Annexe 1 : Expérimentations

Plan de l'annexe 1

Introduction

Au travers de cette annexe, nous allons décrire la mise en application de notre méthodologie qui a été possible grâce à l'utilisation du langage de programmation Python, pour finalement obtenir les différents résultats présentés dans le chapitre 3. L'ensemble de notre code est disponible en **annexe 2** où les différentes parties sont séparées par des titres et des explications textuelles afin de permettre une meilleure clarté. Notre code a été réalisé en Python dans l'environnement Jupyter Notebook.

Avant d'entamer les explications concernant nos expérimentations, nous tenions à revenir sur les notions de méthode et de paramètre qui seront régulièrement employées tout au long de cette annexe. En programmation, une méthode est un code qui permet de réaliser une certaine action. Pour ce faire, elle peut avoir besoin de paramètres et peut également renvoyer un résultat. Prenons un exemple pour clarifier cela. À la Figure A1.1, nous avons codé une méthode très courte en Python permettant de réaliser une addition. Cette méthode se nomme « Addition » et prend deux paramètres identifiables entre les parenthèses : « num1 » et « num2 ». Les paramètres sont donc les éléments nécessaires à la réalisation de la méthode. La deuxième ligne de code correspond au code en lui-même de la méthode, celui qui réalise l'addition. Finalement, la somme est renvoyée par la méthode à la troisième ligne de code. La méthode « Addition » prend donc 2 nombres en paramètres, en effectue la somme et renvoie cette dernière.

```
def Addition (num1, num2):
    somme = num1 + num2
    return somme
```

Figure A1.1. Exemple de méthode en Python

Pour faire appel à une méthode, c'est-à-dire pour mettre en pratique l'application de celle-ci, il suffit simplement d'inscrire le nom de la méthode suivi des paramètres entre parenthèses. À la Figure A1.2, nous faisons appel à la méthode « Addition » créée précédemment et nous entrons comme paramètres « 3 » et « 7 ». La méthode renvoie alors 10 et nous affichons ce résultat.

```
print(Addition(3,7))
```

```
10
```

Figure A1.2. Appel de la méthode créée précédemment

À plusieurs reprises dans ce chapitre, nous décrirons des méthodes que nous avons créées dans notre code. Nous expliquerons alors les paramètres nécessaires pour le fonctionnement de la méthode et ce que celle-ci renvoie.

1. Analyse de la base de données

1.1. Choisir la base de données et créer des subdivisions

Comme cela a été expliqué dans la méthodologie, nous avons choisi une base de données MovieLens contenant 20 000 263 lignes où chacune d'elles comprend l'identifiant d'un utilisateur, celui d'un film, la note attribuée par cet utilisateur à ce film ainsi que le moment auquel cette note a été attribuée (« timestamp »). Nous souhaitons alors ordonner cette base de données en fonction du « timestamp » pour ensuite la diviser en *x* parties temporelles dans lesquelles nous supprimerions les données référant aux utilisateurs qui ne seraient pas présents dans toutes les divisions. Initialement, nous avions choisi une base de données de 100 000 notes mais nous nous sommes aperçus que celle-ci ne contenait pas suffisamment de données pour réaliser notre étude. En effet, une fois divisée en 5 parties en fonction du temps, cette base de données nous permettait d'avoir uniquement 23 utilisateurs présents dans chacune des parties. Nous avons estimé que ce nombre n'était pas assez important pour fournir des résultats fiables. Dès lors, nous avons testé d'autres base de données plus importantes jusqu'à choisir celle de plus de 20 000 000 de notes.

Les 20 000 263 lignes composant la base de données étaient recensées dans un fichier CSV. À travers le code situé en **annexe 2 aux points 1.1 et 1.2**, nous récupérons le contenu du fichier CSV pour ensuite l'ordonner en fonction de la caractéristique *timestamp*.

Une fois la base de données ordonnée dans le temps, nous souhaitions la diviser en *x* parties équivalentes, plusieurs valeurs de ce paramètre de division *x* ont alors été testées : 5, 10 et 20 (cf. **annexe 2 point 1.3**).

À la suite de cela, l'objectif était de conserver uniquement les données concernant les utilisateurs présents dans toutes les parties. Pour ce faire, nous avons créé la méthode « Utilisateurs » (cf. **annexe 2 point 1.4.1**) qui permet de renvoyer une liste contenant les identifiants de tous les utilisateurs présents dans la base de données fournie en paramètre de la méthode. Grâce à cette méthode, nous étions en mesure d'identifier les utilisateurs présents dans chacune des divisions de la base de données pour ainsi établir une liste des utilisateurs présents à la fois dans toutes les parties. Ensuite, dans chaque division de la base de données, nous avons conservé uniquement les lignes concernant les utilisateurs présents dans toutes les divisions (cf. **annexe 2 point 1.4.2**).

Comme nous venons de le mentionner, plusieurs valeurs de *x* ont été testées : 5, 10 et 20. En choisissant de diviser la base de données en 5 parties de temps, nous pouvions recenser 335 utilisateurs qui étaient présents dans chacune des parties. Malheureusement, en la divisant en 10 ou 20 parties, nous avons constaté qu'aucun utilisateur n'était présent dans chacune des subdivisions. Dès lors, nous avons opté pour une division de la base de données en cinq tranches temporelles.

1.2. Enregistrer les données

Ensuite, nous avons choisi d'enregistrer ces informations dans des fichiers CSV afin de pouvoir les récupérer plus facilement par la suite sans devoir exécuter à nouveau tout le travail sur la base de données initiale, celui-ci étant assez long au vu de la taille de cette base de données. Dès lors, nous avons créé cinq fichiers CSV dans lesquels nous avons inscrit les lignes de données correspondant à chacune des cinq divisions de la base de données initiale. Évidemment, uniquement les données faisant référence aux utilisateurs présents dans toutes les parties ont été inscrites (cf. **annexe 2 point 1.5**). De cette manière, les différentes divisions de la base de données contiennent respectivement : 74 968, 80 980, 84 518, 55 186 et 46 393 lignes. Nous recensons alors 335 utilisateurs et 15 738 films.

1.3. Préparer l'exploitation des données par les algorithmes

Finalement, nous créons trois méthodes nous permettant d'obtenir des informations nécessaires contenues dans les fichiers CSV que nous venons d'enregistrer. La première, « Lists_Users_And_Movies », permet de récupérer les identifiants de tous les utilisateurs et de les stocker dans une liste ainsi que ceux de tous les films et de les stocker dans une autre liste (cf. **annexe 2 point 1.6.1**). Cette méthode pourra être utile par la suite afin de connaître l'identifiant d'un certain utilisateur ou celui d'un certain film présents dans la matrice de données.

