Projet Data Mining

Les Systèmes de recommandations

Elaboré par :

Marwen TOUZI

Salma BENNISS

Hibatallah HADJ KACEM

Walid GAFSI

Fourat MASTOURI

Table des matières

Introduction :

My New Mood est une jeune start-up créée en 2015 qui vise l'univers de la nuit. Elle a lancé sa première application « SnapBar » et pour répondre aux besoins de ses utilisateurs elle est en train de développer l'application « ListnGO ». Le but de cette application est de pouvoir faire des recommandations personnalisées pour chaque utilisateur.

Dans le cadre du projet Data mining dans le master "Innovation, Marché et Science des Données", nous avons contacté My New Mood pour travailler sur des problématiques réelles et sur des vraies données. Leurs responsables data nous ont proposé de travailler sur une base de données pédagogique "Movie Lens" afin de développer notre POC "Proof of concept". Une fois c'est fait ils vont nous passer leur vraie base de données.

Objectifs :

1. Comprendre et mettre en place les différents types de systèmes de recommandation.
2. Comparer et choisir le ou les meilleurs modèles.
3. Mettre en place en mode production la version choisie du système de recommandation.

1. Description du jeu de données :

GroupLens Research a collecté et mis à disposition des données du site Web MovieLens (http://movielens.org).

Le premier jeu de données **ratings.csv** de 100004 lignes, contient les notes que les utilisateurs ont attribué aux films :

```
print rating.shape
rating.head(4)
```

```
(100004, 4)
```

	userId	movieId	rating	timestamp
0	1	31	2.5	1260759144
1	1	1029	3.0	1260759179
2	1	1061	3.0	1260759182
3	1	1129	2.0	1260759185

Le deuxième jeu de donnée **movies.csv** contient des informations relatives aux films, notamment le titre et les genres concaténés dans une chaîne de caractères :

```
print movies.shape
movies.head(4)
```

```
(9125, 3)
```

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance

Nous présenterons par la suite le package IMDbPY qui nous a permis d'accéder à la base de données IMDB qui contient plus de caractéristiques relatives aux films.

Ceci nous a permis de construire une nouvelle base de données, contenant de nouvelles variables qui caractérisent au mieux les films.

2. Les Systèmes de recommandations :

2.1. Définitions et étapes :

Un système de recommandation est une sorte de forme spécifique de filtrage de l'information visant à présenter les éléments susceptibles d'intéresser l'utilisateur. La mise en place d'un recommander requiert généralement 3 étapes :

- **Collecter les données** : pour le cas basique ce sont soit les caractéristiques des items ou des utilisateurs, soit la notation soumises par l'utilisateur à un item (un film) qui se présente généralement sous forme d'une note sur 5, des etoiles (1er version de Netflix) ou bien des événements binaires tels que les "like" et "follow".

- **Création du modèle** : En général c'est une matrice qu'on la présente souvent sous forme d'un tableau qui croise les utilisateurs et les items selon l'événement principal.

- **Extraire des recommandations** : L'extraction se fait à partir des modèles et, en appliquant quelques traitements pour trouver les recommandation pour un utilisateur, les utilisateurs ou les items les plus similaires ou en appliquant des filtres.

2.2. Types de systèmes de recommandations :

Selon les besoins et selon les données, on utilise souvent l'un de ces approches :

- **La recommandation non personnalisée** : D'habitude, nous utilisons ce type de recommandation lorsque nous n'avons pas assez d'informations sur un utilisateur (lors de son inscription). Une telle recommandation est indépendante

de l'utilisateur et se présente souvent sous forme d'une de ces affirmations:

-Le bar le plus aimé.

-Le restaurant le plus visité.

-La liste la plus suivie.

- **Le Collaboratif Filtering** : Cette recommandation se base sur les événements qui lient les utilisateurs aux items et qui présente d'une manière implicite ou explicite son goût. Ces recommandations sont souvent présentées sous ces affirmations :

  * Les utilisateurs qui sont similaires à vous, ont aimé aussi.

  * Les utilisateurs qui ont aimé ça, ont aimé ça aussi.

- **Le Content Based**: Il s'agit de recommander les items en se basant sur les caractéristiques et les propriétés qui identifient cet item d'une part, et sur les caractéristiques qui intéressent le plus notre utilisateur.

- **La recommandation hybride** : Cette recommandation utilise les 3 types de recommandation. Si nous n'avons pas de données sur un utilisateur, c'est à dire lors de l'inscription d'un utilisateur, nous allons lui proposer des recommandations non personnalisées comme les films les plus aimé ou les plus notés… Une fois qu'il commence à faire des reviews sur des films, nous pouvons passer à la recommandation personnalisée. Et là si nous avons les caractéristiques, nous accédons à la recommandation "Content Based" sinon nous utiliserons la recommandation "Collaborative Filtering" et si nous avons les deux, nous utiliserons les deux à la fois.

- **La recommandation multicritères :** Cette recommandation utilise à la fois plusieurs événements. Il existe deux types d'événements explicites (note, like, note sur la partie comédie du film…) et implicites (le fait qu'un utilisateur visite la page d'un film, si un utilisateur regarde plus que 80% de la vidéo…). Pour notre

cas, vu que nous allons utiliser seulement l'événement note, nous ne pouvons pas implémenter ce type de recommandation.

3. Collaborative Filtering :

Le principe du collaborative filtering est de recommander des éléments sur la base du comportement passé des utilisateurs similaires, en effectuant une corrélation entre des utilisateurs ayant des préférences et intérêts similaires. L'idée de base est donc de dire que si des utilisateurs ont partagés des mêmes intérêts dans le passé, il y a de fortes chances qu'ils partagent aussi les mêmes goûts dans le futur.

Nous distinguons deux types de recommenders Collaborative Filtering :

3.1. Memory-based

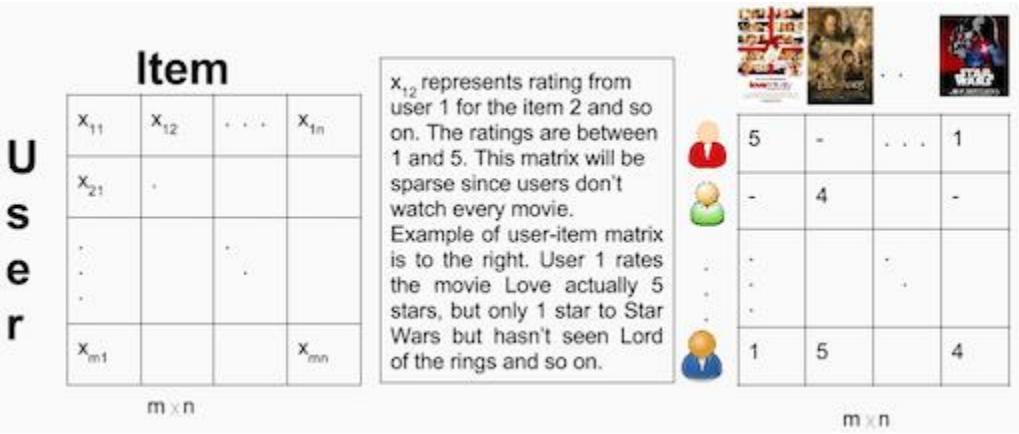
Les approches de filtrage collaboratif Memory-based peuvent être divisées en deux sections principales : le filtrage **User-Item** et le filtrage **Item-Item**. Un filtrage **User-Item** prendra un utilisateur particulier, trouve les utilisateurs qui sont similaires à cet utilisateur en fonction de la similarité des ratings et recommande les items que ces utilisateurs similaires ont aimé. En revanche, le filtrage **Item-Item** prend un item, trouve les utilisateurs qui aiment cet item et trouve d'autres items que ces utilisateurs ou utilisateurs similaires ont également aimé. Il prend des items et produit d'autres éléments comme des recommandations.

