MOVIES RECOMMENDATION SYSTEM

BIA 678 – BIG DATA TECHNOLOGIES

BY ANKITA RAMVIR SINGH YADAV

TABLE OF CONTENTS

INTRODUCTION

A recommendation system helps in predicting the rating or the preference a user will give to a product. It acts as a subclass of the information filtering system. Right from the e-commerce industry to online advertisement everyone suggests products to the user, and it is a thing that is unavoidable in today's time. When a person visits an e-commerce site they are being suggested with products in the form of "customers who bought this also bought this", basically the sites try suggesting items which they feel the user will like and thus purchase based on some analysis done at their end. Similarly, advertisements advertised to the users while they browse the internet to a certain extent try matching the user's area of interest.

The companies have been able to predict the choices of the customer based on the data that is provided by the customer itself. So, whenever a customer likes a certain product or dislikes it, the data gets stored in the company's database, even if a customer visits a product page the data get stored in the form of the number of times a particular person has viewed an item, as well as when the customer rates the product, the data gets stored. So, a variety of data in huge amount is gathered by companies which help them in building a recommendation system. Companies are being able to understand customer preferences, their likes and dislike with the help of this data thereby offering recommendations.

There are two types of recommendation systems that provide a useful and meaningful recommendation, but their way of working is slightly different. They are as mentioned below:

a. Collaborative-filtering engine
b. Content-based filtering engine

a. Collaborative-filtering engine

Suppose there are two users, A and B. Both A and B have seen five same movies and have given similar ratings to those five movies. So, they are being identified as similar users. Now suppose A sees a movie that B has not seen yet, so next time the website shall suggest the same movie to B stating, “similar users have liked this movie”. So, in a collaborative filtering engine, recommendations are based on user similarities.

b. Content-based filtering engine

In a content-based filtering engine, the recommendations are based on understanding the contents of the item. Suppose user A has watched a movie that belongs to the comedy genre, released in 2015, with actors A and B in the English language. So, the next movie that shall be suggested to user A shall be based on similar features of the movie user A has already watched. So, as we can see, the contents are deciding which movie to be suggested next. This type of filtering wherein content plays an important role in providing recommendations is known as a content-based recommendation system.

There are two types of ratings which helps in building a recommendation system:

a. Explicit ratings
b. Implicit ratings

a. Explicit ratings

Explicit ratings are the ones that are quite direct. For example, when a user rates a movie on a scale of 1 to 5, likes or dislikes a movie, or provides any kind of rating directly which passes a message that the user liked or disliked a movie is known as explicit ratings. In this type of rating, the user very clearly states their preference towards the product.

b. Implicit ratings

Implicit ratings are the indirect ones. The information about liking or disliking is not directly provided in the form of ratings. Implicit ratings are based on user behavior. The user behavior is tracked and based on the frequency of their behavior a decision regarding their likes and dislikes is made. For example, suppose a user has watched a romantic movie 20 times and an action movie 1 time so based on this it would be concluded that the romantic movie has a high confidence score, and the action movie has a low confidence score for this customer. So, the movies recommended to this user will belong to the romantic genre.

STEPS TO BUILD A RECOMMENDATION SYSTEM

The steps that were undertaken as a part of my study to build a recommendation system were data understanding, data preparation, modeling, and performance evaluation.

a. Data Understanding

In this step, data gathering took place and certain descriptive statistics were performed.

b. Data Preparation

The data was cleaned and prepared for the modeling stage so that analysis can be conducted on the data and a model can be built successfully.

c. Modeling

In this step, the alternating least square (ALS) algorithm was used to create a model.

d. Performance Evaluation

The performance evaluation of the model took place by using the root mean square error (RMSE) metric.

DATA UNDERSTANDING

The dataset was downloaded from MovieLens, a movie recommendation service available on grouplens. It consists of a 5-star rating and free-text tagging activity from MovieLens. There are 100836 ratings, 3683 tag applications, 9742 movies, and 610 users in the dataset. The dataset consists of two tables, namely ratings and movies.

The ratings table consists of userId, movie, rating, and timestamp columns. The movies table consists of movieId, title, and genre column. The minimum number of ratings given by a user is 20 and the maximum is 2698. The average number of ratings given by a user is 165. The minimum number of ratings received by a movieId is 1 and the maximum is 329. The average number of ratings received by a movieId is 10.

The data sparsity shows what percentage of the matrix is empty. It is calculated by the below formula:

$$Sparsity = 1 - \frac{Number\ of\ Ratings\ in\ Matrix}{(Number\ of\ Users) \times (Number\ of\ Movies)}$$

The data sparsity of the ratings table is 98.30%. This is understandable because not every user shall watch all the movies and rate them. Hence, the sparsity of the data is high.

DATA PREPARATION

The data preparation consisted of four steps. The first step consisted of checking whether the dataframe was in a row-based format wherein each userId rates a movieId and the rating is mentioned. The dataset met this requirement. The next step in data preparation included checking whether the userId, movieid are in the integer format and the requirement was met by the dataset.

The third step consisted of dropping columns from tables that are not required for the analysis. So, the column timestamp from the ratings table was dropped. Finally, the ratings and movies table were joined using the left join. The dataset finally consisted of movieId, userId, rating, title, and genres columns.

