CENG 790: Big Data Analytics, Fall 2020

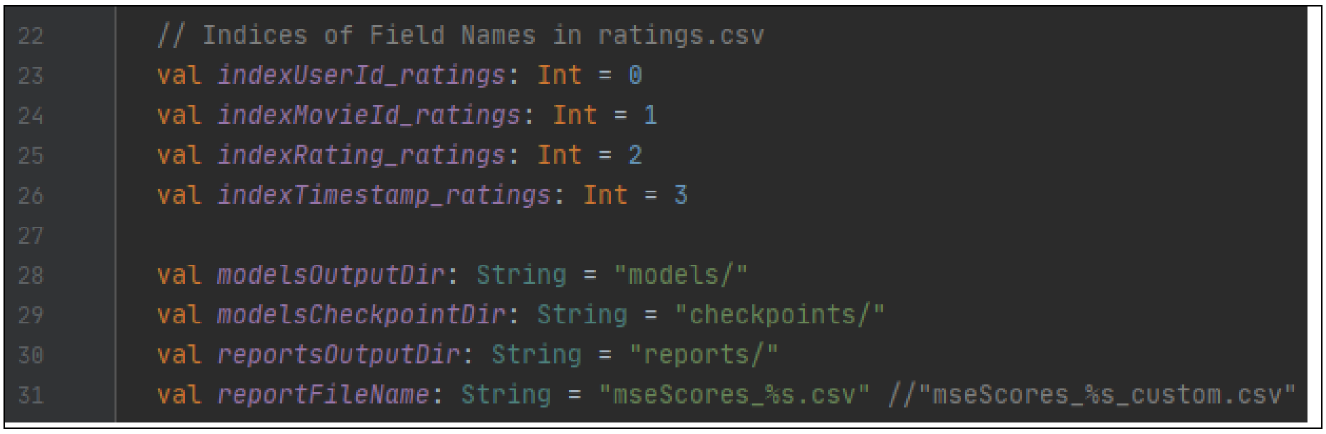
Report of Assignment 2: Recommender Systems

Part 1: Collaborative Filtering

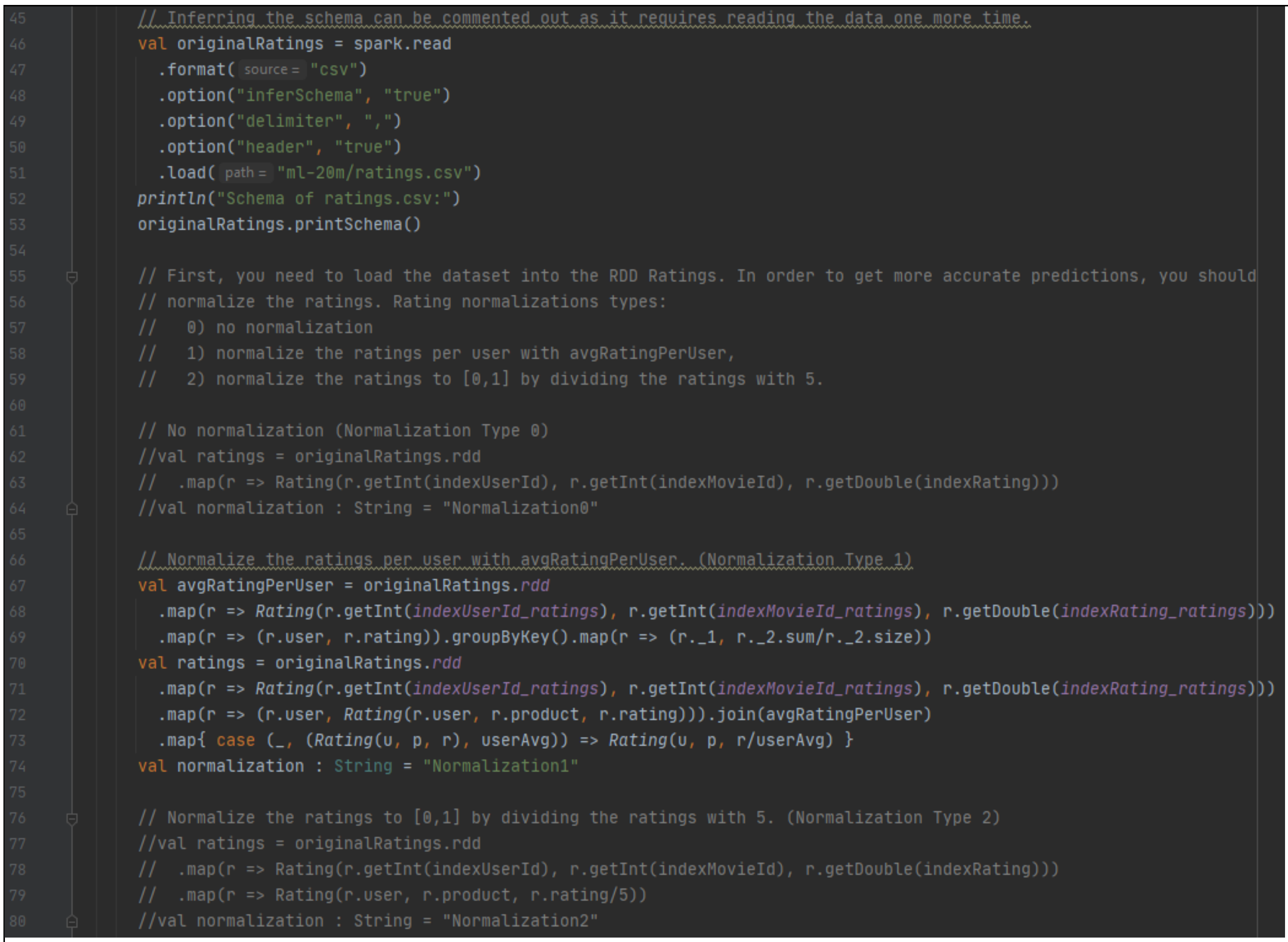
Train a model and tune parameters

First, you need to load the dataset into the RDD ratings. The tuples in this RDD are instance of org.apache.spark.mllib.recommendation.Rating (user: Int, product: Int, rating: Double).

In order to get more accurate predictions, you should normalize the ratings per user with *avgRatingPerUser*.



```scala
// Indices of Field Names in ratings.csv
val indexUserId_ratings: Int = 0
val indexMovieId_ratings: Int = 1
val indexRating_ratings: Int = 2
val indexTimestamp_ratings: Int = 3

val modelsOutputDir: String = "models/"
val modelsCheckpointDir: String = "checkpoints/"
val reportsOutputDir: String = "reports/"
val reportFileName: String = "mseScores_%s.csv" //"mseScores_%s_custom.csv"
```



```scala
    // Inferring the schema can be commented out as it requires reading the data one more time.
    val originalRatings = spark.read
      .format( source = "csv")
      .option("inferSchema", "true")
      .option("delimiter", ",")
      .option("header", "true")
      .load( path = "ml-20m/ratings.csv")
    println("Schema of ratings.csv:")
    originalRatings.printSchema()

    // First, you need to load the dataset into the RDD Ratings. In order to get more accurate predictions, you should
    // normalize the ratings. Rating normalizations types:
    //   0) no normalization
    //   1) normalize the ratings per user with avgRatingPerUser,
    //   2) normalize the ratings to [0,1] by dividing the ratings with 5.

    // No normalization (Normalization Type 0)
    //val ratings = originalRatings.rdd
    //  .map(r => Rating(r.getInt(indexUserId), r.getInt(indexMovieId), r.getDouble(indexRating)))
    //val normalization : String = "Normalization0"

    // Normalize the ratings per user with avgRatingPerUser. (Normalization Type 1)
    val avgRatingPerUser = originalRatings.rdd
      .map(r => Rating(r.getInt(indexUserId_ratings), r.getInt(indexMovieId_ratings), r.getDouble(indexRating_ratings)))
      .map(r => (r.user, r.rating)).groupByKey().map(r => (r._1, r._2.sum/r._2.size))
    val ratings = originalRatings.rdd
      .map(r => Rating(r.getInt(indexUserId_ratings), r.getInt(indexMovieId_ratings), r.getDouble(indexRating_ratings)))
      .map(r => (r.user, Rating(r.user, r.product, r.rating))).join(avgRatingPerUser)
      .map{ case (_, (Rating(u, p, r), userAvg)) => Rating(u, p, r/userAvg) }
    val normalization : String = "Normalization1"

    // Normalize the ratings to [0,1] by dividing the ratings with 5. (Normalization Type 2)
    //val ratings = originalRatings.rdd
    //  .map(r => Rating(r.getInt(indexUserId), r.getInt(indexMovieId), r.getDouble(indexRating)))
    //  .map(r => Rating(r.user, r.product, r.rating/5))
    //val normalization : String = "Normalization2"
```

I tried 3 types of normalization in order to compare:

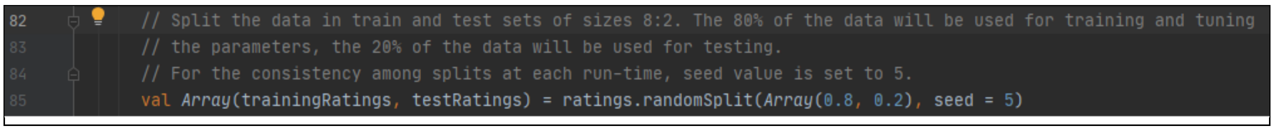
- No normalization,
- Normalize the ratings per user with ***avgRatingPerUser***,
- Normalize the ratings to **[0,1]** range by dividing the ratings with 5.

I selected to go with normalizing the ratings per user with avgRatingPerUser as also stated in the assignment. I thought that normalizing the ratings per user with avgRatingPerUser could be better in the learning process. That is because, if the model predicts a lower value for the movie that a user rated above the average, it will be penalized much more, or vice versa. Thus, this contributes the learning process.

You are ready to train an ALS model. The train method of ALS, has the following parameters:

• **rank** - is the number of latent factors in the model.
• **iterations** - is the number of iterations of ALS to run.
• **lambda** - specifies the regularization parameter in ALS.

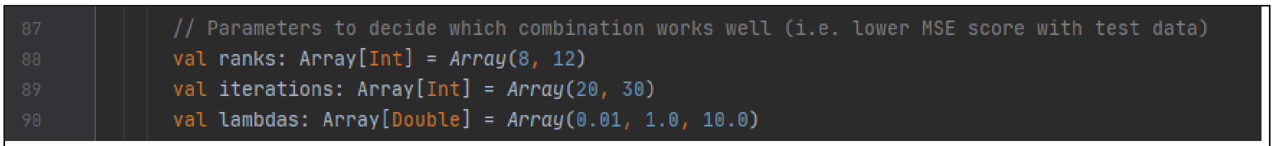
Different values of these parameters result in different models and consequently different quality in predictions. To compare different models, you will use Mean Squared Error (MSE) as an evaluation criterion. To be more objective in the evaluation process, we split the data in train and test sets of sizes 8:2. We train and tune parameters on the 80% of the data, and then test on the 20% we put aside at the beginning.



