SHORYA SETHIA

22B2725

Engineering Physics

Report on SVD based Movie Recommender System

Introduction

The advent of digital streaming platforms has led to an explosion of available content, necessitating the development of sophisticated recommendation systems to assist users' engagement and optimization navigating the vast array of movies. These systems employ advanced algorithms rooted in collaborative filtering, content-based filtering, demographic filtering, and hybrid models to deliver personalized recommendations, thereby enhancing user satisfaction and engagement.

Context

Leveraging data-driven approaches and machine learning techniques, these models strive to optimize the movie selection process by predicting user preferences with great accuracy and relevance.

Types of Recommendations Methods

A comprehensive understanding of movie recommendation systems necessitates an exploration of various methodologies, including:

1. **Collaborative Filtering**: This technique harnesses user-user and user-movie interaction data to construct similarity matrices and infer latent user preferences. Through methods such as matrix factorization and neighborhood-based approaches, collaborative filtering predicts user ratings for unrated movies.
2. **Content-Based Filtering:** Content-based recommendation systems analyze movie attributes such as

genre, cast, plot summaries, and textual features to generate personalized recommendations. Utilizing techniques such as vector space models and natural language processing, content-based filtering identifies similarities between movies based on their intrinsic characteristics.

3. **Hybrid Recommendation Systems:** Hybrid models integrate collaborative and content-based filtering techniques to exploit their respective strengths and mitigate their inherent limitations. By combining collaborative user feedback with content-based movie attributes, hybrid systems offer enhanced recommendation accuracy and diversity.

Sources and References

1. Dataset : https://www.kaggle.com/datasets/netflix-inc/netflix-prize-data
2. Kaggle : https://www.kaggle.com/code/morrisb/how-to-recommend-anything-deep-recommender
3. Blogs :
   https://netflixtechblog.com/netflix-recommendations-beyond-the-5-stars-part-1-55838468f429
   https://medium.com/@sgchandela960/the-bert-llm-redefining-movie-similarity-for-the-digital-age-c1740f4b3c43
4. Reddit :
   https://www.reddit.com/r/MLQuestions/comments/1aw8f0v/movie_recommendation_system/
5. Research Papers :
   https://courses.ischool.berkeley.edu/i290-dm/s11/SECURE/a1-koren.pdf
   https://datajobs.com/data-science-repo/Recommender-Systems-%5BNetflix%5D.pdf
6. GitHub Repos :
   https://github.com/NikosMav/DataAnalysis-Netflix
   https://github.com/abishek-as/Netflix-Movie-Recommendation
   https://apple.github.io/turicreate/docs/userguide/recommender/
7. Surprise lib :
   http://surpriselib.com/
   https://github.com/NicolasHug/Surprise#installation
   http://surprise.readthedocs.io/en/stable/getting_started.html
8. YouTube :
   https://youtu.be/Eeg1DEeWUjA?si=7UT3F63bukJrBwtH
   https://youtu.be/XfAe-HLysOM?si=SmvdKttu-moahLmo
   https://youtu.be/ZspR5PZemcs?si=jOUMeZV6B01boz0j
9. ChatGPT

Dataset Overview

Data files :

1. combined_data_1.txt
2. combined_data_2.txt
3. combined_data_3.txt
4. combined_data_4.txt
5. movie_titles.csv

Data Overview :

The first line of each file "combined_data_{i}.txt" contains the movie id followed by a colon. Each subsequent line in the file corresponds to a rating from a customer and its date in the format:
CustomerID,Rating,Date

1. MovieIDs range from 1 to 17770 sequentially.
2. CustomerIDs range from 1 to 2649429, with gaps. There are 480189 users.
3. Ratings are on a five star (integral) scale from 1 to 5.
4. Dates have the format YYYY-MM-DD.