- **Approche User-Item :** "Les utilisateurs qui sont similaires à vous, ont aimé aussi ..."
- **Approche Item-Item :** "Les utilisateurs qui ont aimé ça, ont aimé aussi ..."

Dans les deux cas, et comme première étape, nous créons la matrice User-Item à partir de l'ensemble des données où on représentera le croisement des utilisateurs et des items à travers les ratings. Puisque nous devons dans un cas pareil passer par la validation croisée,

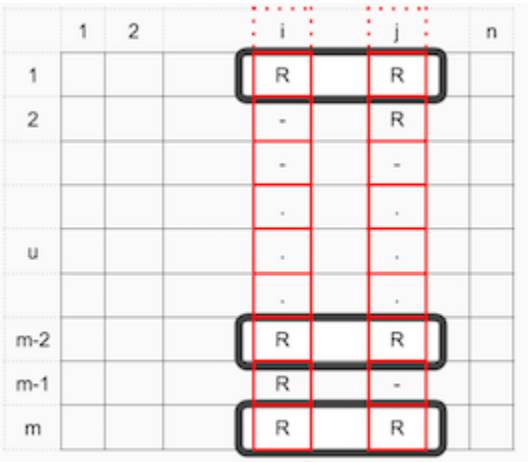
nous obtiendrons deux matrices User-Item Train et Test. La première contiendra 75% des ratings, tandis que la deuxième aura 25% de ces derniers.

Voici dans la figure ci-dessous un exemple de matrice User-Item :

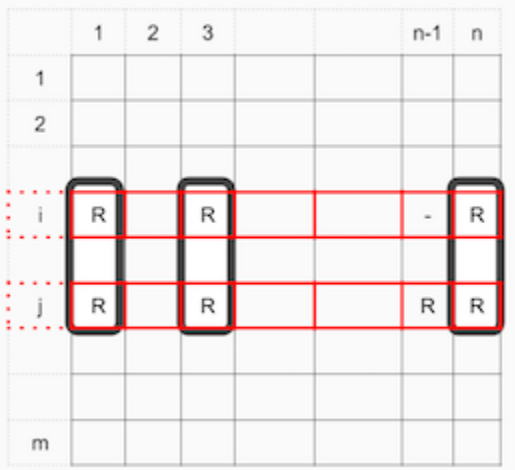


Après avoir construit la matrice User-Item, on doit calculer la similarité et créer une matrice de similarité.

Les valeurs de **similarité** entre deux items i et j dans le cas **Item-Item** sont mesurées en observant tous les utilisateurs ayant noté les deux éléments :



Pour le cas **User-Item**, les valeurs de **similarité** entre deux utilisateurs i et j sont mesurées en observant tous les éléments qui sont notés par les deux utilisateurs.



Pour ce faire, nous avons utilisé les deux métriques :

- Similarité Cosinus (cosine similarity)
- Distance Cityblock.

sur l'ensemble des données d'apprentissage (Train) pour les deux cas User-Item et Item-Item.

Pour évaluer les résultats de similarité obtenus, on doit passer par une métrique d'évaluation pour évaluer l'exactitude des estimations prédites. Bien qu'elles soient nombreuses, mais l'une parmi les plus populaires utilisées est l'erreur quadratique moyenne (RMSE).

Les résultats de similarité sont donnés dans la figure ci-après :

```
User-based CF: The RMSE for the cosine similarity metric is : 1.4977638525
Item-based CF: The RMSE for the cosine similarity metric is :  1.50327465213
User-based CF: The RMSE for the cityblock similarity metric is :   1.5172654194
Item-based CF: The RMSE for the cityblock similarity metric is : 1.55417757528
```

→ Nous pouvons déduire après observation des résultats que le meilleur modèle est celui qui a la plus petite valeur pour RMSE. Pour notre cas c'est User-Item pour -la métrique Cosine.

Pour conclure, on peut dire que les modèles Memory based sont facile à implémenter et produisent une qualité de prévision raisonnable. Ce type de modèles n'est pas scalable, c-à-d n'est pas pratique dans un problème d'une grande base de données vu qu'il calcule à chaque fois la corrélation entre tous les utilisateurs et les items. Ainsi, il ne résout pas le problème de cold start, lorsqu'on commence avec un nouvel utilisateur/item dont on n'a pas assez d'information.

Pour répondre au problème de scalabilité on crée les modèles Model Based qu'on traitera dans la section suivante. Et pour répondre au problème de cold start, on utilise la recommandation Content based qu'on verra aussi ultérieurement.

3.2. Model-Based :

Dans cette partie du projet, nous appliquons le deuxième sous-type du filtrage collaboratif: "Model-based". Il consiste à appliquer la matrice de factorisation (MF) : c'est une méthode d'apprentissage non supervisé de décomposition et de réduction de dimensionnalité pour les variables cachées.

Le but de la matrice de factorisation est d'apprendre les préférences cachées des utilisateurs et les attributs cachés des items depuis les ratings connus dans notre jeu de données, pour enfin prédire les ratings inconnus en multipliant les matrices de variables latentes des utilisateurs et des items.

Il existe plusieurs techniques de réduction de dimensionnalité dans l'implémentation des systèmes de recommandations. Dans notre projet, nous avons utilisé :

- **SVD :** singular value decomposition
- **SGD :** Stochastic Gradient Descent
- **ALS :** Alternating Least Squares

3.2.1. SVD : Singular Value Decomposition :

Cette technique, comme toutes les autres, consiste à réduire la dimensionnalité de la matrice User-Item calculée précédemment. Posons R la matrice User-Item de taille m x n (m : nombre de users, n: nombre d'items) et k: la dimension de l'espace des caractères latents. L'équation générale de SVD est données par : R=USV^T avec:

La matrice U des caractères latents pour les utilisateurs : de taille m*k

La matrice V des caractères latents pour les items : de taille n*k

La matrice diagonale de taille k x k avec des valeurs réelles non-négatives sur la diagonale

On peut faire la prédiction en appliquant la multiplication des 3 matrices.

