Big Data Management System

GROUP 2, BATCH A

Group Members

01

R Sriviswa

CB.EN.U4AIE21046

02

Aman Sirohi

CB.EN.U4AIE21003

03

Vikhyat Bansal

CB.EN.U4AIE21076

04

Rakhil ML

CB.EN.U4AIE21048

01

Introduction

ABOUT THE DATASET

The datasets describe ratings and free-text tagging activities from MovieLens, a movie recommendation service. It contains 200002 ratings and 465564 tag applications across 27278 movies. These data were created by 1384 users.

Users were selected at random for inclusion. All selected users had rated at least 20 movies.

The dataset contains 6 files.

- genome_scores.csv: Contains movie-tag relevance data.
    a. **movieId: Particular ID given to each movie**
    b. **tagId: ID of a tag**
    c. **relevance: How much is the tag relevant to movie**

movieId	tagId	relevance
1	1	0.025
1	2	0.025
1	3	0.05775
1	4	0.09675
1	5	0.14675
1	6	0.217
1	7	0.067
1	8	0.26275
1	9	0.262

ABOUT THE DATASET

2. genome_tags.csv that contains tag descriptions:

- **tagId: ID for a tag**
- **tag: Tags used to describe a movie**

tagId	tag
1	7
2	007 (series)
3	18th century
4	1920s
5	1930s
6	1950s
7	1960s
8	1970s
9	1980s
10	19th century

3. link.csv that contains identifiers that can be used to link to other sources:

- **movieId: ID for the movie**
- **imdbId: IMDB id for the movie**
- **tmbdId: The Movie DB ID for a movie**

movieId	imdbId	tmdbId
1	114709	862
2	113497	8844
3	113228	15602
4	114885	31357
5	113041	11862
6	113277	949
7	114319	11860
8	112302	45325
9	114576	9091
10	113189	710

ABOUT THE DATASET

4. movie.csv that contains movie information:

movieId	title	genres
1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
2	Jumanji (1995)	Adventure|Children|Fantasy
3	Grumpier Old Men (1995)	Comedy|Romance
4	Waiting to Exhale (1995)	Comedy|Drama|Romance
5	Father of the Bride Part II (1995)	Comedy
6	Heat (1995)	Action|Crime|Thriller
7	Sabrina (1995)	Comedy|Romance
8	Tom and Huck (1995)	Adventure|Children
9	Sudden Death (1995)	Action
10	GoldenEye (1995)	Action|Adventure|Thriller

- **movieId: ID for a movie**
- **title: Name of the movie**
- **genres: Genres associated with the movie**

5. **rating.csv** that contains ratings of movies by users:

- **userId: ID of the user**
- **movieId: ID of the movie**
- **rating: rating given by the user to each movie**
- **timestamp: Time of review in UNIX time format**

userId	movieId	rating	timestamp
1	2	3.5	1112486027
1	29	3.5	1112484676
1	32	3.5	1112484819
1	47	3.5	1112484727
1	50	3.5	1112484580
1	112	3.5	1094785740
1	151	4	1094785734
1	223	4	1112485573
1	253	4	1112484940

ABOUT THE DATASET

6. **tag.csv** that contains tags applied to movies by users:

- **userId: ID of the user**
- **movieId: ID of the movie**
- **tag: tag given by the user to a movie**
- **timestamp: time in UNIX format**

userId	movieId	tag
18	4141	Mark Waters
65	208	dark hero
65	353	dark hero
65	521	noir thriller
65	592	dark hero
65	668	bollywood
65	898	screwball comedy
65	1248	noir thriller
65	1391	mars

READING THE DATASET

The data was stored in MySQL and then using a JDBC connector, we connected it to Spark for Analysis.

```
val moviesDF = spark.read.format("jdbc")
   option(
            url="jdbc:mysql://localhost/moviedb",
            driver="com.mysql.jdbc.Driver",
            dbtable="movies",
            user="root"
            password="<password>")
```

```scala
// Define MySQL connection properties
val jdbcHostname = "localhost"
val jdbcPort = "3306"
val jdbcDatabase = "moviedb"
val jdbcUsername = "root"
val jdbcPassword = "viswa@123"

// Set up the JDBC URL for MySQL
val jdbcUrl = s"jdbc:mysql://${jdbcHostname}:${jdbcPort}/${jdbcDatabase}"

// Read data from MySQL into a DataFrame
val moviesDF= spark.read
  .format( source = "jdbc")
  .option("url", jdbcUrl)
  .option("dbtable", "movies")
  .option("user", jdbcUsername)
  .option("password", jdbcPassword)
  .load()
```

Schema

```
moviesDF
root
 |-- movieId: integer (nullable = true)
 |-- title: string (nullable = true)
 |-- genres: string (nullable = true)
tagsDF
root
 |-- userId: integer (nullable = true)
 |-- movieId: integer (nullable = true)
 |-- tag: string (nullable = true)
 |-- timestamp: integer (nullable = true)
genomeTagsDF
root
 |-- tagId: integer (nullable = true)
 |-- tag: string (nullable = true)
```

```
genomeScoresDF
root
 |-- movieId: integer (nullable = true)
 |-- tagId: integer (nullable = true)
 |-- relevance: double (nullable = true)
ratingsDF
root
 |-- userId: integer (nullable = true)
 |-- movieId: integer (nullable = true)
 |-- rating: double (nullable = true)
 |-- timestamp: integer (nullable = true)
linkDF
root
 |-- movieId: integer (nullable = true)
 |-- imdbId: integer (nullable = true)
 |-- tmdbId: integer (nullable = true)
```

