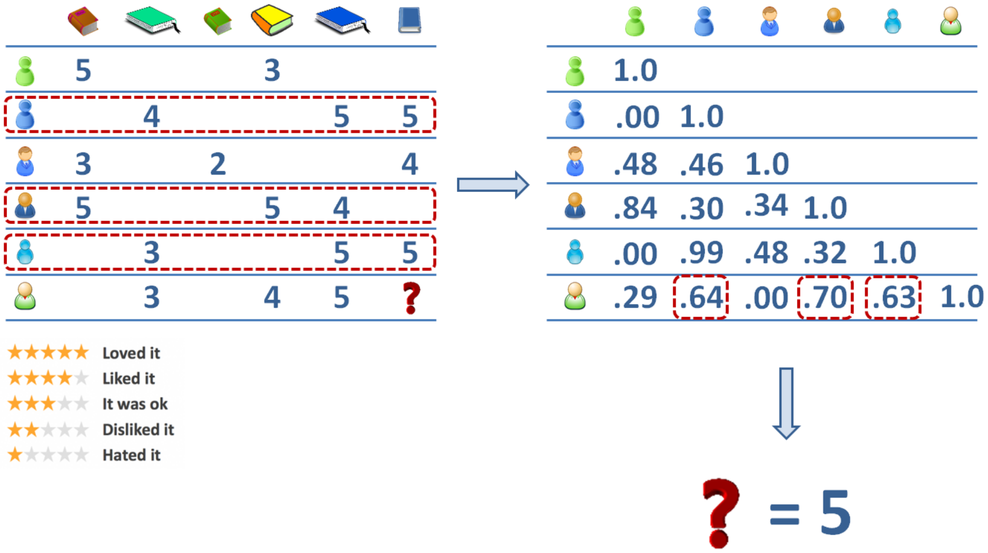
Collaborative filtering using optimized data structures on Spark

CMPE 297 Section-02

Collaborative Filtering



Shailja Kartik

December, 2022

Collaborative Filtering

Collaborative methods for recommender systems are methods that are based solely on the past interactions recorded between users and items to produce new recommendations.
A few characteristics of Collaborative Filtering are:

- In collaborative filtering, we usually use data of what was in any way linked together by an outside sorting entity (e.g. bought together by an online shopper)and show them in an ordered list.
- A collaborative recommendation engine emphasizes user preference.
- In collaborative filtering, a recommendation engine requires the user profiles to suggest relevant content.
- A collaborative recommendation engine doesn't always ensure precise recommendations
- because users with similar tastes may not like the same products.
- Example of Collaborative filtering: we analyze what books have been read by people that have read book A, and the ones with the highest count come at the top of the list.

DATA

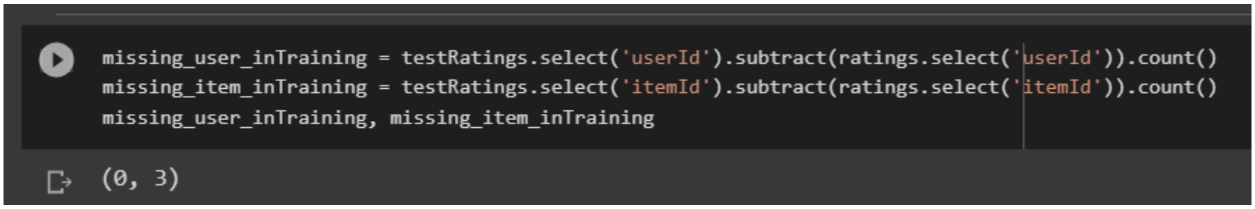
The data we are working for Collaborative Filtering:

Data	Features	Type
Training Set	userId, itemId, rating, timestamp	Integer
Test Set	userId, itemId	Integer

The output generated by each model will be in a series of ratings corresponding to Test Set

