**MOVIE RECOMMENDATION AND ANALYSIS OF DATA USING SPARK**

**Date:** 8 Nov, 2023

**Course Code and Name:** 2CS702 Big Data Analytics

CREATED BY - 20BCE027

20BCE050

20BCE056

20BCE066

20BCE072

| **Sr No** | **Title** |
| --- | --- |
| 1 | Problem Statement |
| 2 | Introduction |
| 3 | Tools |
| 4 | Installation Steps |
| 5 | Dataset |
| 6 | Results |
| 7 | Applications |
| 8 | Conclusion |

**Problem statement:**

Data is rapidly driving the film business, and movie analytics utilizing Scala and Spark answers important difficulties in this dynamic context. The goal of this project is to use the capabilities of Scala, a highly expressive and efficient programming language, and Spark, a distributed big data processing platform, to address significant industrial concerns, optimizing content creation and investment choices.

**Overview:**

Movie analytics in Spark and Scala is a data-driven approach to gaining insights from movie-related datasets. By utilizing the Apache Spark framework and the Scala programming language, it involves data ingestion, transformation, and exploratory analysis to uncover valuable information about users and movies. Leveraging Spark RDDs, Spark SQL, and Spark DataFrames, this process allows for both low-level data manipulation and high-level structured data processing. Whether it's identifying popular movies, user preferences, or applying machine learning for recommendation systems, movie analytics enables data-driven decision-making in the realm of cinema, ultimately improving the movie-watching experience for audiences.

**Introduction**

Big data analytics is the sometimes difficult process of evaluating huge data to identify information – such as hidden patterns, correlations, market trends, and consumer preferences – that may help organisations make educated business choices. On a large scale, data analytics tools and methodologies enable organisations to analyse data sets and obtain new insights. Business intelligence (BI) queries provide answers to fundamental questions about business operations and performance.

Big data analytics is a type of advanced analytics that includes complicated applications with aspects such as predictive models, statistical algorithms, and what-if analysis powered by analytics systems.

With the introduction of big data analytics, the world of cinema has expanded tremendously, and the combination of Scala and Spark has emerged as a powerful combo in transforming the film business. This novel approach to movie analytics not only allows for the dissection of large and complex datasets, but also yields insights that can revolutionise decision-making processes for filmmakers, studios, and distribution platforms.

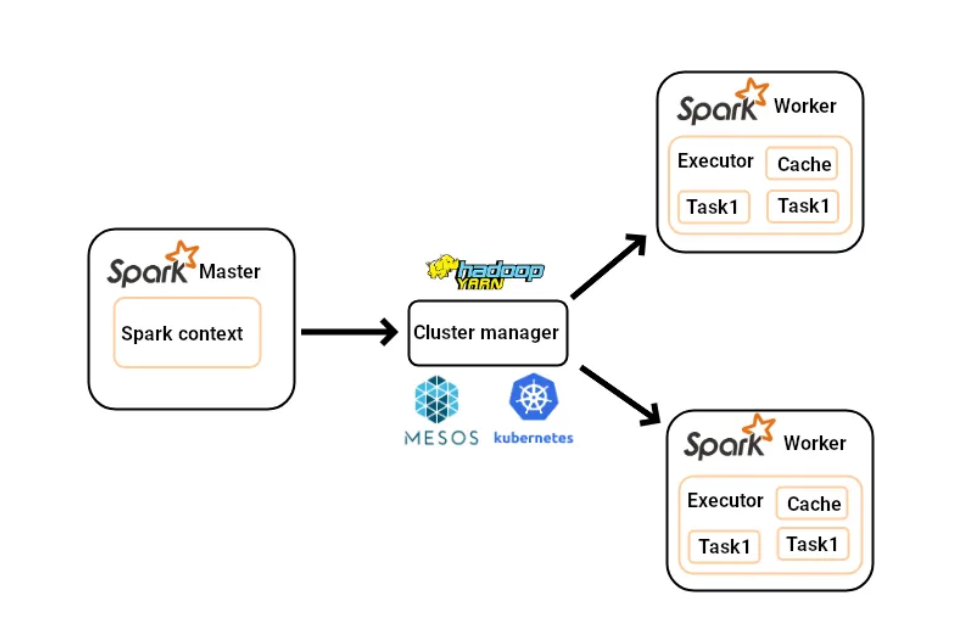
Scala, with its conciseness and solid functional programming features, is an excellent choice for constructing complicated data processing systems. When combined with Apache Spark, a robust big data framework, this combination can easily handle the vast amounts of data connected with the film business.

The use of Scala and Spark for movie analytics has the potential to impact several domains. It enables studios and distributors to forecast box office performance, optimise marketing campaigns, and personalise content suggestions to individual interests. This, in turn, leads to a more personalised cinematic experience for spectators and improved financial possibilities for the industry.

In this age of data-driven decision-making, the combination of Scala and Spark is transforming the cinema landscape, providing important insights and enhancing the way we enjoy and connect with films.