RATINGS ANALYSIS

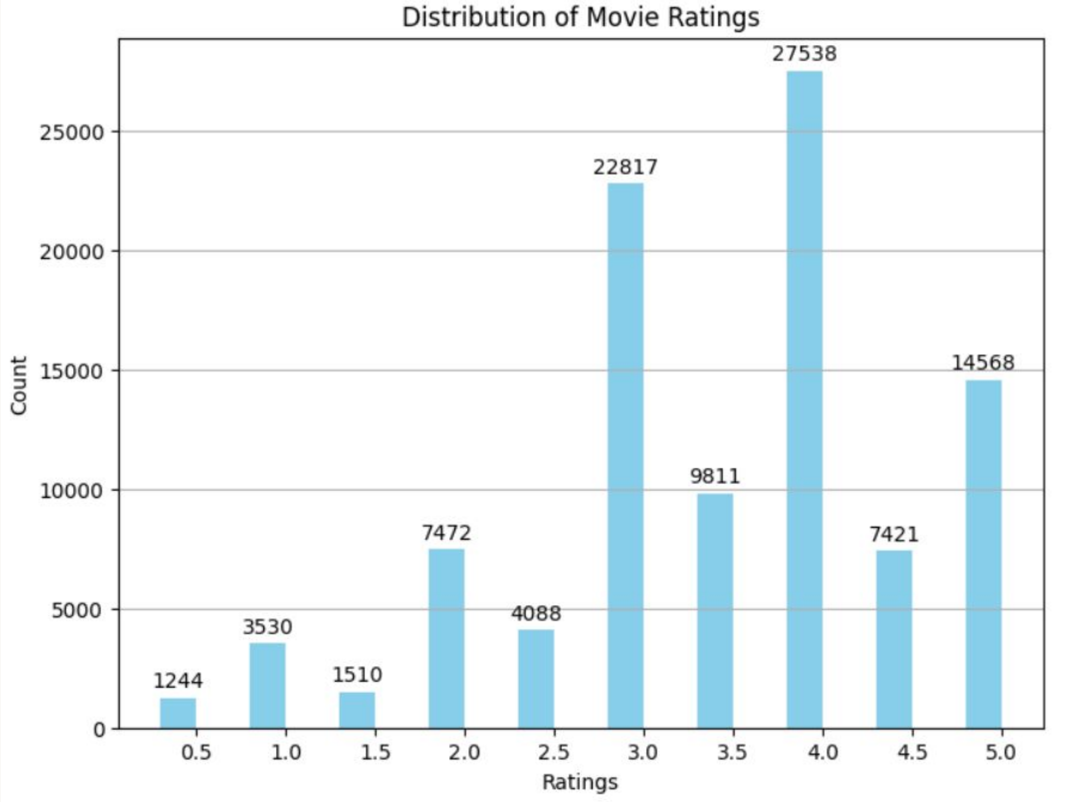
```
val ratingCounts = ratingsDF
 .groupBy("rating")
 .agg(count("rating").alias("count"))
 .sort("rating")

ratingCounts.show()
```

It helps us to understand how users tend to rate movies.

We can see that the number of 3-5 stars rating is high
Compared to the others, this suggests us that users tend
To rate movies more positively.

```
+------+-----+
|rating|count|
+------+-----+
|   0.5| 1244|
|   1.0| 3530|
|   1.5| 1510|
|   2.0| 7472|
|   2.5| 4088|
|   3.0|22817|
|   3.5| 9811|
|   4.0|27538|
|   4.5| 7421|
|   5.0|14568|
+------+-----+
```

PLOT


Distribution of Movie Ratings
Count
25000
20000
15000
10000
5000
0
1244
3530
1510
7472
4088
22817
9811
27538
7421
14568
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Ratings

MOVIES WITH HIGHEST NUMBER OF RATINGS

```
val moviePopularityByRatings = ratingsDF
 .groupBy("movieId")
 .agg(count("rating").alias("numRatings"))
 .sort(desc("numRatings"))

// Join with moviesDF to get movie details
val popularMoviesWithDetails = moviePopularityByRatings
 .join(moviesDF, Seq("movieId"))
 .select("movieId", "title", "numRatings","genres")

val top20PopularMovies = popularMoviesWithDetails
 .sort(desc("numRatings"))
 .limit(20)

top20PopularMovies.show()
```

MOVIES WITH HIGHEST NUMBER OF RATINGS

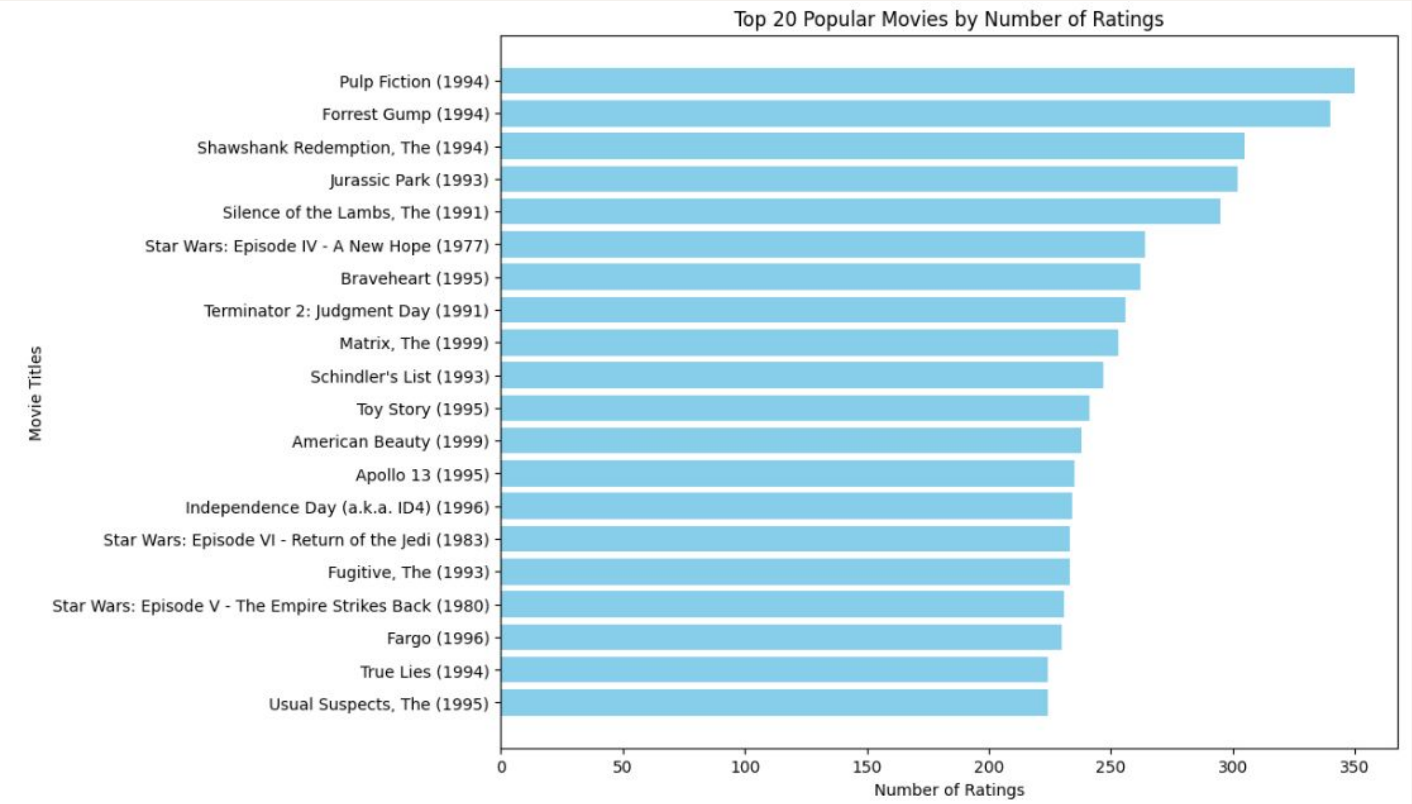
movieId	title	numRatings	genres
296	Pulp Fiction (1994)	350	Comedy|Crime|Dram...
356	Forrest Gump (1994)	340	Comedy|Drama|Roma...
318	Shawshank Redempt...	305	Crime|Drama
480	Jurassic Park (1993)	302	Action|Adventure|...
593	Silence of the La...	295	Crime|Horror|Thri...
260	Star Wars: Episod...	264	Action|Adventure|...
110	Braveheart (1995)	262	Action|Drama|War
589	Terminator 2: Jud...	256	Action|Sci-Fi
2571	Matrix, The (1999)	253	Action|Sci-Fi|Thr...
527	Schindler's List ...	247	Drama|War
1	Toy Story (1995)	241	Adventure|Animati...
2858	American Beauty (...	238	Comedy|Drama
150	Apollo 13 (1995)	235	Adventure|Drama|IMAX
780	Independence Day ...	234	Action|Adventure|...
1210	Star Wars: Episod...	233	Action|Adventure|...
457	Fugitive, The (1993)	233	Thriller
1196	Star Wars: Episod...	231	Action|Adventure|...
608	Fargo (1996)	230	Comedy|Crime|Dram...
380	True Lies (1994)	224	Action|Adventure|...
50	Usual Suspects, T...	224	Crime|Mystery|Thr...