La méthode « Matrix_Ratings » est également créée afin de récupérer le contenu de ce qui avait été stocké dans les fichiers CSV précédemment (cf. **annexe 2 point 1.6.2**). Dans cette même méthode, nous créons des matrices binaires, c'est-à-dire composées uniquement de 0 et de 1. Pour chacun des fichiers CSV récupérés, nous créons une matrice dont chaque ligne correspond à un des 335 utilisateurs et chaque colonne à un des 15 738 films. Le contenu de cette matrice vaut alors 0 partout initialement. Nous complétons ensuite celle-ci de la façon

suivante : lorsqu'un utilisateur a vu un certain film, c'est-à-dire qu'il lui a attribué une note stockée dans le fichier CSV, une valeur de 1 est indiquée à l'intersection de cet utilisateur et de ce film dans la matrice. Grâce à cette méthode, nous créons alors 5 matrices contenant chacune les données provenant d'une division temporelle de la base de données initiale. L'ensemble de ces matrices est stocké dans une variable nommée « matrix_all ».

La troisième méthode, « Films_Consultes », (cf. **annexe 2 point 1.6.3**) permet de récupérer une liste de tous les films qu'un certain utilisateur a vus. Celle-ci nous sera utile par la suite car, dans les différents systèmes de recommandations, nous tâcherons de ne pas recommander à l'utilisateur des films qu'il a déjà vus. Si un film est présent dans la liste renvoyée par cette méthode, il ne sera forcément pas recommandé à l'utilisateur.

2. Réalisation de plusieurs algorithmes

Dans cette partie, nous allons décrire comment nous avons réalisé chacun des algorithmes étudiés ainsi que les différentes décisions que nous avons prises pour chacun d'eux.

2.1. Filtrage collaboratif item-based avec similarité cosinus

Établir des recommandations selon cet algorithme

Pour proposer des recommandations aux utilisateurs sur base d'un algorithme item-based, nous avions tout d'abord besoin de connaître les similarités entre toutes paires d'items. C'est pourquoi, nous avons codé une première méthode, « Sim_Cosine_Items », réalisant une matrice de similarité cosinus entre les items (cf. **annexe 2 point 2.1.1**).

Ensuite, nous déterminons les *k* plus proches voisins de chaque film, dans la méthode « Voisins_Items » (cf. **annexe 2 point 2.1.2**), c'est-à-dire ceux qui ont la plus grande similarité avec le film considéré. Plusieurs valeurs de *k* ont été testées mais nous aborderons ceci dans la description des tests.

Troisièmement, nous avons établi la méthode « Recommandations_ItemBased » (cf. **annexe 2 point 2.1.3**). Celle-ci permet alors d'établir une liste de recommandations pour chaque utilisateur selon notre algorithme décrit dans la méthodologie. Le nombre de recommandations à proposer pour chacun des utilisateurs est fourni en paramètre, celui-ci sera déterminé par la suite.

Tests, réflexions et prises de décisions

À travers le code situé à la Figure A1.3, nous avons pu tester notre algorithme item-based. Étant donné que celui-ci doit être capable de recommander jusqu'à 100 items, nous avons choisi d'effectuer nos tests dans l'hypothèse où nous devrions fournir 100 recommandations à chaque utilisateur. Ces tests ont été réalisés en prenant comme matrice de données la première subdivision de la base de données initiale car il s'agit de celle qui contiendra le moins de données par rapport à toutes les matrices utilisées au cours de notre étude.

```
### Test : Application à une base de données

# Reprendre la base de données de la première période temporelle
matrice = matrix_all[0]

# Nombre de recommandations qui doivent être établies pour chaque utilisateur
nb_recom = 100

# Nombre de voisins pour chaque film
# Nous décidons de fixer le nombre de voisins pour chaque film équivalant au nombre de recommandations
# Pour être certains d'avoir suffisamment de voisins pour constituer la liste de recommandations
k = nb_recom

# Créer les recommandations
list_recommandations_itembased = Recommandations_ItemBased(matrice, nb_recom, k)

# Imprimer les recommandations
for u in range(len(list_recommandations_itembased)):
    print("Liste de recommandations utilisateur "+str(u)+" :")
    print(list_recommandations_itembased[u])
```

Figure A1.3. Code pour appliquer l'algorithme item-based

Plusieurs valeurs de *k* ont été testées car il est nécessaire d'avoir suffisamment de voisins de telle sorte que l'on puisse fournir le nombre de recommandations demandées à chacun des utilisateurs. Toutefois, déterminer un nombre trop important de voisins entraînerait un temps computationnel plus conséquent et ne serait pas utile pour notre étude étant donné que nous avons besoin d'un nombre limité de recommandations. En effet, une fois que la liste de recommandations a atteint le nombre de films demandé, l'algorithme arrête de parcourir les autres voisins étant donné qu'il ne peut plus rien ajouter à la liste.

Pour que notre algorithme soit capable de fournir une liste de 100 recommandations à chacun des utilisateurs, nous avons constaté qu'il était nécessaire de disposer d'au moins 100 voisins. Ceci peut s'expliquer par le fait qu'il est possible que certains utilisateurs n'aient vu qu'un seul film lors de la première période de temps. Dès lors, l'algorithme peut uniquement leur recommander les voisins de cet unique film vu. C'est pourquoi nous avons choisi de fixer la valeur de *k*, le nombre de voisins pour chaque film, égale au nombre de recommandations demandées. Ceci peut être identifié dans le code (cf. Figure A1.3), à travers la ligne « k = nb_recom ».

2.2. Filtrage collaboratif user-based avec similarité cosinus

Établir des recommandations selon cet algorithme

Pour mettre en place cet algorithme user-based, nous avons réalisé une première méthode, « Sim_Cosine_Users » (cf. **annexe 2 point 2.2.1**), permettant de calculer la matrice de similarités entre toutes paires d'utilisateurs selon la formule de similarité cosinus.

Selon les similarités calculées, nous déterminons, dans la méthode « Voisins_Users » (cf. **annexe 2 point 2.2.2**), les *k* plus proches voisins de chaque utilisateur. De la même manière que pour l'algorithme précédent, une réflexion par rapport à la valeur de *k* sera abordée par la suite.

Ensuite, dans la méthode « Recommandations_UserBased » (cf. **annexe 2 point 2.2.3**), nous recommandons à chaque utilisateur une liste de films en fonction des films vus par les voisins de l'utilisateur. De plus amples explications sur la façon dont l'algorithme établit les recommandations ont été présentées dans la méthodologie.

Tests, réflexions et prises de décisions

Nous avons testé notre algorithme à travers le code présenté à la Figure A1.4. Comme pour l'algorithme précédent, nous avons évalué plusieurs valeurs de *k* en reprenant la base de données correspondant à la première période de temps et en supposant que 100 recommandations devaient être fournies pour chaque utilisateur.