**Tools :**

1. **Apache Spark:** Apache Spark is a high-performance, open-source distributed computing system built for large data processing and analytics. Its in-memory processing capabilities and support for several data sources make it a quick and dependable platform for large-scale data processing operations. Spark is well-known for its adaptability, including libraries for machine learning, graph processing, and real-time data streaming. Its fault-tolerant and distributed nature provides scalability, making it suited for a wide range of applications, from data processing to sophisticated, iterative algorithms. Whether for batch processing or real-time data streaming, Apache Spark has become a cornerstone in the big data world, providing efficient, quick, and scalable data processing.



1. **Scala:** Scala is a dynamic programming language that mixes functional and object-oriented concepts. Scala, known for its conciseness and expressiveness, is intended to increase developer productivity. It runs on the Java Virtual Machine (JVM) and interacts with Java easily, making it a good solution for both JVM-based and online applications. Scala’s powerful type system and extensive library support allow for safe and fast code development. It’s useful for large data analytics, distributed computing, and concurrent programming because of its capacity to handle complicated data processing tasks. Scala’s versatility and ease of use make it an excellent choice for a wide range of software development tasks.



**Installation steps of Spark:**

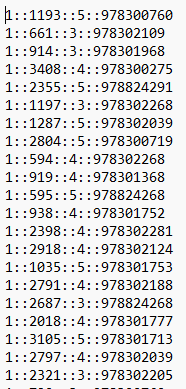
* Download and install python 3.11
* Download and install java SDK 17
* Download Spark 3.4.1 pre built for Apache Hadoop 3.3 & later
* Extract the spark download files
* Download Hadoop winutils from github
* Create folder structure- C:\SPARK & C:\HADOOP\bin
* Copy the downloaded spark and hadoop files into above folders.
* Set the environment variables for java, hadoop and spark
* In cmd from this path - C:/SPARK/bin - execute command spark-shell



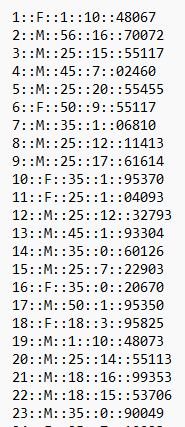
**Dataset:**

We have three datasets named as movies.dat, ratings.dat and users.dat which are explained below:

1. **rating.dat:** This data is consisted of the user ratings. This dataset has columns named are UserID, MovieID, Rating and Timestamp. Following is the description of the columns:
   1. UserID: It has range of 1 to 6040. Each user has rated 20 movies atleast. It references to UserID in users.dat.
   2. MovieID: It has range of 1 to 3952. It references to MovieId in movies.dat.
   3. ratings: ratings are made at scale of 5 star.
   4. Timestamp: It is represented in seconds since the epoch as returned by time.



1. **users.dat:** This data is consisted of the user data. This dataset has columns named are UserID, Gender, Age, Occupation and Zip-code. These information has been provided by user voluntarily. So, It may not be accurate. Following is the description of the columns:
2. UserID: It has range of 1 to 6040. It is unique for all the users.
3. Gender: M is for male and F is for female gender.
4. Age: Age has been categorised in ranges such as under 18, 18-24, 25-34, 35-44, 45-49, 50-55, 56+.
5. Occupation: It has been categorised into 20 different categories such as farmer, artist, homemaker, customer service, healthcare, retired, scientist, engineer, tradesman, unemployed, executive/managerial, K-12 student, educator, programmer, lawyer etc.



1. **movies.dat:** It has information about movie. It has columns of MovieId, Title and Genres. Following is the description of the columns:
2. MovieID: It has range of 1 to 3952. It is unique for all the movies.
3. Titles: It is unique as well. It is provided by IMDB.
4. Genres: More than 10 genres have been included in dataset which are action, adventure, fantasy, drama, war, western, thriller, horror, musical, animation etc.



**Spark Dataframe:**

Apache Spark DataFrames are structured data collections that combine the benefits of conventional databases with the distributed computing capabilities of Spark. They include schema information, making them appropriate for structured and semi-structured data. DataFrames take use of Spark’s optimisation techniques to improve speed, allowing for simple data manipulation and processing via a high-level API. They are dispersed throughout a cluster for parallel processing, and their in-memory capabilities allow for quick data analysis. DataFrames are extensively used for data transformations, SQL-like querying, and seamless integration with diverse data sources, making them a useful tool for large data processing and analysis in languages such as Scala, Python, and Java.

1. Prepare Movies dataset: We clean delimited data and extracting the year and genre from movies dataset.
2. Prepare Users dataset: We load a double delimited csv file into a dataframe and specify schema programatically.
3. Prepare Ratings dataset: We load a double delimited csv file into a dataframe and specify schema programatically. To achieve above results, we have to run command ’sh execute.sh’ in terminal at relavant path.

