
SCALABLE MOVIE RECOMMENDATION SYSTEM

Avani Jindal¹ Sahil Jindal¹ Neelima Gaur¹

ABSTRACT

In our project, we will build a scalable and efficient recommendation system for movies. As it is a very widespread application, there are huge datasets for this purpose. To be able to handle that, our focus will be on making the system scalable so that it can handle large data loads with a huge number of users. For our data storage and processing engine, we will use PostgreSQL and Apache Spark respectively. The recommendation model will be based on collaborative filtering approach which will use information given by users, analyze it and finally recommend movies that are best matched to the user.

1 INTRODUCTION

There exists a huge amount of data of movies and audience ratings and it is ever-growing. It could be challenging to deal with this much data and get insights from it. Such an application caters to a huge population and has billions of users. In this enormous collection of movies, we aim to build a system which will store information related to movies and audience ratings. Using this data, we will build a recommendation system which can suggest movies to a user and will be able to scale with growth in data or users.

Companies like Netflix have well-established and advanced movie recommendation systems using the user data it collects. Our aim is to make a simpler but scalable version of this system using the data collected in the dataset MovieLens.

We will design a scalable movie recommendation system with an efficient data pipeline which can easily analyse the data and use Machine Learning to recommend movies to the user. Our plan is to use PostgreSQL as a scalable data store to store the data efficiently and Spark as a scalable data processing engine. The reason for choosing PostgreSQL as our data store is that the data we are using is structured and using a relational DBMS makes sense. PostgreSQL is a widely used RDBMS tool that also has a user-friendly pgAdmin interface. Moreover, Spark provides a great support for working with SQL under its SparkSQL module so our choice became easy. Making the decision to choose Apache Spark was a no-brainer for us as our workload is iterative and Spark is one of the best tools for such an application. It has efficient operations and can store large datasets in cluster memory. The purpose of our project is to help us understand the details of building a scalable data and analytics pipeline, get familiar with Spark and apply it to a recommendation system. Applying the theoretical knowledge from class to this project will help us understand how scalable systems

are built in industry and how they handle large data loads efficiently.

2 RELATED WORK

As mentioned above recommending movies to a user is one of the most popular problem domains in today's digital streaming world, several researchers have proposed their own recipe to solve this problem. In (Jeong and CHA, 2019) paper, the author has also leveraged Apache Spark to remove redundant data and solve the freeze start problem by applying hybrid and user-based collaborative filtering. Researchers have also tried to solve this problem by combining the usage of R and Apache Spark. In (Xie et al., 2017) paper, authors have proposed a parallel implementation of the ALS recommendation algorithm and overcome the cons of the ALS model by designing a new loss function.

In today's era, the user of a movie recommendation system is huge, thus building an efficient recommender system (ISI, 2015) is not enough, it has to be scalable also to process millions of requests. In (Kitazawa and Yui, 2018) and (Panigrahi et al., 2016) papers, authors have used the approach of SQL to solve the scalability of this problem. They have implemented various common functions as UDF in Apache Hivemall. Suggesting movies in real-time is also a very important aspect for a few applications. In (Sunny et al., 2017) paper, the author leveraged Spark to recommend TV shows channel in real-time. They have used Spark MLlib and a self-adaptive approach to create a model which is built on top of Lambda architecture.

The flip side of having an enormous amount of data is to have quality and relevant data is very tricky. The success of any machine learning-based system is directly proportional to the data we feed into it. In (Hameed et al., 2012) and (Choudhury et al., 2021) papers, the authors have provided a thorough literature review of all popular techniques of

collaborative filtering and which data extraction method to use for a given scenario.

There are several methodologies used for filtering purposes (Cai et al., 2014) such as viewer-movie-based, viewer-viewer-based, and movie-movie. We used in our approach the technique of collaborative filtering (Yang et al., 2016) based on the linear combination of viewer-movie (Kumar and Shrivindhya, 2022). One of the published papers mentions how the amalgamation of Spark with R is trending and has gained momentum in the past few years (Fadhel Aljunid and D H, 2018).