Mesure de performance:

Nous avons calculé la performance avec RMSE entre la matrice prédite et la matrice du test :

```
print 'RMSE: ' + str(rmse(x, test_data_matrix))
```

```
RMSE: 1.46997911037
```

→ **L'erreur de prédiction mesurée avec RMSE est 1.47, une valeur plus petite que pour les modèles Memory based, donc une performance meilleure. Il est à noter aussi que SVD prend énormément moins de temps pour s'exécuter que les modèles Memory based.**

Un tel résultat du RMSE, est influencé par le nombre de variables latentes **k,** un paramètre important du modèle SVD. Nous l'avons donc exécuté en modifiant à chaque fois les valeurs de k pour se rendre compte à la fin que k=100 est le nombre optimal de variables latentes.

k=50 => RMSE: 1.50142530774

k=75 => RMSE: 1.48097179306

k=80 =>RMSE : 1.4772112327

k=100 => RMSE: 1.46997911037

k=110 => RMSE: 1.46779703654

k=125 => RMSE: 1.47084637082

k=150 => RMSE: 1.47754794975

3.2.2. SGD : Stochastic Gradient Descent:

Quand on utilise le filtrage collaboratif pour SGD, on veut estimer 2 matrices P et Q:

La matrice P des caractères latents pour les utilisateurs : de taille m*k (m: nombre d'utilisateurs, k: dimension de l'espace des caractères cachés)

La matrice Q des caractères latents pour les items : de taille n*k (m: nombre d'items, k: dimension de l'espace des caractères cachés)

Après l'estimation de P et Q, on peut alors prédire les ratings inconnus en multipliant les matrices P et la transposée de Q.

Pour mettre à jour P et Q, on peut utiliser le SGD où on itère chaque observation dans le train pour mettre à jour P et Q au fur et à mesure :

$$Q_{i+1} = Q_i + \gamma(e_{ui} \cdot P_u - \lambda \cdot Q_i)$$

$$P_{u+1} = P_u + \gamma(e_{ui} \cdot Q_i - \lambda \cdot P_u)$$

On note :

γ : la vitesse de l'apprentissage

λ : le Terme de régularisation

e : l'erreur qui est la différence entre le rating réel et le rating prédit.

Le modèle SGD repose sur un nombre d'étapes défini au préalable.

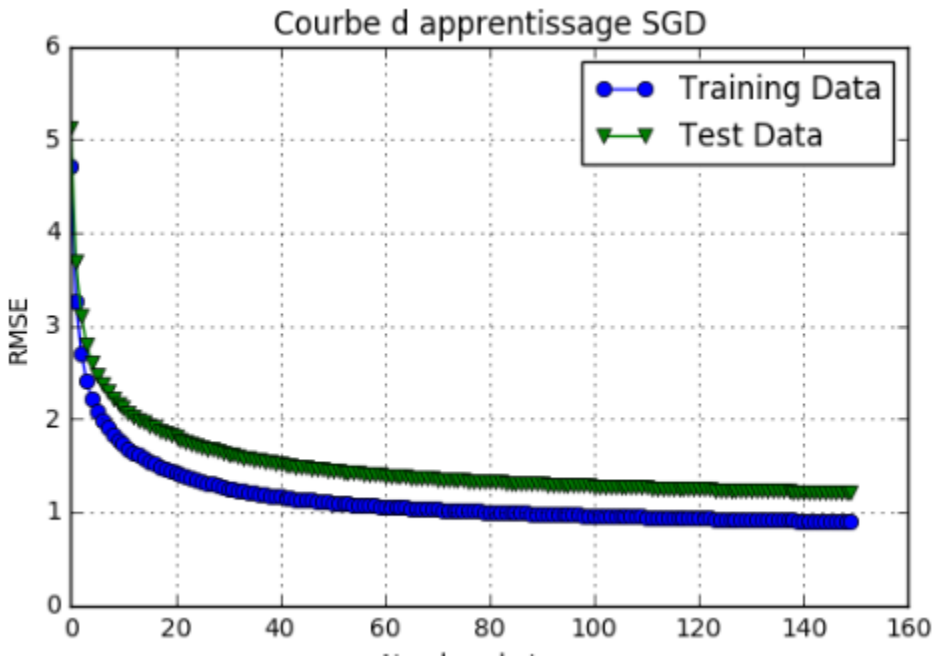
Alors, pour la mesure de performance avec RMSE, nous calculons à chaque étape la RMSE respective. Pour obtenir l'erreur de tout le modèle, nous calculons la moyenne des RMSE à toutes les étapes :

```
print 'RMSE : ' + str(np.mean(test_errors))
RMSE : 1.50060377979
```

→ **La valeur de RMSE obtenue est supérieure à celle obtenue avec SVD mais reste meilleure que les modèles Memory based.**

Nous ne nous sommes pas limités à la valeur de RMSE finale, nous avons donc représenter l'erreur en fonction du nombre d'étapes préalablement fixé pour les données d'apprentissage et de validation.

Nous remarquons que les deux courbes convergent et la valeur de RMSE diminue à chaque nouvelle étape.



3.2.3. ALS : Alternating Least Square:

Dans cette partie, notre objectif avec ALS est d'estimer les matrices P et Q, tout comme SGD décrit dans la section précédente.

Après l'estimation de P et Q, on peut alors prédire les ratings inconnus en multipliant la transposée de la matrice P par la matrice Q.

Comme les anciennes méthodes Model Based traitée, nous avons calculé la RMSE pour le modèle ALS.

Comme pour SVD, ALS admet un nombre d'étapes défini à travers lequel la RMSE diminue jusqu'à converger.

Dans le cas de notre modèle, bien que l'exécution de ALS prenne plus de temps que SGD, nous nous sommes limité à 2 étapes de l'algorithme.

```
[Epoch 1/2] train error: 1.054704
[Epoch 2/2] train error: 0.799782
```

→ Nous remarquons que cette valeur de l'erreur pour ALS est la meilleure parmi toutes les autres méthodes testées.

→ Nous pouvons conclure que le meilleur modèle Collaborative Filtering est ALS.