**Spark RDD:**

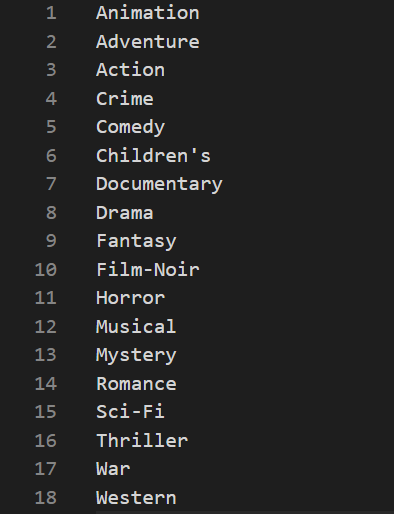
Apache Spark RDD (Resilient Distributed Dataset) is a core data structure of Spark that is intended for distributed data processing. RDDs are data collections that may be processed in parallel across several computers in a cluster. Fault tolerance, immutability, and distributed processing are important aspects of RDDs:

* 1. distributed data: RDDs divide data among numerous nodes, allowing for parallel processing. Because of their distributed nature, RDDs are well suited to handle massive datasets.
  2. resilience: RDDs are resilient in the sense that they can recover from node failures. When a node fails, Spark may rebuild and recompute the lost data partitions using the original source data.
  3. immutability: RDDs are immutable, which means that their data cannot be modified after they are created. Transforms on RDDs, on the other hand, generate new RDDs, maintaining data integrity and simplifying fault recovery.
  4. transformations: RDDs provide numerous transformations such as map, filter, and reduce, allowing users to change data in a distributed and parallel fashion.
  5. actions: RDDs also feature actions, which cause transformations to be executed and results to be returned to the driver program, allowing users to extract information or write data.
  6. caching: Caching: RDDs can be cached in memory, improving efficiency when iterative algorithms or numerous calculations on the same dataset are used. RDDs serve as Spark’s foundation, providing a modular, fault-tolerant, and distributed data structure for large-scale data processing and analytics. We have used RDD for following analytics:
  7. Find the latest movies which have been released.
  8. Find the distinct genres present in the dataset.
  9. Find the number of movies in each genre.
  10. Number of movies starting with letters or numbers.
  11. Find top ten most viewed movies.

**Data Analysis Results:**

1. **To obtain distinct genres in the movie dataset:**

| val movies\_rdd=sc.textFile("../../Movielens/movies.dat")  val genres=movies\_rdd.map(lines=>lines.split("::")(2))  val testing=genres.flatMap(line=>line.split('|'))  val genres\_distinct\_sorted=testing.distinct().sortBy(\_(0))  genres\_distinct\_sorted.saveAsTextFile("result")  System.exit(0) |
| --- |

****

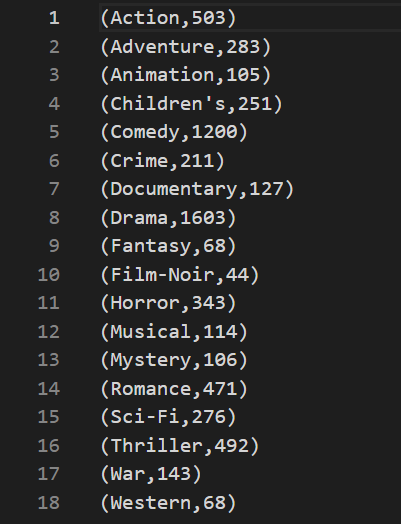
1. **To obtain the most recent movie:**

| val movies\_rdd=sc.textFile("../../Movielens/movies.dat")  val movie\_nm=movies\_rdd.map(lines=>lines.split("::")(1))  val year=movie\_nm.map(lines=>lines.substring(lines.lastIndexOf("(")+1,lines.lastIndexOf(")")))  val latest=year.max  val latest\_movies=movie\_nm.filter(lines=>lines.contains("("+latest+")")).saveAsTextFile("result")  System.exit(0) |
| --- |

****

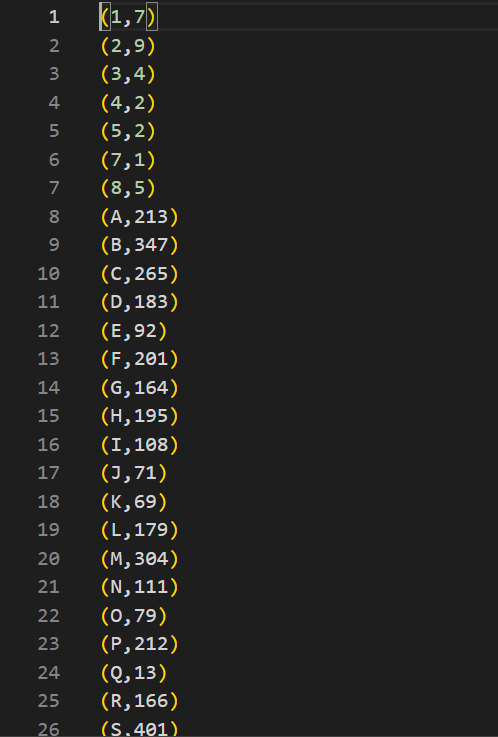
1. **To calculate the total number of films in each category:**

| val movies\_rdd=sc.textFile("../../Movielens/movies.dat")  val genre=movies\_rdd.map(lines=>lines.split("::")(2))  val flat\_genre=genre.flatMap(lines=>lines.split("\\|"))  val genre\_kv=flat\_genre.map(k=>(k,1))  val genre\_count=genre\_kv.reduceByKey((k,v)=>(k+v))  val genre\_sort= genre\_count.sortByKey()  genre\_sort.saveAsTextFile("result-csv")  System.exit(0) |
| --- |