This analysis helps us to identify the movies that have attracted the most attention/engagement from the viewers.

It provides insight into what types of movies tend to attract more ratings or attention from users. This might suggest genres, actors, directors, or specific themes that are popular among viewers.

It could also indicate how users utilize the platform. Do they tend to rate only popular or well-known movies, or do they rate a diverse range of films?

PLOT



Top 20 Popular Movies by Number of Ratings
Pulp Fiction (1994)
Forrest Gump (1994)
Shawshank Redemption, The (1994)
Jurassic Park (1993)
Silence of the Lambs, The (1991)
Star Wars: Episode IV - A New Hope (1977)
Braveheart (1995)
Terminator 2: Judgment Day (1991)
Matrix, The (1999)
Schindler's List (1993)
Toy Story (1995)
American Beauty (1999)
Apollo 13 (1995)
Independence Day (a.k.a. ID4) (1996)
Star Wars: Episode VI - Return of the Jedi (1983)
Fugitive, The (1993)
Star Wars: Episode V - The Empire Strikes Back (1980)
Fargo (1996)
True Lies (1994)
Usual Suspects, The (1995)
Movie Titles
0
50
100
150
200
250
300
350
Number of Ratings

USER ACTIVITY

```
val userActivity = ratingsDF
 .groupBy("userId")
 .agg(count("rating").alias("numRatings"))
 .sort(desc("numRatings"))
```

```
userActivity.show()
```

```
val threshold = 100
```

```
val activeUsers = userActivity.filter( s"numRatings > $threshold")
activeUsers.show()
```

USER ACTIVITY

userId	numRatings
156	2179
586	1431
572	1326
359	1300
208	1288
394	1212
298	1127
116	1110
632	1094
614	1042
104	998
424	918
648	904
587	873
348	786
347	778
637	758
367	739
388	737
54	710

The analysis helps identify a subset of users who are particularly engaged or active in the platform.

These active users are likely to have a more significant impact on any analysis due to their extensive rating history.

DISTRIBUTION OF USER RATINGS

```
val ratingStats = userActivity
 .select("numRatings")
 .summary("min", "25%", "50%", "75%", "max")

ratingStats.show()
```

```
+-------+----------+
|summary|numRatings|
+-------+----------+
|    min|        20|
|    25%|        35|
|    50%|        70|
|    75%|       158|
|    max|      2179|
+-------+----------+
```

GENRE BASED ANALYSIS

```scala
val moviesWithGenres = moviesDF
 .withColumn("genre", explode(split(col("genres"), "\\|")))

val ratingsWithGenres = ratingsDF
 .join(moviesWithGenres, Seq("movieId"))
 .select("userId", "rating", "genre")

// Show the resulting DataFrame
ratingsWithGenres.show()
```

```
+------+------+---------+
|userId|rating|    genre|
+------+------+---------+
|     1|   3.5|  Fantasy|
|     1|   3.5| Children|
|     1|   3.5|Adventure|
|     1|   3.5|   Sci-Fi|
|     1|   3.5|  Mystery|
|     1|   3.5|  Fantasy|
|     1|   3.5|    Drama|
|     1|   3.5|Adventure|
|     1|   3.5| Thriller|
|     1|   3.5|   Sci-Fi|
```

It allows for analysis to understand how users rate movies across different genres

It provides insights into user preferences for specific genres. For instance, do users tend to rate certain genres higher than others?

