# Annexe 2 : Code

## Plan de l'Annexe 2

## 1. Importations des modules

Lors de la création et l'exécution de nos algorithmes, nous ferons appel à plusieurs librairies disponibles avec le langage Python. Tout d'abord, la librairie SciPy sera utilisée pour créer des matrices dites « sparse », des matrices creuses en français. Ce type de matrice contient plus d'éléments égaux à zéro que d'éléments différents de zéro. Cette librairie nous permet donc d'utiliser moins de mémoire et d'économiser du temps de calcul (Gupta, 2021). Ensuite, pour le stockage de nos données après et pendant nos manipulations, nous utiliserons des fichiers de type JSON, JavaScript Object Notation. Ces fichiers sont sous un format léger d'échange de données, en effet, ce format est facilement lisible par les humains et facilement analysé par les machines (JSON, 2021). La librairie json prenant en charge ce type de données nous sera donc d'une grande utilité. De plus, la base de données utilisée qui sera présentée ci-après est stockée en format CSV, la librairie « csv » nous permettra alors d'extraire les données de notre fichier de manière optimale. Enfin, la dernière librairie utilisée sera scikit-learn, un outil simple et efficace pour l'apprentissage machine en Python (sklearn, 2021). Dans ce travail, le sous-module sklearn.metrics.pairwise nous permettra de calculer une mesure de similarité.

```
# pour les matrices
import numpy as np
# matrices creuses
import scipy.sparse
# calcul de la similarité du cosinus
from sklearn.metrics.pairwise import cosine_similarity
# fichiers de données json
import json
# fichiers de départ en csv
import csv
```

## 2. Extraction de la base de données

Ce script permet de lire le fichier comprenant toutes les notations. Il extrait tout d'abord tous les films en comptant le nombre de notations respectives à ceux-ci. Ensuite, il extrait ligne par ligne les utilisateurs et les notations des films ayant au moins 10 notations. Il crée ensuite une matrice adjacente de longueur nombre_d'utilisateurs x nombre_de_films comprenant des 0 et incrémente celle-ci avec des 1 lorsqu'un film a été noté par l'utilisateur. Il stocke cette matrice dans un fichier json.

```
# dictionnaire reprenant les films notés par utilisateur
dico_user={}
# liste reprenant tous les films
movie=[]
# dictionnaire reprenant le nombre de notations par film
```

```
dico_movie={}
# ouverture du fichier
with open('ml-latest-small\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
# réouverture du fichier pour l'extraction des données
with open('ml-latest-small\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()
# 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
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[i-1,movie.index(dico_user.get(i)[0][j])]=1
        matrice_notation[i-
1,movie.index(dico_user.get(i)[0][j])]=dico_user.get(i)[1][j]
# conservation dans un fichier json
with open("Matrice notations", "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)
```

### 2.1. Listes des films visionnés par l'utilisateur

Cette méthode prend en paramètre ;

- Mat : la matrice binaire de notation (type : matrice sparse)

- User : l'identifiant d'un utilisateur (type : entier)

Elle renvoie une liste comprenant les indices de tous les films vus par l'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
```

## 3. Division de la base de données pour la validation croisée

Ce script divise la base de données stockée sous forme de matrice binaire dans un fichier json en dix sous-ensembles qui serviront d'ensemble test. Pour se faire, il constitue un dictionnaire comprenant dix matrices de mêmes dimensions que la matrice binaire d'origine et attribue chaque note de cette matrice à une des dix de manière aléatoire. Ce dictionnaire est ensuite stocké dans un fichier json afin que chaque système puisse utiliser les mêmes ensembles test.

```
# 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 façon 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)
```

## 4. Mesure de similarité du cosinus

### 4.1. Entre les articles

Cette méthode prend en paramètre :

- Mat : matrice binaire de notations (type : matrice sparse)

Elle renvoie une matrice de longueur nombre_d'articles x nombre_d'articles comprenant les cosinus entre chaque paire d'articles. Tout d'abord, cette méthode parcourt pour chaque article de la base de données les notations reçues par les utilisateurs afin de calculer la somme de toutes ces notations au carré. Ces sommes seront stockées dans une matrice pour tous les articles. Ensuite, pour chaque paire d'articles, la méthode calculera la somme de la multiplication des notations communes aux deux articles. Enfin, le cosinus est calculé en divisant la somme des notations communes multipliées par la multiplication des racines carrées des sommes premièrement calculées par article. Cette matrice étant symétrique nous pouvons calculer uniquement la partie inférieure à la diagonale.