****

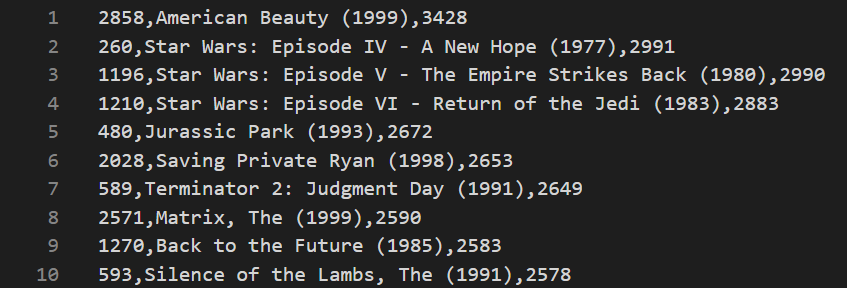
1. **To count the number of films that begin with a number (0-9) or a letter (A-Z):**

| val movies\_rdd=sc.textFile("../../Movielens/movies.dat")  val movies=movies\_rdd.map(lines=>lines.split("::")(1))  val string\_flat=movies.map(lines=>lines.split(" ")(0))  // check for the first character for a letter then find the count  val movies\_letter=string\_flat.filter(word=>Character.isLetter(word.head)).map(word=>(word.head.toUpper,1))  val movies\_letter\_count=movies\_letter.reduceByKey((k,v)=>k+v).sortByKey()  // check for the first character for a digit then find the count  val movies\_digit=string\_flat.filter(word=>Character.isDigit(word.head)).map(word=>(word.head,1))  val movies\_digit\_count=movies\_digit.reduceByKey((k,v)=>k+v).sortByKey()  // Union the partitions into a same file  val result=movies\_digit\_count.union(movies\_letter\_count).repartition(1).saveAsTextFile("result-csv")  System.exit(0) |
| --- |

****

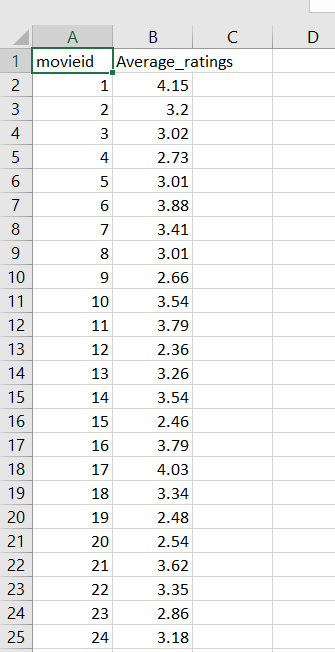
1. **To retrieve the most popular films:**

| val ratingsRDD=sc.textFile("../../Movielens/ratings.dat")  val movies=ratingsRDD.map(line=>line.split("::")(1).toInt)  val movies\_pair=movies.map(mv=>(mv,1))  val movies\_count=movies\_pair.reduceByKey((x,y)=>x+y)  val movies\_sorted=movies\_count.sortBy(x=>x.\_2,false,1)  val mv\_top10List=movies\_sorted.take(10).toList  val mv\_top10RDD=sc.parallelize(mv\_top10List)  val mv\_names=sc.textFile("../../Movielens/movies.dat").map(line=>(line.split("::")(0).toInt,line.split("::")(1)))  val join\_out=mv\_names.join(mv\_top10RDD)  join\_out.sortBy(x=>x.\_2.\_2,false).map(x=> x.\_1+","+x.\_2.\_1+","+x.\_2.\_2).repartition(1).saveAsTextFile("Top-10-CSV")  System.exit(0) |
| --- |

****

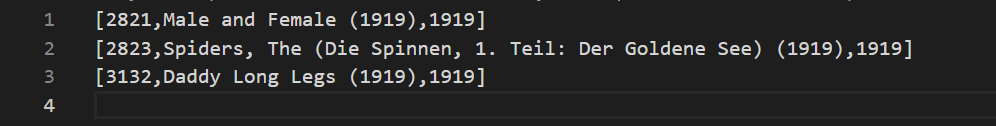
1. **To obtain average rating per movie:**

| spark.sql("""Select movieid,  cast((avg(rating)) as decimal(16,2)) as Average\_ratings  from sparkdatalake.ratings  group by movieid  order by cast(movieid as int) asc  """).repartition(1).write.format("csv").option("header","true").save("result")  System.exit(0) |
| --- |

****

1. **To obtain list of the oldest movies:**

| val movies\_rdd=sc.textFile("../../Movielens/movies.dat")  // 1st method, convert existing rdd into DF using toDF function and then make it into a view  val movies\_DF=movies\_rdd.toDF.createOrReplaceTempView("movies\_view")  // To use spark.sql, it should be at least a temporary view or even an table  spark.sql(""" select  split(value,'::')[0] as movieid,  split(value,'::')[1] as moviename,  substring(split(value,'::')[1],length(split(value,'::')[1])-4,4) as year  from movies\_view """).createOrReplaceTempView("movies");  // To view the records, use spark.sql("select \* from movies").show()  var result=spark.sql("Select \* from movies m1 where m1.year=(Select min(m2.year) from movies m2)").repartition(1).rdd.saveAsTextFile("result")  System.exit(0); |
| --- |

