Configuration Manual

MSc Research Project
Masters in Data Analytics

Shubham Sunil Rajeshirke
Student ID: x21231036

School of Computing
National College of Ireland

Supervisor: Arjun Chikkankod

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Shubham Sunil Rajeshirke
Student ID:	X21231036
Programme:	Masters in Data Analytics **Year:** 2022-2023
Module:	MSc Research Project
Lecturer:	Arjun Chikkankod
Submission Due Date:	14/08/2023
Project Title:	Multi-objective Recommender System for E-commerce using Singular Value Decomposition (SVD) Matrix Factorization Technique.
Word Count:	1521 **Page Count:** 15

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Shubham Sunil rajeshirke

Date: 14/08/2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Shubham Sunil Rajeshirke
Student ID: x21231036

1 Introduction

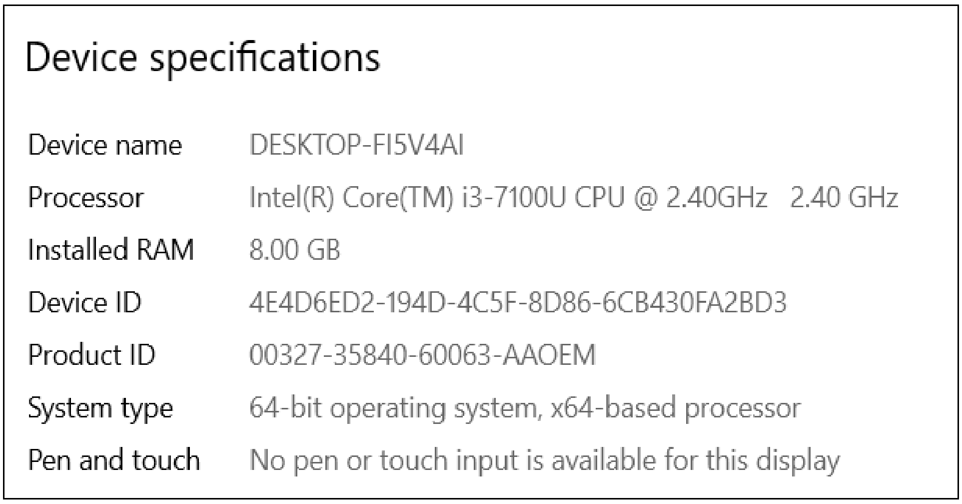
This configuration manual explains the steps to launch the scripts written for this research. Following this document will make sure that the code is running correctly without any errors and with efficiency. The hardware and software requirements for the execution of this research are also mentioned in this document. Overall, this document will be valuable for understanding the prerequisites of the Research and setting up the execution environment for implementing this research.

2 System Configurations

2.1 Hardware Requirements

The Hardware requirements for implementing this research are as follows:

- Windows Edition: Windows 10 Home Single Language
- Processor: Intel(R) Core (TM) i3-7100U CPU @ 2.40GHz   2.40 GHz
- Installed RAM: 8.00 GB
- System Type: 64-bit operating system, x64-based processor
- Pen and Touch: No pen or touch input is available for this display.



Device specifications

Device name	DESKTOP-FI5V4AI
Processor	Intel(R) Core(TM) i3-7100U CPU @ 2.40GHz 2.40 GHz
Installed RAM	8.00 GB
Device ID	4E4D6ED2-194D-4C5F-8D86-6CB430FA2BD3
Product ID	00327-35840-60063-AAOEM
System type	64-bit operating system, x64-based processor
Pen and touch	No pen or touch input is available for this display

Fig 1 : Device Specifications

2.2 Software Requirements

The software requirements used for implementation of this research are as follows:

- Programming language: Python (version – 3.9.13)
- IDE: Jupyter Notebook

3 Project Implementation

This section describes the steps used for implementing this research.

3.1 Programming Environment Setup

The Jupyter Notebook is launched from Anaconda Navigator to start the execution environment used for the implementation purpose.