MODELING

For a recommendation system, a collaborative filtering engine was used, thereby using the alternating least square (ALS) model for the analysis. The ratings of the users are clearly mentioned in the dataset. These ratings are called explicit ratings and hence the alternating least square is used. Also, the ALS model works well with the sparse matrix and since our dataset sparsity was high, the ALS model was a perfect fit for building a model for the recommendation system. The alternating least square algorithm uses non-negative factorization to predict how users will rate movies they have not seen before. A non-negative factorization is used since the factors cannot be negative as they represent ratings. Ratings are positive and negative ratings do not lead to any interpretation.

The alternating least square algorithm consists of three steps, namely, dividing the original matrix into two matrices, performing maximum iterations to get the best root mean squared error, and then filling predicting the ratings for the movies the user has not seen before.

In the first step, the original matrix is divided into two different matrices. These two matrices when multiped back together forms the original matrix. Suppose the shape of the original matrix in (mxn), then the shape of the other two matrices shall be (mxr) and (rxn). The r is the number of latent features. The latent features are called a rank. The rank is the dimensions of factor matrices that do not match. This rank helps in meaningful grouping or categorization of the movies on how similar or different they are.

To have the closest approximation the ALS model fills the two matrices with random numbers. Then it keeps the original matrix and the first matrix constant, and the changes are made in the second matrix. Then the result is evaluated that when multiplied together is it forming the original matrix. If that is not the case, then the original matrix and second matrix are kept constant, and changes are made in the first matrix. This process is continued until the closest approximation of the original matrix is formed. This process is called iterations. This represents the number of maxIter that is needed while running the model. The closest approximation to the original matrix when the two-factor matrices are multiplied together is measured with the root means squared error (RMSE) metric.

The last step in the ALS model consists of predictions. The empty cells of the original matrix are filled with values based on how each user has behaved in the past relative to the behavior of similar users.

The steps performed for building an alternating least square model is as follows:

a. Splitting the dataset into training and test dataset

   The dataset was divided into training data and test data in the ratio of 0.8 and 0.2 respectively.

b. Initializing the ALS model without hyperparameters

   In this step, the ALS model was initialized without the hyperparameters. The parameters set at this step were userCol which means the name of the column which has the user id. The userCol was kept as “userId”. The itemcol is the name of the column which has the product id in our case the movie id’s, it was kept as “movieid”. The ratingCol is the name of the column which has the ratings, kept as “rating”. The next parameter was nonnegative which was kept as True since the ALS model does non-negative matrix factorization. The implicitPrefs parameter was kept as False since we are using explicit ratings for building our model, and finally, coldStartStrategy which was

kept as “drop”. The coldStartStrategy parameter implies that if there is a userId whose all ratings came in the test dataset for predictions and the training dataset does not contain any of his records then such userId should be dropped from the testing dataset.

c. Tuning hyperparameters through grid search

The next step in the modeling process involved hyperparameters tuning through ParamGridBuilder. The ParamGridBuilder tells spark all the hyperparameter values that the model needs to try. The three hyperparameters that were tuned are rank, maxIter, and regParam. The rank is the latent features and the maxIter is the maximum number of iterations the model needs to perform. The regParam is the regularization parameter wherein it is a number that is added to an error metric to keep the algorithm from converging too quickly and overfitting to the training data. The rank parameter was set as (5, 10, 15), the maxIter was set as (10, 15, 20) and the regParam was set as (0.01, 0.05, 0.1). After conducting parameter tuning the most appropriate values for rank, maxIter and regParam came as 10, 20, and 0.1 respectively.

d. Performing 5-fold cross validation

5-fold cross-validation was performed to find the best model. The cross validator fits the model to portions of training data called folds and then generates predictions for each representative holdout portion to see how it performs. The training data was divided into 5-folds and the best fit model was found.

e. Fit the best model on training dataset

The best model is extracted from the above dataset and fitted to the training dataset.

f. Generating predictions on test dataset

After the model is fitted, the predictions are provided for all the userId.

g. Evaluating performance through root mean square error

The model is evaluated through the root mean squared error (RMSE) metric.

RESULTS AND PERFORMANCE EVALUATION

The model performance is evaluated using the root mean squared error metric. The RMSE metric tells us that on average how far the predicted value is from the corresponding actual value. One thing that is kept in mind while calculating the RMSE is that only the original values are considered and not the predicted values during calculation. The RMSE obtained for the model is 0.87. It means that the predictions are either 0.87 above or below the original rating.

For generating the results two setups were used, Spark on databricks and Amazon Web services (AWS). The PySpark programming language was used on both platforms.

To perform the modeling on AWS, Amazon Simple Storage Service (S3) and Amazon Elastic MapReduce (EMR) were used. S3 provides easy-to-use object storage to store and retrieve any amount of data in the cloud and EMR lets you perform big data tasks such as web indexing, data mining, and log file analysis.

The dataset was divided into 4 parts, part 1, part 2, part 3, and part 4 consisting of 25%, 50%, 75%, and 100% of the data respectively. The model was run on each dataset using spark on databricks and amazon web services.

On spark, by databricks the time taken to run 25% of the dataset was 197.27 seconds with the RMSE of 1.05, for 50% of the data 209.52 seconds were taken with the RMSE of 0.96, with 75% of the data it was 226.90 seconds with 0.91 RMSE and for 100% of the dataset it took 237.77 seconds with 0.87 RMSE.