****

**Applications:**

Movie analytics using Spark and Scala has a wide range of applications that can benefit the film industry, streaming platforms, and movie enthusiasts. Some key applications include:

* + 1. Movie Recommendations: Implementing collaborative filtering and content-based recommendation systems to suggest movies to users based on their preferences and viewing history, enhancing user engagement and satisfaction.
    2. Audience Segmentation: Analyzing viewer demographics, preferences, and behavior to segment audiences for targeted marketing and content creation, ensuring that movies cater to specific audience segments.
    3. Content Curation: Automating the selection and curation of movies for streaming platforms by analyzing user interactions and preferences, optimizing content discovery and viewer retention.
    4. Box Office Predictions: Using historical data and machine learning models to predict box office performance, allowing studios to make informed decisions about marketing and distribution strategies.
    5. Content Licensing: Assessing the value of movies by analyzing viewer engagement and demographics, aiding in the negotiation and acquisition of content licensing deals.
    6. Quality Control: Identifying viewer sentiments and feedback through sentiment analysis to improve movie quality, scriptwriting, and production decisions.
    7. Genre Analysis: Analyzing movie genres and trends over time to guide content creation and marketing strategies, ensuring alignment with popular genres.
    8. Profitability Analysis: Estimating the profitability of movies by analyzing production costs, box office revenues, and streaming platform revenues, enabling informed investment decisions.
    9. Viewer Experience Enhancement: Analyzing viewer behavior to enhance the user experience by optimizing content recommendations, personalizing interfaces, and minimizing buffering or streaming issues.
    10. Piracy Detection: Identifying and monitoring illegal distribution of movies through web scraping, user-generated content platforms, or torrents, helping in copyright protection and enforcement.
    11. A/B Testing: Conducting A/B testing with different versions of movie trailers, posters, or promotional material to determine which versions perform best with audiences.
    12. Trend Analysis: Monitoring and capitalizing on emerging movie trends, such as analyzing social media discussions and search trends to inform content creation and marketing campaigns.
    13. Predictive Maintenance: Analyzing the condition of movie screening equipment to prevent downtime and ensure seamless movie screenings in theaters.

Movie analytics in Spark and Scala empowers the industry with data-driven insights and automation, contributing to improved content creation, distribution, and user engagement, as well as informed decision-making across various aspects of the film industry.

**Movie Recommendations Results-**

**RUN the following code from path - C:\SPARK\bin**

**// providing data type to userId, movieId, rating and timestamp and show the table.**

import org.apache.spark.sql.functions.\_

import org.apache.spark.sql.types.\_

import org.apache.spark.ml.evaluation.RegressionEvaluator

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

case class Rating(userId: Int, movieId: Int, rating: Float, timestamp: Long)

def parseRating(str: String): Rating = {

val fields = str.split("::")

assert(fields.size == 4)

Rating(fields(0).toInt, fields(1).toInt, fields(2).toFloat, fields(3).toLong)

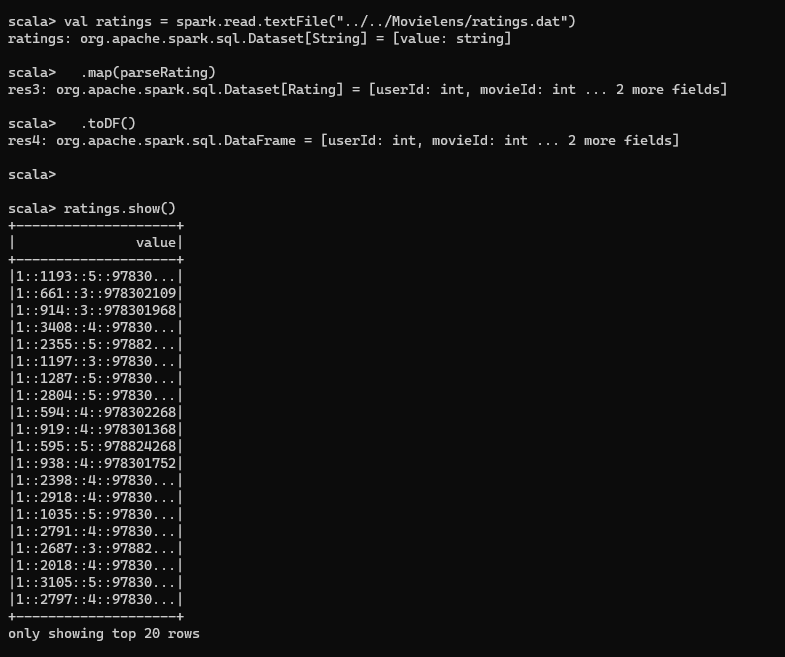
}

val ratings = spark.read.textFile("../../Movielens/ratings.dat")

.map(parseRating)

.toDF()

ratings.show()



