İbrahim Can Gençel
2166429

# CENG 790 BIG DATA ANALYTICS
# ASSIGNMENT II - Recommender Systems

## Part 1: Collaborative Filtering

Files : partI.scala - ALSParameterTuning.scala - collaborative_filtering.scala

**1.1)** Change the values for ranks, lambdas and numIters and create the cross product of 2 different ranks (8 and 12), 2 different lambdas (0.01, 1.0 and 10.0), and two different numbers of iterations (10 and 20). What are the values for the best model? Store these values, you will need them for the next question.

**CODES =**

```
// Declaration of the model
for (each_rank <- rank_variables){
 for (each_iteration <- iterations_variables){
   for (each_lambda <- lambda_variables){
     // Training
     val model = ALS.train(train_set, rank = each_rank, iterations = each_iteration,
lambda = each_lambda)
     // Predicting
     val predictions = model.predict(test_set.map(line => (line.user,
line.product))).map(x =>(x.user, x.product, x.rating + avg_movie_rating(x.user)))
     // Joining predictions
     val predictions_with_key = predictions.map(x=> ((x._1, x._2), (x._3)))
     val test_Set_with_predictions = test_set_with_key.join(predictions_with_key)
     // Calculating the MSE
     val MSE = ALSParameterTuning.Msecalculator(test_Set_with_predictions)
     println(s"Model training with rank:$each_rank, iteration:$each_iteration,
lambda:$each_lambda is completed. MSE is $MSE")
   }
 }
}
```

**Declaration of the model**

```
def Data_splitter(ratings_w_normalize: RDD[Rating]): (RDD[Rating], RDD[Rating]) = {
 // Divide data to training and test sets
 val Array(train_set, test_set) = ratings_w_normalize.randomSplit(Array[Double](0.9,
0.1), seed = 18)
 (train_set, test_set)
}
```

**Data Splitter**

```
def Msecalculator(predictions: RDD[((Int, Int), (Double, Double))]): Double = {
 predictions.map { case ((user, product), (rating, predicted)) =>
   (rating - predicted) * (rating - predicted)
 }.mean()
}
```

**MSE Calculator**

İbrahim Can Gençel
2166429

**RESULTS =** Best Parameters : **Rank : 8 Iteration : 20 Lambda = 0.01**

```
Model training with rank:8, iteration:10, lambda:0.01 is completed. MSE is 0.634582480682759
Model training with rank:8, iteration:10, lambda:1.0 is completed. MSE is 0.9155614982456337
Model training with rank:8, iteration:10, lambda:10.0 is completed. MSE is 0.9155614982456337
Model training with rank:8, iteration:20, lambda:0.01 is completed. MSE is 0.6340357788386997
Model training with rank:8, iteration:20, lambda:1.0 is completed. MSE is 0.9155614982456337
Model training with rank:8, iteration:20, lambda:10.0 is completed. MSE is 0.9155614982456337
Model training with rank:12, iteration:10, lambda:0.01 is completed. MSE is 0.643252368088492
Model training with rank:12, iteration:10, lambda:1.0 is completed. MSE is 0.9155614982456337
Model training with rank:12, iteration:10, lambda:10.0 is completed. MSE is 0.9155614982456337
Model training with rank:12, iteration:20, lambda:0.01 is completed. MSE is 0.6354039982001762
Model training with rank:12, iteration:20, lambda:1.0 is completed. MSE is 0.9155614982456337
Model training with rank:12, iteration:20, lambda:10.0 is completed. MSE is 0.9155614982456337
```

**1.2)** Build the movies Map[Int,String] that associates a movie identifier to the movie title. This data is available in movies.csv. Our goal is now to select which movies you will rate to build your user profile. Since there are 27k movies in the dataset, if we select these movies at random, it is very likely that you will not know about them. Instead, you will select the 200 most famous movies and rate 40 among them.