3.3. Recommandation Collaborative Filtering (CF) :

Après avoir choisi ALS comme le meilleur modèle, nous avons généré la matrice de recommandation User-Item sur tout notre jeu de données :

	0	1	2	3	4	5	6	7	8	9	...
0	2.233775	1.905548	1.979336	1.276262	2.004535	2.211248	2.090420	1.361764	1.833609	2.046868	...
1	3.336108	2.668035	2.227345	1.979687	2.577148	3.245921	2.848151	2.022580	2.452251	3.065714	...
2	3.175533	3.004031	2.542417	1.994644	2.636938	3.253133	2.641424	2.735721	2.437748	2.895012	...
3	4.093425	3.576903	3.055865	2.527552	3.081193	3.956807	3.612208	3.047209	3.127855	3.623088	...
4	3.635113	3.262149	3.111515	1.998816	3.103246	3.583235	3.138600	2.613821	2.845363	3.329217	...
5	2.912161	2.577775	2.175380	1.540490	2.648150	3.357966	2.327505	2.022765	2.096726	2.735201	...
6	3.241829	2.708606	2.048110	1.968460	2.289350	3.180516	2.668291	2.454546	2.091060	2.749421	...
7	3.423545	2.963637	2.700463	1.969871	2.753317	3.460713	2.836455	2.486518	2.556937	2.991934	...
8	3.561172	2.777051	2.373548	1.961444	2.350989	3.278866	2.895296	2.216890	2.158549	2.918526	...

Pour regarder au mieux l'efficacité de notre modèle, nous avons implémenté un ensemble de fonctions qui permettent de faire la recommandation :

- **Recommandation d'Items pour un utilisateur donné :**

```
In [283]: als_recom_it_for_user(10)

Out[283]: [(4.5834732958049287, 59684),
           (4.5834732958049287, 65037),
           (4.4426067179852913, 3112),
           (4.3887630171506746, 9010),
           (4.3857412622619396, 4114),
           (4.3857412622619396, 48682),
           (4.3857412622619396, 107555),
           (4.3857412622619396, 59447),
```

Dans cet exemple, nous cherchons à recommander à l'utilisateur 10 un exemple d'items qui peuvent lui intéresser tout en associant pour chacun les ratings prédits. Nous pouvons voir que l'item "59684" est le plus adéquat.

- **Détecter les utilisateurs similaires à un utilisateur donné :**

```
In [288]: cf_similaire_user(10)

Out[288]: [(0.82778392820449909, 329),
           (0.80420586855707976, 627),
           (0.78467739110222534, 101),
           (0.77747538364823443, 414),
           (0.7754388158030967, 5),
           (0.77189587607207888, 21),
           (0.76747491253035649, 798),
           (0.76467351591964638, 340),
           (0.76084187782585844, 589),
```

Nous voulons à travers cet exemple de retourner les utilisateurs similaires à l'utilisateur 10 avec la corrélation entre eux.

- **Détecter les items similaires à un item donné :**

```
In [313]: cf_similaire_item(1029,20)

Out[313]: [(0.9848840772343318, 596),
           (0.98473741231587475, 135137),
           (0.98238084575734641, 1130),
           (0.98228048466534534, 594),
           (0.9811515773277133, 3897),
           (0.98112934297573706, 2804),
           (0.98107422969374891, 1674),
           (0.98089422842828411, 3252),
           (0.98032462112294283, 1032),
           (0.98024307191820714, 2183),
           (0.97993144137321064, 954),
           (0.97990415799550667, 1086),
           (0.97988711650489102, 357),
           (0.97972014065586077, 39),
           (0.97882431172199891, 4034),
           (0.97873265672299281, 78034),
           (0.97856952952940468, 1304),
           (0.97846266529755288, 2090)]
```

Dans cet exemple, nous affichons les 20 premiers items similaires à l'item "1029" tout en calculant la corrélation entre eux. L'item "596" est le plus similaire avec une corrélation 0.98.

- **Estimer le rating à partir d'un user et d'un item donnés :**

```
In [294]: als_rate_for_user_item(10,1029)
Out[294]: 3.3251129291900243
```

Le rating estimé pour le film “1029” par l’utilisateur “10” est de 3.25

- **Calcul de corrélations :**

Nous avons, en plus des recommandations et l’estimation des notes attribués aux films, implémenté des fonctions qui calculent la corrélation :

- *Entre deux utilisateurs :*

```
In [297]: cf_cor_user_user(3,4)
Out[297]: 0.68815450116517474
```

- *Entre deux items :*

```
In [300]: cf_cor_item_item(3,4)
Out[300]: 0.7362403488008078
```

- *Entre un utilisateur et un item :* Nous calculons à travers cet exemple la corrélation entre l’utilisateur 3 et l’item 4. Cette fonction sera utilisée dans la recommandation hybride que nous traiterons ultérieurement.

```
In [304]: cf_cor_user_item(3,4)
Out[304]: 0.70212099127912952
```

4. Content-Based

4.1. Création de la bases de données des caractéristiques des films avec IMDbPY:

La recommandation objet ou Content-Based Filtering est différente du Collaborative Filtering car elle ne se base pas sur le comportement passé de l'utilisateur. Il s'agit de recommander des objets en se basant sur les qualités et les propriétés intrinsèques de l'objet lui-même et en les corrélant avec les préférences et les intérêts de l'utilisateur. Pour notre cas, nous nous intéresserons qu'à la corrélation d'un objet avec un autre objet.

Notre jeu de données "movies.csv" admet 3 variables "movieId" qui représente l'identifiant du film, "title" le nom du film et "genres" qui permet de catégoriser les films en fonction de leurs thèmes sachant qu'un film peut avoir plusieurs genres.

Pour commencer, nous avons traité la similarité entre les films en se basant uniquement sur la variable "genres". Cela ne nous a pas paru suffisant en termes de degré de similarité, alors nous avons pensé à utiliser le package python IMDbPY pour avoir de nouvelles variables qui caractérisent au mieux les films de notre jeu de données. Les nouvelles variables ajoutées sont les suivantes : genre, year, kind, director, cast, writer, rating, runtimes, countries, languages, production companies. L'objet "ia" est utilisé pour permettre l'accès à la base de données IMDB.

```
# Create the object that will be used to access the IMDb's database.
ia = imdb.IMDb()
```

Cette création du nouveau jeu de données a été réalisée sur quelques étapes. Nous avons commencé par créer un nouveau DataFrame dont on assigne aux variables "movieId" et "titre" les valeurs de notre jeu de données initial, et nous mettons à NULL les autres variables.

```
movies = pd.read_csv('./ml-latest-small/movies.csv')
newMovies=movies[['movieId','title']]
newMovies['genre']=np.NaN
newMovies['year']=np.NaN
newMovies['writer']=np.NaN
newMovies['director']=np.NaN
newMovies['cast']=np.NaN
newMovies['rating']=np.NaN
newMovies['runtimes']=np.NaN
newMovies['countries']=np.NaN
newMovies['languages']=np.NaN
newMovies['production companies']=np.NaN
```

La première difficulté que nous avons rencontrée se manifeste dans la différence entre les identifiants de nos films dans le jeu de données initial et ceux du Package IMDB.