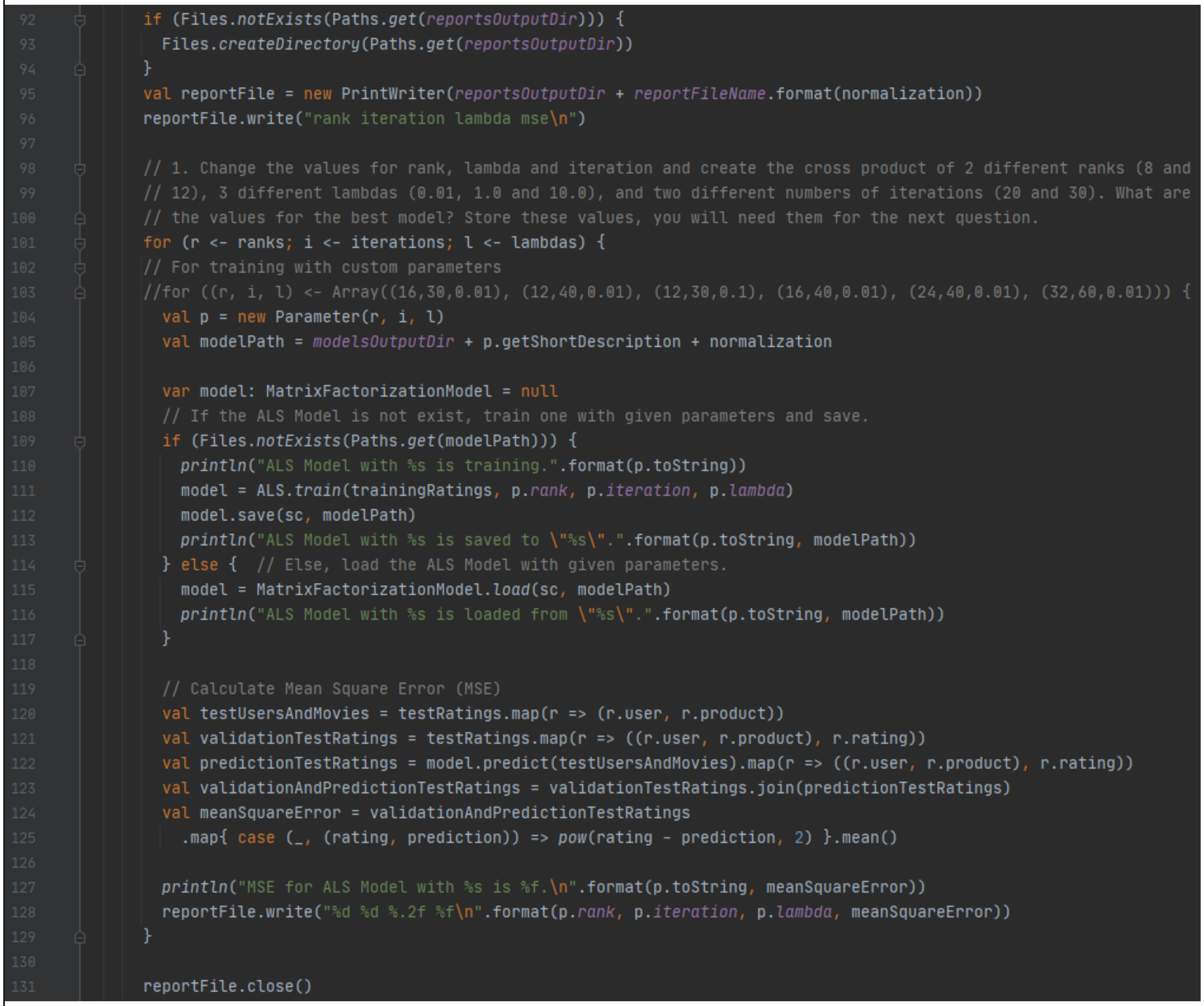
```scala
    // Split the data in train and test sets of sizes 8:2. The 80% of the data will be used for training and tuning
    // the parameters, the 20% of the data will be used for testing.
    // For the consistency among splits at each run-time, seed value is set to 5.
    val Array(trainingRatings, testRatings) = ratings.randomSplit(Array(0.8, 0.2), seed = 5)
```

I used a seed value in order to split the data consistently for each running.

In a file named ***ParameterTuningALS.scala***, provide the code for the split and calculation of MSE. MSE is a simple sum of the errors we make in our prediction compared to the true ratings.



```scala
// Parameters to decide which combination works well (i.e. lower MSE score with test data)
val ranks: Array[Int] = Array(8, 12)
val iterations: Array[Int] = Array(20, 30)
val lambdas: Array[Double] = Array(0.01, 1.0, 10.0)
```



```scala
if (Files.notExists(Paths.get(reportsOutputDir))) {
  Files.createDirectory(Paths.get(reportsOutputDir))
}
val reportFile = new PrintWriter(reportsOutputDir + reportFileName.format(normalization))
reportFile.write("rank iteration lambda mse\n")

// 1. Change the values for rank, lambda and iteration and create the cross product of 2 different ranks (8 and
// 12), 3 different lambdas (0.01, 1.0 and 10.0), and two different numbers of iterations (20 and 30). What are
// the values for the best model? Store these values, you will need them for the next question.
for (r <- ranks; i <- iterations; l <- lambdas) {
// For training with custom parameters
//for ((r, i, l) <- Array((16,30,0.01), (12,40,0.01), (12,30,0.1), (16,40,0.01), (24,40,0.01), (32,60,0.01))) {
  val p = new Parameter(r, i, l)
  val modelPath = modelsOutputDir + p.getShortDescription + normalization

  var model: MatrixFactorizationModel = null
  // If the ALS Model is not exist, train one with given parameters and save.
  if (Files.notExists(Paths.get(modelPath))) {
    println("ALS Model with %s is training.".format(p.toString))
    model = ALS.train(trainingRatings, p.rank, p.iteration, p.lambda)
    model.save(sc, modelPath)
    println("ALS Model with %s is saved to \"%s\".".format(p.toString, modelPath))
  } else {  // Else, load the ALS Model with given parameters.
    model = MatrixFactorizationModel.load(sc, modelPath)
    println("ALS Model with %s is loaded from \"%s\".".format(p.toString, modelPath))
  }

  // Calculate Mean Square Error (MSE)
  val testUsersAndMovies = testRatings.map(r => (r.user, r.product))
  val validationTestRatings = testRatings.map(r => ((r.user, r.product), r.rating))
  val predictionTestRatings = model.predict(testUsersAndMovies).map(r => ((r.user, r.product), r.rating))
  val validationAndPredictionTestRatings = validationTestRatings.join(predictionTestRatings)
  val meanSquareError = validationAndPredictionTestRatings
    .map{ case (_, (rating, prediction)) => pow(rating - prediction, 2) }.mean()

  println("MSE for ALS Model with %s is %f.\n".format(p.toString, meanSquareError))
  reportFile.write("%d %d %.2f %f\n".format(p.rank, p.iteration, p.lambda, meanSquareError))
}

reportFile.close()
```

It creates a directory for reports and creates a report file in that directory which holds the training results. Report file's header is "rank iteration lambda mse" and it stores the training model's parameters and the Mean Square Error (MSE) value.
With given parameters, it searches for a folder. If the folder exists, it loads the model from that folder. Otherwise, it trains a new model and saves it into that folder. This saves time in case of problems in the model training phase.
Then, trained or loaded Matrix Factorization Model is used for calculating the Mean Square Error value.

Ideally, we want to try a large number of combinations of parameters in order to find the best one, i.e. the one with lowest MSE. Due to time constraints, you should start by testing only 12 combinations:

1. **1.** Change the values for rank, lambda and iteration and create the cross product of 2 different ranks (8 and 12), 3 different lambdas (0.01, 1.0 and 10.0), and two different numbers of iterations (20 and 30). What are the values for the best model? Store these values, you will need them for the next question.

Rank	Iteration	Lambda	MSE
8	20	0.01	**0.056105**
8	20	1.0	1.082340
8	20	10.0	1.082389
8	30	0.01	**0.055995**
8	30	1.0	1.082309
8	30	10.0	1.082389
12	20	0.01	**0.055223**
12	20	1.0	1.082363
12	20	10.0	1.082389
12	30	0.01	**0.054984**
12	30	1.0	1.082329
12	30	10.0	1.082389

In terms of rank, higher rank seems to be better when lambda value is 0.01.
In terms of iteration, higher iteration also seems to be better option.
In terms of lambda, Mean Square Error values are lower when the lambda is 0.01 whereas MSE scores are higher when the lambda is higher (1.0 or 10.0).
It can be deducted from these scores that models are starting to converge when the lambda values are high as there is little or no difference between the scores of models with high lambda values and different iterations.
Therefore, I chose to continue with high rank value and iteration count with low lambda value:
Rank: 12, Iteration: 30, Lambda: 0.01

However, I tried training another models with custom parameters which is deducted from the results above:

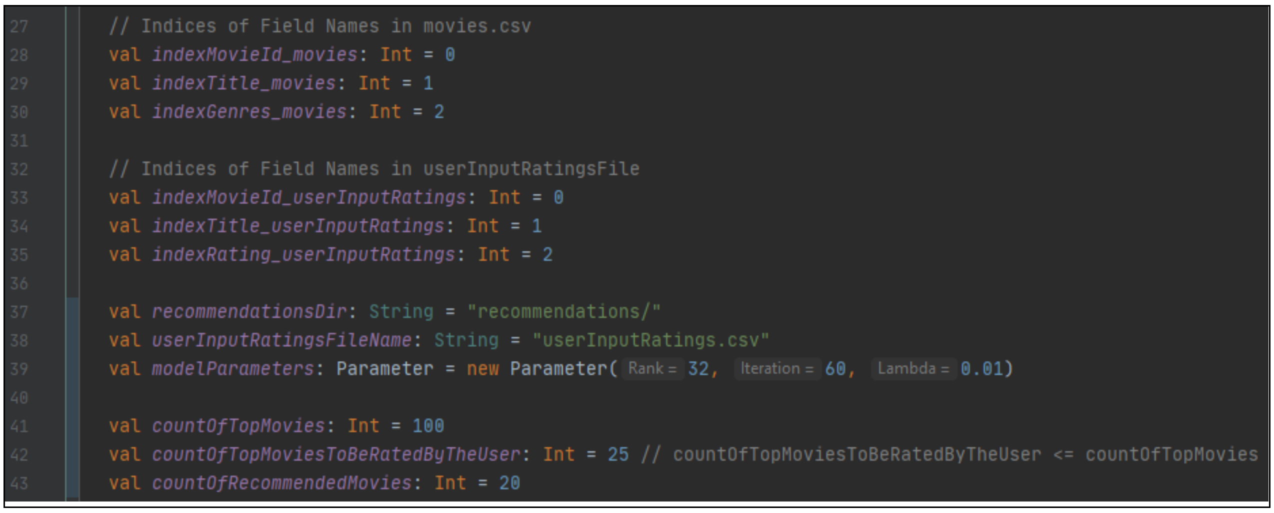
Rank	Iteration	Lambda	MSE
12	30	0.01	**0.054984**
16	30	0.01	0.054503
12	40	0.01	0.054946
12	30	0.1	0.076856
12	30	0.001	0.060304
16	40	0.01	0.054446
24	40	0.01	0.054033
32	60	0.01	**0.053832**

I tried increasing only the rank value which resulted in improvement.
I tried increasing only the iteration count which resulted in slight improvement.
I tried increasing only the lambda value which resulted in deterioration.
I tried decreasing only the lambda value which resulted in deterioration. However, it can be improved with increasing the iteration count, but I didn't try that.

After training models with custom parameters, I chose to go with the model with parameters:
Rank: 32, Iteration: 60, Lambda: 0.01
It can be inferred from the parameter tuning part that training a model with higher rank, lower lambda values and many iterations can be resulted in better model.

