que l'âge, le sexe, la date d'inscription, etc. Pour chaque profil test unique, les distances ont été calculées entre les profils de test et d'entraînement à l'aide de la distance euclidienne, les données étant continues ou catégorielles transformées.
— **Basée sur la mémoire (Memory-Based)** : construite à partir de l'historique d'achats des clients. La distance de Manhattan a été utilisée pour les données binaires. Chaque utilisateur test est comparé aux clients d'entraînement, et les distances inversées pondèrent les historiques d'achats afin d'estimer les probabilités d'achat pour chaque produit.

Méthodologie :
— Calcul des distances entre les profils de test et d'entraînement selon le type de données.
— Pondération des historiques d'achat à l'aide de l'inverse des distances.
— Suppression des probabilités d'achat pour les produits déjà possédés.
— Fusion finale avec les données d'origine pour obtenir une probabilité d'achat par produit et par utilisateur.

6. **Combinaison des probabilités issues des matrices**

Les probabilités d'achat issues des matrices de similarité démographique et comportementale sont combinées à l'aide d'une moyenne pondérée. Cinq combinaisons ont

été testées avec des poids différents (de 0 à 1) afin de trouver le meilleur équilibre entre les deux sources d'information :

— $P_{1.0} = 1.0 \times P_{\text{comportementale}} + 0.0 \times P_{\text{démographique}}$
— $P_{0.9} = 0.9 \times P_{\text{comportementale}} + 0.1 \times P_{\text{démographique}}$
— $P_{0.7} = 0.7 \times P_{\text{comportementale}} + 0.3 \times P_{\text{démographique}}$
— $P_{0.5} = 0.5 \times P_{\text{comportementale}} + 0.5 \times P_{\text{démographique}}$
— $P_{0.0} = 0.0 \times P_{\text{comportementale}} + 1.0 \times P_{\text{démographique}}$

Suppression des doublons :
Pour éviter de recommander des produits que l'utilisateur possède déjà, une étape de « nullification » est appliquée. Elle consiste à croiser les probabilités prédites avec l'historique réel des achats et à annuler les probabilités des produits déjà achetés.

7. **Génération des recommandations**
À partir des probabilités prédites pour chaque produit et chaque client, une liste des 7 produits les plus susceptibles d'être ajoutés est générée par client.

Les probabilités sont d'abord traitées pour ne garder que les valeurs uniques (afin d'optimiser la vitesse d'exécution). Ensuite, pour chaque vecteur de probabilité, les produits sont triés par ordre décroissant de probabilité, et les 7 premiers sont retenus. Ces produits recommandés sont ensuite concaténés pour chaque client, formant ainsi les recommandations finales.

Cette étape produit différents ensembles de recommandations correspondant à différentes stratégies de mélange de probabilités (ex. : 100%, 90%, etc.), qui seront évaluées par la suite.

8. **Réexécution du modèle avec tous les jeux de données pour le paramètre de mélange optimal**
L'analyse des résultats obtenus lors de l'évaluation a montré que le meilleur compromis entre les données d'historique d'achats (memory-based) et les données démographiques se situait autour de 85%. En conséquence, l'ensemble du processus est relancé en utilisant cette valeur optimale de mélange et en s'appuyant cette fois sur l'intégralité des données d'entraînement disponibles.

Les probabilités sont recalculées pour les deux sources d'information (achats et démographie), puis une moyenne pondérée est appliquée (85% memory, 15% démographie). Après cela, les produits déjà détenus par chaque client sont supprimés des probabilités, et les produits les plus probables (top 7) sont sélectionnés comme recommandations.

Partie la plus intéressante : La stratégie de combinaison pondérée (85% données d'historique d'achats et 15% données démographiques) est particulièrement pertinente. Elle montre une démarche d'optimisation fine et pragmatique qui améliore légèrement mais significativement les performances, sans recourir à des modèles complexes. Le fait d'obtenir un score compétitif uniquement avec une approche de filtrage collaboratif basé sur la mémoire est un résultat fort.

Partie manquante ou peu claire : La méthode exacte utilisée pour déterminer que 85% est la meilleure valeur du paramètre de mélange n'est pas détaillée. Une description ou un graphique montrant la performance en fonction du taux de

mélange (par exemple 70%, 80%, 85 %, 90%, etc.) aurait permis de mieux comprendre cette décision. De plus, la fonction purchase_nullifier n'est pas expliquée, son rôle est deviné, mais une clarification sur son mécanisme exact serait utile.

b - Comment il a traité les valeurs manquantes ?

Le traitement des valeurs manquantes dépend du type de variable :

— **Variables qualitatives** :
  - Imputation par la modalité la plus fréquente.
  - Création d'une catégorie si nécessaire.

— **Variables numériques** :
  - Imputation par la moyenne de la province.

— **Variables liées aux produits** :
  - Les valeurs manquantes sont remplacées par 0 (non possession).

c - Qu'est qu'il utilise comme variables explicatives en plus de ce qu'il appelle les `demographic_cols` ?