AVERAGE RATING BY GENRE

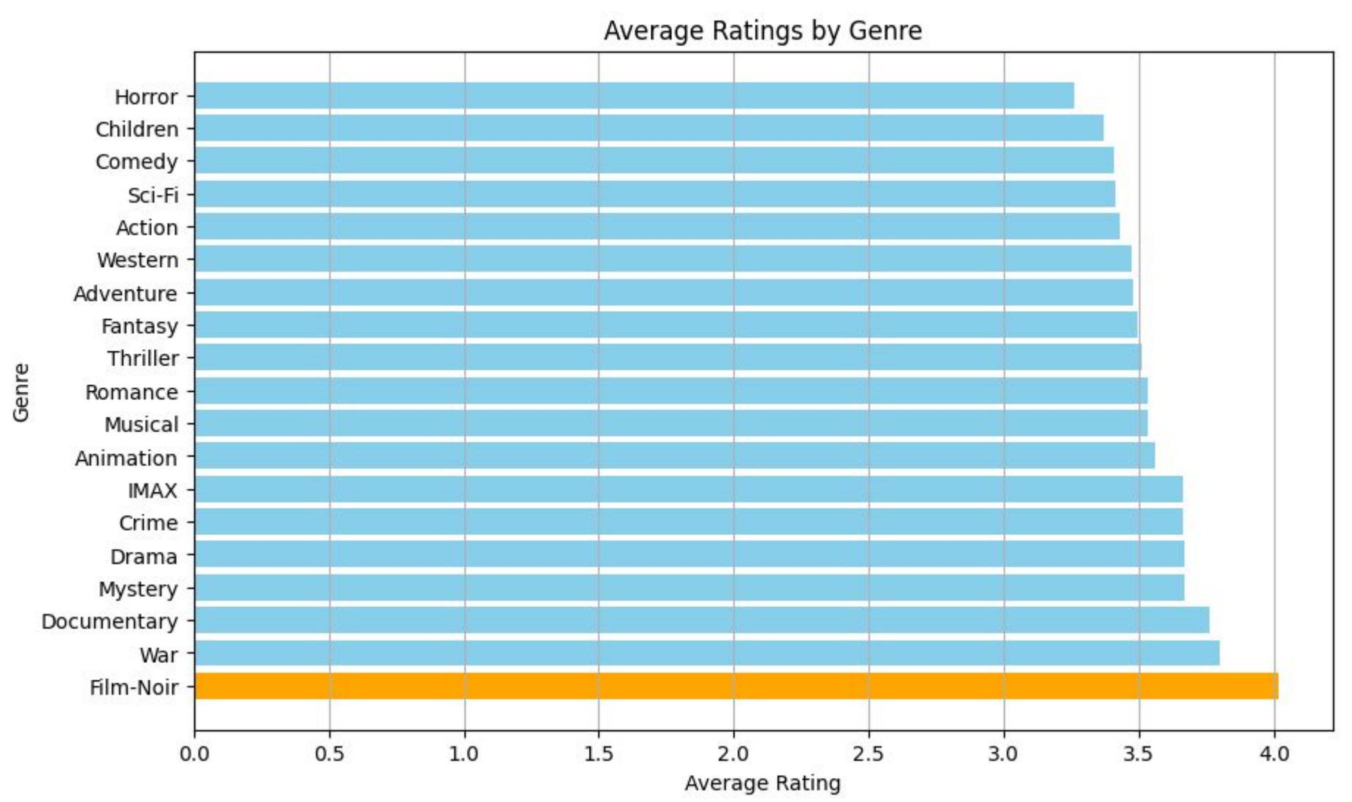
```
val avgRatingByGenre = ratingsWithGenres
  .groupBy("genre")
  .agg(avg("rating").alias("avgRating"))
  .sort(desc("avgRating"))

avgRatingByGenre.show()
```

It helps us to what kind of movies are highly liked by the audience.

Allows for comparisons between genres to determine which genres are generally better received or preferred by users

```
+-----------+------------------+
|      genre|         avgRating|
+-----------+------------------+
|  Film-Noir| 4.016393442622951|
|        War| 3.799306625577812|
|Documentary|3.7591389114541025|
|    Mystery|3.6695906432748537|
|      Drama| 3.667434911516724|
|      Crime| 3.665608432992233|
|       IMAX| 3.662633305988515|
|  Animation| 3.560841881853555|
|    Musical|3.5345509539320616|
|    Romance|3.5328068043742404|
|   Thriller|3.5096385542168673|
|    Fantasy|3.4921072295908835|
|  Adventure|3.4764559256741445|
|    Western|3.4753086419753085|
|     Action| 3.429536442432537|
|     Sci-Fi|3.4119467657072113|
|     Comedy|3.4091980205796872|
|   Children| 3.366832976954146|
|     Horror|3.2613436272517946|
+-----------+------------------+
```

PLOT



Average Ratings by Genre
Genre
Horror
Children
Comedy
Sci-Fi
Action
Western
Adventure
Fantasy
Thriller
Romance
Musical
Animation
IMAX
Crime
Drama
Mystery
Documentary
War
Film-Noir
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Average Rating

RATINGS PER GENRE

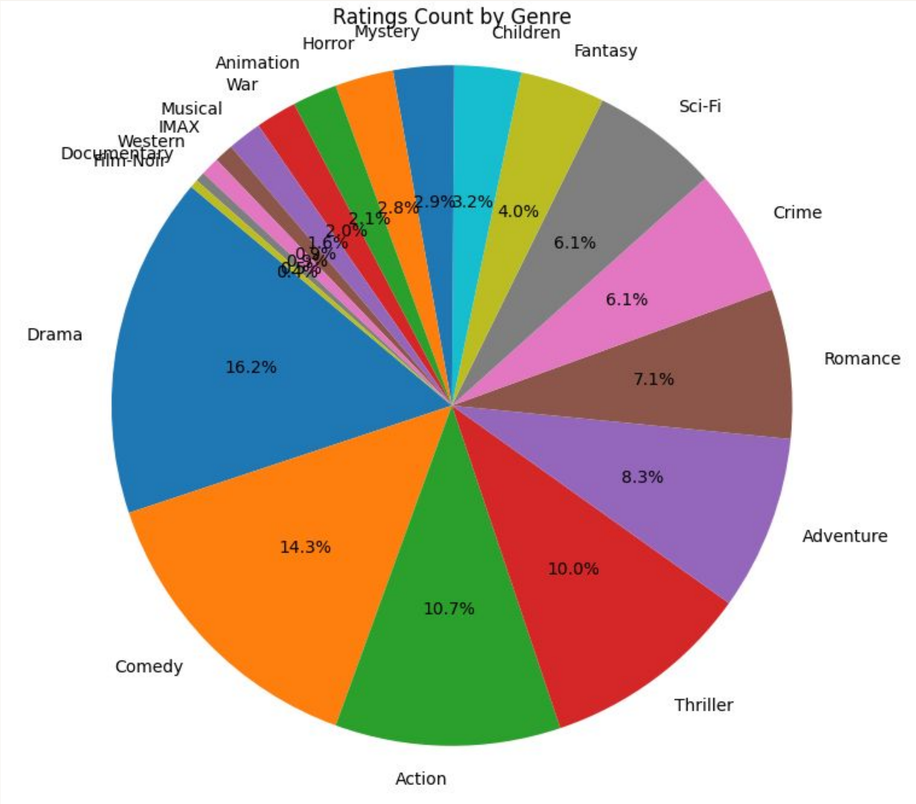
```
val ratingsCountByGenre = ratingsWithGenres
 .groupBy("genre")
 .agg(count("rating").alias("numRatings"))
 .sort(desc("numRatings"))
```

```
ratingsCountByGenre.show()
```

```
Helps us to understand the type of movies that is
Preferred by the users
```

```
+-----------+----------+
|      genre|numRatings|
+-----------+----------+
|      Drama|     43172|
|     Comedy|     38193|
|     Action|     28497|
|   Thriller|     26560|
|  Adventure|     22065|
|    Romance|     18929|
|      Crime|     16222|
|     Sci-Fi|     16155|
|    Fantasy|     10706|
|   Children|      8418|
|    Mystery|      7695|
|     Horror|      7383|
|  Animation|      5654|
|        War|      5192|
|    Musical|      4298|
|       IMAX|      2438|
|    Western|      2349|
|Documentary|      1231|
|  Film-Noir|      1037|
+-----------+----------+
```

