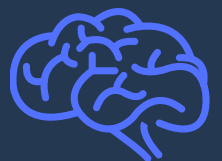


PROJET EN BIG DATA ANALYTICS

LE CAS DE LA DATASET ML-100K

REALISEE PAR :

HAJAR LACHHEB



RAPPORT
2021

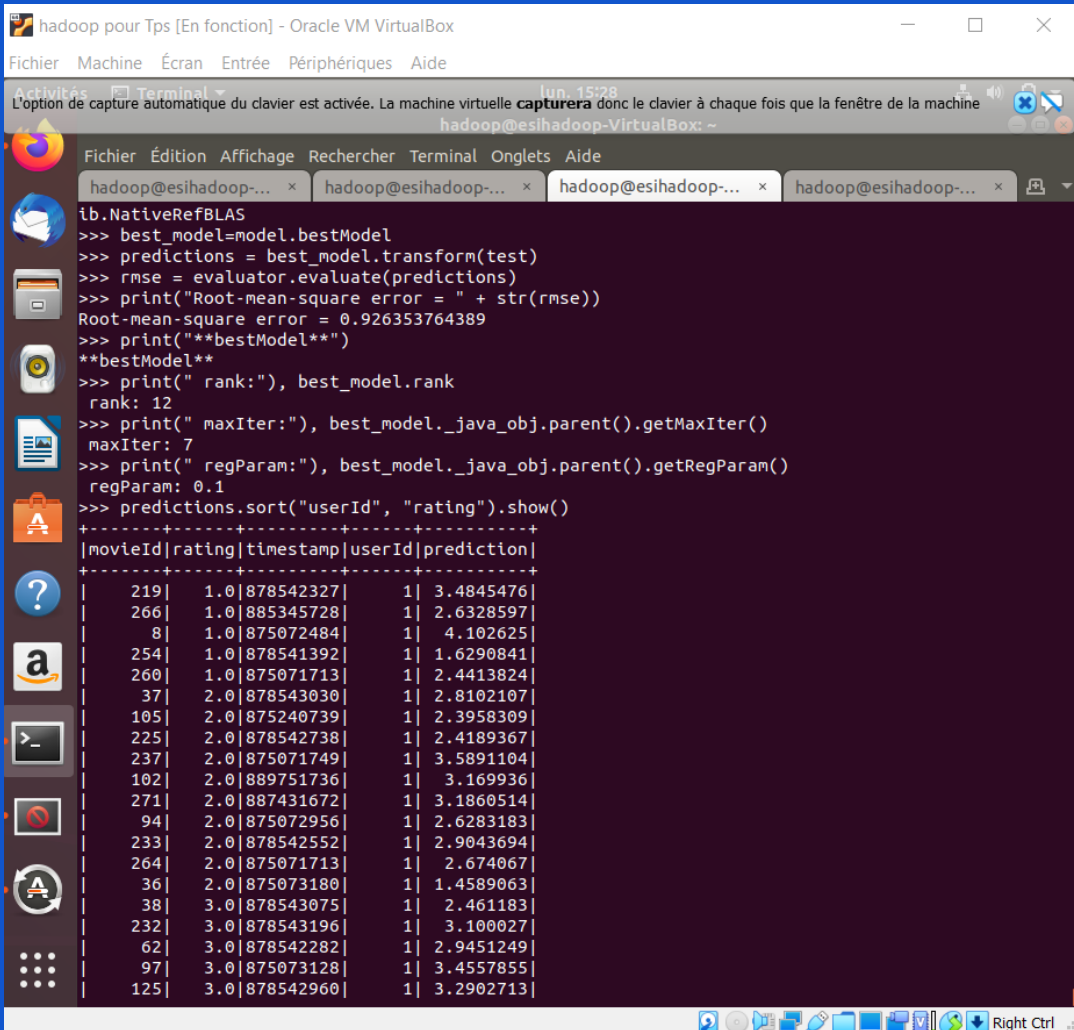
Projet en Big Data Analytics

Partie 1 : Problématique

Durant notre travail précédent afin de créer un système de recommandation qui va nous permettre de recommander des films en utilisant la dataset **ml-100k**. Nous avons opté pour un filtrage collaboratif, et c'est une méthode qui va nous permettre de faire des prédictions automatiques (filtrage) sur les intérêts d'un utilisateur en collectant les préférences ou les informations de goût de nombreux utilisateurs.

Nous avons utilisé comme modèle **l'ALS**. En effet, l'idée était essentiellement de prendre une grande matrice et de la factoriser dans une représentation plus petite de la matrice d'origine par le biais des moindres carrés alternés.

Nous allons nous retrouver en finale avec deux matrices de dimension inférieure ou plus dont le produit est égal à celui d'origine.



```
hadoop pour Tps [En fonction] - Oracle VM VirtualBox
Fichier Machine Écran Entrée Périphériques Aide
L'option de capture automatique du clavier est activée. La machine virtuelle capturera donc le clavier à chaque fois que la fenêtre de la machine
hadoop@esihadoop-VirtualBox: ~
ib.NativeRefBLAS
>>> best_model=model.bestModel
>>> predictions = best_model.transform(test)
>>> rmse = evaluator.evaluate(predictions)
>>> print("Root-mean-square error = " + str(rmse))
Root-mean-square error = 0.926353764389
>>> print("**bestModel**")
**bestModel**
>>> print(" rank:"), best_model.rank
rank: 12
>>> print(" maxIter:"), best_model._java_obj.parent().getMaxIter()
maxIter: 7
>>> print(" regParam:"), best_model._java_obj.parent().getRegParam()
regParam: 0.1
>>> predictions.sort("userId", "rating").show()
+-----+-----+-----+-----+-----+
|movieId|rating|timestamp|userId|prediction|
+-----+-----+-----+-----+
| 219| 1.0|878542327| 1| 3.4845476|
| 266| 1.0|885345728| 1| 2.6328597|
| 8| 1.0|875072484| 1| 4.102625|
| 254| 1.0|878541392| 1| 1.6290841|
| 260| 1.0|875071713| 1| 2.4413824|
| 37| 2.0|878543030| 1| 2.8102107|
| 105| 2.0|875240739| 1| 2.3958309|
| 225| 2.0|878542738| 1| 2.4189367|
| 237| 2.0|875071749| 1| 3.5891104|
| 102| 2.0|889751736| 1| 3.169936|
| 271| 2.0|887431672| 1| 3.1860514|
| 94| 2.0|875072956| 1| 2.6283183|
| 233| 2.0|878542552| 1| 2.9043694|
| 264| 2.0|875071713| 1| 2.674067|
| 36| 2.0|875073180| 1| 1.4589063|
| 38| 3.0|878543075| 1| 2.461183|
| 232| 3.0|878543196| 1| 3.100027|
| 62| 3.0|878542282| 1| 2.9451249|
| 97| 3.0|875073128| 1| 3.4557855|
| 125| 3.0|878542960| 1| 3.2902713|
```

Pour s'assurer de la bonne qualité et pertinence de notre prédiction, on utilise le RMSE. En effet, la racine de l'erreur quadratique moyenne est une mesure fréquemment utilisée des différences entre les valeurs prédites par un modèle ou estimateur et les valeurs observées.

On a eu comme résultat un **RMSE = 0.92** donc la différence entre la valeur réelle et celle prédite est **0.92** au niveau des votes. Ceci pourrait fausser notre série de "ratings" des films.

Donc la solution serait de réduire au maximum le RMSE afin d'avoir des résultats cohérents et les résultats tant voulus.

Partie 2 : Changement de paramètre

La première solution serait de changer les paramètres du **ALS**. J'ai donné au **ALS** un rank de **50** et **40** et un max itération de **10** et **12** et un lambda de **0.01** et **0.02**.

Le résultat du **RMSE** était alors de **0.98**. En essayant encore une fois de changer les paramètres, on finit par avoir des résultats compris entre **0.91** et **0.98**, ce qui ne correspond pas à nos attentes et nos objectifs.

