

-Partie 1 : Présentation de Use Case

On va exploiter [un Dataset Open Source](#) qui continent 100K notes introduits par 1000 Utilisateurs sur 1700 Films afin de programmer un système qui va nous introduire les films similaires à un film donné , Le principe qu'on va suivre est : "Si la majorité des utilisateurs ayant vu les deux films ont donné deux notes similaires aux deux films : On juge que ces deux films sont similaires"

Par exemple : Si l'utilisateur "Metidji Sid Ahmed" a noté la série "Breaking Bad" avec 5/5 et la série "Better Call Saul" 4.5/5 : on peut juger que ces deux séries sont similaires (c'est à dire si un utilisateur apprécie l'un de ces deux : Il va fort probablement apprécier l'autre)

-Partie 2 : Présentation de processus suivi - Algorithme - :

Comme n'importe quel problème en Big Data , la résolution de ce processus va suivre une architecture MapReduce avec plusieurs Pipelines :

et Voici les étapes à suivre pour avoir les films similaires à un film prédéfini : "film1"

- On trouve les différents couples de films (film1 , filmX) qui ont été regardés par le même utilisateur , on nomme le couple de notes données à ces deux films par cet utilisateur par (rating1_0 , ratingX_0)
- Pour chaque couple de films (film1 , filmX) trouvé , On trouve tous les autres N utilisateurs qui ont regardé ce couple également , on aura donc N couple de notes (rating1_1 , ratingX_1) , (rating1_2 , rating X_2) , , (rating1_N , ratingX_N)
- On définit une fonction de similarité qui va mesurer le taux de similarité entre les notes données à le Film1 avec les notes données à Film 2 : c'est à dire on calcule la similarité entre rating1_0 , rating1_1 , rating1_2 , ... , rating1_N avec ratingX_0 , ratingX_1 , ratingX_2 , , ratingX_N . Cette fonction va nous donner comme un output ((film1 , filmX) , Similarity)
- On sort les résultats obtenues par le score de Similarity et on affiche les 10 films ayant le plus grand score de similarité avec le Film1

-Partie 3 : Présentation de processus suivi - Programation - :

L'algorithme présenté au dessus est un algorithme qui fonctionne parfaitement mais avant de l'implémenter , on doit la reformuler en suivant le paradigme de Spark

1.On Initialise l'environnement de Pyspark

```
conf = SparkConf().setMaster("local[*]").setAppName("MovieSimilarities")
sc = SparkContext(conf = conf)
```

- on a mis local[*] afin d'utiliser tous les coeurs de processeurs pour réduire le temps d'exécution

2. On lit la table qui contient les noms des films (afin de faire la correspondance entre MovieId et Movie Name) avec un script Python simple , Ensuite on commence le traitement en lisant le fichier u.data qui contient notre dataset

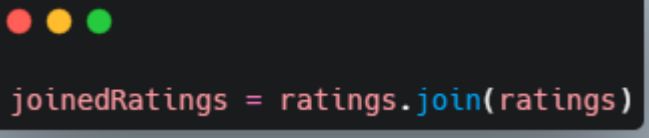
```
print("\nLoading movie names...")
nameDict = loadMovieNames()

data = sc.textFile("file:///SparkCourse/ml-100k/u.data")
```

3. on fait le Mapping de chaque note d'utilisateur à une entité (Key , Value) avec Key = userId et Value = (movieId , rating)

```
ratings = data.map(lambda l: l.split()).map(lambda l: (int(l[0]), (int(l[1]), float(l[2]))))
```


4. grâce à le Self-Join : On va obtenir tous les combinaisons possibles (deux à deux) de films notés par le même utilisateur



```
joinedRatings = ratings.join(ratings)
```

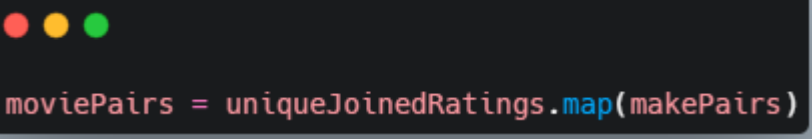
- output: `userId => ((film1 , rating1) , (film2 , rating2))`

5. Puisque l'ordre d'éléments de "Value" n'est pas important , on aura des duplications sur le résultat obtenu de Self-Join (on aura une entité (`userId , ((film1 , rating1) , (film2 , rating2))`) comme on aura (`userId , ((film2 , rating2) , (film1 , rating1))`). Donc on doit appeler une fonction de filtre pour se débarrasser de cette duplication



```
uniqueJoinedRatings = joinedRatings.filter(filterDuplicates)
```