PLOT


Ratings Count by Genre
Mystery
Horror
Animation
War
Musical
IMAX
Western
Documentary
Film-Noir
Children
Fantasy
Sci-Fi
Crime
Romance
Adventure
Thriller
Action
Comedy
Drama
2.9%
2.8%
2.1%
2.0%
1.6%
3.2%
4.0%
6.1%
6.1%
7.1%
8.3%
10.0%
10.7%
14.3%
16.2%

GENRE TRENDS OVER TIME

```
val moviesWithGenres = moviesDF
  .withColumn("genre", explode(split(col("genres"), "\\|")))

val ratingsWithGenres = ratingsDF
  .join(moviesWithGenres, Seq("movieId"), "inner")
  .select("userId", "rating", "genre", "timestamp")

val ratingsWithYear = ratingsWithGenres
  .withColumn("year", year(from_unixtime(col("timestamp"))))

val genreTrends = ratingsWithYear
  .groupBy("genre", "year")
  .agg(count("rating").alias("numRatings"))
  .sort("genre", "year")
```

GENRE TRENDS OVER TIME

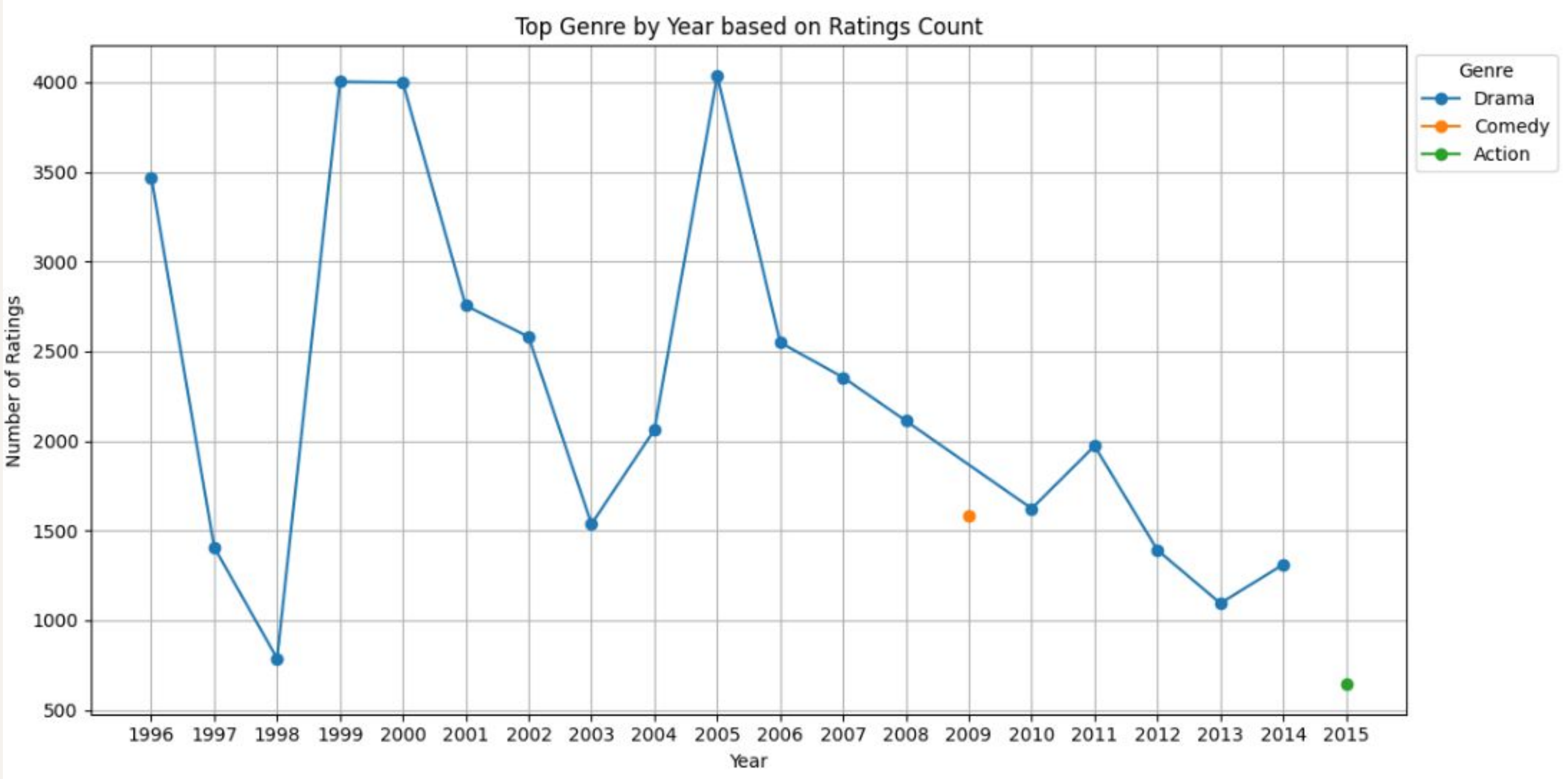
```
val windowSpec =
Window.partitionBy("year").orderBy(desc("numRatings"))

val rankedGenres = genreTrends.withColumn( "rank",
row_number().over(windowSpec))

val topGenreByYear = rankedGenres.filter( col("rank") ===
1).select("year", "genre", "numRatings")

topGenreByYear.show()
```

GENRE TRENDS OVER TIME

Helps us to understand how the preferences of audience has changed over time

year	genre	numRatings
1996	Drama	3471
1997	Drama	1408
1998	Drama	788
1999	Drama	4004
2000	Drama	3999
2001	Drama	2755
2002	Drama	2581
2003	Drama	1537
2004	Drama	2062
2005	Drama	4035
2006	Drama	2549
2007	Drama	2355
2008	Drama	2112
2009	Comedy	1580
2010	Drama	1623
2011	Drama	1972
2012	Drama	1391
2013	Drama	1095
2014	Drama	1311
2015	Action	646

PLOT


Top Genre by Year based on Ratings Count
Genre
Drama
Comedy
Action
Number of Ratings
4000
3500
3000
2500
2000
1500
1000
500
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
Year