La fonction “search_movie(title)” en assignant le nom du film ne fournit pas directement toutes les caractéristiques de l’objet dont nous avons besoin, elle retourne une liste de films contenant uniquement des informations basiques comme le titre et l’année de diffusion. En ajoutant l’instance de variable “MovieID”, cela nous permet de retourner l’identifiant du film dans IMDB.

```
# Search for a movie (get a list of Movie objects).
s_result = ia.search_movie('Up Close and Personal (1996)')
#s_result
myID=s_result[0].movieID
```

Une fois l’identifiant de l’objet récupéré, nous pouvons avoir les valeurs des nouvelles variables à l’aide de la méthode “get_movie(ID)”. Cette fonction retourne un objet de type IMDB Movie.

```
item=ia.get_movie(myID)
```

Maintenant que nous possédons les caractéristiques de tous les films à partir de leurs titres et leurs identifiants IMDB, nous pouvons créer notre nouveau jeu de données avec tous les films et leurs caractéristiques.

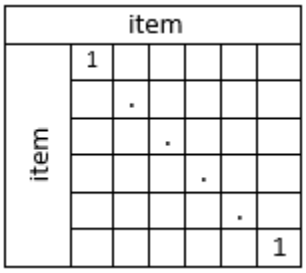
Pour regarder au mieux la performance de notre solution, nous avons voulu mesurer le temps d'exécution et regarder à chaque ligne du DataFrame, combien de temps pourrait

prendre les méthodes search_movie() et get_movie() pour savoir à la fin le temps d'exécution final sur tout le jeu de données.

```
ia.search_movie:  0.776999950409
ia.get_movie:  2.43899989128
ia.search_movie:  0.795000076294
ia.get_movie:  2.14100003242
ia.search_movie:  0.895999908447
ia.get_movie:  2.31299996376
ia.search_movie:  0.842000007629
ia.get_movie:  3.13700008392
```

4.2. Mesure de la similarité entre les items

L'idée ici est de créer une matrice carré (l x l : avec l= nombre d'item) qui contient la distance entre une paire d'Item afin d'en déduire les items les plus proche entre eux, et



donc les plus similaires :

Traitons le cas où on veut mesurer la similarité des items en fonction de la variable **"genre"**(romance, aventure, children..).

Dans notre base de données actuelle la variable genre s'écrit sous la forme :

Dans un premier temps nous devons convertir la colonne genre en une matrice X contenant 0 ou 1 : tel que les lignes de la matrice représentent les items (les films) et les colonnes représentent tous les différents types que l'on possède dans notre base de

	movield	title	genre	year	writer	director	cast	rating	runtimes	countries	languag
0	1	Toy Story (1995)	Animation,Adventure,Comedy,Family,Fantasy	1995.0	John Lasseter,Pete Docter,Andrew Stanton,Joe R...	John Lasseter,	Tom Hanks,Tim Allen,Don Rickles,Jim Varney,Wal...	8.3	81	USA,	English,

données et donc X[i,j]= 1 si parmi les genres du film i, il existe le genre j :

item	comedy	animation	children	Family
1	1	0	1	0
2	0	1	0	0
3	1	1	1	0
4	0	0	0	1
5	1	0	0	1

Pour réaliser cette matrice on a utilisé la fonction **vectorizer()** du package **sklearn** de **Python :**

```
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(data['genre'].tolist())
Xarray=X.toarray()
```

Maintenant qu'on a construit la matrice que l'on souhaitait, il est temps de choisir une métrique pour calculer la similarité entre deux items.

Il existe plusieurs métriques pour mesurer la similarité entre deux items, dans notre cas on a choisi **cosine_similarity.**

- La métrique **cosine_similarity :**

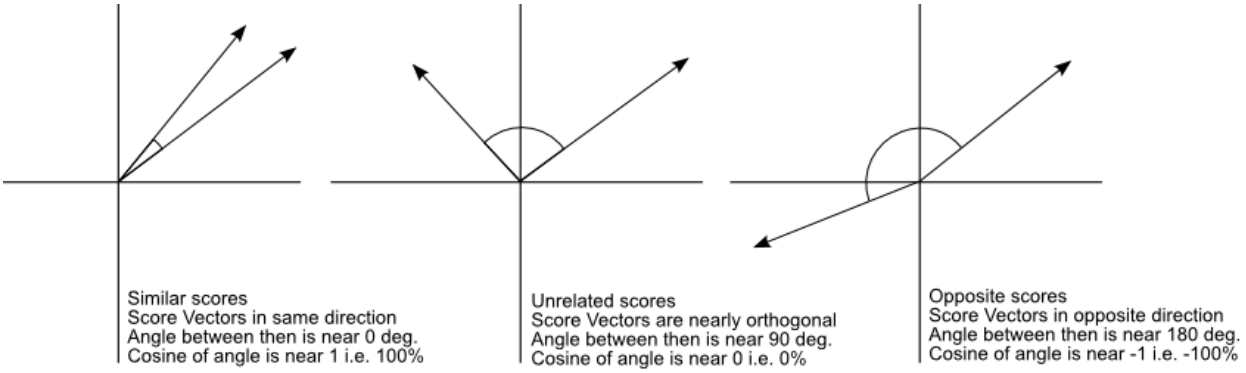
cosine_similarity entre deux vecteurs (ou deux documents sur l'espace vectoriel) est une mesure qui calcule le cosinus de l'angle entre eux. Cette métrique est une mesure d'orientation, elle peut être considérée comme une comparaison entre des items sur un espace normalisé parce que nous ne prenons pas en considération uniquement l'importance de chaque nombre de mots (tf-idf) de chaque item, mais aussi l'angle entre les items.

Ce que nous devons faire pour construire l'équation de **cosine_similarity** est de résoudre l'équation suivante :

$$\vec{a} \cdot \vec{b} = \|\vec{a}\| \|\vec{b}\| \cos\theta$$

$$\cos\theta = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \|\vec{b}\|}$$

Cosine_similarity générera une métrique qui indique comment sont liés deux items en regardant l'angle au lieu de la grandeur, comme dans les exemples ci-dessous:



Après l'application de la métrique **cosine_similarity**, on obtient finalement notre matrice qui mesure la similarité entre toutes paires d'item :

```
pd.DataFrame(cosine_similarities1)
```