DATA PRE-PROCESSING

To work with Matrix Factorization, each UserId and ItemId inside the Testing Set needed to be present in the matrix we'll create using the Training Set.
To check this condition, both datasets were compared as below:



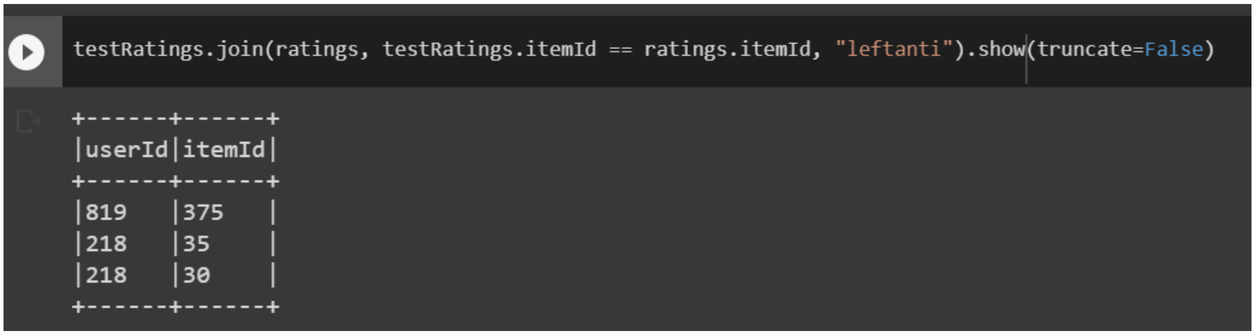
```
missing_user_inTraining = testRatings.select('userId').subtract(ratings.select('userId')).count()
missing_item_inTraining = testRatings.select('itemId').subtract(ratings.select('itemId')).count()
missing_user_inTraining, missing_item_inTraining
```

```
(0, 3)
```

The above query showed that 3 ItemIds are in the Testing set but missing from the Training Set.

To resolve this problem, the below steps were used:

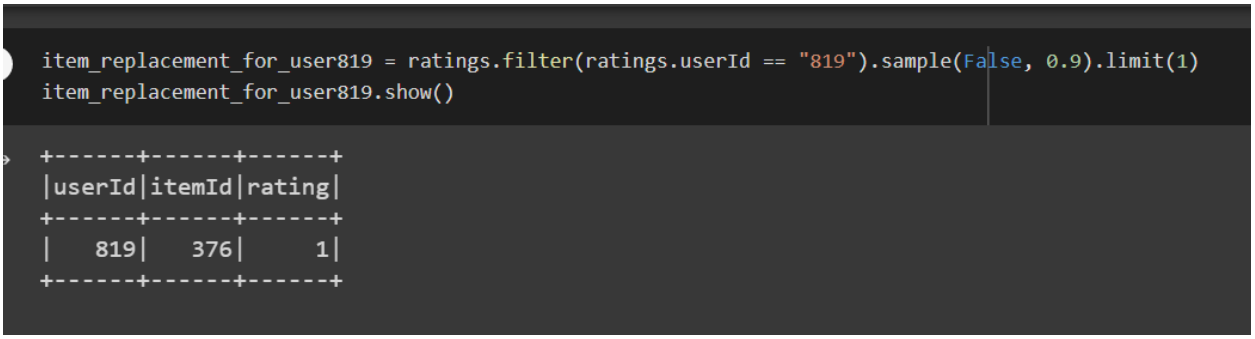
1. Select which users have this unmatched ItemId in the Testing set



```
testRatings.join(ratings, testRatings.itemId == ratings.itemId, "leftanti").show(truncate=False)
```

