

Table des matières de l'annexe 1

1. Codes utilisés.....	1
1.1. Extraction des données.....	1
1.1.1. Données générales	1
1.1.2. Données pour le content based	3
1.1.3. Films_Vus.....	4
1.1.4. Nombres_rec.....	5
1.2. Validation croisée.....	5
1.3. Listes_Importance.....	6
1.3.1. En fonction de la fréquence de notation	6
1.3.2. En fonction de la date de la dernière évaluation.....	8
1.3.3. En fonction du nombre d'évaluation	10
1.4. Modèle de système de recommandation	11
1.4.1. User_Based model	11
1.4.2. Item_Based model	13
1.4.3. Content_Based model.....	16
1.5. Evaluation_Précision_poids	17
1.5.1. Recall	17
1.5.2. RecallPoids	17
1.5.3. Précision	18
1.5.4. PrécisionPoids	19
1.5.5. nDCG.....	20
1.5.6. nDCGPoids	21
1.6. Evaluation_poids.....	22
1.6.1. DistanceUtilisateur	22
1.6.2. DistanceUtilisateurPoids	23
1.6.3. ILD.....	24
1.6.4. ILDPoids.....	25

1. Codes utilisés

1.1. Extraction des données

1.1.1. Données générales

Ce code va venir extraire les informations concernant la notation de films par des utilisateurs du CSV '**data/ratings.csv**'. Notons que nous n'extrayons que les films ayant reçu plus de 10 notations.

Trois structures de données sont créées :

- **dico_user** : un dictionnaire qui va stocker les films notés par chaque utilisateur. Les clés sont les identifiants des utilisateurs, et les valeurs sont des listes contenant les identifiants des films notés et les notations correspondantes.
- **movie** : une liste qui va stocker tous les films ayant plus de 10 notations.
- **dico_movie** : un dictionnaire qui va reprendre le nombre de notations par film. Les clés sont les identifiants des films, et les valeurs sont les nombres de notations correspondants.

Les matrices **matrice_adjacente** et **matrice_notation** sont créées en utilisant la classe **scipy.sparse.csr_matrix**. Ces matrices seront de dimensions correspondant au nombre d'utilisateurs et au nombre de films.

Les matrices **matrice_adjacente** et **matrice_notation** sont conservées dans des fichiers JSON.

```
# dictionnaire reprenant les films notés par utilisateur
dico_user = {}

# liste reprenant tous les films (attention: ayant plus de 10 ratings)
movie = []

# dictionnaire reprenant le nombre de notations par film
dico_movie = {}

# ouverture du fichier
with open('data/ratings.csv') as csvfile:
    spamreader = csv.reader(csvfile, delimiter=',')
    # lecture du fichier ligne par ligne
    for row in spamreader:
        # si pas les headers
        if spamreader.line_num != 1:
            # ajout du film au dictionnaire de films
```

```

        if int(row[1]) not in dico_movie:
            dico_movie[int(row[1])] = 1
        else:
            # si déjà dedans, ajout d'une notation au compteur
            count = dico_movie.get(int(row[1])) + 1
            dico_movie[int(row[1])] =
count

# affichage du nombre de films notés
print(len(dico_movie))

# réouverture du fichier pour l'extraction des données
with open('data/ratings.csv') as csvfile:
    spamreader = csv.reader(csvfile, delimiter=',')
    # lecture ligne par ligne
    for row in spamreader:
        # si pas les headers et films obtenant plus de 10 notations
        if spamreader.line_num != 1 and dico_movie.get(int(row[1])) >=10:
            # ajout de l'utilisateur au dictionnaire
            if int(row[0]) not in dico_user:
                dico_user[int(row[0])]=[[int(row[1])],[float(row[2])]]
            else:
                # si déjà dedans, ajout du film à sa liste avec sa
notation
                dico_user.get(int(row[0]))[0].append(int(row[1]))
                dico_user.get(int(row[0]))[1].append(float(row[2]))
            # ajout du film à la liste de films
            if int(row[1]) not in movie:
                movie.append(int(row[1]))

# trie de la liste de films
movie.sort()
print('Nombre users:', len(dico_user), 'Nombre films ayant plus de 10
ratings:', len(movie))

# matrice sparse contenant servant de matrice binaire
matrice_adjacente=scipy.sparse.csr_matrix((len(dico_user),len(movie)))

# matrice reprenant les notations
matrice_notation=scipy.sparse.csr_matrix((len(dico_user),len(movie)))

# pour chaque utilisateur
list_user=list(dico_user.keys())
for i in dico_user:
    # pour chaque film visionné par l'utilisateur i
    for j in range(len(dico_user.get(i)[0])):
        # initialisation des bonnes valeurs dans les matrices

```

```

        matrice_adjacente[list_user.index(i),movie.index(dico_user.get(i)[0][j
])]]=1
        matrice_notation[list_user.index(i),movie.index(dico_user.get(i)[0][j]
)]=dico_user.get(i)[1][j]

#Vérification
nombre_lignes, nombre_colonnes = matrice_adjacente.shape
print(nombre_lignes, nombre_colonnes)

# conservation dans un fichier json
with open("Matrice notations.json", "w") as f:
    json.dump(list(matrice_notation.toarray().tolist()),f)
with open("Matrice adjacente.json", "w") as f:
    json.dump(list(matrice_adjacente.toarray().tolist()),f)

#le premier film de la liste sera à l'indice 0 de la matrice, de même pour le
premier user'''

```

1.1.2. Données pour le content based

Ce code analyse le fichier "movies.csv", extrait les genres et les années des films, et crée des matrices creuses pour représenter ces informations. Ces matrices sont ensuite sauvegardées dans des fichiers JSON. Ces matrices nous seront utiles pour notre modèle de système de recommandation basé sur le contenu. Encore une fois, nous ne travaillons qu'avec les films ayant plus de 10 notations.

```

import csv
import scipy.sparse
import json
import statistics
import datetime
import re

# Listes des genres et des années
liste_genres=["Adventure", "Animation", "Children", "Comedy", "Fantasy",
"Drama", "Crime", "Thriller",
            "Romance", "Horror", "Sci-Fi", "IMAX", "Mystery", "Musical",
"Action", "Documentary", "War",
            "Film-Noir", "Western"]

liste_annee=list(range(1902, 2019))

# Liste combinée des genres et des années
total_liste=liste_genres+liste_annee

# Ouverture du fichier movies.csv
with open('data/tiny/content/movies.csv') as csvfile:

