Systems for Data Science : Milestone 1

Aiday Marlen Kyzy
Sciper : 283505

March 2021

1 Introduction

In this project I have built a simple recommendation system as was described in the Milestone 1 document. The entire Milestone was coded on the command line, by connecting to the EPFL network through the VPN. It was a very interesting project, and I learned a lot about Resilient Distributed Datasets, as well as DataFrames (although not used here).

2 Answers to Questions

[Question 3.1.1]
The first task was to report the global average rating. The way this was done was by using the following command :

```
val avg = data.map(r => r.rating).reduce(_+_)/data.count
```

Here above I take the RDD called data, map each rating to it's actual rating value, add all these ratings and finally divide by the number of rows in the RDD. As such I find the following average : 3.52986. This is higher than 3 by approximately half a point. Hence there are more positive reviews than negative ones.

[Question 3.1.2 and 3.1.3]
In this problem I want to compute the average rating for each user. To do this, I use the following code:

```
val groupedUsers = data.groupBy(x => x.user).map{x => (x._1,
x._2.toList.map(r => r.rating).reduce(_+_)/x._2.toList.size)}
```

Here I group the data according to the users. I find that the result is of the form (r.user, CompactBuffer{(r.user, r.item, r.rating)}. I map each user to it's ID followed by an averaging operation over the ratings specific to the user. The same code is used for the items. Now I want to find the minimum and the maximum average among the users. To this end, I sort the RDD in ascending or descending order, respectively, and then take the head element each time.

```
val maxUsers = groupedUsers.sortBy(_._2, false).take(1).map(_._2)
```

The maximum among user averages is : 4.86956. The minimum among user averages is : 1.49195. The maximum among item averages is : 5.0. The minimum among item averages is : 1.0. All users do not rate close to the average, as can be seen by the following calculation I do, to find the ratio of users having an average rating that deviates by less than 0.5 from the global average.

```
val usersCloseToAverage = groupedUsers.filter(r => r._2 < avg + 0.5 && r._2 > avg - 0.5)
.count.toDouble/100000
```

The same calculation is done for the items. Surprisingly a small portion of the 100'000 users and items falls into this category. The ratio for the users is : 0.00704. Whereas the ratio for the items is : 0.00824.

[Question 3.1.4]
Below you can find the MAE values for the four different predictions methods seen above, when evaluated on the test data :

MAE global	0.96804
MAE per user	0.85006
MAE per item	0.82626
MAE for baseline	0.72718

What I notice from the table above is that when I use the baseline prediction, the corresponding MAE is the lowest out of the four methods available. Followed by the method that does prediction using the average per item, then per user, and then using the global average. The reason why the baseline prediction proposed is the best is because it takes into account the per user averages as well as a scaled version of the per item averages. Hence more data and known statistics are used in the baseline prediction, and the prediction is therefore more accurate. Conversely, for the same reason the global average as a prediction performs the worst, because it assigns the same general prediction to all users and ratings, without taking into account personal preferences, nor the popularity of the specific item.

To illustrate the results found, I include below an explanation of the code used to compute the MAE for the baseline prediction method. Starting from the groupedUsers and groupedItems RDDs defined previously, I do two joins in order to create a new RDD having as columns the following values (User, Item, Rating, Average rating for user, Average rating for item). This RDD is called averagesRDD. From there I perform a new map operation on averagesRDD to add a new column containing for each rating the following value $(r_{u,i} - \bar{r}_{u,\bullet})/scale(r_{u,i}, \bar{r}_{u,\bullet})$. From this latter RDD, I create a groupedNormalizedDeviations RDD where I group by the item number. Then I join the RDD groupedNormalizedDeviations with averagesRDD with the following command :

```
val baselineRDD = averagesRDD.keyBy{t => (t._2)}.join(groupedNormalizedDeviations.keyBy{t => t._1})
.map{ case(item, ((user, _, rating, average_rating_user, average_rating_item),
(_, global_deviation_item))) => (user, item, rating, average_rating_user +
global_deviation_item*scale((average_rating_user + global_deviation_item),average_rating_user))
```

From here the MAE for the baseline prediction is calculated as follows :

```
val maeBaseline = baselineRDD.map(r => scala.math.abs(r._3 - r._4)).reduce(_+_)/
baselineRDD.count.toDouble
```

Note that a scale function was defined in the code in order to simplify the calculations.