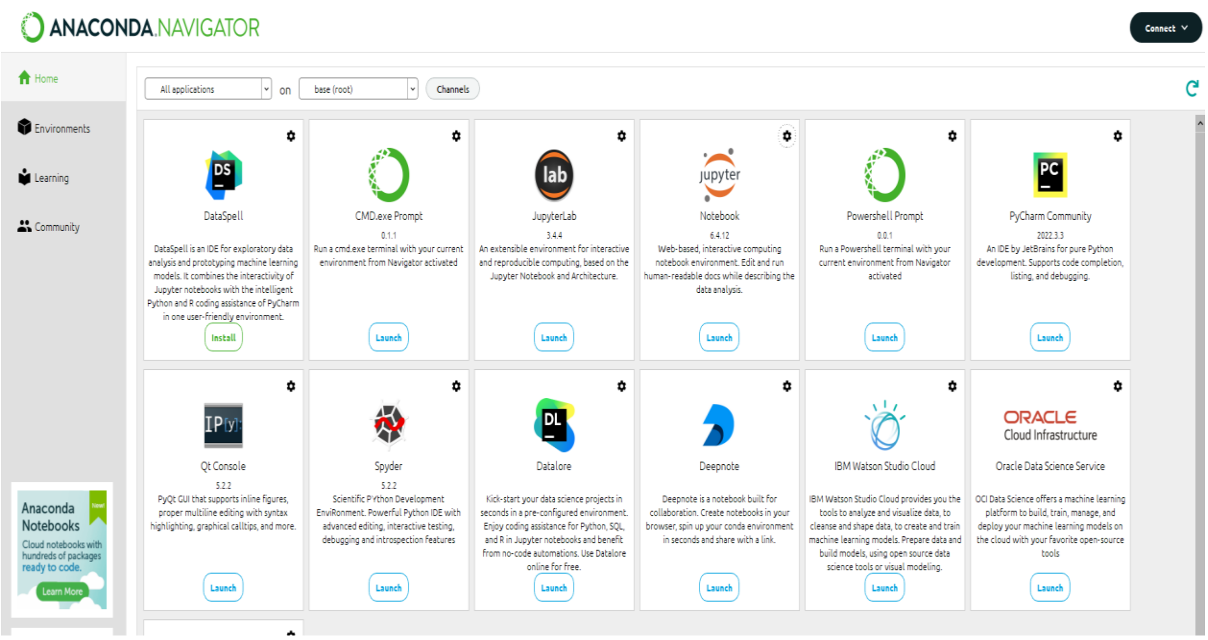


Fig 2: Screenshot of Anaconda Navigator Home Page

Fig 3: Jupyter Notebook Home Page

The Fig 2 above shows the interface of Anaconda Navigator from where we can launch the Jupyter Notebook. The Fig 3 depicts the Home page of Jupyter Notebook where we can create new notebooks or files and start executing code in it.

3.2 Data Collection

The dataset for this research is one of the largest datasets that can be used for implementation of Recommender systems, and it is sourced from Kaggle (OTTO – Multi-Objective Recommender System). This dataset contains the user behaviour data which is in the form of sessions. There are two different file train and test, and both the files are in Jsonl format in the source location. Below are the columns available in the dataset:

- Session: It is the unique id in the dataset
- Events: It represents the number of events that took place in that session.
    - Aid: It represents the product id of the item associated with the event
    - Ts: it represents the timestamp of the event.
    - Type: It represents the type of the event like ‘Clicks’, ‘Carts’, ‘Orders’.

In the dataset there can be multiple sessions for single customer but there is no visibility for the same in the data. Since the data is in session format the end output is also at the session level.
A python file (with .ipynb extension) named as “x21231036_Recommender_System_Thesis” has been created. All the scripts regarding the implementation of these Research are written and executed in this file.

3.3 Importing Libraries

There are multiple libraries used in this research. Below is the list of libraries used and their version.