```

```

spamreader = csv.reader(csvfile, delimiter=',')
rows=list(spamreader)

# Initialisation des matrices creuses pour les genres et les années
matrice_genres_annees = scipy.sparse.lil_matrix((len(movie),
len(total_liste)))
matrice_genres=scipy.sparse.lil_matrix((len(movie),len(liste_genres)))
matrice_annee=scipy.sparse.lil_matrix((len(movie),len(liste_annee)))

# Lecture du fichier ligne par ligne
j=0
for i, row in enumerate(rows[1:]):
    # Vérification si le film est présent dans le dictionnaire et a au
    moins 10 notations
    if dico_movie.get(int(row[0])) and dico_movie[(int(row[0]))] >=10:
        # Extraction de l'année à partir du titre du film
        extract_year = re.search(r"\\((\\d{4})\\)", row[1])
        extract_genres = row[2].split("|")

        if extract_year:
            annee = int(extract_year.group(1))
            # Mise à jour de la matrice des genres et des années
            matrice_genres_annees[j, total_liste.index(annee)] = 1
            matrice_annee[j, liste_annee.index(annee)] = 1

            for genre in extract_genres:
                if genre in liste_genres:
                    # Mise à jour de la matrice des genres
                    matrice_genres_annees[j, total_liste.index(genre)] = 1
                    matrice_genres[j, liste_genres.index(genre)] = 1
                else:
                    pass
            j=j+1

# Sauvegarde des matrices dans des fichiers JSON
with open("matrice_genres_annees.json", "w") as f:
    json.dump(list(matrice_genres_annees.toarray().tolist()),f)

with open("matrice_genres.json", "w") as f:
    json.dump(list(matrice_genres.toarray().tolist()),f)

with open("matrice_annee.json", "w") as f:
    json.dump(list(matrice_annee.toarray().tolist()),f)

```

1.1.3. Films_Vus

La fonction **Films_Vus** prend deux arguments en entrée : **mat** qui représente une matrice adjacente, et **user** qui est l'indice de l'utilisateur dont on souhaite obtenir la liste des films vus.

La fonction renvoie une liste contenant les indices de tous les films qui ont été vus par cet utilisateur.

```
def Films_Vus(mat, user):
    # renvoie une liste avec l'indice de tous les films vus par l'utilisateur
    user
    # liste à renvoyer
    liste=[]
    for i in range(mat.get_shape()[1]):
        # matrice adjacente égale à un si le film i a été vu par l'utilisateur
        user
        if mat[user,i]==1:
            # remplit la liste avec les films vus
            liste.append(i)

    return liste
```

1.1.4. Nombres_rec

Ces deux fonctions permettent de créer des listes de rangs (rank) contenant les 10 premiers éléments de chaque rang pour la fonction **Liste10**, et les 20 premiers éléments de chaque rang pour la fonction **Liste20**. Les rangs sont passés en paramètre sous forme de liste.

```
import scipy.sparse
import json

# Fonction qui renvoie une liste de rangs contenant les 10 premiers éléments
de chaque rang
def Liste10(rank):
    new_rank = []
    for i in range(len(rank)):
        new_rank.append(rank[i][:10])
    return new_rank

# Fonction qui renvoie une liste de rangs contenant les 20 premiers éléments
de chaque rang
def Liste20(rank):
    new_rank = []
    for i in range(len(rank)):
        new_rank.append(rank[i][:20])
    return new_rank
```

1.2. Validation croisée

Ce code génère 10 matrices de test en distribuant aléatoirement les films à partir de la matrice binaire. Ces matrices de test seront utilisées pour la validation croisée dans le cadre de nos modèles de recommandation. Nos différentes matrices de test sont ensuite sauvegardé dans un

fichier JSON afin d'être utilisé par la suite.

```
import json
import scipy.sparse
import random

# ouverture du fichier reprenant la matrice binaire
with open("Matrice adjacente.json", 'r') as f:
    # récupération de la matrice
    matrice_binaire=json.load(f)
    # création d'un dictionnaire contenant 10 matrice de test
    dictionnaire_de_mat={1:[0 for i in range(len(matrice_binaire[0]))] for j
in range(len(matrice_binaire))},
                        2:[0 for i in range(len(matrice_binaire[0]))] for j
in range(len(matrice_binaire))},
                        3:[0 for i in range(len(matrice_binaire[0]))] for j
in range(len(matrice_binaire))},
                        4:[0 for i in range(len(matrice_binaire[0]))] for j
in range(len(matrice_binaire))},
                        5:[0 for i in range(len(matrice_binaire[0]))] for j
in range(len(matrice_binaire))},
                        6:[0 for i in range(len(matrice_binaire[0]))] for j
in range(len(matrice_binaire))},
                        7:[0 for i in range(len(matrice_binaire[0]))] for j
in range(len(matrice_binaire))},
                        8:[0 for i in range(len(matrice_binaire[0]))] for j
in range(len(matrice_binaire))},
                        9:[0 for i in range(len(matrice_binaire[0]))] for j
in range(len(matrice_binaire))},
                        10:[0 for i in range(len(matrice_binaire[0]))] for j
in range(len(matrice_binaire))}]
    # pour chaque notation de la matrice binaire
    for i in range(len(matrice_binaire)):
        for j in range(len(matrice_binaire[i])):
            if matrice_binaire[i][j]==1:
                # distribuer dans une matrice test de facon aléatoire
                ind=random.randint(1,10)
                dictionnaire_de_mat.get(ind)[i][j]=1
# conservation dans un fichier json
file=open("Cross-validation.json", 'w')
json.dump(dictionnaire_de_mat,file)
```