The paper (Bokde et al., 2015) explored various collaborative filtering algorithms using matrix Factorization (Mnih and Salakhutdinov, 2007). They discussed different models such as Singular Value Decomposition (SVD), Principal Component Analysis (PCA), and Probabilistic Matrix Factorization (PMF). This was a comprehensive survey of the Matrix Factorization model and addresses the challenges of Collaborative Filtering algorithms.

Along with user preference, there are other factors (like time, and demography) as well which can contribute to improving the recommendation. In (Zhang and Yang, 2020) paper, the author has incorporated a time-based collaborative filtering technique in addition to contextual user preference. The proposed method in this paper is using Apache Spark and a matrix factorization model with additional individual time-based dynamic parameters.

3 PROBLEM STATEMENT

We aim to address the problem of building a scalable system that would be efficient in handling huge loads of data. This paper discusses and implements a movie recommendation system using Apache Spark. The implementation is carried out using the MovieLens dataset, where we are using two datasets of different sizes. The first one is the MovieLens small dataset of 100k ratings from 1000 users on 1700 movies and the second one is a larger dataset including 1 million movie ratings from 6000 users on 4000 movies. We will eventually carry out the performance evaluation on the larger dataset to ensure that our system is scalable.

There are many .csv files in both the datasets such as movies.csv, ratings.csv, tags.csv, etc. Movies.csv consists of movieId, the title of the movie, and movie genres and another file ratings.csv consist of userId, ratings for the movie with a specific movieId. More details about the dataset can be found in Section 3. We intend to evaluate that our system can scale to recommend movies to users based on the similarity between the user and the movie genres. Alternating Least Squares implements collaborative filtering which seems suitable for our application. The evaluation can be done by applying the trained model to a list of movies

that the user has rated and based on that a movie would be recommended to a user.

Label	# Count
Distinct Users	283228
Movies	58098
Movies Rated by Users	53889
Genres	20

Table 1. Data Statistics

4 DATA

We plan to use the MovieLens dataset (<https://grouplens.org/datasets/movielens/latest/>). This dataset describes a 5-star rating and free-text tagging activity from MovieLens. It contains 25 million ratings and 1 million tag applications across 62423 movies. There are 20 genres of movies in this dataset. These data were created by 162k users. For this dataset, random users were included and were given unique user ids. Figure[1]

We will be using data from the following files: movies.csv, ratings.csv, links.csv, and tags.csv. Movies.csv contains the movie id, movies, and their genres while ratings.csv tells us what rating a particular user gave to a movie. Links.csv contains information about IMDB links of a particular movie. Tags.csv stores the tags given by a user to a movie.

```
+-----+-----+-----+
|userId|movieId|rating|
+-----+-----+-----+
| 1| 2986| 2.5|
| 4| 292| 3.5|
| 4| 836| 2.5|
| 10| 1093| 2.0|
| 14| 160718| 4.5|
| 19| 379| 4.0|
| 55| 1431| 1.5|
| 56| 2005| 1.0|
| 56| 3255| 1.0|
| 56| 91542| 1.5|
| 70| 736| 4.0|
| 72| 1213| 3.5|
| 81| 61248| 3.0|
| 82| 2968| 4.5|
| 88| 4226| 4.0|
| 111| 912| 5.0|
| 114| 539| 4.0|
| 134| 611| 2.5|
| 134| 4016| 4.0|
| 134| 128838| 3.0|
+-----+-----+-----+
only showing top 20 rows
```

(a) Movie Rating dataframe

```
['IMAX',
 '(no genres listed)',
 'Sci-Fi',
 'Crime',
 'Mystery',
 'Horror',
 'Children',
 'Adventure',
 'Animation',
 'Thriller',
 'Documentary',
 'Fantasy',
 'Romance',
 'War',
 'Film-Noir',
 'Comedy',
 'Western',
 'Drama',
 'Musical',
 'Action']
```

(b) All genres of movies

Figure 1. Data

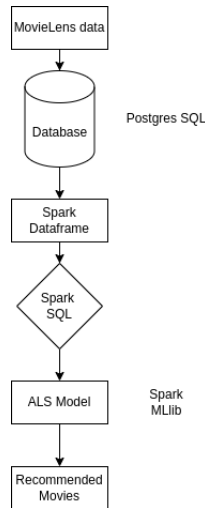


Figure 2. System Architecture

5 TECHNICAL APPROACH

This section is divided into further two sections. The first section gives a brief overview of our technical approach that we intended to implement. The second section will go in the detail of implementation, fixing issues that we ran into and other steps taken to make the project run successfully.