```
def calcul_Cosinus_Article(mat):
# méthode qui renvoie une matrice avec les cosinus entre paires
d'articles
    # matrice comprenant les cosinus
    cos=np.zeros([mat.get_shape()[1],mat.get_shape()[1]])
    # matrice comprenant la somme des carrés des notations de chaque
article
    matsum=np.zeros([mat.get_shape()[1]])
    # pour chaque article
    for i in range(mat.get_shape()[1]):
        # somme mise à zéro
        sumi=0
        # pour chaque utilisateur
        for j in range(mat.get_shape()[0]):
            # si l'article i a été noté par l'utilisateur j
            if mat[j,i] !=0:
                # ajout de la notation au carré à la somme
                sumi+=pow(mat[j,i],2)
        # ajout de cette somme dans la liste
        matsum[i]=sumi
    # pour chaque paire d'articles
    for i in range(mat.get_shape()[1]):
        for j in range(mat.get_shape()[1]):
            # somme de la multiplication des notations entre les paires
d'articles
            sumu=0
```

```
            # pour chaque utilisateur
            for c in range(mat.get_shape()[0]):
                # si notation commune
                if mat[c,i] != 0 and mat[c,j] != 0:
                    # ajout de la multiplication à la somme
                    sumu+=(mat[c,i]*mat[c,j])
            # pas de division par zéro
            if matsum[i]*matsum[j] !=0:
                # matrice symétrique
                cos[i][j]=sumu/pow((matsum[i]*matsum[j]),1/2)
                cos[j][i] = sumu / pow((matsum[i] * matsum[j]), 1 / 2)
    return cos
```

### 4.2. Entre les utilisateurs

Cette méthode prend en paramètre :

- Mat : matrice binaire de notations (type : matrice sparse)

Elle renvoie une matrice de longueur nombre_d'utilisateurs x nombre_d'utilisateurs comprenant les cosinus entre chaque paire d'utilisateurs. Tout d'abord, cette méthode parcourt pour chaque utilisateur de la base de données les notations données par celui-ci afin de calculer la somme de toutes ses notations au carré. Ces sommes seront stockées dans une matrice pour tous les utilisateurs. Ensuite, pour chaque paire d'utilisateurs, la méthode calculera la somme de la multiplication des notations communes aux deux utilisateurs. Enfin, le cosinus est calculé en divisant la somme des notations communes multipliées par la multiplication des racines carrées des sommes premièrement calculées par utilisateur. Cette matrice étant symétrique nous pouvons calculer uniquement la partie inférieure à la diagonale.

```
def calcul_Cosinus_Utilisateur(mat):
# méthode qui renvoie une matrice avec les cosinus entre paires
d'utilisateurs
    # matrice comprenant les cosinus
    cos=np.zeros([mat.get_shape()[0],mat.get_shape()[0]])
    # matrice comprenant la somme des carrés des notations de chaque
utilisateur
    matsum=np.zeros([mat.get_shape()[0]])
    # pour chaque utilisateur
    for i in range(mat.get_shape()[0]):
        # somme des notations
        sumi=0
        # pour chaque article
        for j in range(mat.get_shape()[1]):
            # s'il a été noté par l'utilisateur i
            if mat[i,j] !=0:
                sumi+=pow(mat[i,j],2)
        # ajout de la somme à la liste
        matsum[i]=sumi
    # pour chaque paire d'utilisateurs
    for i in range(mat.get_shape()[0]):
```