En plus des `demographic_cols`, le modèle utilise les variables explicatives suivantes :

- **L'historique des produits bancaires** : il s'agit du statut des produits pour les mois précédents. Par exemple, les variables telles que `lag_1_ind_ahor_fin_ult1`, `lag_1_ind_cco_fin_ult1`, etc., indiquent si un client possédait un certain produit au mois précédent.

- **Des variables dérivées** : ces variables résument ou comparent les produits détenus dans le temps. Elles incluent par exemple :
  — `num_products` : le nombre total de produits détenus par le client,
  — `delta_products` : la variation du nombre de produits entre deux mois consécutifs,
  — `new_products` : nombre de nouveaux produits acquis ce mois-ci,
  — `removed_products` : nombre de produits supprimés ce mois-ci.

  Ces variables permettent d'enrichir les données et d'améliorer les performances du modèle prédictif.

d - Dans cette ligne Python, est-ce que le paramètre `suffixes=['', '_previous']` est nécessaire ou non ?

On a la commande suivante :

```
testdat_use = pd.merge(testdat, traindat[traindat.fecha_dato
    =='2016-05-28'][test_col],
on='ncodpers', how='left', suffixes=['', '_previous'])
```

Le paramètre `suffixes=['', '_previous']` est **nécessaire** pour éviter les conflits de noms entre les colonnes communes aux deux DataFrames.

Expliquez pourquoi ?

Il permet de distinguer clairement les colonnes issues des deux sources :
— les colonnes de `testdat` conservent leur nom d'origine, tandis que
— les colonnes de `traindat` reçoivent le suffixe `_previous`
Cela est particulièrement utile lorsque deux DataFrames partagent les mêmes noms de colonnes.

Ce mécanisme facilite l'analyse comparée entre la situation actuelle et la situation antérieure.

Sans ce paramètre, Pandas ajouterait par défaut des suffixes génériques comme `_x` et `_y`, qui sont moins explicites et peuvent compliquer la lecture des résultats.

e - Que fait l'instruction python `gc.collect()` surtout après un delete d'objets par exemple de type dataframe ?

La fonction `gc.collect()` force le ramasse-miettes à libérer la mémoire occupée par les objets supprimés, comme les DataFrames.

f - Est-ce qu'il utilise toutes les variables `demographic_cols` comme variables explicatives ou une partie de ces variables ?

Non, il n'utilise qu'une **partie** des variables contenues dans `demographic_cols` comme variables explicatives.

Lesquelles ?

Les variables démographiques effectivement utilisées comme variables explicatives sont celles définies dans `demog_col` :

```
demog_col = ['sexo', 'age', 'fecha_alta', 'ind_nuevo', '
    indrel', 'indresi', 'indfall', 'tipodom', 'ind_actividad_
    cliente']
```

Ainsi, plusieurs variables de `demographic_cols`, telles que `ind_empleado`, `antiguedad`, `pais_residencia`, `canal_entrada`, `renta`, etc., **ne sont pas utilisées** comme variables explicatives dans le modèle. L'auteur a donc conservé uniquement certaines variables pertinentes pour construire la matrice de similarité.

g - Que fait cette expression Python ?

```
testdat_final.drop('ncodpers', axis=1).drop_duplicates().
    copy().reset_index(drop=True)
```

Cette commande effectue une série d'opérations sur le DataFrame `testdat_final` :

- `drop('ncodpers', axis=1)` : supprime la colonne `ncodpers` (colonne d'identifiant client), car elle n'est pas utile pour l'analyse des similarités.
- `drop_duplicates()` : supprime les lignes dupliquées, ne conservant que les lignes uniques.
- `copy()` : crée une copie indépendante de l'objet résultant.
- `reset_index(drop=True)` : réinitialise les index du DataFrame. Le paramètre `drop=True` évite que l'ancien index soit ajouté en tant que colonne.

Ainsi, cette expression produit un nouveau DataFrame sans identifiant client, sans doublons, avec des index réinitialisés et prêt à être utilisé pour des analyses.

h - C'est quoi `testdat_final_unique` ?

`testdat_final_unique` : construit à partir de `testdat_final`, qui contient uniquement les colonnes des nouveaux produits (`new_product_col_aug`). On y supprime la colonne `ncodpers`, les doublons, puis on réinitialise les index.

```
testdat_final_unique = testdat_final.drop('ncodpers', 1).
    drop_duplicates().copy().reset_index(drop=True)
```

- C'est quoi `testdat_demog_final_unique` ?