Implementations

Part 1

- Predicting Top 10 movies for a given movie
- Cross-checking the above list via Google's Gemini

Part 2

- Validating matrix-factorization models are superior to classic nearest-neighbor techniques for producing product recommendations
- Minimizing the difference predicted and actual ratings via two performance metric : RMSE and MAPE

Part 3

- Using SVD matrix-factorization model for predicting Top 10 unwatched movies for a given user and a given watched movie
- Comparing my SVD model via current SOTA algorithm

---

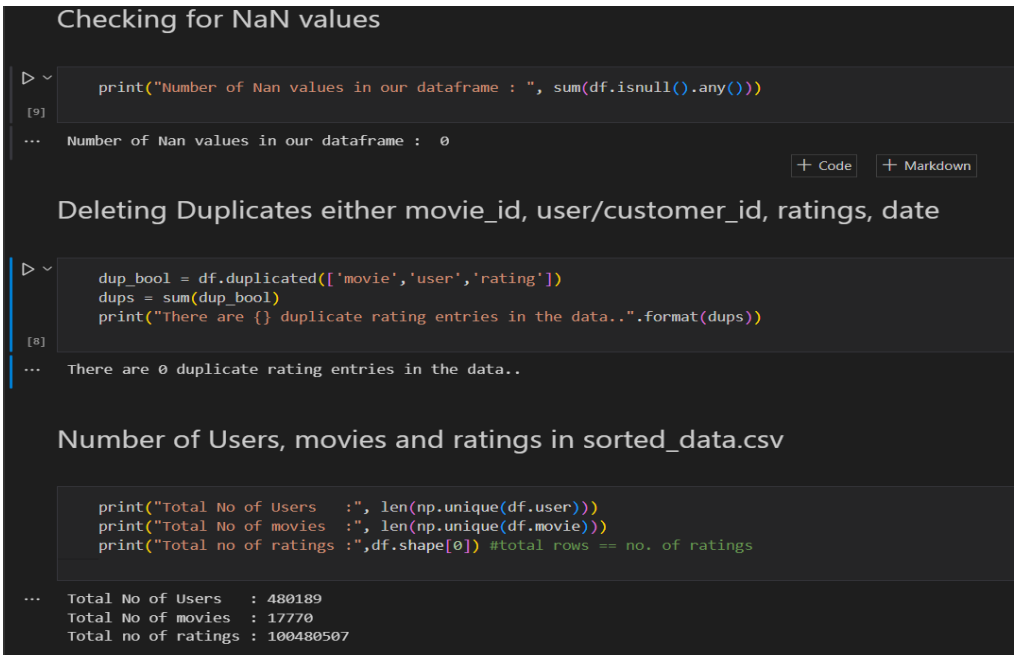
movie-recommender.ipynb Overview

(Based on **Collaborative Filtering**)

- Concatenated all four combined_data_1.txt,… , combined_data_2.txt into a single sorted_data.csv (sorted date wise) and made separate movie_id, user_id, ratings and date columns

	movie	user	rating	date
0	10341	510180	4	1999-11-11
1	1798	510180	5	1999-11-11
2	10774	510180	3	1999-11-11
3	8651	510180	2	1999-11-11
4	14660	510180	2	1999-11-11

- Then checked for any null or duplicate values present and calculated total user, ratings and movies



- Split data into train and test data, 80 : 20 respectively

```
Numbers for train.csv
Total No of Users   : 405041
Total No of movies  : 17424
Total no of ratings : 80384405

Numbers for test.csv
Total No of Users   : 349312
Total No of movies  : 17757
Total no of ratings : 20096102
```

- Performed Exploratory Data Analysis on training data
- To reduce memory usages, build test and train sparse matrices and calculated their sparsity
- Then calculated average global rating, rating per user and rating per movie for train data

```
train_averages = dict()

train_global_average = train_sparse_matrix.sum()/train_sparse_matrix.count_nonzero()
train_averages['global'] = train_global_average
print(f"Global Average of Ratings in training data is {train_averages}")
```

```
Global Average of Ratings in training data is {'global': 3.582890686321557}
```

```
train_averages['user'] = get_average_ratings(train_sparse_matrix, of_users=True)
# user = random.randint(1,train_sparse_matrix.shape [0])
# print(user)

# Generate a random user ID within the valid range
valid_users = list(train_averages['user'].keys())  # Get the list of valid user IDs
user = random.choice(valid_users)
print(f'Average rating of user {user} :',train_averages['user'][user])
```

```
Average rating of user 573242 : 4.138339920948616
```

```
train_averages['movie'] =  get_average_ratings(train_sparse_matrix, of_users=False)

valid_movies = list(train_averages['movie'].keys())
movie = random.choice(valid_movies)
print(f'Average rating of movie {movie} :',train_averages['movie'][movie])
```

```
Average rating of movie 12910 : 2.7708333333333335
```

