Ceng790 Big Data Analytics
Assignment 2

Egemen Berk Galatali
e2099018@ceng.metu.edu.tr

Note: Implementations are explained in three files ALSParameterTuning.scala, collaborative_filtering.scala and NearestNeighbors.scala by comments.

Part I

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.

   I have calculated these experiments in a file called ALSParameterTuning.scala. Process is commented and explained there. Results of each experiment is in the below.

```
RANK ——— LAMBDA ——— ITERATION ——— MSE
8 ————— 0.01 —————10 ————— 0.057007881355108133
8 ————— 0.01 —————20 ————— 0.055775090517082376
8 ————— 1.0 —————10 ————— 1.0815251193870754
8 ————— 1.0 —————20 ————— 1.0814380844180125
8 ————— 10.0 —————10 ————— 1.0815890141943867
8 ————— 10.0 —————20 ————— 1.0815890141943867
12 ————— 0.01 —————10 ————— 0.054980334950302044
12 ————— 0.01 —————20 ————— 0.05457710651737128
12 ————— 1.0 —————10 ————— 1.081535328833949
12 ————— 1.0 —————20 ————— 1.0814692332020546
12 ————— 10.0 —————10 ————— 1.0815890141943867
12 ————— 10.0 —————20 ————— 1.0815890141943867
```

2. Build the movies Map[Int,String] that associates a movie identifier to the movie title.

   We create an class for Movies like the following for ease of use. We read movie.csv and create Movie objects out of it and hold these in RDD[Movie]. I did not create an value or variable for this map, insetead I created map[Int,String] on the fly.

```
case class Movie(movieId:Int, title:String, genre:Array[String])
```

   Building movies Map is done by

```
movies:RDD[Movie].map(m => (m.movieId,m.title)).toMap
```

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.

   We first choose most rated movies for that reason after reading "rating.csv" file and create a Rating object which is provided by the spark for each, we groupBy(rating.product), so that we create a RDD[movieId, Iterable[Rating]]. We get how many times that movie is rated by taking the size of the Iterable[Rating] and sort the whole RDD according to it. We take top 200 and then shuffle it then take 40 of them to ask to the user.

```scala
// Create mostRatedMovies as an Array(movieId, Number Of ratings)
val mostRatedMovies = ratings.groupBy(_.product)
                    .map(f=> (f._1, f._2.size))
                    .takeOrdered(200)(Ordering[Int].reverse.on(_._2))

// Select 40 of them
val selectedMovies = shuffle(mostRatedMovies)
                    .map(f => (f._1,
                    movies.filter(_.movieId == f._1).map(m => m.title )
                    .take(1)(0) ) )
                    .take(40).toMap

```

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 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.
   In this function we ask user 40 movies from selectedMovies and ask user to rate each. We return RDD[Rating] so that we can add the new user's ratings to the dataset and train model.

```scala
def elicitateRatings(selectedMovies: Map[Int, String],
                    spark: SparkSession): RDD[Rating] = {
    var userId = 0
    var userRatingForSelectedMovies  = ArrayBuffer.empty[Rating]
    var i = 1
    for(movie <- selectedMovies) {
        breakable {
        while(true) {
            try {
            println(i.toString + ") What is your rating for the movie "
            + movie._2 + "\"" )
            val rating = scala.io.StdIn.readDouble()
            userRatingForSelectedMovies += Rating(0, movie._1, rating)
            i += 1
            break
            } catch {
                case e: Exception => println("Please Give correct input");
            }
        }
    }
    }
    return spark.sparkContext.parallelize(userRatingForSelectedMovies)
}
```

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.

   To be honest, I only find 1 or two movies to be interesting. But, I did not watch nearly 1/3 of the 40 movies I was asked.

Part II

1. Build the goodRatings RDD by transforming the ratings RDD to only keep, for each user, ratings that are above their average.

   We first read whole ratings into RDD[Rating] then groupBy userId, then create a avgRatings : RDD[userId : Int, averageRatingsOfUser: Double, Iterable[Rating]] then eliminate ratings that are below average for each user.

2. Build the movieNames Map[Int,String] that associates a movie identifier to the movie name.

   Actually, I created this movieNames as RDD[Int, String], because I created movies:RDD[Movies] before and mapping it to movieId and title will result in RDD again. I can transform this new RDD to Map however, I faced with time and memory problems. So I stick with RDD whenever possible.

```
val movieNames = movies.map(m => (m.movieId, m.title))
```

3. Build the movieGenres Map[Int, Array[String]] that associates a movie identifier to the list of genres it belongs to.

```
val movieGenres = movies.map(m => (m.movieId, m.genre))\textbf{}
```

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.

```
// avgRatings2 :RDD[Rating] holds ratings that are above avereage
// This is goodRatings without doing groupBy(rating.user)

// movieId, rating
val ratedMovieGenres = avgRatings2.map(rat => (rat.product, rat))

// userId, Genres of users
val userGenres = ratedMovieGenres.join(movieGenres)
                    .map(x => (x._2._1.user, x._2._2)).groupByKey()

// userId, genres with count
val userGenresWithCounts = userGenres.map(u => (u._1, u._2
                          .flatMap(genreArr => genreArr.map(genre => (genre,1)))))
```

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.

   These functions are defined in appropriate file with detailed comments, so I do not repeat the same things again here.

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 think second implementation gives results more intuitively because it is based on movie genres and generally if the person does not like specific genre at all there is no need to recommend from that genre too much. I found that the second implementation gives better recommendations compared to the first one.