La solution serait de chercher une autre alternative.

```
>>> (training, test) = ratings.randomSplit([0.8, 0.2])
>>> als = ALS(userCol="userId", itemCol="movieId", ratingCol="rating", coldStartStrategy="drop", nonnegative=True)
>>> param_grid=ParamGridBuilder().addGrid(als.rank,[50,40]).addGrid(als.maxIter,[10,12]).addGrid(als.regParam,[.01,.02,.03]).build()
>>> evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating", predictionCol="prediction")
>>> tvs= TrainValidationSplit(estimator=als, estimatorParamMaps=param_grid,evaluator=evaluator)
>>> model = tvs.fit(training)
21/12/25 18:31:12 WARN netlib.BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeSystemBLAS
21/12/25 18:31:12 WARN netlib.BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeRefBLAS
>>> best_model=model.bestModel
>>> predictions = best_model.transform(test)
>>> rmse = evaluator.evaluate(predictions)
>>> print("Root-mean-square error = " + str(rmse))
Root-mean-square error = 0.989773335234
```

Partie 3 : Solution Alternative en utilisant la similarité

- Explication du principe de la Cosine Similarity

Dans mon cas, je vais utiliser la Cosine Similarity. En effet, la Cosine Similarity est une méthode pour mesurer la différence entre deux vecteurs non nuls d'un espace produit.

Name	Iron Man (2008)	Pride & Prejudice (2005)
Bernard	4	3
Clarissa	5	5

Supposons qu'il faut vérifier si Bernard et Clarissa ont des préférences cinématographiques similaires et que je n'ai que deux critiques de films. Les critiques sont des scores de 1 à 5, où 5 est le meilleur

score et 1 le pire, et 0 signifie qu'une personne n'a pas regardé le film. Les vecteurs seraient alors représentés comme suit.

$$\vec{b} = \begin{bmatrix} 4 \\ 3 \end{bmatrix} \quad \vec{c} = \begin{bmatrix} 5 \\ 5 \end{bmatrix}$$

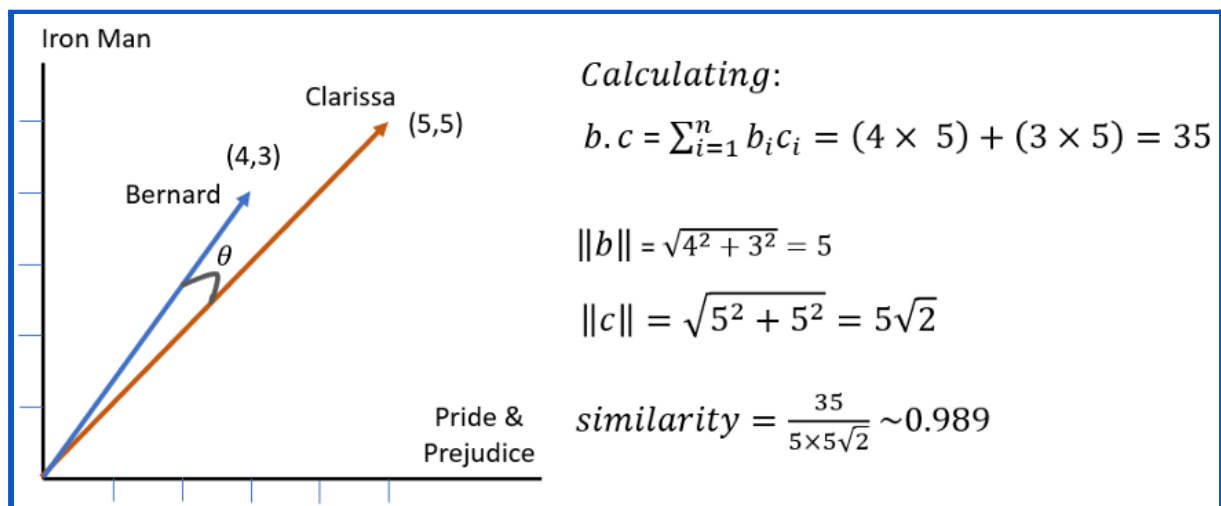
La similarité en cosinus mesurera la similarité entre ces deux vecteurs, ce qui est une mesure de la similarité des préférences entre ces deux personnes.

$$\text{similarity} = \cos \theta = \frac{b \cdot c}{\|b\| \|c\|}$$

$b \cdot c \Rightarrow$ Is the Dot product of the two vectors

$\|b\| \|c\| \Rightarrow$ Is the product of each vector's magnitude

Dans l'image, en dessous de chaque vecteur représente les préférences d'une personne et ils ont un angle θ entre eux. Les vecteurs similaires auront un angle **θ plus faible**, et les vecteurs dissemblables (différentes préférences de film) auront **un θ plus grand**.



→ La similitude 0,989 est proche de la valeur maximale de 1, cela signifie qu'étant donné que deux critiques de films seulement, les deux utilisateurs ont des préférences similaires.

Théoriquement, la similarité du cosinus peut être n'importe quel nombre entre -1 et +1 à cause de l'image de la fonction cosinus, mais dans ce cas, il n'y aura pas de note négative pour le film donc l'angle θ sera compris entre 0° et 90° délimitant le cosinus similarité entre 0 et 1. Si l'angle $\theta = 0^\circ \Rightarrow$ similarité cosinus = 1, si $\theta = 90^\circ \Rightarrow$ similarité cosinus = 0.

On commence par faire un start-all puis on utilise **SCALA** au lieu de **Pyspark**.

Scala a été utilisé vu sa similarité avec **Pyspark**, et c'était l'occasion afin d'essayer de comprendre la syntaxe de **Scala** et essayer de convertir notre code en **Pyspark**.

On lance un spark-shell – packages et on détermine le **JBLAS** package dont on a besoin. (Algèbre linéaire nécessaire pour effectuer notre cosine similarity).

```
hadoop@esihadoop-VirtualBox:~$ spark-shell --packages org.jblas:jblas:1.2.4
Ivy Default Cache set to: /home/hadoop/.ivy2/cache
The jars for the packages stored in: /home/hadoop/.ivy2/jars
:: loading settings :: url = jar:file:/usr/local/spark/jars/ivy-2.4.0.jar!/org/apache/ivy/core
/settings/ivysettings.xml
org.jblas#jblas added as a dependency
:: resolving dependencies :: org.apache.spark#spark-submit-parent-98cc704d-db4b-4d4d-869b-aa0d4969d792;1.0
  confs: [default]
  found org.jblas#jblas;1.2.4 in central
:: resolution report :: resolve 132ms :: artifacts dl 4ms
  :: modules in use:
  org.jblas#jblas;1.2.4 from central in [default]
-----
|               |          modules          ||   artifacts   |
|               | number| search|dwnlded|evicted|| number|dwnlded|
|-----|-----|-----|-----|-----|
| default      | 1    | 0    | 0    | 0    || 1    | 0    |
|-----|-----|-----|-----|-----|
:: retrieving :: org.apache.spark#spark-submit-parent-98cc704d-db4b-4d4d-869b-aa0d4969d792
  confs: [default]
  0 artifacts copied, 1 already retrieved (0kB/4ms)
21/12/25 19:34:59 WARN Util.Utills: Your hostname, esihadoop-VirtualBox resolves to a loopback
address: 127.0.0.1; using 10.0.2.15 instead (on interface enp0s3)
21/12/25 19:34:59 WARN util.Utills: Set SPARK_LOCAL_IP if you need to bind to another address
21/12/25 19:34:59 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your pl
atform... using builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Spark context Web UI available at http://10.0.2.15:4040
Spark context available as 'sc' (master = local[*], app id = local-1640457306198).
Spark session available as 'spark'.
Welcome to
```

- **Création de notre recommandation model**

On fait un import des données dont on a besoin. C'est similaire à la syntaxe de **pyspark**. On précise le chemin de notre **ml-100k/u.data** qui contient le **UserID**, **MovieID**, le **Rating** et le **Timestamp**. Nous pouvons visualiser le résultat de notre importation en utilisant le `rawData.first()`

```
val rawData = sc.textFile("/data/ml-100k/u.data")
rawData.first()
```