**// Converts the above data to distinct columns for userId, movieId, rating and timestamp**

val splitData = ratings.withColumn("split\_values", split(col("value"), "::"))

val parsedData = splitData.select(

splitData("split\_values")(0).cast(IntegerType).alias("userId"),

splitData("split\_values")(1).cast(IntegerType).alias("movieId"),

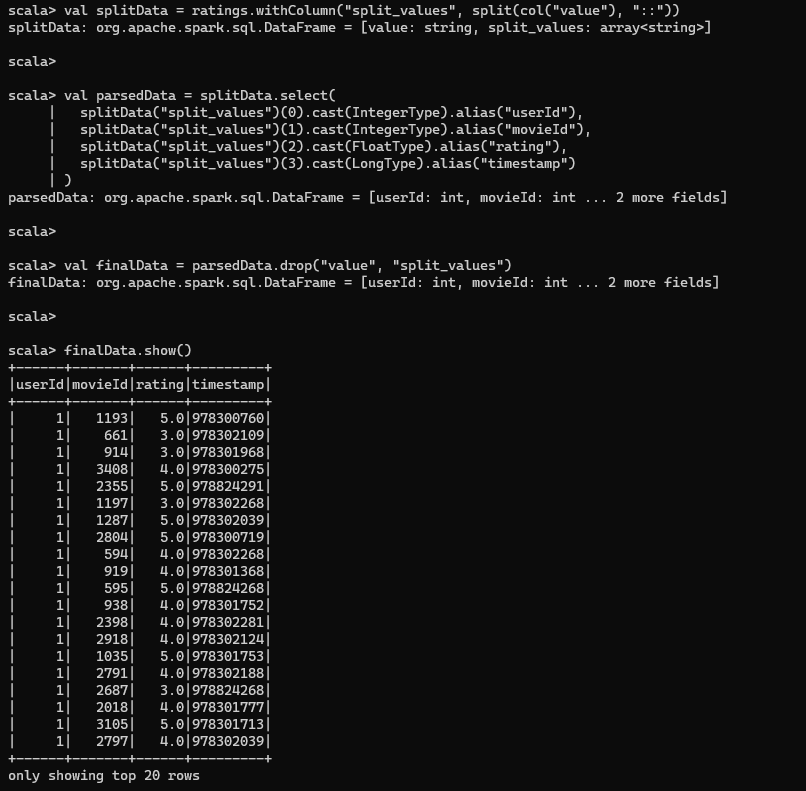
splitData("split\_values")(2).cast(FloatType).alias("rating"),

splitData("split\_values")(3).cast(LongType).alias("timestamp")

)

val finalData = parsedData.drop("value", "split\_values")

finalData.show()



**// Split the data into training(80%) and testing (20%)**

val ratings=finalData

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

// Creates and train a collaborative filtering recommendation model using Alternative Least Square(ALS) algorithm

val als = new ALS()

.setMaxIter(10)

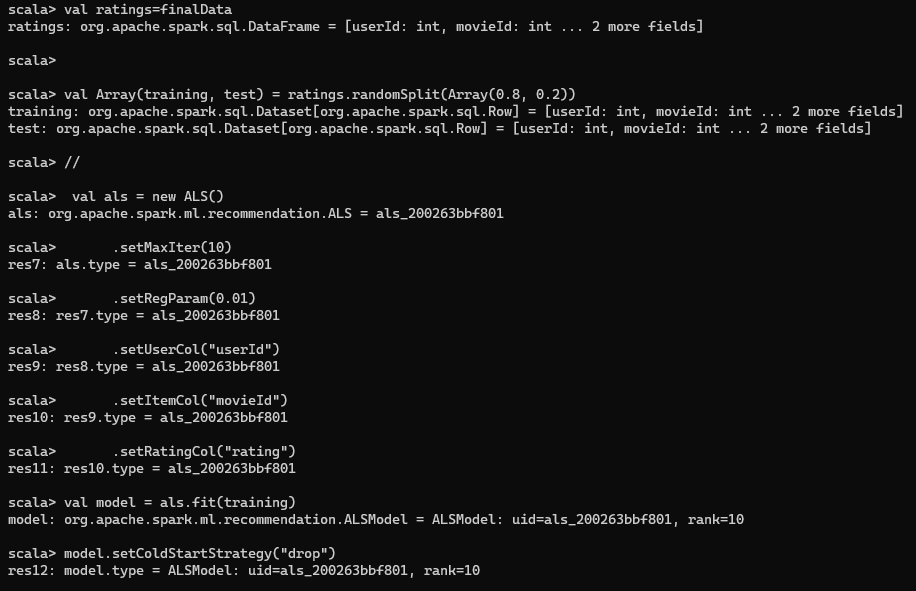
.setRegParam(0.01)

.setUserCol("userId")

.setItemCol("movieId")

.setRatingCol("rating")

val model = als.fit(training)



**// will drop rows having NaN value and transform use to make prediction for user-item pair on test data**

model.setColdStartStrategy("drop")

val predictions = model.transform(test)

**// Evaluating the performance of model using a regression evaluation metric**

val evaluator = new RegressionEvaluator()