Nous avons remarqué que fixer la valeur de *k* à 15 permettait de fournir 100 recommandations à chaque utilisateur. Même si seulement 10 ou 20 recommandations doivent être établies, nous sommes certains que l'algorithme sera capable de fournir le nombre de recommandations demandées étant donné que cela fonctionne pour 100 recommandations.

```
### Test : Application à une base de données

# Reprendre la base de données de la première période temporelle
matrice = matrix_all[0]

# Nombre de recommandations qui doivent être établies pour chaque utilisateur
nb_recom = 100

# Nombre de voisins
k=15

# Créer les recommandations
list_recommandations_userbased = Recommandations_UserBased(matrice, nb_recom, k)

# Imprimer les recommandations
for u in range(len(list_recommandations_userbased)):
    print("Liste de recommandations utilisateur "+str(u)+" :")
    print(list_recommandations_userbased[u])
```

Figure A1.4. Code pour appliquer l'algorithme user-based

En comparaison avec l'algorithme item-based, nous constatons que le nombre de voisins nécessaire est plus faible pour l'algorithme user-based. Cela peut s'expliquer par le fait qu'ici, même si un utilisateur n'a vu qu'un seul film, l'algorithme va reprendre tous les films vus par les voisins de l'utilisateur donc ça ne pose pas de problème que celui-ci n'ait vu qu'un seul film. Alors que pour l'item-based, l'algorithme ne pouvait reprendre que les films voisins de l'unique film vu par l'utilisateur. Il était alors nécessaire d'avoir une valeur de *k* équivalente au nombre de recommandations demandées, ce qui n'est pas le cas ici.

2.3. Algorithme de filtrage collaboratif intégrant une plus grande diversité au sein des recommandations

Établir des recommandations selon cet algorithme

Afin de réaliser cet algorithme, il est nécessaire de connaître la diversité présente au sein d'un ensemble d'items. Dans ce but, nous avons réalisé la méthode « Diversity » prenant en paramètre un ensemble d'items et renvoyant la diversité calculée entre tous ces éléments (cf. **annexe 2 point 2.3.1**). Étant donné qu'il est nécessaire de connaître la similarité entre toutes paires d'items afin de pouvoir calculer la diversité, la méthode prend également en paramètre une matrice de similarités entre les items. Nous utiliserons la même matrice de similarités que celle calculée dans le premier algorithme, c'est-à-dire une matrice de similarités cosinus entre les items.

Ensuite, nous créons une deuxième méthode, « Best_Diverse_Item » (cf. **annexe 2 point 2.3.2**) qui permet de trouver l'item apportant la plus grande diversité, par rapport à d'autres items potentiels, lorsqu'on l'ajoute à une liste déjà composée d'items à recommander. Pour cela, elle prend notamment deux listes en paramètres : la liste des recommandations déjà établies, ainsi que la liste contenant l'ensemble des items que l'on peut potentiellement ajouter aux recommandations. Pour chacun des éléments composant la liste de recommandations potentielles, la méthode va calculer la diversité (en faisant appel à la méthode précédente) si on ajoutait cet élément à la liste de recommandations actuelles. Ensuite, l'item permettant d'offrir la plus grande diversité avec la liste de recommandations actuelles est renvoyé.

Notre troisième méthode concernant cet algorithme établit les recommandations pour chacun des utilisateurs, il s'agit de la méthode « Recommandations_Diversifiees » (cf. **annexe 2 point 2.3.3**). La façon dont cet algorithme établit les recommandations pour chaque utilisateur a été décrite dans le chapitre concernant la méthodologie.

Tests, réflexions et prises de décisions

Le code présenté à la Figure A1.5 nous a permis de tester notre algorithme.

```
### Test : Application à une base de données

# Reprendre la base de données de la première période temporelle
matrice = matrix_all[0]

# Nombre de recommandations qui doivent être fournies pour chaque utilisateur
nb_recom = 100

# Nombre de voisins
k=15

# Créer les recommandations
list_recommandations_diverses = Recommandations_Diversifiees(matrice, nb_recom, k)

# Imprimer les recommandations
for u in range(len(list_recommandations_diverses)):
    print("Liste de recommandations utilisateur "+str(u)+" :")
    print(list_recommandations_diverses[u])
```

Figure A1.5. Code pour appliquer l'algorithme introduisant plus de diversité

Pour rappel, afin d'établir 100 recommandations pour chaque utilisateur, notre algorithme constitue préalablement une liste composée du quintuple d'éléments, à savoir 500

recommandations et sélectionne alors parmi ces recommandations potentielles celles offrant une plus grande diversité. Toutefois, après avoir réalisé des tests, nous nous sommes rendu compte que le temps computationnel était beaucoup trop important lorsque nous voulions avoir 100 recommandations finales à proposer à chaque utilisateur à partir d'une liste de 500 recommandations potentielles.

En effet, l'algorithme prenait environ 1 minute pour déterminer la liste de recommandations finales pour chaque utilisateur. Sachant que notre base de données est composée de 335 utilisateurs, il aurait fallu plus de 5h30 pour que notre algorithme soit capable de fournir 100 recommandations pour chaque utilisateur. Cela peut s'expliquer par le fait que, pour ajouter un élément à la liste de recommandations finales, l'algorithme calcule la diversité de chacun des éléments de la liste potentielle avec l'ensemble des éléments déjà présents dans la liste finale. Dès lors, plus la liste de recommandations potentielles est importante, plus le nombre d'éléments à parcourir à chaque étape est important également.

Nous avons alors pris la décision de plafonner la liste de recommandations potentielles à 200 éléments maximum. Effectivement, environ 1h30 a été nécessaire pour que l'algorithme puisse fournir 100 recommandations finales à chaque utilisateur, à partir d'une liste de 200 recommandations potentielles. Nous estimons qu'au-delà, le temps d'attente pour obtenir les résultats serait trop important. Dès lors la liste de recommandations potentielles sera composée du quintuple du nombre de recommandations à fournir avec un plafond à 200 éléments.

Étant donné que nous faisons appel à l'algorithme user-based dans un premier temps afin de fournir une liste de recommandations de base, il est nécessaire de déterminer un nombre de voisins, *k*, à appliquer pour l'appel de cet algorithme. De la même manière que pour les deux algorithmes précédents, plusieurs valeurs de *k* ont été testées et nous nous sommes aperçus que fixer $k = 15$ permettait de fournir une liste de recommandations potentielles constituées de 200 éléments.

2.4. Algorithme de filtrage collaboratif intégrant une plus grande nouveauté dans les recommandations

Établir des recommandations selon cet algorithme

Afin de pouvoir recommander des films ayant un degré de nouveauté plus important, il était tout d'abord nécessaire de connaître la popularité de chacun des films. Dès lors, nous avons réalisé la méthode « Vues_Films » (cf. **annexe 2 point 2.4.1**) qui permet de compter, pour chaque film, le nombre d'utilisateurs ayant vu ce film, ce qui traduit la popularité de celui-ci.