Dans notre cas, on ne va se contenter que des trois premières colonnes c'est a dire le **MovieID**, **UserID** et le **Rating**.

```
val rawRatings = rawData.map(_.split("\t").take(3))
rawRatings.first()
```

On fait les importations de modèle de recommandation **ALS** ainsi que le **Rating**.

```
import org.apache.spark.mllib.recommendation.ALS
```

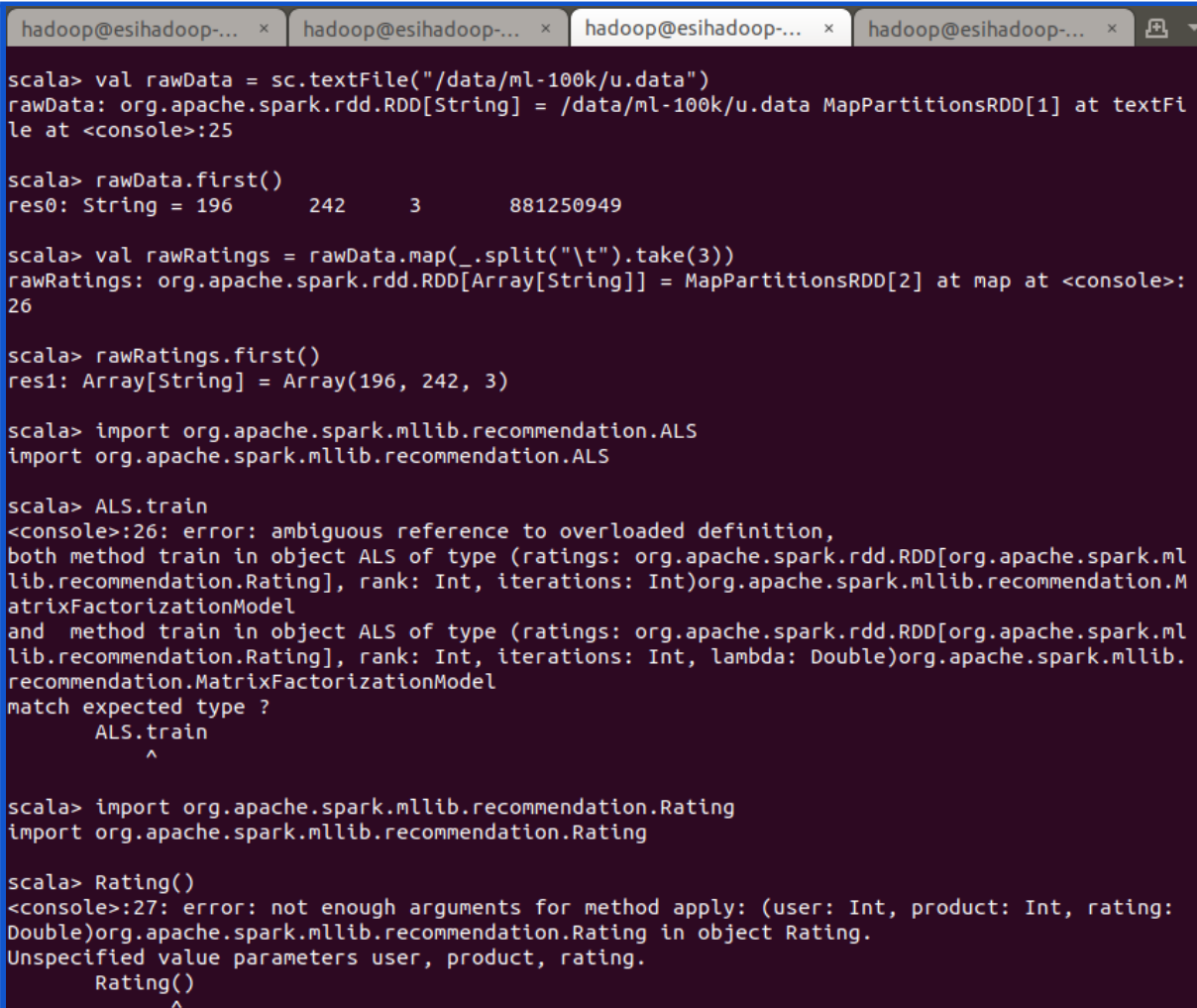


```
import org.apache.spark.mllib.recommendation.Rating
```

On précise après, tout comme **pyspark** les colonnes/attributs avec lesquelles on va travailler ainsi que leur types respectifs. Ceci devient alors un **RDD**.

```
val ratings = rawRatings.map { case Array(user, movie, rating) => Rating(user.toInt, movie.toInt, rating.toDouble) }
```

Ci-dessous le résultat de chaque requête mentionnée précédemment.



```
hadoop@esihadoop-... x hadoop@esihadoop-... x hadoop@esihadoop-... x hadoop@esihadoop-... x
scala> val rawData = sc.textFile("/data/ml-100k/u.data")
rawData: org.apache.spark.rdd.RDD[String] = /data/ml-100k/u.data MapPartitionsRDD[1] at textFile at <console>:25

scala> rawData.first()
res0: String = 196      242      3      881250949

scala> val rawRatings = rawData.map(_.split("\t").take(3))
rawRatings: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[2] at map at <console>:26

scala> rawRatings.first()
res1: Array[String] = Array(196, 242, 3)

scala> import org.apache.spark.mllib.recommendation.ALS
import org.apache.spark.mllib.recommendation.ALS

scala> ALS.train
<console>:26: error: ambiguous reference to overloaded definition,
both method train in object ALS of type (ratings: org.apache.spark.rdd.RDD[org.apache.spark.mllib.recommendation.Rating], rank: Int, iterations: Int)org.apache.spark.mllib.recommendation.MatrixFactorizationModel
and method train in object ALS of type (ratings: org.apache.spark.rdd.RDD[org.apache.spark.mllib.recommendation.Rating], rank: Int, iterations: Int, lambda: Double)org.apache.spark.mllib.recommendation.MatrixFactorizationModel
match expected type ?
    ALS.train
      ^

scala> import org.apache.spark.mllib.recommendation.Rating
import org.apache.spark.mllib.recommendation.Rating

scala> Rating()
<console>:27: error: not enough arguments for method apply: (user: Int, product: Int, rating: Double)org.apache.spark.mllib.recommendation.Rating in object Rating.
Unspecified value parameters user, product, rating.
    Rating()
      ^
```

On précise ci-dessous les paramètres de notre modèle **ALS**. Dans notre cas, le **rank** serait de **50**, les **itérations** de **10** et pour le **lambda**, il serait de **0.01**.

```
val model = ALS.train(ratings, 50, 10, 0.01)
```

Model.userFeatures nous renvoie un **RDD** apparié, où le premier élément est le user et le second est un tableau de features correspondant à ce user.

Alors que le **Model.productFeatures** nous renvoie un **RDD** apparié, où le premier élément est le produit et le second est un tableau de features correspondant à ce produit.

```
model.userFeatures
model.userFeatures.count
model.productFeatures.count
```

Pour un user dont l'**ID** est **789** et un Movie dont l'**ID** est **123**, quel est le rating prédit de ce user sur ce film ? Pour avoir le résultat, on utilise un **model.predict (789,123)**. On obtient alors un **rating** de **3.77/5**.

La méthode predict prend un **RDD** composée de **(user, item) IDs** en tant qu'input et va nous générer ainsi des prédictions pour chacun de ces éléments. On va utiliser cette méthode afin de réaliser des prédictions pour un grand nombre de users et d'items en même temps.

```
val predictedRating = model.predict(789, 123)
```

Pour générer le **top-10** de movie recommandée pour un user précis, le Modèle de la Matrix Factorization nous fournit une méthode pratique dont le nom est **recommendProducts**. Cette méthode prend deux arguments : user et num, ou le user est le user ID, et le num est le numéro des items à recommander.

Ceci va nous retourner comme résultat le top 10 des items triés par **ordre** en prenant en considération le **rating** prédit. Les **ratings** sont en fait calculés comme étant le **produit scalaire de l'user-factor vector et chaque item-factor vector**.