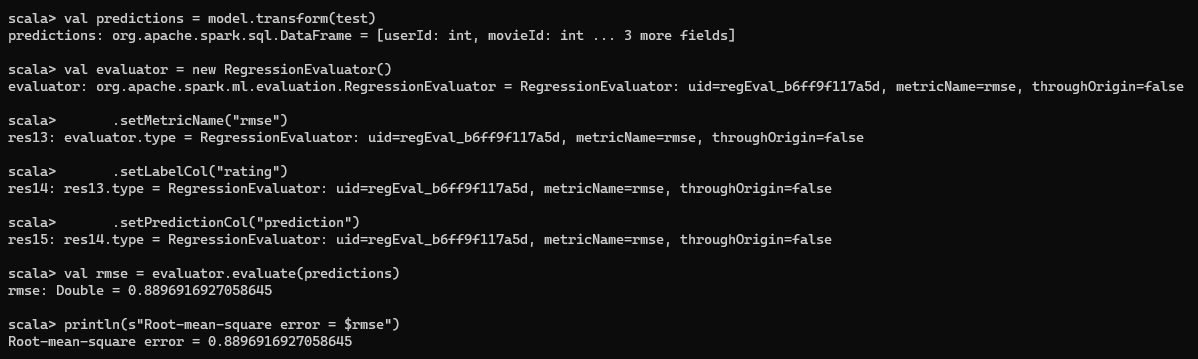
.setMetricName("rmse")

.setLabelCol("rating")

.setPredictionCol("prediction")

val rmse = evaluator.evaluate(predictions)

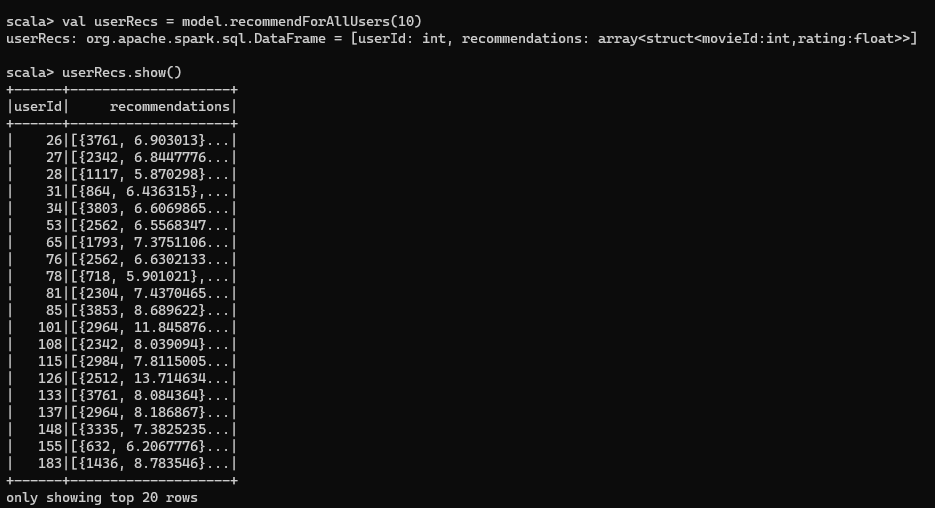
println(s"Root-mean-square error = $rmse")



**//Generate top 10 movie recommendations for each user**

val userRecs = model.recommendForAllUsers(10)

userRecs.show()



**//show 2nd column**

val row = userRecs.take(1)(0)

val arrayValue = row.getAs[Seq[org.apache.spark.sql.Row]](1)

**// Iterate through the array and print its content**

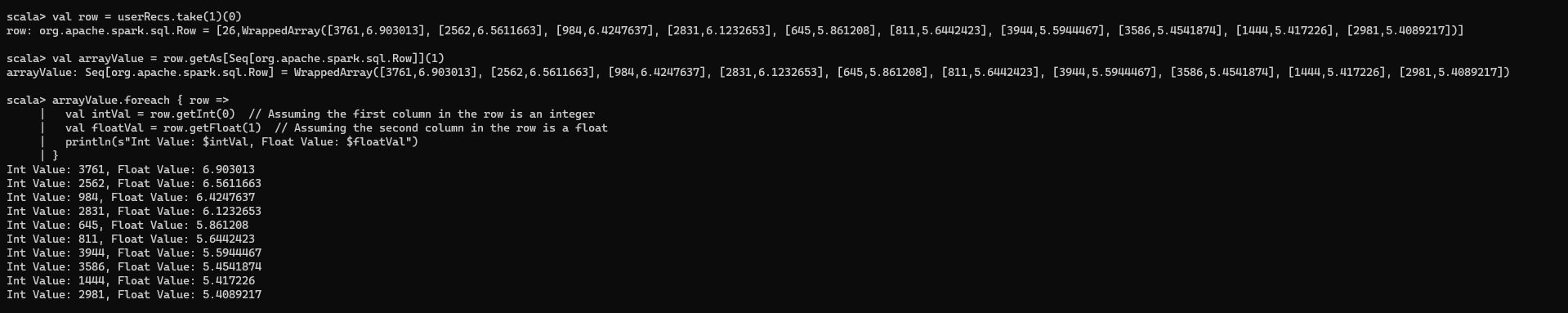
arrayValue.foreach { row =>

val intVal = row.getInt(0) // Assuming the first column in the row is an integer

val floatVal = row.getFloat(1) // Assuming the second column in the row is a float

println(s"Int Value: $intVal, Float Value: $floatVal")