When the program was run on AWS the 25% of the data took 11.92 seconds to run with the RMSE of 1.07, 50% took 13.47 seconds with the RMSE of 0.94, 75% took 19.18 seconds with the RMSE of 0.91 and 100% of the dataset took 22.38 seconds with the RMSE of 0.88.

The dataset was run on spark and parallelization were applied. As the parallelization was increased in spark the time to run the small dataset (25%) kept increasing from 26.95 seconds to 30.69 seconds, to 36.65 seconds and 41.68 seconds respectively. However, when the same dataset was run on AWS and the number of instances was increased from 1 to 4 the time taken was less as the number of instances increased. The 25% data with just 1 instance took 26.09 seconds, with 2 instances took 18.01 seconds, 3 instances took 16.35 seconds, and 4 instances took 10.73 seconds. The average RMSE on both spark and AWS were 1.05 and 1.04 respectively.

When 100% of the dataset was run on spark with parallelization implemented then the time taken was 125.85 seconds, 109.13 seconds, 102.65 seconds, and 99.86 seconds. Similarly, when 100% dataset was running on AWS with 1 instance it took 33.39 seconds, with 2 instances it took 22.75 seconds, with 3 instances it took 16.10 seconds, and with 4 instances it took 10.96 seconds. The average RMSE for 100% of the dataset both on Spark and AWS were 0.88.

ANALYSIS

The analysis helped us find the impact of the scale of data on quality (RMSE) and on-time performance.

The measure of the impact of the scale of data on quality (RSME) showed that as scale increases the RMSE value decreases. The increase in the scale of data improved the quality of prediction. Both spark databricks and AWS showed a similar impact of the scale of data on the quality (RMSE).

The measure of the impact of the scale of data on-time performance showed that as the scale of data increased the time increased. With the increase in scale, the time increased in both spark databricks and AWS.

The measure of the impact of parallel computation on performance as scale increases compared to non-parallel execution showed that in the case of the small dataset in databricks when parallelization was increased the time increased however for the large dataset when parallelization increased the time decreased. In AWS, both for small and large datasets the time decreased as the number of instances increased.

RECOMMENDATIONS

The top five recommendations for all userIDs were predicted. A single userId prediction was compared to the movies the userId has originally watched. The movies already seen by the userId and given the rating of 4 and 5 consisted of drama, comedy, children, and the userId were recommended similar movies through the alternating least square algorithm.

CONCLUSION

The Recommendation systems can successfully generate recommended movies with low RMSE. The final RMSE are similar for both, however, cloud computing has a better performance than local, especially for the runtime.

As the scale increases the performance time increases in local as well as the cloud. As the scale increases the RMSE quality improves in local as well as the cloud.

The Performance time decreases for large datasets when instances and parallelization increase in local as well as the cloud. The Performance time increases for small datasets when parallelization increases in local while in the cloud the performance time decreases.

REFERENCES

- https://towardsdatascience.com/build-recommendation-system-with-pyspark-using-alternating-least-squares-als-matrix-factorisation-ebe1ad2e7679
- https://medium.com/analytics-vidhya/movie-recommendation-with-collaborative-filtering-in-pyspark-8385dccecfca
- https://spark.apache.org/docs/latest/mllib-collaborative-filtering.html
- https://towardsdatascience.com/prototyping-a-recommender-system-step-by-step-part-2-alternating-least-square-als-matrix-4a76c58714a1

APPENDIX

Databricks notebook source

 # IMPORTING LIBRARIES

```
from pyspark import SparkContext
from pyspark.sql import SparkSession, SQLContext, Row
from pyspark.sql.functions import col, min, max, avg
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.recommendation import ALS
```

```
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

import time

from pyspark.sql.types import *

import pandas as pd

import matplotlib as mpl

import matplotlib.pyplot as plt

import seaborn as sns

import warnings; warnings.filterwarnings(action='once')

from matplotlib.axes import Axes


# COMMAND ----------
```

 # DATA UNDERSTANDING

```
# COMMAND ----------


# LOADING THE DATASET

ratings = spark.read.format("csv").option("inferSchema", "true").option("header",

"true").load("/FileStore/tables/ratings.csv")

movies = spark.read.format("csv").option("inferSchema", "true").option("header",

"true").load("/FileStore/tables/movies.csv")
```

```
# COMMAND ----------


# printing the column names and the top 5 rows of the column

print(ratings.columns)

print(ratings.show(5))


# COMMAND ----------


# printing the column names and the top 5 rows of the column

print(movies.columns)

print(movies.show(5))


# COMMAND ----------


# calculating sparsity

numerator = ratings.select("rating").count()

num_users = ratings.select("userId").distinct().count()

num_movies = ratings.select("movieId").distinct().count()

print(num_users)

print(num_movies)

denominator = num_users * num_movies

sparsity = (1.0 - (numerator *1.0)/denominator)*100
```

```
# COMMAND ----------

# printing sparsity

print(sparsity)

# COMMAND ----------

# distinct userId in ratings dataset

ratings.select("userId").distinct().count()

# COMMAND ----------

# distinct movieId in ratings dataset

ratings.select("movieId").distinct().count()

# COMMAND ----------

# number of ratings in the ratings dataset

ratings.select("rating").count()

# COMMAND ----------
```

```
# number of ratings given by each user in ratings dataset

userId_rating_count = ratings.groupBy("userId").count()

userId_rating_count.show()


# COMMAND ----------


# number of ratings given by each userId

userId_rating_count.sort("count", ascending = True).show()


# COMMAND ----------


# minimum count of rating by a userId

userId_rating_count.select(min("count")).show()


# COMMAND ----------


# maximum count of rating by a userId

userId_rating_count.select(max("count")).show()


# COMMAND ----------
```

```
# average count of rating by a userId

userId_rating_count.select(avg("count")).show()


# COMMAND ----------


# number of ratings received by a movie in ratings dataset

movieId_rating_count = ratings.groupBy("movieId").count()

movieId_rating_count.show()


# COMMAND ----------


# sorting the ratings received by a movie in ratings dataset

movieId_rating_count.sort("count", ascending = True).show()


# COMMAND ----------


# minimum count of rating by a movieId

movieId_rating_count.select(min("count")).show()


# COMMAND ----------


# maximum count of rating by a movieId
```

```
movieId_rating_count.select(max("count")).show()


# COMMAND ----------


# average count of rating by a movieId

movieId_rating_count.select(avg("count")).show()


# COMMAND ----------


# total number of movieId in the movies dataset

movies.select("movieId").count()


# COMMAND ----------
```