1.3. Listes_Importance

1.3.1. En fonction de la fréquence de notation

Ce code permet de déterminer l'importance de chaque utilisateur en fonction de la fréquence de ses notations. Il va rechercher le timestamp de la première et de la dernière évaluation ainsi que le nombre de notation de l'utilisateur, puis calcul la fréquence de notation grâce à ces

informations. Celle-ci est alors stockée dans une liste d'importance en format JSON afin d'être utilisée lors de l'évaluation de nos modèles.

```
# Importation des modules nécessaires
import csv
import scipy.sparse
import json
import statistics
import datetime

# Dictionnaire pour stocker les timestamps des notations par utilisateur
dico_timestamp = {}

# Dictionnaire pour stocker le nombre total de notations par utilisateur
dico_nb_note = {}

# Ouverture du fichier "ratings.csv"
with open('data/tiny/evidence/ratings.csv') as csvfile:
    spamreader = csv.reader(csvfile, delimiter=',')
    # Lecture du fichier ligne par ligne
    for row in spamreader:
        # Si ce n'est pas la première ligne (les en-têtes)
        if spamreader.line_num != 1:
            # Si nouvel utilisateur on ajout le timestamp au dictionnaire
            dico_timestamp
            if int(row[0]) not in dico_timestamp:
                dico_timestamp[int(row[0])] = [int(row[3])]
            else:
                # Si déjà présent, ajout ddu timestamp à la liste
                dico_timestamp[int(row[0])].append(int(row[3]))

            # Ajout du nombre de notation au dictionnaire dico_nb_note
            if int(row[0]) not in dico_nb_note:
                dico_nb_note[int(row[0])] = 1
            else:
                # Si déjà présent, augmentation du nombre de notations
                count = dico_nb_note.get(int(row[0])) + 1
                dico_nb_note[int(row[0])] = count

# Création de la liste des utilisateurs
list_user = list(dico_timestamp.keys())

# Création de la liste de fréquence des notations par utilisateur
list_frequence = [i * 0 for i in range(len(list_user))]

# Calcul de la fréquence des notations pour chaque utilisateur
#pour chaque utilisateur
```

```

for cle, liste in dico_timestamp.items():
    #prends le timestamp maximum et minimum pour l'utilisateur
    maximum = max(liste)
    minimum = min(liste)
    #transformation des timestamps en date
    datemax = datetime.datetime.fromtimestamp(maximum)
    datemin = datetime.datetime.fromtimestamp(minimum)
    #calcul de la période entre la première notation et la dernière
    periode = datemax - datemin
    #on va chercher le nombre de notation pour cet utilisateur
    nb_notation = dico_nb_note.get(cle)

    #Calcul la fréquence en divisant le nombre de notation par la période
    if periode.days == 0:
        freq = nb_notation / 1
    else:
        freq = nb_notation / periode.days

    #Ajout de la valeur trouvée à la liste des fréquences
    list_frequence[list_user.index(cle)] = freq

# Affichage de quelques statistiques sur la liste de fréquence
print('longueur liste: ', len(list_frequence))
print('max liste: ', max(list_frequence))
print('min liste: ', min(list_frequence))
print('moyenne liste: ', statistics.mean(list_frequence))

# Création de la liste d'importance basée sur la fréquence des notations
importance_frequence = [i * 0 for i in range(len(list_user))]
for i in range(len(list_user)):
    val = list_frequence[i]
    importance_frequence[i] = val

# Sauvegarde de la liste d'importance dans un fichier JSON
with open("Importance_frequence.json", "w") as f:
    json.dump(importance_frequence, f)

```

1.3.2. En fonction de la date de la dernière évaluation

Ce code permet de déterminer l'importance de chaque utilisateur en fonction de la dernière évaluation fournie par l'utilisateur. Il va extraire les timestamps des évaluations de l'utilisateur et stocker la plus ancienne. Ensuite, un score d'importance lui sera attribuer en fonction de la catégorie dans lequel il se trouve (5 catégories). La liste d'importance sera sauvegardée en format JSON afin d'être utilisée lors de l'évaluation de nos modèles.

```

import csv
import scipy.sparse
import json
import statistics
import datetime

dico_timestamp = {}

# Ouverture du fichier CSV contenant les ratings
with open('data/tiny/evidence/ratings.csv') as csvfile:
    spamreader = csv.reader(csvfile, delimiter=',')

    # Lecture du fichier ligne par ligne
    for row in spamreader:
        # Si ce n'est pas la première ligne (les headers)
        if spamreader.line_num != 1:
            # Ajout du film au dictionnaire de films
            if int(row[0]) not in dico_timestamp:
                dico_timestamp[int(row[0])] = [int(row[3])]
            else:
                # Si le film est déjà présent, ajout d'une notation au
compteur
                dico_timestamp[int(row[0])].append(int(row[3]))

# Création d'une liste contenant les utilisateurs
list_user = list(dico_timestamp.keys())

# Initialisation de la liste contenant les timestamps max et l'importance des
timestamps
list_timestamp_max = [i * 0 for i in range(len(list_user))]
importance_timestamp = [i * 0 for i in range(len(list_user))]

# Recherche du timestamp maximum pour chaque utilisateur
for cle, liste in dico_timestamp.items():
    maximum = max(liste)
    list_timestamp_max[list_user.index(cle)] = maximum

# Affichage des statistiques sur les timestamps
print('Longueur de la liste : ', len(list_timestamp_max))
print('Valeur maximale de la liste : ', max(list_timestamp_max))
print('Valeur minimale de la liste : ', min(list_timestamp_max))
print('Moyenne des valeurs de la liste : ',
statistics.mean(list_timestamp_max))

# Création d'une liste contenant les années correspondant aux timestamps max
listannee = [i * 0 for i in range(len(list_user))]