```
        for j in range(mat.get_shape()[0]):
            # somme de la multiplication des notations communes
            sumu=0
            # pour chaque article
            for c in range(mat.get_shape()[1]):
                # si notation commune
                if mat[i,c] != 0 and mat[j,c] != 0:
                    sumu+=(mat[i,c]*mat[j,c])
            if matsum[i]*matsum[j] !=0:
                # matrice symétrique
                cos[i][j]=sumu/pow((matsum[i]*matsum[j]),1/2)
                cos[j][i] = sumu / pow((matsum[i] * matsum[j]), 1 / 2)
    return cos
```

## 5. Mesures de centralité sur graphe

### 5.1. Intermédiarité

Cette méthode prend en paramètres ;

- G : liste reprenant tous les nœuds du graphe (type : liste)
- Matrix : matrice adjacente reprenant les liens existants (type : matrice sparse)

Cette méthode calcule la longueur du chemin le plus court entre chaque paire de nœuds, ainsi que le nombre de chemins géodésiques et la liste des nœuds faisant partie de ces chemins. Grâce à cela, elle calcule la mesure d'intermédiarité en additionnant le nombre de fois qu'un nœud apparait sur un chemin géodésique. Chaque apparition est divisée par le nombre de chemins géodésiques existants. Cette valeur est stockée dans une liste contenant toutes les valeurs d'intermédiarité de chaque nœud du graphe. Cette méthode renvoie ensuite cette liste.

```
def betweenness(G, matrix):
# renvoie une liste avec la valeur d'intermédiarité pour chaque nœud du
graphe G
    infinity = float("inf")
    bet=[0]*len(G)
    undirected=False
    # pour chaque noeud du graphe
    for i in G:
        # liste de tous les noeuds entre le noeud i et le noeud j
        Predecssor_ij=[[i]]*len(G)
        # nombre de chemins entre le noeud i et le noeud j
        num_path=[0] * len(G)
        # longueur du plus court chemin entre les deux noeuds
        min_cost=np.array([infinity]*len(G))
        # chemin entre le noeud et lui-même
        num_path[i]=1
        # Noeuds déjà atteints
        S=[]
        # Noeuds restants à atteindre
        Q=list(scipy.sparse.find(matrix[i,:])[1])
        # Noeuds directement liés au noeud i
        for k in Q:
```

```
            min_cost[k]=matrix[i,k]
            num_path[k] = 1
        # distance entre le noeud i et lui-même
        min_cost[i] = 0
        # Tant que pas tous les noeuds atteints
        while Q:
            mymin, j = infinity, -infinity
            for k in Q:
                # plus court chemin trouvé
                if min_cost[k] < mymin:
                    mymin = min_cost[k]
                    j = k
            # graphe non connecté
            if mymin == infinity:
                undirected=True
            # On a atteint j de tous les façons possibles
            else:
                Q.remove(j)
                S.append(j)
            # pour chaque noeud directement lié à j
            for k in scipy.sparse.find(matrix[j,:])[1]:
                # noeud pas encore atteint jusque là
                if min_cost[k] == infinity:
                    min_cost[k]=min_cost[j]+matrix[j,k]
                    Q.append(k)
                # nouveau chemin le plus court trouvé
                elif min_cost[j]+matrix[j,k]<min_cost[k]:
                    min_cost[k]=min_cost[j]+matrix[j,k]
                    num_path[k]=1
                    Predecssor_ij[k]=Predecssor_ij[j][1:]+[j]
                # chemin géodésique supplémentaire
                if min_cost[k]==min_cost[j]+matrix[j,k]:
                    num_path[k]+=num_path[j]

Predecssor_ij[k]=Predecssor_ij[k]+Predecssor_ij[j][1:]+[j]
                    num_path[k]=num_path[k]+num_path[j]
        # calcul de l'intermédiarité de manière récursive
        dep=[0]*len(G)
        while S:
            k=S[-1]
            S.pop(-1)
            for j in Predecssor_ij[k]:
                if j != i and j !=k:
                    dep[j] += ((num_path[j] / num_path[k]) * (1 +
dep[k]))
            bet[k] += dep[k]
        if undirected:
            for b in range(len(bet)):
                bet[b] = 0.5 * bet[b]
    return bet
```

### 5.2. Proximité

Cette méthode prend en paramètres ;

- G : liste reprenant tous les nœuds du graphe (type : liste)
- Matrix : matrice adjacente reprenant les liens existants (type : matrice sparse)

Cette méthode calcule la longueur du chemin le plus court entre chaque paire de nœuds. Grâce à cela, elle calcule la mesure de proximité en additionnant pour un nœud donné la longueur des chemins géodésiques entre ce nœud et tous les autres nœuds du graphe. Cette somme permet ensuite de calculer la proximité de ce nœud, cette valeur est stockée dans une liste contenant toutes les valeurs de proximité de chaque nœud du graphe. Cette méthode renvoie ensuite cette liste.