Library	Version
Json	2.0.9
Pandas	1.4.4
Numpy	1.23.5
Tensorflow	2.12.0
Sklearn.model_selection.train_test_split	1.0.2
Seaborn	0.11.2
Matplotlib	3.5.2

Fig 4: Libraries with version

```
In [1]: # importing the necessary libraries
        import json
        import pandas as pd
        import numpy as np
        from tensorflow.keras.models import Sequential
        from tensorflow.keras.layers import Embedding, LSTM, Dense
        from tensorflow.keras.preprocessing.sequence import pad_sequences
        from tensorflow.keras.utils import to_categorical
        from sklearn.model_selection import train_test_split
        from sklearn.metrics.pairwise import cosine_similarity
        import matplotlib.pyplot as plt
```

Fig 5: Importing Libraries

Fig 5 represents how multiple libraries used in the entire implementation of the research are being imported in implementation python file.
Using the ‘json’ library both the train and test datasets are imported and converted from ‘jsonl’ to ‘json’ format and stored in two different data frames ‘train_df’ and ‘test_df’.

```
In [7]: train_df.head()
Out[7]:
```

	session	events
0	12899779	[{'aid': 59625, 'ts': 1661724000278, 'type': '...
1	12899780	[{'aid': 1142000, 'ts': 1661724000378, 'type':...
2	12899781	[{'aid': 141736, 'ts': 1661724000559, 'type': ...
3	12899782	[{'aid': 1669402, 'ts': 1661724000568, 'type':...
4	12899783	[{'aid': 255297, 'ts': 1661724000572, 'type': ...

Fig 6.a: Train data frame

```
In [34]: test_df.head()
Out[34]:
```

	session	events
0	12899779	[{'aid': 59625, 'ts': 1661724000278, 'type': '...
1	12899780	[{'aid': 1142000, 'ts': 1661724000378, 'type':...
2	12899781	[{'aid': 141736, 'ts': 1661724000559, 'type': ...
3	12899782	[{'aid': 1669402, 'ts': 1661724000568, 'type':...
4	12899783	[{'aid': 255297, 'ts': 1661724000572, 'type': ...

Fig 6.b: Test data frame

The above figures 6.a and 6.b shows the train and test data frames created after importing the dataset.

3.4 Exploratory Data Analysis

In this section firstly the statistics of datasets are calculated, and some basic exploratory analysis is done.

```
In [8]: # Calculating statistics for Train Data
total_sessions = train_df['session'].nunique()
total_products = train_df['events'].apply(lambda x: len(set(event['aid'] for event in x))).sum()
total_events = train_df['events'].apply(len).sum()
total_clicks = train_df['events'].apply(lambda x: sum(1 for event in x if event['type'] == 'clicks')).sum()
total_carts = train_df['events'].apply(lambda x: sum(1 for event in x if event['type'] == 'carts')).sum()
total_orders = train_df['events'].apply(lambda x: sum(1 for event in x if event['type'] == 'orders')).sum()

# Print statistics
print("Statistics for Train Data:")
print("Total unique sessions:", total_sessions)
print("Total number of products:", total_products)
print("Total number of events:", total_events)
print("Total number of clicks:", total_clicks)
print("Total number of carts:", total_carts)
print("Total number of orders:", total_orders)

Statistics for Train Data:
Total unique sessions: 5000
Total number of products: 19637
Total number of events: 28722
Total number of clicks: 25982
Total number of carts: 2335
Total number of orders: 405
```

Fig 7: Statistics for Train Data

As seen in Fig 7 Statistics of the train data is calculated. In this metrics like total number of sessions, total number of products, total number of events, total number of clicks, total number of carts and total number of orders are calculated.

