**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*.

|  |
| --- |
|  |
|  |
| 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.

|  |
| --- |
|  |
| 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.

|  |
| --- |
|  |
|  |
| 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. 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** | 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** |
| 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 |
|  | | | | |
| However, I tried training another models with custom parameters which is deducted from the results above: | | | | | |
| **Rank** | | **Iteration** | **Lambda** | **MSE** | 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. |
| 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** |
| 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.

|  |
| --- |
|  |

1. 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.

|  |
| --- |
|  |

1. 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.

|  |
| --- |
|  |

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.

|  |
| --- |
|  |

1. 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.

|  |
| --- |
|  |

1. 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.

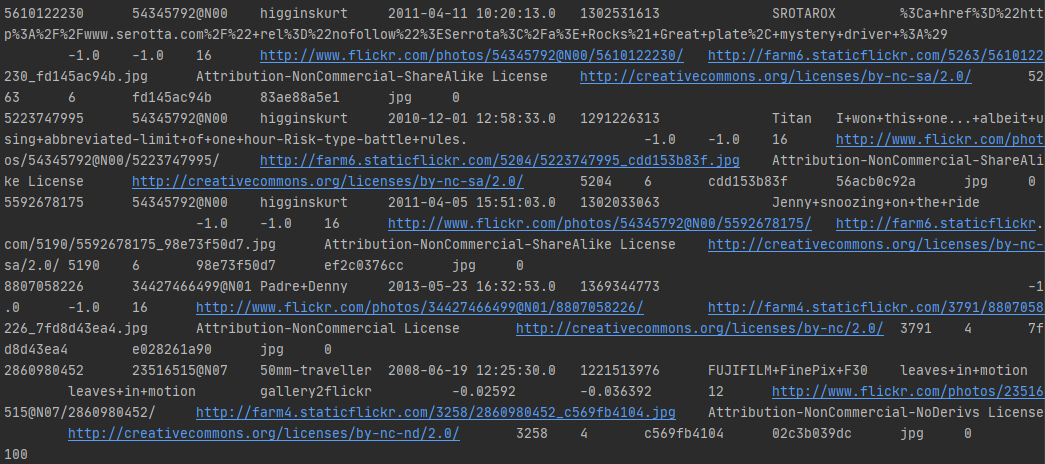
|  |
| --- |
| If user’s ratings and related trained model exist, ask the user whether he/she wants to rate again. |
|  |
| If user’s ratings or related trained model don’t exist, or the user wants to rate the movies again: |
| Otherwise: |
|  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 liked, and there is chance that I can like some of the rest. |

**Part 2: Processing data using RDDs**

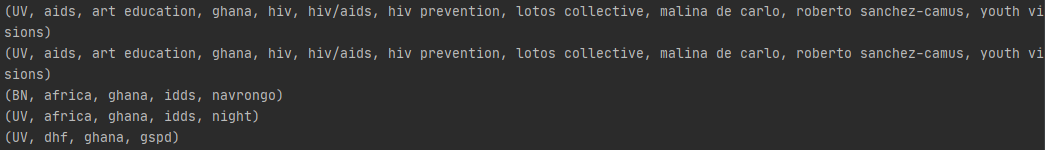
1. Display the 5 lines of the RDD (take(5)) and display the number of elements in the RDD (count()).



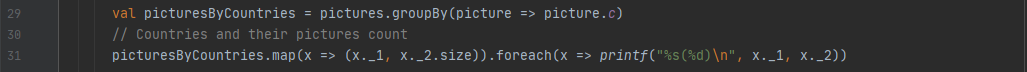


1. Transform the RDD[String] in RDD[Picture] using the Picture class. Only keep interesting pictures having a valid country and tags. To check your program, display 5 elements.





1. Now group these images by country (groupBy). Print the list of images corresponding to the first country. What is the type of this RDD?

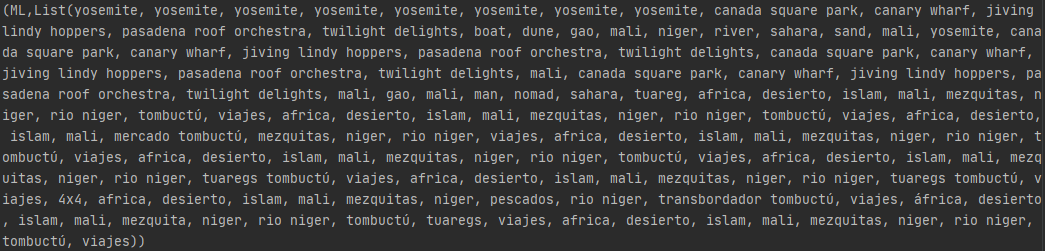




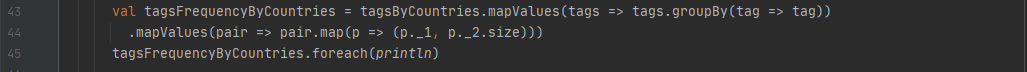
The type of picturesByCountries is RDD[(Country, Iterable[Picture])]. Additionally, I printed picture counts of countries.

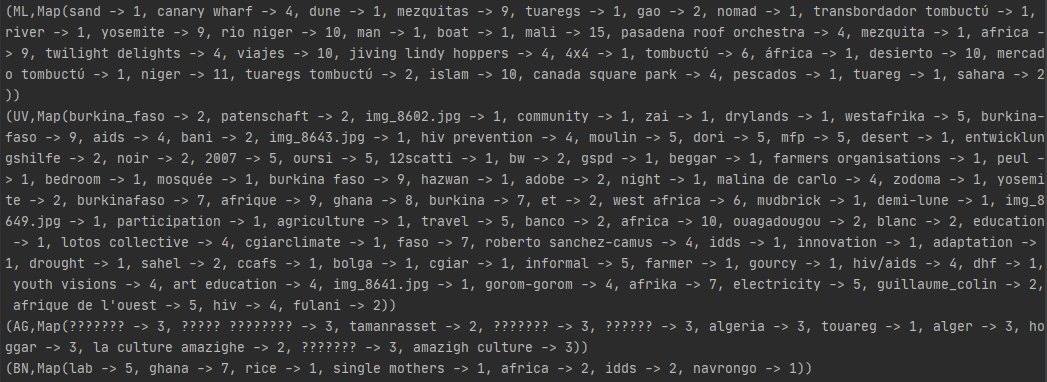
1. We wish to avoid repetitions in the list of tags, and would rather like to have each tag associated to its frequency. Hence, we want to build a RDD of type RDD[(Country, Map[String, Int])]. The groupBy(identity) function, equivalent to groupBy(x=>x) could be useful.





1. We wish to avoid repetitions in the list of tags, and would rather like to have each tag associated to its frequency. Hence, we want to build a RDD of type RDD[(Country, Map[String, Int])]. The groupBy(identity) function, equivalent to groupBy(x=>x) could be useful.





1. There are often several ways to obtain a result. The method we used to compute the frequency of tags in each country quickly reaches a state in which the size of the RDD is the number of countries. This can limit the parallelism of the execution as the number of countries is often quite small. Can you propose another way to reach the same result without reducing the size of the RDD until the very end?

Instead of Creating RDD[(Country, Map[String, Int])], we could create RDD[(Country, String)] where String represents the tag then we could group by each country, tag pair or simply RDD[(Country, String, Int)] where String still represents the tag and Int is the frequency of that tag in the country.