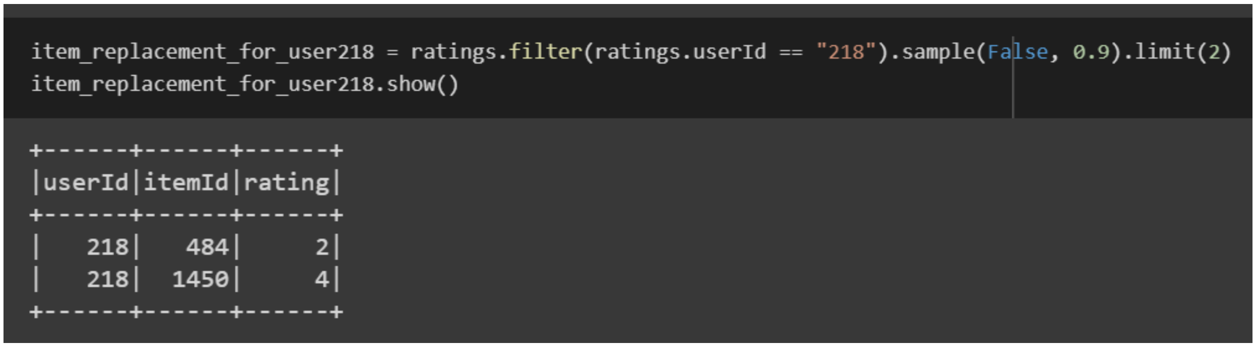
```
+------+------+
|userId|itemId|
+------+------+
|819   |375   |
|218   |35    |
|218   |30    |
+------+------+
```

2. Randomly sample these two UserIds - 819 and 218 in the Training set.
   Because the UserId 819 has one mismatch, only one random sample is generated, and the UserId 218 has two mismatches, and two random samples are generated.



```
item_replacement_for_user819 = ratings.filter(ratings.userId == "819").sample(False, 0.9).limit(1)
item_replacement_for_user819.show()
```

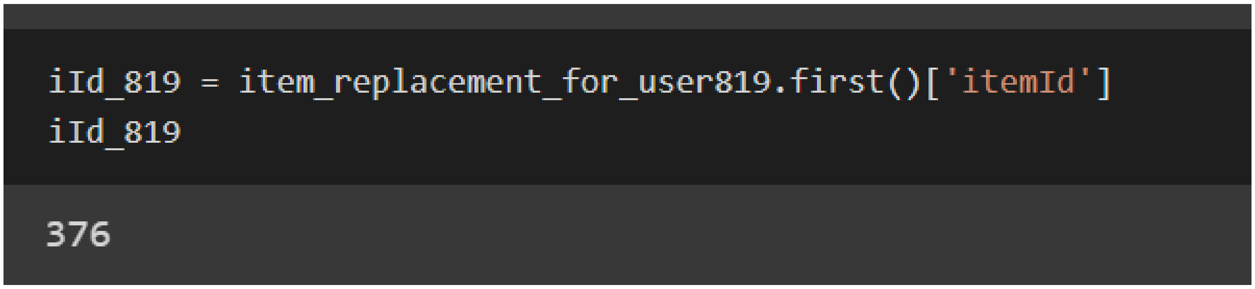
```
+------+------+------+
|userId|itemId|rating|
+------+------+------+
|   819|   376|     1|
+------+------+------+
```



```
item_replacement_for_user218 = ratings.filter(ratings.userId == "218").sample(False, 0.9).limit(2)
item_replacement_for_user218.show()
```

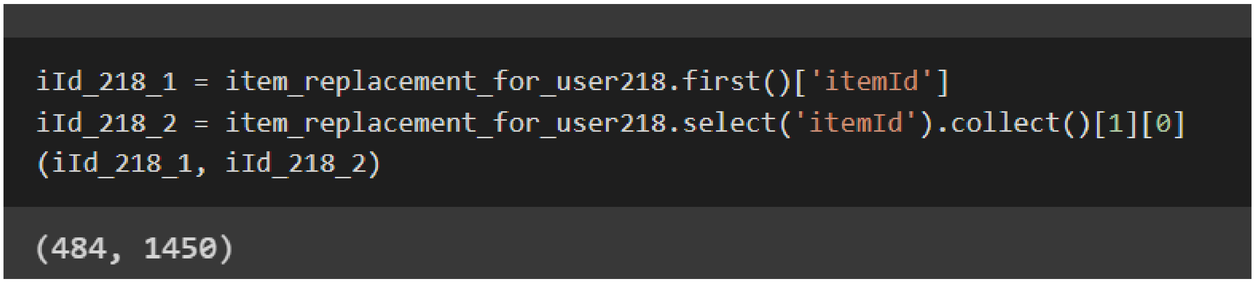
```
+------+------+------+
|userId|itemId|rating|
+------+------+------+
|   218|   484|     2|
|   218|  1450|     4|
+------+------+------+
```