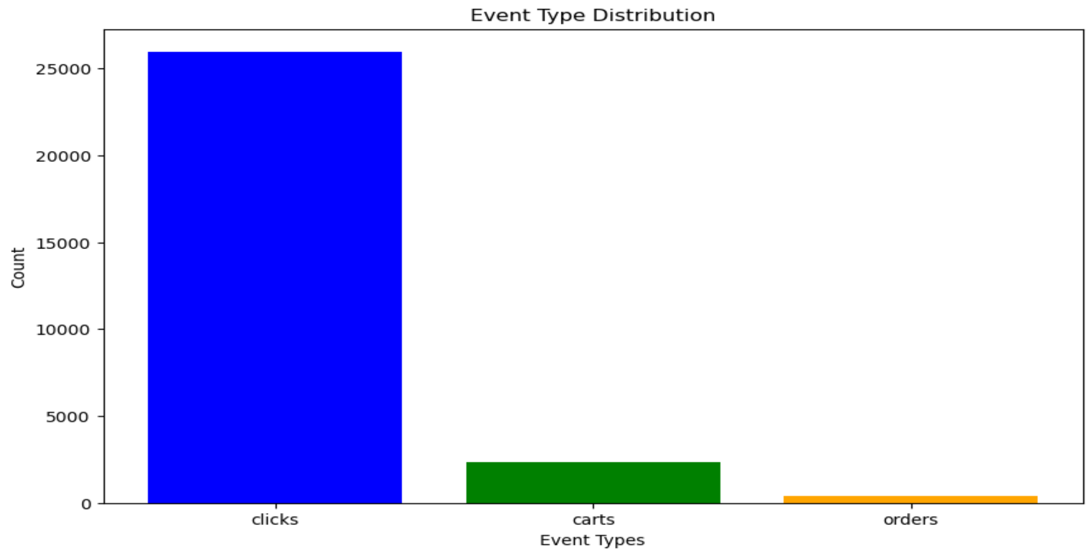


Fig 8: Bar Plot for Event type distribution

In Fig 8, Bar plot for Event type distribution is showcased where we can see maximum events in the data is of clicks event type followed by carts and orders.

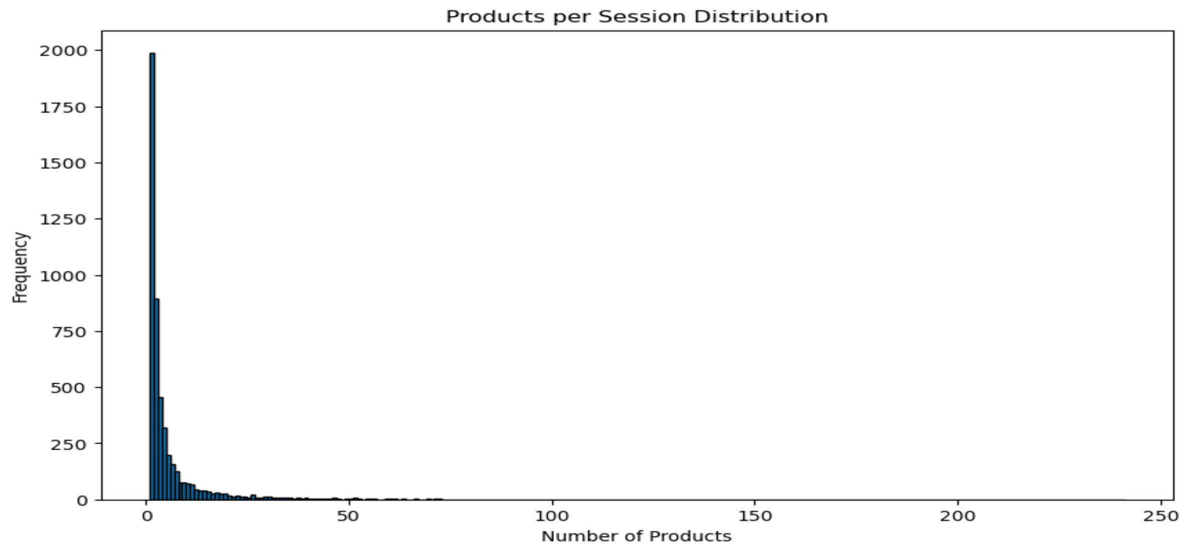


Fig 9: Histogram for Products per session

In Fig 9, Histogram for products per session is showcased, where we can see that in max number of cases there are around 1-50 products and there are very few sessions with more than 50 products.