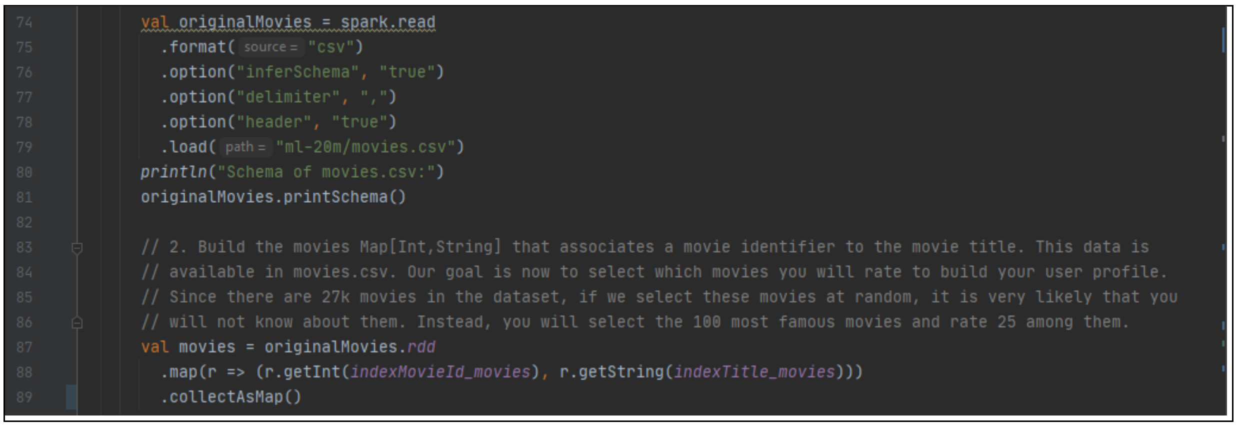
Getting your own recommendations

Now that you have experimented with ALS, you know which parameters should be used to configure the algorithm. Create another file named ***CollabFiltering.scala***. Our objective now is to go beyond aggregated performance measures such as MSE and see if you would be satisfied by the recommendations of the system you just built. To do this, you need to add you as a user in the dataset.



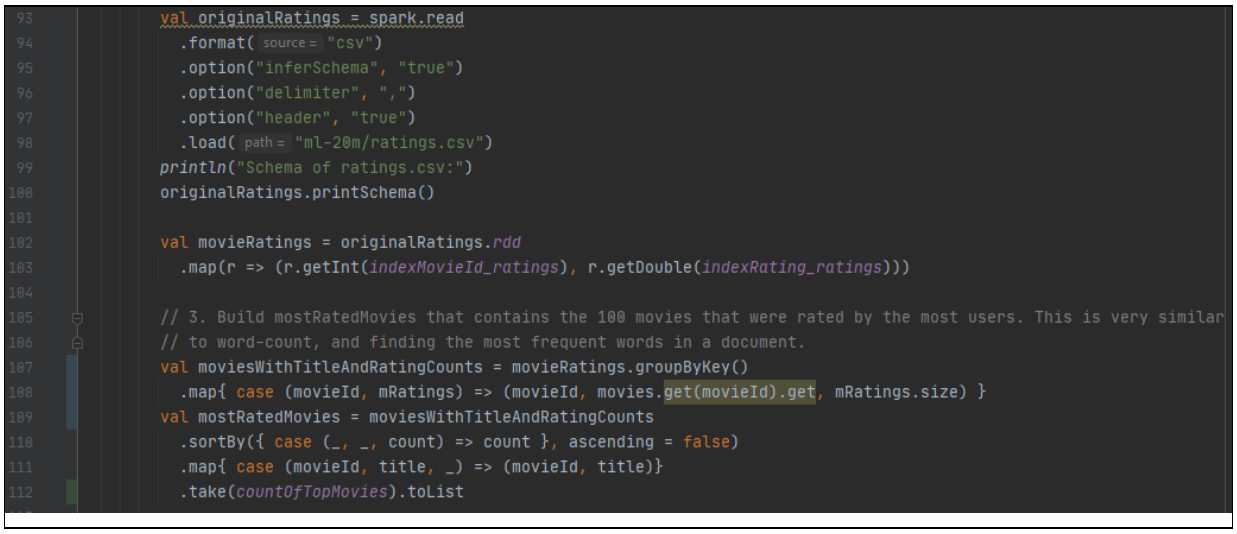
```scala
// Indices of Field Names in movies.csv
val indexMovieId_movies: Int = 0
val indexTitle_movies: Int = 1
val indexGenres_movies: Int = 2

// Indices of Field Names in userInputRatingsFile
val indexMovieId_userInputRatings: Int = 0
val indexTitle_userInputRatings: Int = 1
val indexRating_userInputRatings: Int = 2

val recommendationsDir: String = "recommendations/"
val userInputRatingsFileName: String = "userInputRatings.csv"
val modelParameters: Parameter = new Parameter( Rank = 32,  Iteration = 60,  Lambda = 0.01)

val countOfTopMovies: Int = 100
val countOfTopMoviesToBeRatedByTheUser: Int = 25 // countOfTopMoviesToBeRatedByTheUser <= countOfTopMovies
val countOfRecommendedMovies: Int = 20
```

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 100 most famous movies and rate 25 among them.



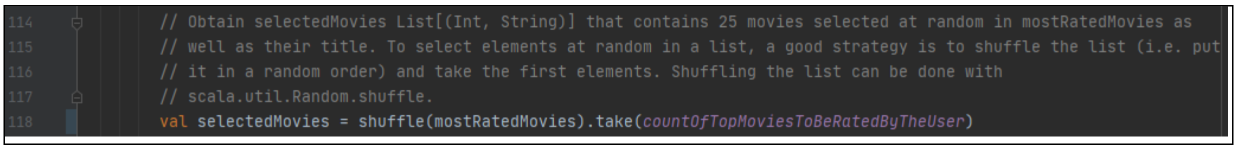
```scala
val originalMovies = spark.read
  .format( source = "csv")
  .option("inferSchema", "true")
  .option("delimiter", ",")
  .option("header", "true")
  .load( path = "ml-20m/movies.csv")
println("Schema of movies.csv:")
originalMovies.printSchema()

// 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 100 most famous movies and rate 25 among them.
val movies = originalMovies.rdd
  .map(r => (r.getInt(indexMovieId_movies), r.getString(indexTitle_movies)))
  .collectAsMap()
```

3. Build mostRatedMovies that contains the 100 movies that were rated by the most users. This is very similar to wordcount and finding the most frequent words in a document.



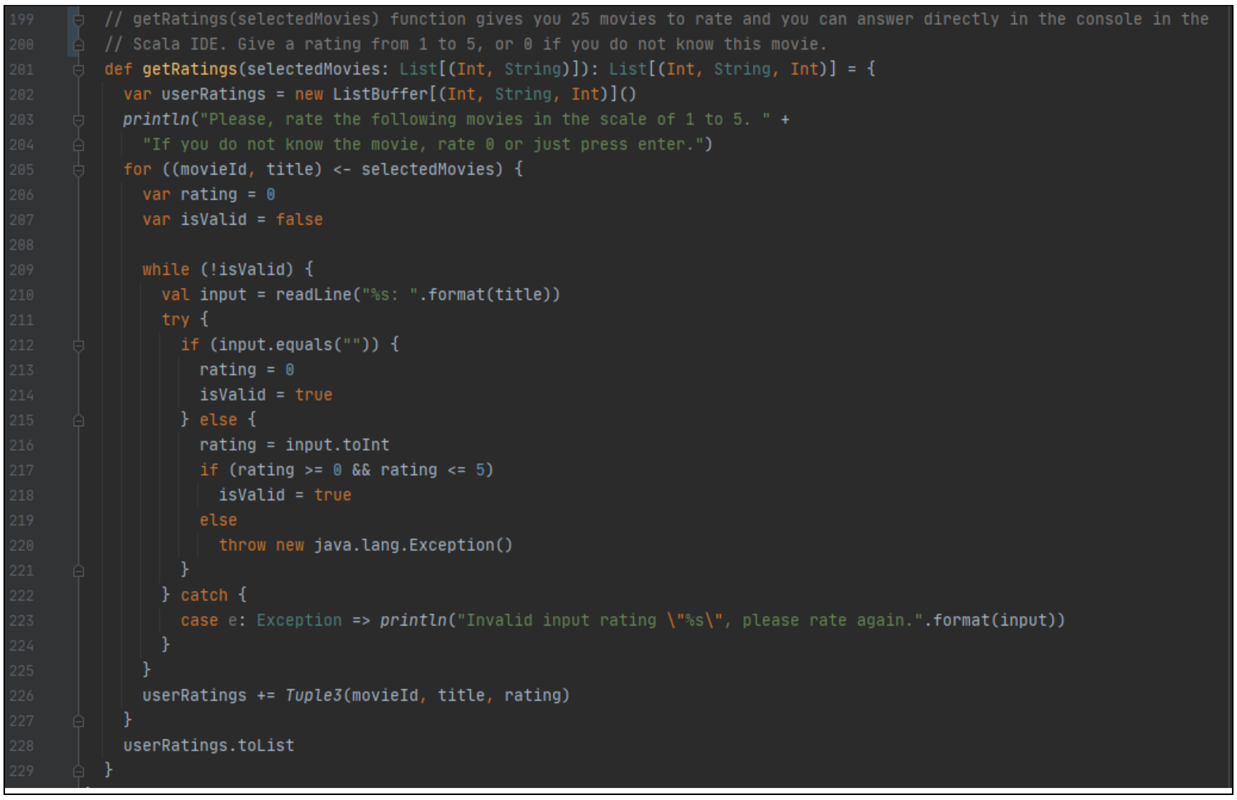
```scala
    val originalRatings = spark.read
      .format( source = "csv")
      .option("inferSchema", "true")
      .option("delimiter", ",")
      .option("header", "true")
      .load( path = "ml-20m/ratings.csv")
    println("Schema of ratings.csv:")
    originalRatings.printSchema()

    val movieRatings = originalRatings.rdd
      .map(r => (r.getInt(indexMovieId_ratings), r.getDouble(indexRating_ratings)))

    // 3. Build mostRatedMovies that contains the 100 movies that were rated by the most users. This is very similar
    // to word-count, and finding the most frequent words in a document.
    val moviesWithTitleAndRatingCounts = movieRatings.groupByKey()
      .map{ case (movieId, mRatings) => (movieId, movies.get(movieId).get, mRatings.size) }
    val mostRatedMovies = moviesWithTitleAndRatingCounts
      .sortBy({ case (_, _, count) => count }, ascending = false)
      .map{ case (movieId, title, _) => (movieId, title)}
      .take(countOfTopMovies).toList
```

Obtain selectedMovies List[(Int, String)] that contains 25 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.



```scala
    // Obtain selectedMovies List[(Int, String)] that contains 25 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.
    val selectedMovies = shuffle(mostRatedMovies).take(countOfTopMoviesToBeRatedByTheUser)
```