**CODES =**

```
// Read movies csv file
val movies_file_w_header = spark.sparkContext.textFile("ml-20m/movies.csv")
// In order to remove header from RDD
val data_header_for_movies = movies_file_w_header.first()
val movies_data = movies_file_w_header.filter(x => x != data_header_for_movies)
// Visualize the first 10 movies
movies_data.take(10).foreach(println)

// Splitting and Mapping movies
val movies =
movies_data.map(collaborative_filtering.parseLineforMovies).collectAsMap()
```

**RESULTS =**

```
1,Toy Story (1995),Adventure|Animation|Children|Comedy|Fantasy
2,Jumanji (1995),Adventure|Children|Fantasy
3,Grumpier Old Men (1995),Comedy|Romance
4,Waiting to Exhale (1995),Comedy|Drama|Romance
5,Father of the Bride Part II (1995),Comedy
6,Heat (1995),Action|Crime|Thriller
7,Sabrina (1995),Comedy|Romance
8,Tom and Huck (1995),Adventure|Children
9,Sudden Death (1995),Action
10,GoldenEye (1995),Action|Adventure|Thriller
```

**Movies Data Example**

İbrahim Can Gençel
2166429

**1.3)** Build mostRatedMovies that contains the 200 movies that were rated by the most users. This is very similar to wordcount, and finding the most frequent words in a document.

Obtain selectedMovies List[(Int, String)] that contains 40 movies selected at random in mostRatedMovies as well as their title. To select elements at random in a list, a good strategy is to shuffle the list (i.e. put it in a random order) and take the first elements. Shuffling the list can be done with scala.util.Random.shuffle.

**CODES =**

```
// Reading ratings csv file
val movie_ratings_w_header = spark.sparkContext.textFile("ml-20m/ratings.csv")
// In order to remove header from RDD
val data_header_for_movie_ratings = movie_ratings_w_header.first()
val movie_ratings = movie_ratings_w_header.filter(x => x !=
data_header_for_movie_ratings)
movie_ratings.take(10).foreach(println)

// In order to find most rated movies, only movieID has taken from ratings and they
were counted desc, filtered first 200 movieIDs
val ratings_2 = movie_ratings.map(collaborative_filtering.parseLineforRatings).map(x
=> (x))
val most_rated_movie_ids = ratings_2.countByValue().toArray.sortWith(_._2 >
_._2).take(200).map(_._1).toSet

// Merge most rated movies and their details
// Shuffle them and take first 40 movies
val selectedMovies_200_movie = movies.filterKeys(most_rated_movie_ids).toList
val selectedMovies = Random.shuffle(selectedMovies_200_movie).take(40)
```

**1.4)** You can now use your recommender system by executing the program you wrote! Write a function elicitateRatings(selectedMovies) gives you 40 movies to rate and you can answer directly in the console in the Scala IDE. Give a rating from 1 to 5, or 0 if you do not know this movie.

**CODES =**

```
def elicitateRatings(selectedMovies: List[(Int, String)]): (Array[Rating]) = {
 val user_id = 19031903
 var format_is_correct = 0
 var terminal_user_ratings =  Array.empty[Rating]
 for (movie <- selectedMovies) {
   format_is_correct = 0
   while (format_is_correct < 1) {
     println("Give a rating for movie:" + movie._2)
     val user_rating = readLine()
     try {
       // "If" executed whether format of rating is correct
       if (user_rating.toInt >= 0 && user_rating.toInt <= 5) {
         val movie_rating = Rating(user_id, movie._1, user_rating.toInt)
         terminal_user_ratings = terminal_user_ratings ++ Array(movie_rating)
         format_is_correct = 1
       }
```

İbrahim Can Gençel
2166429

```
        else {
          println("Please give a rating between 1 and 5 (if you don't know the movie you can give 0)")
        }
      } catch {
        case _: Exception => println("Please give a rating between 1 and 5 (if you don't know the movie you can give 0)")
      }
    }
  }
```

**RESULT =**

```
Give a rating for movie:As Good as It Gets (1997)
4
Give a rating for movie:Dumb & Dumber (Dumb and Dumber) (1994)
0
Give a rating for movie:Kill Bill: Vol. 1 (2003)
4
Give a rating for movie:Finding Nemo (2003)
```

**Example of User Terminal**

**1.5)** After finishing the rating, your program should display the top 20 movies that you might like. Look at the recommendations, are you happy about your recommendations? Comment.

