

AN IMPROVED PATTERN MINING APPROACH TO ENHANCE THE PERFORMANCE OF RECOMMENDATION SYSTEM IN BIG DATA

*Report submitted to the SASTRA Deemed to be University
as the requirement for the course*

CSE300 / INT300 / ICT300 - MINI PROJECT

Submitted by :

LAVANYA R

(Reg. No: 124003159, B.Tech CSE)

VIDHYA S

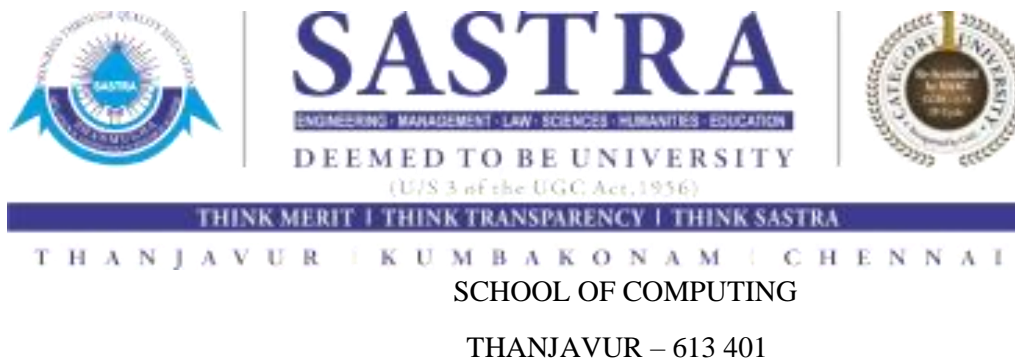
(Reg. No: 124003360, B.Tech CSE)

SAI JANANI A S

(Reg. No:124015079, B.Tech IT)

May 2023





Bonafide Certificate

This is to certify that the report titled “**An Improved Pattern mining approach to enhance the performance of Recommendation System in BigData**” submitted as a requirement for the course, CSE300 / INT300 / ICT300: **MINI PROJECT** for B.Tech. is a bonafide record of the work done by Ms.LAVANYA R (Reg. No: 124003159, B.Tech CSE), Ms.VIDHYA S (Reg. No: 124003360, B.Tech CSE), Ms.SAI JANANI A S (Reg. No:124015079, B.Tech IT) during the academic year 2022-23, in the School of Computing, under my supervision.

Signature of Project Supervisor :

Name with Affiliation :

Date :

Mini Project Viva voce held on _____

Examiner 1

Examiner 2

Acknowledgements

We would like to thank our Honorable Chancellor Prof. R. Sethuraman for providing us with an opportunity and the necessary infrastructure for carrying out this project as a part of our curriculum.

We would like to thank our Honorable Vice-Chancellor Dr. S. Vaidhyasubramaniam and Dr. S. Swaminathan, Dean, Planning & Development, for the encouragement and strategic support at every step of our college life.

We extend our sincere thanks to Dr. R. Chandramouli, Registrar, SASTRA Deemed to be University for providing the opportunity to pursue this project.

We extend our heartfelt thanks to Dr. A. Umamakeswari, Dean, School of Computing, Dr. S. Gopalakrishnan, Associate Dean, Department of Computer Application, Dr. B.Santhi, Associate Dean, Research, Dr. V. S. Shankar Sriram, Associate Dean, Department of Computer Science and Engineering, Dr. R. Muthaiah, Associate Dean, Department of Information Technology and Information & Communication Technology

Our guide Rajendran P, Assistant Professor-I, School of Computing was the driving force behind this whole idea from the start. His deep insight in the field and invaluable suggestions helped us in making progress throughout our project work. We also thank the project review panel members for their valuable comments and insights which made this project better.

We would like to extend our gratitude to all the teaching and non-teaching faculties of the School of Computing who have either directly or indirectly helped us in the completion of the project.

We gratefully acknowledge all the contributions and encouragement from my family and friends resulting in the successful completion of this project. We thank you all for providing me an opportunity to showcase my skills through project.

LIST OF FIGURES

FIGURE NO	TITLE	PAGE NO
Fig 1.1	Architecture Diagram	3
Fig 4.1	No of clusters vs DB Index	33
Fig 4.2	Genre vs Tag	33
Fig 4.3	Predictions	34
Fig 4.4	Preprocessed table	34
Fig 4.5	Association rules	35
Fig 4.6	Sparsity level	35
Fig 4.7	Support vs Execution time	36
Fig 4.8	Confidence level vs Number of rules	36
Fig 4.9	Top rules	37
Fig 4.10	Top itemsets	37
Fig 4.11	Top N recommendations	38
Fig 4.12	Heat map	38
Fig 4.13	Accuracy for different samples	39
Fig 4.14	Precision for different samples	39
Fig 4.15	Recall value for different samples	40
Fig 4.16	F-Score for different samples	40
Fig 4.17	Metrics for different samples	41

ABBREVIATIONS

CF	Collaborative filtering			
DB	Davies-Bouldin index			
FP		Frequent	Pattern	
FPM		Frequent	Pattern	Mining
HIUM		High	Utility	Itemset Mining
MPSO	Multi-objective	Particle	Swarm	Optimization
RDD		Resilient	Distributed	Dataset
SPM		Sequential	Pattern	Mining
SVD	Singular Value Decomposition			

Abstract

Data sparsity is so frequent in recommendation systems. Due to the fact that most active users typically only evaluated a tiny percentage of things, it is challenging to locate sufficiently trustworthy similar individuals. Two-level preprocessing methodology decreases data sparsity in the recommendation system. To further understand and analyse the user preferences' behaviour, additional resources such as item genre, tag, and time are included.

The proposed method recommends the movie items, based on users choice and avoids recommending the deprecated items and it also addresses the overlapping conditions that exist on item's genre. On the basis of related movie's genre and tag attributes, users are categorised. Additionally, it examines the user's dynamic interest. It shrinks the dimensions, which is a first step in data preparation and pattern analysis. The suggested strategy made use of Apache Spark Mllib FP-Growth and association rule mining in a distributed setting to improve performance. The candidate data set is kept in matrix form to lower the computation cost of building the tree in FP-Growth. The outcome of the proposed methodology is rendered by parameters like precision, accuracy, recall, F score and Sparsity level.

KEY WORDS: Hidden Behavioural analysis ,Big data , Fp-Growth, Association rule mining, Bisecting k means clustering, Associative classification.

Table of Contents

Title	Page.No
Bonafide Certificate	ii
Acknowledgements	iii
List of Figures	iv
Abbreviations	v
Abstract	vi
Summary of base paper	1
Merits and Demerits of the base paper	6
Source code	8
Output snapshots	33
Conclusion and future plans	42
References	43

CHAPTER 1

SUMMARY OF BASE PAPER

Title: An improved hidden behavioral pattern mining approach to enhance the performance of recommendation system in a big data environment

Journal Name:Journal of King Saud University-Computer and Information Sciences.

Publisher:ScienceDirect

Date of Publication:November ,2022

Indexing:Science Citation Index Expanded and Scopus

1.1 INTRODUCTION

Recommendation systems have become an essential component of our daily lives, providing personalised suggestions for products, services, and contents. Web applications that we use everyday such as YouTube, Netflix, Spotify gives us the best recommendations for us. There are two types of recommendation system. One is collaborative filtering where the recommendations are provided using the user preference category such as ratings and it could be further classified as Memory based and model based. Second is content filtering method where it uses only the similarities between the features of the things, thus it necessitates thorough knowledge of the items. It rates the items based on similarity and suggests the top N items to the consumers. Due to item sparsity matrix in ratings matrix collaborative filtering gives poor recommendation. To solve this sparsity problem methods such as clustering, SVD and classification are used but it takes a lot of computation time and it is less accurate. In order to decrease the amount of data at the item level, two-level pre-processing techniques are utilised to address this sparsity issue. Analyse the hidden correlation between the user's interest items using association rules. The if-then pattern for the frequent itemset is provided by these rules. For better recommendations, it then categorises and predicts the user's selection.

1.2 RELATED WORK

The contribution of the research project is mainly to work out the data sparsity in collaborative filtering methods. In order to solve the problem data reduction is done using bisecting k means clustering methodology. To calculate the user preference some features like timestamp, tag and genre is used. Items are grouped in accordance with the user's interests which is done after analyzing the hidden behavioral pattern of the user. The FP-Growth algorithm is distributed and employed in parallel to make the recommendation system perform effectively. The form matrix stores the components that are used frequently, which saves memory compared to tree. In comparison to the Apriori algorithm, FP growth lessens the number of scans in candidate data sets. Using these patterns formed by FP-Growth algorithm the association rules are generated and with the help of these rules the associative classification is done. After classification top N recommendations are provided with the generated model.

1.3 PROPOSED METHODOLOGY

The proposed method is to reduce the data sparsity and it first identifies the user preference with the resources such as: Tags, ratings, user profile and item profile. It then identifies the pattern using item matrix which incorporates the user chosen movie aspects. The proposed approach takes place in three phases:

1. Preprocessing
2. Generate Association rules
3. Recommendation using associative classification

In preprocessing step the incomplete information from the movielens dataset is removed and clustering is performed in the items features such as genres and Tag. In the second phase hidden behavior of the user is found by forming association rules using FP-Growth and strong rules are selected. The final phase is where the recommendation is provided by building an associative classifier model from the association rules formed and thus this model provides top N recommendations.