- Computed user-user similarity matrix using cosine similarity with 17k dimesnions, it was taking a lot of compute time, power and kernel stopped, so I started

computing similarity for top 100 users for each user, but this also failed, then again started it but to find top 100 similar users for 200 users, and to avoid huge memory usages, converted similarity matrix into sparse matrix.

```
Computing strted for top 200 similarities for each user...
Computing done for 20 users [  time elapsed : 0:01:04.300235  ]
Computing done for 40 users [  time elapsed : 0:01:57.593116  ]
Computing done for 60 users [  time elapsed : 0:02:48.430230  ]
Computing done for 80 users [  time elapsed : 0:03:38.995090  ]
Computing done for 100 users [  time elapsed : 0:04:30.595433  ]
Computing done for 120 users [  time elapsed : 0:05:25.550533  ]
Computing done for 140 users [  time elapsed : 0:06:17.773366  ]
Computing done for 160 users [  time elapsed : 0:07:08.075599  ]
Computing done for 180 users [  time elapsed : 0:08:07.425267  ]
Computing done for 200 users [  time elapsed : 0:09:08.188622  ]
Creating Sparse matrix from the computed similarities
Time taken for user-user cf with 17k dimensions per user : 0:09:13.713449
```

- Calculating user-user Similarity_Matrix (user-user collaborative filtering) is not an easy task
- For top 200 users it took **0:09:13.713449** time, and as users count increases, complexity increases as one could find more and more similarities.

- On avg per time consumed for searching similarity for one user = (9*60 + 13.71)/200 = **2.76 seconds**
- training data have 405041 users, so approximately it would take **405041*2.76 = 1117913 seconds = 12.93 days**
- It will take almost **13** days to just find similarities !

- Hence, i would try to find user-user similarity via reduced dimensions

And did a basic calculation, how much time it would take to run for all users, it came out to be approx 13 days on my system.

- Retried, but this time with reduced dimensions using Truncated SVD and tried it with 100 latent factors, and checked gain in explained variance for all 100 components, and found it will take 500 components to be 60% accurate

```
expl_var = np.cumsum(netflix_svd.explained_variance_ratio_)
expl_var
```

```
array([0.23362135, 0.26270872, 0.28323418, 0.29936103, 0.31129667,
       0.32272449, 0.33168545, 0.33816688, 0.34421001, 0.34939129,
       0.35412811, 0.35790579, 0.36145969, 0.36481079, 0.36796535,
       0.3709693 , 0.37381048, 0.37654066, 0.37892266, 0.38128434,
       0.38355732, 0.38573246, 0.38787214, 0.38996681, 0.39201513,
       0.39393495, 0.39577018, 0.39753914, 0.39924786, 0.40091947,
       0.40251418, 0.40408101, 0.40563205, 0.40715363, 0.40864418,
       0.41009275, 0.41151715, 0.41291575, 0.41428276, 0.4156209 ,
       0.41692581, 0.41818944, 0.41941626, 0.4206308 , 0.42183602,
       0.42301205, 0.42417439, 0.42530795, 0.42642577, 0.42753769,
       0.42862981, 0.4297012 , 0.43074982, 0.43178112, 0.43281107,
       0.43382522, 0.434825  , 0.43580279, 0.43677693, 0.43773492,
       0.43868205, 0.43962079, 0.44054526, 0.44145286, 0.44236078,
       0.44325366, 0.44413545, 0.4450134 , 0.44587698, 0.44672994,
       0.44757815, 0.44841497, 0.44924205, 0.45006438, 0.45087095,
       0.45167476, 0.45246745, 0.45326091, 0.45404195, 0.45482063,
       0.45558687, 0.45634511, 0.45709084, 0.45783477, 0.45856918,
       0.45929436, 0.4600164 , 0.46073887, 0.46145237, 0.46215905,
       0.46286149, 0.46355766, 0.46424323, 0.46492155, 0.46559893,
       0.4662713 , 0.46693426, 0.46759437, 0.46824951, 0.4688893 ])
```