**RESULTS =** Actually, I do not know most of the movies the program recommended to me; therefore, I cannot say I am happy with these results.

```
Recommended movies for terminal user
Gamera: The Giant Monster (Daikaijû Gamera) (1965)
The Heart of the World (2000)
Angel of Death (2009)
Celsius 41.11: The Temperature at Which the Brain... Begins to Die (2004)
On the Silver Globe (Na srebrnym globie) (1988)
"Temptation of St. Tony
Live! (2007)
"Marriage Made in Heaven
Children of Nature (Börn náttúrunnar) (1991)
Phantom (O Fantasma) (2000)
Shinjuku Incident (San suk si gin) (2009)
The Epic of Everest (1924)
"Castle
Al Capone (1959)
Cats (1998)
Free the Nipple (2014)
Khodorkovsky (2011)
Welcome to Macintosh (2008)
Olympia Part Two: Festival of Beauty (Olympia 2. Teil - Fest der Schönheit) (1938)
Gurren Lagann: The Lights in the Sky are Stars (Gekijô ban Tengen toppa guren ragan: Ragan hen) (2009)
```

İbrahim Can Gençel
2166429

## Part 2: Content-based Nearest Neighbors

**2.1)** For this part, you will transform ratings into binary information. There are movies the user liked and movies the user did not like. In a file named nearestneighbors.scala, build the goodRatings RDD by transforming the ratings RDD to only keep, for each user, ratings that are above their average. For instance, if a user rates on average 2.8, we only keep their ratings that are greater or equal to 2.8.

```
// In order to use "Rating", I have used
https://spark.apache.org/docs/latest/mllib-collaborative-filtering.html website
val ratings = data.map(_.split(',') match { case Array(user, item, rate, ts) =>
 Rating(user.toInt, item.toInt, rating = rate.toDouble)
})
// --------------------------------------------- //
// In order to calculate mean of rating for each user, Rating is grouped according to
user
val ratings_grouped_by_user = ratings.groupBy(line => line.user)
// Sum of ratings per user is mapped as -> (user, Sum of ratings)
val sum_of_ratings_per_user = ratings_grouped_by_user.map(x => (x._1,
x._2.map(coproduct => coproduct.rating).sum))
// Number of ratings per used is mapped as -> (user, Number of ratings)
val number_of_ratings_per_used = ratings_grouped_by_user.map(x => (x._1, x._2.size))
// Avg Rating Per User and convert it to Map
(https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.RDD.collectAsM
ap.html)
val avg_movie_rating = sum_of_ratings_per_user.join(number_of_ratings_per_used).
 map { case (user, (sum_of_ratings, number_of_ratings))
 => (user, sum_of_ratings / number_of_ratings)
 }.collectAsMap()

//------------------------- PART 2.1 -------------------------//
// In order to only keep ratings which are higher than avg. ratings
val goodRatings = ratings.filter(f =>
 f.rating > avg_movie_rating(f.user)
)
```

**2.2)** Build the movieNames Map[Int,String] that associates a movie identifier to the movie name. You have already done this in the previous part of this assignment.

```
def parseLineforMovies(line: String): (Int, String) = {
 val fields = line.split(",")
 val movieID = fields(0).toInt
 val title = fields(1)
 return (movieID, title)
}
//------------------------- PART 2.2 -------------------------//
// Read movies csv file
val movies_file_w_header = spark.sparkContext.textFile("ml-20m/movies.csv")
// This part is also made in Part 1 therefore I copied this part
val data_header_for_movies = movies_file_w_header.first()
val movies_data = movies_file_w_header.filter(x => x != data_header_for_movies)
```

İbrahim Can Gençel
2166429

```
val movieNames =
movies_data.map(collaborative_filtering.parseLineforMovies).collectAsMap()
```

**2.3)** Build the movieGenres Map[Int, Array[String]] that associates a movie identifier to the list of genres it belongs to. This information is available in the movies.csv file, in the third column, and movies are separated by "|". If you use split, you will need to write "\\|" as a parameter.