	0	1	2	3	4	5	6	7	8	9	...	889
0	1.000000	0.600000	0.316228	0.258199	0.516398	0.000000	0.316228	0.547723	0.000000	0.258199	...	0.316228
1	0.600000	1.000000	0.000000	0.000000	0.258199	0.447214	0.000000	0.365148	0.516398	0.774597	...	0.000000
2	0.316228	0.000000	1.000000	0.816497	0.816497	0.000000	0.500000	0.577350	0.000000	0.000000	...	0.500000
3	0.258199	0.000000	0.816497	1.000000	0.666667	0.288675	0.816497	0.707107	0.000000	0.000000	...	0.816497
4	0.516398	0.258199	0.816497	0.666667	1.000000	0.000000	0.408248	0.707107	0.000000	0.000000	...	0.408248
5	0.000000	0.447214	0.000000	0.288675	0.000000	1.000000	0.353553	0.204124	0.866025	0.577350	...	0.353553
6	0.316228	0.000000	0.500000	0.816497	0.408248	0.353553	1.000000	0.577350	0.000000	0.000000	...	1.000000
7	0.547723	0.365148	0.577350	0.707107	0.707107	0.204124	0.577350	1.000000	0.000000	0.235702	...	0.577350
8	0.000000	0.516398	0.000000	0.000000	0.000000	0.866025	0.000000	0.000000	1.000000	0.666667	...	0.000000
9	0.258199	0.774597	0.000000	0.000000	0.000000	0.577350	0.000000	0.235702	0.666667	1.000000	...	0.000000
10	0.258199	0.000000	0.816497	1.000000	0.666667	0.288675	0.816497	0.707107	0.000000	0.000000	...	0.816497
11	0.516398	0.258199	0.408248	0.333333	0.333333	0.000000	0.408248	0.235702	0.000000	0.000000	...	0.408248
12	0.600000	0.400000	0.000000	0.258199	0.258199	0.223607	0.316228	0.547723	0.000000	0.258199	...	0.316228

4.3. Recommandation Content Based (CB)

4.3.1. Recommandation basée sur la variable “genre”:

Après avoir obtenu la matrice de similarité, pour chaque item on récupère les indices des items qui sont le plus similaires à notre item sélectionné : plus la métrique cosine_similarity est grande plus les deux items sont similaires.

Deux items sont parfaitement identiques si cosine_similarity = 1

Testons notre fonction sur le premier item : 1

```
cb_similaire_item1(1)
```

```
[(0.9999999999999989, 2090),
 (0.9999999999999989, 2355),
 (0.9128709291752769, 1031),
 (0.9128709291752769, 1030),
 (0.9128709291752769, 1025),
 (0.89442719099991586, 2294),
 (0.89442719099991586, 2050),
```

La fonction nous retourne des items classés selon l'ordre de la similarité. Dans notre exemple la fonction nous indique que les films les plus proche au film d'Id 1 **en termes de genre**, sont : 2090,2355, 1031, 1030.

Vérifions cela :

```
data.query('movieId==1 | movieId==2090 |  movieId==2355 | movieId==1031 | movieId==1030' )
```

	movieId	title	genre	year	writer
0	1	Toy Story (1995)	Animation,Adventure,Comedy,Family,Fantasy	1995.0	JohnLasseter,PeteD
754	1030	Pete's Dragon (1977)	Animation,Adventure,Comedy,Family,Fantasy,Musical	1977.0	MalcolmMarmorstein
755	1031	Bedknobs and Broomsticks (1971)	Animation,Adventure,Comedy,Family,Fantasy,Musical	1971.0	RalphWright,TedBer
1488	2090	Rescuers, The (1977)	Animation,Adventure,Comedy,Family,Fantasy	1977.0	MargerySharp,Larry(
1688	2355	Bug's Life, A (1998)	Animation,Adventure,Comedy,Family,Fantasy	1998.0	JohnLasseter,Andre

On voit bien que le résultat obtenu est très pertinent, puisque l'item le plus ressemblant à notre film "**Toy Story"** est bien le film "**Rescuers, The (1977)**" du moment qu'ils sont identiques au niveau du genres.

4.3.2. Recommandation basée sur la variable "writer":

On fait la même chose qu'on avait fait sur la variable "genre", on obtient les résultats ci-dessous:

```
cb_similaire_item2(1)
```

```
[(0.67082039324993692, 2355),
 (0.44721359549995793, 1604),
 (0.22360679774997896, 2089),
 (0.15811388300841897, 1690),
 (0.087705801930702931, 595),
 (0.067419986246324198, 709),
 (0.057735026918962574, 364),
```

Vérifions cela :

```
data.query('movieId==1 | movieId==2355 |  movieId==1604' )[["movieId","title","writer"]]
```

	movieId	title	writer
0	1	Toy Story (1995)	JohnLasseter,PeteDocter,AndrewStanton,JoeRanft,JossWhedon,AndrewStanton,JoelCohen,AlecSokolow,
1150	1604	Money Talks (1997)	JoelCohen,AlecSokolow,
1688	2355	Bug's Life, A (1998)	JohnLasseter,AndrewStanton,JoeRanft,AndrewStanton,DonMcEnery,BobShaw,

Ce résultat est très pertinent puisqu'on remarque bien que le film **Bug's Life**, d'Id 2355 a 3 écrivant en commun avec le film **Toy Story** sur lequel on a fait notre recherche.

4.3.3. Recommandation basée sur la variable "director":

On applique la même fonction que précédemment on obtient :

```
cb_similaire_item3(1)
```

```
[(0.70710678118654746, 2355),
```

```
(data.query('movieId==1 | movieId==2355 ' ))[['movieId','title','director']]
```

	movieId	title	director
0	1	Toy Story (1995)	JohnLasseter,
1688	2355	Bug's Life, A (1998)	JohnLasseter,AndrewStanton,

Les deux films **Bug's Life** et **Toy Story** ont tous les deux John Lasseter comme directeur

4.3.4. Recommandation basée sur la variable "Production companies":

On applique la même fonction que précédemment on obtient :

```
cb_similaire_item5(1)
```

```
[(0.9999999999999978, 2355),
 (0.70710678118654746, 1020),
 (0.70710678118654746, 1282),
 (0.70710678118654746, 2101),
```

```
(data.query('movieId==1 | movieId==2355 | movieId==1020 ' ))[['movieId','title','production companies']]
```

	movieId	title	production companies
0	1	Toy Story (1995)	PixarAnimationStudios,WaltDisneyPictures,
745	1020	Cool Runnings (1993)	WaltDisneyPictures,
1688	2355	Bug's Life, A (1998)	PixarAnimationStudios,WaltDisneyPictures,

Les deux films **Bug's Life** et **Toy Story** ont exactement les mêmes "production companies", ce qui prouve que nos résultats sont très pertinents.

Remarque : En fonction de tous ces résultats qu'on a eu précédemment, on remarque que l'item d'indice 2355, apparaît sur tous les résultats, en tant qu'item très similaire à notre item sélectionné.

4.3.5. Recommandation générale :

L'idée ici est de combiner toutes les recommandations faites précédemment, pour en déduire une recommandation finale qui sera encore plus performante.

Pour ceci, pour chaque item pour lequel on veut chercher les items similaires, nous avons appliqué une fonction qui crée un DataFrame qui contient tous les **Id_movies** avec leurs scores (de similarité par rapport à l'item sélectionné) pour chaque recommandation.