```
def closeness(G, matrix):
# méthode qui renvoie la valeur de proximité pour chaque noeud du graphe
G
    infinity = float("inf")
    #liste contenant la proximité
    clo=[0]*len(G)
    # pour chaque noeud du graphe G
    for i in G:
        # plus court chemin entre le noeud i et tous les autres noeuds
        min_cost=[infinity]*len(G)
        # noeuds déjà atteints
        S=[]
        # noeuds restants (au départ, cette liste comprend tous les
noeuds directement liés au noeud i)
        Q=list(scipy.sparse.find(matrix[i,:])[1])
        # pour chaque noeud restant
        for k in Q:
            # plus court chemin de longueur un
            min_cost[k]=matrix[i,k]
        # entre le noeud et lui-même
        min_cost[i] = 0
        # tant qu'il reste des noeuds non atteints
        while Q:
            # trouve le chemin le plus court, initialisation du min à
l'infinie
            mymin, j = infinity, -infinity
            # pour chaque noeud restant
            for k in Q:
                # si on a atteint le noeud
                if min_cost[k] < mymin:
                    mymin = min_cost[k]
                    j = k
                    # on le retire des noeuds restants
                    Q.remove(j)
                    S.append(j)
            # on parcourt chaque noeud depuis le noeud le plus proche
trouvé
            for k in scipy.sparse.find(matrix[j,:])[1]:
                # si min_cost toujours égal à l'infinie c'est que le
noeud n'avait pas été trouvé avant
                if min_cost[k] == infinity:
                    min_cost[k]=min_cost[j]+matrix[j,k]
                    Q.append(k)
                # si on a trouvé un chemin plus court vers ce noeud
                elif min_cost[j]+matrix[j,k]<min_cost[k]:
                    min_cost[k]=min_cost[j]+matrix[j,k]
        # somme de tous les chemins les plus courts
        total_cost=0
        for k in range(len(min_cost)):
```

```
            total_cost+=min_cost[k]
        # calcul de la proximité
        clo[i]=(len(G)-1)/total_cost
    return clo
```

## 6. Systèmes de recommandation de référence

### 6.1. Basé sur les utilisateurs

Cette méthode prend en paramètre ;

- Mat : matrice binaire de notations (type : matrice sparse)
- Sim : matrice comprenant les mesures de similarité entre utilisateurs (type : matrice)
- K : nombre de voisins dans un voisinage (type : entier)
- Nb_rec : nombre de recommandations à produire dans une liste (type : entier)

Cette méthode renvoie une liste de recommandations pour chaque utilisateur, cette liste sera de longueur nb_rec. Pour chaque utilisateur, cette méthode choisit tout d'abord les k plus proches voisins de celui-ci grâce à la mesure de similarité entrée en paramètre. Ensuite, elle remplit une liste comprenant tous les films visionnés par tous les voisins de cet utilisateur. Les scores des films de cette liste seront calculés excepter pour les films visionnés par l'utilisateur d'intérêt, ces derniers auront le score de -1 afin d'éviter de les recommander. Enfin, les films recommandés seront ceux ayant obtenu le plus haut score.