 # DATA PREPARATION

```
# COMMAND ----------


# checking the ratings dataset format

print(ratings.show())


# COMMAND ----------
```

```
# checking data type of ratings columns

ratings.printSchema()


# checking data type of movies columns

movies.printSchema()


# COMMAND ----------


# checking data type of movies columns

movies.printSchema()


# COMMAND ----------


# checking if it dropped succesfully

ratings.show()


# COMMAND ----------


# joining ratings and movies data set

movies_ratings_data = ratings.join(movies, on='movieId', how='leftouter')
```

COMMAND ----------

final dataset

movies_ratings_data.show()

COMMAND ----------

 # MODELING

COMMAND ----------

using 100% of the dataset

sample_size = [100836]

creating empty lists

rank_list = []

maxIter_list = []

regParam_list = []

time_list = []

RMSE_list = []

```
# creating a for loop for modeling

for s in sample_size:


    # taking sample through takeSample function

    sample_data = movies_ratings_data.rdd.takeSample(False, s, 42)


    # creating spark dataframes of sample_data

    sample = sqlContext.createDataFrame(sample_data)


    # recording the start time

    ALS_start = time.time()


    # spliting the dataset

    (training_data, test_data) = sample.randomSplit([0.8, 0.2], seed = 42)


    # building a generic ALS model without hyperparameters

    als = ALS(userCol="userId", itemCol="movieId", ratingCol="rating", nonnegative = True, implicitPrefs
= False,\

        coldStartStrategy = "drop")


    # adding hyperparameters and their respective values to param_grid for parameter tuning
```

```
param_grid= ParamGridBuilder().addGrid(als.rank, [5, 10, 15]).addGrid(als.maxIter, [10, 15, 20]).addGrid(als.regParam, [0.01, 0.05, 0.1]).build()

# defining evaluator as RMSE
evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating", predictionCol="prediction")

# building cross validation using CrossValidator
cv = CrossValidator(estimator=als, estimatorParamMaps=param_grid, evaluator=evaluator, numFolds=5)

# fitting cross validator on the training data
model = cv.fit(training_data)

# extracting the best combination of values (best model) from cross validation
best_model = model.bestModel

# generate test_data predictions
test_predictions = best_model.transform(test_data)

# evaluating the test_predictions using RMSE
rmse = evaluator.evaluate(test_predictions)
```

```
    # recording the end time
    ALS_end = time.time()

    # appending the rank, maxIter, and regParam in the list
    rank_list.append((s, best_model.rank))
    maxIter_list.append((s, best_model._java_obj.parent().getMaxIter()))
    regParam_list.append((s, best_model._java_obj.parent().getRegParam()))

    # calculating the total_time
    total_time = ALS_end - ALS_start

    # appending time and rmse in the list
    time_list.append((s, total_time))
    RMSE_list.append((s, rmse))

# setting the schema for time and rmse
cSchema_time = StructType([StructField("size", IntegerType()),StructField("total_time", FloatType())])
cSchema_rmse = StructType([StructField("size", IntegerType()),StructField("rmse", FloatType())])

# creating the dataframe for time and rmse
df_time = spark.createDataFrame(time_list, schema = cSchema_time)
df_rmse = spark.createDataFrame(RMSE_list, schema = cSchema_rmse)
```

```
# displaying the time and rmse dataframe

df_time.show()

df_rmse.show()


# COMMAND ----------


# MEASURING THE IMPACT OF SCALE ON PERFORMANCE AND QUALITY


# COMMAND ----------


# distributing data into 25% size

sample_size = [25209]


# creating empty list

time_list_1 = []

RMSE_list_1 = []


# creating a for loop for modeling

for s in sample_size:


  # taking sample through takeSample function
```

```
    sample_data = movies_ratings_data.rdd.takeSample(False, s, 42)


    # creating spark dataframes of sample_data
    sample = sqlContext.createDataFrame(sample_data)


    # recording the start time
    ALS_start = time.time()


    # spliting the dataset
    (training_data, test_data) = sample.randomSplit([0.8, 0.2], seed = 42)


    # building a generic ALS model without hyperparameters
    als = ALS(userCol="userId", itemCol="movieId", ratingCol="rating", rank = 10, maxIter = 20, regParam
= 0.1, nonnegative = True,\
            implicitPrefs = False, coldStartStrategy = "drop")


    # fitting cross validator on the training data
    model = als.fit(training_data)


    # generate test_data predictions
    test_predictions = model.transform(test_data)
```

```
    # telling spark to evaluate predictions
    evaluator = RegressionEvaluator(metricName = "rmse", labelCol = "rating", predictionCol = "prediction")

    # evaluating the test_predictions using RMSE
    rmse = evaluator.evaluate(test_predictions)

    # recording the end time
    ALS_end = time.time()

    # calculating the total_time
    total_time = ALS_end - ALS_start

    # appending time and rmse in the list
    time_list_1.append((s, total_time))
    RMSE_list_1.append((s, rmse))

print(time_list_1)
print(RMSE_list_1)

# COMMAND ----------
```

```
# distributing data into 50% size

sample_size = [50418]


# creating empty list

time_list_2 = []

RMSE_list_2 = []


# creating a for loop for modeling

for s in sample_size:


    # taking sample through takeSample function

    sample_data = movies_ratings_data.rdd.takeSample(False, s, 42)


    # creating spark dataframes of sample_data

    sample = sqlContext.createDataFrame(sample_data)


    # recording the start time

    ALS_start = time.time()


    # spliting the dataset

    (training_data, test_data) = sample.randomSplit([0.8, 0.2], seed = 42)
```

```
    # building a generic ALS model without hyperparameters
    als = ALS(userCol="userId", itemCol="movieId", ratingCol="rating", rank = 10, maxIter = 20, regParam
= 0.1, nonnegative = True,\
            implicitPrefs = False, coldStartStrategy = "drop")

    # fitting cross validator on the training data
    model = als.fit(training_data)