On voudrait avoir un top 10 des ratings de l'utilisateur dont l'**ID** est **789**. On reçoit une liste de movies IDs ainsi que leur ratings respectifs qui ont été prédit par notre modèle.

```
val userId = 789
val K = 10
val topKRecs = model.recommendProducts(userId, K)
println(topKRecs.mkString("\n"))
```

[Ci-dessous le résultat de chaque requête mentionnée précédemment.](#)

```

hadoop@esihadoop-... x hadoop@esihadoop-... x hadoop@esihadoop-... x hadoop@esihadoop-... x
scala> val ratings = rawRatings.map { case Array(user, movie, rating) => Rating(user.toInt, movie.toInt, rating.toDouble) }
ratings: org.apache.spark.rdd.RDD[org.apache.spark.mllib.recommendation.Rating] = MapPartitionsRDD[3] at map at <console>:28

scala> ratings.first()
res4: org.apache.spark.mllib.recommendation.Rating = Rating(196,242,3.0)

scala> val model = ALS.train(ratings, 50, 10, 0.01)
21/12/25 20:14:58 WARN netlib.BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeSystemBLAS
21/12/25 20:14:58 WARN netlib.BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeRefBLAS
21/12/25 20:14:58 WARN netlib.LAPACK: Failed to load implementation from: com.github.fommil.netlib.NativeSystemLAPACK
21/12/25 20:14:58 WARN netlib.LAPACK: Failed to load implementation from: com.github.fommil.netlib.NativeRefLAPACK
model: org.apache.spark.mllib.recommendation.MatrixFactorizationModel = org.apache.spark.mllib.recommendation.MatrixFactorizationModel@66968a15

scala> model.userFeatures
res5: org.apache.spark.rdd.RDD[(Int, Array[Double])] = users MapPartitionsRDD[209] at mapValues at ALS.scala:271

scala> model.userFeatures.count
res6: Long = 943

scala> model.productFeatures.count
res7: Long = 1682

scala> val predictedRating = model.predict(789, 123)
predictedRating: Double = 3.7757464325316596

```

```

hadoop@esihadoop-... x hadoop@esihadoop-... x hadoop@esihadoop-... x hadoop@esihadoop-... x
scala> val userId = 789
userId: Int = 789

scala> val K = 10
K: Int = 10

scala> val topKRecs = model.recommendProducts(userId, K)
topKRecs: Array[org.apache.spark.mllib.recommendation.Rating] = Array(Rating(789,56,6.352680147508943), Rating(789,346,6.2944313425488385), Rating(789,135,5.538795444555381), Rating(789,922,5.4605549610515665), Rating(789,175,5.365779481713565), Rating(789,177,5.357688721848327), Rating(789,789,5.255166338447388), Rating(789,32,5.198282246691683), Rating(789,502,5.1481956043256965), Rating(789,101,5.142164106541459))

scala> println(topKRecs.mkString("\n"))
Rating(789,56,6.352680147508943)
Rating(789,346,6.2944313425488385)
Rating(789,135,5.538795444555381)
Rating(789,922,5.4605549610515665)
Rating(789,175,5.365779481713565)
Rating(789,177,5.357688721848327)
Rating(789,789,5.255166338447388)
Rating(789,32,5.198282246691683)
Rating(789,502,5.1481956043256965)
Rating(789,101,5.142164106541459)

scala> val movies = sc.textFile("/data/ml-100k/u.item")
movies: org.apache.spark.rdd.RDD[String] = /data/ml-100k/u.item MapPartitionsRDD[214] at textFile at <console>:27

```

Afin de donner plus de sens à notre prédiction et réduire notre **RMSE**, on va ajouter le u.item qui se trouve au niveau de la ml-100k et contient les titres de nos movies.

Nous avons alors besoin de charger les données des movies en utilisant le `sc.TextFile` et précisant le chemin de notre file. Nous opterons pour une collecte des titres en utilisant la méthode **Map[Int, String]** ce qui va nous aider à mapper le MovieID au titre du Movie :


```
val movies = sc.textFile("/data/ml-100k/u.item")
val titles = movies.map(line => line.split("\\|").take(2)).map(array => (array(0).toInt,
array(1))).collectAsMap()
titles(123)
```

Le **titles(123)** nous fait afficher le **title** du movie dont l'**ID** est **123**. Ceci nous donne un résultat de notre mappage réussi.

[Ci-dessous le résultat de chaque requête mentionnée précédemment.](#)

```
scala> val titles = movies.map(line => line.split("\\|").take(2)).map(array => (array(0).toInt,
array(1))).collectAsMap()
titles: scala.collection.Map[Int,String] = Map(137 -> Big Night (1996), 891 -> Bent (1997), 550 -> Die Hard: With a Vengeance (1995), 1205 -> Secret Agent, The (1996), 146 -> Unhook the Stars (1996), 864 -> My Fellow Americans (1996), 559 -> Interview with the Vampire (1994), 218 -> Cape Fear (1991), 568 -> Speed (1994), 227 -> Star Trek VI: The Undiscovered Country (1991), 765 -> Boomerang (1992), 1115 -> Twelfth Night (1996), 774 -> Prophecy, The (1995), 433 -> Heathers (1989), 92 -> True Romance (1993), 1528 -> Nowhere (1997), 846 -> To Gillian on Her 37th Birthday (1996), 1187 -> Switchblade Sisters (1975), 1501 -> Prisoner of the Mountains (Kavkazsky Plennik) (1996), 442 -> Amityville Curse, The (1990), 1160 -> Love! Valour! Compassion!
```

```
scala> titles(123)
res9: String = Frighteners, The (1996)
```

Précédemment, nous avons eu le **top10** des recommandations de l'utilisateur dont l'ID est 789. Maintenant, pour notre utilisateur **789**, nous pouvons savoir quels films ils ont évalués et prendre que les 10 films avec la note la plus élevée.

On va utiliser la fonction **keyBy Spark** pour créer un **RDD** de paires **key-value** à partir de nos ratings, où la key sera le UserID. Nous utiliserons ensuite la fonction **lookup** pour avoir que les ratings de cet utilisateur (which is our key).

```
val moviesForUser = ratings.keyBy(_._user).lookup(789)
```

Avec le **moviesForUser.size**, on pourrait avoir le nombre de movies que l'utilisateur a voté sur. Dans notre cas, l'utilisateur a pu donner son rating sur **33** movies différents.

```
println(moviesForUser.size)
```

Ensuite, on va prendre que les **10 movies** avec le rating le plus **élevé** en les triant.