3. Now, we take this rating of these random samples into a variable



```
iId_819 = item_replacement_for_user819.first()['itemId']
iId_819
```

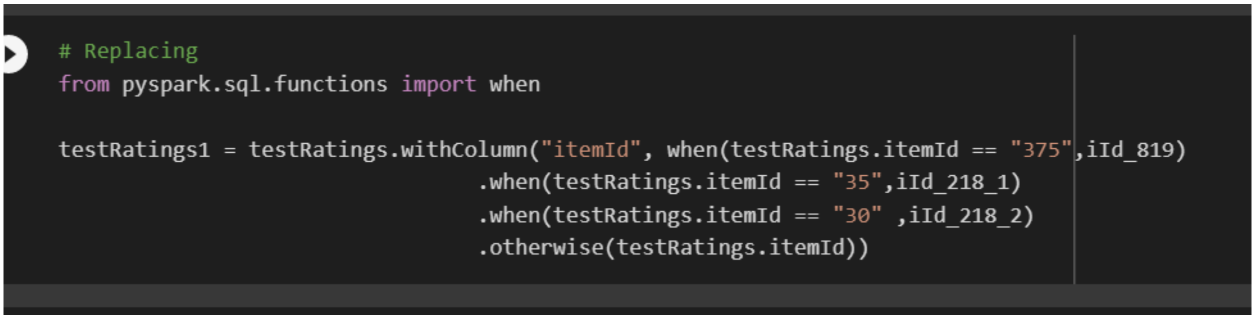
```
376
```



```
iId_218_1 = item_replacement_for_user218.first()['itemId']
iId_218_2 = item_replacement_for_user218.select('itemId').collect()[1][0]
(iId_218_1, iId_218_2)
```

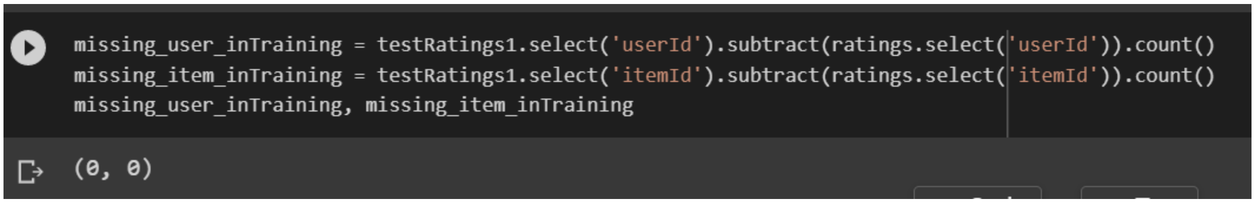
```
(484, 1450)
```

4. And then replace the unwanted ratings in the Testing set with these data



```
# Replacing
from pyspark.sql.functions import when

testRatings1 = testRatings.withColumn("itemId", when(testRatings.itemId == "375",iId_819)
                                    .when(testRatings.itemId == "35",iId_218_1)
                                    .when(testRatings.itemId == "30" ,iId_218_2)
                                    .otherwise(testRatings.itemId))
```