5.1 Overview of Approach

We have used Python for this project. We divided the task of building the data pipeline into 3 stages Figure[2]. The first stage is to build a PostgreSQL database to store the structured data of the MovieLens dataset. Spark works well with relational SQL and has specific modules like SparkSQL to work with structured data.

Secondly, we used the data-processing engine provided by Spark to process and manipulate the data. Spark uses a Resilient Distributed Dataset (RDD) which saves time in reading and writing operations and stores the data in the RAM of servers which allows quick access and accelerates the speed of analytics. Spark can store big datasets in cluster memory and can effectively run various machine learning algorithms.

Lastly, we implemented a Machine Learning model in Spark for recommending the movies. We used in-built libraries of Spark such as MLlib to help with making and training the model. Spark MLlib seamlessly integrates with other Spark Components and allows for preprocessing, training of models, and making predictions at scale on data. It has the ability to run the same ML code on a big cluster seamlessly without breaking down.

For recommendation systems, there are two main types of approaches - Collaborative Filtering and Content-Based Fil-

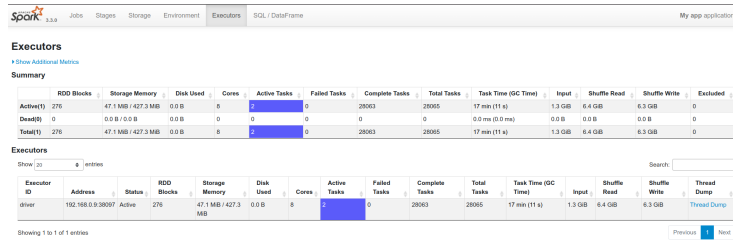
tering. The main difference between these two approaches is that the former one recommends items by looking at similar users while the latter recommends items by looking for similar content. We made use of a Collaborative Filtering based approach that takes the history of a user to make predictions on the ratings and likeness of other movies. Using a matrix factorization algorithm is the state-of-the-art solution for a sparse data problem in Collaborative Filtering. Alternating Least Squares (ALS) is also a matrix factorization algorithm and it runs itself in a parallel manner. It is implemented in Spark MLlib as well and built for large-scale collaborative filtering problems to make the predictions. More reasons for us to choose ALS were its high scalability and the ability to solve the sparseness of the ratings data. It runs its gradient descent in parallel across several partitions of the underlying training data from a cluster of machines. During the implementation, we used a package of Spark's machine learning library, called spark.mllib to operate on the data in Resilient Distributed Datasets (RDDs). We fine-tuned the parameters of this model using 5-fold cross-validation over training and test data.

5.2 Implementation Details

We started the project by setting up the PostgreSQL database on a local server and stored the MovieLens dataset in it in multiple relations. The schema of each relation corresponds to the column headers in the .csv files. We did this by using several SQL queries that are available in our GitHub repository. On the local machine, the connection is made via Unix domain sockets. Next, we established a connection of PostgreSQL with our Spark application. We established that connection by using a simple JDBC connector. From that connection, we read all the relations to Spark Dataframes which have underlying RDDs so that Spark can perform computations on the data. By doing this, we could use SparkSQL to run SQL queries on that data and also use Spark MLlib to train a recommendation model on that data. Figure[2] Next, we ran into the first major problem of using a huge dataset. The ratings DataFrame has 27 million rows and therefore, performing any sort of data exploration on it was a huge operation and we were not able to run it. It ran into a Java Heap Space - Out of Memory Error. We were able to fix these errors for all future data exploration operations (precisely actions in Spark) by amending two things -

1. Fixing configuration parameters of the JDBC Connection - Initially, we were using a basic connection we no parameters for parallelism. After reading through the documentation, we specified several parameters such as `fetchSize`, `partitionColumn`, `lowerBound`, `upperBound` and `numOfPartitions`. By setting