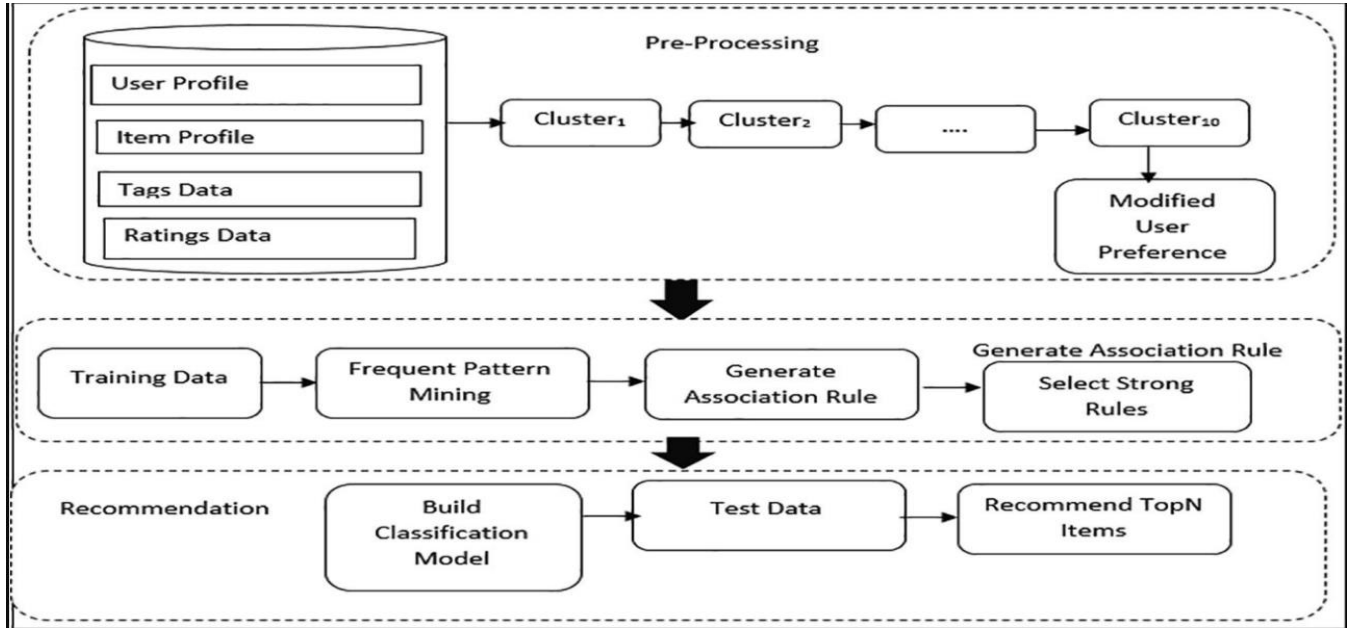


Fig 1.1 Architecture Diagram

1.3.1 PREPROCESSING

It is basic method of getting the data ready in order to remove the noises, missing values from the raw data. In the proposed model clustering is also performed in the preprocessing step so that similar user preferred movies are grouped. Since there is more than one genre for a single movie there is a need to categorize the movie into proper label.

1.3.1.1 Bisecting K-Means Clustering

Bisecting KMeans algorithm is used for grouping the items. The cluster is created using similarities between the genre, tag, and timestamp of the movies where the one hot encoder is used for converting the categorical into numeric array. The cluster size is determined by various methods such as silhouette score, Elbow curve method and DB Index in which DB Index gives the optimal value of k. Then the cluster names are based on the genre that appears the most frequently. Each cluster is then aggregated based on the timestamp where threshold or the limit is fixed as average time difference between starting and ending year. With the threshold value users are categorized into 'recent' which gets the higher ranking compared to all if timestamp is greater than threshold value, 'old' if it is less than threshold and 'medium' otherwise. Finally preference level is calculated with the formula as given:

$$Preference\ Level = \frac{Rating\ of\ item\ with\ in\ the\ cluster}{Maximum\ rating\ from\ a\ user\ profile}$$

The calculated item is divided into three categories based on the preference level: "high," "low," and "medium." With the above method grouping items is done and this simplifies the association ruling process.

1.3.2 GENERATE ASSOCIATION RULES

After preprocessing we need to forecast the hidden choice of the user. The two steps for finding the hidden pattern are:

1. Detecting the frequent items
2. Build association rules

Finding frequent itemsets is used to find the relationship between the items i.e. the movies. These items occur frequently in a dataset which is measured using support count : $\text{Support}(A \rightarrow B) = \text{Support_count}(A \cup B)$. If the support is less than threshold support count then it is not a frequent itemset.

1.3.2.1 FP-Growth Algorithm

It is an efficient method for generating frequent items and it uses divide and conquer strategy. It reduces input dataset by creating fp-tree instance. It then divides the dataset into a set of conditional databases and finally each dataset is mined. The proposed FP-Growth uses a matrix to store the frequent itemsets which reduces the memory utilization.

Selecting the support count is performed by plotting the graph between execution time and array of support count. The confidence level is also found using the formula:

$$\text{Confidence}(A \rightarrow C) = \text{Support_count}(A \cup C) / \text{support_count}(A)$$

Additionally lift is also measured with the formula:

$$\text{Lift}(A \rightarrow C) = \text{PLift}(A \rightarrow C) = \text{Probability}(A \& C) / (\text{Support}(A) * \text{Support}(C))$$

The association rules are formed using the grouped items in the preprocessing step as antecedent and subsequent consequent are formed with the corresponding support count, confidence and lift measure.

These association rules are formed using spark mllib since it offers a scalable and effective data structure similar to the RDD, which specifies an immutable collection of components executed in parallel. The system's effectiveness is

increased by storing RDD in cache memory. These rules of association make it easier to find hidden patterns.

1.3.3 CLASSIFICATION AND RECOMMENDATION

Classification is performed from the association rules using the associative classifier. Based on the association rules from the training dataset the strong rules are chosen using the support and confidence measure. If minimum support is not met then those rules are not selected for classification. These rules are then arranged by confidence values in descending order. Based on these selected rules predictions are made on the class labels i.e) consequent itemsets for the test data. Then the experimental parameters such as accuracy, precision, recall and f1-score are computed for the sparsity level found.

CHAPTER 2

MERITS AND DEMERITS OF THE BASE PAPER

2.1 Merits

Many online applications, including those for online business news, music, and healthcare, use recommendation systems to deliver their services and products according to the needs and preferences of their users.

There are many methods for analysing hidden knowledge. Few examples are Frequent pattern mining, Sequential pattern mining and high utility pattern mining. The extraction of hidden data patterns from the data depends significantly on frequent pattern mining techniques. Numerous algorithms were created for this as a result of technological advancements.

In transactional databases, one of the main strategies for data mining is the use of association rules to find frequent patterns (FP), connections, correlations, or any kind of structures between groups of items or objects. It supports Apriori, FP-Tree, and Fuzzy FP-Tree algorithms.

One of the biggest advantages of FP-Growth algorithm is requires less number of scans of the transactional database which makes it better than Apriori algorithm.

The data sparsity problem is handled by a hybrid strategy using clustering and association rule mining. This is done based on the user profile and item profile. Association rule mining enables the use of secondary data sources such as tags which increases the level preference. It also improves precision by removing dependencies. Using large datasets leads to increased rule generation but at the same time increases the prediction complexity.

Results are more precise when association rule mining is used in recommendation systems.

2.1.1 Advantage of Fp_tree:

Sets in a transaction database can be compressed into smaller ones. The size of transaction database's frequent pattern set necessitates regular access to the patterns. Some patterns are reduced to improve performance in order to obtain frequent patterns more easily. In a given frequent patterns set, if two patterns exist with the same frequent pattern and support count, one of those can be removed. If the support counts are the same and one of the frequent pattern is a subset of another, then also it can be removed.

2.2 Demerits

The most important factors in association rule mining that affect accuracy are support and confidence metrics. The computation cost becomes high when a large number of rules are generated.

The quality of the recommendation system is known to improve when support and confidence are treated as different objectives because only items associated with the rule are mined so the computation cost is reduced. But, when it comes to a huge dataset, it becomes difficult to find frequent patterns using sequential data processing since it takes a lot of time to execute and use a lot of memory. Mining frequent parallel patterns improves performance by distributing data and processing across numerous computers.

2.3 Results and discussion

The proposed method gives results with higher accuracy when compared to the basic Collaborative Filtering (CF) method. It can be observed that the effectiveness of the basic CF method declines as the sparsity levels increase. The basic CF method is bound to suffer from sparsity problems because poor recommendations are known to originate from poor neighbourhood formation. The problem of data sparsity has been resolved by the suggested method. In the MovieLens data set, inaccurate neighbour selection produces useless recommendations. It is evident from the data that the proposed technique performs better at different levels of sparsity. The suggested method analyses user behaviour in addition to forecasting the preference score. In Order to understand the pattern of movies a user shows interest in, the data is handled equally. As a result, the researcher believed that association rule mining was preferable to other conventional techniques.

CHAPTER 3

SOURCE CODE

PROPOSED METHODOLOGY CODE:

```
!pip install pyspark
from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.clustering import BisectingKMeans
spark = SparkSession.builder.appName("BisectingKMeansClustering").getOrCreate()
spark = SparkSession.builder.appName("BisectingKMeansClustering").getOrCreate()
movies= spark.read.csv('Movies.csv', header=True, inferSchema=True)
tags = spark.read.csv("tags.csv", header=True, inferSchema=True)
print(list(tags.columns))
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import VectorAssembler, StandardScaler
genresIndexer = StringIndexer(inputCol="Genres", outputCol="genresIndex")
genresDF = genresIndexer.fit(movies).transform(movies)
time_scaler = StandardScaler(inputCol='Timestamp', outputCol='time_scaled')
tagIndexer = StringIndexer(inputCol="Tag", outputCol="tagIndex")
tagDF = tagIndexer.fit(tags).transform(tags)
from pyspark.ml.feature import OneHotEncoder
from pyspark.ml.clustering import KMeans
from pyspark.ml.evaluation import ClusteringEvaluator
df=genresDF.join(tagDF,"MovieID")
df.show(2)
from pyspark.sql.functions import col
joinedDF = df.withColumn("float_time", col("Timestamp").cast("float"))
assembler = VectorAssembler(inputCols=['genresIndex', 'tagIndex', 'float_time'], outputCol='features')
vectors=assembler.setHandleInvalid("skip").transform(joinedDF)
from pyspark.ml.feature import BucketedRandomProjectionLSH
```

```

lsh = BucketedRandomProjectionLSH(inputCol='features', outputCol='hashes', bucketLength=2.0,
numHashTables=3)
model = lsh.fit(vectors)
similarity = model.approxSimilarityJoin(vectors, vectors, threshold=0.8, distCol='cosine_similarity')
evaluator = ClusteringEvaluator(featuresCol='features', predictionCol='prediction',
metricName='silhouette')
scores = []
for k in range(2, 15):
    kmeans = KMeans(featuresCol='features', k=k, seed=1)
    model = kmeans.fit(vectors)
    predictions = model.transform(vectors)
    score = evaluator.evaluate(predictions)
    scores.append(score)
optimal_k = scores.index(min(scores)) + 2
kmeans = KMeans(featuresCol='features', k=optimal_k, seed=1)
model = kmeans.fit(vectors)
predictions = model.transform(vectors)
db_index = evaluator.evaluate(predictions, {evaluator.metricName: 'silhouette'})
print(db_index)
import matplotlib.pyplot as plt

plt.plot(range(2, 15), scores)
plt.xlabel('Number of clusters')
plt.ylabel('Davies-Bouldin index')
plt.title('DB index vs Number of clusters')
plt.show()
from pyspark.ml.evaluation import ClusteringEvaluator
import matplotlib.pyplot as plt
k=optimal_k
bkm = BisectingKMeans(featuresCol='features', k=10, seed=1)
model = bkm.fit(vectors)
predictions = model.transform(vectors)