- It basically is the gain of variance explained, if we add one additional latent factor to it via np.cumsum()
- By adding one by one latent factore to it, **gain in explained variance** is decreasing.
- To take it to greter than 0.60, we have to take almost 400-500+ latent factors. It's totally useless (more compute power and memory loss)

- Then chose to truncate the initial user-user similarity sparse matrix, no doubt compute time decreases significantly, but this was also taking around 4-5 days to compute, which is still too much

```
Computing strted for top 50 similarities for each user...
Computing done for 10 users [  time elapsed : 0:00:07.767252  ]
Computing done for 20 users [  time elapsed : 0:00:15.033117  ]
Computing done for 30 users [  time elapsed : 0:00:22.483663  ]
Computing done for 40 users [  time elapsed : 0:00:29.950224  ]
Computing done for 50 users [  time elapsed : 0:00:37.384297  ]
Creating Sparse matrix from the computed similarities
time: 0:00:40.147739
```

+ Code + Markdown

- Time taken per user = 0:00:44.379248 / 50 = **0.88 seconds**
- We have total users = 405041, which means u-u similarity presize computation would take 405041*0.88 = 4.125 days
- No doubt, svd has decreased the time of computation, but 4+ days time is also a very long time. It would take lot of memory and computation power, which is very very hard to execute.

- But later discovered some of this time might be because of double counting, I mean code was again calculating similarity between user2 and user 1 even after calculating similarity between user 1 and user 2

Alternative/Modification to traditional SVD

But one drawback i noticed in my above method is, it re-calculate the similarities of a user with another user in some iterations. To minimize/optimize it:

- I will maintain a binary Vector for users, which tells us whether program has already computed top(say, 100) similarities for a user or not.
- **If not** : Compute top (say, 100) most similar users for this user, and add this to our datastructure, so that we can just access it(similar users) without recomputing it again. The way which i did above
- But **If It is already Computed** : Just get it directly from our datastructure. In due time,i might have to recompute similarities, if it is computed a long time ago. Because user preferences changes over time.
- So, program could maintain some kind of **Timer**, which when expires, we have to update it (recompute it).

- In some sense I need to calculate upper triangular matrix for similarity and lower one would be deducible then, but max to max, it will decrease time form 4-5 days to 2-3 days, which is still too much

- So, I started with movie-movie similarity matrix

```
start = datetime.now()
if not os.path.isfile('m_m_sim_sparse.npz'):
    start = datetime.now()
    m_m_sim_sparse = cosine_similarity(X=train_sparse_matrix.T, dense_output=False)
    # store this sparse matrix in disk before using it. For future purposes.
    sparse.save_npz("m_m_sim_sparse.npz", m_m_sim_sparse)
    print("Done.")
else:
    print("m_m_sim_saprse.npz is there already, Loading it...")
    m_m_sim_sparse = sparse.load_npz("m_m_sim_sparse.npz")
    print("Done.")

# print("m_m_sim_sparse.npz is a ",m_m_sim_sparse.shape," dimensional matrix")

print(datetime.now() - start)
```

[23]

```
m_m_sim_saprse.npz is there already, Loading it...
Done.
0:00:14.750535
```

```
m_m_sim_sparse.shape
```

[24]

```
(17771, 17771)
```

- Even though we have similarity measure of each movie, with all other movies. But generally one don't care much about least similar movies.
- Most of the times platforms recommends only top_xx similar items (here, item = movie). It may be top 10 or 100.
- So, its better to take only top similar movie ratings and store them in a saperate dictionary.