Movie Recommendation System

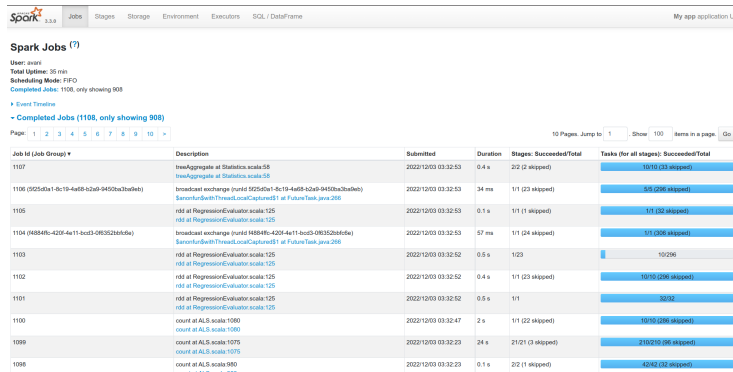


The screenshot shows the SparkUI Executors page. The top navigation bar includes links for Jobs, Stages, Storage, Environment, Executors, and SQL / DataFrame. The Executors page has a 'Summary' section with a table of executor statistics and a 'Executors' section with a detailed table of individual executors.

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Excluded
Active(1)	276	47.1 MB / 427.3 MB	0.0 B	8	0	0	28063	28065	17 min (11 s)	1.3 GB	6.4 GB	6.3 GB	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	0
Total(1)	276	47.1 MB / 427.3 MB	0.0 B	8	0	0	28063	28065	17 min (11 s)	1.3 GB	6.4 GB	6.3 GB	0

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Thread Dump
driver	192.168.0.0.38087	Active	276	47.1 MB / 427.3 MB	0.0 B	8	0	0	28063	28065	17 min (11 s)	1.3 GB	6.4 GB	6.3 GB	Thread Dump

Figure 3. Executor details from SparkUI



The screenshot shows the SparkUI Spark Jobs page. It displays a list of completed jobs with details such as Job ID, Description, Submitted time, Duration, Stages, and Tasks. The jobs are sorted by duration, and the first job is highlighted in blue.

Job ID (Job Group)	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1107	treeAggregate at Statistics.scala:58	2022/12/03 03:32:53	0.4 s	2/2 (2 skipped)	10/10 (10 skipped)
1108 (DCE5d6c1-fc19-4a69-8c4b-9453a2a7a6e4)	broadcast exchange (part=1025d6c1-fc19-4a69-8c4b-9453a2a7a6e4)	2022/12/03 03:32:53	34 ms	1/1 (23 skipped)	3/5 (23 skipped)
1105	rd at RegistratorEvaluator.scala:125	2022/12/03 03:32:53	0.1 s	1/1 (1 skipped)	1/1 (32 skipped)
1104 (93846f-4209-4e11-ba25-0832328d6d4c)	broadcast exchange (part=93846f-4209-4e11-ba25-0832328d6d4c)	2022/12/03 03:32:53	57 ms	1/1 (24 skipped)	1/1 (24 skipped)
1103	rd at RegistratorEvaluator.scala:125	2022/12/03 03:32:52	0.5 s	1/23	10/206
1102	rd at RegistratorEvaluator.scala:125	2022/12/03 03:32:52	0.4 s	1/1 (23 skipped)	10/10 (206 skipped)
1101	rd at RegistratorEvaluator.scala:125	2022/12/03 03:32:52	0.5 s	1/1	30/32
1100	count at ALS.scala:1080	2022/12/03 03:32:47	2 s	1/1 (22 skipped)	10/10 (206 skipped)
1099	count at ALS.scala:1075	2022/12/03 03:32:23	24 s	21/21 (3 skipped)	2102/5 (96 skipped)
1098	count at ALS.scala:980	2022/12/03 03:32:23	0.1 s	2/2 (1 skipped)	62/62 (22 skipped)

Figure 4. Job details from SparkUI

values for these, we made use of Spark's parallelism as we were able to fetch several rows per executor parallelly by partitioning on an integer column which greatly increased the speed. The `show()` action on the ratings DataFrame which was giving an OOM Error now worked in 2-3 seconds.