Deuxièmement, la méthode « Degre_Nouveaute » (cf. **annexe 2 point 2.4.2**) a été créée. Celle-ci calcule le degré de nouveauté de tous les films de la base de données. Pour cela, nous faisons appel à la méthode précédente afin de déterminer le nombre d'utilisateurs ayant vu chaque film. Ensuite, nous divisons ce nombre par le nombre d'utilisateurs repris dans la base de données afin de connaître le degré de popularité. Après cela, nous calculons le degré de nouveauté de tous les films sur base de leur degré de popularité. La méthode renvoie alors une liste comprenant le degré de nouveauté de tous les films de la base de données.

Ensuite, nous avons établi la méthode « Recommandations_Nouvelles » (cf. **annexe 2 point 2.4.3**) afin de constituer, pour chaque utilisateur, une liste de recommandations intégrant plus de nouveauté. Pour cela, nous établissons tout d'abord une liste de recommandations potentielles, composée du quintuple du nombre de recommandations demandées, en faisant appel à la méthode « Recommandations_UserBased » (cf. **annexe 2 point 2.2.3**). Après cela, nous ordonnons les films par degré de nouveauté décroissant, les degrés de nouveauté étant connus grâce à l'appel à la méthode « Degre_Nouveaute » (cf. **annexe 2 point 2.4.2**). Finalement, nous récupérons le nombre de recommandations demandées parmi les premiers éléments de la liste ordonnée afin de constituer la liste de recommandations. L'ensemble des listes de recommandations est alors renvoyé par la méthode.

Tests, réflexions et prises de décisions

Contrairement à l'algorithme précédent, ici, le temps computationnel nécessaire pour déterminer une liste de recommandations finales pour chaque utilisateur est plus faible. En effet, à partir de la liste de recommandations potentielles, nous ajoutons en une seule fois, à la liste de recommandations finales, l'ensemble des items les plus nouveaux de manière à

atteindre le nombre de recommandations demandées. Alors que pour la réalisation de l'algorithme précédent, nous devions ajouter un à un chacun des éléments à la liste de recommandations finales et qui plus est, nous devions à chaque fois regarder l'ensemble des recommandations potentielles restantes afin de sélectionner celle qui permettait d'offrir la plus grande diversité avec la liste de recommandations finales jusqu'à présent constituée.

Dès lors, nous sommes en mesure de conserver notre idée de base présentée dans la méthodologie, à savoir de constituer une liste de recommandations potentielles dont le nombre correspond au quintuple du nombre de recommandations finales à fournir, peu importe la taille de celui-ci.

Nous avons pu tester notre algorithme à travers le code présenté à la Figure A1.6. Étant donné que nous faisons appel à l'algorithme user-based afin de constituer une liste de recommandations potentielles, il était nécessaire de déterminer le nombre de voisins, *k*, repris comme paramètre de cet algorithme. Plusieurs valeurs ont été testées et nous nous sommes rendus compte qu'une valeur de $k = 15$ permettait de fournir 100 recommandations finales à chaque utilisateur, c'est-à-dire à partir d'une liste de 500 recommandations potentielles.

```
### Test : Application à une base de données

# Reprendre la base de données de la première période temporelle
matrice = matrix_all[0]

# Nombre de recommandations qui doivent être établies pour chaque utilisateur
nb_recom = 100

# Nombre de voisins
k=15

# Créer les recommandations
list_recommandations_nouvelles = Recommandations_Nouvelles(matrice, nb_recom, k)

# Imprimer les recommandations
for u in range(len(list_recommandations_nouvelles)):
    print("Liste de recommandations utilisateur "+str(u)+" :")
    print(list_recommandations_nouvelles[u])
```

Figure A1.6. Code pour appliquer l'algorithme introduisant plus de nouveauté

2.5. Algorithme fournissant des recommandations aléatoires

Cet algorithme étant plus simple à mettre en place que les précédents, nous n'avons eu besoin de réaliser qu'une seule méthode (cf. **annexe 2 point 2.5**) : « Recommandations_Aleatoires ». Celle-ci permet de recommander aux utilisateurs des films sélectionnés aléatoirement parmi les 15 738 films composant notre base de données. Pour chaque utilisateur, nous sélectionnons des films aléatoirement parmi tous les films que l'utilisateur n'a pas encore vus jusqu'à ce que la liste des recommandations soit remplie, c'est-à-dire que le nombre de recommandations demandées soit atteint.

Le code qui nous a servi à tester notre algorithme sur la matrice contenant les données de la première division temporelle est le suivant (cf. Figure A1.7). Cela nous a permis de vérifier que notre algorithme fournissait bien des listes de recommandations pour chaque utilisateur et d'afficher ces listes de recommandations.

```
### Test : Application à une base de données

# Reprendre la base de données de la première période temporelle
matrice = matrix_all[0]

# Nombre de recommandations qui doivent être établies pour chaque utilisateur
nb_recom = 100

# Créer les recommandations
list_recommandations_aleatoires = Recommandations_Aleatoires(matrice, nb_recom)

# Imprimer les recommandations
for u in range(len(list_recommandations_aleatoires)):
    print("Liste de recommandations utilisateur "+str(u)+" :")
    print(list_recommandations_aleatoires[u])
```

Figure A1.7. Code pour appliquer l'algorithme aléatoire

2.6. Algorithme recommandant les films populaires

Lorsque nous parlons de films populaires, nous faisons référence aux films qui ont été vus par le plus grand nombre d'utilisateurs de notre base de données. Pour cet algorithme, nous n'avons également réalisé qu'une seule méthode : « Recommandations_Populaires » (cf. **annexe 2 point 2.6.1**). Dès le début, celle-ci fait appel à la méthode « Vues_Films » créée précédemment (cf. **annexe 2 point 2.4.1**), qui permet de connaître le nombre d'utilisateurs ayant vu chacun des films à la période de temps considérée. Grâce à cela, il nous sera possible d'identifier les films les plus populaires à recommander.

Ensuite, les films sont ordonnés en fonction de leur popularité et sont ajoutés un à un à la liste de recommandations par ordre de popularité décroissante en veillant à ne pas recommander à l'utilisateur un film qu'il a déjà vu. Les films populaires sont donc ajoutés à la liste de recommandations finales jusqu'à ce que celle-ci ait atteint le nombre de recommandations demandées.