- movie_id given to movies would be not helpgul in cross validating the recommended list, so I used Netflix's movie_titles.csv to deduce movie names via movie ids.

```
movie_titles = pd.read_csv("Data/movie_titles.csv", sep=',', header = None,
                           names=['movie_id', 'year_of_release', 'title'],
                           usecols=[0, 1, 2], verbose=True,
                       index_col = 'movie_id', encoding = "ISO-8859-1")
#encoding necessary as movie_titles.csv has characters outside ASCII range
```

[27]

```
Tokenization took: 0.81 ms
Type conversion took: 9.01 ms
Parser memory cleanup took: 0.00 ms
C:\Users\shory\AppData\Local\Temp\ipykernel_15432\3737456495.py:1: FutureWarning: The 'verbose' keyword in pd.read_csv is deprecated and will be removed in a future version.
  movie_titles = pd.read_csv("Data/movie_titles.csv", sep=',', header = None,
```

```
movie_titles.head()
```

[28]

movie_id	year_of_release	title
1	2003.0	Dinosaur Planet
2	2004.0	Isle of Man TT 2004 Review
3	1997.0	Character
4	1994.0	Paula Abdul's Get Up & Dance
5	2004.0	The Rise and Fall of ECW

- Movie Recommendations simulation

```
mv_id = 1061

print(f"Movie id {mv_id} corresponds to ",movie_titles.loc[mv_id].values[1])

print("It has {} Ratings from users.".format(train_sparse_matrix[:,mv_id].getnnz()))

print(f"Movide id = {mv_id}" + " have {} movies which are similar to this and but only top 100 most similar ones are of interest.".format(m_m_sim_sparse[:,mv_id].getnnz()))
```

[29] Python

```
Movie id 1061 corresponds to  Spider-Man vs. Doc Ock
It has 744 Ratings from users.
Movide id = 1061 have 17320 movies which are similar to this and but only top 100 most similar ones are of interest.
```


```
# Top 10 similar movies for choosed moive_id
movie_titles.loc[similar_indices[:10]]
```

[31]

movie_id	year_of_release	title
12524	2004.0	Spider-Man: The Return of the Green Goblin
2279	2002.0	Spider-Man: The Ultimate Villain Showdown
13274	1996.0	Daredevil vs. Spiderman
1231	1999.0	Batman Beyond: Tech Wars / Disappearing Inque
14184	1967.0	Spider-Man: The '67 Classic Collection
2912	2003.0	Spider-Man: The New Animated Series: Season 1
14017	1999.0	Batman Beyond: School Dayz / Spellbound
11088	1992.0	Adventures of Batman & Robin: The Joker/Fire &...
4342	1992.0	Batman: The Animated Series: Out of the Shadows
7903	1992.0	Adventures of Batman & Robin: Poison Ivy/The P...

- But this is would require one to run code blocks again and again, so made a streamlit interface and incorporated Google's Gemini response to validate and cross check similarities, via GOOGLE_API_KEY
  Video Demo
- Now, to validate the fact that matrix-factorization models are superior to classic nearest-neighbor techniques for producing product (here movie) recommendations, I have basically used the following machine learning pathway as mentioned in research papers I followed.
  To reduce compute cost and time, I have created a small sample from the original data,
  For sample train data, (user,movies)=(8000,800)
  For sample test data, (user,movies)=(4000,400)

Train Data

```
It is not present in pwd...
Original Matrix : (users, movies) -- (405041 17424)
Original Matrix : Ratings -- 80384405

Sampled Matrix : (users, movies) -- (8000 800)
Sampled Matrix : Ratings -- 78751
Saving it into pwd for furthur operations...
Done.
0:01:27.232241
```

