Big Data Capstone Project
Group 71
Siyan Liu
May 15, 2024
Github Repository: https://github.com/nyu-big-data/capstone-project-cap71

## Question 1:

To find 'movie watching twins', I tried several implementations.

The first version was to build an empty matrix of size (`num_users` * `num_movies`), and all values were initialized to zero; if a user has rated a movie, then the value at the corresponding movie index (note that it's the index of the movie, not the ID - these two values are different) will be set to 1. However, this approach did not work when running on dataproc using the large dataset, because such a matrix requires ~230 GiB of memory.

After reading some spark documentation, I learned that there is a built-in Sparse Vector type, and I converted the `ratings.csv` table to a list of sparse vectors. Then, my choice lies in whether to implement a minHash function on my own, building all the wheels from scratch, or use the existing minHashLSH module in PySpark.

I tried my own implementation, which uses 3 permutations to hash the vectors into buckets. Then, for vectors (individual users) in each bucket, calculate the pairwise similarity. This approach would work in theory, but unfortunately, in practice, many (on the scale of hundreds) of users hash into the same buckets - people who watch popular movies are likely to have a lot of overlap! I tried improving the hash function implementation and adding more hash functions, but again and again, dataproc kills the job, and I had to find something more efficient.

Finally, I decided to use `` `pyspark.ml.feature.MinHashLSH` ``, and that gave me very accurate results on movie twins (minimal Jaccard distance) at a fairly acceptable speed. This was unexpected, because the module uses `` `approxSimilarityJoin` `` to compute **all** pairwise distances, which will be on the scale of ~100 million O(1) operations. My own implementation attempts to hash the user records into **buckets** precisely because I wanted to avoid this costly operation. Anyways, I decided to filter out users who only watched a couple of movies by setting the threshold of vector length to 20. This way, we get similar users who watched at least 20 movies.

A side comment: I must say, chatGPT was extremely unhelpful for this project. I have never experienced such a level of GPT incompetence in my other software engineering projects. Perhaps because the logic is too custom and no one has written something similar before, at least not with PySpark. I had to dig into lots of documentation and even read PySpark's source code to understand the input/output a function expects. A huge learning curve!

## Question 2:

Please refer to the csv files attached or the PDF files at the end. Notably, randomly picked pairs usually only have less than 0.1 similarity (defined by Jaccard distance), and twin pairs chosen by minHashLSH all have 1.0 similarities (they are all identical! Their minHashLSH distances were all 0.0). The results were shockingly good so I verified some pairs by printing out their respective SparseVector. Pair (userId 85653, userId 95175) looks like this:

SparseVector(86537, {9: 1.0, 148: 1.0, 151: 1.0, 159: 1.0, 163: 1.0, 183: 1.0, 206: 1.0, 228: 1.0, 250: 1.0, 288: 1.0, 292: 1.0, 312: 1.0, 314: 1.0, 324: 1.0, 334: 1.0, 339: 1.0, 344: 1.0, 375: 1.0, 429: 1.0, 452: 1.0, 580: 1.0, 582: 1.0, 584: 1.0, 585: 1.0, 587: 1.0})

SparseVector(86537, {9: 1.0, 148: 1.0, 151: 1.0, 159: 1.0, 163: 1.0, 183: 1.0, 206: 1.0, 228: 1.0, 250: 1.0, 288: 1.0, 292: 1.0, 312: 1.0, 314: 1.0, 324: 1.0, 334: 1.0, 339: 1.0, 344: 1.0, 375: 1.0, 429: 1.0, 452: 1.0, 580: 1.0, 582: 1.0, 584: 1.0, 585: 1.0, 587: 1.0})

Feel free to try more pairs so it doesn't come across that I'm fabricating data!

The similarity between twin pairs is overwhelmingly higher than randomly chosen pairs.

## Question 3:

To implement a latent factor model, we cannot simply split test and training data randomly. That way, some users may not be included in the training set, which means that the user matrix will not have corresponding rows for them. We need all three sets to have data from each individual user. Therefore, I first grouped the records by user, and then split according by a fixed ratio (e.g. training=0.7, validation=0.2, test=0.1) within each user group. Some sources suggest that the records be split in a synchronous way - older records should be the training data, and the most recent ones test data - but I find that to be unreasonable. A user may get more interested in the sci-fi genre recently in a spontaneous way, which does not mean they stopped enjoying the romance genre they used to consume a lot. Hence, the records within a user were randomly assigned to training, validation, and test.

## Question 4:

As I have briefly discussed with Professor Wallisch after the final exam, I decided to calculate popularity for each movie by **summing** all ratings it has received. The reasoning is as follows: if we do not add a dampening factor, then the model will favor those 'critically received' indie movies that few people watched but have great average scores. If we do add a dampening factor to boost the ranking of truly 'popular' movies, the actual value of the factor is not known a priori - it's just an arbitrary value we impose on the data. So, to select truly popular movies that many people watched and liked, we simply aggregate over the ratings that each movie received. I think this is a better representation of the 'short head'.