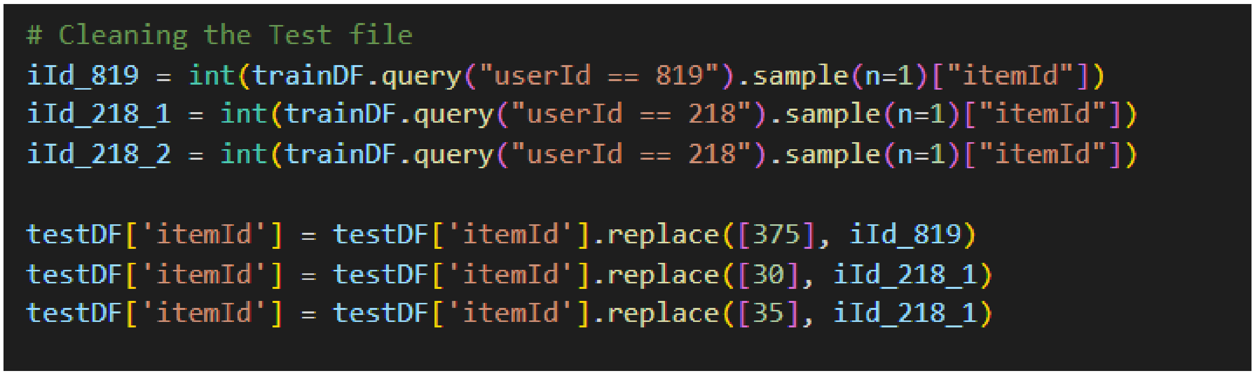
5. This way, a new sample is generated from the same User domain every time.
6. Then, we compare the datasets again.



```
missing_user_inTraining = testRatings1.select('userId').subtract(ratings.select('userId')).count()
missing_item_inTraining = testRatings1.select('itemId').subtract(ratings.select('itemId')).count()
missing_user_inTraining, missing_item_inTraining
```

```
(0, 0)
```

7. Equivalent Python Code for the above steps:



```
# Cleaning the Test file
iId_819 = int(trainDF.query("userId == 819").sample(n=1)["itemId"])
iId_218_1 = int(trainDF.query("userId == 218").sample(n=1)["itemId"])
iId_218_2 = int(trainDF.query("userId == 218").sample(n=1)["itemId"])

testDF['itemId'] = testDF['itemId'].replace([375], iId_819)
testDF['itemId'] = testDF['itemId'].replace([30], iId_218_1)
testDF['itemId'] = testDF['itemId'].replace([35], iId_218_1)
```

We have cleaned our Testing set.

1.LSH to predict rating

Library used:

- ## BucketedRandomProjectionLSH

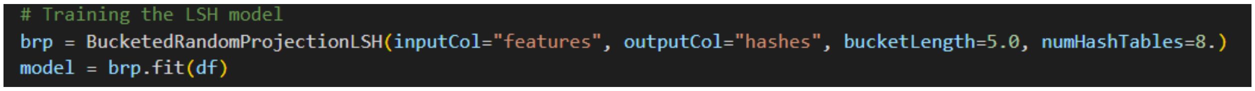
  The general idea of LSH is to use a family of functions ("LSH families") to hash data points into buckets so that the data points which are close to each other are in the same buckets with high probability. In contrast, data points far away from each other are very likely in different buckets.