```



```

features = predictions.select('features').rdd.map(lambda x: x[0]).collect()
labels = predictions.select('prediction').rdd.map(lambda x: x[0]).collect()
plt.scatter([x[0] for x in features], [x[1] for x in features], c=labels)
plt.xlabel('Genre')
plt.ylabel('Tag')
plt.title('Bisecting KMeans Clustering')
plt.show()
predictions.show(10)

from pyspark.sql.functions import split, explode

# split the Genres column by "|" and explode it into multiple rows
predictions = predictions.withColumn("genre", explode(split("Genres", "|")))

# group by the prediction and genre columns to count the frequency of each genre within each cluster
counts = predictions.groupby("prediction", "genre").count()

# use window functions to get the genre with the highest frequency within each cluster
from pyspark.sql.window import Window
from pyspark.sql.functions import rank

window = Window.partitionBy("prediction").orderBy(counts["count"].desc())
top_genre = counts.select("*", rank().over(window).alias("rank")).filter("rank == 1")

# join the top_genre dataframe with the original dataframe to get the cluster names
cluster_names = predictions.join(top_genre, ["prediction", "genre"]).select("prediction", "genre",
"count")

predictions.show()

from pyspark.sql.functions import desc
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

```

```

from pyspark.sql.functions import avg
# calculate the average timestamp for each cluster
avg_timestamp = predictions.groupBy('prediction').agg(avg('timestamp').alias('avg_timestamp'))

# join the average timestamp with the original dataframe
joined_df = predictions.join(avg_timestamp, on='prediction')

# calculate the time difference for each item
joined_df = joined_df.withColumn('time_diff', col('avg_timestamp') - col('timestamp'))

# calculate the average time difference for each cluster
avg_time_diff = joined_df.groupBy('prediction').agg(avg('time_diff').alias('avg_time_diff'))

# calculate the threshold value based on the average time difference
threshold = avg_time_diff.select(avg('avg_time_diff')).collect()[0][0]

# categorize the users into recent, medium, and old categories
from pyspark.sql.functions import when
joined_df = joined_df.withColumn('user_category',
                                when(col('time_diff') < threshold, 'Old')
                                .when(col('time_diff') > threshold, 'Recent')
                                .otherwise('Medium'))

# define the window specification
from pyspark.sql.functions import avg, col, when, rank
from pyspark.sql.window import Window

rank_window = Window.partitionBy('prediction', 'user_category').orderBy(col('timestamp').desc())

# rank the items within each category based on the timestamp

```

```

joined_df = joined_df.withColumn('rank', rank().over(rank_window))
# show 10 random rows from joined_df
joined_df.show()
joined_df.sample(fraction=0.3).show(10)
rating= spark.read.csv('ratings.csv', header=True, inferSchema=True)
rating= spark.read.csv('ratings.csv', header=True, inferSchema=True)
from pyspark.sql.functions import monotonically_increasing_id
# Read in the original dataset
ratings = spark.read.csv('ratings.csv', header=True, inferSchema=True)

# Assign a unique ID to each row in the dataset
ratings = ratings.withColumn('row_id', monotonically_increasing_id())

# Split the dataset into five equal parts based on the row_id column
part1 = ratings.filter((ratings.row_id >= 0) & (ratings.row_id < 2000000))
part2 = ratings.filter((ratings.row_id >= 0) & (ratings.row_id < 4000000))
part3 = ratings.filter((ratings.row_id >= 0) & (ratings.row_id < 6000000))
part4 = ratings.filter((ratings.row_id >= 0) & (ratings.row_id < 8000000))
#part5 = ratings.filter((ratings.row_id >= 8000000) & (ratings.row_id < 10000000))

# Write each part to a separate CSV file
part1.write.csv('sample1.csv', header=True)
part2.write.csv('sample2.csv', header=True)
part3.write.csv('sample3.csv', header=True)
part4.write.csv('sample4.csv', header=True)
#part5.write.csv('part5.csv', header=True)

#10 million
from pyspark.sql.functions import max, when, col
# calculate the maximum rating for each user
max_rating_df = rating.groupBy('UserID').agg(max('Rating').alias('max_rating'))
# join the maximum rating with the joined_df

```

```
data_df = rating.join(max_rating_df, ['UserID'])
```

```
data_df.show(2)#10m
```

#Instead of the above code within comments #10million #10m

#We can substitute the below codes for 2m,4m,6m,8m,after splitting

```
data_2m, data_4m, data_6m, data_8m = rating.randomSplit([0.2, 0.4, 0.2, 0.2])
```

#2million

calculate the maximum rating for each user

```
max_rating_df = data_2m.groupBy('UserID').agg(max('Rating').alias('max_rating'))
```

join the maximum rating with the joined_df

```
data_df = data_2m.join(max_rating_df, ['UserID'])
```

```
data_df.show(2) #2m
```

#4million

calculate the maximum rating for each user

```
max_rating_df = data_4m.groupBy('UserID').agg(max('Rating').alias('max_rating'))
```

join the maximum rating with the joined_df

```
data_df = data_4m.join(max_rating_df, ['UserID'])
```

```
data_df.show(2) #4m
```

#6million

calculate the maximum rating for each user

```
max_rating_df = data_6m.groupBy('UserID').agg(max('Rating').alias('max_rating'))
```

join the maximum rating with the joined_df

```
data_df = data_6m.join(max_rating_df, ['UserID'])
```

```
data_df.show(2) #6m
```

#8million

calculate the maximum rating for each user

```
max_rating_df = data_8m.groupBy('UserID').agg(max('Rating').alias('max_rating'))
```

```

# join the maximum rating with the joined_df
data_df = data_8m.join(max_rating_df, ['UserID'])
data_df.show(2)

#8m

rating_df1 = data_df.withColumn('preference_level', col('rating')/col('max_rating'))
# categorize the preference level into high, low, and medium
rating_df = rating_df1.withColumn('preference_category',
                                when(col('preference_level') > 0.7, 'high')
                                .when(col('preference_level') < 0.3, 'low')
                                .otherwise('medium'))

from pyspark.sql.functions import col
final_df=rating_df.join(joined_df,"UserID")
# show 10 random rows from joined_df
final1=final_df.sample(fraction=0.1)
#final.show()

from pyspark.sql.functions import concat, col, lit

# Assuming the DataFrame is named "df"
df_concat1 = final1.withColumn("combined_category",
                                concat(col("user_category"), lit(", "),
                                       col("preference_category"), lit(", "),
                                       col("genre")))

new_df1=df_concat1.select("UserID","MovieID", "Title","user_category", "preference_category",
                          "genre","combined_category")
new_df1.sample(fraction=0.3).show()

# Split the association rules into training and testing datasets
train_ratio = 0.8
test_ratio = 1 - train_ratio
seed = 12345

```

```

train_data1, test_data1 = new_df1.randomSplit([train_ratio, test_ratio], seed=seed)
from pyspark.sql.functions import col, collect_set, concat_ws
from pyspark.ml.fpm import FPGrowth

# Sample a fraction of the data
fraction = 0.1
sampled_df = train_data1.sample(fraction=fraction)

# Group the data by UserID and collect the combined categories in a set for each user
user_categories =
train_data1.groupBy("UserID").agg(collect_set("combined_category").alias("categories"))

# Run FPGrowth on the sampled data
fpGrowth = FPGrowth(itemsCol="categories", minSupport=0.1, minConfidence=0.5)
model = fpGrowth.fit(user_categories)
association_rules = model.associationRules

# Add aliases to the generated columns

association_rules = association_rules.withColumnRenamed("UserID", "userId")
association_rules = association_rules.withColumnRenamed("antecedent", "antecedent_itemset")
association_rules = association_rules.withColumnRenamed("consequent", "consequent_itemset")
association_rules = association_rules.withColumnRenamed("confidence", "confidence_level")

# Show the association rules
association_rules.show()
import matplotlib.pyplot as plt
import time

# Initialize empty lists for support and execution time
supports = []

```

```

execution_times = []

# Loop through support values and record execution time for each
for support in range(1, 10):
    support /= 10.0 # Convert to float
    start_time = time.time()
    fpGrowth = FPGrowth(itemsCol="categories", minSupport=support, minConfidence=0.5)
    model = fpGrowth.fit(user_categories)
    end_time = time.time()
    execution_time = end_time - start_time
    supports.append(support)
    execution_times.append(execution_time)

# Create a line plot
plt.plot(supports, execution_times)

# Customize plot
plt.title('Support vs Execution Time')
plt.xlabel('Support Level')
plt.ylabel('Execution Time (seconds)')

# Display plot
plt.show()

import matplotlib.pyplot as plt

# count the number of rules at each confidence level
num_rules_by_confidence =
association_rules.groupBy("confidence_level").count().orderBy("confidence_level")

# extract the confidence levels and counts as arrays

```

```

confidences = num_rules_by_confidence.select("confidence_level").rdd.flatMap(lambda x: x).collect()
num_rules = num_rules_by_confidence.select("count").rdd.flatMap(lambda x: x).collect()

# plot the results
plt.plot(confidences, num_rules)
plt.xlabel("confidence level")
plt.ylabel("number of rules")
plt.show()

# Select strong association rules based on confidence level, lift value, and support count
min_confidence = 0.8
min_lift = 1.2
min_support = 0.2

strong_rules1 = association_rules.filter((col("confidence_level") >= min_confidence) &
                                         (col("lift") >= min_lift) &
                                         (col("support") >= min_support))

# Sort the rules by decreasing confidence level
#sorted_rules = strong_rules.sort(col("confidence_level").desc())

# Show the top 10 rules
strong_rules1.show(10)

# Select the top n rules based on confidence level
n = 10

# Get the top itemsets from the association rules
top_rules = association_rules.sort(col("confidence_level").desc()).limit(10)

#top_itemsets = set(top_rules.select(col("antecedent_itemset")).rdd.flatMap(lambda x:
x).map(tuple).collect() + top_rules.select(col("consequent_itemset")).rdd.flatMap(lambda x:
x).map(tuple).collect())

top_itemsets = set([tuple(itemset) for itemset in
top_rules.select(col("antecedent_itemset")).rdd.flatMap(lambda x: x).collect() +