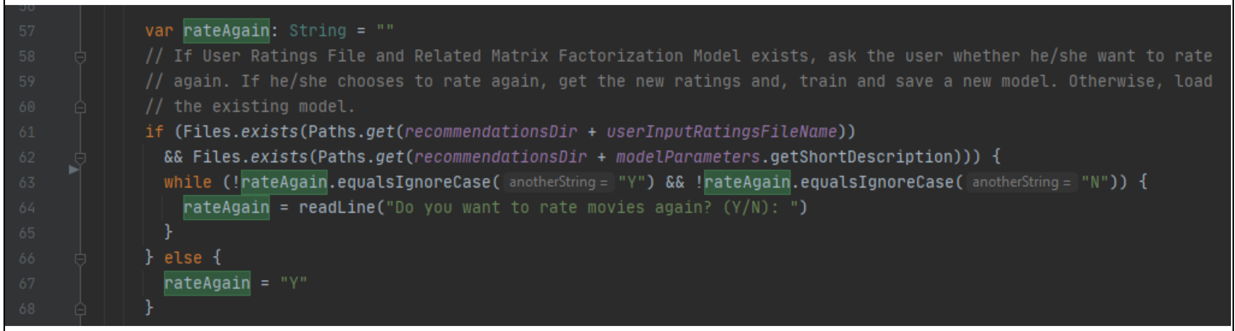
4. You can now use your recommender system by executing the program you wrote! Write a function getRatings(selectedMovies) gives you 25 movies to rate and you can answer directly in the console in the Scala IDE. Give a rating from 1 to 5, or 0 if you do not know this movie.



```scala
  // getRatings(selectedMovies) function gives you 25 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.
  def getRatings(selectedMovies: List[(Int, String)]): List[(Int, String, Int)] = {
    var userRatings = new ListBuffer[(Int, String, Int)]()
    println("Please, rate the following movies in the scale of 1 to 5. " +
      "If you do not know the movie, rate 0 or just press enter.")
    for ((movieId, title) <- selectedMovies) {
      var rating = 0
      var isValid = false

      while (!isValid) {
        val input = readLine("%s: ".format(title))
        try {
          if (input.equals("")) {
            rating = 0
            isValid = true
          } else {
            rating = input.toInt
            if (rating >= 0 && rating <= 5)
              isValid = true
            else
              throw new java.lang.Exception()
          }
        } catch {
          case e: Exception => println("Invalid input rating \"%s\", please rate again.".format(input))
        }
      }
      userRatings += Tuple3(movieId, title, rating)
    }
    userRatings.toList
  }
```

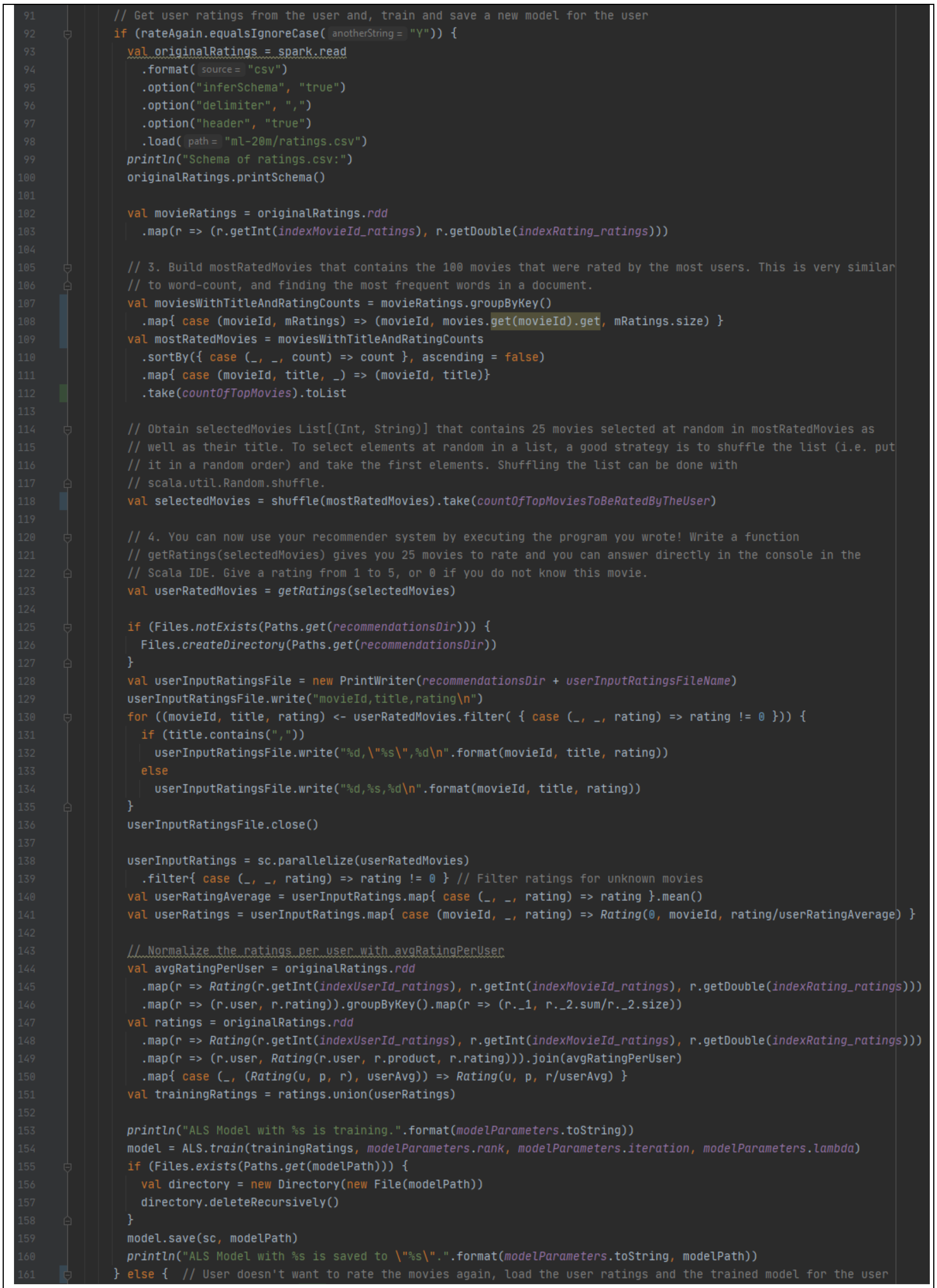
5. After finishing the rating, your program should display the top 20 movies that you might like. Look at the recommendations, are you happy about your recommendations? Comment.

If user’s ratings and related trained model exist, ask the user whether he/she wants to rate again.



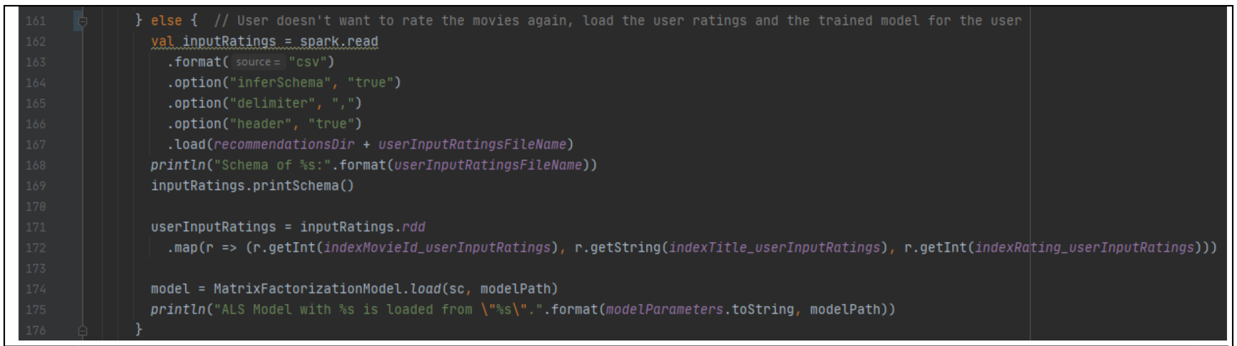
```
var rateAgain: String = ""
// If User Ratings File and Related Matrix Factorization Model exists, ask the user whether he/she want to rate
// again. If he/she chooses to rate again, get the new ratings and, train and save a new model. Otherwise, load
// the existing model.
if (Files.exists(Paths.get(recommendationsDir + userInputRatingsFileName))
  && Files.exists(Paths.get(recommendationsDir + modelParameters.getShortDescription))) {
  while (!rateAgain.equalsIgnoreCase( anotherString = "Y") && !rateAgain.equalsIgnoreCase( anotherString = "N")) {
    rateAgain = readLine("Do you want to rate movies again? (Y/N): ")
  }
} else {
  rateAgain = "Y"
}
```

If user’s ratings or related trained model don’t exist, or the user wants to rate the movies again:

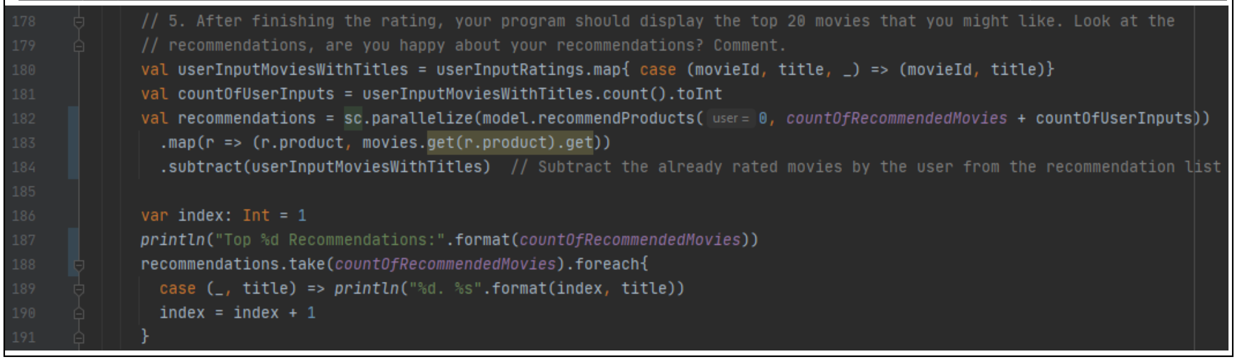


```scala
    // Get user ratings from the user and, train and save a new model for the user
    if (rateAgain.equalsIgnoreCase("Y")) {
      val originalRatings = spark.read
        .format("csv")
        .option("inferSchema", "true")
        .option("delimiter", ",")
        .option("header", "true")
        .load("ml-20m/ratings.csv")
      println("Schema of ratings.csv:")
      originalRatings.printSchema()

      val movieRatings = originalRatings.rdd
        .map(r => (r.getInt(indexMovieId_ratings), r.getDouble(indexRating_ratings)))

      // 3. Build mostRatedMovies that contains the 100 movies that were rated by the most users. This is very similar
      // to word-count, and finding the most frequent words in a document.
      val moviesWithTitleAndRatingCounts = movieRatings.groupByKey()
        .map{ case (movieId, mRatings) => (movieId, movies.get(movieId).get, mRatings.size) }
      val mostRatedMovies = moviesWithTitleAndRatingCounts
        .sortBy({ case (_, _, count) => count }, ascending = false)
        .map{ case (movieId, title, _) => (movieId, title)}
        .take(countOfTopMovies).toList

      // Obtain selectedMovies List[(Int, String)] that contains 25 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.
      val selectedMovies = shuffle(mostRatedMovies).take(countOfTopMoviesToBeRatedByTheUser)

      // 4. You can now use your recommender system by executing the program you wrote! Write a function
      // getRatings(selectedMovies) gives you 25 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.
      val userRatedMovies = getRatings(selectedMovies)

      if (Files.notExists(Paths.get(recommendationsDir))) {
        Files.createDirectory(Paths.get(recommendationsDir))
      }
      val userInputRatingsFile = new PrintWriter(recommendationsDir + userInputRatingsFileName)
      userInputRatingsFile.write("movieId,title,rating\n")
      for ((movieId, title, rating) <- userRatedMovies.filter( { case (_, _, rating) => rating != 0 })) {
        if (title.contains(","))
          userInputRatingsFile.write("%d,\"%s\",%d\n".format(movieId, title, rating))
        else
          userInputRatingsFile.write("%d,%s,%d\n".format(movieId, title, rating))
      }
      userInputRatingsFile.close()

      userInputRatings = sc.parallelize(userRatedMovies)
        .filter{ case (_, _, rating) => rating != 0 } // Filter ratings for unknown movies
      val userRatingAverage = userInputRatings.map{ case (_, _, rating) => rating }.mean()
      val userRatings = userInputRatings.map{ case (movieId, _, rating) => Rating(0, movieId, rating/userRatingAverage) }

      // Normalize the ratings per user with avgRatingPerUser
      val avgRatingPerUser = originalRatings.rdd
        .map(r => Rating(r.getInt(indexUserId_ratings), r.getInt(indexMovieId_ratings), r.getDouble(indexRating_ratings)))
        .map(r => (r.user, r.rating)).groupByKey().map(r => (r._1, r._2.sum/r._2.size))
      val ratings = originalRatings.rdd
        .map(r => Rating(r.getInt(indexUserId_ratings), r.getInt(indexMovieId_ratings), r.getDouble(indexRating_ratings)))
        .map(r => (r.user, Rating(r.user, r.product, r.rating))).join(avgRatingPerUser)
        .map{ case (_, (Rating(u, p, r), userAvg)) => Rating(u, p, r/userAvg) }
      val trainingRatings = ratings.union(userRatings)

      println("ALS Model with %s is training.".format(modelParameters.toString))
      model = ALS.train(trainingRatings, modelParameters.rank, modelParameters.iteration, modelParameters.lambda)
      if (Files.exists(Paths.get(modelPath))) {
        val directory = new Directory(new File(modelPath))
        directory.deleteRecursively()
      }
      model.save(sc, modelPath)
      println("ALS Model with %s is saved to \"%s\".".format(modelParameters.toString, modelPath))
    } else {  // User doesn't want to rate the movies again, load the user ratings and the trained model for the user
```