## Question 5:

Since PySpark offers an ALS module, implementing the latent factor model was easy. I experimented with the hyperparameters on the smaller dataset. I understand that optimal hyperparameters for the smaller dataset do not always mean equally optimal for the large dataset, but in this case, the large dataset seems to have good performance with them. I set rank=10, reg_param=0.1, max_iter=10. The overall loss would only decrease slightly if I increase the rank and max iteration.

There are two ways we can evaluate the models: first is our good old RMSE model, where loss(u_i) = E[(R_ij - <u_i, v_j>)^2]. This loss function works for both the popularity based model and the latent factor model, as we already have the U, V matrix.

Another approach is to evaluate the bipartite ranking (specifically, mean average precision at k=100). This is where we would run into the **curse of sparsity**. Namely, a user has only interacted with an average of ~100 movies, and there are ~80000 movies in the database. The top 100 movies recommended to users are popularity-agnostic, meaning that even if only one person has watched it, as long as the movie gets a high <u_i, v_j> score, it will be recommended to the user. I tried the approach and was disappointed to find out that almost none of the movies recommended to a user is actually in the test dataset (i.e. held-out movies that users actually watched and rated). We can calculate a 'dummy' score for movies that are never watched based on the U, V matrices, but that will lead to data leakage!

In [54]:
```
from pyspark.sql.functions import collect_list

# Group by userId and collect movieIds into a list
ground_truth_train = train_data.groupBy("userId").agg(collect_list("movieId").alias("items"))
ground_truth_validation = validation_data.groupBy("userId").agg(collect_list("movieId").alias("items"))
ground_truth_test = test_data.groupBy("userId").agg(collect_list("movieId").alias("items"))

# Show the ground truth data
ground_truth_test.show()
```

```
[Stage 545:=========================================>          (155 + 40) / 200]
```

```
+------+--------------------+
|userId|               items|
+------+--------------------+
|   148|[780, 592, 527, 2...|
|   463|[58559, 1, 2858, ...|
|   471|     [480, 296, 593]|
|   496|          [590, 356]|
|   833|[7153, 5952, 2571...|
|  1088|[480, 32, 296, 71...|
|  1238|          [296, 592]|
|  1342|[47, 318, 4306, 3...|
|  1645|[1, 4226, 5952, 2...|
|  1829|     [457, 480, 588]|
|  1959|[1196, 1, 79132, ...|
|  2122|          [47, 2571]|
|  2142|[32, 457, 150, 59...|
|  2366|[1721, 1704, 780,...|
|  2659|        [2571, 1196]|
|  2866|     [527, 296, 593]|
|  3175|    [2858, 780, 588]|
|  3749|[260, 5952, 296, ...|
|  3794|       [79132, 2571]|
|  3918|[47, 590, 2959, 2...|
+------+--------------------+
only showing top 20 rows
```

As you can see from the ground_truth_test data frame above, many users only have two or three movies as their 'ground truth'. This indeed causes the recommended list to appear futile - we cannot find a single match in the ground truth set!

The Precision Mean Average results were not very encouraging:
Mean Average Precision (MAP) for ALS validation set: 0.15688806723860668
Mean Average Precision (MAP) for ALS test set: 0.29139083007708294

Mean Average Precision (MAP) for Popularity validation set: 0.14580795556161816
Mean Average Precision (MAP) for test set: 0.23233645525437063

Our ALS model has higher MAP than the popularity based model.

In addition, I have also tried the Reciprocal Rank (MRR) metrics. It does not work for sparse data either, because users are more likely to have watched popular movies than the custom ones we recommend - therefore, the popularity based model will have large MRR scores.

Taking the points discussed above into consideration, I evaluated the RMSE for both models, on validation and test datasets:

ALS RMSE on test data: 0.8173991724196419
ALS RMSE on validation data: 0.810838107029413

popular RMSE on test data is:0.8160830696978044
popular RMSE on validation data: 0.812323400320376