    # generate test_data predictions
    test_predictions = model.transform(test_data)

    # telling spark to evaluate predictions
    evaluator = RegressionEvaluator(metricName = "rmse", labelCol = "rating", predictionCol =
"prediction")

    # evaluating the test_predictions using RMSE
    rmse = evaluator.evaluate(test_predictions)

    # recording the end time
    ALS_end = time.time()

    # calculating the total_time
```

```
    total_time = ALS_end - ALS_start

    # appending time and rmse in the list
    time_list_2.append((s, total_time))
    RMSE_list_2.append((s, rmse))

print(time_list_2)
print(RMSE_list_2)

# COMMAND ----------

# distributing data into 75% size
sample_size = [75627]

# creating empty list
time_list_3 = []
RMSE_list_3 = []

# creating a for loop for modeling
for s in sample_size:

    # taking sample through takeSample function
```

```
    sample_data = movies_ratings_data.rdd.takeSample(False, s, 42)


    # creating spark dataframes of sample_data
    sample = sqlContext.createDataFrame(sample_data)


    # recording the start time
    ALS_start = time.time()


    # spliting the dataset
    (training_data, test_data) = sample.randomSplit([0.8, 0.2], seed = 42)


    # building a generic ALS model without hyperparameters
    als = ALS(userCol="userId", itemCol="movieId", ratingCol="rating", rank = 10, maxIter = 20, regParam
= 0.1, nonnegative = True,\
            implicitPrefs = False, coldStartStrategy = "drop")


    # fitting cross validator on the training data
    model = als.fit(training_data)


    # generate test_data predictions
    test_predictions = model.transform(test_data)
```

```
    # telling spark to evaluate predictions
    evaluator = RegressionEvaluator(metricName = "rmse", labelCol = "rating", predictionCol = "prediction")

    # evaluating the test_predictions using RMSE
    rmse = evaluator.evaluate(test_predictions)

    # recording the end time
    ALS_end = time.time()

    # calculating the total_time
    total_time = ALS_end - ALS_start

    # appending time and rmse in the list
    time_list_3.append((s, total_time))
    RMSE_list_3.append((s, rmse))

print(time_list_3)
print(RMSE_list_3)

# COMMAND ----------
```

```
# distributing data into 100% size

sample_size = [100836]


# creating empty list

time_list_4 = []

RMSE_list_4 = []


# creating a for loop for modeling

for s in sample_size:


    # taking sample through takeSample function

    sample_data = movies_ratings_data.rdd.takeSample(False, s, 42)


    # creating spark dataframes of sample_data

    sample = sqlContext.createDataFrame(sample_data)


    # recording the start time

    ALS_start = time.time()


    # spliting the dataset

    (training_data, test_data) = sample.randomSplit([0.8, 0.2], seed = 42)
```

```
    # building a generic ALS model without hyperparameters
    als = ALS(userCol="userId", itemCol="movieId", ratingCol="rating", rank = 10, maxIter = 20, regParam
= 0.1, nonnegative = True,\
            implicitPrefs = False, coldStartStrategy = "drop")

    # fitting cross validator on the training data
    model = als.fit(training_data)

    # generate test_data predictions
    test_predictions = model.transform(test_data)

    # telling spark to evaluate predictions
    evaluator = RegressionEvaluator(metricName = "rmse", labelCol = "rating", predictionCol =
"prediction")

    # evaluating the test_predictions using RMSE
    rmse = evaluator.evaluate(test_predictions)

    # recording the end time
    ALS_end = time.time()

    # calculating the total_time
```

```
    total_time = ALS_end - ALS_start

    # appending time and rmse in the list
    time_list_4.append((s, total_time))
    RMSE_list_4.append((s, rmse))

print(time_list_4)
print(RMSE_list_4)

# COMMAND ----------

# MEASURING THE IMPACT OF PARALLEL AND NON_PARALLEL COMPUTATION AS SCALE INCREASES ON
TIME

# COMMAND ----------

# distributing data into 100% size with numBlocks 1
sample_size = [100836]

# creating empty list
time_list_1 = []
RMSE_list_1 = []
```

```
# creating a for loop for modeling
for s in sample_size:

    # taking sample through takeSample function
    sample_data = movies_ratings_data.rdd.takeSample(False, s, 42)

    # creating spark dataframes of sample_data
    sample = sqlContext.createDataFrame(sample_data)

    # recording the start time
    ALS_start = time.time()

    # spliting the dataset
    (training_data, test_data) = sample.randomSplit([0.8, 0.2], seed = 42)

    # building a generic ALS model without hyperparameters
    als = ALS(numUserBlocks = 1, numItemBlocks = 1, userCol="userId", itemCol="movieId",
ratingCol="rating", rank = 10, maxIter = 20, regParam = 0.1, nonnegative = True,\
            implicitPrefs = False, coldStartStrategy = "drop")

    # fitting cross validator on the training data
```

```
    model = als.fit(training_data)


    # generate test_data predictions
    test_predictions = model.transform(test_data)


    # telling spark to evaluate predictions
    evaluator = RegressionEvaluator(metricName = "rmse", labelCol = "rating", predictionCol =
"prediction")


    # evaluating the test_predictions using RMSE
    rmse = evaluator.evaluate(test_predictions)


    # recording the end time
    ALS_end = time.time()


    # calculating the total_time
    total_time = ALS_end - ALS_start


    # appending time and rmse in the list
    time_list_1.append((s, total_time))
    RMSE_list_1.append((s, rmse))
```

```
print(time_list_1)

print(RMSE_list_1)


# COMMAND ----------


# distributing data into 100% size with 2=numBlocks 2

sample_size = [100836]


# creating empty list

time_list_2 = []

RMSE_list_2 = []


# creating a for loop for modeling

for s in sample_size:


    # taking sample through takeSample function

    sample_data = movies_ratings_data.rdd.takeSample(False, s, 42)


    # creating spark dataframes of sample_data

    sample = sqlContext.createDataFrame(sample_data)


    # recording the start time
```

```