De la même manière que pour les algorithmes précédents, nous avons testé notre algorithme avec le code suivant (cf. Figure A1.8) en nous basant sur l'hypothèse que nous devrions fournir 100 recommandations à chaque utilisateur à partir des données reprises dans la matrice correspondant à la première division temporelle de notre base de données initiale.

```
### Test : Application à une base de données

# Reprendre la base de données de la première période temporelle
matrice = matrix_all[0]

# Nombre de recommandations qui doivent être établies pour chaque utilisateur
nb_recom = 100

# Créer les recommandations
list_recommandations_populaires = Recommandations_Populaires(matrice, nb_recom, 0)

# Imprimer les recommandations
for u in range(len(list_recommandations_populaires)):
    print("Liste de recommandations utilisateur "+str(u)+" :")
    print(list_recommandations_populaires[u])
```

Figure A1.8. Code pour appliquer l'algorithme populaire

2.7. Algorithme de filtrage collaboratif avec factorisation matricielle

Déterminer les facteurs latents pour chaque utilisateur et pour chaque film

L'objectif de cet algorithme est de représenter chaque utilisateur et chaque film par une série de facteurs latents. Nous avons alors créé la méthode « Facteurs_Latents » (cf. **annexe 2 point 2.7.1**) qui établit, sur base d'un processus itératif, les facteurs latents de chaque utilisateur et ceux de chaque film. Cette méthode requiert 5 paramètres qui sont les suivants :

- La matrice de données sur laquelle l'algorithme se base pour constituer les facteurs latents. Les lignes correspondent aux utilisateurs, les colonnes aux films et une valeur de 1 est inscrite lorsqu'un utilisateur a vu un certain film, 0 sinon.
- Le nombre de facteurs latents à établir pour chaque utilisateur et pour chaque film.

- Le nombre d'itérations qui devront être réalisées.
- Le taux d'apprentissage (η), nécessaire pour l'actualisation des facteurs latents à chaque itération.
- Le paramètre λ, également nécessaire pour l'actualisation des facteurs latents à chaque itération.

Dans notre méthode, nous attribuons tout d'abord des valeurs aléatoires à tous les facteurs latents. Ensuite, nous entamons le processus itératif permettant de faire évoluer ces valeurs. À chaque itération, nous parcourons toutes les paires utilisateur-film pour lesquelles une valeur non nulle est inscrite dans la matrice de données. Pour chacune de ces paires, nous estimons alors la valeur prédite par l'algorithme. Ensuite, nous comparons cette valeur prédite avec celle réellement observée dans la matrice de données et nous constituons alors le terme d'erreur pour cette paire utilisateur u – film i. Toujours dans la même itération, nous actualisons les valeurs de chaque facteur latent. Les différentes formules, appliquées pour réaliser ceci, ont été présentées dans la méthodologie. Finalement, notre méthode renvoie deux matrices : celle regroupant les facteurs latents de tous les utilisateurs et celle reprenant les facteurs latents de tous les films.

Établir les recommandations

Disposant des facteurs latents caractérisant chaque utilisateur et chaque film, il nous était alors possible d'établir les listes de recommandations pour tous les utilisateurs. Pour ce faire, nous avons créé la méthode « Recommandations_Factorisation » (cf. **annexe 2 point 2.7.2**). Cette méthode requiert trois paramètres : la matrice contenant les facteurs latents de tous les utilisateurs (i.e. les lignes correspondent aux utilisateurs et les colonnes aux facteurs latents) ; la matrice contenant les facteurs latents de tous les items (i.e. les lignes correspondent aux items et les colonnes aux facteurs latents) ; le nombre de recommandations à fournir pour chaque utilisateur.

Tout d'abord, nous parcourons chaque utilisateur et estimons la valeur prédite pour chaque film de la base de données. Ensuite, toujours pour le même utilisateur, nous ordonnons les films par ordre de valeur prédite décroissante. Nous ajoutons alors ces films un à un à la liste de recommandations de l'utilisateur en veillant que ce dernier n'ait pas déjà vu le film. Finalement, la méthode renvoie les listes de recommandations de tous les utilisateurs.

Remise en question de l'application de cet algorithme

Lors de la réalisation de cet algorithme, nous avons fait face à une difficulté majeure remettant en cause l'application de celui-ci. En effet, nous avons testé notre algorithme à travers le code présenté à la Figure A1.9.

```
### Test : Application à une base de données

# Reprendre la base de données de la première période temporelle
matrice = matrix_all[0]

# Nombre de recommandations qui doivent être établies pour chaque utilisateur
nb_recom = 100

# Déterminer les facteurs
fact_users, fact_items = Facteurs_Latents (matrice, 15, 200, 0.005, 0.1)

# Créer les recommandations
list_recommandations_factorisation = Recommandations_Factorisation(fact_users, fact_items , nb_recom)

# Imprimer les recommandations
for u in range(len(list_recommandations_factorisation)):
    print("Liste de recommandations utilisateur "+str(u)+" :")
    print(list_recommandations_factorisation[u])
```

Figure A1.9. Code pour appliquer l'algorithme avec factorisation matricielle

En affichant les recommandations établies pour tous les utilisateurs (cf. Figure A1.10), nous avons réalisé que pratiquement les mêmes recommandations étaient faites à tous les utilisateurs, à la seule différence que les films déjà vus par un utilisateur n'étaient pas dans sa liste. En dehors de ça, les mêmes recommandations étaient proposées aux utilisateurs.

```
Liste de recommandations utilisateur 0 :
[661, 622, 1609, 889, 1302, 2173, 894, 2002, 1265, 1658, 2118, 2058, 2083, 1250, 1303, 2003, 1298,
588, 751, 1244, 1469, 2244, 1783, 1452, 2007, 1280, 1760, 1349, 636, 1804, 1890, 1343, 1466, 2144,
1843, 1770, 1781, 1885, 1761, 1380, 1559, 2226, 1104, 2019, 650, 1782, 815, 1692, 2255, 2184, 215
6, 1812, 1936, 730, 1477, 1230, 1968, 1854, 1450, 1520, 2097, 1275, 2188, 1290, 2065, 631, 1031, 2
032, 1891, 911, 1247, 1617, 664, 2008, 1268, 839, 517, 1697, 810, 2304, 1248, 1424, 1278, 2210, 12
08, 2182, 1338, 747, 1965, 1229, 781, 1543, 2034, 678, 2178, 1228, 835, 2025, 786, 637]
Liste de recommandations utilisateur 1 :
[661, 1609, 2010, 1302, 1668, 2173, 894, 2002, 1265, 2118, 1963, 2058, 2083, 1250, 1303, 1464, 200
3, 1298, 588, 1244, 751, 1469, 2244, 2005, 1783, 1465, 1452, 1893, 2007, 1579, 1280, 1760, 1349, 6
36, 1804, 1890, 1648, 1343, 802, 1466, 2144, 1843, 1770, 1781, 909, 1885, 1951, 1761, 1380, 1559,
1375, 2226, 1689, 2019, 1062, 650, 350, 1782, 787, 815, 1886, 1664, 1692, 1327, 1271, 2255, 2184,
2156, 1812, 1196, 1936, 730, 1477, 1242, 1230, 1968, 1854, 1183, 1520, 2097, 1275, 2077, 1454, 91
8, 521, 2188, 1290, 2065, 631, 1337, 2032, 1476, 1891, 911, 1247, 1617, 744, 664, 2008, 1268]
Liste de recommandations utilisateur 2 :
[661, 1609, 889, 2010, 1302, 1668, 2173, 894, 2002, 1265, 1658, 2118, 1963, 2058, 2083, 1250, 130
3, 1464, 2003, 1298, 588, 751, 1244, 1469, 2244, 2005, 1783, 1465, 1452, 1893, 2007, 1579, 1280, 1
760, 1349, 636, 1804, 1890, 1648, 1343, 802, 1466, 2144, 1843, 1770, 1781, 909, 1951, 1761, 1380,
1375, 1559, 2226, 1104, 2019, 650, 350, 1782, 787, 1886, 815, 1664, 1692, 872, 1327, 1271, 2255, 2
```