3.5 Data Preprocessing

In the Train and Test data frames which were created in previous step, the events column contains a list multiple dictionaries in it which represents a series of events that took place in that session.

To apply models there was a need of normalizing this data and for that ".explode" function is used and the data is normalized.

```
In [28]: # Selecting the relevant columns
         # Extracting user-item interactions
         user_item_train = train_df.explode('events') # Exploding the df dataframe
         user_item_train['aid'] = user_item_train['events'].apply(lambda x: x['aid']) #
         user_item_train['type'] = user_item_train['events'].apply(lambda x: x['type'])
         user_item_train['ts'] = user_item_train['events'].apply(lambda x: x['ts']) # Ex
```

Fig 10: Code for Normalizing the data frame

```
In [32]: final_df_train.head()
Out[32]:
```

	session	aid	ts	type
1	12899780	1142000	1661724000378	clicks
1	12899780	582732	1661724058352	clicks
1	12899780	973453	1661724109199	clicks
1	12899780	736515	1661724136868	clicks
1	12899780	1142000	1661724155248	clicks

Fig 11: Output after normalizing the data frame.

In Fig 11 we can see that as the output data frame after normalizing it there are multiple rows for single session with unique aid and the new columns aid, ts and type are created.

Once this data frame is created then the weights are assigned as per the events where maximum weightage is given to orders, followed by carts and clicks.

```
In [47]:
final_df_train['ts'] = pd.to_datetime(final_df_train['ts'], unit='ms')  # Convert timestamp to datetime
weights = {'orders': 3, 'carts': 2, 'clicks': 1}  # Assigning weights to each event type
final_df_train['weight'] = final_df_train['type'].map(weights) # defining weight column

In [49]: # Grouping the data by session id and aid and sum up the weights
# Aggregating weights of duplicate entries within each session for the same product
session_product_weights = final_df_train.groupby(['session', 'aid'])['weight'].sum().reset_index()
```

Fig 12: Code for assigning weights.

In Fig 12 we can see the code is written where maximum weight '3' is assigned to orders followed by carts as '2' and clicks as '1'. A new column 'weight' is also added in the data frame.

After that the data is grouped based on session id and aid and the respective weights are aggregated and the sum is stored in the weight column. Also, these changes are stored in a new data frame "session_product_weights".

```
In [50]: session_product_weights.head()
Out[50]:
```

	session	aid	weight
0	12899780	582732	1
1	12899780	736515	1
2	12899780	973453	1
3	12899780	1142000	2
4	12899781	57315	1

Fig 13: Output after assigning weights.

```
In [51]: # Creating ground truth DataFrame with actual products for each session
         ground_truth_df = session_product_weights.groupby('session')['aid'].apply(list).reset_index()
         ground_truth_df.columns = ['session', 'actual_products']

In [52]: # Displaying the ground truth DataFrame
         print(ground_truth_df)

           session                                    actual_products
0         12899780                  [582732, 736515, 973453, 1142000]
1         12899781            [57315, 141736, 194067, 199008, 918667]
2         12899782  [45034, 127404, 229748, 363336, 406001, 413962...
3         12899783  [198385, 255297, 300127, 607638, 1114789, 1216...
4         12899784  [22981, 476216, 1036375, 1190477, 1269952, 154...
...            ...                                                ...
2112      12904768                 [886392, 1237853, 1475121, 1591574]
2113      12904770  [181414, 454926, 479970, 717779, 1200925, 1494...
2114      12904772  [2732, 14449, 239801, 543928, 1045144, 1082732...
2115      12904773                    [68526, 252695, 457060, 1683249]
2116      12904776                                   [247240, 954193]

[2117 rows x 2 columns]
```

Fig 14: Output of 'ground_truth_df'

Also, In Fig 14 a new data frame "ground_truth_df" is created where for each session the actual products are stored. This data frame will be useful for the evaluation purpose at the end.