Ci-dessous un exemple :

recommendation based on all

```
l,c=data.shape

def content_based(i):#i est l'indice du film pour lequel on veut chercher les item similaire
    cb1=cb_similaire_item1(i,l)#similarity based on genre
    cb2=cb_similaire_item2(i,l)#similarity based on writer
    cb3=cb_similaire_item3(i,l)#similarity based on director
    cb4=cb_similaire_item4(i,l)#similarity based on countries
    cb5=cb_similaire_item5(i,l)#similarity based on production companies

    df1=pd.DataFrame(cb1,columns=['score_genre','movieId'])
    df2=pd.DataFrame(cb2,columns=['score_writer','movieId'])
    df3=pd.DataFrame(cb3,columns=['score_director','movieId'])
    df4=pd.DataFrame(cb4,columns=['score_countries','movieId'])
    df5=pd.DataFrame(cb5,columns=['score_production_companies','movieId'])

    df_all=df1.join(df2.set_index('movieId'), on='movieId').join(df3.set_index('movieId'), on='movieId').join(df4.set_index('m
    df_all["score_all"]=df_all["score_genre"]+df_all["score_writer"]+df_all["score_director"]+df_all["score_countries"]+df_all[
    df_all=df_all.sort_values(by='score_all',ascending=False)

    return df_all
```

```
content_based(1).head(4)
```

	score_genre	movieId	score_writer	score_director	score_countries	score_production_companies	score_all
1	1.000000	2355	0.67082	0.707107	1.0	1.000000	4.377927
24	0.800000	1282	0.00000	0.000000	1.0	0.707107	2.507107
73	0.670820	1020	0.00000	0.000000	1.0	0.707107	2.377927
11	0.845154	1566	0.00000	0.000000	1.0	0.500000	2.345154

Dans notre cas on peut conclure que l'**item 2355** est le plus similaire à l'**item 1**, en terme du **genre, writer, director, countries et production companies**, puisqu'il a un score totale de **4.37** .

Ceci vient confirmer notre remarque qu'on avait fait précédemment.

5. Recommandation Hybride :

Après avoir présenté le Collaboratif filtering et le Content Based, l'idée derrière cette partie est de créer un modèle qui combine les résultats des deux méthodes précédentes : la méthode **Hybride**.

Dans cette partie on a essayé de combiner entre les scores de similarités des items (par rapport à un certain item) obtenu par les deux méthodes : CF et CB.

Pour illustrer notre idée, on prend à titre d'exemple le cas où on veut récupérer les indices d'Items similaires à l'Item 1, en combinant le Collaboratif Filtering et le content based basé uniquement sur le genre.

On obtient les résultats suivant :

```
l=n_items
def hybrid(i):#i est l'indice du film pour lequel on veut chercher les items similaires
    cb=cb_similaire_item(i,l)#content based filtering based on genre only !!
    cf=cf_similaire_item(i,l)#collaborative filtering

    df1=pd.DataFrame(cb,columns=['score_cb','movieId'])
    df2=pd.DataFrame(cf,columns=['score_cf','movieId'])

    df_all=df1.join(df2.set_index('movieId'), on='movieId')
    df_all["score_all"]=df_all["score_cb"]+df_all["score_cf"]
    df_all=df_all.sort_values(by='score_all',ascending=False)

    return df_all
```

```
(hybrid(1)).head(4)
```

	score_cb	movieId	score_cf	score_all
5	1.0	4886	0.983677	1.983677
4	1.0	3114	0.982578	1.982578
0	1.0	2294	0.962589	1.962589
8	1.0	136016	0.938667	1.938667

On fonction de ce résultat, on pourrait conclure que l'item 4886 est très similaire à l'item 1.

6. Mise en production

6.1. Langages et technologies utilisées:

- **Python** :

Python est l'un des langages les plus simples pour faire du machine learning. Pour cela, nous avons décidé de construire et comparer tous les modèles avec Python. Une fois, nous avons trouvé le meilleur modèle, nous l'avons implémenté avec Spark/Scala.

- **Scala** :

Scala est un langage de programmation multi paradigme, c'est à dire qui intègre les paradigmes de programmation orienté objet (qui met en avant le changement d'état) et les paradigmes de programmation fonctionnelle (qui met en avant l'application des fonctions). Scala offre à l'utilisateur de choisir le paradigme qui lui convient selon ses besoins.

Un code Scala est compilé en byte code et exécutable sous la JVM (Java Virtual Machine). Avec Scala, on peut utiliser les bibliothèques écrites en Java et il est possible d'invoquer du code Scala à partir des programmes Java ce qui facilite la transition Java-Scala.

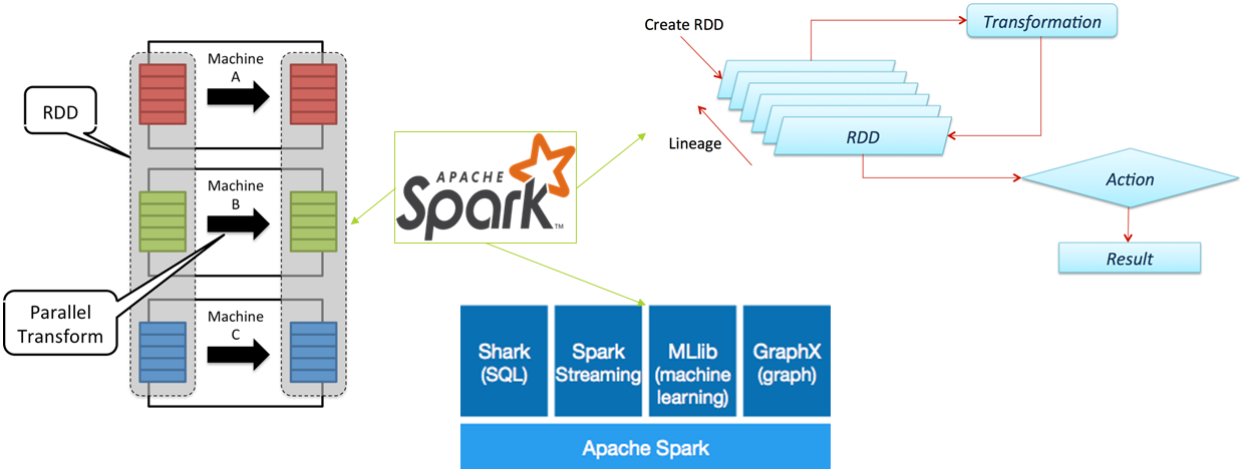
Nous avons choisi le langage Scala pour faciliter la tâche de l'intégration dans l'application développé en Java Scripte par la startup.

- **Spark**:

Spark est un Framework open source de calcul distribué qui travaille en mémoire vive ce qui est cent fois plus rapide que Hadoop qui utilise le paterne de conception Map Reduce sur disques.

La philosophie de Spark se base sur la notion des RDD (Resilient Distributed Dataset). Les RDD présentent les collections des données qu'on va les utiliser. Ce qui caractérise les RDD c'est l'exécution implicite des opérations en parallèle. C'est à dire, nous ne sont pas censé de programmer la partie de calcul distribué, mais, n'importe quelle opération tournera automatiquement en suivant le paradigme Map Reduce. De plus, avec les autres solutions de Big Data, On perd toujours du temps pour transférer les données du disque dur à la mémoire vive, faire les opérations puis stocker le tout sur le disque dur. Ce qui est nouveau chez Spark, c'est que nous pouvons mettre directement les données sur la mémoire cache ce qui réduit énormément le temps d'exécution.