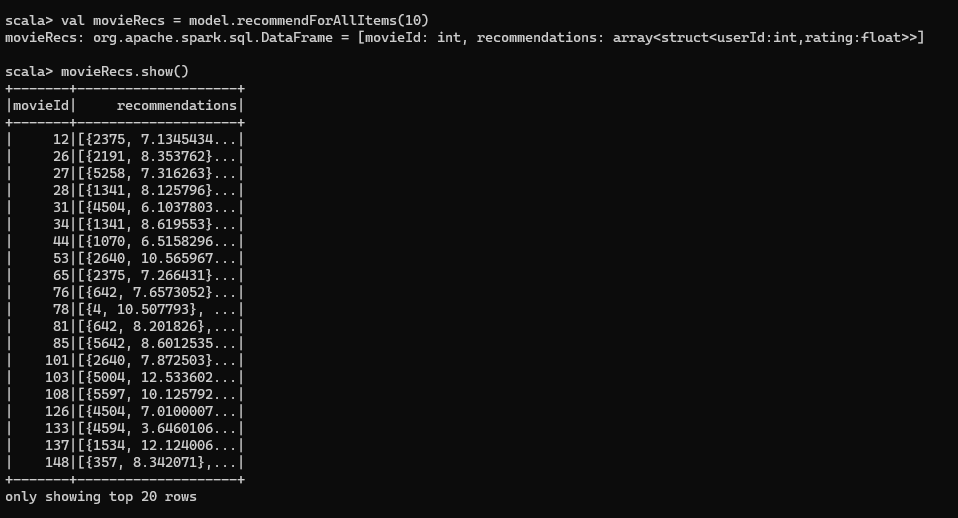
}



**// Generate top 10 user recommendations for each movie**

val movieRecs = model.recommendForAllItems(10)

movieRecs.show()



**// Generate top 10 movie recommendations for a specified set of users**

val users = ratings.select(als.getUserCol).distinct().limit(3)

val userSubsetRecs = model.recommendForUserSubset(users, 10)

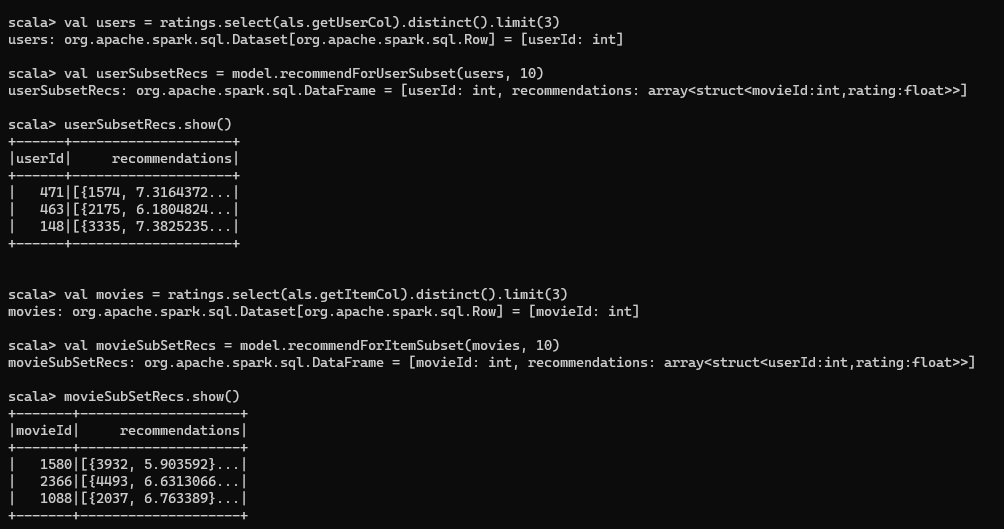
userSubsetRecs.show()

**// Generate top 10 user recommendations for a specified set of movies**

val movies = ratings.select(als.getItemCol).distinct().limit(3)

val movieSubSetRecs = model.recommendForItemSubset(movies, 10)

movieSubSetRecs.show()



**Conclusion:**

When we talk about Scala and Apache Spark collaborating, it’s like a dynamic combo in the field of visual data analytics. These two technologies excel in parallel data processing, and the impact on data analysis speed is rather impressive. Speed and scalability are critical in movie analytics, where we’re working with massive datasets. The ability of Apache Spark to process data in-memory is a game changer. It saves and reuses intermediate data in memory, reducing dependency on slower disk-based procedures. This results to lightning-fast data processing, which is a game changer in the realm of movie analytics. There is, however, a trade-off here. Spark demands a lot of RAM since it is memory-hungry. To put it another way, you need a large memory to get the most out of it. However, the payoff is well worth it: a significant increase in performance, making Scala with Spark a formidable combo for addressing big data analytics in the film industry. For the optimal movie data analytics experience, it’s all about finding the right balance of memory resources and performance