TEMPORAL TRENDS

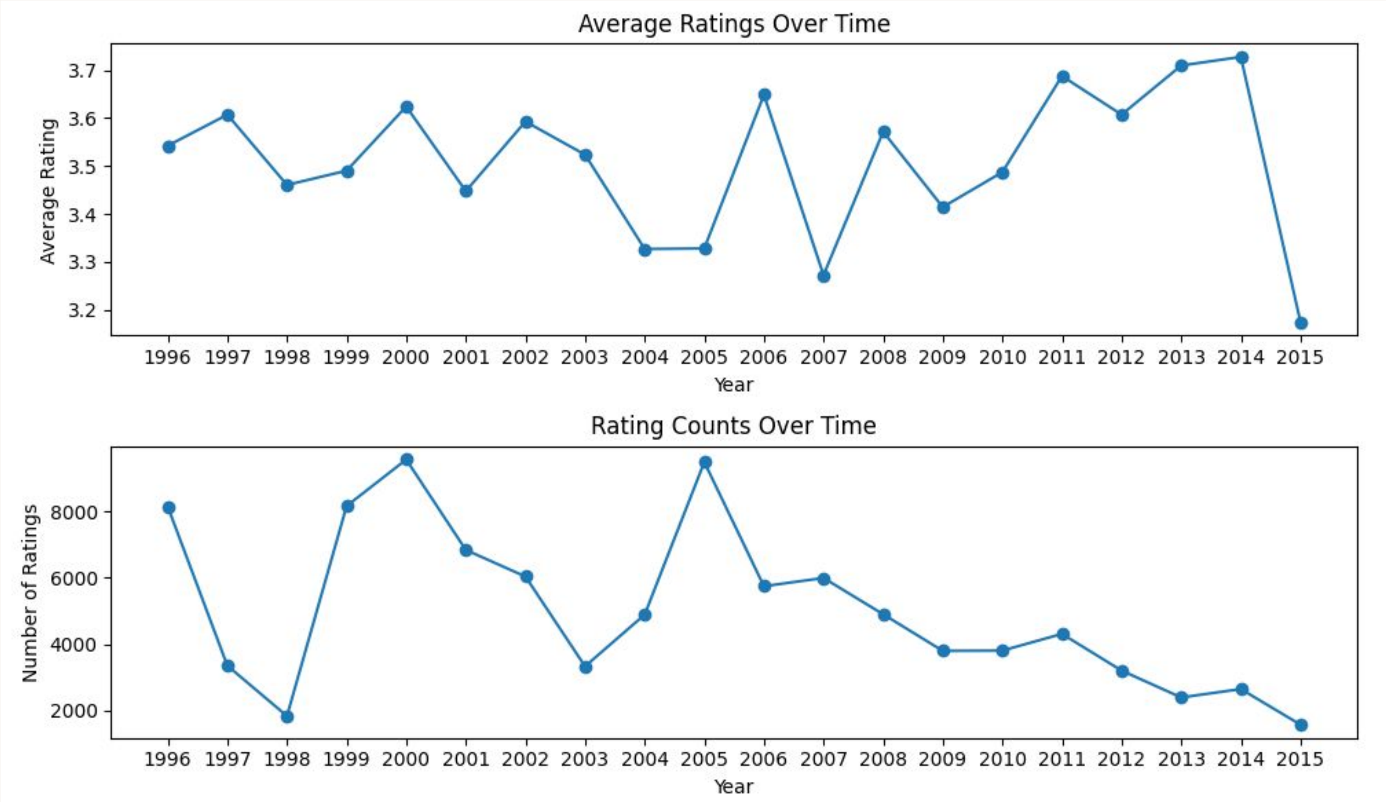
```
val ratingsWithTime = ratingsDF
 .withColumn("year", year(from_unixtime(col("timestamp"))))

val avgRatingsOverTime = ratingsWithTime
 .groupBy("year")
 .agg(avg("rating").alias("avgRating"))
 .sort("year")

avgRatingsOverTime.show()

val ratingCountsOverTime = ratingsWithTime
 .groupBy("year")
 .agg(count("rating").alias("numRatings"))
 .sort("year")

ratingCountsOverTime.show()
```

TEMPORAL TRENDS

year	avgRating
1996	3.5424331616384594
1997	3.6072922893006574
1998	3.460526315789474
1999	3.490390500673277
2000	3.624308238488044
2001	3.4476914085330215
2002	3.592880794701987
2003	3.523752254960914
2004	3.3268562078134587
2005	3.328034529950521
2006	3.6488343771746696
2007	3.2723028180757043
2008	3.5716326530612243
2009	3.4143384290985765
2010	3.4873750657548657
2011	3.6878773803994425
2012	3.6074444791992493
2013	3.709853249475891
2014	3.7279272451686243
2015	3.174344209852847

year	numRatings
1996	8154
1997	3346
1998	1824
1999	8169
2000	9577
2001	6844
2002	6040
2003	3326
2004	4889
2005	9499
2006	5748
2007	5997
2008	4900
2009	3794
2010	3802
2011	4306
2012	3197
2013	2385
2014	2639
2015	1563

PLOT


Average Ratings Over Time
Average Rating
3.7
3.6
3.5
3.4
3.3
3.2
1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015
Year
Rating Counts Over Time
Number of Ratings
8000
6000
4000
2000
1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015
Year

CO OCCURRENCE ANALYSIS

```
val moviePairs = ratingsDF.alias( "r1")
 .join(ratingsDF.alias( "r2"), col("r1.userId") === col("r2.userId")&&
col("r1.movieId") < col("r2.movieId"))
 .select(col("r1.movieId").alias("movie1"),
col("r2.movieId").alias("movie2"))

val movieConnections = moviePairs
 .groupBy("movie1", "movie2")
 .agg(count("*").alias("coOccurrences"))
 .orderBy(desc("coOccurrences"))

movieConnections.show()
```

CO-OCCURRENCE ANALYSIS

movie1	movie2	coOccurrences
356	480	243
296	593	228
296	356	225
296	318	221
356	593	209
296	480	206
480	589	204
318	593	198
318	356	198
356	589	195
260	1196	192
50	296	190
260	1210	186
480	593	185
47	296	185
1196	1210	184
110	356	183
457	480	182
110	296	182
318	480	180

ASSOCIATION MINING

```
val movieBaskets = ratingsDF
 .groupBy("userId")
 .agg(collect_set("movieId").alias("ratedMovies"))

movieBaskets.show()

import org.apache.spark.ml.fpm.{FPGrowth, FPGrowthModel}

val fpGrowth = new FPGrowth()
 .setItemsCol("ratedMovies")
 .setMinSupport(0.1)
 .setMinConfidence(0.5)

val model: FPGrowthModel = fpGrowth.fit(movieBaskets)

val frequentItemsets = model.freqItemsets

frequentItemsets.show()
```

```
+------+--------------------+
|userId|         ratedMovies|
+------+--------------------+
|     1|[2644, 3479, 2, 1...|
|     2|[589, 3173, 3937,...|
|     3|[610, 1272, 2615,...|
|     4|[356, 589, 531, 4...|
|     5|[110, 589, 364, 1...|
|     6|[494, 1, 52, 743,...|
```