`testdat_demog_final_unique` : obtenu à partir de `testdat_demog_final`, qui contient les variables démographiques et `ncodpers`. La même opération est appliquée pour retirer les doublons et l'identifiant.

```
testdat_demog_final_unique = testdat_demog_final.drop('
    ncodpers', 1).drop_duplicates().copy().reset_index(drop=
    True)
```

→ Ces DataFrames sont utilisés pour supprimer les redondances et faciliter les calculs dans les étapes suivantes du modèle.

i - En step 4, pourquoi l'auteur a-t-il éclaté le `traindat_final` en 2 dataframes : `traindat_train` et `traindat_test` ?

Selon le code suivant :

```
traindat_index = np.random.rand(len(traindat_final)) < 0.8
# create traindat_train
```

```
traindat_train = traindat_final[traindat_index]
# create traindat_test
traindat_test = traindat_final[~traindat_index]
```

La séparation permet de diviser aléatoirement les données en deux ensembles : un pour l'entraînement (`traindat_train`) et un pour le test (`traindat_test`). Cela permet d'évaluer la performance du modèle sur des données non vues pendant l'apprentissage, de détecter un éventuel surapprentissage (overfitting) et de suivre sa capacité de généralisation.

j - Quelle est la distance choisie pour trouver les lignes du training set les plus proches de la ligne à prédire ?

La **distance de Manhattan** est utilisée pour mesurer la similarité.

Définition

La **distance de Manhattan** mesure la distance entre deux points dans un espace en sommant la valeur absolue des différences de leurs coordonnées.
Pour deux vecteurs $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ et $\mathbf{y} = (y_1, y_2, \ldots, y_n)$, la distance de Manhattan est donnée par :

$$D_{\text{Manhattan}}(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} |x_i - y_i|$$

Exemple :
Entre $\mathbf{x} = (1, 2, 3)$ et $\mathbf{y} = (2, 0, 4)$:

$$D = |1 - 2| + |2 - 0| + |3 - 4| = 1 + 2 + 1 = 4$$

- Pourquoi l'auteur a-t-il utilisé cette distance ?

Cette distance est particulièrement adaptée lorsque les données sont **binaires** ou **parcimonieuses (sparse)**, comme dans le cas des historiques d'achat (présence ou absence de produits).

k - Dans la fonction `probability_calculation()`, peut-on remplacer l'expression `row_use = row.to_frame().T` par `row_use = row.values.reshape(1,-1)` ?

Oui, on peut remplacer :

```
row.to_frame().T
```

par :

```
row.values.reshape(1, -1)
```

à condition que les colonnes soient déjà bien alignées avec celles de `training` et que l'on n'ait pas besoin des noms de colonnes à cette étape. Cela peut légèrement améliorer les performances, car `reshape` renvoie un tableau NumPy plus léger que la création d'un DataFrame.

Attention : Si `row_use` est utilisé plus tard avec des noms de colonnes (ex. `row_use[used_columns]`), cette substitution ne fonctionnera pas sans modification du reste du code dans la fonction.

l - Que représente `traindat_purchases_train` dans :

```
probabilities_memory = probability_calculation(traindat_test
    _unique, traindat_train, traindat_purchases_train, new_
    product_col, 'manhattan', traindat_test)
```

Le dataframe `traindat_purchases_train` représente les données d'achat des clients de l'ensemble d'entraînement (`traindat_train`), utilisées pour calculer les probabilités d'achat pour les clients de l'ensemble de test en fonction de leur similarité. C'est une dataframe indiquant les achats par produit pour chaque client d'entraînement.

m - En step 6, quelles sont les 5 combinaisons de pondération (de poids) proposées par l'auteur ?

L'auteur propose cinq combinaisons de pondération entre la similarité mémoire (basée sur les produits) et la similarité démographique (basée sur les utilisateurs) afin d'évaluer l'impact de chacune sur les prédictions. Ces combinaisons sont :

- 100 % mémoire / 0 % démographie
- 90 % mémoire / 10 % démographie
- 70 % mémoire / 30 % démographie
- 50 % mémoire / 50 % démographie
- 0 % mémoire / 100 % démographie

Cela permet de tester différents équilibres entre les deux approches pour optimiser la qualité des recommandations.

n - En step 7, que veut dire `arank.values[:,::-1]` dans cette fonction ?

```
def probabilities_to_predictions(probabilities
ncodpers, print_option=False):
...
```

La ligne `arank.values[:, ::-1][:, :7]` sert à extraire les indices des 7 produits les plus probables pour chaque utilisateur.

L'expression `[:, ::-1]` inverse l'ordre des colonnes (tri décroissant), permettant de sélectionner directement les 7 produits les plus recommandés à partir des probabilités. Cela permet de générer la prédiction finale sous forme des 7 recommandations les plus pertinentes pour chaque client.

NB

Le code source de cette partie, incluant les modifications apportées, est fourni en pièce jointe sous le nom `"Code_Part3.ipynb"`.

Nous nous sommes appuyés sur le code du notebook proposé dans le cadre de la compétition Kaggle, disponible à l'adresse suivante (comme demandé) :

https://www.kaggle.com/code/jake126/memory-based-collaborative-filtering.