Otherwise:



```
} else {  // User doesn't want to rate the movies again, load the user ratings and the trained model for the user
  val inputRatings = spark.read
    .format("csv")
    .option("inferSchema", "true")
    .option("delimiter", ",")
    .option("header", "true")
    .load(recommendationsDir + userInputRatingsFileName)
  println("Schema of %s:".format(userInputRatingsFileName))
  inputRatings.printSchema()

  userInputRatings = inputRatings.rdd
    .map(r => (r.getInt(indexMovieId_userInputRatings), r.getString(indexTitle_userInputRatings), r.getInt(indexRating_userInputRatings)))

  model = MatrixFactorizationModel.load(sc, modelPath)
  println("ALS Model with %s is loaded from \"%s\".".format(modelParameters.toString, modelPath))
}
```



```
// 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.
val userInputMoviesWithTitles = userInputRatings.map{ case (movieId, title, _) => (movieId, title)}
val countOfUserInputs = userInputMoviesWithTitles.count().toInt
val recommendations = sc.parallelize(model.recommendProducts(0, countOfRecommendedMovies + countOfUserInputs))
  .map(r => (r.product, movies.get(r.product).get))
  .subtract(userInputMoviesWithTitles)  // Subtract the already rated movies by the user from the recommendation list

var index: Int = 1
println("Top %d Recommendations:".format(countOfRecommendedMovies))
recommendations.take(countOfRecommendedMovies).foreach{
  case (_, title) => println("%d. %s".format(index, title))
  index = index + 1
}
```

I rated the following 25 movies from the top rated 100 movies:

- Jurassic Park (1993): **4**
- Matrix, The (1999): **5**
- Star Trek: Generations (1994):
- Good Will Hunting (1997):
- Taxi Driver (1976): **4**
- Toy Story (1995): **5**
- American Beauty (1999):
- Aladdin (1992):
- Crouching Tiger, Hidden Dragon (Wo hu cang long) (2000):
- Blade Runner (1982):
- One Flew Over the Cuckoo's Nest (1975):
- Willy Wonka & the Chocolate Factory (1971): **4**
- Babe (1995):
- Fifth Element, The (1997):
- Braveheart (1995): **5**
- Clear and Present Danger (1994):
- Silence of the Lambs, The (1991): **4**
- Sleepless in Seattle (1993):
- Interview with the Vampire: The Vampire Chronicles (1994):
- Seven (a.k.a. Se7en) (1995):
- Aliens (1986):
- Shrek (2001): **5**
- Fight Club (1999): **5**
- X-Men (2000): **5**
- Dances with Wolves (1990):

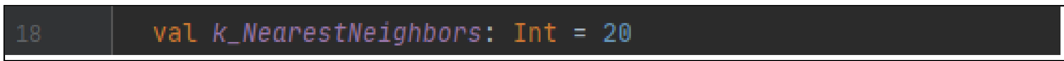
And, I got the following movies as top 20 recommendation:

1. Bo Burnham: what. (2013)
2. Miss You Can Do It (2013)
3. Lord of the Rings: The Return of the King, The (2003)
4. Batman Begins (2005)
5. Usual Suspects, The (1995)
6. Star Wars: Episode IV - A New Hope (1977)
7. Pearl Jam: Immagine in Cornice - Live in Italy 2006 (2007)
8. Incredibles, The (2004)
9. Liberty (1929)
10. 2013 Rock and Roll Hall of Fame Induction Ceremony, The (2013)
11. Star Wars: Episode V - The Empire Strikes Back (1980)
12. Lost Thing, The (2010)
13. Boys Don't Cry (Chlopaki nie placza) (2000)
14. Spider-Man 2 (2004)
15. Avengers, The (2012)
16. Iron Man (2008)
17. Smashing Pumpkins: Vieuphoria (1994)
18. X2: X-Men United (2003)
19. Pirates of the Caribbean: The Curse of the Black Pearl (2003)
20. Toy Story 2 (1999)

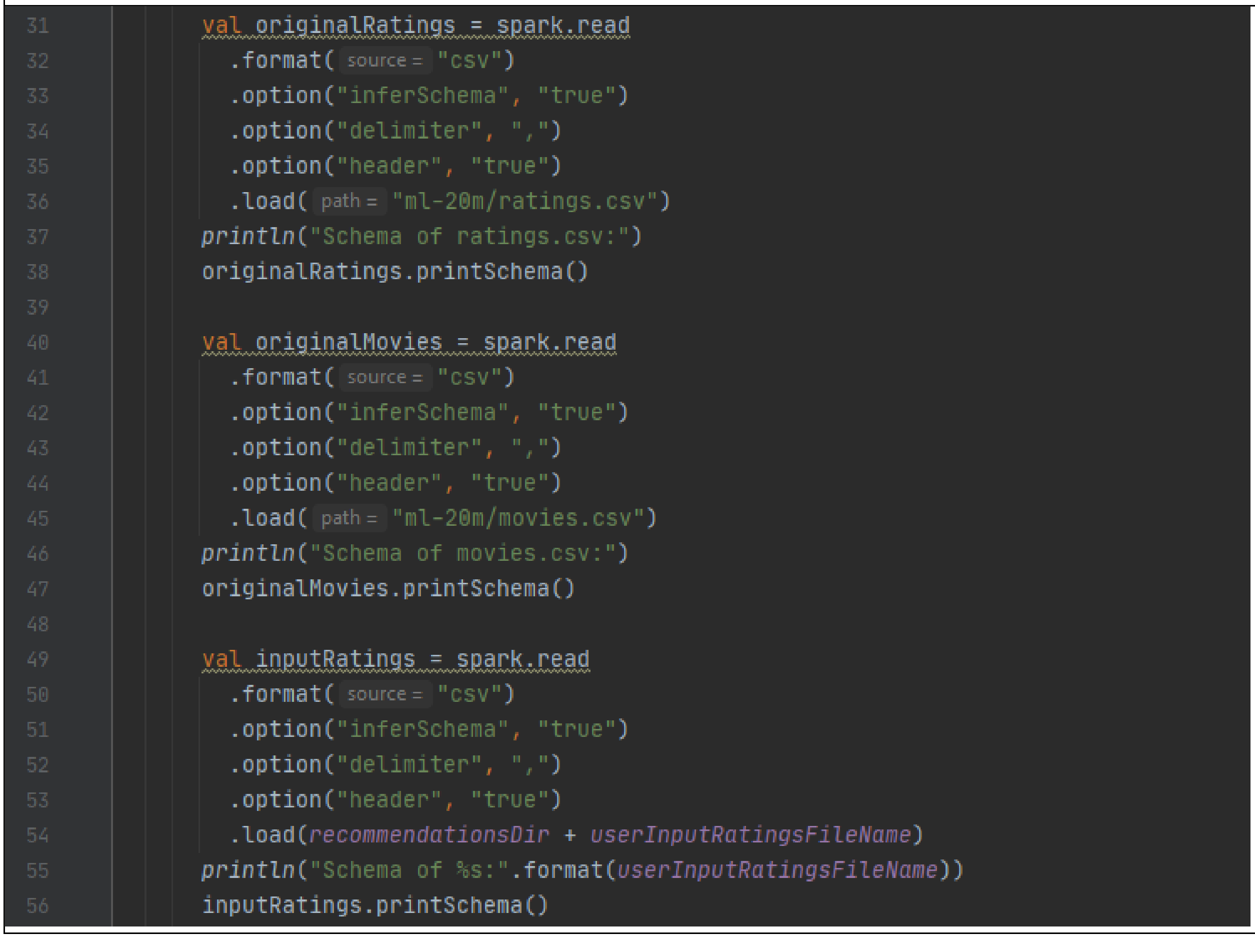
I do think that this recommender can recommend better if I rate more movies. I rated 10 movies out of 25 movies and got at least 12 good movie recommendations (colored green) out of 20 movies. I can say that I am happy with these recommendations because there are 12 movies that I already known and liked, and there is chance that I can like some of the rest.

Part 2: Content-based Nearest Neighbors

For collaborative filtering, you relied purely on rankings and did not use movie attributes (genres) at all. For this part of the assignment, you will use a different method: content-based recommendation. Our goal here is to build for each user a vector of features (genres) describing their interests. Then, we will find the k users that are most similar using those vectors and cosine similarity to obtain a recommendation.

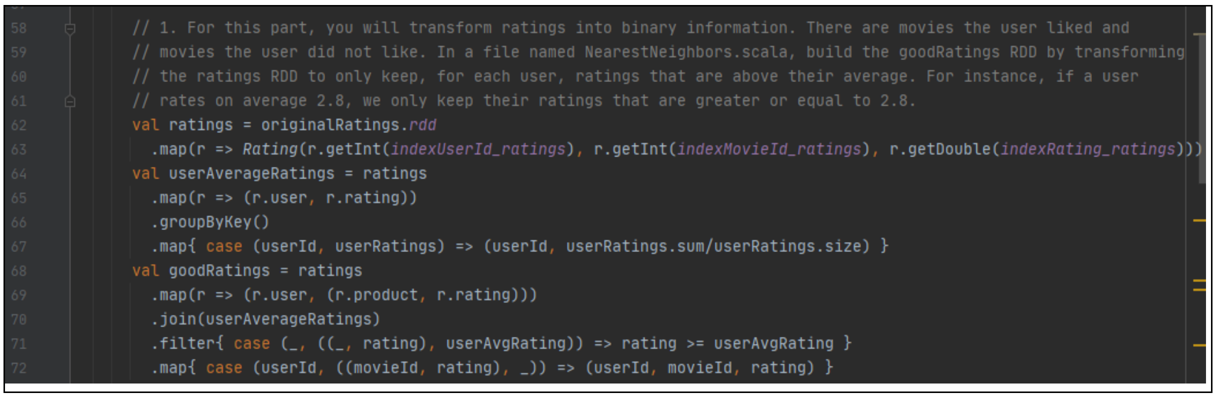


```
val k_NearestNeighbors: Int = 20
```



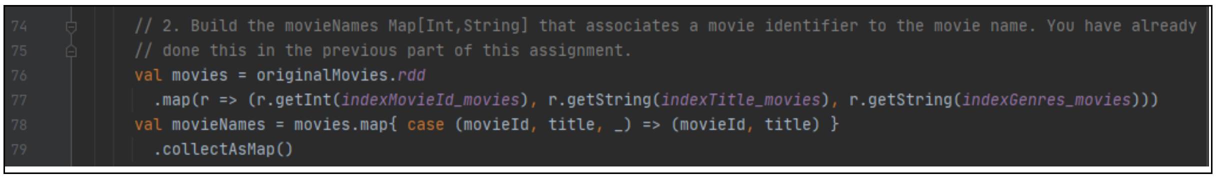
```
val originalRatings = spark.read
  .format( source = "csv")
  .option("inferSchema", "true")
  .option("delimiter", ",")
  .option("header", "true")
  .load( path = "ml-20m/ratings.csv")
println("Schema of ratings.csv:")
originalRatings.printSchema()

val originalMovies = spark.read
  .format( source = "csv")
  .option("inferSchema", "true")
  .option("delimiter", ",")
  .option("header", "true")
  .load( path = "ml-20m/movies.csv")
println("Schema of movies.csv:")
originalMovies.printSchema()

val inputRatings = spark.read
  .format( source = "csv")
  .option("inferSchema", "true")
  .option("delimiter", ",")
  .option("header", "true")
  .load(recommendationsDir + userInputRatingsFileName)
println("Schema of %s:".format(userInputRatingsFileName))
inputRatings.printSchema()
```

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.