ASSOCIATION MINING

items	freq
[95]	105
[95, 648]	74
[95, 32]	76
[95, 457]	71
[95, 780]	83
[95, 589]	71
[95, 380]	71
[95, 736]	74
[95, 480]	77
[7438]	103
[7438, 4226]	76
[7438, 2571]	81
[7438, 2571, 296]	72

RATINGS PREDICTION

Comparative Analysis between Different Prediction Models

PREPARING THE DATA

```
val joinedDF = ratingsDF.join(moviesDF, Seq( "movieId"), "inner")
val selectedData = joinedDF.select( "userId", "movieId", "genres",
"timestamp", "rating")

// Feature Encoding for Categorical Features (like genres)
val indexer = new
StringIndexer().setInputCol( "genres").setOutputCol( "genreIndex")
val indexed = indexer.fit(selectedData).transform(selectedData)
val encoder = new
OneHotEncoder().setInputCol( "genreIndex").setOutputCol( "genreVec").fit(indexe
d)
val encoded = encoder.transform(indexed)

// Assemble Features into a Single Vector (including categorical and
numerical features)
val assembler = new VectorAssembler().setInputCols(Array( "userId", "movieId",
"genreVec", "timestamp")).setOutputCol( "features")

val assembledData = assembler.transform(encoded).select( "rating", "features")

val Array(training, test) = assembledData.randomSplit(Array( 0.8, 0.2))
```

LINEAR REGRESSION

```
// Train the Regression Model using Multiple Features
val lr = new
LinearRegression().setLabelCol("rating").setFeaturesCol("features")
val lrModel = lr.fit(training)
```

```
// Make predictions on the test set
val predictions_lr = lrModel.transform(test)
```

RANDOM FOREST

```
val rfr = new
RandomForestRegressor().setLabelCol("rating").setFeaturesCol("features"
)
val rfrModel = rfr.fit(training)

// Make predictions on the test set
val predictions_rfr = rfrModel.transform(test)
```

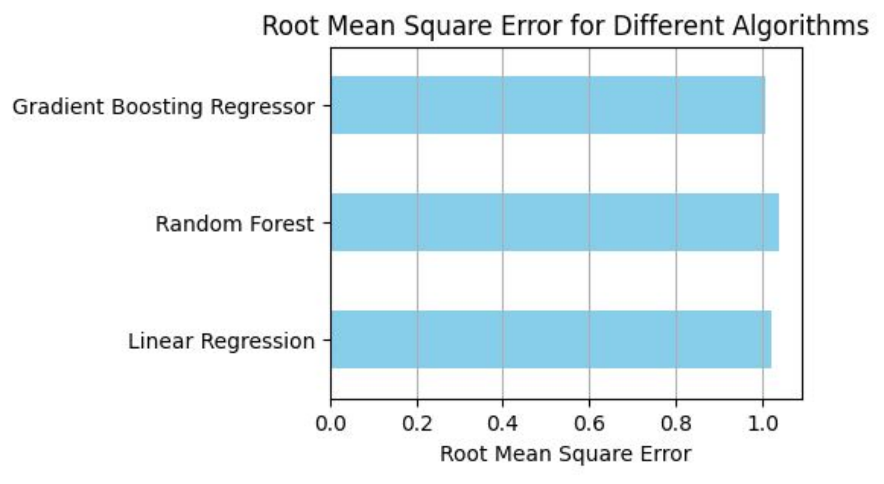
Gradient Boosting Regression

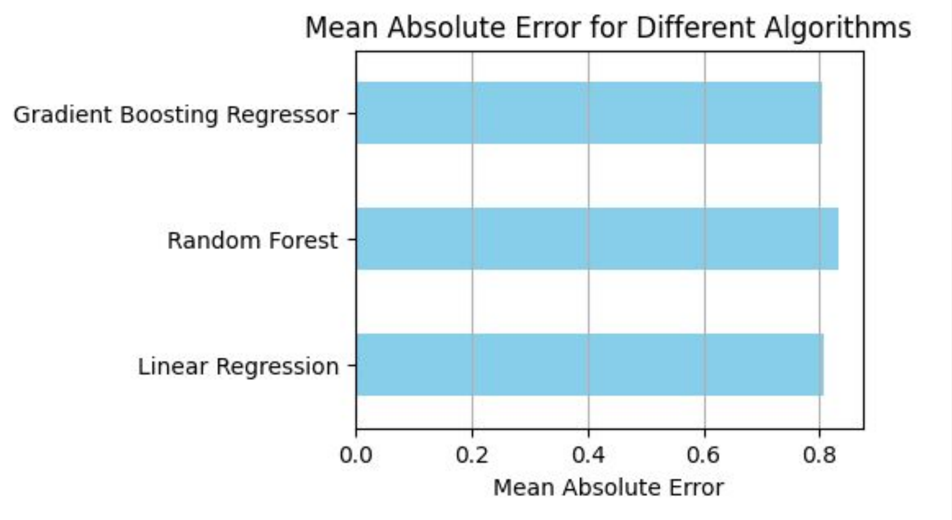
```
val gbt = new
GBTRegressor().setLabelCol("rating").setFeaturesCol("features")
val gbtModel = gbt.fit(training)

// Make predictions on the test set
val predictions_gbt = gbtModel.transform(test)
```