```

```

# Conversion des timestamps en années et calcul de l'importance des timestamps
# Ceux-ci sont divisé en 5 catégories en fonction de leurs année de sortie
for i in range(len(list_user)):
    val = list_timestamp_max[i]
    date = datetime.datetime.fromtimestamp(val)
    annee = date.year
    listannee[i] = annee

    if annee <= 2001:
        importance_timestamp[i] = 1
    elif annee > 2001 and annee <= 2005:
        importance_timestamp[i] = 2
    elif annee > 2005 and annee <= 2009:
        importance_timestamp[i] = 3
    elif annee > 2009 and annee <= 2013:
        importance_timestamp[i] = 4
    elif annee > 2013:
        importance_timestamp[i] = 5

# Écriture de la liste d'importance des timestamps dans un fichier JSON
with open("Importance_timestamp.json", "w") as f:
    json.dump(importance_timestamp, f)

```

1.3.3. En fonction du nombre d'évaluation

Ce code permet de déterminer l'importance de chaque utilisateur en fonction du nombre de notation fourni par l'utilisateur. Il va compter le nombre de notation fourni par utilisateurs, puis la stocker dans une liste d'importance en format JSON afin d'être utilisée lors de l'évaluation de nos modèles.

```

import csv
import scipy.sparse
import json
import statistics

dico_nb_note = {}

# Ouverture du fichier CSV contenant les ratings
with open('data/ratings.csv') as csvfile:
    spamreader = csv.reader(csvfile, delimiter=',')

    # Lecture du fichier ligne par ligne
    for row in spamreader:
        # Si ce n'est pas la première ligne (les headers)
        if spamreader.line_num != 1:
            # Ajout du film au dictionnaire de films

```

```

        if int(row[0]) not in dico_nb_note:
            dico_nb_note[int(row[0])] = 1
        else:
            # Si le film est déjà présent, ajout d'une notation au
compteur
            count = dico_nb_note.get(int(row[0])) + 1
            dico_nb_note[int(row[0])] = count

    # Affichage du dictionnaire contenant le nombre total de ratings par film
    print(dico_nb_note)
    print('Longueur du dictionnaire : ', len(dico_nb_note))
    print('Valeur maximale du dictionnaire : ', max(dico_nb_note.values()))
    print('Valeur minimale du dictionnaire : ', min(dico_nb_note.values()))
    print('Moyenne des valeurs du dictionnaire : ',
statistics.mean(dico_nb_note.values()))

list_user = list(dico_nb_note.keys())

# Création de la liste d'importance en fonction du nombre total de ratings
importance_nb_ratings = [i * 0 for i in range(len(list_user))]
for i in dico_nb_note:
    val = dico_nb_note.get(i)
    importance_nb_ratings[list_user.index(i)] = val

# Écriture de la liste d'importance du nombre de ratings dans un fichier JSON
with open("Importance nb ratings.json", "w") as f:
    json.dump(importance_nb_ratings, f)

```

1.4. Modèle de système de recommandation

1.4.1. User_Based model

La fonction **System_Userbased** effectue des recommandations basées sur les utilisateurs en utilisant une approche de filtrage collaboratif. Elle calcule les scores d'intérêt pour les films non vus par chaque utilisateur, en se basant sur les notations données par ses voisins les plus proches. Elle génère ensuite une liste de recommandations pour chaque utilisateur, en ordonnant les films en fonction de ces scores d'intérêt.

Elle nécessite en paramètre une matrice d'adjacence de type sparse « **mat** », une matrice de similarité également de type sparse « **sim** », un nombre entier « **k** » correspondant au nombre de voisins à prendre en considération et un nombre entier « **nb_rec** » correspondant au nombre d'évaluation que le modèle doit inclure dans les listes de recommandations. La fonction renvoie une liste de listes de recommandations pour chaque utilisateur.

```

from typing import TextIO

import numpy as np
import scipy.sparse
from sklearn.metrics.pairwise import cosine_similarity
import json
%run Evaluation_poids.ipynb
%run Evaluation_Précision_poids.ipynb
%run Nombre_rec.ipynb
%Films_Vus

def System_Userbased(mat,sim, k, nb_rec):
# renvoie une liste comprenant la liste de nb_rec recommandations pour chaque
utilisateur
    # matrice sparse avec les nouveaux scores calculés
    rank=scipy.sparse.csr_matrix((mat.shape[0],mat.shape[1]),dtype=np.single)
    # liste à renvoyer
    listes_recommandations=[]

    # pour chaque utilisateur de la matrice
    for i in range(mat.shape[0]):
        # liste comprenant les similarités des voisins pour identifier les
plus proches
        vois = []
        # similarité entre l'utilisateur d'intérêt et les autres
        simi = sim[i]
        for v in range(len(simi)):
            if v != i:
                # ajout des similarités sauf celle de l'utilisateur avec lui
même
                vois.append(simi[v])
        # liste comprenant les k plus proches voisins de l'utilisateur
d'intérêt
        voisin = []
        sim_total = 0
        # tant que la liste n'atteint pas le nombre de k, on ajoute le ou les
voisins le plus proche
        while len(voisin) < k:
            # indice(s) de(s) voisin(s) le(s) plus proche(s)
            indices_voisins = np.where(simi == max(vois))[0]
            # valeur de la similarité maximale
            sim_max = max(vois)
            # pour tous les voisins avec la similarité maximale
            for y in indices_voisins:
                if len(voisin) < k:
                    # ajout du voisin à la liste des voisins
                    voisin.append(int(y))
                    # on retire la similarité maximale
                    vois.remove(sim_max)