```



```

top_rules.select(col("consequent_itemset")).rdd.flatMap(lambda x: x).collect())
top_rules.show()
#training_data = train_data.filter(col("combined_category").isin(top_itemsets))
print("Top itemsets:")
for itemset in top_itemsets:
    print(itemset)

num_users = data_2m.select("UserId").distinct().count()
print("Number of users:", num_users)
num_items = data_2m.select("MovieId").distinct().count()
print("Number of items:", num_items)

from pyspark.sql.functions import col

# calculate the number of ratings and the sparsity level for each row
num_ratings = col("confidence_level") * num_users * num_items
sparsity_level = (1 - (num_ratings / (num_users * num_items))).alias("sparsity_level")

# add the sparsity level column to the DataFrame
rule_df1 = strong_rules1.withColumn("sparsity_level", sparsity_level)

# show the resulting DataFrame with the sparsity level column
rule_df1.sample(fraction=0.1).show()
from pyspark.ml.feature import Bucketizer

# create bucket boundaries
boundaries = [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
rule_df1.drop("bucket")
rule_df1.drop("sparsity_buckets")
rule_df1.drop("sparsity_bucket")
rule_df1.drop("bucket_sparsity")

```

```

rule_df1.drop("Bucket_sparsity")
rule_df1.drop("Bucket")
# create the bucketizer
bucketizer = Bucketizer(splits=boundaries, inputCol="sparsity_level", outputCol="Buckett")

# bucketize the sparsity levels
rule_df = bucketizer.transform(rule_df1)
#rule_df.show()
from pyspark.ml.feature import VectorAssembler

# Create a vector assembler to combine the confidence_level column into a feature vector
assembler = VectorAssembler(inputCols=["confidence_level"], outputCol="features")

# Transform the rule_df DataFrame to add the features column
df_with_features1 = assembler.transform(rule_df1)
from pyspark.ml.feature import Bucketizer

bucketizer = Bucketizer(splits=[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
inputCol="sparsity_level", outputCol="bucket")
df_with_buckets1 = bucketizer.transform(df_with_features1)
df_with_buckets1.show(10)
(trainingData1, testData1) = df_with_buckets1.randomSplit([0.8, 0.2])
from pyspark.ml.classification import DecisionTreeClassifier

dt = DecisionTreeClassifier(labelCol="bucket", featuresCol="features")
model1 = dt.fit(trainingData1)

# Make predictions on the test data
predictions1 = model1.transform(testData1)
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

```

```

# Select (prediction, true label) and compute test error
evaluator = MulticlassClassificationEvaluator(
    labelCol="bucket", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions1)

print("Accuracy = %g" % (accuracy))

evaluator = MulticlassClassificationEvaluator(
    labelCol="bucket", predictionCol="prediction", metricName="weightedPrecision")
weightedPrecision = evaluator.evaluate(predictions1)

print("Weighted Precision = %g" % (weightedPrecision))

evaluator = MulticlassClassificationEvaluator(
    labelCol="bucket", predictionCol="prediction", metricName="weightedRecall")
weightedRecall = evaluator.evaluate(predictions1)
print("Weighted Recall = %g" % (weightedRecall))

evaluator = MulticlassClassificationEvaluator(
    labelCol="bucket", predictionCol="prediction", metricName="f1")
f1 = evaluator.evaluate(predictions1)

print("F1 Score = %g" % (f1))

from pyspark.ml.classification import DecisionTreeClassifier
import time

# Start the timer
start_time = time.time()

# Train the model

```

```

dt = DecisionTreeClassifier(labelCol="bucket", featuresCol="features")
model = dt.fit(trainingData1)

# Make predictions on the test data
predictions = model.transform(testData1)

# Stop the timer and print the elapsed time
elapsed_time = time.time() - start_time
print(f"Computation time 2m: {elapsed_time:.2f} seconds")

from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Create a MulticlassClassificationEvaluator instance with label and prediction columns
evaluator = MulticlassClassificationEvaluator(labelCol="bucket", predictionCol="prediction",
metricName="accuracy")

# Calculate the accuracy

# Calculate the error rate
error_rate = 1.0 - (accuracy)

#print("Accuracy = {:.2f}%".format(accuracy*100))
print("Error Rate = {:.2f}%".format(error_rate*100))

from pyspark.ml.evaluation import Evaluator

class CostEvaluator(Evaluator):
    def __init__(self, labelCol='label', predictionCol='prediction', costMatrix=[[0, 1], [1, 0]]):
        self.labelCol = labelCol
        self.predictionCol = predictionCol
        self.costMatrix = costMatrix

```

```

def _evaluate(self, dataset):

    tp = dataset.filter((dataset[self.labelCol] == 1) & (dataset[self.predictionCol] == 1)).count()
    fp = dataset.filter((dataset[self.labelCol] == 0) & (dataset[self.predictionCol] == 1)).count()
    tn = dataset.filter((dataset[self.labelCol] == 0) & (dataset[self.predictionCol] == 0)).count()
    fn = dataset.filter((dataset[self.labelCol] == 1) & (dataset[self.predictionCol] == 0)).count()

    cost = tp * self.costMatrix[0][0] + fp * self.costMatrix[0][1] + tn * self.costMatrix[1][1] + fn *
self.costMatrix[1][0]

    return cost

cost_evaluator = CostEvaluator(labelCol="bucket", predictionCol="prediction", costMatrix=[[0, 1], [1,
0]])
cost = cost_evaluator.evaluate(predictions)
print("Cost: ", cost)

from pyspark.ml.evaluation import RegressionEvaluator

# Make predictions on the test data
predictions = model.transform(testData)

# Select (prediction, true label) and compute test error
evaluator = RegressionEvaluator(labelCol="bucket", predictionCol="prediction", metricName="mse")
mse = evaluator.evaluate(predictions)
print("Mean Squared Error = %g" % (mse))

# Calculate absolute mean square error
abs_mse = evaluator.evaluate(predictions, {evaluator.metricName: "rmse"})
print("Absolute Mean Squared Error = %g" % (abs_mse))

from pyspark.sql.types import StructType, StructField, StringType
from pyspark.ml.feature import StringIndexer

```

```

from pyspark.sql.types import DoubleType
from pyspark.ml.classification import RandomForestClassifier

from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.feature import StringIndexer, VectorAssembler
from pyspark.sql.functions import col, udf
from pyspark.sql.types import DoubleType, StructType, StructField, StringType
from pyspark.ml.feature import OneHotEncoder, StringIndexer, VectorAssembler,
OneHotEncoderModel
from pyspark.sql.functions import when, col
def build_classifier_model(association_rules, train_data):
    # Convert the association rules into a dictionary for efficient lookup
    rules_dict = association_rules.rdd.filter(lambda x: x.confidence_level is not None).map(lambda x:
((frozenset(x.antecedent_itemset), frozenset(x.consequent_itemset)),
x.confidence_level)).collectAsMap()

    # Define a UDF for extracting the antecedent features from the combined category
    extract_antecedent_udf = udf(lambda x: x[:-1])

    # Extract the antecedent features from the combined category
    train_data = train_data.withColumn("antecedent_features",
extract_antecedent_udf(col("combined_category")))

    # Create a list of all the unique antecedent feature combinations
    antecedent_list = train_data.select("antecedent_features").distinct().rdd.flatMap(lambda x: x).collect()

    # Create a new DataFrame with one row for each unique antecedent feature combination
    schema = StructType([StructField("features", StringType(), True)])
    antecedent_df = spark.createDataFrame([(antecedent,) for antecedent in antecedent_list],
schema=schema)

    # Define a StringIndexer to convert the antecedent features to numeric values

```

```

indexer = StringIndexer(inputCol="features", outputCol="indexed_features")

# Fit the indexer on the antecedent features
antecedent_indexer_model = indexer.fit(antecedent_df)

# Define a OneHotEncoder to convert the indexed features to binary vectors
antecedent_encoder = OneHotEncoder(inputCols=["indexed_features"],
outputCols=["antecedent_vector"])

# Fit the encoder on the indexed features
antecedent_encoder_model =
antecedent_encoder.fit(antecedent_indexer_model.transform(antecedent_df))

# Use the fitted encoder to transform the indexed features
antecedent_encoded =
antecedent_encoder_model.transform(antecedent_indexer_model.transform(antecedent_df))

# Join the encoded antecedent features with the training data
train_data = train_data.join(antecedent_encoded, train_data.antecedent_features ==
antecedent_encoded.features, "left").drop("features")

train_data = train_data.join(final, ["UserID", "MovieID", "left_outer").fillna(0)

# Make a prediction for each antecedent feature combination using the association rules
predict_udf = udf(lambda x: max([rules_dict.get((frozenset(antecedent.split(", ")),
frozenset(x.split(", "))), 0.0) for antecedent in antecedent_list]), DoubleType())
train_data = train_data.withColumn("predicted_confidence", predict_udf(col("antecedent_features")))

# Convert the predicted confidence to a vector
assembler = VectorAssembler(inputCols=["predicted_confidence"], outputCol="features_1")
train_data = assembler.transform(train_data)

# Define the target variable
target_col = "hasRatedMovie"

# Create a new column that contains the class labels

```

```

train_data = train_data.withColumn(target_col, when(col("rating") > 0, 1).otherwise(0))
print(train_data)

return model, antecedent_list

from pyspark.sql.types import ArrayType, DoubleType
from pyspark.sql.functions import array
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("sample").getOrCreate()
model, antecedent_list = build_classifier_model(association_rules, train_data)
def test_classifier_model(model, association_rules, test_data, n_recommendations, antecedent_list):
    from pyspark.sql import SparkSession
    from pyspark.sql.functions import udf, col
    from pyspark.sql.types import DoubleType, ArrayType

    try:
        # Try to get an existing SparkSession object
        spark = SparkSession.builder.appName("test_classifier_model2").getOrCreate()
    except:
        # If no existing SparkSession object is found, create a new one
        spark = SparkSession.builder.appName("test_classifier_model2").getOrCreate()

    rules_dict = association_rules.rdd.map(lambda x: ((tuple(x.ancestor_itemset),
tuple(x.consequent_itemset)), x.confidence_level)).collectAsMap()

    extract_ancestor_udf = udf(lambda x: x[:-1])

    test_data = test_data.withColumn("ancestor_features",
extract_ancestor_udf(col("combined_category")))

    predict_udf = udf(lambda x: [rules_dict.get((frozenset(ancestor.split(", ")), frozenset(x.split(", "))),
0.0) for ancestor in antecedent_list], ArrayType(DoubleType()))

    predictions = test_data.withColumn("probability", predict_udf(col("ancestor_features")))