```
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.get_shape()[0],mat.get_shape()[1]),d
type=np.single)
    # liste à renvoyer
    listes_recommandations=[]

    # pour chaque utilisateur de la matrice
    for i in range(mat.get_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 = []
        # tant que la liste n'atteint pas le nombre de k, on ajoute le
ou les voisins le plus proches
        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)
            # liste contenant les films vus par les voisins
            liste_films_voisins=[]
            # pour chaque voisin
            for v in voisin:
                # récupère 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.get_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
```

### 6.2. Basé sur les articles

Cette méthode prend en paramètre ;

- Mat : matrice binaire de notations (type : matrice sparse)
- Sim : matrice comprenant les mesures de similarité entre articles (type : matrice)
- K : nombre de voisins dans un voisinage (type : entier)
- Nb_rec : nombre de recommandations à produire dans une liste (type : entier)

Cette méthode renvoie une liste de recommandations pour chaque utilisateur, cette liste sera de longueur nb_rec. Pour chaque article, cette méthode choisit tout d'abord les k plus proches voisins de celui-ci grâce à la mesure de similarité entrée en paramètre. Ensuite, pour chaque utilisateur, elle remplit une liste comprenant tous les films similaires aux films visionnés par l'utilisateur. Les scores des films de cette liste seront calculés excepter pour les films visionnés par l'utilisateur d'intérêt, ces derniers auront le score de -1 afin d'éviter de les recommander. Enfin, les films recommandés seront ceux ayant obtenu le plus haut score.

```
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.get_shape()[0],mat.get_shape()[1]),dtyp
e=np.single)
    # liste à renvoyer
    listes_recommandations=[]
    pourcent_sim=0
    #liste comprenant les voisins de chaque article
    voisins=[]
    # pour chaque article de la matrice
    for i in range(mat.get_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 = []
        # tant que la liste n'atteint pas le nombre de k, on ajoute le ou
les voisins le plus proches
        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.get_shape()[0]):
        # récupère 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:
            # récupère 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)

        # pour chaque article
        for i in range(mat.get_shape()[1]):
         # 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
        print(j)
        # ajout de la liste de l'utilisateur j à la liste totale
        listes_recommandations.append(liste_recommandations)
    return listes_recommandations
```

## 7. Systèmes de recommandation intégrant des mesures de centralité sur graphe

### 7.1. Basé sur les utilisateurs intégrant une mesure d'intermédiarité

Cette méthode prend en paramètre ;

- Mat : matrice binaire de notations (type : matrice sparse)
- Sim : matrice comprenant les mesures de similarité entre utilisateurs (type : matrice)
- K : nombre de voisins dans un voisinage (type : entier)
- Nb_rec : nombre de recommandations à produire dans une liste (type : entier)
- Bet : liste comprenant les mesures d'intermédiarité pour chaque utilisateur (type : liste)

Cette méthode renvoie une liste de recommandations pour chaque utilisateur, cette liste sera de longueur nb_rec. Pour chaque utilisateur, cette méthode choisit tout d'abord les k plus proches voisins de celui-ci grâce à une nouvelle mesure de similarité calculée en multipliant la mesure de similarité et d'intermédiarité entrées en paramètre. Ensuite, elle remplit une liste comprenant tous les films visionnés par tous les voisins de cet utilisateur. Les scores des films de cette liste seront calculés excepter pour les films visionnés par l'utilisateur d'intérêt, ces derniers auront le score de -1 afin d'éviter de les recommander. Enfin, les films recommandés seront ceux ayant obtenu le plus haut score.

```
def System_Userbased_Bet(mat,sim, k, nb_rec, bet):
# 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.get_shape()[0],mat.get_shape()[1]),dtyp
e=np.single)
    # liste à renvoyer
    listes_recommandations=[]

    # pour chaque utilisateur de la matrice
    for i in range(mat.get_shape()[0]):
        # liste comprenant les similarités multipliées par l'indice
d'intermédiarité des voisins pour identifier les plus proches
        vois = []
        # similarité entre l'utilisateur d'intérêt et les autres
        simi = sim[i]
        new_sim=[]
        for v in range(len(simi)):
            if v != i:
                # ajout des similarités multipliées par l'intermédiarité
sauf celle de l'utilisateur avec lui-même
                vois.append(simi[v]*bet[v])
                new_sim.append(simi[v]*bet[v])
            else:
                new_sim.append(1)
        # liste comprenant les k plus proches voisins de l'utilisateur
d'intérêt
        voisin = []
        # tant que la liste n'atteint pas le nombre de k, on ajoute le ou
les voisins le plus proches
        while len(voisin) < k:
            # indice(s) de(s) voisin(s) le(s) plus proche(s)
            indices_voisins = np.where(new_sim == 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)
        # liste contenant les films vus par les voisins
        liste_films_voisins=[]
        # pour chaque voisin
```