[Question 3.1.5]
In order to compute the time required for computing the predictions, the code defined previously is placed inside a for loop which runs ten times, and the time itself is computed with the System.nanoTime method, as follows :

```
val t1 = System.nanoTime
val globalPredTime = train.map(r => r.rating).reduce(_+_)/train.count
val t2 = System.nanoTime
timeGlobalPrediction += t2 - t1
```

Since the given time is in nano seconds, the time needs to be divided by 1000 in order to find the time in milliseconds. Note that in the above timeGlobalPrediction is an ArrayBuffer of Doubles, hence to find the minimum or the maximum it is sufficient to write later : timeGlobalPrediction.min (or .max). In order to calculate the standard deviation I defined the following function :

```
def calculateSD(arrayTime : ArrayBuffer[Double]) : Double = {
  val mean = arrayTime.sum/10
  var arrayOfTerms = new ArrayBuffer[Double]()
  for (i <- 0 to 9) {
     arrayOfTerms += (arrayTime(i)-mean)*(arrayTime(i)-mean)/(1000*1000)
  }
  return math.sqrt(arrayOfTerms.sum/10)
}
```

The division by 1000 is used to convert the time from nanoseconds to microseconds. This gives the following results:

-	Global	Per User	Per Item	Baseline
min	170973.059	21724.646	18201.053	3325139.825
max	229138.687	56803.239	34454.258	4234527.197
average	190529.602	33392.980	23380.980	3853401.238
standard deviation	18462.993	10880.492	5531.808	340811.538

I ran all the code using the VPN on the EPFL network and am sadly unaware of the technical specifications. Of the four predictions methods, the one that is most expensive to compute is unsurprisingly the baseline method. According to a calculation done in the code, computing the baseline prediction method takes 20.22 times more time than the global prediction method.

[**Question 4.1.1**]
I have added 33 rating in the personal dataset. I haven't seen all the movies I rated, but gave ratings depending on whether the movie title seemed interesting or not. These ratings that I added are seen below :

ID	My Rating	ID	My Rating	ID	My Rating	ID	My Rating
1	5.0	610	3.0	672	5.0	736	3.0
3	3.0	621	4.0	691	5.0	740	2.0
5	1.0	635	2.0	701	5.0	765	2.0
318	3.0	655	5.0	713	2.0	774	5.0
507	1.0	656	1.0	728	4.0	776	5.0

ID	My Rating	ID	My Rating	ID	My Rating
779	4.0	797	2.0	805	2.0
786	1.0	799	5.0	806	1.0
787	4.0	801	5.0	871	4.0
790	3.0	803	4.0	-	-
796	4.0	804	3.0	-	-

I perform the prediction using the baseline predictive model from the previous part, and then display the top recommendations. As was expected, the top 10 recommendations all score 5.0 and after this the predicted ratings start to slowly fall. I include here a table showing the first 5 recommended movies as well as recommended movies 11 through to 15 to show predictions lower than 5.0. Since 10 movies score 5.0, I include the movies 1 through to 5 in increasing order of title length.

Order	ID	Movie	Prediction
2	1293	Star Kid (1997)	5.0
1	1189	Prefontaine (1997)	5.0
4	814	Great Day in Harlem	5.0
3	1536	Aiqing wansui (1994)	5.0
5	1467	Saint of Fort Washington	5.0
...	...	...	...
11	1449	Pather Panchali (1955)	4.6385
12	1642	Some Mother's Son (1996)	4.5871
13	1398	Anna (1996)	4.5613
14	119	Maya Lin: A Strong Clear Vision (1994)	4.5431
15	1191	Letter From Death Row	4.4792

I have not seen these movies, but by name some of them seem interesting. The last line of code used to construct the above table is the following :

```
val myRecommendations = normalizedRDD.groupBy(x => x._2)
.map(x => (x._1, x._2.toList.map(r => r._6).reduce(_+_)/x._2.toList.size))
.map(x => (x._1, myAverage+x._2*scale(myAverage + x._2, myAverage))).sortBy(r => r._2, false)
```

Then I just run the following command to show the predictions :

```
myRecommendations.take(20).foreach(println)
```

[**Question 4.1.2**]
A simple way to take into account the popularity of the movie, is to multiply the baseline prediction by the number of times the specific movie is rated, divided by the maximum number of times a movie has been rated in the data-set. In this way a movie that is popular and has been rated a lot of times, will like be multiplied by a factor close to one and retain it's baseline, while an unpopular movie will be multiplied by a factor closer to zero, and the predicted rating will be smaller. Suppose that the maximum number of times a movie has been rated is n_{max}, then the new prediction will be : $\frac{n_i \cdot p_{u,i}}{n_{max}}$. Here n_i denotes the number of times movie i was rated. The top 5 recommended movies found are the following :

Order	ID	Movie	Prediction
1	50	Star Wars (1977)	4.2921
2	100	Fargo (1996)	3.5652
4	181	Return of the Jedi (1983)	3.3965
3	258	Contact (1997)	3.2609
5	174	Raiders of the Lost Ark (1981)	3.0101

This modified version seems indeed to work, as the top three movies are indeed well-known movies. The last line of code used to construct the above table is :

```
val myRecommendationsModified = normalizedRDDModified.groupBy(x => x._2)
.map{x => (x._1, x._2.toList.map(r => r._6)
.reduce(_+_)/x._2.toList.size, x._2.toList.map(r=> r._7).head)}
.map{x => (x._1, x._3.toDouble/maxRatings.toDouble*(myAverage+x._2*scale(myAverage + x._2, myAverage)))}
.sortBy(_._2, false)
```

In the above, I added the code `x._3.toDouble/maxRatings.toDouble` to account for the popularity of the respective movies.

3 Conclusion

This was a somewhat challenging project as I had only briefly studied Spark in the past, but it was also quite rewarding upon completion.