    ALS_start = time.time()


    # spliting the dataset
    (training_data, test_data) = sample.randomSplit([0.8, 0.2], seed = 42)


    # building a generic ALS model without hyperparameters
    als = ALS(numUserBlocks = 2, numItemBlocks = 2, userCol="userId", itemCol="movieId", ratingCol="rating", rank = 10, maxIter = 20, regParam = 0.1, nonnegative = True,\
            implicitPrefs = False, coldStartStrategy = "drop")


    # fitting cross validator on the training data
    model = als.fit(training_data)


    # generate test_data predictions
    test_predictions = model.transform(test_data)


    # telling spark to evaluate predictions
    evaluator = RegressionEvaluator(metricName = "rmse", labelCol = "rating", predictionCol = "prediction")


    # evaluating the test_predictions using RMSE
    rmse = evaluator.evaluate(test_predictions)
```

```
    # recording the end time
    ALS_end = time.time()

    # calculating the total_time
    total_time = ALS_end - ALS_start

    # appending time and rmse in the list
    time_list_2.append((s, total_time))
    RMSE_list_2.append((s, rmse))

print(time_list_2)
print(RMSE_list_2)

# COMMAND ----------

# distributing data into 100% size with numBlocks 3
sample_size = [100836]

# creating empty list
time_list_3 = []
RMSE_list_3 = []
```

```
# creating a for loop for modeling
for s in sample_size:

    # taking sample through takeSample function
    sample_data = movies_ratings_data.rdd.takeSample(False, s, 42)

    # creating spark dataframes of sample_data
    sample = sqlContext.createDataFrame(sample_data)

    # recording the start time
    ALS_start = time.time()

    # spliting the dataset
    (training_data, test_data) = sample.randomSplit([0.8, 0.2], seed = 42)

    # building a generic ALS model without hyperparameters
    als = ALS(numUserBlocks = 3, numItemBlocks = 3, userCol="userId", itemCol="movieId",
ratingCol="rating", rank = 10, maxIter = 20, regParam = 0.1, nonnegative = True,\
            implicitPrefs = False, coldStartStrategy = "drop")

    # fitting cross validator on the training data
```

```
    model = als.fit(training_data)


    # generate test_data predictions
    test_predictions = model.transform(test_data)


    # telling spark to evaluate predictions
    evaluator = RegressionEvaluator(metricName = "rmse", labelCol = "rating", predictionCol =
"prediction")


    # evaluating the test_predictions using RMSE
    rmse = evaluator.evaluate(test_predictions)


    # recording the end time
    ALS_end = time.time()


    # calculating the total_time
    total_time = ALS_end - ALS_start


    # appending time and rmse in the list
    time_list_3.append((s, total_time))
    RMSE_list_3.append((s, rmse))
```

```
print(time_list_3)

print(RMSE_list_3)


# COMMAND ----------


# distributing data into 100% size with numBlocks 4

sample_size = [100836]


# creating empty list

time_list_4 = []

RMSE_list_4 = []


# creating a for loop for modeling

for s in sample_size:


  # taking sample through takeSample function

  sample_data = movies_ratings_data.rdd.takeSample(False, s, 42)


  # creating spark dataframes of sample_data

  sample = sqlContext.createDataFrame(sample_data)


  # recording the start time
```

```
    ALS_start = time.time()


    # spliting the dataset
    (training_data, test_data) = sample.randomSplit([0.8, 0.2], seed = 42)


    # building a generic ALS model without hyperparameters
    als = ALS(numUserBlocks = 4, numItemBlocks = 4, userCol="userId", itemCol="movieId", ratingCol="rating", rank = 10, maxIter = 20, regParam = 0.1, nonnegative = True,\
            implicitPrefs = False, coldStartStrategy = "drop")


    # fitting cross validator on the training data
    model = als.fit(training_data)


    # generate test_data predictions
    test_predictions = model.transform(test_data)


    # telling spark to evaluate predictions
    evaluator = RegressionEvaluator(metricName = "rmse", labelCol = "rating", predictionCol = "prediction")


    # evaluating the test_predictions using RMSE
    rmse = evaluator.evaluate(test_predictions)
```

```
    # recording the end time
    ALS_end = time.time()

    # calculating the total_time
    total_time = ALS_end - ALS_start

    # appending time and rmse in the list
    time_list_4.append((s, total_time))
    RMSE_list_4.append((s, rmse))

print(time_list_4)
print(RMSE_list_4)

# COMMAND ----------

# RECOMMENDATIONS

# COMMAND ----------

# generate 5 recommendations for all users
ALS_recommendations = model.recommendForAllUsers(5)
```

```
# COMMAND ----------


# showing the recommendations

ALS_recommendations.show(5)


# COMMAND ----------


# creating a temporary table

ALS_recommendations.registerTempTable('ALS_recs_temp')


# COMMAND ----------


# selecting data from temporary table

new_recs = spark.sql("SELECT userId,\

                      movieIds_and_ratings.movieId AS movieId,\

                      movieIds_and_ratings.rating AS prediction\

                FROM ALS_recs_temp\

                LATERAL VIEW explode(recommendations) exploded_table\

                AS movieIds_and_ratings")


# COMMAND ----------
```

```
# showing records of the newtable created

new_recs.show(5)


# COMMAND ----------


# joining the new records table with the movies table using left join

new_recs_2 = new_recs.join(movies, ['movieId'], "left")

new_recs_2.show(5)


# COMMAND ----------


# joining the table created above with the ratings table and setting the filter to suggest movies which

the userId has not seen

final_recs = new_recs_2.join(ratings, ["userId", 'movieId'], "left").filter(ratings.rating.isNull())


# COMMAND ----------


# showing the final recommendations

final_recs.show(5)


# COMMAND ----------
```

```
# seeing the movies rated by userId 60

movies_ratings_data.filter(col("userId") == 60).show()


# COMMAND ----------


# seeing the recommednations for userId 60

clean_recs.filter(col("userId") == 60).show()


# COMMAND ----------
```