2. Modifying configuration of SparkConf given to the SparkSession - Likewise for SparkConf, we were using the most basic configuration with the jars needed for PostgreSQL. On adding more parameters, we were able to increase the number of executors, the memory for each executor, etc. which helped in running each task parallelly to increase speed and efficiency. The final configuration which we used looks like this-

```
sparkConf = SparkConf()

sparkConf.setAppName("Movie Recommender
System").set("spark.jars", "/home/
avani/UMass/Fall_2022/CS_532/Project
/postgresql-42.5.0.jar")
sparkConf.set("spark.dynamicAllocation.
enabled", "true")
sparkConf.set("spark.executor.cores",
8)
sparkConf.set("spark.dynamicAllocation.
minExecutors", "1")
sparkConf.set("spark.dynamicAllocation.
maxExecutors", "5000")
```

```
sparkConf.set("spark.executor.memory",
"32g")
sparkConf.set("spark.ui.port", "4050")
sparkConf.set("spark.memory.fraction",
0.7)

spark = SparkSession.builder.master('
local[*]').config(conf=sparkConf).
getOrCreate()
```

Listing 1. Spark Configuration for better performance

After fixing these issues, we were able to efficiently fetch data from PostgreSQL into Spark DataFrames and perform data analysis on a huge dataset in a matter of few seconds. Some statistics are displayed in Figure[1].

Then, we prepared our DataFrames to feed them into the ALS model by cleaning them. We split the data into training and test data in the ratio of 0.8 to 0.2. To test our model with various training parameters, we used ParamGrid from Spark's ML tuning library. In that, we specified multiple rank, regularization parameter and iteration values. Because of this, we were able to tune our parameters easily and 16 models were trained with all possible permutations of the values in the ParamGrid. Finally, we used a CrossValidator with 5 folds to train all the models. The best model from the 16 had the following parameters-

```
Rank: 50
MaxIter: 15
RegParam: 0.15
```

Listing 2. Best Model Parameters

We have also added from screenshots from Spark UI showing details about executors Figure[3], stages of jobs and the jobs while the model was training Figure[4]. Lastly, we tested our model and made fun predictions using it! Details about that are explored in Section[6].

6 EXPERIMENT

This section explains our experimentation process in detail.

We evaluated our model on the test set and got the predicted ratings which are shown in Figure[5]. We can observe that the predicted values of the test set are very close to the true ratings given by the users.

userId	movieId	rating	prediction
463	1088	3.5	3.2851524
580	44022	3.5	3.2771707
597	471	2.0	3.8366563
597	1580	3.0	3.569401
368	1580	3.0	2.927928
368	2366	4.0	3.1160734
368	3918	2.0	2.6714442
28	3175	1.5	2.8194048
587	3175	5.0	3.8273165
332	2366	3.5	3.4877331

Figure 5. Predicted Ratings on Test Data

6.1 Evaluation

We evaluated our best model using Root Mean Squared Error(RMSE) value. Root Mean Square Error (RMSE) is the standard deviation of the residuals (prediction errors). Residuals are a measure of how far from the regression line data points are; RMSE is a measure of how spread out these residuals are. In other words, it tells you how concentrated the data is around the line of best fit. We can also say that the RMSE score shows what is the deviation of the predictions from measured true values using Euclidean distance. For our best model, we got an RMSE score of approximately 0.87. This means that on average the model predicts 0.87 above or below the values of the original rating matrix. The model with a lower RMSE will be a better model. For our best model, we also got an RMSE score of 0.66 over the whole dataset.