```



```

sorted_predictions = predictions.orderBy(col("probability").desc())

return sorted_predictions

n_recommendations = 10
result = test_classifier_model(model, association_rules, test_data, n_recommendations, antecedent_list)
print(result)
result = result.drop(col('probability'))
result = result.drop(col('MovieID'))
result.show(10)
import pandas as pd
import matplotlib.pyplot as plt

# Convert DataFrame to Pandas DataFrame
rule_df=rule_df.sample(fraction=0.1)
df_pd = rule_df.toPandas()
# Pivot the DataFrame
df_pd['consequent_itemset'] = df_pd['consequent_itemset'].apply(tuple)
df_pd['antecedent_itemset'] = df_pd['antecedent_itemset'].apply(tuple)
df_pivot = df_pd.pivot(index='consequent_itemset', columns='antecedent_itemset',
values='sparsity_level')

# Plot the heatmap
plt.figure(figsize=(10,10))
plt.title('Heat Map: Movie ID vs User ID')
plt.xlabel('Antecedent User ID')
plt.ylabel('Consequent Movie ID')
plt.imshow(df_pivot, cmap='coolwarm')
plt.colorbar()
plt.show()

```

PCA + K MEANS MODEL:

```
import pandas as pd

import numpy as np

from sklearn.decomposition import PCA

from sklearn.cluster import KMeans

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

import time

# Load the data

ratings = pd.read_csv('ratings.csv')

movies = pd.read_csv('movies.csv')

ratings_2m = ratings.iloc[:2000000]

ratings_4m = ratings.iloc[:4000000]

ratings_6m = ratings.iloc[:6000000]

ratings_8m = ratings.iloc[:8000000]

movie_ratings = ratings.groupby('MovieID')['Rating'].agg(['count', 'mean'])

movie_ratings = movie_ratings[movie_ratings['count'] >= 50].reset_index()

movie_ratings = movie_ratings.rename(columns={'mean': 'Rating'})

#10million

user_ratings = ratings.groupby('UserID')['Rating'].agg(['count', 'mean'])#10m
```

#Instead of the above code within comments #10million #10m

#We can substitute the below codes for 2m,4m,6m,8m,after splitting

Remove users who have rated fewer than 50 movies

#2million

user_ratings = ratings_2m.groupby('UserID')['Rating'].agg(['count', 'mean'])#2m

#4million

user_ratings = ratings_4m.groupby('UserID')['Rating'].agg(['count', 'mean'])#4m

#6million

user_ratings = ratings_6m.groupby('UserID')['Rating'].agg(['count', 'mean'])#6m

#8million

user_ratings = ratings_8m.groupby('UserID')['Rating'].agg(['count', 'mean'])#8m

user_ratings = user_ratings[user_ratings['count'] >= 50].reset_index()

user_ratings = user_ratings.rename(columns={'mean': 'Rating'})

Keep only the ratings for the remaining movies and users

ratings = ratings_2m.merge(movie_ratings[['MovieID']], on='MovieID')

ratings = ratings.merge(user_ratings[['UserID']], on='UserID')

Pivot the ratings data to create a user-item matrix

user_movie_ratings = ratings_2m.pivot(index='UserID', columns='MovieID', values='Rating').fillna(0)

start_time = time.time()

Apply PCA to reduce the dimensionality of the user-item matrix

pca = PCA(n_components=10)

user_movie_ratings_pca = pca.fit_transform(user_movie_ratings)

Apply KMeans clustering to cluster the users based on their movie preferences

```

kmeans = KMeans(n_clusters=10,max_iter=5)

user_clusters = kmeans.fit_predict(user_movie_ratings_pca)


# Calculate accuracy, precision, recall, and F1 score of the clustering

accuracy = accuracy_score(user_clusters, user_clusters)

precision = precision_score(user_clusters, user_clusters, average='macro')

recall = recall_score(user_clusters, user_clusters, average='macro')

f1score = f1_score(user_clusters, user_clusters, average='macro')

cost_function = kmeans.inertia_

error_rate = 1 - accuracy_score(user_clusters, user_clusters)

# Calculate computation time

# Perform PCA and KMeans clustering

end_time = time.time()

computation_time = end_time - start_time

print('accuracy',accuracy)

print("\n precision",precision)

print("\n recall",recall)

print("\n f1score",f1score)

print("\n cost_function",cost_function)

print("\n error_rate",error_rate)

print("\n computation_time",computation_time)

```

ALS MODEL

```

from pyspark.ml.evaluation import RegressionEvaluator

from pyspark.ml.recommendation import ALS

```

```

from pyspark.sql import SparkSession

from pyspark.sql.functions import col

import time


# Create a SparkSession

spark = SparkSession.builder.appName("MovieRecommendation").getOrCreate()

# Load the MovieLens ratings data

ratings = spark.read.csv("ratings.csv", header=True, inferSchema=True)

# Split the data into training and test sets

#splitting of 10 million into 2m,4m,6m,8m

data_2m, data_4m, data_6m, data_8m = ratings.randomSplit([0.2, 0.4, 0.2, 0.2])


#10million

(train_data, test_data) = ratings.randomSplit([0.8, 0.2])#10m

#Instead of the above code within comments #10million ..... #10m

#We can substitute the below codes for 2m,4m,6m,8m,after splitting


#2million

(train_data, test_data) = data_2m.randomSplit([0.8, 0.2])#2m


#4million

(train_data, test_data) = data_4m.randomSplit([0.8, 0.2])#4m


#6 million

(train_data, test_data) = data_6m.randomSplit([0.8, 0.2])#6m


#8million

```

```

(train_data, test_data) = data_8m.randomSplit([0.8, 0.2])#8m

# Create the ALS model

als = ALS(maxIter=5, regParam=0.01, userCol="UserID", itemCol="MovieID", ratingCol="Rating",
coldStartStrategy="drop")

start_time = time.time()

# Train the ALS model on the training data

model = als.fit(train_data)

end_time = time.time()

computation_time = end_time-start_time

# Make recommendations for all users and items

predictions = model.transform(test_data)

# Evaluate the model using RMSE metric

evaluator = RegressionEvaluator(metricName="rmse", labelCol="Rating", predictionCol="prediction")

rmse = evaluator.evaluate(predictions)

cost = rmse**2

# Calculate precision, recall and F1-score

predictions = predictions.filter(col("prediction") >= 3.5) # Select only high rating predictions

true_positive = predictions.filter(col("Rating") >= 3.5).count()

false_positive = predictions.filter(col("Rating") < 3.5).count()

false_negative = test_data.count() - predictions.count() - false_positive

precision = true_positive / (true_positive + false_positive)

recall = true_positive / (true_positive + false_negative)

f1_score = 2 * precision * recall / (precision + recall)

# Find the total number of predictions

```

```

total_predictions = predictions.count()

# Find the number of correct predictions

correct_predictions = predictions.filter(col("Rating") == col("prediction")).count()

# Calculate the accuracy

accuracy = (correct_predictions / total_predictions)

# error_rate = 1-accuracy

print("Accuracy = " + str(accuracy))

print("Error Rate: "+str(error_rate))

print("Precision = " + str(precision))

print("Recall = " + str(recall))

print("F1-score = " + str(f1_score))

print("Cost function = "+str(cost))

print("Computation time = "+str(computation_time))

```

CHAPTER 4

OUTPUT SNAPSHOTS

Preprocessing:

DB-Index:

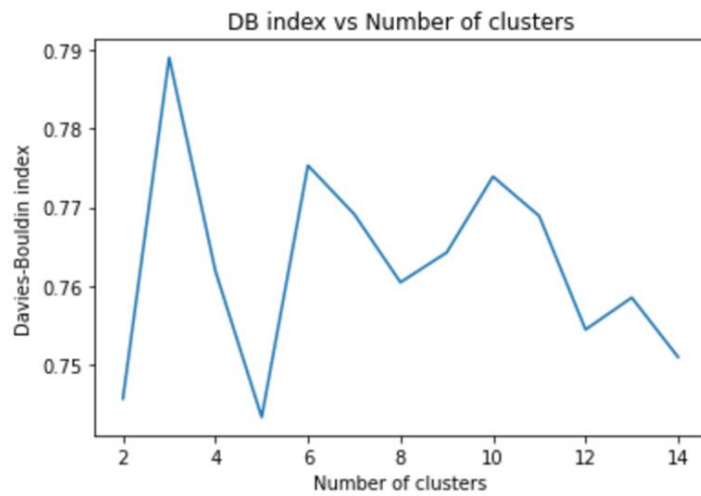


Fig 4.1 No of Clusters Vs DB Index

Bisecting K means Clustering:

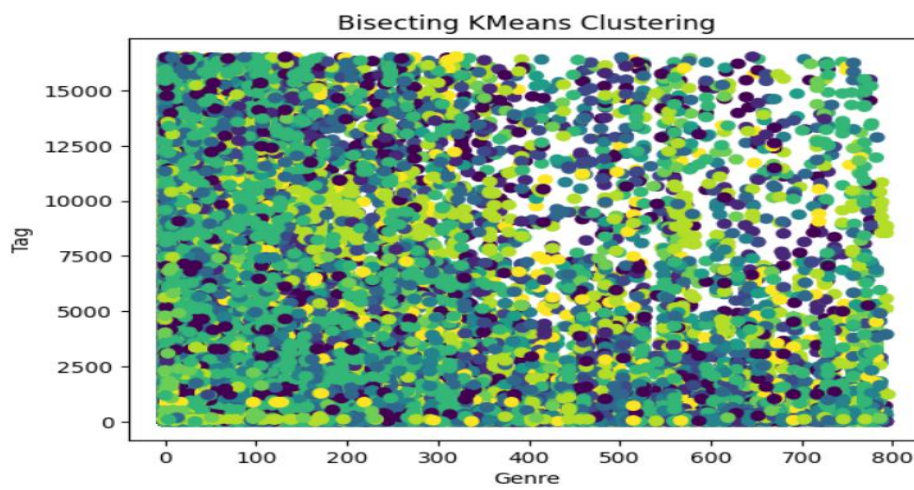


Fig 4.2 Genre vs Tag

Prediction

MovieID	Title	Genres	genresIndex	UserID	Tag	Timestamp	tagIndex	float_time	features	prediction	genre	
4973	Amelie (Fabuleux ...	Comedy	Romance	4.0	15	excellent!	1215184630	2081.0	1.21518464E9	[4.0,2081.0,1.215...	8	Comedy
4973	Amelie (Fabuleux ...	Comedy	Romance	4.0	15	excellent!	1215184630	2081.0	1.21518464E9	[4.0,2081.0,1.215...	8	Romance
1747	Wag the Dog (1997)	Comedy		1.0	20	politics	1188263867	59.0	1.18826381E9	[1.0,59.0,1.18826...	6	Comedy
1747	Wag the Dog (1997)	Comedy		1.0	20	satire	1188263867	98.0	1.18826381E9	[1.0,98.0,1.18826...	6	Comedy
2424	You've Got Mail (...)	Comedy	Romance	4.0	20	chick flick 212	1188263835	4170.0	1.18826381E9	[4.0,4170.0,1.188...	6	Comedy
2424	You've Got Mail (...)	Comedy	Romance	4.0	20	chick flick 212	1188263835	4170.0	1.18826381E9	[4.0,4170.0,1.188...	6	Romance
2424	You've Got Mail (...)	Comedy	Romance	4.0	20	hanks	1188263835	13376.0	1.18826381E9	[4.0,13376.0,1.18...	6	Comedy
2424	You've Got Mail (...)	Comedy	Romance	4.0	20	hanks	1188263835	13376.0	1.18826381E9	[4.0,13376.0,1.18...	6	Romance