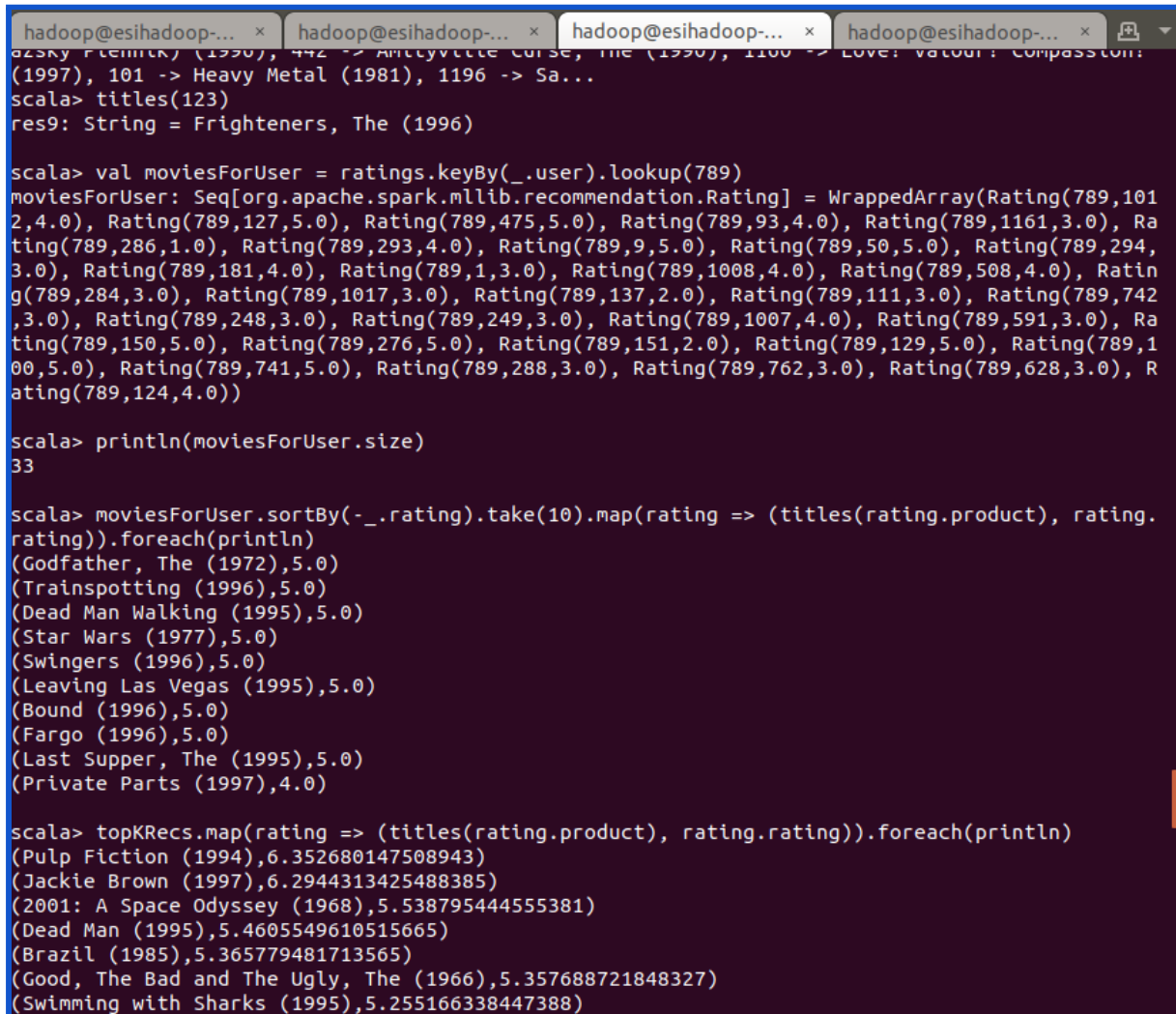
On va après extraire les titres des movies correspondant à l'ID des movies dont l'utilisateur **789** a fait ses ratings. Cette extraction va se faire à partir de titles (qui contient un ID et le nom du movie). Et comme résultat on va faire un mapping pour avoir que le titre du Movie et son rating fait par l'utilisateur **789**.

```
moviesForUser.sortBy(_._rating).take(10).map(rating => (titles(rating._1),
rating._2)).foreach(println)
```

Le **topKRecs** c'est le **top 10** des ratings **PRÉDIT** par le modèle dans le cas de l'utilisateur dont l'ID est 789. On remarque que la différence entre ce qui a été prédit et le résultat réel est trop flagrante.

```
topKRecs.map(rating => (titles(rating.product), rating.rating)).foreach(println)
```

[Ci-dessous le résultat de chaque requête mentionnée précédemment.](#)



```
hadoop@esihadoop-... x hadoop@esihadoop-... x hadoop@esihadoop-... x hadoop@esihadoop-... x
azsky Plennik) (1990), 442 -> Amityville Curse, The (1990), 1100 -> Love: Valour: Compassion:
(1997), 101 -> Heavy Metal (1981), 1196 -> Sa...
scala> titles(123)
res9: String = Frighteners, The (1996)

scala> val moviesForUser = ratings.keyBy(_.user).lookup(789)
moviesForUser: Seq[org.apache.spark.mllib.recommendation.Rating] = WrappedArray(Rating(789,101
2,4.0), Rating(789,127,5.0), Rating(789,475,5.0), Rating(789,93,4.0), Rating(789,1161,3.0), Ra
ting(789,286,1.0), Rating(789,293,4.0), Rating(789,9,5.0), Rating(789,50,5.0), Rating(789,294,
3.0), Rating(789,181,4.0), Rating(789,1,3.0), Rating(789,1008,4.0), Rating(789,508,4.0), Ratin
g(789,284,3.0), Rating(789,1017,3.0), Rating(789,137,2.0), Rating(789,111,3.0), Rating(789,742
,3.0), Rating(789,248,3.0), Rating(789,249,3.0), Rating(789,1007,4.0), Rating(789,591,3.0), Ra
ting(789,150,5.0), Rating(789,276,5.0), Rating(789,151,2.0), Rating(789,129,5.0), Rating(789,1
00,5.0), Rating(789,741,5.0), Rating(789,288,3.0), Rating(789,762,3.0), Rating(789,628,3.0), R
ating(789,124,4.0))

scala> println(moviesForUser.size)
33

scala> moviesForUser.sortBy(-_.rating).take(10).map(rating => (titles(rating.product), rating.
rating)).foreach(println)
(Godfather, The (1972),5.0)
(Trainspotting (1996),5.0)
(Dead Man Walking (1995),5.0)
(Star Wars (1977),5.0)
(Swingers (1996),5.0)
(Leaving Las Vegas (1995),5.0)
(Bound (1996),5.0)
(Fargo (1996),5.0)
(Last Supper, The (1995),5.0)
(Private Parts (1997),4.0)

scala> topKRecs.map(rating => (titles(rating.product), rating.rating)).foreach(println)
(Pulp Fiction (1994),6.352680147508943)
(Jackie Brown (1997),6.2944313425488385)
(2001: A Space Odyssey (1968),5.538795444555381)
(Dead Man (1995),5.4605549610515665)
(Brazil (1985),5.365779481713565)
(Good, The Bad and The Ugly, The (1966),5.357688721848327)
(Swimming with Sharks (1995),5.255166338447388)
```

Notre modèle de Factorisation de matrices ne supporte pas directement les calculs de similarité entre items. On va utiliser alors la métrique de similarité cosinus, et nous utiliserons l'algèbre linéaire jblas pour calculer les produits scalaires vectoriels.

En effet, il est à préciser que notre méthode est similaire à la façon dont fonctionnent les méthodes de prédiction et de recommandation, à l'exception que nous utiliserons la similarité en cosinus plutôt que le produit scalaire.

On va essayer de comparer le factor vector de notre item (qu'on va choisir) avec d'autres items, en utilisant la métrique de similarité. Afin de performer les calculs d'algèbre linéaire, on a besoin de créer un vector object à partir des factor vectors, qui sont sous la forme d'`Array[Double]`. La classe **JBLAS**, **DoubleMatrix**, prennent comme input un **Array[Double]**.

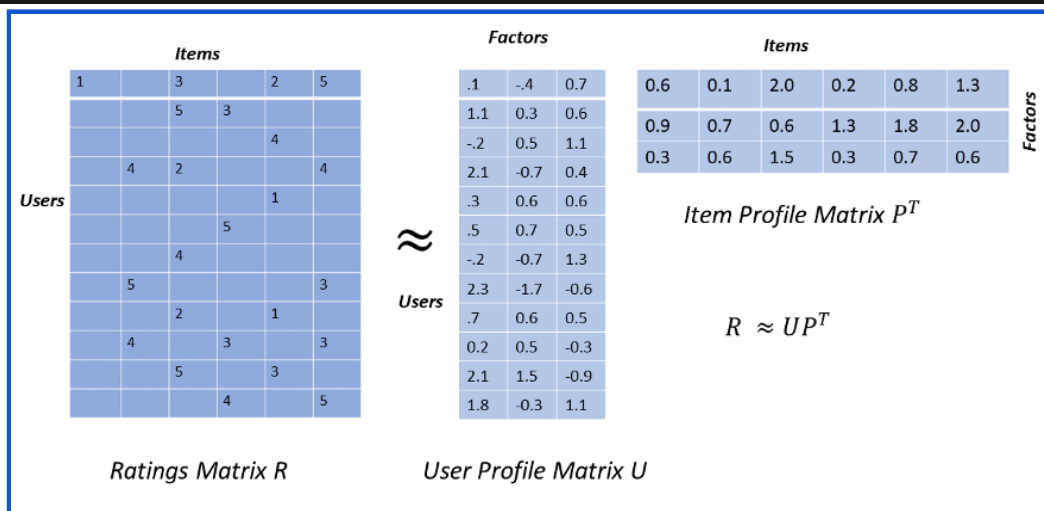
```
import org.jblas.DoubleMatrix
```

Afin d'utiliser le **JBLAS**, il faut télécharger le jar et l'intégrer dans le Spark Path.

```
val aMatrix = new DoubleMatrix(Array(1.0, 2.0, 3.0))
```