```
        for v in voisin:
            # récupère 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.get_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]]*bet[voisin[v]])
                    denominator+=simi[int(voisin[v])]*bet[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
            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
        print(i)
        # ajout de la liste de l'utilisateur i à la liste totale
        listes_recommandations.append(liste_recommandations)
    return listes_recommandations
```

### 7.2. Basé sur les utilisateurs intégrant une mesure de proximité

Cette méthode prend en paramètre :

- Mat : matrice binaire de notations (type : matrice sparse)
- Clo : liste comprenant les mesures de proximité de chaque article (type : liste)

Elle renvoie une matrice de longueur nombre_d'utilisateurs x nombre_d'utilisateurs comprenant les cosinus entre chaque paire d'utilisateurs. Tout d'abord, cette méthode parcourt pour chaque utilisateur de la base de données les notations données par celui-ci afin de calculer la somme de toutes ses notations au carré. Les notations seront divisées par la valeur de proximité de l'article noté au carré avant d'être comptabilisées dans la somme. Ces sommes seront stockées dans une matrice pour tous les utilisateurs. Ensuite, pour chaque paire d'utilisateurs, la méthode calculera la somme de la multiplication des notations communes aux deux utilisateurs. Encore une fois, les notations seront divisées par la valeur de proximité de l'article au carré. Enfin, le cosinus est calculé en divisant la somme des notations communes

multipliées par la multiplication des racines carrées des sommes premièrement calculées par utilisateur. Cette matrice étant symétrique nous pouvons calculer uniquement la partie inférieure à la diagonale.

```
def calcul_Cosinus_Utilisateur(mat,clo):
    cos=np.zeros([mat.get_shape()[0],mat.get_shape()[0]])
    matsum=np.zeros([mat.get_shape()[0]])
    for i in range(mat.get_shape()[0]):
        sumi=0
        for j in range(mat.get_shape()[1]):
            if mat[i,j] !=0:
                sumi+=pow(mat[i,j]/pow(clo[j],2),2)
        matsum[i]=sumi
    for i in range(mat.get_shape()[0]):
        for j in range(i+1):
            sumu=0
            for c in range(mat.get_shape()[1]):
                if mat[i,c] != 0 and mat[j,c] != 0:
                    sumu+=(mat[i,c]/(clo[c]**2)*mat[j,c]/(clo[c]**2))
            if matsum[i]*matsum[j] !=0:
                cos[i][j]=sumu/pow((matsum[i]*matsum[j]),1/2)
                cos[j][i] = sumu / pow((matsum[i] * matsum[j]), 1 / 2)
    return cos
```

### 7.3. Basé sur les articles intégrant une mesure d'intermédiarité

Cette méthode prend en paramètre ;

- Mat : matrice binaire de notations (type : matrice sparse)
- Sim : matrice comprenant les mesures de similarité entre articles (type : matrice)
- K : nombre de voisins dans un voisinage (type : entier)
- Nb_rec : nombre de recommandations à produire dans une liste (type : entier)
- Bet : liste comprenant les mesures d'intermédiarité pour chaque article (type : liste)

Cette méthode renvoie une liste de recommandations pour chaque utilisateur, cette liste sera de longueur nb_rec. Pour chaque article, cette méthode choisit tout d'abord les k plus proches voisins de celui-ci grâce à la mesure de similarité entrée en paramètre. Ensuite, pour chaque utilisateur, elle remplit une liste comprenant tous les films similaires aux films visionnés par l'utilisateur. Les scores des films de cette liste seront calculés excepter pour les films visionnés par l'utilisateur d'intérêt, ces derniers auront le score de -1 afin d'éviter de les recommander. Enfin, les films recommandés seront ceux ayant obtenu le plus haut score. La mesure d'intermédiarité entrée en paramètre est utilisée lors du calcul du score.