  - Here, the data points are the relationship between userId and itemId

  We created these points using the **Vector Assembler** library of Pyspark

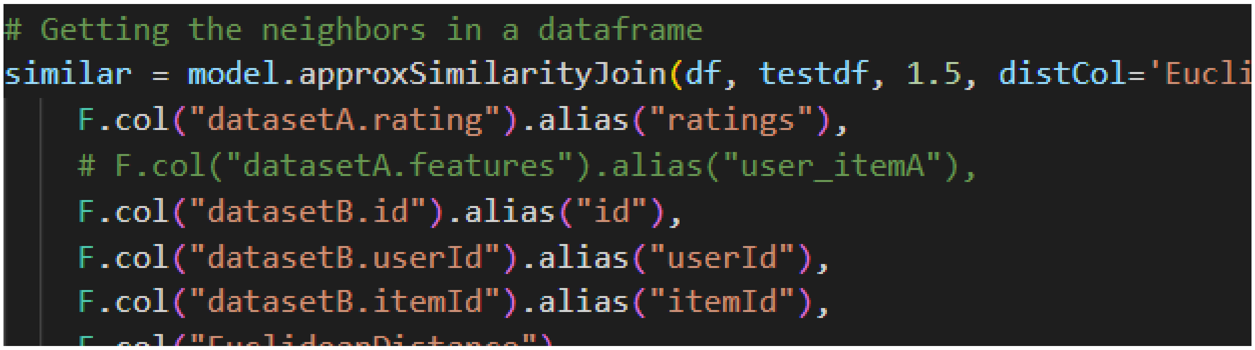
```
# Preparing the Data
# Vector Assembler
def vector_assembler(dataframe, indep_cols):
    assembler = VectorAssembler(inputCols = indep_cols,
                                outputCol = 'features')
    output = assembler.transform(dataframe)
    return output

df = vector_assembler(ratings, indep_cols = ratings.drop('rating',
'id').columns)
testdf = vector_assembler(testRatings1, indep_cols =
testRatings1.drop('id').columns)
```

- The BucketedRandomProjectionLSH takes the input columns in vector format, converts it into hashes.



```
# Training the LSH model
brp = BucketedRandomProjectionLSH(inputCol="features", outputCol="hashes", bucketLength=5.0, numHashTables=8.)
model = brp.fit(df)
```

- In addition to input features, we also pass bucket length and number of Hash tables we want the library to create.
  Here bucket length used is 5, and the number of hash tables we want is 8
- After training the model, we use **Approximate Similarity Join,** another library of pyspark to find the neighbors as below:



```
# Getting the neighbors in a dataframe
similar = model.approxSimilarityJoin(df, testdf, 1.5, distCol='Eucli
    F.col("datasetA.rating").alias("ratings"),
    # F.col("datasetA.features").alias("user_itemA"),
    F.col("datasetB.id").alias("id"),
    F.col("datasetB.userId").alias("userId"),
    F.col("datasetB.itemId").alias("itemId"),
    F.col("EuclideanDistance")
```

Approximate similarity join takes two datasets and approximately returns pairs of rows in the datasets whose distance is smaller than a user-defined threshold.

- The output we have is a pyspark dataframe which consists of neighbors having the same or closer hash-values between the training's userId-itemId set and the testing userId-itemId set.
- We then used the Python function to extract the ratings of the testing set.
- The final implementation of this track gave an **RMSE score of 1.5804 on the 50% of the Test set**
- **Entire Source Codes for LSH implementation:**

1. **LSH_ratings.py**

```
''' Rating Prediction using LSH '''
# Importing libraries
import time
from pyspark.sql import Row
from pyspark.sql.functions import monotonically_increasing_id
from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler, BucketedRandomProjectionLSH
from pyspark.sql.functions import when
import pyspark.sql.functions as F


spark = SparkSession.builder.appName('LSH').getOrCreate()

# Loading the training dataset
lines = spark.read.text('train.dat').rdd
parts = lines.map(lambda row: row.value.split("\t"))
ratingsRDD = parts.map(lambda p: Row(userId=int(p[0]), itemId=int(p[1]), rating=int(p[2])))