 # ANALYSIS

```
# COMMAND ----------


# IMPACT OF SCALE ON PERFORMANCE (TIME)


# COMMAND ----------


# building a dataframe of time on local machine

time_local = [197.27, 209.52, 226.90, 237.77]

size = ['25', '50', '75', '100']
```

```
df_time_local = pd.DataFrame(list(zip(size, time_local_1)))

df_time_local.columns = ['size', 'time']

df_time_local = df_time_local.set_index('size')

df_time_local


# COMMAND ----------


# building a dataframe of time on aws

time_aws = [11.92, 13.47, 19.18, 22.38]

size = ['25', '50', '75', '100']

df_time_aws = pd.DataFrame(list(zip(size, time_aws)))

df_time_aws.columns = ['size', 'time']

df_time_aws = df_time_aws.set_index('size')

df_time_aws


# COMMAND ----------


# plotting the graph to measure the impact of scale on performance (time)

plt.figure(figsize=(7,5), dpi= 120);

plt.plot(df_time_local.index, df_time_local['time'], color='darkorange', label = 'Local', lw=2);

plt.plot(df_time_aws.index, df_time_aws['time'], color='darkblue', label = 'Cloud', lw=2);

plt.legend(loc="upper left");
```

```
plt.scatter(df_time_local.index,df_time_local['time'], color='darkorange', lw=2);

plt.scatter(df_time_aws.index,df_time_aws['time'], color='darkblue', lw=2);

plt.xlabel('SCALE',fontsize=12);

plt.ylabel('RUNTIME (SECONDS)',fontsize=12);

plt.title('IMPACT OF SCALE ON PERFORMANCE (TIME)', fontdict={'size':14});

plt.grid(alpha=0.5)

labels = ['25%','50%', '75%', '100%']

plt.show();
```

```
# COMMAND ----------
```

```
# IMPACT OF SCALE ON QUALITY (RMSE)
```

```
# COMMAND ----------
```

```
# building a dataframe for RMSE genearted on local

size = [25, 50, 75, 100]

rmse_local = [1.05, 0.96, 0.91, 0.87]

df_rmse_local = pd.DataFrame(list(zip(size, rmse_local)))

df_rmse_local.columns = ['Size', 'RMSE']
```

```
df_rmse_local = df_rmse_local.set_index('Size')

df_rmse_local


# COMMAND ----------


# building a dataframe for RMSE genearted on aws

size1 = [25, 50, 75, 100]

rmse_aws = [1.07, 0.94, 0.91, 0.88]

df_rmse_aws = pd.DataFrame(list(zip(size1, rmse_aws)))

df_rmse_aws.columns = ['Size', 'RMSE']

df_rmse_aws = df_rmse_aws.set_index('Size')

df_rmse_aws


# COMMAND ----------


# plotting the graph to measure the impact of scale on quality (rmse)

plt.figure(figsize=(7,5), dpi= 120);

plt.plot(df_rmse_local.index, df_rmse_local['RMSE'], color='darkorange', label = 'Local', lw=2);

plt.plot(df_rmse_aws.index, df_rmse_aws['RMSE'], color='darkblue', label = 'Cloud', lw=2);

plt.legend(loc="upper right");

plt.scatter(df_rmse_local.index,df_rmse_local['RMSE'], color='darkorange', lw=2);

plt.scatter(df_rmse_aws.index,df_rmse_aws['RMSE'], color='darkblue', lw=2);
```

```
plt.xlabel('SCALE',fontsize=12);

plt.ylabel('RMSE',fontsize=12);

plt.title('IMPACT OF SCALE ON QUALITY', fontdict={'size':14});

plt.grid(alpha=1)

labels = ['25%','50%', '75%', '100%']

plt.show();


# COMMAND ----------


# IMPACT OF PARALLEL COMPUTATION AS SCALE INCREASES COMPARED TO NON-PARALLEL EXECUTION


# COMMAND ----------


# building a dataframe with respect to time for 100% data on local

number = [1, 2, 3, 4]