3.6 Modelling

In this section the actual modelling is done on the normalized data frame "session_product_weights". This is done in 5 steps as explained below:

a) Creating user-item matrix

```
In [53]: # Creating a user-item interaction matrix
         user_item_matrix = session_product_weights.pivot(index='session', columns='aid', values='weight').fillna(0)

In [54]: user_item_matrix.head()
Out[54]:
```

aid	38	114	160	240	284	691	1246	1249	1514	1612	...	1854329	1854421	1854499	1854540	1854665	1854762	1854775	1855264	1855508	18
session																					
12899780	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
12899781	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
12899782	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
12899783	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
12899784	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	

5 rows × 14068 columns

Fig 15: User- item matrix

In Fig 15 we can see that user-item matrix is created. In this matrix we have session id as rows and product code (aid) as columns and the weights are stored as values.

b) Matrix factorization using Singular Value Decomposition (SVD)

```
In [55]: # Performing matrix factorization using Singular Value Decomposition (SVD)
         #  SVD is a linear algebra technique that decomposes a matrix into three matrices: U, sigma, and Vt.
         # This will help to understand underlying patterns in the data and reduce the dimensionality.
         U, sigma, Vt = np.linalg.svd(user_item_matrix.values)
```

Fig 16: Applying Matrix Factorization using SVD.

In this step matrix factorization is applied using Singular value decomposition (SVD). This technique decomposes the matrix into three matrices: U, sigma, and Vt. These matrices will help in understanding the hidden patterns in the data and will help in reducing the dimensionality.

c) Choosing Latent Features (k)

```
In [56]: # Choose the number of latent features (k)
         # Latent features represent the underlying patterns that help us approximate the original interaction matrix.
         k = 2000
         U_k = U[:, :k]
         sigma_k = np.diag(sigma[:k])
         Vt_k = Vt[:k, :]
```

Fig 17: Choosing Latent Features (k)

In this step the latent feature (k) value is set. This represents the underlying patterns which helps in approximating the original interaction matrix.

d) Reconstructing user item interaction matrix

In [57]:
```
# Reconstructing the user-item matrix using the reduced dimension representation
user_item_matrix_reconstructed = np.dot(np.dot(U_k, sigma_k), Vt_k)
```

In [58]:
```
# Converting the reconstructed matrix back to a DataFrame
user_item_df = pd.DataFrame(user_item_matrix_reconstructed, index=user_item_matrix.index, columns=user_item_matrix.columns)
```

In [59]:
```
user_item_df.head()
```

Out[59]:

aid session	38	114	160	240	284	691	1246	1249	1514	1612	...	1854329	18544
12899780	-5.511947e-09	-4.989551e-12	4.304837e-08	9.878557e-20	7.757214e-13	-1.004171e-08	-3.263617e-19	-1.105088e-08	3.029552e-19	-8.561736e-25	...	-4.764418e-24	1.500974
12899781	1.943951e-08	-5.005362e-11	-4.375313e-07	5.743307e-17	3.529891e-11	-1.183943e-05	-1.992068e-18	7.939758e-11	-1.123346e-17	1.896684e-24	...	4.792519e-18	-8.600213
12899782	3.116410e-08	7.397697e-12	-6.087782e-08	-2.900878e-18	-3.798416e-12	3.356809e-07	-2.147135e-19	-5.452430e-11	-5.376803e-18	-8.294426e-26	...	2.970826e-20	2.418894
12899783	1.595107e-17	-2.286391e-16	4.931353e-17	5.188972e-17	-1.721346e-16	-2.393055e-17	-5.846312e-18	5.579925e-17	-4.875116e-17	1.295784e-24	...	2.498061e-19	-3.952107
12899784	2.533114e-17	1.680562e-17	6.062375e-17	1.688208e-17	5.531468e-17	5.804031e-16	-6.800324e-18	1.675094e-16	-1.749356e-16	2.261214e-24	...	-1.185203e-17	-4.625680