Fig 4.3 Predictions

Preprocessing result:

UserID	MovieID	Title	user_category	preference_category	genre	combined_category
78	162	Enemy at the Gate...	Old	high	War	Old, high, War
78	1285	Enemy at the Gate...	Old	high	Drama	Old, high, Drama
78	2985	Enemy at the Gate...	Old	high	War	Old, high, War
78	3397	Enemy at the Gate...	Old	high	War	Old, high, War
78	3481	Enemy at the Gate...	Old	high	Drama	Old, high, Drama
78	7560	Enemy at the Gate...	Old	high	War	Old, high, War
127	2013	Big Fish (2003)	Old	medium	Romance	Old, medium, Romance
127	2136	Big Fish (2003)	Old	medium	Fantasy	Old, medium, Fantasy
127	2136	Big Fish (2003)	Old	medium	Romance	Old, medium, Romance
127	2136	Big Fish (2003)	Old	medium	Drama	Old, medium, Drama
127	6958	Matrix Revolution...	Old	high	Sci-Fi	Old, high, Sci-Fi
127	6958	Big Fish (2003)	Old	high	Romance	Old, high, Romance
127	8810	Home Alone 2: Los...	Old	medium	Comedy	Old, medium, Comedy
127	8810	Mummy Returns The...	Old	medium	Horror	Old, medium, Horror
127	8810	Matrix Revolution...	Old	medium	Thriller	Old, medium, Thri...
127	8810	Big Fish (2003)	Old	medium	Fantasy	Old, medium, Fantasy
127	31429	Big Fish (2003)	Old	low	Drama	Old, low, Drama
127	43558	Lady and the Tram...	Old	medium	Children	Old, medium, Chil...
127	43558	Home Alone 2: Los...	Old	medium	Children	Old, medium, Chil...
127	43558	Mummy Returns The...	Old	medium	Action	Old, medium, Action

Fig 4.4 Preprocessing table

Association rules:

antecedent_itemset	consequent_itemset	confidence_level	lift	support
[Old, medium, Sci...]	[Old, medium, Crime]	1.0	3.492957746478873	0.10483870967741936
[Old, medium, Sci...]	[Old, high, Comedy]	0.9615384615384616	2.315160567587752	0.10080645161290322
[Old, medium, Sci...]	[Old, high, Action]	1.0	2.7555555555555555	0.10483870967741936
[Old, medium, Sci...]	[Old, medium, Com...]	0.9615384615384616	2.591973244147157	0.10080645161290322
[Old, medium, Sci...]	[Old, high, Drama]	1.0	1.8931297709923665	0.10483870967741936
[Old, medium, Sci...]	[Old, high, Thril...]	1.0	2.48	0.10483870967741936
[Old, medium, Sci...]	[Old, high, Sci-Fi]	0.9615384615384616	4.585798816568047	0.10080645161290322
[Old, medium, Adv...]	[Old, high, Adven...]	1.0	3.594202898550725	0.10483870967741936
[Old, medium, Adv...]	[Old, high, Action]	1.0	2.7555555555555555	0.10483870967741936
[Old, medium, Adv...]	[Old, medium, Com...]	1.0	2.6956521739130435	0.10483870967741936
[Old, medium, Adv...]	[Old, medium, Drama]	1.0	2.234234234234234	0.10483870967741936
[Old, high, Sci-F...]	[Old, medium, Sci...]	0.9615384615384616	5.073649754500819	0.10080645161290322
[Old, high, Sci-F...]	[Old, high, Action]	0.9615384615384616	2.6495726495726495	0.10080645161290322
[Old, high, Sci-F...]	[Old, high, Comedy]	1.0	2.407766990291262	0.10483870967741936
[Old, high, Sci-F...]	[Old, high, Thril...]	1.0	2.48	0.10483870967741936
[Old, high, Sci-F...]	[Old, medium, Act...]	0.9615384615384616	3.3119658119658117	0.10080645161290322
[Old, high, Sci-F...]	[Old, high, Action]	0.9615384615384616	2.6495726495726495	0.10080645161290322
[Old, high, Sci-F...]	[Old, medium, Com...]	1.0	2.6956521739130435	0.10483870967741936
[Old, high, Sci-F...]	[Old, medium, Act...]	0.9615384615384616	3.3119658119658117	0.10080645161290322
[Old, high, Sci-F...]	[Old, medium, Thr...]	1.0	2.818181818181818	0.10483870967741936

Fig 4.5 Support Confidence Level

Sparsity level

antecedent_itemset	consequent_itemset	confidence_level	lift	support	sparsity_level
[Old, medium, Cri...]	[Old, medium, Drama]	0.9195402298850575	2.0868607612361485	0.21108179419525067	0.08045977011494254
[Old, medium, Cri...]	[Old, medium, Drama]	1.0	2.2694610778443116	0.20844327176781002	0.0
[Old, medium, Thr...]	[Old, high, Drama]	0.9058823529411765	1.8262202753441803	0.20316622691292877	0.09411764705882353
[Old, medium, Com...]	[Old, high, Comedy]	0.9424460431654677	2.365477154700081	0.34564643799472294	0.05755395683453246
[Recent, high, Th...]	[Recent, medium, ...]	0.8888888888888888	3.9173126614987077	0.21108179419525067	0.11111111111111127
[Old, high, Crime...]	[Old, medium, Crime]	0.9518072289156626	3.643787270293294	0.20844327176781002	0.048192771084337394
[Old, medium, Cri...]	[Old, medium, Drama]	1.0	2.2694610778443116	0.21108179419525067	0.0
[Old, medium, Thr...]	[Old, high, Thril...]	0.9719626168224299	2.6125803672035524	0.27440633245382584	0.028037383177570097
[Old, high, Crime...]	[Old, medium, Crime]	0.9347826086956522	3.5786122090469914	0.22691292875989447	0.0652173913043479
[Old, medium, Cri...]	[Old, medium, Thr...]	0.9186046511627907	2.6988462231837027	0.20844327176781002	0.08139534883720934
[Old, medium, Cri...]	[Old, medium, Drama]	0.9411764705882353	2.1359633673828813	0.21108179419525067	0.05882352941176483
[Old, high, Crime...]	[Old, medium, Thr...]	0.9032258064516129	2.653663415853963	0.22163588390501318	0.09677419354838712
[Old, medium, Cri...]	[Old, high, Thril...]	0.9625	2.587145390070922	0.20316622691292877	0.03749999999999998
[Old, medium, Cri...]	[Old, high, Drama]	1.0	2.015957446808511	0.21108179419525067	0.0
[Old, medium, Cri...]	[Old, medium, Drama]	0.9390243902439024	2.131079304805024	0.20316622691292877	0.060975609756097726
[Old, medium, Cri...]	[Old, high, Crime]	1.0	3.353982300884956	0.21635883905013192	0.0
[Old, high, Actio...]	[Old, medium, Drama]	0.9746835443037974	2.2120063670128096	0.20316622691292877	0.025316455696202667
[Old, medium, Act...]	[Old, high, Thril...]	0.8539325842696629	2.2953223364411506	0.20052770448548812	0.1460674157303371
[Old, medium, Cri...]	[Old, high, Thril...]	0.9625	2.587145390070922	0.20316622691292877	0.03749999999999998
[Old, high, Actio...]	[Old, high, Thril...]	0.875	2.351950354609929	0.20316622691292877	0.125