**CODES =**

```
def parseLineforMovieswithGenres(line: String): (Int, Array[String]) = {
 val fields = line.split(",")
 val movieID = fields(0).toInt
 val genres = fields(2).split("\\|")
 return (movieID, genres)
}
//------------------------- PART 2.3 -------------------------//
val movieGenres =
movies_data.map(nearestneighbors.parseLineforMovieswithGenres).collectAsMap()
```

**2.4)** Provide the code that builds the userVectors RDD. This RDD contains (Int, Map[String, Int]) pairs in which the first element is a user ID, and the second element is the vector describing the user. If a user has liked 2 action movies, then this vector will contain an entry (“action”, 2). Write the userSim function that computes the cosine similarity between two user vectors. The mathematical formula is available on the slides. To perform a square root operation, use Math.sqrt(x) .

**CODES =**

```
def userSim(user1_genres: Map[String, Int], user2_genres: Map[String, Int]): (Double)
= {
 var length_of_user1 = 0.0
 var length_of_user2 = 0.0
 var similarity_mult = 0.0

 // Calculating the length of the vectors
 for (each_genre <- user1_genres) {
   length_of_user1 += (each_genre._2) * (each_genre._2)
 }
 for (each_genre <- user2_genres) {
   length_of_user2 += (each_genre._2) * (each_genre._2)
 }
 length_of_user1 = Math.sqrt(length_of_user1)
 length_of_user2 = Math.sqrt(length_of_user2)
 // -------------//

 for (each_genre <- user1_genres) {
   Breaks.breakable {
     try {
       val genre_value = user2_genres(each_genre._1)
       val mult_of_genre_values = genre_value * user1_genres(each_genre._1)
       similarity_mult += mult_of_genre_values
     }
```

```
      catch {
        case _ =>
          Breaks.break
      }
    }
  }
  val similarity = (similarity_mult / length_of_user1) / length_of_user2
  return similarity
}
//------------------------- PART 2.4 -------------------------//
val user_w_genres = goodRatings.map(f => (f.user,
movieGenres(f.product))).groupByKey.map(eachuser => (eachuser._1,
eachuser._2.toArray.flatten))
val userVectors = user_w_genres.map(each_user => (each_user._1,
each_user._2.groupBy(identity).map(each_genre => (each_genre._1,
each_genre._2.length))))
```

**2.5)** Now, write a function named knn that takes a user profile named testUser. Then the function selects the list of k users that are most similar to the testUser, and returns recom, the list of movies recommended to the user.

**2.6)** Congratulations, you can now experiment with your recommender system by modifying the vector of testUser and see which recommendations you get. Use the profile you built for yourself in Part 1 and list the recommendations. Comment on the performance of recommendations. Also, compare the two methods you implemented in Part 1 and 2.

**- I give the answers for 2.5 and 2.6 in the same block because they are connected to each other.**

**CODES =**

```
def get_terminal_user(movies_data: RDD[String], movie_ratings: RDD[String]):
Array[Rating] = {
  // This function is taken from Part1.scala //
  // -------------------------------------- //
  // Splitting and Mapping movies
  val movies =
movies_data.map(collaborative_filtering.parseLineforMovies).collectAsMap()

  // In order to find most rated movies, only movieID has taken from ratings and they
were counted desc, filtered first 200 movieIDs
  val ratings_2 = movie_ratings.map(collaborative_filtering.parseLineforRatings).map(x
=> (x))
  val most_rated_movie_ids = ratings_2.countByValue().toArray.sortWith(_._2 >
_._2).take(200).map(_._1).toSet

  // Merge most rated movies and their details
  // Shuffle them and take first 40 movies
  val selectedMovies_200_movie = movies.filterKeys(most_rated_movie_ids).toList
  val selectedMovies = Random.shuffle(selectedMovies_200_movie).take(40)

  // Terminal user ratings are collected
  val terminal_user_ratings = collaborative_filtering.elicitateRatings(selectedMovies)
  // Normalize user ratings
```