5 rows × 14068 columns

Fig 18: Reconstructing Matrix

In this step the user- item interaction matrix is reconstructed using the reduced dimension representation.

e) Generating Recommendations

In [60]:
```
# defining a function to get top N recommendations for a given session
def get_top_n_recommendations(session, n=3):
    top_recommendations = user_item_df.loc[session].sort_values(ascending=False).index[:n]
    return top_recommendations
```

In [61]:
```
# Generating recommendations for each session
unique_sessions = final_df_train['session'].unique()
all_recommendations = {}
```

In [62]:
```
for session in unique_sessions:
    top_recommendations = get_top_n_recommendations(session)
    all_recommendations[session] = top_recommendations
```

In [63]:
```
# Displaying the final recommendations for each session
for session, recommendations in all_recommendations.items():
    print(f"Session {session}: Rec1: {recommendations[0]}, Rec2: {recommendations[1]}, Rec3: {recommendations[2]}")
```

```
Session 12899780: Rec1: 1142000, Rec2: 973453, Rec3: 736515
Session 12899781: Rec1: 199008, Rec2: 57315, Rec3: 141736
Session 12899782: Rec1: 779477, Rec2: 834354, Rec3: 595994
Session 12899783: Rec1: 255297, Rec2: 300127, Rec3: 1817895
Session 12899784: Rec1: 476216, Rec2: 1190477, Rec3: 22981
Session 12899785: Rec1: 453905, Rec2: 258458, Rec3: 41655
Session 12899787: Rec1: 1682750, Rec2: 1024433, Rec3: 1340855
Session 12899788: Rec1: 1259911, Rec2: 39846, Rec3: 245131
Session 12899789: Rec1: 1569899, Rec2: 631398, Rec3: 525156
Session 12899790: Rec1: 1219653, Rec2: 1830166, Rec3: 1735585
Session 12899791: Rec1: 915175, Rec2: 1305729, Rec3: 1501214
Session 12899792: Rec1: 1537160, Rec2: 132681, Rec3: 1672169
Session 12899793: Rec1: 1585431, Rec2: 808362, Rec3: 1132001
Session 12899798: Rec1: 1344706, Rec2: 558573, Rec3: 99354
Session 12899799: Rec1: 413826, Rec2: 1325402, Rec3: 1191715
Session 12899801: Rec1: 1750538, Rec2: 812857, Rec3: 1645990
Session 12899803: Rec1: 929024, Rec2: 1651971, Rec3: 1684753
Session 12899805: Rec1: 22940, Rec2: 967498, Rec3: 199008
Session 12899807: Rec1: 1407421, Rec2: 200257, Rec3: 409527
```

Fig 19: Generating Recommendations

In this step the function ‘get_top_n_recommendations(session, n=3)’ is defined for generating the recommendations. After that a loop is defined, where it loops through all the sessions and calls ‘get_top_n_recommendations()’ function to get the top 3 recommendations and store it into a dictionary named “all_recommendations”.
Then a for loop is defined for reading this data from the dictionary and print it in the required format “session: Rec1, rec2, rec3”.

3.7 Evaluation

In this step the functions are defined to calculate the evaluation metrics and then they are printed in required format. Also, some graphical representations of the evaluation results are generated to understand the results in a better way.

```
In [64]: # defining a function to evaluate recommendations for a given session
         def evaluate_recommendations(actual_products, recommended_products):
             # Calculate the number of true positives (common products between actual and recommended)
             true_positives = len(set(actual_products) & set(recommended_products))

             # Calculate precision
             precision = true_positives / len(recommended_products) if len(recommended_products) > 0 else 0

             # Calculate recall
             recall = true_positives / len(actual_products) if len(actual_products) > 0 else 0

             # Calculate F1-score (harmonic mean of precision and recall)
             f1_score = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

             return precision, recall, f1_score

In [65]: # List to store evaluation results
         evaluation_results = []

In [66]: for session, actual_products in ground_truth_df[['session', 'actual_products']].values:
             # Get the recommended products for the current session from the all_recommendations dictionary
             recommended_products = all_recommendations[session].tolist()

             # Evaluate the recommendations for the current session
             precision, recall, f1_score = evaluate_recommendations(actual_products, recommended_products)

             # Append the evaluation results to the list
             evaluation_results.append({'Session': session, 'Precision': precision, 'Recall': recall, 'F1-score': f1_score})

In [67]: # Converting the evaluation results to a DataFrame for easier analysis
         evaluation_df = pd.DataFrame(evaluation_results)

In [68]: # Calculating the average precision, recall, and F1-score across all sessions
         average_precision = evaluation_df['Precision'].mean()
         average_recall = evaluation_df['Recall'].mean()
         average_f1_score = evaluation_df['F1-score'].mean()
```