$$RMSE = \sqrt{\sum_{i=1}^N \frac{(z_f - z_o)^2}{N}}$$

here,

z_f = predicted values,
 z_o = observed values,
 N = Total data points

userId	movieId	prediction	rating
9	4495	4.6397157	null
120	3379	4.609119	null
133	177593	3.6991415	null
137	132333	4.5028896	null
218	96004	4.610119	null
309	7842	4.4007373	null
367	6732	5.0505524	null
372	177593	4.2397866	null
530	25906	4.6744313	null
596	171495	4.4235716	null
604	134796	4.806554	null
122	170355	5.929523	null
193	89904	4.4845576	null
321	117531	4.5056376	null
358	170355	4.5756516	null
451	5915	4.921042	null
481	170355	3.910033	null
545	3379	4.792139	null
20	3379	4.9206986	null
93	7842	5.554637	null

Figure 6. Movies not rated by users

userId	title	userId	title
37	The Big Bus (1976)	436	Thief (1981)
37	Mulholland Dr. (1...	436	Hello, Dolly! (1969)
37	On the Beach (1959)	436	The Big Bus (1976)
37	Seve (2014)	436	Mulholland Dr. (1...
37	Dragon Ball Z: Th...	436	On the Beach (1959)
37	Jetée, La (1962)	436	Seve (2014)
37	Cosmos	436	Dragon Ball Z: Th...
37	Dune (2000)	436	Babes in Toyland ...
37	Saving Face (2004)	436	Saving Face (2004)
37	Man Bites Dog (C'...	436	Shall We Dance (1...

(a) UserId = 37

(b) UserId = 436

Figure 7. Top 10 Recommended movies

6.2 Results

We first used join in SparkSQL to get the movies that were not rated by the users. Movies that were not rated were represented by *null* in the movie rating data frame. Figure[6] shows the data frame which was created and shows the predicted ratings of each user for a particular movie.

To test out our system, we randomly selected two users and tried to predict the recommended movies from a list of movies that they didn't rate. The top 10 recommended movies can be seen in Figure[7]. These are sorted based on the predicted values of the ratings from the data frame. As seen in Figure[6], the system recommended "The Big Bus

(1976)” for the userID 37 and Thief for userID 436. These are based on the type of movies, the user has seen in the past. Since we are doing collaborative filtering, our system uses similarities between users and items simultaneously to provide recommendations.

Using the dataframe as shown in Figure[6], we can also check the predicted ratings of each movie given by a particular user. Like movieId 4495 has got a rating of 4.63 by userID 9. More use cases like recommending a particular genre of movie for a particular user can also be generated by our system seamlessly using the data frames.

Thus, this project helps users to choose a movie according to their taste. One can also try to recommend a movie that suits the taste of two or maybe more users using a cosine similarity function in the model. We could not provide these results here as it was out of scope and needed extra modifications in our model but it can be a possible extension for the future.

6.3 Performance

Our major aim in this project was to make a scalable and efficient system and for that reason we chose to work with the bigger dataset with 27 million ratings. Due to lack of computational resources on our personal laptops, it took a huge amount of time to train the model on such huge data even once, let alone with cross-validation. The time ranged from around 4-6 hours. This was also achieved after adding parallelism in the ALS model. Since that was not sufficient for us to experiment with the model and make good predictions, we also repeated the whole project with the smaller dataset so that we could train multiple models along with cross-validation to get the best model. As was expected, we were able to do in a practical amount of time with the smaller dataset with the resources at our hand. It took around 45 minutes to train 16 models with 5 folds in cross-validation. Therefore, we were able to achieve high efficiency with this dataset. The scalability aspect in our project was also successful but we could not do any experimentation with it. From the best model, we are able to make predictions in just a few nanoseconds!

7 CONCLUSION

We got to learn a lot from this project and achieved what we intended to in the beginning.

1. We learnt how to setup a PostgreSQL server on our machine and push data in it using SQL queries.
2. We got hands on experience with Spark which made our concepts learnt in class even more clear. We could see drivers and executors in action.

3. Using Spark Web UI helped a lot to understand how the jobs were being run, at what point they failed and what their DAG visualization looked like.
4. We learned about the two recommendation approaches and how to put them to use in the real world. Knowing about the ALS model was completely new and fascinating.