```
def System_Itembased_Bet(mat,sim, k, nb_rec, bet):
# 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.get_shape()[0],mat.get_shape()[1]),dtyp
e=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.get_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 = []
        # tant que la liste n'atteint pas le nombre de k, on ajoute le ou
les voisins le plus proches
        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.get_shape()[0]):
        # récupère 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 film vu par l'utilisateur j
        for f in liste_films_vus:
            # récupère 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.get_shape()[1]):
        # 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[voisins[i][v],j]*sim[i][voisins[i][v]]*bet[voisins[i][v]]
```

```
)

denominator+=sim[i][int(voisins[i][v])]*bet[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
```

### 7.4. Basé sur les articles intégrant une mesure de proximité

Cette méthode prend en paramètre ;

- Mat : matrice binaire de notations (type : matrice sparse)
- Sim : matrice comprenant les mesures de similarité entre articles (type : matrice)
- K : nombre de voisins dans un voisinage (type : entier)
- Nb_rec : nombre de recommandations à produire dans une liste (type : entier)
- Clo : liste comprenant les mesures de proximité pour chaque article (type : liste)

Cette méthode renvoie une liste de recommandations pour chaque utilisateur, cette liste sera de longueur nb_rec. Pour chaque article, cette méthode choisit tout d'abord les k plus proches voisins de celui-ci grâce à la mesure de similarité entrée en paramètre. Ensuite, pour chaque utilisateur, elle remplit une liste comprenant tous les films similaires aux films visionnés par l'utilisateur. Les scores des films de cette liste seront calculés excepter pour les films visionnés par l'utilisateur d'intérêt, ces derniers auront le score de -1 afin d'éviter de les recommander. Enfin, les films recommandés seront ceux ayant obtenu le plus haut score. La mesure de proximité entrée en paramètre est utilisée lors du calcul du score.

```
def System_Itembased_Clo(mat,sim, k, nb_rec, clo):
# 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.get_shape()[0],mat.get_shape()[1]),dtyp
```

```
e=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.get_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 proches
        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.get_shape()[0]):
        # récupère 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 film vu par l'utilisateur j
        for f in liste_films_vus:
            # récupère 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.get_shape()[1]):
        # 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]]/clo[voisins[i][v]]
)
```

```
denominator+=sim[i][int(voisins[i][v])]/clo[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
```

## 8. Mesures d'évaluation

### 8.1. Mesures de précision

Cette méthode prend en paramètres ;

- Rank : liste reprenant les listes de recommandations de tous les utilisateurs (type : liste)
- Tmat : matrice de l'ensemble test (type : matrice sparse)

Elle renvoie la mesure de recall du système de recommandation. Tout d'abord, cette méthode extrait pour chaque utilisateur les films notés par celui-ci dans l'ensemble test. Elle parcourt ensuite cette liste de films notés et compte le nombre d'entre eux qui sont compris dans la liste de recommandations et qui n'y sont pas.

```
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.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)
```

Cette méthode prend en paramètres ;

- Rank : liste reprenant les listes de recommandations de tous les utilisateurs (type : liste)
- Tmat : matrice de l'ensemble test (type : matrice sparse)

Elle renvoie la mesure de précision du système de recommandation. Tout d'abord, cette méthode extrait pour chaque utilisateur les films notés par celui-ci dans l'ensemble test. Elle parcourt ensuite la liste de recommandations de l'utilisateur et compte le nombre de films présents dans la liste des films visionnés et le nombre qui ne sont pas repris dans cette liste.