Test Data

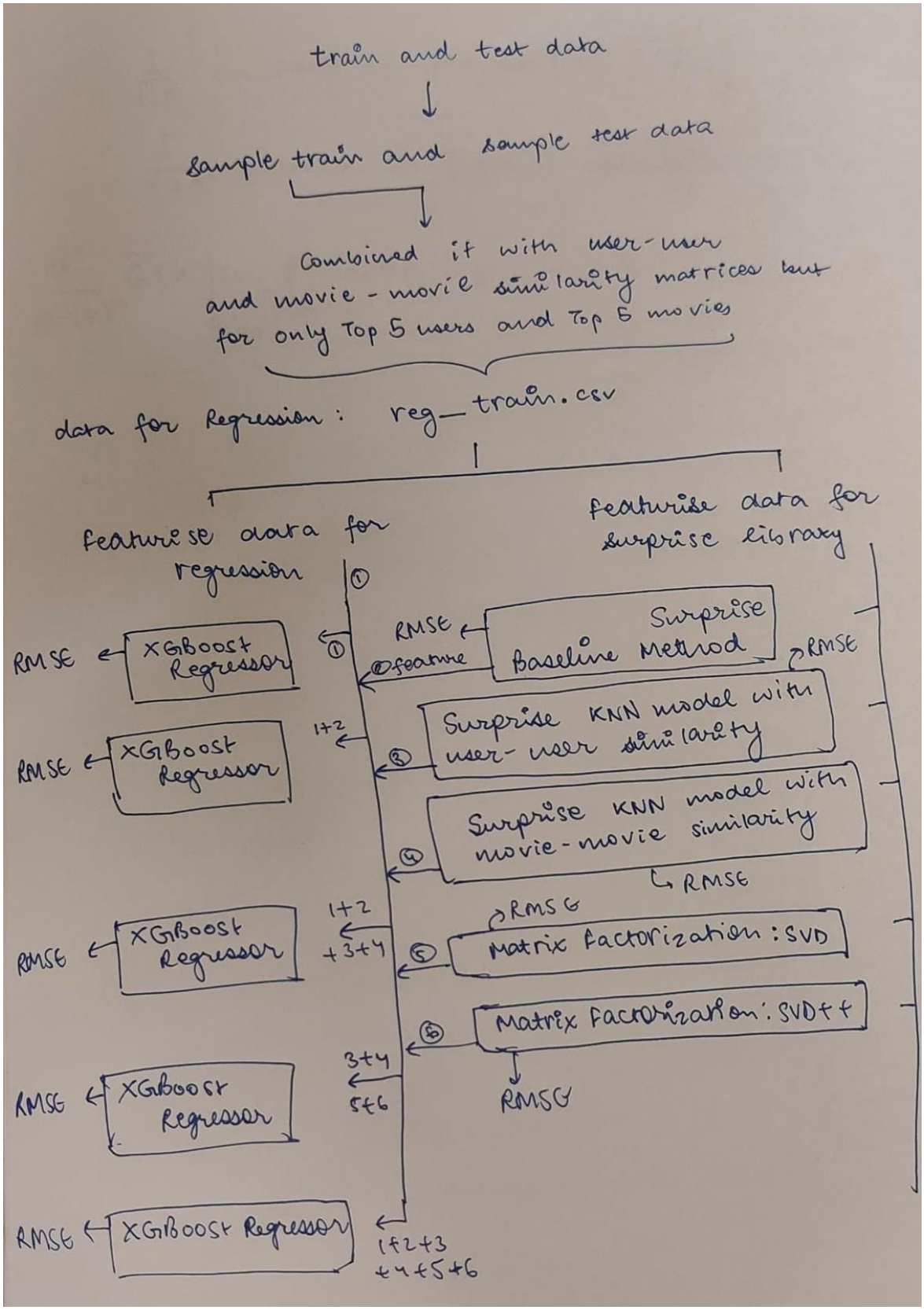
```
It is not present in pwd...
Original Matrix : (users, movies) -- (349312 17757)
Original Matrix : Ratings -- 20096102

Sampled Matrix : (users, movies) -- (4000 400)
Sampled Matrix : Ratings -- 4530
Saving it into pwd for furthur operations...
Done.
0:00:17.270051
```