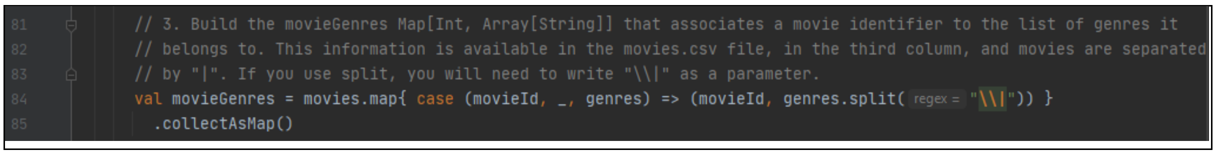
```scala
// 1. For this part, you will transform ratings into binary information. There are movies the user liked and
// movies the user did not like. In a file named NearestNeighbors.scala, build the goodRatings RDD by transforming
// the ratings RDD to only keep, for each user, ratings that are above 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.
val ratings = originalRatings.rdd
  .map(r => Rating(r.getInt(indexUserId_ratings), r.getInt(indexMovieId_ratings), r.getDouble(indexRating_ratings)))
val userAverageRatings = ratings
  .map(r => (r.user, r.rating))
  .groupByKey()
  .map{ case (userId, userRatings) => (userId, userRatings.sum/userRatings.size) }
val goodRatings = ratings
  .map(r => (r.user, (r.product, r.rating)))
  .join(userAverageRatings)
  .filter{ case (_, ((_, rating), userAvgRating)) => rating >= userAvgRating }
  .map{ case (userId, ((movieId, rating), _)) => (userId, movieId, rating) }
```

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.



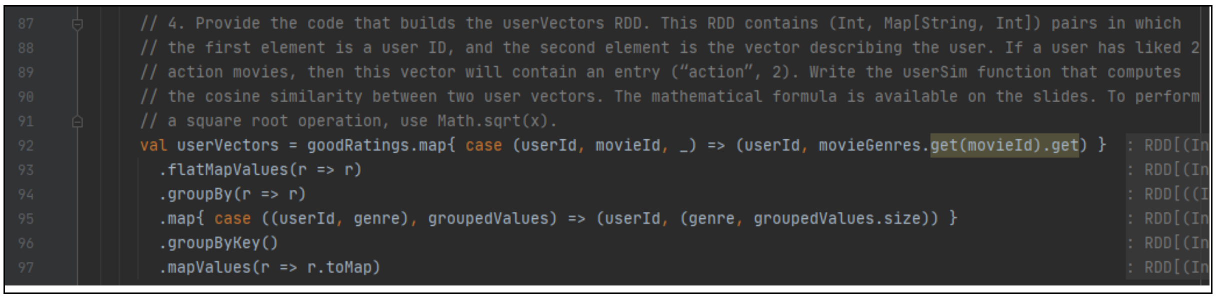
```scala
// 2. Build the movieNames Map[Int,String] that associates a movie identifier to the movie name. You have already
// done this in the previous part of this assignment.
val movies = originalMovies.rdd
  .map(r => (r.getInt(indexMovieId_movies), r.getString(indexTitle_movies), r.getString(indexGenres_movies)))
val movieNames = movies.map{ case (movieId, title, _) => (movieId, title) }
  .collectAsMap()
```

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.

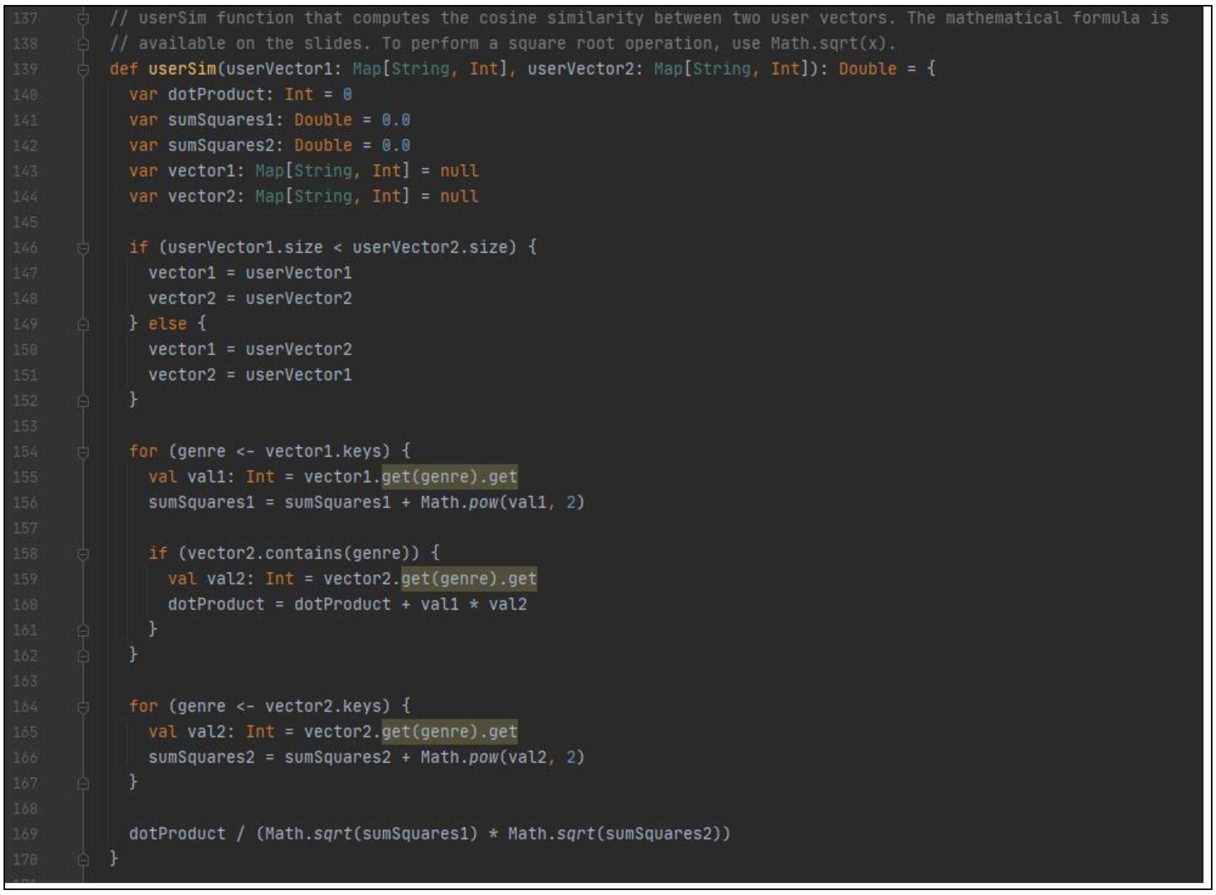


```scala
// 3. Build the movieGenres Map[Int, Array[String]] that associates a movie identifier to the list of genres it
// belongs to. This information is available in the movies.csv file, in the third column, and movies are separated
// by "|". If you use split, you will need to write "\\|" as a parameter.
val movieGenres = movies.map{ case (movieId, _, genres) => (movieId, genres.split( regex = "\\|")) }
  .collectAsMap()
```

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).



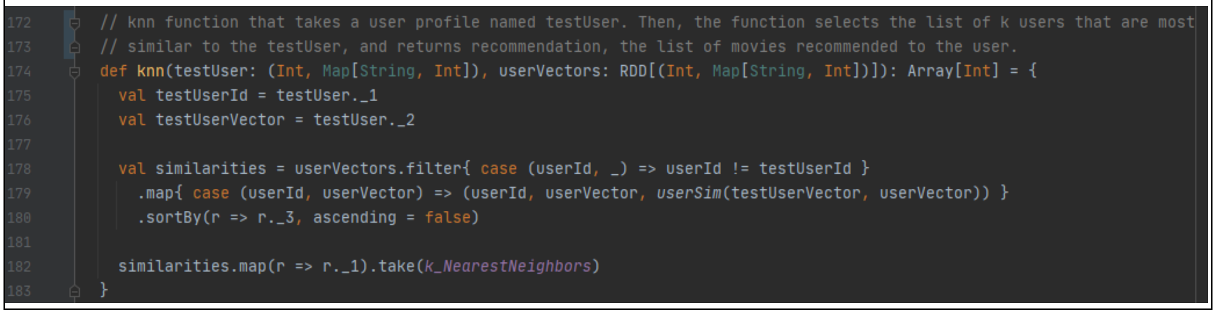
```scala
// 4. Provide the code that builds the userVectors RDD. This RDD contains (Int, Map[String, Int]) pairs in which
// the first element is a user ID, and the second element is the vector describing the user. If a user has liked 2
// action movies, then this vector will contain an entry ("action", 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).
val userVectors = goodRatings.map{ case (userId, movieId, _) => (userId, movieGenres.get(movieId).get) }  : RDD[(In
  .flatMapValues(r => r)                                                                                  : RDD[(In
  .groupBy(r => r)                                                                                        : RDD[((I
  .map{ case ((userId, genre), groupedValues) => (userId, (genre, groupedValues.size)) }                  : RDD[(In
  .groupByKey()                                                                                           : RDD[(In
  .mapValues(r => r.toMap)                                                                                : RDD[(In
```