```

```

        sim_total+=simi[int(y)]
    pourcent_sim+=sim_total/len(voisin)

    # liste contenant les films vus par les voisins
    liste_films_voisins=[]
    # pour chaque voisin
    for v in voisin:
        # recupère la liste de films vus par le voisin
        liste = Films_Vus(mat, v)
        # ajout de chaque film à la liste
        for f in liste:
            liste_films_voisins.append(f)
    # pour chaque article
    for j in range(mat.shape[1]):
        # calcul du score pour les films vus par les voisins mais pas par
        l'utilisateur i
        if mat[i,j]==0 and j in liste_films_voisins:
            numerator,denominator=0,0
            for v in range(len(voisin)):
                numerator+=(mat[voisin[v],j]*simi[voisin[v]])
                denominator+=simi[int(voisin[v])]
            # pas de division par zero
            if denominator!=0:
                # score d'intérêt pour le film j par l'utilisateur i
                rank[i,j]=numerator/denominator
            # ne pas recommander des films déjà vus par l'utilisateur i
        elif mat[i, j] == 1:
            rank[i, j] = -1
    # liste pour ordonner les recommandations
    ordre=rank[i].copy()
    # liste de recommandations pour l'utilisateur i
    liste_recommandations=[0]*nb_rec
    for rec in range(len(liste_recommandations)):
        # ajout du score le plus élevé à la liste
        liste_recommandations[rec]=ordre.argmax()
        # suppression de la recommandation ajoutée
        ordre[0,ordre.argmax()]=-0.5
    # ajout de la liste de l'utilisateur i à la liste totale
    listes_recommandations.append(liste_recommandations)
    return listes_recommandations

```

1.4.2. Item_Based model

La fonction **System_Itembased** effectue des recommandations basées sur les articles (ou films) en utilisant une approche de filtrage collaboratif. Elle calcule les scores d'intérêt pour les films similaires aux films vus par chaque utilisateur, en se basant sur les similarités entre les articles

et les notations données par les utilisateurs. Elle génère ensuite une liste de recommandations pour chaque utilisateur, en ordonnant les films en fonction de ces scores d'intérêt.

De la même manière que pour **System_Userbased**, elle prend en paramètre une matrice d'adjacence de type sparse « **mat** », une matrice de similarité également de type sparse « **sim** », un nombre entier « **k** » correspondant au nombre de voisins à prendre en considération et un nombre entier « **nb_rec** » correspondant au nombre d'évaluation que le modèle doit inclure dans les listes de recommandations. La fonction renvoie une liste de listes de recommandations pour chaque utilisateur.

```
import numpy as np
import scipy.sparse
from sklearn.metrics.pairwise import cosine_similarity
import json
%run Evaluation_poids.ipynb
%run Evaluation_Précision_poids.ipynb
%run Nombre_rec.ipynb
%run Films_Vus

def System_Itembased(mat,sim, k, nb_rec):
# renvoie une liste comprenant la liste de nb_rec recommandations pour chaque
utilisateur
    # matrice sparse avec les nouveaux scores calculés
    rank=scipy.sparse.csr_matrix((mat.shape[0],mat.shape[1]),dtype=np.single)
    # liste à renvoyer
    listes_recommandations=[]
    #liste comprenant les voisins de chaque article
    voisins=[]
    # pour chaque article de la matrice
    for i in range(mat.shape[1]):
        # liste comprenant les similarités des voisins pour identifier les
plus proches
        vois = []
        # similarité entre l'article d'intérêt et les autres
        simi = sim[i]
        for v in range(len(simi)):
            if v != i:
                # ajout des similarités sauf celle de l'article avec lui même
                vois.append(simi[v])
        # liste comprenant les k plus proches voisins de l'article d'intérêt
        voisin = []
        sim_total = 0
        # tant que la liste n'atteint pas le nombre de k, on ajoute le ou les
voisins le plus proche
        while len(voisin) < k:
            # indice(s) de(s) voisin(s) le(s) plus proche(s)
```

```

        indices_voisins = np.where(simi == max(vois))[0]
        # valeur de la similarité maximale
        sim_max = max(vois)
        # pour tous les voisins avec la similarité maximale
        for y in indices_voisins:
            if len(voisin) < k:
                # ajout du voisin à la liste des voisins
                voisin.append(int(y))
                # on retire la similarité maximale
                vois.remove(sim_max)
        #ajout de la liste des voisins de l'article i à la liste totale
        voisins.append(voisin)

# pour chaque utilisateur
for j in range(mat.shape[0]):
    # recupère la liste de films vus par l'utilisateur j
    liste_films_vus = Films_Vus(mat, j)
    # liste contenant les films similaires par aux films vus par
l'utilisateur j
    liste_films_voisins = []
    # pour chaque films vus par l'utilisateur j
    for f in liste_films_vus:
        # recupère la liste de films similaires au film vu
        liste_voisins = voisins[f]
        # ajout de chaque film à la liste
        for flm in liste_voisins:
            liste_films_voisins.append(flm)
    for i in range(mat.shape[0]):
        # calcul du score pour les films similaires aux films vus
l'utilisateur i qu'il n'a pas vu
        if mat[j,i]==0 and i in liste_films_voisins:
            numerator,denominator=0,0
            for v in range(len(voisins[i])):
                numerator+=(mat[j,voisins[i][v]]*sim[i][voisins[i][v]])
                denominator+=sim[i][int(voisins[i][v])]
            # pas de division par zero
            if denominator!=0:
                # score d'intérêt pour le film i par l'utilisateur j
                rank[j,i]=numerator/denominator
        elif mat[j,i] == 1:
            rank[j,i] = -1
    # liste pour ordonner les recommandations
    ordre=rank[j].copy()
    # liste de recommandations pour l'utilisateur j
    liste_recommandations=[0]*nb_rec
    for rec in range(len(liste_recommandations)):
        # ajout du score le plus élevé à la liste
        liste_recommandations[rec]=ordre.argmax()

```

```

        # suppression de la recommandation ajoutée
        ordre[0,ordre.argmax()]=-0.5
    # ajout de la liste de l'utilisateur j à la liste totale
    listes_recommandations.append(liste_recommandations)

    return listes_recommandations

```

1.4.3. Content_Based model

La fonction **System_ContentBased** implémente un système de recommandation basé sur le contenu. Elle utilise les similarités entre les articles et les préférences des utilisateurs pour générer des recommandations personnalisées. Les recommandations sont basées sur les scores de similarité entre les articles déjà vus par l'utilisateur et les articles non vus.