size1 = [100, 100, 100, 100']

time2 = [125.85, 109.13, 102.65, 99.86]

df_time_local2 = pd.DataFrame(list(zip(size1, number, time2)))

df_time_local2.columns = ['Size of the Dataset', 'Number', 'Time']

df_time_local2 = df_time_local2.set_index('Size of the Dataset')

df_time_local2
```

```
# COMMAND ----------


# building a dataframe with respect to time for 100% data on aws

number = [1, 2, 3, 4]

time3 = [33.39, 22.75, 16.10, 10.96]

size3 = [100, 100, 100, 100]

df_time_aws3 = pd.DataFrame(list(zip(size3, number, time3)))

df_time_aws3.columns = ['Size of the Dataset', 'Number', 'Time']

df_time_aws3 = df_time_aws3.set_index('Size of the Dataset')

df_time_aws3


# COMMAND ----------


# plotting a graph to study the impact of parallel and non-parallel computation as scale increases

plt.figure(figsize=(7,5), dpi= 120);

plt.plot(df_time_local2['Number'], df_time_local2['Time'], color='darkorange', label = 'Local', lw=2);

plt.plot(df_time_aws3['Number'], df_time_aws3['Time'], color='darkblue', label = 'Cloud', lw=2);

plt.legend(loc="upper right");

plt.scatter(df_time_local2['Number'],df_time_local2['Time'], color='darkorange', lw=2);

plt.scatter(df_time_aws3['Number'],df_time_aws3['Time'], color='darkblue', lw=2);

plt.xlabel('NUMBER',fontsize=12);

plt.ylabel('RUNTIME (SECONDS)',fontsize=12);
```

```
plt.title('IMPACT OF PARALLEL COMPUTATION ON PERFORMANCE (TIME)', fontdict={'size':14});

plt.grid(alpha=0.5)

labels = ['1', '2', '3', '4']

plt.show();


# COMMAND ----------


# building a dataframe with respect to time for 25% data on local

number = [1, 2, 3, 4]

size = [25, 25, 25, 25]

time1_local = [26.95, 30.69, 36.65, 41.68]

df_time_local2 = pd.DataFrame(list(zip(size, number, time1_local)))

df_time_local2.columns = ['Size of the Dataset', 'Number', 'Time']

df_time_local2 = df_time_local2.set_index('Size of the Dataset')

df_time_local2


# COMMAND ----------


# building a dataframe with respect to time for 25% data on aws

number = [1, 2, 3, 4]

size = [25, 25, 25, 25]

time_aws2 = [26.09, 18.01, 16.35, 10.73]
```

```
df_time_aws2 = pd.DataFrame(list(zip(size, number, time_aws2)))

df_time_aws2.columns = ['Size of the Dataset', 'Number', 'Time']

df_time_aws2 = df_time_aws2.set_index('Size of the Dataset')

df_time_aws2


# COMMAND ----------


# plotting a graph to study the impact of parallel and non-parallel computation as scale increases

plt.figure(figsize=(7,5), dpi= 120);

plt.plot(df_time_local2['Number'], df_time_local2['Time'], color='darkorange', label = 'Local', lw=2);

plt.plot(df_time_aws2['Number'], df_time_aws2['Time'], color='darkblue', label = 'Cloud', lw=2);

plt.legend(loc="upper left");

plt.scatter(df_time_local2['Number'],df_time_local2['Time'], color='darkorange', lw=2);

plt.scatter(df_time_aws2['Number'],df_time_aws2['Time'], color='darkblue', lw=2);

plt.xlabel('NUMBER',fontsize=12);

plt.ylabel('RUNTIME (SECONDS)',fontsize=12);

plt.title('IMPACT OF PARALLEL COMPUTATION ON PERFORMANCE (TIME)', fontdict={'size':14});

plt.grid(alpha=0.5)

labels = ['1', '2', '3', '4']

plt.show();
```