EVALUATING THE MODEL

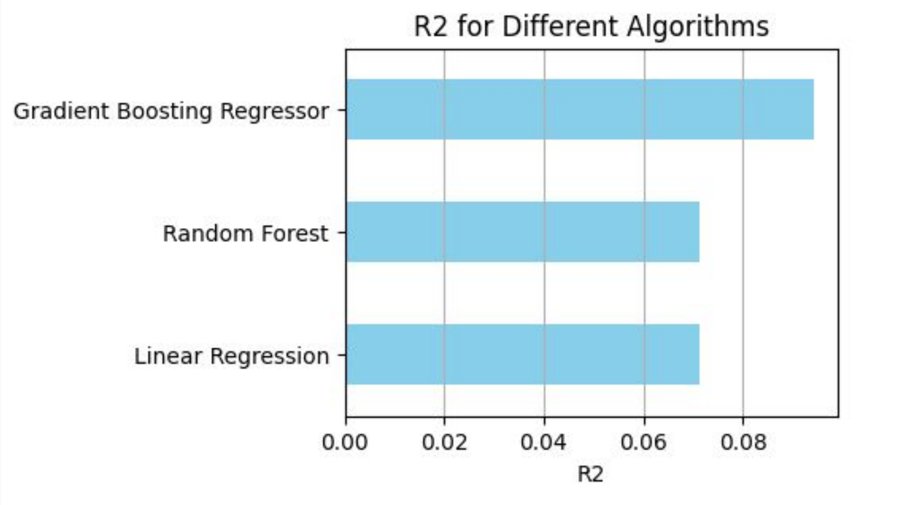
```
//RMSE
predictions.select(sqrt(avg((col("rating") - col("prediction")) * (col("rating") - col("prediction"))))).as("rmse").show()

//MAE
predictions.select(avg(abs(col("rating") - col("prediction")))).as("mae").show()

//R-squared error
predictions.select(
 corr(col("prediction"), col("rating")) * corr(col("prediction"), col("rating"))
).as("r2").show()

//Accuracy
val threshold = 0.5
val accuracy = predictions.select((sum(when(abs(col("rating") - col("prediction")) <= threshold, 1).otherwise(0)) / count(col("rating"))).as("accuracy")).show()
```

PLOTS

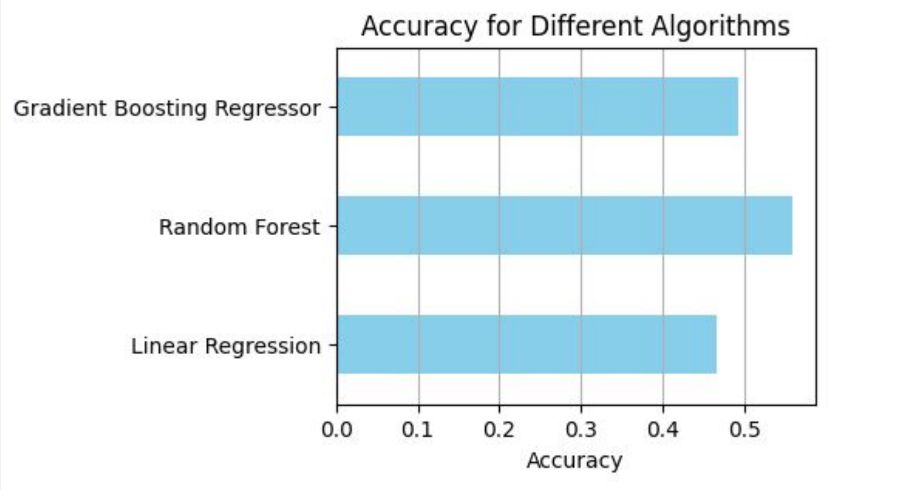

Root Mean Square Error for Different Algorithms
Gradient Boosting Regressor
Random Forest
Linear Regression
0.0
0.2
0.4
0.6
0.8
1.0
Root Mean Square Error


Mean Absolute Error for Different Algorithms
Gradient Boosting Regressor
Random Forest
Linear Regression
0.0
0.2
0.4
0.6
0.8
Mean Absolute Error

PLOTS



R2 for Different Algorithms
Gradient Boosting Regressor
Random Forest
Linear Regression
0.00
0.02
0.04
0.06
0.08
R2



Accuracy for Different Algorithms
Gradient Boosting Regressor
Random Forest
Linear Regression
0.0
0.1
0.2
0.3
0.4
0.5
Accuracy

MOVIE RECOMMENDATION

COLLABORATIVE FILTERING + CONTENT BASED FILTERING

RECOMMENDATION SYSTEM

```scala
import org.apache.spark.ml.recommendation.ALS

val ratings = ratingsDF.select("userId", "movieId", "rating")
 .na.drop()

// Create the ALS model
val als = new ALS()
 .setMaxIter(10)
 .setRegParam(0.01)
 .setUserCol("userId")
 .setItemCol("movieId")
 .setRatingCol("rating")

// Fitting the ALS model to the training data
val model = als.fit(ratings)
```

RECOMMENDATION SYSTEM

```
val genreFilteredMovies = moviesWithGenresDF.filter( col("genre") ===
userGenre)

// Get tags for movies
val userTagRelevance = genomeTagsDF.filter( col("tag") ===
userTag).select("tagId").first().getInt(0)
val tagRelevanceDF = genomeScoresDF.filter( col("tagId") ===
userTagRelevance)

val userPredictions = model.recommendForAllUsers( 50)
val relevantMoviesDF = genreFilteredMovies.join(tagRelevanceDF,
"movieId")
```

RECOMMENDATION SYSTEM

```
import org.apache.spark.sql.functions.{ col, expr}
```

```
val userTopRecommendations = relevantMoviesDF.alias( "relevant")
 .join(userPredictions.select( col("userId"),
explode(col("recommendations")))
   .select(col("userId"), col("col.movieId"), col("col.rating"))
   .alias("predictions"), expr("relevant.movieId =
predictions.movieId"))
 .orderBy(col("predictions.rating").desc)
 .select(col("relevant.movieId"), col("relevant.title"),
col("predictions.rating"), col("relevant.relevance"))
 .limit(50)
```

RECOMMENDATION SYSTEM

```
val distinctRecommendations =
userTopRecommendations.dropDuplicates("movieId")
```

```
val finalRecommendations = distinctRecommendations
 .join(linkDF, Seq("movieId"), "left")
 .select("movieId", "title", "rating", "relevance", "ImdbID", "TmdbID")
```

```
finalRecommendations.show(10)
```

RECOMMENDATIONS

```
val userId = 123
val userGenre = "Comedy"
val userTag = "thriller"
```

movieId	title	rating	relevance	ImdbID	TmdbID
187	Party Girl (1995)	13.802541	0.123	114095	36196
663	Kids in the Hall:...	12.055248	0.09025	116768	18414
921	My Favorite Year ...	11.439761	0.15200000000000002	84370	31044
1541	Addicted to Love ...	11.322601	0.11975000000000002	118556	2058
1546	Schizopolis (1996)	11.303277	0.10899999999999999	117561	16375
1772	Blues Brothers 20...	11.702422	0.041499999999999998	118747	11568
2014	Freaky Friday (1977)	11.5768385	0.064249999999999997	76054	16084
2163	Attack of the Kil...	13.445128	0.062	80391	2182
2583	Cookie's Fortune ...	12.734982	0.1385	126250	9465
2618	Castle, The (1997)	12.247027	0.1195	118826	13852

THANK YOU