There can be a lot done on this in the future. For instance, with more computing resources, we can fit in an even bigger dataset to train the model. We can also make a large Param-Grid to have more choices for the best model to improve the RMSE value. Apart from this, we can combine both approaches: Collaborative and Content-based filtering to group both the users and the movies having a hybrid sort of approach. This way we will be able to recommend more similar movies to a user who liked a certain movie. We will be able to draw advantages available in both approaches, thus having a stronger recommendation model. We can also try various other Machine Learning algorithms and study their comparative results.

8 TEAM CONTRIBUTIONS

We all worked on the project together; helping each other when someone got stuck. Here is a division of the tasks-

- Avani - PostgreSQL setup, storing dataset in PostgreSQL, establishing connection between Spark and PostgreSQL, reading tables to Spark using JDBC, ALS model training, research pertaining to Spark, Documentation
- Sahil - Dataset research, Data Analysis and Exploration, Data Preprocessing, Model Testing, Evaluation, Results, Experimentation, Documentation
- Neelima - Literature Review, SparkSQL, Data extraction from HDFS to Spark RDDs, Docker implementation to host HDFS, Related work research, Documentation

REFERENCES

- (2015). Recommendation systems: Principles, methods and evaluation. *Egyptian Informatics Journal*, 16(3):261–273.
- Bokde, D., Girase, S., and Mukhopadhyay, D. (2015). Matrix factorization model in collaborative filtering algorithms: A survey. *Procedia Computer Science*, 49:136–146. Proceedings of 4th International Conference on Advances in Computing, Communication and Control (ICAC3’15).
- Cai, Y., Leung, H.-f., Li, Q., Min, H., Tang, J., and Li, J. (2014). Typicality-based collaborative filtering recommendation. *IEEE Transactions on Knowledge and Data Engineering*, 26(3):766–779.

- Choudhury, S. S., Mohanty, S. N., and Jagadev, A. K. (2021). Multimodal trust based recommender system with machine learning approaches for movie recommendation. *International Journal of Information Technology*, 13(2):475–482.
- Fadhel Aljunid, M. and D H, M. (2018). Movie recommender system based on collaborative filtering using apache spark.
- Hameed, M. A., Al Jadaan, O., and Ramachandram, S. (2012). Collaborative filtering based recommendation system: A survey. *International Journal on Computer Science and Engineering*, 4(5):859.
- Jeong, H. and CHA, K. J. (2019). An efficient mapreduce-based parallel processing framework for user-based collaborative filtering. *Symmetry*, 11(6).
- Kitazawa, T. and Yui, M. (2018). Query-based simple and scalable recommender systems with apache hivemall. RecSys '18, page 502–503, New York, NY, USA. Association for Computing Machinery.
- Kumar, D. and ShriVindhya, A. (2022). *A Novel Approach for Movie Suggestion System with Empirical Risk Minimization and Decision Tree Algorithms*.
- Mnih, A. and Salakhutdinov, R. R. (2007). Probabilistic matrix factorization. In Platt, J., Koller, D., Singer, Y., and Roweis, S., editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc.
- Panigrahi, S., Lenka, R. K., and Stitipragyan, A. (2016). A hybrid distributed collaborative filtering recommender engine using apache spark. *Procedia Computer Science*, 83:1000–1006. The 7th International Conference on Ambient Systems, Networks and Technologies (ANT 2016) / The 6th International Conference on Sustainable Energy Information Technology (SEIT-2016) / Affiliated Workshops.
- Sunny, B. K., Janardhanan, P. S., Francis, A. B., and Murali, R. (2017). Implementation of a self-adaptive real time recommendation system using spark machine learning libraries. In *2017 IEEE International Conference on Signal Processing, Informatics, Communication and Energy Systems (SPICES)*, pages 1–7.
- Xie, L., Zhou, W., and Li, Y. (2017). Application of improved recommendation system based on spark platform in big data analysis. *Cybernetics and Information Technologies*, 16(6):245–255.
- Yang, C., Yu, X., Liu, Y., Nie, Y., and Wang, Y. (2016). Collaborative filtering with weighted opinion aspects. *Neurocomput.*, 210(C):185–196.
- Zhang, J. and Yang, D. (2020). Time-aware parallel collaborative filtering movie recommendation based on spark. *International Journal of Embedded Systems*, 13(4):449–458.