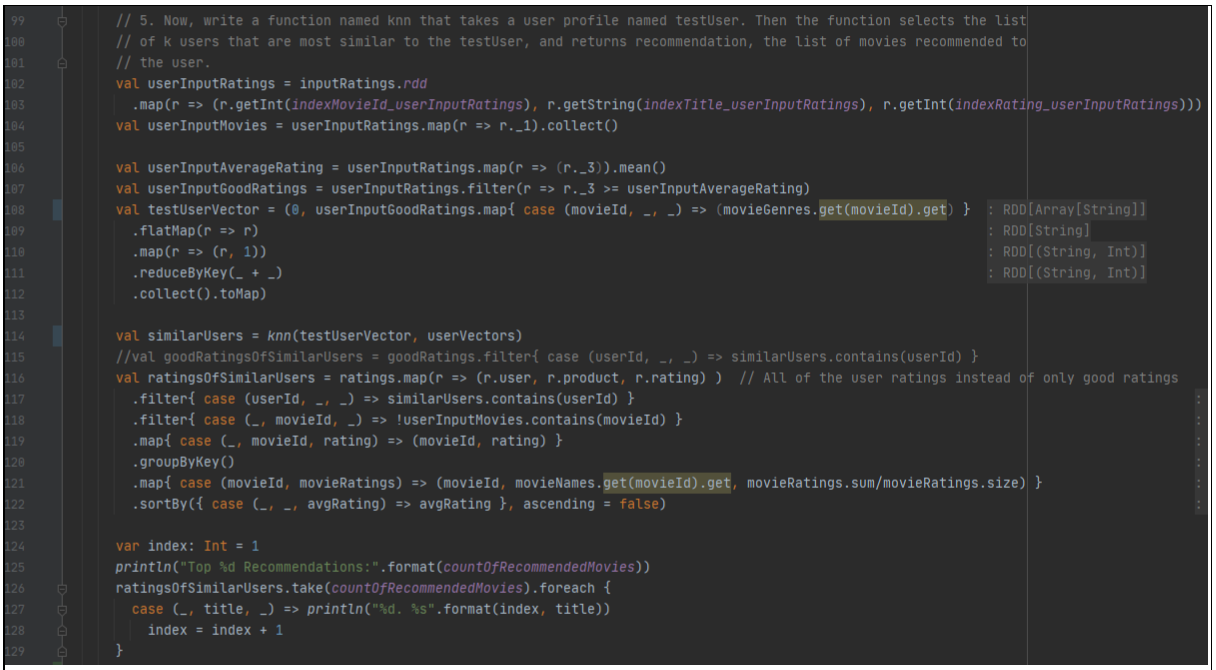
```scala
// userSim function that computes the cosine similarity between two user vectors. The mathematical formula is
// available on the slides. To perform a square root operation, use Math.sqrt(x).
def userSim(userVector1: Map[String, Int], userVector2: Map[String, Int]): Double = {
  var dotProduct: Int = 0
  var sumSquares1: Double = 0.0
  var sumSquares2: Double = 0.0
  var vector1: Map[String, Int] = null
  var vector2: Map[String, Int] = null

  if (userVector1.size < userVector2.size) {
    vector1 = userVector1
    vector2 = userVector2
  } else {
    vector1 = userVector2
    vector2 = userVector1
  }

  for (genre <- vector1.keys) {
    val val1: Int = vector1.get(genre).get
    sumSquares1 = sumSquares1 + Math.pow(val1, 2)

    if (vector2.contains(genre)) {
      val val2: Int = vector2.get(genre).get
      dotProduct = dotProduct + val1 * val2
    }
  }

  for (genre <- vector2.keys) {
    val val2: Int = vector2.get(genre).get
    sumSquares2 = sumSquares2 + Math.pow(val2, 2)
  }

  dotProduct / (Math.sqrt(sumSquares1) * Math.sqrt(sumSquares2))
}
```

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 recommendation, the list of movies recommended to the user.

I made the ***knn*** function to output k most similar users to the testUser.



```scala
// knn function 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 recommendation, the list of movies recommended to the user.
def knn(testUser: (Int, Map[String, Int]), userVectors: RDD[(Int, Map[String, Int])]): Array[Int] = {
  val testUserId = testUser._1
  val testUserVector = testUser._2

  val similarities = userVectors.filter{ case (userId, _) => userId != testUserId }
    .map{ case (userId, userVector) => (userId, userVector, userSim(testUserVector, userVector)) }
    .sortBy(r => r._3, ascending = false)

  similarities.map(r => r._1).take(k_NearestNeighbors)
}
```



```scala
// 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 recommendation, the list of movies recommended to
// the user.
val userInputRatings = inputRatings.rdd
  .map(r => (r.getInt(indexMovieId_userInputRatings), r.getString(indexTitle_userInputRatings), r.getInt(indexRating_userInputRatings)))
val userInputMovies = userInputRatings.map(r => r._1).collect()

val userInputAverageRating = userInputRatings.map(r => (r._3)).mean()
val userInputGoodRatings = userInputRatings.filter(r => r._3 >= userInputAverageRating)
val testUserVector = (0, userInputGoodRatings.map{ case (movieId, _, _) => (movieGenres.get(movieId).get) }  : RDD[Array[String]]
  .flatMap(r => r)                                                                                              : RDD[String]
  .map(r => (r, 1))                                                                                             : RDD[(String, Int)]
  .reduceByKey(_ + _)                                                                                           : RDD[(String, Int)]
  .collect().toMap)

val similarUsers = knn(testUserVector, userVectors)
//val goodRatingsOfSimilarUsers = goodRatings.filter{ case (userId, _, _) => similarUsers.contains(userId) }
val ratingsOfSimilarUsers = ratings.map(r => (r.user, r.product, r.rating) )  // All of the user ratings instead of only good ratings
  .filter{ case (userId, _, _) => similarUsers.contains(userId) }
  .filter{ case (_, movieId, _) => !userInputMovies.contains(movieId) }
  .map{ case (_, movieId, rating) => (movieId, rating) }
  .groupByKey()
  .map{ case (movieId, movieRatings) => (movieId, movieNames.get(movieId).get, movieRatings.sum/movieRatings.size) }
  .sortBy({ case (_, _, avgRating) => avgRating }, ascending = false)

var index: Int = 1
println("Top %d Recommendations:".format(countOfRecommendedMovies))
ratingsOfSimilarUsers.take(countOfRecommendedMovies).foreach {
  case (_, title, _) => println("%d. %s".format(index, title))
    index = index + 1
}
```

I used the following formula to calculate prediction for the movie *i* of test user (user *x*):

$\widehat{r_{xi}} = \frac{1}{k} \sum_{y \in N} r_{yi}$ where r_{xi} is the vector of user ***x***'s rating on movie ***I*** and ***N*** is the set of ***k*** users most similar to ***x*** who have rated item ***i.***

Also, the following formula could be used to calculate the prediction:

$\widehat{r_{xi}} = \frac{\sum_{y \in N} s_{xy} r_{yi}}{\sum_{y \in N} s_{xy}}$ where s_{xy} is the cosine similarity of user ***x*** and user ***y***.

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 used my movie ratings that was used in Part 1:

- Jurassic Park (1993): **4**
- Matrix, The (1999): **5**
- Taxi Driver (1976): **4**
- Toy Story (1995): **5**
- Willy Wonka & the Chocolate Factory (1971): **4**
- Braveheart (1995): **5**
- Silence of the Lambs, The (1991): **4**
- Shrek (2001): **5**
- Fight Club (1999): **5**
- X-Men (2000): **5**

And my user vector was:

(Action,4), (Adventure,3), (Children,2), (Sci-Fi,2), (Drama,2), (Fantasy,2), (Comedy,2), (Thriller,2), (Animation,2), (Crime,1), (Romance,1), (War,1)

This time, my top 20 recommendations:

1. Prince of Tides, The (1991)
2. Contact (1997)
3. Hard Rain (1998)
4. Lethal Weapon (1987)
5. Star Trek IV: The Voyage Home (1986)

11. Twelve Monkeys (a.k.a. 12 Monkeys) (1995)
12. Lord of the Rings: The Two Towers, The (2002)
13. Taking of Pelham One Two Three, The (1974)
14. Dirty Dozen, The (1967)
15. Caddyshack (1980)
16. Princess Mononoke (Mononoke-hime) (1997)
17. Charlie and the Chocolate Factory (2005)

6. Butch Cassidy and the Sundance Kid (1969)
7. Network (1976)
8. U.S. Marshals (1998)
9. Superman (1978)
10. Cocoon: The Return (1988)
18. Net, The (1995)
19. Patriot, The (2000)
20. Nick of Time (1995)

This time, I got at least 5 good movie recommendations (colored green) out of 20 movies. I can say that I am not satisfied with these recommendations as the other recommendation model because there are only 5 movie that I already known and liked whereas it was 12 in the other model. Maybe, there is a chance that I like some of the rest.
I tried different ***k*** values such as 5, 10, 15, 20 and using **5** was the most promising one with only 5 good movie recommendation. Maybe, I am the reason that I judge poorly this recommendation model as I am not watching a lot of movies and I don't know most of the movies in the dataset. I will try both models with more user ratings and I will try them on my friends.

Extras

1. I increased the number of rated movies from the top 100 movies to 100 from 25.

I rated the movies below from the top 100 movies:
- Clockwork Orange, A (1971): **2**
- Gladiator (2000): **5**
- Toy Story (1995): **5**
- Forrest Gump (1994): **5**
- Fight Club (1999): **5**
- Shawshank Redemption, The (1994): **4**
- Lord of the Rings: The Two Towers, The (2002): **5**
- Braveheart (1995): **5**
- Men in Black (a.k.a. MIB) (1997): **4**
- Shrek (2001): **5**
- Lord of the Rings: The Fellowship of the Ring, The (2001): **5**
- Mask, The (1994): **4**
- X-Men (2000): **5**
- Léon: The Professional (a.k.a. The Professional) (Léon) (1994): **4**
- Lion King, The (1994): **5**
- Home Alone (1990): **5**
- Batman (1989): **5**
- Taxi Driver (1976): **4**
- Titanic (1997): **4**
- Pulp Fiction (1994): **5**
- GoldenEye (1995): **5**
- Pirates of the Caribbean: The Curse of the - Black Pearl (2003): **5**
- Godfather, The (1972): **4**
- Godfather: Part II, The (1974): **4**
- Sixth Sense, The (1999): **3**
- Die Hard (1988): **5**
- Die Hard: With a Vengeance (1995): **5**
- Mission: Impossible (1996): **5**
- Batman Forever (1995): **5**
- Lord of the Rings: The Return of the King, The (2003): **5**
- Saving Private Ryan (1998): **5**
- Matrix, The (1999): **5**
- Silence of the Lambs, The (1991): **3**

Collaborative Filtering Recommendations	**Content-Based Nearest Neighbors Recommendations**
1. Grave Decisions (Wer früher stirbt, ist länger tot) (2006)	**1.** King Arthur (2004)
2. Mater and the Ghostlight (2006)	**2.** Jet Li's Fearless (Huo Yuan Jia) (2006)
3. Ocho apellidos vascos (2014)	**3.** Prefontaine (1997)
	4. Ip Man (2008)

4. Brooklyn Castle (2012) **5.** Distant Voices, Still Lives (1988) **6.** Lust for Love (2014) **7.** Good Dick (2008) **8.** Christmas Toy, The (1986) **9.** Batman Begins (2005) **10.** Didier (1997) **11.** Deathstalker II (1987) **12**. Pearl Jam: Immagine in Cornice - Live in Italy 2006 (2007) **13.** PK (2014) **14.** Smashing Pumpkins: If All Goes Wrong (2008) **15.** "Diebuster ""Top wo Narae 2"" (2004)" **16.** Tom Segura: Completely Normal (2014) **17.** One Piece Film: Strong World (2009) **18.** Crazy Stone (Fengkuang de shitou) (2006) **19.** Starsuckers (2009) **20.** Indiana Jones and the Last Crusade (1989)	**5.** Beautiful Mind, A (2001) **6.** Boondock Saints, The (2000) **7.** Black Knight (2001) **8.** Lord of the Rings, The (1978) **9.** Prestige, The (2006) **10.** Man on Fire (2004) **11.** Hitch (2005) **12.** Chariots of Fire (1981) **13.** Dead Poets Society (1989) **14.** Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb (1964) **15.** Dumb & Dumber (Dumb and Dumber) (1994) **16.** Ice Age (2002) **17.** Rock, The (1996) **18.** Bourne Ultimatum, The (2007) **19.** Emperor's New Groove, The (2000) **20.** Good Morning, Vietnam (1987)
My user vector: (Action,14), (Adventure,12), (Drama,8), (Thriller,8), (Comedy,7), (Crime,6), (Fantasy,6), (Children,4), (War,3), (Animation,3), (Sci-Fi,2), (Romance,2), (Musical,1), (Mystery,1), (IMAX,1) I think that recommendations of content-based nearest neighbors are better.	