| user_id_1 | user_id_2 | jaccard_similarity |
|---|---|---|
| 131695 | 4592 | 0.0239234 |
| 310427 | 217769 | 0.005291 |
| 209383 | 184581 | 0.1170068 |
| 326980 | 82753 | 0.0227273 |
| 276509 | 282499 | 0.0384615 |
| 322988 | 206986 | 0 |
| 102018 | 170246 | 0.0555556 |
| 120095 | 196785 | 0.044164 |
| 64421 | 148543 | 0.1066667 |
| 130900 | 314043 | 0.0313178 |
| 12954 | 139971 | 0.2439024 |
| 87921 | 222187 | 0.0230415 |
| 190486 | 199245 | 0.0472441 |
| 186937 | 65916 | 0.0253165 |
| 4336 | 78539 | 0.0745342 |
| 40660 | 294846 | 0.0672451 |
| 234133 | 25766 | 0.0074013 |
| 116503 | 144632 | 0.0298507 |
| 329861 | 24904 | 0.047619 |
| 189776 | 117791 | 0.0108254 |
| 231971 | 184386 | 0.106599 |
| 220089 | 9914 | 0.047619 |
| 276154 | 30203 | 0.0423729 |
| 28406 | 81821 | 0.0452174 |
| 112111 | 211784 | 0.0555556 |
| 202992 | 230587 | 0.0482315 |
| 97473 | 175095 | 0.0360577 |
| 151837 | 40397 | 0 |
| 20794 | 289049 | 0.0128205 |
| 228920 | 303471 | 0.0652174 |
| 32519 | 210000 | 0.0763052 |
| 207619 | 242704 | 0.0308483 |
| 88444 | 232729 | 0.0343348 |
| 32858 | 203181 | 0.0146628 |
| 14747 | 28734 | 0 |
| 80465 | 262343 | 0.0325733 |
| 28911 | 78665 | 0.1192661 |
| 322731 | 86313 | 0 |
| 68008 | 163811 | 0.0941176 |
| 258164 | 319200 | 0.0752351 |
| 203006 | 153895 | 0.0421053 |
| 84194 | 50175 | 0.0532374 |
| 281480 | 126733 | 0.010989 |

| | | |
|---|---|---|
| 186875 | 289524 | 0.07277 |
| 127368 | 88675 | 0 |
| 285838 | 277581 | 0.05625 |
| 160713 | 203583 | 0.0448179 |
| 150980 | 277005 | 0.083877 |
| 295410 | 43785 | 0.0608696 |
| 322219 | 309298 | 0.028169 |
| 96821 | 87301 | 0.0144578 |
| 57394 | 186497 | 0.0595745 |
| 143758 | 84168 | 0.026738 |
| 278653 | 157927 | 0.2274882 |
| 253374 | 306905 | 0.0103627 |
| 49299 | 131726 | 0.1048593 |
| 251294 | 271642 | 0.0827586 |
| 69716 | 155845 | 0 |
| 206930 | 267241 | 0 |
| 111009 | 95933 | 0.0359712 |
| 143126 | 169341 | 0.0196078 |
| 57155 | 224536 | 0.0263158 |
| 59149 | 210009 | 0 |
| 94248 | 271398 | 0.0272727 |
| 313470 | 149868 | 0.2316076 |
| 171219 | 295918 | 0.0690789 |
| 314023 | 325455 | 0.013369 |
| 310024 | 322407 | 0.1268882 |
| 71905 | 54512 | 0.0533333 |
| 247709 | 5084 | 0.0986395 |
| 249580 | 202891 | 0.0954198 |
| 136501 | 63679 | 0.0731183 |
| 232194 | 251544 | 0.0285714 |
| 232248 | 55845 | 0.0114943 |
| 168986 | 9370 | 0.0153846 |
| 129692 | 160767 | 0.05 |
| 302756 | 310659 | 0.021978 |
| 5110 | 149208 | 0.009901 |
| 296416 | 22049 | 0.1387755 |
| 199427 | 263789 | 0 |
| 221236 | 170578 | 0.0117371 |
| 221114 | 321238 | 0.0227273 |
| 107024 | 197716 | 0.0347826 |
| 83696 | 287949 | 0.0322581 |
| 175405 | 55032 | 0.0432961 |
| 95628 | 115635 | 0.015138 |
| 44746 | 95325 | 0 |

| | | |
|---|---|---|
| 316962 | 103961 | 0.0434783 |
| 295403 | 47444 | 0.0740741 |
| 241577 | 263112 | 0.0105263 |
| 86039 | 246072 | 0 |
| 201975 | 112391 | 0.0804598 |
| 297813 | 329124 | 0.0197183 |
| 23024 | 174730 | 0.019305 |
| 235222 | 304126 | 0.0116279 |
| 63277 | 230897 | 0.035074 |
| 91566 | 157364 | 0.0215264 |
| 176165 | 128003 | 0.0484848 |
| 234933 | 52732 | 0.0318471 |
| 209665 | 24700 | 0 |