İbrahim Can Gençel
2166429

```
 val avg_rating_of_terminal_user = terminal_user_ratings.groupBy(x => x.user).map(x
=> (x._2.map(x=> x.rating).sum/x._2.length)).sum
 val normalized_terminal_user_ratings = terminal_user_ratings.filter(x =>
(x.rating>=avg_rating_of_terminal_user))
 return normalized_terminal_user_ratings
}
def knn(testUser: Map[String, Int], userVectors: RDD[(Int, Map[String, Int])], k:
Int, goodRatings: RDD[Rating]): (Set[Int]) = {
 var test_user_mapping: Map[Int, Double] = Map()
 for(each_user <- userVectors.collectAsMap()){
   val user_map = each_user._2
   val user_id = each_user._1
   // For each user similarity is calculated
   val user_similarity = nearestneighbors.userSim(testUser, user_map)
   test_user_mapping += (user_id -> user_similarity)
 }
 // Mapping is sorted as descending order for similarity
 val sorted_Map = Map(test_user_mapping.toSeq.sortWith(_._2 > _._2):_*)
 // Top k result is taken
 val Map_k_users = sorted_Map.take(k).keys.toSet
 // Top k users' favourite movie IDs are returned
 val rated_movie_ids = goodRatings.filter{line =>
Map_k_users.contains(line.user)}.map(rate => rate.product).collect().toSet
 return rated_movie_ids
}
```

**Get Terminal User Input and KNN Functions**

```
//------------------- PART 2.5 and PART 2.6 -------------------//
val terminal_user_ratings = get_terminal_user(movies_data, data)
// I have converted it to RDD with parallelize function
val terminal_user_ratings_RDD = spark.sparkContext.parallelize(terminal_user_ratings)
// User movies with genres
val terminal_w_genres = terminal_user_ratings_RDD.map(f => (f.user,
movieGenres(f.product))).groupByKey.map(eachuser => (eachuser._1,
eachuser._2.toArray.flatten))
// Vectors for terminal user is created
val terminalVectors = terminal_w_genres.map(each_user => (each_user._1,
each_user._2.groupBy(identity).map(each_genre => (each_genre._1,
each_genre._2.length))))
// It is converted to map
val terminalVectors_map = terminalVectors.map(_._2).collect()(0)
// Recom -> it is a set which includes movie ids
val recom = nearestneighbors.knn(terminalVectors_map, userVectors, k = 2,
goodRatings)
println("Recommended Movies:")
movieNames.filterKeys(recom).foreach(println)
```

İbrahim Can Gençel
2166429

**RESULTS =**

```
Recommended Movies:
(35836,"40-Year-Old Virgin)
(783,"Hunchback of Notre Dame)
(1097,E.T. the Extra-Terrestrial (1982))
(3071,Stand and Deliver (1988))
(3289,Not One Less (Yi ge dou bu neng shao) (1999))
(1961,Rain Man (1988))
(3361,Bull Durham (1988))
(3863,"Cell)
(2069,"Trip to Bountiful)
(53953,1408 (2007))
(81562,127 Hours (2010))
(51372,"""Great Performances"" Cats (1998)")
(3534,28 Days (2000))
(2915,Risky Business (1983))
(3471,Close Encounters of the Third Kind (1977))
(7293,50 First Dates (2004))
(51662,300 (2007))
(72378,2012 (2009))
(1204,Lawrence of Arabia (1962))
(56949,27 Dresses (2008))
(3360,Hoosiers (a.k.a. Best Shot) (1986))
(1234,"Sting)
(26068,"4 Horsemen of the Apocalypse)
(3468,"Hustler)
(7541,100 Girls (2000))
(1228,Raging Bull (1980))
(2971,All That Jazz (1979))
(58803,21 (2008))
(1096,Sophie's Choice (1982))
(2917,Body Heat (1981))
(1278,Young Frankenstein (1974))
(3548,Auntie Mame (1958))
(608,Fargo (1996))
(2959,Fight Club (1999))
(1299,"Killing Fields)
(69757,(500) Days of Summer (2009))
(1212,"Third Man)
(4370,A.I. Artificial Intelligence (2001))
```

**COMMENT =** According to performance, Part 2 (KNN) perform better than ALS model in my own recommendations. I have given my ratings for top 40 movies, I even did not know the most of the movies output of ALS. However, output of the KNN model was better, I have seen most of the movies output of KNN. Also, implementation and running is faster in KNN model. Therefore, when I need a recommendation system, I will give a try first to KNN model.