Notre choix de Spark s'est basé sur la grande masse de données généré chaque jours par la startup ce qui posera à long terme des problèmes par rapport au temp d'exécution.



6.2. Réalisation :

Comme notre meilleur modèle construit avec les modèles Collaboratif filtering était l'ALS: Alternating least squares, nous l'avons implémenté avec Spark et Scala.

étape 1: Charger et parser les données (création des RDDs)

```
val small_ratings_raw_data = sc.textFile("ratings.csv")
```

```
val splits = small_ratings_data.randomSplit(Array(6, 2, 2), seed=0L)//60%train, 20%validation, 20%test
val training_RDD =splits(0) //X,Y train matrix
val validation_RDD=splits(1)//X,Y validation matrix
val test_RDD=splits(2)//X,Y test matrix
```

```
//param de modele:
val seed = 5L
val iterations = 10
val regularization_parameter = 0.1
val ranks = Array(4, 8, 12)

val tolerance = 0.02
val min_error: Double = Double.PositiveInfinity
val best_rank = -1
val best_iteration = -1
val ratings_train = training_RDD.map{case (user, item, rate) =>
    Rating(user, item, rate)
  } //xy train rating matrix
val ratings_valid = validation_RDD.map{case (user, item, rate) =>
    Rating(user, item, rate)
  } //xy validation matrix
val rating_test= test_RDD.map{case (user, item, rate) =>
    Rating(user, item, rate)
  } //xy test matrix
```

étape 2: Collaborative filtering

on utilise la librairie Spark MLlib qui contient l'implémentation de la méthode du Collaborative Filtering utilisant la méthode ALS (Alternating Least Squares)

- Choix de paramètre rank (nombre de variable latentes à créer) du modèle ALS: construction du modèle sur une base de train et le tester sur une base de test

Voilà l'implémentation de notre algorithme :

```
var errors = Array(0.0, 0.0, 0.0)
var err = 0
for (rank <- ranks){
    val model= ALS.train(ratings_train, rank, iterations, regularization_parameter) //construction du modele
    val predictions =
      model.predict(ratings_valid.map{case Rating(u,p,r)=>(u,p)}).map { case Rating(user, product, rate) =>
        ((user, product), rate)
      }//prediction sur xy validation matrix Rating
    val ratesAndPreds = ratings_valid.map { case Rating(user, product, rate) =>
      ((user, product), rate)
    }.join(predictions) //jointure de xy validation & xy prediction donne un truc de la forme (u1,i2),(4.01,4.2)
    var error = ratesAndPreds.map { case ((user, product), (r1, r2)) =>
      (r1 - r2)*(r1-r2)
    }.mean()
    errors(err) = error
    err = err +1

}
```

```
In [18]: errors //meilleur err est pour rank==24
Out[18]: Array(0.4876676022721085, 0.39997350625431854, 0.3481989677701417, 0.288097924016866)
```

étape 3: Concentrer la recommandation sur les films les moins notés. Un des grands problème des systèmes de recommandations est le fait de recommander toujours le 20% de la base qui représente les items les plus populaires (vu qu'il sont les plus notés, il seront les plus corrélés avec les autre items).

```
In [25]: //recomandation for a user:
// we want to recommand movies with minimal number of rates:
def get_counts_and_averages(x:Int,y:Array[Double])={//prend Rating Matrix
    val nratings = y.size
    var s=0.0
    for (elt<-y) {
    s=s+elt
    }
    s=s/nratings
    (x,(nratings, s))

}
```

étape 4: Création d'un nouvel utilisateur avec des ratings, et fusionner ce nouveau RDD avec l'ancienne base de données => on utilisera cette fonction pour ajouter des utilisateurs à la base de données, ou pour mettre à jour la base des notes si on a un nouvel événement "review".

```
In [22]: //Adding new user rating id==0 is not used in the data base
val new_user_ID=0
val new_user_ratings = Array(
     (0,260,4), // Star Wars (1977)
     (0,1,3), // Toy Story (1995)
     (0,16,3), // Casino (1995)
     (0,25,4), // Leaving Las Vegas (1995)
     (0,32,4), // Twelve Monkeys (a.k.a. 12 Monkeys) (1995)
     (0,335,1), // Flintstones, The (1994)
     (0,379,1), // Timecop (1994)
     (0,296,3), // Pulp Fiction (1994)
     (0,858,5) , // Godfather, The (1972)
     (0,50,4) // Usual Suspects, The (1995)
    )
val new_user_ratings_RDD = sc.parallelize(new_user_ratings)//transform array to RDD
```

étape 5: Pour un utilisateur donné, afficher les cinq films les plus susceptibles à lui plaire (Parmis les films qui n'a jamais noté) => on utilisera cette fonction dans l'application pour générer les recommandations aux utilisateurs

```
In [59]: val top_movies = new_user_recommendations_rating_title_and_count_RDD_format.filter(x=>x._3>=25)
             .map(x=>(x._1,x._2)).map(item => item.swap).sortByKey(false, 1).map(item => item.swap).take(5)

In [41]: top_movies.take(5)

Out[41]: Array((Modern Times (1936),4.265634643048782), (There Will Be Blood (2007),4.2168618496493835), ("Lives of Others,4.18043946618
         4994), (Cinema Paradiso (Nuovo cinema Paradiso) (1989),4.143232017416185), ("Third Man,4.127757003770054))
```

étape 6: Pour un utilisateur donné et pour un film donné, on peut estimer la note et la comparer avec la note réelle => on utilisera cette fonction dans l'application pour afficher à un utilisateur sur une page d'établissement, la note prédite grâce à notre algorithme, afin de l'inciter à donner sa propre note.

```
In [45]: new_ratings_model.predict(0,260)//dans la base c'est 4! //c'est la prediction du notre us=0, it=260

Out[45]: 3.418232923178837
```

Conclusion :

La librairie MlLib de spark offre des méthodes d'analyses de données très performants, notamment la méthode ALS du collaborative filtering, Model based qui s'adapte parfaitement à notre système de recommandation.

Naturellement sur des grandes échelles de données, même les méthodes de factorisation de matrice peuvent devenir assez coûteuses puisqu'elles nécessitent la mise à jour du modèle, après l'ajout des utilisateurs, des items et/ou de leurs préférences. D'où l'intérêt de l'utilisation des systèmes de calcul distribué comme Spark qui se base à la fois sur les algorithmes "map reduce" et sur le traitement en mémoire.

Pour assurer et faciliter le passage en production, on peut utiliser des technologies comme le serveur du machine learning Open Source "PredictionIO".