Les paramètres de la fonction sont semblable à ceux des systèmes précédents, à savoir, une matrice d'adjacence de type sparse « **mat** », une matrice de similarité également de type sparse « **sim** », un nombre entier « **k** » correspondant au nombre de voisins à prendre en considération et un nombre entier « **nb_rec** » correspondant au nombre d'évaluation que le modèle doit inclure dans les listes de recommandations. La fonction renvoie une liste de listes de recommandations pour chaque utilisateur.

```

import numpy as np
import scipy.sparse
from sklearn.metrics.pairwise import cosine_similarity
import json
%run Evaluation_poids.ipynb
%run Evaluation_Précision_poids.ipynb
%run Nombre_rec.ipynb

def System_ContentBased(mat,sim, nb_rec):
    rank=scipy.sparse.csr_matrix((mat.shape[0],mat.shape[1]),dtype=np.single)
    listes_recommandations=[]
    # pour chaque utilisateur
    for j in range(mat.shape[0]):
        # recupère la liste de films vus par l'utilisateur j
        liste_films_vus = Films_Vus(mat, j)
        nb_films_vus= len(liste_films_vus)
        scores = np.sum(sim[:, liste_films_vus], axis=1)/nb_films_vus # Somme
des similarités avec les films déjà visionnés
        scores[liste_films_vus] = -1 #Attribue un score de -1 pour les films
déjà visionnés
        indices_tri = np.argsort(scores)[::-1] # Trie les indices en fonction
des scores (du plus grand au plus petit)
        top_indices = indices_tri[:nb_rec] # Prend les top_k indices avec les
scores les plus élevés

```

```
listes_recommandations.append(list(top_indices))

return listes_recommandations
```

1.5. Evaluation_Précision_poids

1.5.1. Recall

La fonction **Recall** prend en entrée deux arguments : **rank** qui représente les listes de recommandations générées par un système de recommandation, et **tmat** qui représente une matrice des préférences des utilisateurs pour les articles.

Elle calcule le taux de rappel (recall) d'un système de recommandation en comparant les articles recommandés aux articles notés par les utilisateurs. Le recall mesure la capacité du système à recommander des articles pertinents parmi ceux qui ont été notés.

```
def Recall(rank, tmat):
# renvoie le recall du système de recommandation
# nombre d'article recommandé et noté
n_vp=0
# nombre d'article noté et non recommandé
n_fn=0
# pour chaque liste de recommandations
for i in range(len(rank)):
# récupère la liste des films notés par l'utilisateur
liste_films_vus = Films_Vus(tmat, i)
# pour chaque films notés par l'utilisateur
for j in range(len(liste_films_vus)):
# recommandé ou non
if liste_films_vus[j] in rank[i]:
n_vp+=1
else:
n_fn+=1
# recall
return (n_vp)/(n_vp+n_fn) #items bien recommandés/ensemble des items pertinents
```

1.5.2. RecallPoids

La fonction **RecallPoids** est une variation de la fonction **Recall** avec une pondération des utilisateurs. Elle prend en entrée trois arguments : **rank** qui représente les listes de recommandations générées par un système de recommandation, **tmat** qui représente une matrice des préférences des utilisateurs pour les articles, et **importance** qui représente l'importance pondérée de chaque utilisateur.

La fonction **RecallPoids** calcule le taux de rappel (recall) d'un système de recommandation en tenant compte des poids (importance) des utilisateurs. Les recommandations des utilisateurs

sont pondérées en fonction de leur importance, ce qui permet de prendre en considération l'impact des utilisateurs les plus importants dans le calcul du recall.

```
def RecallPoids(rank, tmat, importance):
# renvoie le recall du système de recommandation
    tot_importance=sum(importance)
    # nombre d'article recommandé et noté
    n_vp=0
    # nombre d'article noté et non recommandé
    n_fn=0
    # pour chaque liste de recommandations (recommandations faites à
    l'utilisateur)
    for i in range(len(rank)):
        # récupère la liste des films notés par l'utilisateur
        liste_films_vus = Films_Vus(tmat, i)
        #nombre d'article recommandé et noté pour l'utilisateur
        n_vp_ut=0
        #Nombre d'article non recommandé et noté pour l'utilisateur
        n_fn_ut=0
        # pour chaque films notés par l'utilisateur
        for j in range(len(liste_films_vus)):
            # recommandé ou non
            if liste_films_vus[j] in rank[i]:
                n_vp_ut+=1
            else:
                n_fn_ut+=1
        #prise en compte de l'importance de l'utilisateur dans le calcul de la
    métrique
        n_vp+=n_vp_ut*importance[i]/tot_importance
        n_fn+=n_fn_ut*importance[i]/tot_importance

    # recall

    return (n_vp)/(n_vp+n_fn) #items bien recommandés/ensemble des items
pertinents
```

1.5.3. Précision

La fonction **Precision** calcule la précision d'un système de recommandation. Elle prend en entrée deux arguments : **rank**, qui représente les listes de recommandations générées par le système, et **tmat**, qui représente une matrice des préférences des utilisateurs pour les articles.

Elle évalue la performance d'un système de recommandation en calculant la proportion d'articles recommandés qui ont également été notés par les utilisateurs. Une précision plus élevée indique que le système de recommandation fournit des suggestions pertinentes et correspondantes aux préférences des utilisateurs.

```
def Precision(rank, tmat):
# renvoie la precision du système de recommandation
    # nombre d'article recommandé et noté
    n_vp=0
    # nombre d'article non noté et recommandé
    n_fp=0
    # pour chaque liste de recommandations
    for i in range(len(rank)):
        # récupère la liste des films notés par l'utilisateur
        liste_films_vus = Films_Vus(tmat, i)
        # pour chaque films notés par l'utilisateur
        for j in range(len(rank[i])):
            # recommandé ou non
            if rank[i][j] in liste_films_vus:
                n_vp+=1
            else:
                n_fp+=1
    # recall
    return (n_vp)/(n_vp+n_fp)
```

1.5.4. PrécisionPoids

La fonction **PrecisionPoids** calcule la précision d'un système de recommandation en prenant en compte l'importance des utilisateurs. Elle prend trois arguments en entrée : **rank** qui représente les listes de recommandations générées par le système, **tmat** qui représente une matrice des préférences des utilisateurs pour les articles, et **importance** qui représente l'importance attribuée à chaque utilisateur.