La **Cosine Similarity** est calculé en calculant d'abord le **produit scalaire entre les vecteurs** et puis **diviser le résultat par un dénominateur**, qui est la norme de chaque vecteur multiplié ensemble. La fonction définie ci-dessous nous permet de calculer la Cosine Similarity.

```
def cosineSimilarity(vec1: DoubleMatrix, vec2: DoubleMatrix): Double = {  
    vec1.dot(vec2) / (vec1.norm2() * vec2.norm2())  
}
```



```
val itemId = 567
```

On se fixe un **ItemID = 567** afin d'y chercher le **itemFactor** et constituer par la suite un **itemVector**. On va collecter un **item factor** a partir de notre modèle. On va le faire en utilisant la méthode **lookup**. On va aussi utiliser la fonction **head** pour avoir que la première valeur qui sera le **factor vector** de notre **item**.

```
val itemFactor = model.productFeatures.lookup(itemId).head
```

Le résultat est un Array, d'où la nécessité d'utiliser la DoubleMatrix pour avoir une matrice.

```
val itemVector = new DoubleMatrix(itemFactor)
```

```

hadoop@esihadoop-... x hadoop@esihadoop-... x hadoop@esihadoop-... x hadoop@esihadoop-... x
(0.0211 (1993),5.150371348113383)
(Good, The Bad and The Ugly, The (1966),5.357688721848327)
(Swimming with Sharks (1995),5.255166338447388)
(Crumb (1994),5.198282246691683)
(Bananas (1971),5.1481956043256965)
(Heavy Metal (1981),5.142164106541459)

scala> import org.jblas.DoubleMatrix
import org.jblas.DoubleMatrix

scala> val aMatrix = new DoubleMatrix(Array(1.0, 2.0, 3.0))
aMatrix: org.jblas.DoubleMatrix = [1,000000; 2,000000; 3,000000]

scala> def cosineSimilarity(vec1: DoubleMatrix, vec2: DoubleMatrix): Double = {
  |   vec1.dot(vec2) / (vec1.norm2() * vec2.norm2())
  | }
cosineSimilarity: (vec1: org.jblas.DoubleMatrix, vec2: org.jblas.DoubleMatrix)Double

scala> val itemId = 567
itemId: Int = 567

scala> val itemFactor = model.productFeatures.lookup(itemId).head
itemFactor: Array[Double] = Array(-0.06750568747520447, 0.604738712310791, 0.12517742812633514
, -0.7440899610519409, -0.17012329399585724, -0.060635898262262344, 0.9445353746414185, -0.705
9495449066162, -0.21000833809375763, -0.46101129055023193, -0.9763296246528625, 0.657725572586
0596, 1.0560203790664673, -0.24074159562587738, 0.19662274420261383, 0.8822855353355408, 0.286
12953424453735, 0.6332834959030151, -0.31904375553131104, 0.9555342793464661, -0.4609279334545
1355, 0.15662050247192383, 0.4882694184780121, -0.23693415522575378, -1.048483967781067, 0.058
80000442266464, 0.7558631896972656, 0.7122653126716614, -0.07980716973543167, -0.2773748934268
9514, -0.3068051040172577, 0.8087260127067566, 0.18820466101169586, -0.6435108780860901, 0.460
7091248035431, -0.5548925399780273, 0.6010468...)

scala> val itemVector = new DoubleMatrix(itemFactor)
itemVector: org.jblas.DoubleMatrix = [-0,067506; 0,604739; 0,125177; -0,744090; -0,170123; -0,
060636; 0,944535; -0,705950; -0,210008; -0,461011; -0,976330; 0,657726; 1,056020; -0,240742; 0
,196623; 0,882286; 0,286130; 0,633283; -0,319044; 0,955534; -0,460928; 0,156621; 0,488269; -0,
236934; -1,048484; 0,058800; 0,755863; 0,712265; -0,079807; -0,277375; -0,306805; 0,808726; 0,
188205; -0,643511; 0,460709; -0,554893; 0,601047; -0,006951; -0,328337; 0,502880; -0,473982; 0
,086382; -0,187388; -0,721613; 0,330022; 1,555888; -0,489314; -0,614487; 0,158088; -0,749333]

scala> cosineSimilarity(itemVector, itemVector)

```

On calcule la similarité de **itemVector** de l'ID en question avec le même **itemVector**. Il faut avoir un résultat de **1** vu qu'il s'agit des mêmes **itemsVectors**, sinon notre fonction **CosineSimilarity** est faussée.

```
cosineSimilarity(itemVector, itemVector)
```

On va maintenant appliquer notre **cosineSimilarity** sur chaque item disponible en prenant en compte l'**itemVector** liée à notre **ID** choisi. On va comparer les vecteur des items avec celui de notre item.

```

val sims = model.productFeatures.map{ case (id, factor) =>
  |   val factorVector = new DoubleMatrix(factor)
  |   val sim = cosineSimilarity(factorVector, itemVector)
  |   (id, sim)}

```

On fait un tri du résultat des **10 premiers résultats similaire** à notre **ItemID**. Le **tri** se fera en prenant en considération le résultat de **similarité**. De manière à avoir comme résultat les **10 items les plus similaires** à notre item de base.

```
val sortedSims = sims.top(K)(Ordering.by[(Int, Double), Double] { case (id, similarity) => similarity })
```