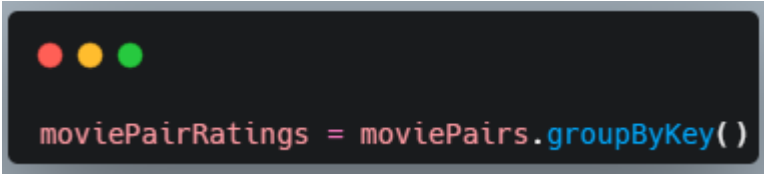
6. On appelle la fonction 'Map' à le résultat actuel afin de rendre le Key de nos données le couple (`film1 , film2`) au lieu de `UserId` , et la Value est (`rating1 , rating2`)



```
moviePairs = uniqueJoinedRatings.map(makePairs)
```

- output: `(film1 , film2) => (rating1 , rating2)`

7. on appelle la fonction GroupByKey afin de regrouper les données qui représentent les notes données à le même couple de films



```
moviePairRatings = moviePairs.groupByKey()
```

- output: (film1 , film2) => ((rating1 , rating2) , (rating1 , rating2) ,)

8. Maintenant , Pour chaque ligne On va appeler notre fonction qui calcule la similarité entre les films (film1, film2) en se basant sur ((rating1 , rating2) , (rating1 , rating2) ,) , et on la stocke dans la cache afin d'être exploiter par les différents noeuds sans avoir besoin de refaire le même calcul



```
moviePairSimilarities = moviePairRatings.mapValues(computeCosineSimilarity).cache()
```

output: (film1 , film2) => similarity

9. Etape optionnel: on peut stocker le output obtenu de 8eme résultat dans un fichier afin d'éviter de refaire les calculs faits dans tous ces étapes
10. Le programme principal: L'utilisateur va introduire le ID de movie qui l'a apprécié : movieID afin de le recommander des films similaires (on le donne les lignes Key,Value où (film1 = MovieID OU film2 = movieID) et on sort le résultat obtenu selon le score de similarité

```
# Extract similarities for the movie we care about that are "good".

scoreThreshold = 0.97
coOccurrenceThreshold = 50

movieID = int(input("Enter movie ID: "))

# Filter for movies with this sim that are "good" as defined by
# our quality thresholds above
filteredResults = moviePairSimilarities.filter(lambda pairSim: \
    (pairSim[0][0] == movieID or pairSim[0][1] == movieID) \
    and pairSim[1][0] > scoreThreshold and pairSim[1][1] > coOccurrenceThreshold)

# Sort by quality score.
results = filteredResults.map(lambda pairSim: (pairSim[1], pairSim[0])).sortByKey(ascending = False).take(10)

print("Top 10 similar movies for " + nameDict[movieID])
for result in results:
    (sim, pair) = result
    # Display the similarity result that isn't the movie we're looking at
    similarMovieID = pair[0]
    if (similarMovieID == movieID):
        similarMovieID = pair[1]
    print(nameDict[similarMovieID] + "\tscore: " + str(sim[0]) + "\tstrength: " + str(sim[1]))
print("--- %s seconds ---" % (time.time() - start_time))
```

-Partie 4 : Un exemple d'exécution

Notre utilisateur veut savoir les films similaires à le film “Star Wars”

```

Loading movie names...
Enter movie ID: 80
22/06/30 18:33:45 WARN ProcsMetricsGetter: Exception when trying to compute pagesize, as a result reporting of ProcessTree metrics is stopped

[Stage 1:=====] (2 + 2) / 4]

Top 10 similar movies for Star Wars (1977)
Empire Strikes Back, The (1980) score: 0.9895522078385338 strength: 345
Return of the Jedi (1983) score: 0.9857230861253026 strength: 480
Raiders of the Lost Ark (1981) score: 0.981760098872619 strength: 380
20,000 Leagues Under the Sea (1954) score: 0.9789385605497993 strength: 68
12 Angry Men (1957) score: 0.9776576120448436 strength: 109
Close Shave, A (1995) score: 0.9775948291054827 strength: 92
African Queen, The (1951) score: 0.9764692222674887 strength: 138
Sting, The (1973) score: 0.9751512937740359 strength: 204
Wrong Trousers, The (1993) score: 0.9748681355460885 strength: 103
Wallace & Gromit: The Best of Aardman Animation (1996) score: 0.9741816128302572 strength: 58
--- 88.03608059883118 seconds ---

Process finished with exit code 0

```

- Partie 5 : Reprendre le même exemple d'exécution sur un Cluster Spark déployé sur GCP (Google Cloud Platform) :