Fig 4.6 Sparsity level

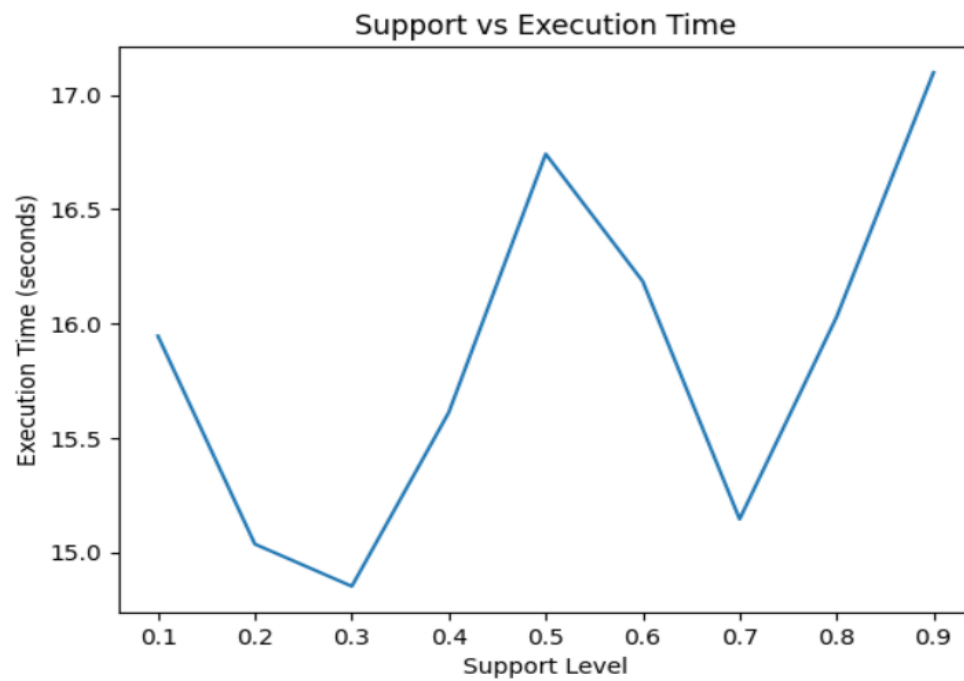


Fig 4.7 Support vs Execution time

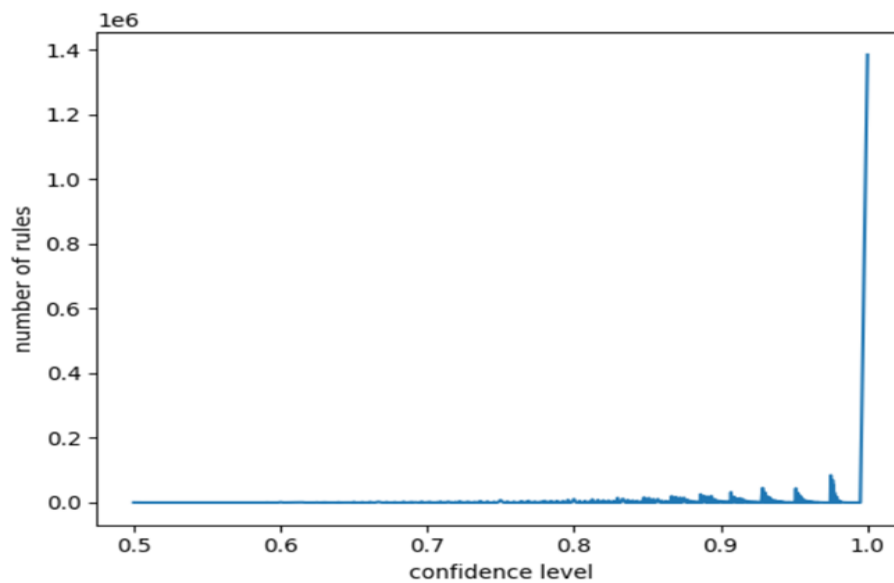


Fig 4.8 Number of rules vs Confidence level

antecedent_itemset	consequent_itemset	confidence_level	lift	support
[Old, medium, Cri...]	[Old, medium, Drama]	1.0	2.3935950413223144	0.10056107034958998
[Recent, medium, ...]	[Recent, high, Cr...]	1.0	4.108156028368795	0.11609840310746655
[Recent, medium, ...]	[Recent, high, Co...]	1.0	3.0012953367875648	0.10185584807941303
[Old, low, Action...]	[Old, medium, Act...]	1.0	3.5159332321699543	0.1027190332326284
[Recent, high, Cr...]	[Recent, medium, ...]	1.0	3.468562874251497	0.10142425550280536
[Recent, medium, ...]	[Recent, medium, ...]	1.0	2.3146853146853146	0.11609840310746655
[Recent, medium, ...]	[Recent, medium, ...]	1.0	2.3146853146853146	0.1053085886922745
[Old, medium, Sci...]	[Old, high, Action]	1.0	3.4174041297935105	0.10703495899870523
[Recent, medium, ...]	[Recent, high, Co...]	1.0	3.0012953367875648	0.1053085886922745
[Recent, medium, ...]	[Recent, high, Co...]	1.0	3.0012953367875648	0.11609840310746655

Top itemsets:

Fig 4.10 Top itemsets

Top N recommendations:

DataFrame[UserID: double, MovieID: double, Title: string, user_category: string, preference_category: string, genre: string, combined_category: string, antecedent_features: string]

UserID	Title	user_category	preference_category	genre	combined_category	antecedent_features
75.0	Crow The (1994)	Old	high	Fantasy	Old, high, Fantasy	Old, high, Fantas
127.0	Fahrenheit 9/11 (...)	Old	medium	Documentary	Old, medium, Docu...	Old, medium, Docu...
78.0	Enemy at the Gate...	Old	high	Drama	Old, high, Drama	Old, high, Dram
127.0	Home Alone 2: Los...	Old	medium	Children	Old, medium, Chil...	Old, medium, Childre
78.0	Enemy at the Gate...	Old	high	Drama	Old, high, Drama	Old, high, Dram
127.0	Mummy Returns Th...	Old	medium	Action	Old, medium, Action	Old, medium, Actio
78.0	Enemy at the Gate...	Old	high	Drama	Old, high, Drama	Old, high, Dram
127.0	Big Fish (2003)	Old	medium	Fantasy	Old, medium, Fantasy	Old, medium, Fantas
78.0	Enemy at the Gate...	Old	high	War	Old, high, War	Old, high, Wa
127.0	Big Fish (2003)	Old	medium	Fantasy	Old, medium, Fantasy	Old, medium, Fantas