Nous évaluons la performance d'un système de recommandation en calculant la précision en tenant compte de l'importance attribuée à chaque utilisateur. Cela permet de donner plus de poids aux recommandations des utilisateurs importants dans le calcul de la métrique.

```
def PrecisionPoids(rank, tmat, importance):
# renvoie la precision du système de recommandation
    tot_importance=sum(importance)
    # nombre d'article recommandé et noté
    n_vp=0
    # nombre d'article non noté et recommandé
    n_fp=0
    # pour chaque liste de recommandations
    for i in range(len(rank)):
        # récupère la liste des films notés par l'utilisateur
        liste_films_vus = Films_Vus(tmat, i)
        #nombre d'article recommandé et noté par l'utilisateur
        n_vp_ut=0
        #nombre d'article non recommandé et noté par l'utilisateur
```

```

n_fp_ut=0
# pour chaque films notés par l'utilisateur
for j in range(len(rank[i])):
    # recommandé ou non
    if rank[i][j] in liste_films_vus:
        n_vp_ut+=1

    else:
        n_fp_ut+=1
#prise en compte de l'importance de l'utilisateur dans le calcul de la
métrique
n_vp+=n_vp_ut*importance[i]/tot_importance
n_fp+=n_fp_ut*importance[i]/tot_importance
# recall
return (n_vp)/(n_vp+n_fp)

```

1.5.5. nDCG

La fonction **nDCG** calcule le nDCG (Normalized Discounted Cumulative Gain) moyen des listes de recommandations générées par un système de recommandation. Elle prend deux arguments en entrée : **rank**, qui représente les listes de recommandations générées par le système, et **tmat**, qui représente une matrice des préférences des utilisateurs pour les articles.

Elle permet d'évaluer la qualité des listes de recommandations d'un système en calculant le nDCG moyen. Le nDCG tient compte de la position des articles recommandés dans les listes et de leur pertinence, en se basant sur les notations des utilisateurs dans le test set.

```

import math
def nDCG(rank, tmat):
# renvoie le nDCG moyen des listes de recommandations
    # somme de tous les nDCG
    sumndcg=0
    # pour chaque liste de recommandations
    for i in range(len(rank)):
        # liste de recommandations idéale selon le test set
        dcg=0
        idcg=0
        # position de l'article dans la liste ideale
        pos=1
        liste_rec_t=[]
        # calcul du IDCG
        for j in range(tmat.shape[1]):
            if tmat[i,j] ==1:
                idcg+=(1/math.log(pos+1,2))
                liste_rec_t.append(j)
                pos+=1
        # position de l'article dans la liste de recommandation

```

```

    pos=1
    # calcul du DCG
    for j in range(len(rank[i])):
        dcg+=((math.pow(2,tmat[i,rank[i][j]])-1)/math.log(pos+1,2))
        pos+=1
    # calcul du nDCG
    try:
        sumndcg+=(dcg/idcg)
    except:
        sumndcg +=0
    # Moyenne des nDCG
    return sumndcg/len(rank)

```

1.5.6. nDCGPoids

La fonction **nDCGPoids** calcule le nDCG (Normalized Discounted Cumulative Gain) moyen des listes de recommandations générées par un système de recommandation, en tenant compte de l'importance attribuée à chaque utilisateur.

Elle prend trois arguments en entrée : **rank** qui représente les listes de recommandations générées par le système, **tmat** qui représente une matrice des préférences des utilisateurs pour les articles, et **importance** qui représente l'importance attribuée à chaque utilisateur.

```

import math

def nDCGPoids(rank, tmat, importance):
    # renvoie le nDCG moyen des listes de recommandations

    # somme de tous les nDCG
    sumndcg = 0

    # pour chaque liste de recommandations
    for i in range(len(rank)):
        # liste de recommandations idéale selon le test set
        dcg = 0
        idcg = 0

        # position de l'article dans la liste idéale
        pos = 1
        liste_rec_t = []

        # calcul du IDCG
        for j in range(tmat.shape[1]):
            if tmat[i, j] == 1:
                idcg += (1 / math.log(pos + 1, 2))
                liste_rec_t.append(j)
                pos += 1

```

```

    # position de l'article dans la liste de recommandation
    pos = 1

    # calcul du DCG
    for j in range(len(rank[i])):
        dcg += ((math.pow(2, tmat[i, rank[i][j]]) - 1) / math.log(pos + 1,
2))
        pos += 1

    # calcul du nDCG en tenant compte de l'importance de l'utilisateur
    try:
        sumndcg += (dcg / idcg) * importance[i]
    except:
        sumndcg += 0

    # Moyenne des nDCG en tenant compte de l'importance attribuée à chaque
utilisateur
    total_importance = sum(importance)
    if total_importance != 0:
        return sumndcg / total_importance
    else:
        return 0

```

1.6. Evaluation_poids

1.6.1. DistanceUtilisateur

La fonction **DistanceUtilisateur** calcule la moyenne des distances d'utilisateur par rapport aux articles dans leur liste de recommandations. Elle nous permet de déterminer le degré de nouveauté généré par un système de recommandation.

Elle évalue la proximité entre les articles recommandés et les articles vus par les utilisateurs en calculant la moyenne des distances d'utilisateur. La distance d'utilisateur est définie comme la dissimilarité entre les articles, mesurée en utilisant la similarité entre les articles. Plus la distance d'utilisateur est faible, plus les articles recommandés sont similaires aux articles vus par les utilisateurs.