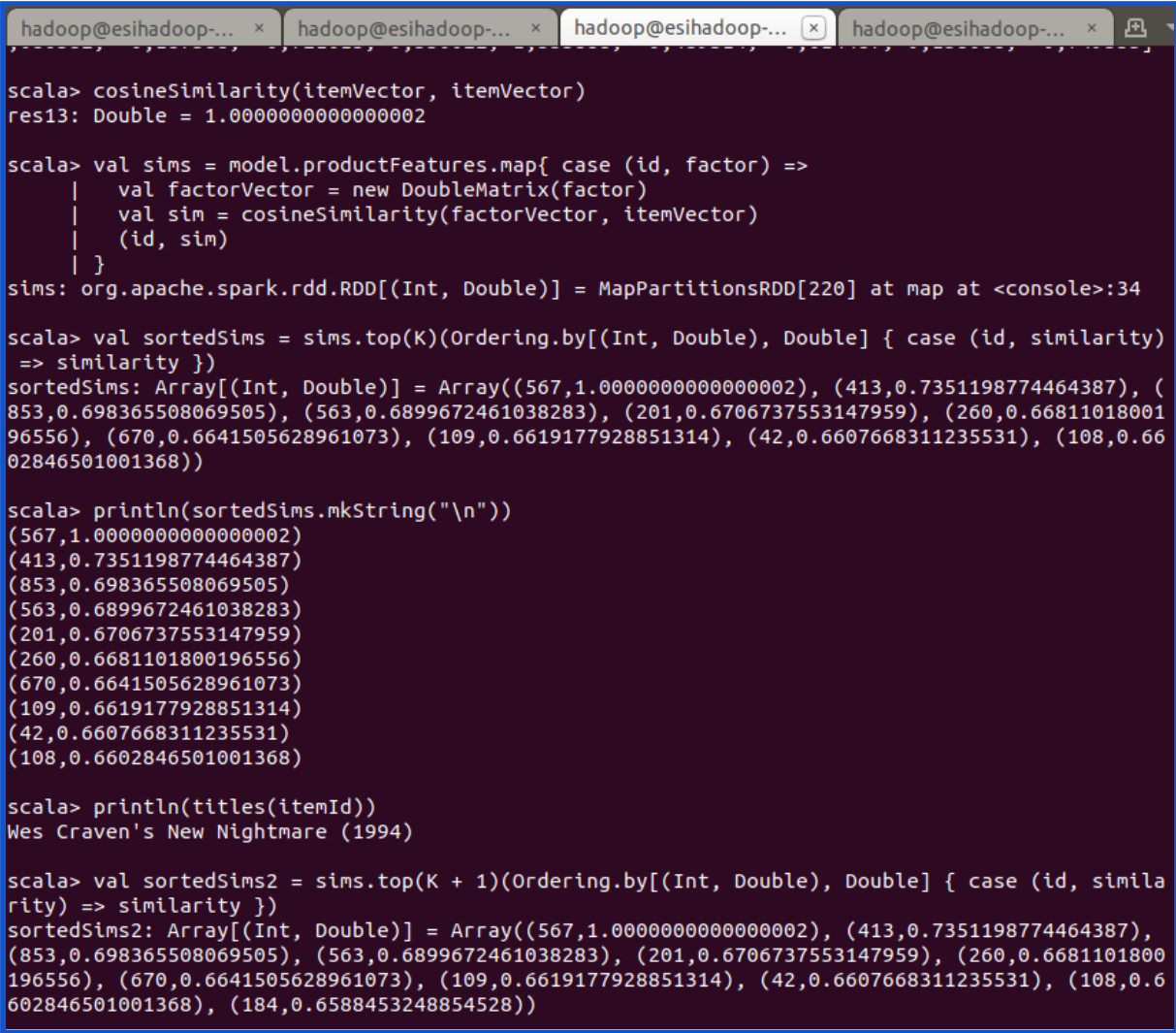
```
println(titles(itemId))
```

Si on souhaite retirer notre **itemID** des résultats affichés, on va utiliser un **K + 1**, de manière à avoir que les résultats à partir de **1** jusqu'à **11**.

Tous les résultats affichés sont compris entre **0.73** et **0.66**, ce qui reflète la **similarité** des items avec celui qu'on lui a donné en premier. Si la similarité s'avère **négative**, ceci pourrait nous prouver qu'il n'y a **aucune similarité entre les items**.

```
val sortedSims2 = sims.top(K + 1)(Ordering.by[(Int, Double), Double] { case (id, similarity) => similarity })
```

[Ci-dessous le résultat de chaque requête mentionnée précédemment.](#)



```
scala> cosineSimilarity(itemVector, itemVector)
res13: Double = 1.0000000000000002

scala> val sims = model.productFeatures.map{ case (id, factor) =>
  |   val factorVector = new DoubleMatrix(factor)
  |   val sim = cosineSimilarity(factorVector, itemVector)
  |   (id, sim)
  | }
sims: org.apache.spark.rdd.RDD[(Int, Double)] = MapPartitionsRDD[220] at map at <console>:34

scala> val sortedSims = sims.top(K)(Ordering.by[(Int, Double), Double] { case (id, similarity)
=> similarity })
sortedSims: Array[(Int, Double)] = Array((567,1.0000000000000002), (413,0.7351198774464387), (
853,0.698365508069505), (563,0.6899672461038283), (201,0.6706737553147959), (260,0.66811018001
96556), (670,0.6641505628961073), (109,0.6619177928851314), (42,0.6607668311235531), (108,0.66
02846501001368))

scala> println(sortedSims.mkString("\n"))
(567,1.0000000000000002)
(413,0.7351198774464387)
(853,0.698365508069505)
(563,0.6899672461038283)
(201,0.6706737553147959)
(260,0.6681101800196556)
(670,0.6641505628961073)
(109,0.6619177928851314)
(42,0.6607668311235531)
(108,0.6602846501001368)

scala> println(titles(itemId))
Wes Craven's New Nightmare (1994)

scala> val sortedSims2 = sims.top(K + 1)(Ordering.by[(Int, Double), Double] { case (id, simila
rity) => similarity })
sortedSims2: Array[(Int, Double)] = Array((567,1.0000000000000002), (413,0.7351198774464387),
(853,0.698365508069505), (563,0.6899672461038283), (201,0.6706737553147959), (260,0.6681101800
196556), (670,0.6641505628961073), (109,0.6619177928851314), (42,0.6607668311235531), (108,0.6
602846501001368), (184,0.6588453248854528))
```

On fait un autre **tri** mais de manière à ce qu'on remplace les **IDs** des **movies** par les **titres** des movies. On prend l' **ID** qu'on a eu comme résultat et on l'associe aux id au niveau de notre table titles.


```
sortedSims2.slice(1, 11).map{ case (id, sim) => (titles(id), sim) }.mkString("\n")
```

- Evaluation de notre modèle et de sa performance

Après avoir eu les résultats tant souhaités, il est maintenant temps de mesurer la performance de notre modèle.

On définit d'abord un **Actual Rating** et un **Predicted Rating**. On va prendre dans notre cas, le user dont l'**ID** est **789** et on va utiliser la **moviesForUser** qu'on a défini précédemment afin de définir le vrai rating.

Dans le cas du predicted Rating, on va utiliser notre modèle tout en précisant l'Id de l'utilisateur et le produit qu'il faut qu'on vote. Dans notre cas, c'est le **movie** dont l'ID est **1012**.

```
val actualRating = moviesForUser.take(1)(0)
```

```
val predictedRating = model.predict(789, actualRating.product)
```

On obtient un **résultat prédit** de **3.93** qui est très proche du **résultat du rating réel** qui est de **4**.

ID du movie	Rating Réel	Rating Prédit
1012	4.0	3.93

```
val squaredError = math.pow(predictedRating - actualRating.rating, 2.0)
```

SquaredError nous donne un bon résultat aussi, mais ceci s'applique que pour un seul utilisateur et un seul item. Pour avoir un **MSE général** et qui englobe toute la dataset, il faut répéter les mêmes étapes pour tous les éléments de la dataset.

```
val usersProducts = ratings.map{ case Rating(user, product, rating) => (user, product)}
```

Nous devons calculer cette **erreur quadratique** pour chaque (**User, Movie, Real Rating, Predicted Rating**).

Tout d'abord, nous allons extraire les **UserID** et **ProductsID** de notre **RDD** et faire des prédictions pour chaque paire **user-item** en utilisant notre `model.predict`. Nous utiliserons l'élément paire user-item comme key et le rating prédit comme value.