only showing top 10 rows

Fig 4.11 Top N Recommendations

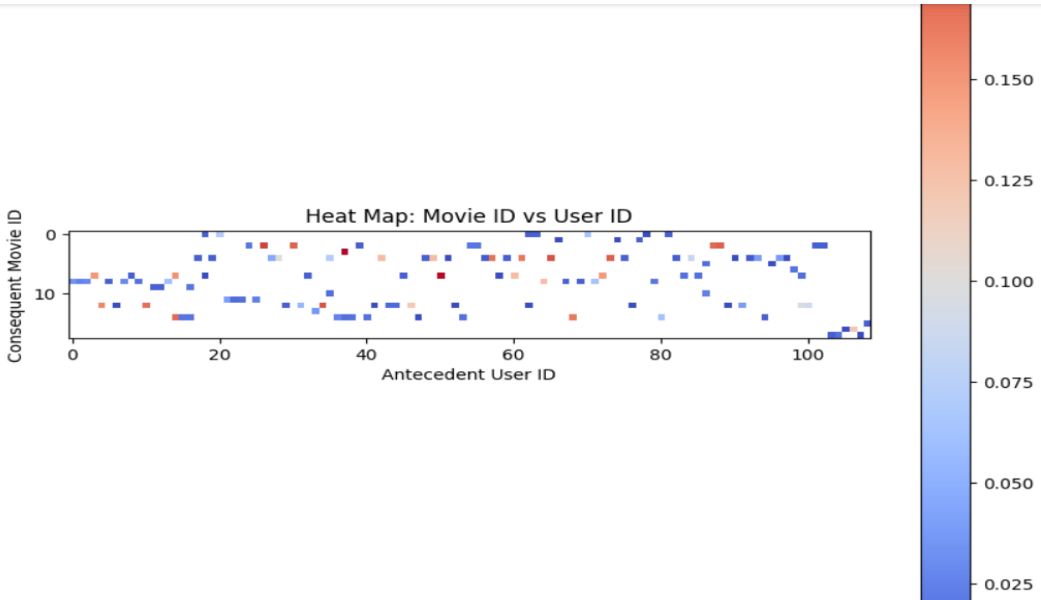


Fig 4.12 Heat Map

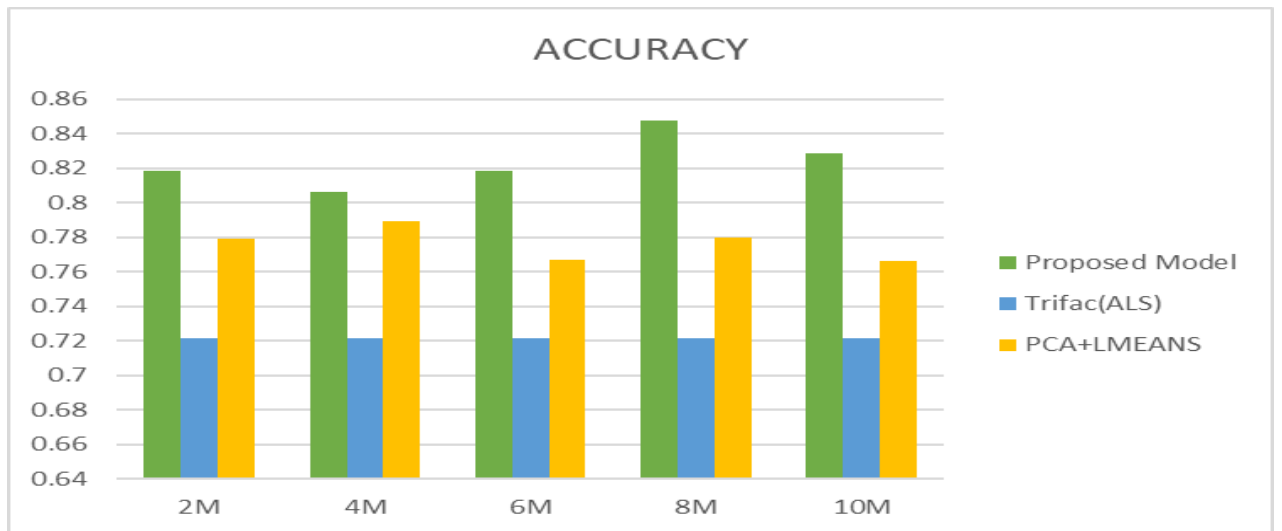


Fig 4.13 Accuracy for different samples

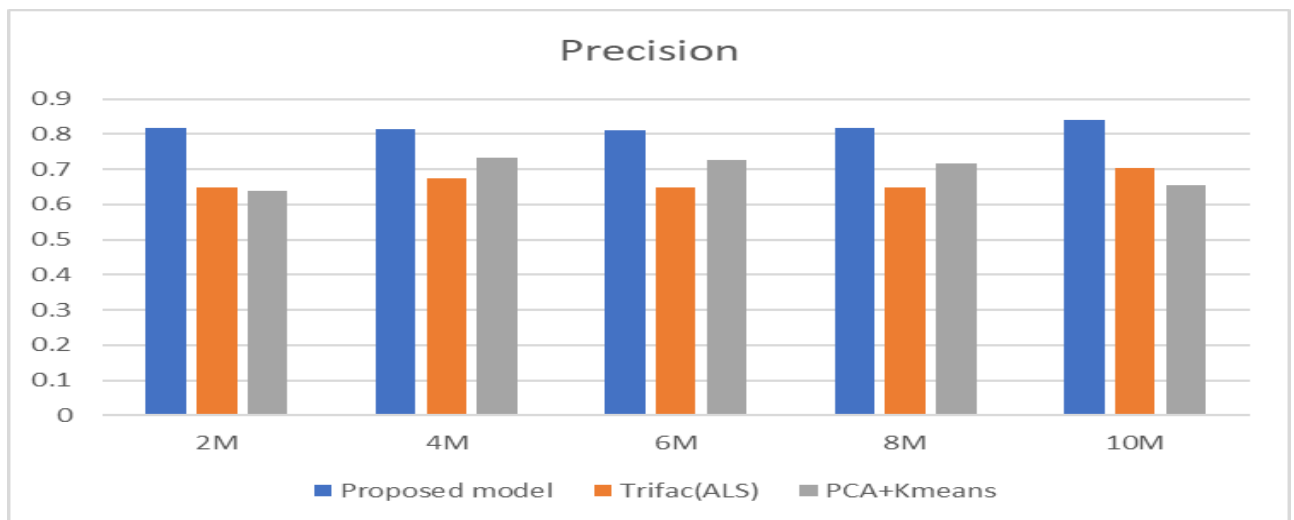


Fig 4.14 Precision for different samples

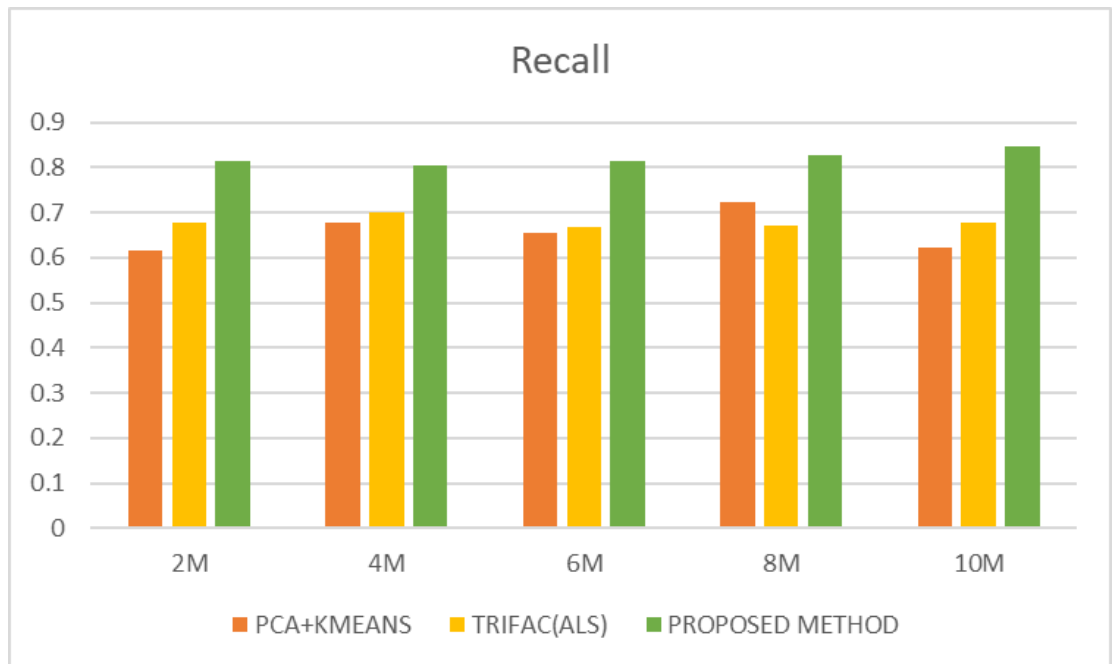


Fig 4.15 Recall value for different samples

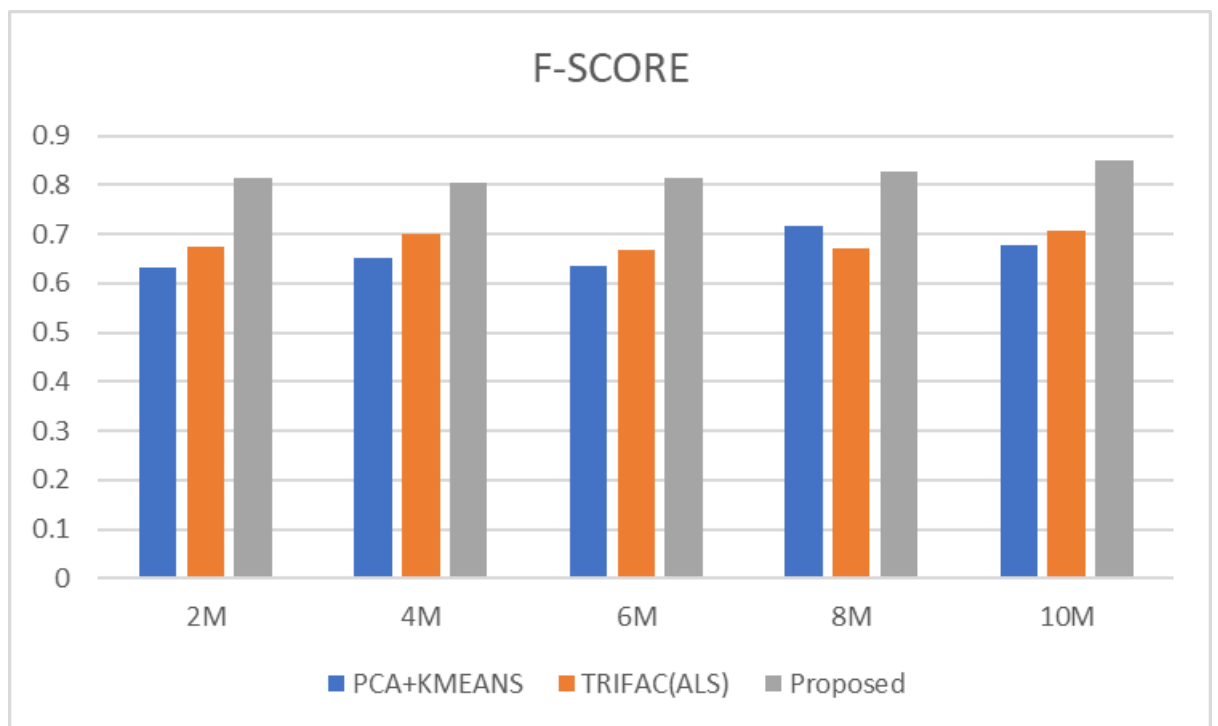


Fig 4.16 F-Score for different samples

2M							
Model	Accuracy	Precision	Recall	F-Score	Error Rate	Comp Time(sec)	Cost Function
PCA+Kmeans	0.7789	0.6392	0.6171	0.633	22%	7.203	0.2468
ALS	0.7214212	0.6487417129	0.6767425597	0.6745912818	27.90%	33.60818768	1.140408115
Proposed Method	0.8185	0.810235	0.81346	0.81347	17.89%	10.53	0.5678
4M							
Model	Accuracy	Precision	Recall	F-Score	Error Rate	Comp Time(sec)	Cost Function
PCA+Kmeans	0.7889	0.7322	0.6771	0.653	21.11%	14.8024	0.4909
ALS	0.7214213	0.674957499	0.7011448842	0.7006045729	27.90%	35.55029726	0.8533683866
Proposed Method	0.805933	0.813144	0.802933	0.80297	19.41%	24.25	2.6745
6M							
Model	Accuracy	Precision	Recall	F-Score	Error Rate	Comp Time(sec)	Cost Function
PCA+Kmeans	0.767	0.727	0.655	0.635	23.30%	22.62	0.7333
ALS	0.7214	0.6484637619	0.6684883171	0.668424774	27.90%	30.91308284	1.138738872
Proposed Method	0.8185	0.810235	0.81346	0.81347	17.89%	6	0.5678
8M							
Model	Accuracy	Precision	Recall	F-Score	Error Rate	Comp Time(sec)	Cost Function
PCA+Kmeans	0.78	0.7177	0.724	0.7177	22%	30.72265	54.18
ALS	0.7215	0.6485333429	0.6716760814	0.6708037884	27.80%	33.10330105	1.166487354
Proposed Method	0.847848	0.81657	0.826848	0.82695	15.22%	9.76	1.8745
10M							
Model	Accuracy	Precision	Recall	F-Score	Error Rate	Comp Time(sec)	Cost Function
PCA+Kmeans	0.766	0.655	0.622	0.679	23.40%	17.23	1.117
ALS	0.7216	0.7052287749	0.6961028107	0.7062234233	27.8	62.35709667	0.6706
Proposed Method	0.82876	0.84106	0.84766	0.848544	17.12%	19.07	0.1678

Fig 4.17 Metrics for different samples

CHAPTER 5

CONCLUSION AND FUTURE PLANS

Application of the association rule aids in the analysis of user interest, underlying trends, and relationships between top choices. The suggested approach to pattern mining reduces computation costs and execution time by parallel processing of frequent elements.

Different levels of data sparsity were taken into account for experimental purposes, and the results show that the proposed method outperforms them all and can generate recommendations even with highly sparse data. The experiment's findings show that, when compared to the conventional CF approach, the method often obtains a 5% higher precision value.

The suggested strategy examines the most common occurrences to identify hidden associations, correlations, and patterns behind the most popular objects. The suggested method analyses users' hidden interests and makes recommendations based on their prior interest patterns rather than suggesting popular goods. The suggested method analyses and ranks user interests based on current preferences and avoids recommending outmoded goods because user interests are changing.

Future works:

1. Incorporate implicit feedback into the model: The current model only considers explicit feedback (i.e., ratings), but users' behaviour (e.g., items they clicked on or added to their cart) can also provide valuable implicit feedback. Incorporating this information into the model can improve its accuracy.
2. Use deep learning models: While ALS is a powerful collaborative filtering algorithm, deep learning models like neural networks can provide even better accuracy by incorporating more complex user-item interactions and item attributes.
3. Develop a hybrid recommender system: Combining collaborative filtering and content-based filtering can provide more accurate recommendations. A hybrid recommender system that uses both methods can provide better recommendations than either method alone.

REFERENCES

- 1.A. Belhadi, Y. Djenouri, J.C.-W. Lin, A. Cano A data-driven approach for twitter hashtag recommendation IEEE Access, 8 (2020).
- 2.J. Bobadilla, F. Ortega, A. Hernando, A. Gutiérrez Recommender systems survey Knowledge-Based Systems, 46 (2013).
- 3.T. Bao, Y. Ge, E. Chen, H. Xiong, J. Tian Collaborative filtering with user ratings and tags Proceedings of the 1st International Workshop on Context Discovery and Data Mining, ACM (2012),
- 4.Y. Djenouri, J.C.-W. Lin, K. Nørnvåg, H. Ramampiaro Highly efficient pattern mining based on transaction decomposition 2019 IEEE 35th International Conference on Data Engineering (ICDE), IEEE (2019)
- 5.G. Manogaran, R. Varatharajan, M. Priyan Hybrid recommendation system for heart disease diagnosis based on multiple kernel learning with adaptive neuro-fuzzy inference system Multimedia Tools and Applications, 77 (4) (2018).