Figure A1.10. Partie des résultats fournis par le code situé en Figure A1.9

Dès lors, nous avons cherché d'où cela pouvait provenir et nous avons fini par réaliser que les vecteurs reprenant les facteurs latents, que ce soit ceux des utilisateurs ou ceux des films, sont

tous pratiquement identiques. Dans la Figure A1.11, nous affichons le vecteur reprenant les facteurs latents de l'utilisateur 0, celui de l'utilisateur 52, celui du film 21 et celui du film 162. Nous avons sélectionné ces utilisateurs et ces films de manière aléatoire, le même test a été réalisé avec d'autres utilisateurs et films également. Nous pouvons constater que le premier facteur de chacun des vecteurs est pratiquement identique pour tous les vecteurs. Il en va de même pour les autres facteurs de chacun des vecteurs.

```
print("Utilisateur "+str(0))
print(fact_users[0,:])
print("Utilisateur "+str(52))
print(fact_users[52,:])
print("Film "+str(21))
print(fact_items[21,:])
print("Film "+str(162))
print(fact_items[162,:])
```

```
Utilisateur 0
[0.27463806 0.24733699 0.22701373 0.22764069 0.27753942 0.24501019
 0.24531082 0.22650905 0.25946275 0.24111785 0.21401244 0.26718806
 0.23517    0.21592873 0.28027589]
Utilisateur 52
[0.27410503 0.24703847 0.22662711 0.22759302 0.27736707 0.2446162
 0.24497843 0.22615172 0.25888946 0.24100461 0.21369362 0.26681671
 0.23481392 0.21557139 0.27989512]
Film 21
[0.27145844 0.24462844 0.2244227  0.22545198 0.27461097 0.24228883
 0.24261946 0.22398363 0.25645432 0.23862882 0.21166607 0.26429824
 0.23249124 0.21361123 0.27716441]
Film 162
[0.27190173 0.24426113 0.22452108 0.22549771 0.27464095 0.2428686
 0.24233367 0.22383133 0.25720575 0.23842728 0.21189693 0.26450354
 0.23280529 0.21403846 0.27767879]
```

Figure A1.11. Affichage de plusieurs vecteurs reprenant des facteurs latents

Donc, il est logique que si les vecteurs sont pratiquement tous identiques, les résultats fournis soient également identiques pour tous les utilisateurs. Cela peut probablement s'expliquer par le fait que nous utilisons une matrice de données contenant uniquement des valeurs binaires (1 si l'utilisateur a vu le film, 0 sinon). En effet, après avoir fait plusieurs recherches dans la littérature, nous avons constaté que cet algorithme était toujours utilisé sur une matrice contenant des données variant sur une certaine échelle, par exemple de 0 à 5, et non pas sur une matrice uniquement binaire (Koren, 2010 ; Nguyen & Zhu, 2012 ; Paterek, 2007 ; Yuan, Luo & Shang, 2018 ; Yu et al., 2014).

Dès lors, nous avons choisi de ne pas étudier cet algorithme dans la suite de nos recherches.

3. Métriques évaluant les recommandations

Comme cela a été présenté dans la méthodologie, nous avons choisi de nous concentrer sur deux métriques permettant d'étudier respectivement la diversité et la nouveauté au sein des recommandations établies pour les utilisateurs. Nous allons donc, dès à présent, décrire les méthodes qui nous ont permis de réaliser le calcul de ces métriques ainsi que les différents choix que nous avons élaborés.

3.1. La diversité

Calcul de la diversité au sein des recommandations

Avant d'entamer les explications concernant cette métrique, nous tenions à rappeler la formule qui permet de calculer la diversité présente au sein d'un ensemble d'éléments (8)

$$Diversity(i_1, \dots, i_n) = \frac{\sum_{j=1..n-1} \sum_{k=j+1..n} \left(1 - sim(i_j, i_k)\right)}{\frac{n}{2} * (n-1)} \quad (8)$$

Où *Diversity(i_1, …, i_n)* signifie la diversité de l'ensemble contenant les items i_1 à i_n ;

n correspond au nombre d'items composant l'ensemble dont nous souhaitons calculer la diversité ;

sim(i_j,i_k) représente la similarité entre l'item i_j et l'item i_k.

Pour réaliser le calcul de cette métrique de diversité, nous avons choisi d'utiliser les similarités cosinus entre les items calculées à partir des données reprises dans la base de données initiale. Pour rappel, nous avions divisé au départ notre base de données en 5 matrices reprenant chacune les données relatives à une période temporelle. À présent, nous souhaitons disposer de l'ensemble de ces données. Dès lors, nous avons créé la méthode « Rassemblement_Matrices » (cf. **annexe 2 point 3.1.1**) permettant de rassembler en une seule matrice l'ensemble des matrices de divisions temporelles de notre base de données initiale. La matrice renvoyée par cette méthode est donc une matrice binaire où les lignes correspondent aux utilisateurs, les colonnes aux films et où une valeur de 1 est indiquée lorsque l'utilisateur a vu le film considéré, 0 sinon.

Ensuite, nous avons appliqué cette méthode à nos données (cf. **annexe 2 point 3.1.2**) et nous avons ainsi créé la matrice « matrice_init » reprenant l'ensemble des données, toutes divisions temporelles confondues. Cela nous permettait alors de calculer les similarités cosinus entre toutes paires d'items. Pour cela, nous avons fait appel à la méthode « Sim_Cosine_Items » (cf. **annexe 2 point 2.1.1**) créée précédemment.