```
def Precision(rank, tmat):
# renvoie la précision du système de recommandation
    # nombre d'articles recommandé et noté
    n_vp=0
    # nombre d'articles 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.Films_Vus(tmat, i)
        # pour chaque film noté par l'utilisateur
        for j in range(len(rank[i])):
            # noté ou non
            if rank[i][j] in liste_films_vus:
                n_vp+=1
            else:
                n_fp+=1
    # précision
    return (n_vp)/(n_vp+n_fp)
```

Cette méthode prend en paramètres ;

- Rank : liste reprenant les listes de recommandations de tous les utilisateurs (type : liste)
- Tmat : matrice de l'ensemble test (type : matrice sparse)

Pour chaque liste de recommandations, cette méthode calcule le dcg et le idcg. Tout d'abord, elle calcule le idcg en parcourant les notations de l'utilisateur présentes dans l'ensemble test. L'idcg est donc calculé en positionnant ces articles dans le haut de la liste. Ensuite, elle parcourt la liste de recommandations réelle de l'utilisateur et calcule le dcg en actualisant les articles de l'ensemble test avec leur position dans la liste. Enfin, le ndcg est le résultat de la division du dcg par le idcg. Elle renvoie la moyenne des ndcg pour toutes les listes de recommandations.

```
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 idéale
        pos=1
        liste_rec_t=[]
        # calcul du IDCG
        for j in range(tmat.get_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
        if idcg != 0:
            sumndcg+=(dcg/idcg)
    # Moyenne des nDCG
    return sumndcg/len(rank)
```

### 8.2. De nouveauté

Cette méthode prend en paramètres ;

- Rank : liste reprenant les listes de recommandations de tous les utilisateurs (type : liste)

Cette méthode renvoie la moyenne des mesures de popularité moyenne des articles de toutes les listes de recommandations. Tout d'abord, elle extrait une mesure de popularité qui est stockée dans le fichier json sous forme de liste. Ensuite, pour chaque liste de recommandations, elle calcule la popularité moyenne des articles présents dans cette liste.

```
def popularite(rank):
# renvoie la moyenne des popularités moyennes des articles présents dans
les listes de recommandations
    # somme des popularités moyennes
    sum_total=0
    # liste contenant la popularité des articles
    clo=[]
    # chargement des valeurs dans la liste de popularité
    with open("Closeness.json","r") as f:
        clo=json.load(f)
    # pour chaque liste de recommandation
    for i in range(len(rank)):
        # somme de la popularité dans la liste
        sum = 0
        count = 0
        # pour chaque article de la liste
        for j in range(len(rank[i])):
            sum+=clo[rank[i][j]]
            count+=1
        # moyenne de popularité sur la liste d'articles
        sum_total+=sum/count
```

```
    # moyenne des popularités moyennes
    return sum_total/len(rank)
```

Cette méthode prend en paramètres ;

- Rank : liste reprenant les listes de recommandations de tous les utilisateurs (type : liste)
- Sim_article : matrice comprenant les mesures de similarité entre articles (type : matrice)

Cette méthode renvoie la moyenne des mesures de distance d'un utilisateur de toutes les listes de recommandations. Tout d'abord, cette méthode extrait la matrice de notations binaires stockée dans un fichier json sous forme de matrice. Ensuite, pour chaque liste de recommandations, elle extrait la liste de tous les films visionnés par l'utilisateur, que ce soit dans l'ensemble d'entrainement ou dans l'ensemble test. Elle parcourt alors pour chaque film recommandé, la liste des films visionnés afin de calculer la dissimilarité entre l'article recommandé et l'article vu. Elle effectue la somme de ces dissimilarités qui sera ensuite divisée par la longueur de la liste des films visionnés multipliée par la longueur de la liste de recommandations. Elle renvoie la moyenne de ces mesures.

```
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.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)
```

### 8.3. De diversité

Cette méthode prend en paramètres ;

- Rank : liste reprenant les listes de recommandations de tous les utilisateurs (type : liste)
- Sim_article : matrice comprenant les mesures de similarité entre articles (type : matrice)

Cette méthode calcule la moyenne des mesures de dissimilarités intraliste de toutes les listes de recommandations. Pour chaque liste de recommandations, elle parcourt donc chaque paire d'articles présents dans cette liste et somme les dissimilarités entre chaque paire. Cette somme est ensuite divisée par la longueur de la liste de recommandations au carré. Elle renvoie la moyenne de cette mesure pour toutes les listes de recommandations.

```
def ILD(rank, sim_article):
    # somme des dissimilirités intraliste 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])))
    # moyenne des IDL
    return sum_total/len(rank)
```