Elle prend deux arguments en entrée : **rank**, qui représente les listes de recommandations générées par le système, et **tmat**, qui représente une matrice des préférences des utilisateurs pour les articles.

```

def DistanceUtilisateur(rank, sim_article):
    # renvoie la moyenne des distances d'utilisateur par rapport aux articles dans
leur liste
    # matrice adjacente pour identifier les films vus par l'utilisateur

```

```

mat_not = []
with open("Matrice adjacente.json", "r") as file:
    mat_not = json.load(file)
mat_not=scipy.sparse.csr_matrix(mat_not)
# somme de tous les distances d'utilisateur
sum_total=0
# pour chaque liste de recommandations
for i in range(len(rank)):
    # liste des films vus par l'utilisateur
    liste_films_vus= Films_Vus(mat_not,i)
    # somme des dissimilarités des articles
    sum=0
    # pour chaque article de la liste de recommandations
    for j in range(len(rank[i])):
        # pour chaque articles vus
        for k in range(len(liste_films_vus)):
            # dissimilarité entre l'article i et l'article k
            sum+=1-(sim_article[rank[i][j]][liste_films_vus[k]])
        # ajout de la mesure à la somme totale
        sum_total+=(sum/(len(rank[i])*len(liste_films_vus)))
# moyenne de toutes les distances
return sum_total/len(rank)

```

1.6.2. DistanceUtilisateurPoids

La fonction **DistanceUtilisateurPoids** est une version modifiée de la fonction **DistanceUtilisateur** qui prend également en compte l'importance attribuée à chaque utilisateur dans le calcul de la moyenne des distances d'utilisateur.

Elle calcule la moyenne des distances d'utilisateur par rapport aux articles dans leur liste de recommandations, en tenant compte de l'importance attribuée à chaque utilisateur. Cette mesure permet d'évaluer la proximité entre les articles recommandés et les articles vus par les utilisateurs, en accordant plus de poids aux utilisateurs importants.

Elle prend trois arguments en entrée : **rank** qui représente les listes de recommandations générées par le système, **tmat** qui représente une matrice des préférences des utilisateurs pour les articles, et **importance** qui représente l'importance attribuée à chaque utilisateur.

```

def DistanceUtilisateurPoids(rank, sim_article, importance):
    # renvoie la moyenne des distances d'utilisateur par rapport aux articles
    dans leur liste
    # matrice adjacente pour identifier les films vus par l'utilisateur
    mat_not = []
    with open("Matrice adjacente.json", "r") as file:
        mat_not = json.load(file)

```

```

mat_not = scipy.sparse.csr_matrix(mat_not)

# somme de toutes les distances d'utilisateur
sum_total = 0

# pour chaque liste de recommandations
for i in range(len(rank)):
    # liste des films vus par l'utilisateur
    liste_films_vus = Films_Vus(mat_not, i)

    # somme des dissimilarités des articles
    sum = 0

    # pour chaque article de la liste de recommandations
    for j in range(len(rank[i])):
        # pour chaque article vu
        for k in range(len(liste_films_vus)):
            # dissimilarité entre l'article i et l'article k
            sum += 1 - sim_article[rank[i][j]][liste_films_vus[k]]

    # ajout de la mesure à la somme totale, en tenant compte de
    # l'importance de l'utilisateur
    sum_total += (sum / (len(rank[i]) * len(liste_films_vus))) *
importance[i]

    # moyenne de toutes les distances en tenant compte de l'importance
    # attribuée à chaque utilisateur
    total_importance = np.sum(importance)
    if total_importance != 0:
        return sum_total / total_importance
    else:
        return 0

```

1.6.3. ILD

La fonction **ILD** calcule la moyenne des dissimilarités intra-liste (IDL) pour toutes les listes de recommandations. Cela permet d'évaluer la diversité des articles au sein des listes de recommandations, en mesurant la dissimilarité entre les articles recommandés les uns par rapport aux autres au sein de chaque liste.

Elle prend deux arguments en entrée : **rank**, qui représente les listes de recommandations générées par le système, et **tmat**, qui représente une matrice des préférences des utilisateurs pour les articles.

```

def ILD(rank, sim_article):
    # somme des dissimilarités intra-liste de tous les utilisateurs
    sum_total=0

```

```

# pour chaque liste de recommandations
for i in range(len(rank)):
    # somme des dissimilarités des articles de la liste
    sum=0
    for j in range(len(rank[i])):
        for k in range(len(rank[i])):
            sum+=1-(sim_article[rank[i][j]][rank[i][k]])
    # ajout de l'IDL à la somme
    sum_total+=(sum/(len(rank[i])*(len(rank[i])-1)))
# moyenne des IDL
return sum_total/len(rank)

```

1.6.4. ILDPoids

La fonction **ILDPoids** est une extension de la fonction **ILD** qui prend en compte l'importance attribuée à chaque utilisateur.

Elle calcule la moyenne des dissimilarités intra-liste pondérées pour toutes les listes de recommandations, en tenant compte de l'importance attribuée à chaque utilisateur. Cela permet d'évaluer la diversité des articles au sein des listes de recommandations en accordant plus de poids aux utilisateurs importants.

Elle prend trois arguments en entrée : **rank** qui représente les listes de recommandations générées par le système, **tmat** qui représente une matrice des préférences des utilisateurs pour les articles, et **importance** qui représente l'importance attribuée à chaque utilisateur.

```

def ILDPoids(rank, sim_article, importance):
    # somme des dissimilarités intra-liste de tous les utilisateurs
    sum_total = 0

    # pour chaque liste de recommandations
    for i in range(len(rank)):
        # somme des dissimilarités des articles de la liste
        sum = 0

        for j in range(len(rank[i])):
            for k in range(len(rank[i])):
                # dissimilarité entre l'article j et l'article k dans la liste
                sum += 1 - sim_article[rank[i][j]][rank[i][k]]

        # ajout de l'IDL à la somme totale, en tenant compte de l'importance
        # de l'utilisateur
        sum_total += (sum / (len(rank[i]) * (len(rank[i])-1))) * importance[i]

    # moyenne des IDL en tenant compte de l'importance attribuée à chaque
    # utilisateur

```

```
total_importance = np.sum(importance)
if total_importance != 0:
    return sum_total / total_importance
else:
    return 0
```