- Then formed an reg_train.csv from sample training data, it took 3 hours 29minutes to form with 78,751 user-movie tuples

	user	movie	GAvg	sur1	sur2	sur3	sur4	sur5	smr1	smr2	smr3	smr4	smr5	UAvg	MAvg	rating
0	692	14621	3.61198	2.0	4.0	5.0	5.0	5.0	4.0	4.0	4.0	4.0	4.0	4.000000	4.329095	4
1	1179	2239	3.61198	5.0	3.0	3.0	2.0	2.0	3.0	5.0	4.0	3.0	4.0	3.714286	2.909091	5
2	1179	4352	3.61198	4.0	3.0	2.0	3.0	3.0	4.0	4.0	3.0	4.0	5.0	3.714286	3.140845	3
3	1179	6464	3.61198	3.0	5.0	4.0	2.0	4.0	4.0	4.0	2.0	5.0	3.0	3.714286	3.400396	4
4	1179	6510	3.61198	4.0	4.0	3.0	4.0	3.0	5.0	4.0	4.0	3.0	4.0	3.714286	3.936614	4

- GAvg : Avg Global Rating
- UAvg : User's avg movie rating
- MAvg : Movie's avg rating
- rating : rating given by user to movie
- sur1,sur2,sur3,sur4,sur5 : top 5 similar users who rated that movie
- sm1r,smr2,smr3,smr4,smr5 : top 5 movies similar to that movie


train and test data
sample train and sample test data
Combined it with user-user and movie-movie similarity matrices but for only Top 5 users and Top 5 movies
data for Regression : reg_train.csv
featurise data for regression
featurise data for surprise library
XGBoost Regressor
RMSE
XGBoost Regressor
RMSE
XGBoost Regressor
RMSE
XGBoost Regressor
RMSE
XGBoost Regressor
RMSE
1+2
1+2+3+4
3+4
5+6
1+2+3+4+5+6
Surprise Baseline Method
RMSE
+feature
Surprise KNN model with user-user similarity
RMSE
Surprise KNN model with movie-movie similarity
RMSE
Matrix Factorization : SVD
RMSE
Matrix Factorization : SVD++
RMSE

- Comparison: (technique : RMSE)

```
svd               1.0821912114098393
knn_bsl_m         1.0821949464872673
bsl_algo          1.0822513101982842
knn_bsl_u         1.0822531134111517
svdpp             1.0822543401393163
first_algo        1.1075854286551927
xgb_knn_bsl         1.11901091975606
xgb_bsl           1.1234619836629212
xgb_final         1.1234619836629212
xgb_all_models    1.1367862682953647
Name: rmse, dtype: object
```

This also shows matrix-factorization: SVD is better.

- To get top 10 recommendations of movies for a given user who has watched a given movie

Following were movie recommendations for,

user_id = 2281810

movie_id = 14410

```
Movie ID: 33 | Predicted Rating: 4.452740626759515
Movie ID: 476 | Predicted Rating: 4.269301845585642
Movie ID: 135 | Predicted Rating: 4.202249728257774
Movie ID: 180 | Predicted Rating: 4.201643573376772
Movie ID: 485 | Predicted Rating: 4.141063943869716
Movie ID: 57 | Predicted Rating: 4.1073335467807475
Movie ID: 162 | Predicted Rating: 4.016947654954951
Movie ID: 648 | Predicted Rating: 3.946583086330561
Movie ID: 587 | Predicted Rating: 3.896274620410515
Movie ID: 186 | Predicted Rating: 3.8573703512727664
```

Top 10 movies recommended by this method and top 10 movies recommended earlier will be different, because:

- Here, I used small sample / number of user, movie tuples
- And above movie-movie predictor was based on m_m_sim_sparse.npz which was made by considering every user, movie tuples

Also made a streamlit interface for this. It takes targeted user_id and movie_id watched by it as input and displays list of top 10 movie recommendations along with predicted ratings.

Video Demo

Comparison with SOTA

- As per this research paper, the Netflix system achieves RMSE = 0.9514 on the same dataset
- While the grand prize's required accuracy is RMSE = 0.8563, and it was won by BigChaos solution with RMSE=0.8567
- I achieved minimum RMSE = 1.0821912114098393
- Due to high computational power and time, I have completed this case study on (8000,800) training dataset and (4000,400) testing dataset.
- Small decrease in RMSE score is observed, but this can be drastically improved by using the whole dataset for modelling. (Not feasible on my system)