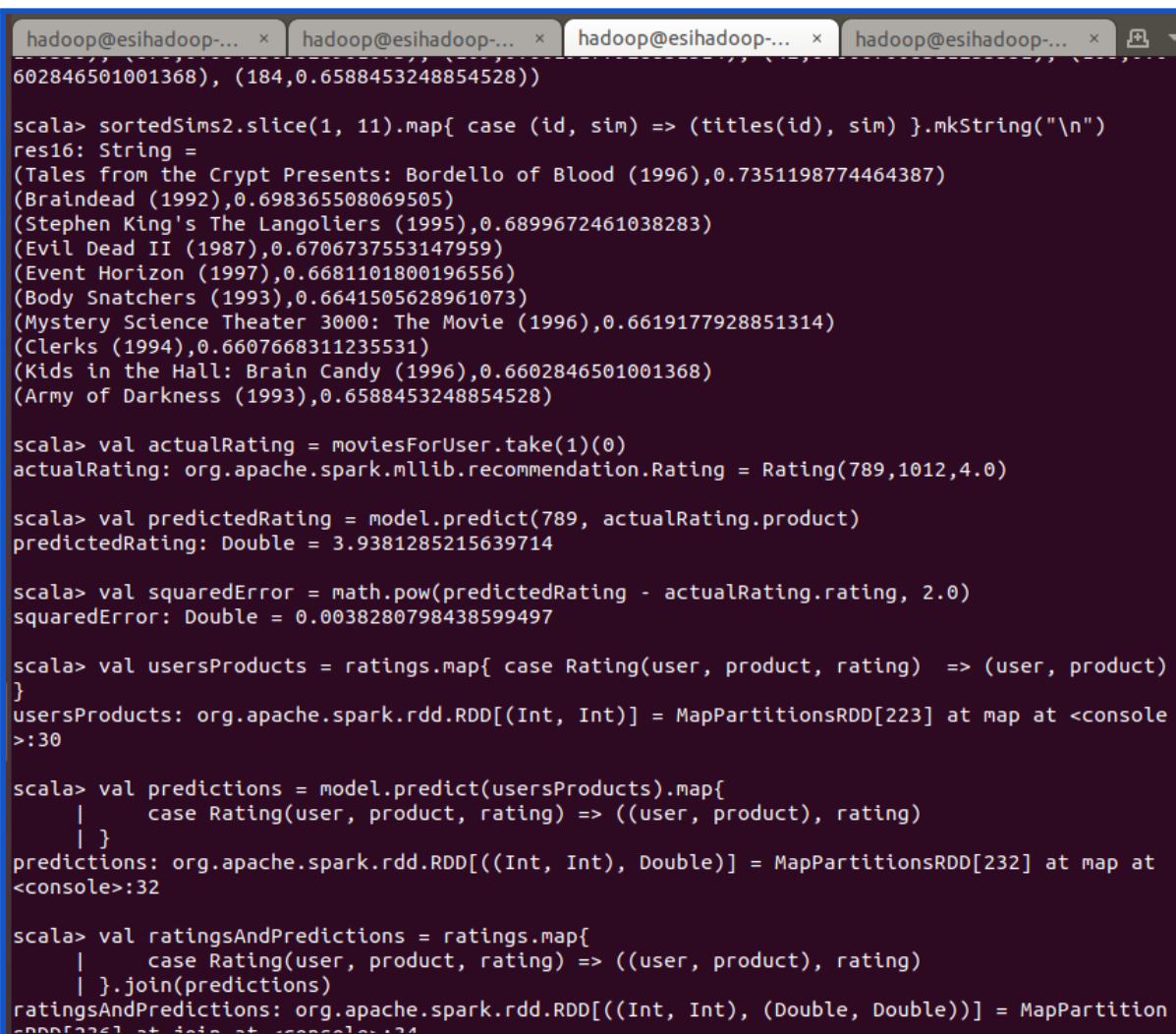
```
val predictions = model.predict(usersProducts).map{  
  case Rating(user, product, rating) => ((user, product), rating)  
}
```

Nous allons ensuite extraire les Ratings réels et faire un mapping entre les ratings de la RDD afin que les paires user-item deviennent un key et le rating réel est la valeur. On a maintenant **deux RDDs** créés, on pourrait les combiner vu qu'ils ont la même **forme de key** mais des **values différentes**.

On aura ainsi **un RDD** qui combine les paires **User-Item**, le **rating prédit** et le **rating réel**.

```
val ratingsAndPredictions = ratings.map{
  case Rating(user, product, rating) => ((user, product), rating)
}.join(predictions)
```

Ci-dessous le résultat de chaque requête mentionnée précédemment.



```
hadoop@esihadoop-... x hadoop@esihadoop-... x hadoop@esihadoop-... x hadoop@esihadoop-... x
602846501001368), (184,0.6588453248854528))

scala> sortedSims2.slice(1, 11).map{ case (id, sim) => (titles(id), sim) }.mkString("\n")
res16: String =
(Tales from the Crypt Presents: Bordello of Blood (1996),0.7351198774464387)
(Braindead (1992),0.698365508069505)
(Stephen King's The Langoliers (1995),0.6899672461038283)
(Evil Dead II (1987),0.6706737553147959)
(Event Horizon (1997),0.6681101800196556)
(Body Snatchers (1993),0.6641505628961073)
(Mystery Science Theater 3000: The Movie (1996),0.6619177928851314)
(Clerks (1994),0.6607668311235531)
(Kids in the Hall: Brain Candy (1996),0.6602846501001368)
(Army of Darkness (1993),0.6588453248854528)

scala> val actualRating = moviesForUser.take(1)(0)
actualRating: org.apache.spark.mllib.recommendation.Rating = Rating(789,1012,4.0)

scala> val predictedRating = model.predict(789, actualRating.product)
predictedRating: Double = 3.9381285215639714

scala> val squaredError = math.pow(predictedRating - actualRating.rating, 2.0)
squaredError: Double = 0.0038280798438599497

scala> val usersProducts = ratings.map{ case Rating(user, product, rating) => (user, product)
}
usersProducts: org.apache.spark.rdd.RDD[(Int, Int)] = MapPartitionsRDD[223] at map at <console>:30

scala> val predictions = model.predict(usersProducts).map{
  |   case Rating(user, product, rating) => ((user, product), rating)
  | }
predictions: org.apache.spark.rdd.RDD[((Int, Int), Double)] = MapPartitionsRDD[232] at map at <console>:32

scala> val ratingsAndPredictions = ratings.map{
  |   case Rating(user, product, rating) => ((user, product), rating)
  | }.join(predictions)
ratingsAndPredictions: org.apache.spark.rdd.RDD[((Int, Int), (Double, Double))] = MapPartitionsRDD[233] at join at <console>:34
```

On calcule maintenant le **MSE** en ajoutant tous les **MSEs** de chaque **items** en utilisant la méthode **Reduce**, et on divise le tout après sur le **count** des records au niveau de notre **RDD**.

```
val MSE = ratingsAndPredictions.map{
  case ((user, product), (actual, predicted)) => math.pow((actual - predicted), 2)
}.reduce(_ + _) / ratingsAndPredictions.count
```

```
println("Mean Squared Error = " + MSE)
```

Le **RMSE** est la **racine carrée du MSE** qui est dans notre cas égal à : **0.08**.

Notre RMSE est ainsi égal à : 0.289862

```
scala> println("Root Mean Squared Error = " + RMSE)
Root Mean Squared Error = 0.28986262996892626
```

```
val RMSE = math.sqrt(MSE)
println("Root Mean Squared Error = " + RMSE)
```

Ci-dessous le résultat de chaque requête mentionnée précédemment.

```
hadoop@esihadoop-... x hadoop@esihadoop-... x hadoop@esihadoop-... x hadoop@esihadoop-... x
scala> val predictedRating = model.predict(789, actualRating.product)
predictedRating: Double = 3.9381285215639714

scala> val squaredError = math.pow(predictedRating - actualRating.rating, 2.0)
squaredError: Double = 0.0038280798438599497

scala> val usersProducts = ratings.map{ case Rating(user, product, rating) => (user, product)
}
usersProducts: org.apache.spark.rdd.RDD[(Int, Int)] = MapPartitionsRDD[223] at map at <console>:30

scala> val predictions = model.predict(usersProducts).map{
|   case Rating(user, product, rating) => ((user, product), rating)
| }
predictions: org.apache.spark.rdd.RDD[((Int, Int), Double)] = MapPartitionsRDD[232] at map at <console>:32

scala> val ratingsAndPredictions = ratings.map{
|   case Rating(user, product, rating) => ((user, product), rating)
| }.join(predictions)
ratingsAndPredictions: org.apache.spark.rdd.RDD[((Int, Int), (Double, Double))] = MapPartitionsRDD[236] at join at <console>:34

scala> val MSE = ratingsAndPredictions.map{
|   case ((user, product), (actual, predicted)) => math.pow((actual - predicted), 2)
| }.reduce(_ + _) / ratingsAndPredictions.count
MSE: Double = 0.08402034425250267

scala> println("Mean Squared Error = " + MSE)
Mean Squared Error = 0.08402034425250267

scala> val RMSE = math.sqrt(MSE)
RMSE: Double = 0.28986262996892626

scala> println("Root Mean Squared Error = " + RMSE)
Root Mean Squared Error = 0.28986262996892626
```

→ On a pu réduire le **RMSE** initial avec succès. Une réduction d'une valeur de 0,7.

La différence entre le rating réel et le rating prédit est uniquement un 0.28 ou presque un 0.3 ce qui est peu.