Disposant des similarités entre les items, il nous était alors possible de créer la méthode « Diversité_Moyenne » (cf. **annexe 2 point 3.1.3**) qui permet, comme son nom l'indique, de calculer la diversité moyenne à partir des diversités obtenues pour différentes listes de recommandations. Cette méthode parcourt tout d'abord chaque liste de recommandations et en calcule la diversité en faisant appel à la méthode « Diversity » créée précédemment (cf. **annexe 2 point 2.3.1**). Une fois la diversité de chaque liste de recommandations connue, il est possible de calculer la moyenne de ces diversités, c'est cette valeur qui est renvoyée par notre méthode. La valeur obtenue sur cette métrique de diversité pourra varier entre 0 et 1.

Calcul de la diversité globale pour chaque période

Comme cela a été expliqué dans la méthodologie, nous souhaitons pouvoir comparer les mesures de diversités obtenues au sein de recommandations établies par les différents algorithmes avec une mesure de diversité globale. De cette manière, nous pourrions identifier si certains algorithmes enferment les utilisateurs dans une bulle ou si, au contraire, ils leur offrent une plus grande diversité.

Afin de connaître la diversité globale présente à une certaine période de temps, nous avons réalisé la méthode « Div_Globale » (cf. **annexe 2 point 3.1.4**) qui prend un seul paramètre : « t » indiquant la période de temps dont nous souhaitons connaître la diversité globale. Nous reprenons alors la matrice de données correspondant à cette période de temps, issue de la base de données initiale. Grâce à celle-ci, nous identifions l'ensemble des films qui ont été vus durant cette période de temps. Ensuite, nous calculons la diversité présente au sein de tous ces films et nous obtenons ainsi la diversité globale au temps *t* qui est renvoyée par la méthode.

3.2. La nouveauté

À titre de rappel, nous tenions à indiquer à nouveau la formule permettant de connaître la nouveauté d'un ensemble d'items (14).

$$Novelty(i_1, \dots, i_n) = \frac{\sum_{j=1..n}\left(1 - pop(i_j)\right)}{n} \quad (14)$$

Où *Novelty(i_1, …, i_n)* signifie la nouveauté de l'ensemble contenant les items i_1 à i_n ;

n correspond au nombre d'items composant l'ensemble dont nous souhaitons calculer la nouveauté ;

pop(i_j) représente le degré de popularité de l'item i_j et est compris entre 0 et 1.

Nous avons créé la méthode « Novelty » (cf. **annexe 2 point 3.2.1**) permettant de calculer la nouveauté d'un ensemble d'items en mettant en application la formule ci-dessus (14). Cette méthode requiert deux paramètres : le premier correspond à la matrice binaire contenant les données (quel utilisateur a vu quel film) et le second reprend la liste des éléments dont on souhaite connaître la nouveauté. La matrice de données encodée en premier paramètre permet de calculer les degrés de popularité de tous les films composants la base de données. Afin de connaître les degrés de nouveauté de tous les items composant la liste, nous faisons appel à la méthode « Degre_Nouveaute » (cf. **annexe 2 point 2.4.2**) créée précédemment. Les degrés de nouveauté des items composant la liste sont alors sommés et cette somme est divisée par le nombre d'éléments de la liste. La valeur obtenue, correspondant à la nouveauté de cet ensemble, est alors renvoyée par la méthode.

À la suite de cela, nous avons créé une deuxième méthode : « Nouveauté_Moyenne » (cf. **annexe 2 point 3.2.2**) permettant de calculer la nouveauté présente, en moyenne, au sein de plusieurs listes de recommandations. La valeur de la nouveauté moyenne sera forcément bornée entre 0 et 1.

Obtention de la nouveauté globale pour chaque période

Tout comme pour la diversité, nous avons choisi de réaliser une mesure de diversité globale pour chaque période de temps. En comparant celle-ci aux résultats de nouveauté obtenus au sein des recommandations faites par les algorithmes, il nous sera possible d'étudier le potentiel enfermement des utilisateurs dans une bulle.

Afin de calculer ces nouveautés globales, nous avons réalisé la méthode « Nov_Globale » (cf. **annexe 2 point 3.2.3**) qui prend un seul paramètre : « t » correspondant à la période de temps dont nous souhaitons connaître la nouveauté globale. Le même procédé que pour la méthode « Div_Globale » est appliqué pour cette méthode si ce n'est que nous calculons la nouveauté, et non pas la diversité, présente au sein de l'ensemble des films vus au temps *t*. La méthode renvoie alors la nouveauté globale présente dans la base de données au temps *t*.

4. Simulation du comportement des utilisateurs

Nous avons choisi de tester cinq scénarios afin de simuler les comportements de réponse des utilisateurs et d'intégrer ou non les composantes algorithmiques et humaines. Chaque scénario étudié fera évoluer de manière différente la matrice de données, sur laquelle les algorithmes se baseront pour établir les recommandations à la période de temps suivante. Pour rappel, cette évolution de la base de données sera fondée sur deux hypothèses : l'intégration ou non de la composante humaine ainsi que le comportement de réponse simulé des utilisateurs. Nous allons alors construire une méthode permettant d'intégrer la composante humaine ainsi que différentes méthodes permettant de simuler le comportement des utilisateurs. Cela nous permettra de simuler les 5 scénarios en faisant appel, pour chacun d'eux, aux méthodes représentant les décisions sur lesquelles ils se basent et qui permettent de les différencier.

4.1. Intégrer la composante humaine

La méthode que nous avons créée, « Comp_Humaine » (cf. **annexe 2 point 4.1**), permet d'intégrer à la base de données, sur laquelle les algorithmes se basaient pour établir des recommandations à la période de temps précédente, l'ensemble des données provenant de la matrice pour la période de temps suivante, issue de la base de données initiale. Cela permet alors d'intégrer la composante humaine car ces données ajoutées correspondent aux films que les utilisateurs ont choisi de regarder sans l'influence d'un quelconque système de recommandation.

La méthode nécessite de prendre en paramètres deux éléments : la matrice de données précédemment utilisée par les algorithmes, ainsi que la période de temps dont nous devons intégrer les données initiales. Alors, elle effectue le rassemblement de la matrice issue du premier paramètre avec les données provenant de la base de données initiale pour la période de temps indiquée dans le second paramètre. La matrice rassemblée est alors renvoyée par la méthode.

4.2. Simulation du comportement où les utilisateurs acceptent toutes les recommandations

Afin de traduire le comportement des utilisateurs où ces derniers acceptent toutes les recommandations, nous avons créé la méthode « Rec_Acceptees » (cf. **annexe 2 point 4.2**). Cette méthode nécessite deux paramètres. Le premier correspond à la matrice précédemment utilisée par les algorithmes pour établir des recommandations et à laquelle nous souhaitons ajouter d'autres données simulant le comportement des utilisateurs. Le second paramètre reprend un ensemble de listes correspondant chacune à la liste de recommandations établies pour un utilisateur.