| user_id_1 | user_id_2 | jaccard_simi | minHashDistance |
|---|---|---|---|
| 85653 | 95175 | 1 | 0 |
| 305904 | 314085 | 1 | 0 |
| 51160 | 62784 | 1 | 0 |
| 57540 | 282312 | 1 | 0 |
| 77600 | 132822 | 1 | 0 |
| 64872 | 293880 | 1 | 0 |
| 82321 | 236540 | 1 | 0 |
| 158283 | 329248 | 1 | 0 |
| 82321 | 277862 | 1 | 0 |
| 19061 | 241234 | 1 | 0 |
| 97685 | 134981 | 1 | 0 |
| 22437 | 23527 | 1 | 0 |
| 234310 | 325068 | 1 | 0 |
| 3799 | 97685 | 1 | 0 |
| 12125 | 47942 | 1 | 0 |
| 50132 | 87155 | 1 | 0 |
| 183207 | 222150 | 1 | 0 |
| 122621 | 235844 | 1 | 0 |
| 77318 | 227884 | 1 | 0 |
| 83065 | 183123 | 1 | 0 |
| 93572 | 177183 | 1 | 0 |
| 3799 | 162798 | 1 | 0 |
| 14186 | 253377 | 1 | 0 |
| 51160 | 313610 | 1 | 0 |
| 93572 | 119474 | 1 | 0 |
| 22437 | 313463 | 1 | 0 |
| 3799 | 227884 | 1 | 0 |
| 238288 | 265718 | 1 | 0 |
| 59869 | 63309 | 1 | 0 |
| 147789 | 279965 | 1 | 0 |
| 77600 | 163485 | 1 | 0 |
| 123530 | 161017 | 1 | 0 |
| 14186 | 249611 | 1 | 0 |
| 156709 | 235844 | 1 | 0 |
| 263035 | 304933 | 1 | 0 |
| 47942 | 134981 | 1 | 0 |
| 14186 | 87155 | 1 | 0 |
| 120522 | 309198 | 1 | 0 |
| 248461 | 266708 | 1 | 0 |
| 18356 | 265259 | 1 | 0 |
| 77318 | 170112 | 1 | 0 |
| 49647 | 168141 | 1 | 0 |
| 14186 | 22437 | 1 | 0 |

| | | | |
|---|---|---|---|
| 183123 | 249077 | 1 | 0 |
| 87155 | 93572 | 1 | 0 |
| 71302 | 279965 | 1 | 0 |
| 169845 | 277862 | 1 | 0 |
| 88603 | 283959 | 1 | 0 |
| 85653 | 291625 | 1 | 0 |
| 14186 | 50132 | 1 | 0 |
| 12125 | 88344 | 1 | 0 |
| 264494 | 266605 | 1 | 0 |
| 175379 | 248242 | 1 | 0 |
| 163485 | 243711 | 1 | 0 |
| 92534 | 313668 | 1 | 0 |
| 74957 | 91881 | 1 | 0 |
| 12125 | 77318 | 1 | 0 |
| 230806 | 277862 | 1 | 0 |
| 169860 | 277862 | 1 | 0 |
| 134981 | 248242 | 1 | 0 |
| 77318 | 88344 | 1 | 0 |
| 47942 | 65538 | 1 | 0 |
| 12125 | 170112 | 1 | 0 |
| 56180 | 264494 | 1 | 0 |
| 82321 | 141201 | 1 | 0 |
| 119474 | 253377 | 1 | 0 |
| 313668 | 325068 | 1 | 0 |
| 82915 | 99030 | 1 | 0 |
| 87155 | 253377 | 1 | 0 |
| 47942 | 97685 | 1 | 0 |
| 33604 | 234131 | 1 | 0 |
| 7038 | 169493 | 1 | 0 |
| 107031 | 198462 | 1 | 0 |
| 38643 | 172316 | 1 | 0 |
| 23527 | 313463 | 1 | 0 |
| 65538 | 162798 | 1 | 0 |
| 101017 | 305828 | 1 | 0 |
| 88344 | 162798 | 1 | 0 |
| 88344 | 134981 | 1 | 0 |
| 162798 | 170112 | 1 | 0 |
| 113414 | 311227 | 1 | 0 |
| 40479 | 107031 | 1 | 0 |
| 2626 | 227267 | 1 | 0 |
| 49138 | 81265 | 1 | 0 |
| 115422 | 279965 | 1 | 0 |
| 86856 | 273017 | 1 | 0 |
| 184938 | 246671 | 1 | 0 |

| | | | |
|---|---|---|---|
| 187276 | 319748 | 1 | 0 |
| 184938 | 324016 | 1 | 0 |
| 49651 | 85653 | 1 | 0 |
| 50132 | 253377 | 1 | 0 |
| 7038 | 325068 | 1 | 0 |
| 7038 | 313668 | 1 | 0 |
| 70108 | 96548 | 1 | 0 |
| 87155 | 177183 | 1 | 0 |
| 50132 | 93572 | 1 | 0 |
| 82321 | 230806 | 1 | 0 |
| 92534 | 325068 | 1 | 0 |
| 169845 | 308490 | 1 | 0 |
| 50132 | 177183 | 1 | 0 |