Fig 20: Evaluation function for Precision, Recall and F1-score.

In this step the function ‘evaluate_recommendations()’ is defined which takes actual_products and recommended_products as the parameters and in that function firstly true positives are calculated followed by precision , recall and F1-score. All these results are stored in a list called ‘evaluation_results’.
This list is then converted into data frame called ‘evaluation_df’ and then printed in the required format.

```
In [69]: # Displaying the evaluation results
         print(evaluation_df)
         print(f"\nAverage Precision: {average_precision}")
         print(f"Average Recall: {average_recall}")
         print(f"Average F1-score: {average_f1_score}")

            Session  Precision    Recall  F1-score
     0     12899780   1.000000  0.750000  0.857143
     1     12899781   1.000000  0.600000  0.750000
     2     12899782   1.000000  0.078947  0.146341
     3     12899783   1.000000  0.333333  0.500000
     4     12899784   1.000000  0.428571  0.600000
     ...        ...        ...       ...       ...
     2112  12904768   1.000000  0.750000  0.857143
     2113  12904770   1.000000  0.375000  0.545455
     2114  12904772   1.000000  0.300000  0.461538
     2115  12904773   1.000000  0.750000  0.857143
     2116  12904776   0.666667  1.000000  0.800000

     [2117 rows x 4 columns]

     Average Precision: 0.9195402298850599
     Average Recall: 0.6413722633807162
     Average F1-score: 0.6817285135355987
```

Fig 21: Evaluation results for Precision, Recall and F1-score.

```
In [71]: # Calculating hit rate and coverage
         hit_rate_list = []
         recommended_products_set = set()

         for session, actual_products in ground_truth_df[['session', 'actual_products']].values:
             recommended_products = all_recommendations.get(session, [])

             if len(recommended_products) > 0:
                 hit_rate_list.append(1)
                 recommended_products_set.update(recommended_products)
             else:
                 hit_rate_list.append(0)

In [72]: # Calculating metrics
         total_sessions = len(ground_truth_df)
         hit_rate = np.mean(hit_rate_list)

In [73]: # Calculating the total number of unique products
         unique_products = set()
         for products in ground_truth_df['actual_products']:
             unique_products.update(products)
         total_unique_products = len(unique_products)

In [74]: # Calculating hit rate and coverage
         coverage = len(recommended_products_set) / total_unique_products

         print(f"Hit Rate: {hit_rate}")
         print(f"Coverage: {coverage}")

         Hit Rate: 1.0
         Coverage: 0.36174296275234574
```

Fig 22: Code and Results for Hit Rate and Coverage

The Fig 22 above shows that how Hit Rate and Coverage is Calculated.

Finally, we got precision as 0.92, recall as 0.64, F1-score as 0.68, Hit Rate as 1 and Coverage as 0.36. Precision of 0.92 is quite good and recall value of 0.64 is acceptable as it suggests that 64% of recommendations generated were relevant. the F1-score of 0.68 is balancing both precision and recall quite well. The system achieved Hit rate of 1 which is excellent as it is the ideal value, and it reflects that at least 1 product from the recommended once is from the user's session whereas coverage of 0.36 is a point of concern which means that the recommender system is not recommending a wider range of products. To solve that a larger pool of products is to be considered in future.