2. I increased the number of top movies to 250 from 100 and I rated all of them.

I rated the movies below from the top 250 movies:

- Clockwork Orange, A (1971): **2**	- Dark Knight, The (2008): **5**
- Die Hard: With a Vengeance (1995): **5**	- Pirates of the Caribbean: The Curse of the Black Pearl (2003): **5**
- Pulp Fiction (1994): **5**	- Spider-Man (2002): **5**
- Léon: The Professional (a.k.a. The - Professional) (Léon) (1994): **4**	- Batman Forever (1995): **5**
- Matrix, The (1999): **5**	- Monsters, Inc. (2001): **5**
- Mission: Impossible II (2000): **5**	- Harry Potter and the Sorcerer's Stone (a.k.a. Harry Potter and the Philosopher's Stone) (2001): **5**
- Home Alone (1990): **5**	- Taxi Driver (1976): **4**
- Lord of the Rings: The Two Towers, The (2002): **5**	- Beautiful Mind, A (2001): **5**
- Lord of the Rings: The Return of the King, The (2003): **5**	- Die Hard 2 (1990): **5**
- Die Hard (1988): **5**	- X-Men (2000): **5**
- Top Gun (1986): **5**	- Shawshank Redemption, The (1994): **4**
- Truman Show, The (1998): **5**	- Forrest Gump (1994): **5**
- Braveheart (1995): **5**	- Matrix Reloaded, The (2003): **5**
- Charlie's Angels (2000): **4**	- Chicken Run (2000): **5**
- GoldenEye (1995): **5**	- Spider-Man 2 (2004): **5**

- Sixth Sense, The (1999): **3** - Batman Begins (2005): **5** - Dead Poets Society (1989): **4** - Finding Nemo (2003): **5** - Gladiator (2000): **5** - Godfather: Part II, The (1974): **4** - Saving Private Ryan (1998): **5** - Titanic (1997): **4** - Lion King, The (1994): **5** - Toy Story (1995): **5** - Full Metal Jacket (1987): **4** - Mask, The (1994): **4**	- Silence of the Lambs, The (1991): **3** - Shrek (2001): **5** - Fight Club (1999): **5** - Mission: Impossible (1996): **5** - Men in Black (a.k.a. MIB) (1997): **4** - Toy Story 2 (1999): **5** - X2: X-Men United (2003): **5** - American Pie (1999): **5** - Godfather, The (1972): **4** - Lord of the Rings: The Fellowship of the Ring, The (2001): **5**
Collaborative Filtering Recommendations	**Content-Based Nearest Neighbors Recommendations**
1. Grave Decisions (Wer früher stirbt, ist länger tot) (2006)	**1.** Defiance (2008)
2. Mater and the Ghostlight (2006)	**2.** Cloudy with a Chance of Meatballs (2009)
3. Broken Sky (El cielo dividido) (2006)	**3.** Star Trek IV: The Voyage Home (1986)
4. Ocean's Eleven (2001)	**4.** Kung Fu Panda (2008)
5. Ocho apellidos vascos (2014)	**5.** The Count of Monte Cristo (2002)
6. Brooklyn Castle (2012)	**6.** Elf (2003)
7. Tito and Me (Tito i ja) (1992)	**7.** Game, The (1997)
8. Distant Voices, Still Lives (1988)	**8.** Taken (2008)
9. Lust for Love (2014)	**9.** Blind Side, The (2009)
10. Family Meeting (2007)	**10.** Talladega Nights: The Ballad of Ricky Bobby (2006)
11. Surf's Up (2007)	**11.** Spirited Away (Sen to Chihiro no kamikakushi) (2001)
12. Good Dick (2008)	**12.** Hero (Ying xiong) (2002)
13. Agony and the Ecstasy of Phil Spector, The (2009)	**13.** Howl's Moving Castle (Hauru no ugoku shiro) (2004)
14. Sunny (Sseo-ni) (2011)	**14.** Blood Diamond (2006)
15. Absolute Giganten (1999)	**15.** Boondock Saints, The (2000)
16. Pearl Jam: Immagine in Cornice - Live in Italy 2006 (2007)	**16.** 3-Iron (Bin-jip) (2004)
17. Watermark (2014)	**17.** Cowboy Bebop: The Movie (Cowboy Bebop: Tengoku no Tobira) (2001)
18. Loners (Samotári) (2000)	**18.** Into the Woods (1991)
19. One Piece Film: Strong World (2009)	**19.** Harry Potter and the Half-Blood Prince (2009)
20. Jim Gaffigan: Obsessed (2014)	**20.** Sneakers (1992)
My user vector: (Action,22), (Adventure,22), (Comedy,13), (Thriller,12), (Drama,11), (Children,9), (Fantasy,9), (Crime,7), (Sci-Fi,7), (Animation,7), (IMAX,5), (Romance,5), (War,3), (Musical,1), (Mystery,1) I still think that recommendations of content-based nearest neighbors are better.	

I have expected that recommendations of collaborative filtering would get better when I increased my rated movie count; however, I cannot say that it got any better in terms of my thought of what movie recommendation is considered good.

3. I tried the recommendation models on two of my friends:

She rated 25 movies from the top 100 movies:	
- Pulp Fiction (1994): **5**	- Star Trek: Generations (1994): **4**
- Ghostbusters (a.k.a. Ghost Busters) (1984): **4**	- Sixth Sense, The (1999): **4**
- Pirates of the Caribbean: The Curse of the Black Pearl (2003): **5**	- Pretty Woman (1990): **3**
- Usual Suspects, The (1995): **5**	- Apollo 13 (1995): **4**
- Twelve Monkeys (a.k.a. 12 Monkeys) (1995): **5**	- Alien (1979): **4**
- Matrix, The (1999): **4**	- X-Men (2000): **4**
- One Flew Over the Cuckoo's Nest (1975): **5**	- Toy Story (1995): **5**
Collaborative Filtering Recommendations	**Content-Based Nearest Neighbors Recommendations**
1. Grave Decisions (Wer früher stirbt, ist länger tot) (2006)	**1.** Happy Gilmore (1996)
2. Geri's Game (1997)	**2.** Lord of the Rings: The Two Towers, The (2002)
3. Winnie the Pooh and the Honey Tree (1966)	**3.** Finding Nemo (2003)
4. Marc Maron: Thinky Pain (2013)	**4.** Lord of the Rings: The Fellowship of the Ring, The (2001)
5. The War at Home (1979)	**5.** Silence of the Lambs, The (1991)
6. Monty Python's Life of Brian (1979)	**6.** American History X (1998)
7. Wallace & Gromit: The Wrong Trousers (1993)	**7.** Incredibles, The (2004)
8. Patton Oswalt: Werewolves and Lollipops (2007)	**8.** Star Trek: First Contact (1996)
9. Lifted (2006)	**9.** Philadelphia (1993)
10. Liberty (1929)	**10.** Princess Bride, The (1987)
11. Up (2009)	**11.** Shawshank Redemption, The (1994)
12. Muppet Family Christmas, A (1987)	**12.** Heat (1995)
13. Intervista (1987)	**13.** Monsters, Inc. (2001)
14. Bill Hicks: Sane Man (1989)	**14.** Nightmare Before Christmas, The (1993)
15. Lost Thing, The (2010)	**15.** Seven (a.k.a. Se7en) (1995)
16. Wallace & Gromit: A Close Shave (1995)	**16.** Better Off Dead... (1985)
17. Nazis: A Warning from History, The (1997)	**17.** Lord of the Rings: The Return of the King, The (2003)
18. Misérables in Concert, Les (1996)	**18.** Finding Neverland (2004)
19. Balance (1989)	**19.** Memento (2000)
20. Casting By (2012)	**20.** Untouchables, The (1987)
Her user vector: (Comedy,3), (Thriller,3), (Crime,2), (Adventure,2), (Mystery,2), (Drama,2), (Fantasy,2), (Action,1), (Children,1), (Sci-Fi,1), (Animation,1) She said that recommendations of content-based nearest neighbors were better.	

He rated 25 movies from the top 100 movies:

- One Flew Over the Cuckoo's Nest (1975): **5** - Shrek (2001): **4** - Mask, The (1994): **4** - X-Men (2000): **3** - Léon: The Professional (a.k.a. The Professional) (Léon) (1994): **5** - Lord of the Rings: The Fellowship of the Ring, The (2001): **5** - Shawshank Redemption, The (1994): **5**	- Twelve Monkeys (a.k.a. 12 Monkeys) (1995): **5** - Godfather: Part II, The (1974): **5** - Fargo (1996): **5** - Star Wars: Episode I - The Phantom Menace (1999): **4** - Silence of the Lambs, The (1991): **4** - Fight Club (1999): **5**
Collaborative Filtering Recommendations	**Content-Based Nearest Neighbors Recommendations**
1. Grave Decisions (Wer früher stirbt, ist länger tot) (2006) **2.** Fingersmith (2005) **3.** Still Life (2013) **4.** Seven Samurai (Shichinin no samurai) (1954) **5.** Herod's Law (Ley de Herodes, La) (2000) **6.** Absolute Giganten (1999) **7.** I Belong (Som du ser meg) (2012) **8.** Chinaman (Kinamand) (2005) **9.** Crooks in Clover (a.k.a. Monsieur Gangster) (Les tontons flingueurs) (1963) **10.** Godfather, The (1972) **11.** Ernest & Célestine (Ernest et Célestine) (2012) **12.** Fallen Art (Sztuka spadania) (2004) **13.** Bill Hicks: Sane Man (1989) **14.** Good, the Bad and the Ugly, The (Buono, il brutto, il cattivo, Il) (1966) **15.** Pulp Fiction (1994) **16.** Marathon Family, The (Maratonci Trce Pocasni Krug) (1982) **17.** Family Resemblances (Un air de famille) (1996) **18.** Nazis: A Warning from History, The (1997) **19.** Jim Jefferies: Alcoholocaust (2010) **20.** Love Me No More (Deux jours à tuer) (2008)	**1.** Training Day (2001) **2.** Cabin in the Woods, The (2012) **3.** Felon (2008) **4.** Bill Hicks: Revelations (1993) **5.** Town, The (2010) **6.** Sting, The (1973) **7.** Donnie Brasco (1997) **8.** Django Unchained (2012) **9.** Shining, The (1980) **10.** Black Mirror (2011) **11.** Day of the Doctor, The (2013) **12.** Big Lebowski, The (1998) **13.** Gone Girl (2014) **14.** Raging Bull (1980) **15.** American Hustle (2013) **16.** Inglourious Basterds (2009) **17.** Princess Bride, The (1987) **18.** Hoop Dreams (1994) **19.** Clockwork Orange, A (1971) **20.** Wolf of Wall Street, The (2013)
His user vector: (Drama,6), (Crime,5), (Thriller,4), (Action,2), (Adventure,1), (Sci-Fi,1), (Mystery,1), (Fantasy,1), (Comedy,1) He also said that recommendations of content-based nearest neighbors were better. My friend told that he recently watched the Donnie Brasco and Inglourios Basterds and those films were in the recommendation list!	