À travers cette méthode, nous créons une nouvelle matrice reprenant les mêmes données que celles reprises dans le premier paramètre. Dans cette nouvelle matrice, nous attribuons également une valeur de 1 pour tous les films recommandés auprès de chaque utilisateur, signifiant ainsi que chacun d'eux a vu tous les films qui lui étaient recommandés. Cette méthode renvoie finalement la nouvelle matrice de données.

4.3. Simulation du comportement où les utilisateurs acceptent les recommandations avec une certaine probabilité

Ce comportement simulé est assez similaire au précédent hormis le fait que les utilisateurs n'acceptent pas systématiquement les recommandations qui leur sont faites, ils les acceptent seulement avec une probabilité de 25 %. Nous avons établi la méthode « Rec_Acceptees_Proba » (cf. **annexe 2 point 4.3**) qui permet de simuler ce comportement de réponse des utilisateurs. Cette méthode est assez similaire à la précédente et prend d'ailleurs les mêmes paramètres. Mais, une valeur de 1 n'est pas systématiquement accordée dans la matrice pour toutes les recommandations des utilisateurs. En effet, nous générons un nombre de manière aléatoire entre 0 et 1 et seulement si ce dernier est strictement inférieur à la probabilité choisie (25 %), une valeur de 1 est accordée dans la matrice à l'intersection de l'utilisateur et du film qui lui était recommandé. La nouvelle matrice élaborée est alors renvoyée par la méthode.

4.4. Construction des différents scénarios

Suite à l'élaboration des trois méthodes précédentes, nous sommes maintenant en mesure de construire nos différents scénarios. Pour ce faire, nous avons créé la méthode « Simulation_Scenarios » (cf. **annexe 2 point 4.4** ou Figure A1.12). Cette méthode nécessite quatre paramètres : le premier correspond à la matrice précédemment utilisée par les algorithmes pour établir des recommandations ; le second paramètre indique la période de temps suivante étudiée ; le troisième contient l'ensemble des listes de recommandations établies ; et le dernier indique le numéro du scénario à appliquer.

La matrice qui sera finalement renvoyée par la méthode contiendra les données reprises dans la matrice passée dans le premier paramètre de la méthode ainsi que les données ajoutées selon le scénario choisi. Afin de permettre une meilleure compréhension, nous tenions à incorporer directement l'image correspondant au code de cette méthode (cf. Figure A1.12). Celui-ci est également disponible en **annexe 2 au point 4.4**.

```
### Méthode qui permet la simulation de chacun des scénarios

def Simulation_Scenarios (matrice_avant, t, all_recom, num_sce):

    # Matrice ajoutant la composante humaine
    matrice_comp_hum = Comp_Humaine (matrice_avant, t)

    # 1er scénario : intégration de la composante humaine & les utilisateurs acceptent toutes les recommandations
    if num_sce == 1:
        # Matrice incluant les modifictions liées au scénario choisi
        matrice_apres = Rec_Acceptees (matrice_comp_hum, all_recom)

    # 2ème scénario : intégration de la composante humaine & les utilisateurs acceptent les recommandations avec une probabilité
    if num_sce == 2:
        # Matrice incluant les modifictions liées au scénario choisi
        matrice_apres = Rec_Acceptees_Proba (matrice_comp_hum, all_recom)

    # 3ème scénario : intégration de la composante humaine & les utilisateurs n'acceptent aucune recommandation
    if num_sce == 3:
        # Matrice incluant les modifictions liées au scénario choisi
        matrice_apres = matrice_comp_hum

    # 4ème scénario : pas de composante humaine & les utilisateurs acceptent toutes les recommandations
    if num_sce == 4:
        # Matrice incluant les modifictions liées au scénario choisi
        matrice_apres = Rec_Acceptees (matrice_avant, all_recom)

    # 5ème scénario : pas de composante humaine & les utilisateurs acceptent les recommandations avec une probabilité
    if num_sce == 5:
        # Matrice incluant les modifictions liées au scénario choisi
        matrice_apres = Rec_Acceptees_Proba (matrice_avant, all_recom)

    # Renvoyer la matrice modifiée en fonction du scénario
    return matrice_apres
```

Figure A1.12. Code de la méthode simulant les scénarios

5. Obtention des résultats

Grâce à toutes les méthodes créées jusqu'à présent, nous disposons maintenant de tous les éléments nécessaires pour obtenir les résultats et pouvoir les analyser par la suite. Dès lors, nous avons créé la méthode « Resultats » (cf. **annexe 2 point 5**) permettant d'exécuter le code associé à un certain algorithme et ce, pour toutes les périodes de temps et tous les scénarios.

Cette méthode requiert trois paramètres. Le premier correspond au nombre de recommandations qui doivent être fournies pour chaque utilisateur. Le second indique le nom de l'algorithme à appliquer. Et, le dernier paramètre spécifie le nombre de voisins à sélectionner pour réaliser l'algorithme choisi. Toutefois, certains algorithmes ne nécessitent pas ce paramètre, il s'agit des algorithmes populaire et aléatoire. Dans ces cas, il sera alors nécessaire d'attribuer une valeur négative à ce paramètre pour qu'il ne soit pas utilisé.

À travers cette méthode, nous calculons les mesures de diversité et de nouveauté obtenues au sein des recommandations fournies par l'algorithme sélectionné. Au départ, la base de données sur laquelle l'algorithme se fonde pour établir des recommandations correspond à la division de la base de données initiale pour la première période de temps. Nous parcourons tous les scénarios que nous avons mis en place et, au sein de chacun d'eux, nous parcourons les différentes période de temps. Pour chaque scénario et chaque période de temps, l'algorithme fournit des recommandations dont nous calculons les mesures de diversité et de nouveauté. Finalement, nous créons deux fichiers CSV contenant respectivement les résultats obtenus pour la mesure de diversité et celle de nouveauté. Dans ces fichiers CSV, les lignes correspondent aux périodes de temps et les colonnes aux scénarios appliqués.

Nous avons appliqué cette méthode 18 fois. En effet, cela a été réalisé pour chacun de nos 6 algorithmes et ce, en fournissant 10, 20 et 100 recommandations. À chaque fois, nous avons obtenu deux fichiers CSV (i.e. un pour les mesures de diversité, l'autre pour les mesures de nouveauté) spécifiant l'algorithme et le nombre de recommandations appliqués.

Par exemple, la Figure A1.13 présente l'application pour l'algorithme introduisant plus de diversité, lorsque 100 recommandations doivent être fournies pour chaque utilisateur.

```
# Application
nb_recom = 100
k=15
Resultats (nb_recom, Recommandations_Diversifiees, k)
```

Figure A1.13. Exemple d'application de la méthode permettant d'obtenir les résultats