```

```python
ratings = spark.createDataFrame(ratingsRDD)
ratings = ratings.select("*").withColumn("id", monotonically_increasing_id())
(training, test) = ratings.randomSplit([0.8, 0.2])
# ratings.count() # 85724

# Loading the Test dataset
lines = spark.read.text('test.dat').rdd
parts = lines.map(lambda row: row.value.split("\t"))
testRatingsRDD = parts.map(lambda p: Row(userId=int(p[0]), itemId=int(p[1])))

testRatings = spark.createDataFrame(testRatingsRDD)
testRatings = testRatings.select("*").withColumn("id", monotonically_increasing_id())
# testRatings.count() # 2154

# Data Cleaning
item_replacement_for_user819 = ratings.filter(ratings.userId == "819").sample(False, 0.9).limit(1)
iId_819 = item_replacement_for_user819.first()['itemId']
item_replacement_for_user218 = ratings.filter(ratings.userId == "218").sample(False, 0.9).limit(2)
iId_218_1 = item_replacement_for_user218.first()['itemId']
iId_218_2 = item_replacement_for_user218.select('itemId').collect()[1][0]

# Replacing
testRatings1 = testRatings.withColumn("itemId", when(testRatings.itemId == "375",iId_819)
                                  .when(testRatings.itemId == "35",iId_218_1)
                                  .when(testRatings.itemId == "30" ,iId_218_2)
                                  .otherwise(testRatings.itemId))

```

```
missing_user_inTraining = testRatings1.select('userId').subtract(ratings.select('userId')).count()
missing_item_inTraining = testRatings1.select('itemId').subtract(ratings.select('itemId')).count()
missing_user_inTraining, missing_item_inTraining

# Preparing the Data
# Vector Assembler
def vector_assembler(dataframe, indep_cols):
    assembler = VectorAssembler(inputCols = indep_cols,
                                outputCol = 'features')
    output = assembler.transform(dataframe)
    return output

df = vector_assembler(ratings, indep_cols = ratings.drop('rating', 'id').columns)
testdf = vector_assembler(testRatings1, indep_cols = testRatings1.drop('id').columns)

start_train_time = time.time()
# Training the LSH model
brp = BucketedRandomProjectionLSH(inputCol="features", outputCol="hashes", bucketLength=5.0, numHashTables=8.)
model = brp.fit(df)
end_train_time = time.time()

start_neighbor_time = time.time()
# Getting the neighbors in a dataframe
similar = model.approxSimilarityJoin(df, testdf, 1.5, distCol='EuclideanDistance').select(
    F.col("datasetA.rating").alias("ratings"),
    # F.col("datasetA.features").alias("user_itemA"),
    F.col("datasetB.id").alias("id"),
    F.col("datasetB.userId").alias("userId"),
    F.col("datasetB.itemId").alias("itemId"),
    F.col("EuclideanDistance")
).orderBy(F.col('EuclideanDistance'))
end_neighbor_time = time.time()

# Converting the neighbor dataframe and test dataframe to Pandas
test_set_df = testRatings1.toPandas()
similar_set_df = similar.toPandas()

# Function to predict rating
def predict_ratings(df):
  rating_file=open('lsh_ratings.dat', 'w')

  def get_rating(u, i):
    k = similar_set_df.loc[(similar_set_df['userId'] == u) & (similar_set_df['itemId'] == i)]["ratings"]
    k = k.tolist()[0] if bool(k.tolist()) else 0
    if k == 0:
      k = similar_set_df.loc[(similar_set_df['userId'] == u)]["ratings"]
      k = k.tolist()[0] if bool(k.tolist()) else 0
    return k if k else 1

  for ind in df.index:
      userId = df['userId'][ind]
      itemId = df['itemId'][ind]
      rating = get_rating(userId, itemId)
      rating_file.write(str(rating))
        if ind < len(df.index) - 1:
          rating_file.write('\n')

    rating_file.close()

  # Predicting the ratings
  start_pred_time = time.time()
```

```
predict_ratings(test_set_df)
end_pred_time = time.time()

evaluation_file=open('lsh_rating_evaluation.txt', 'w')
evaluation_file.write(f'Time taken for training the model : {end_train_time - start_train_time} seconds\n')
evaluation_file.write(f'Time taken to find the closest neighbors : {end_neighbor_time - start_neighbor_time} seconds\n')
evaluation_file.write(f'Time taken for making predictions : {end_pred_time - start_pred_time} seconds\n')

spark.stop()
```

2. Latent Factors to predict rating

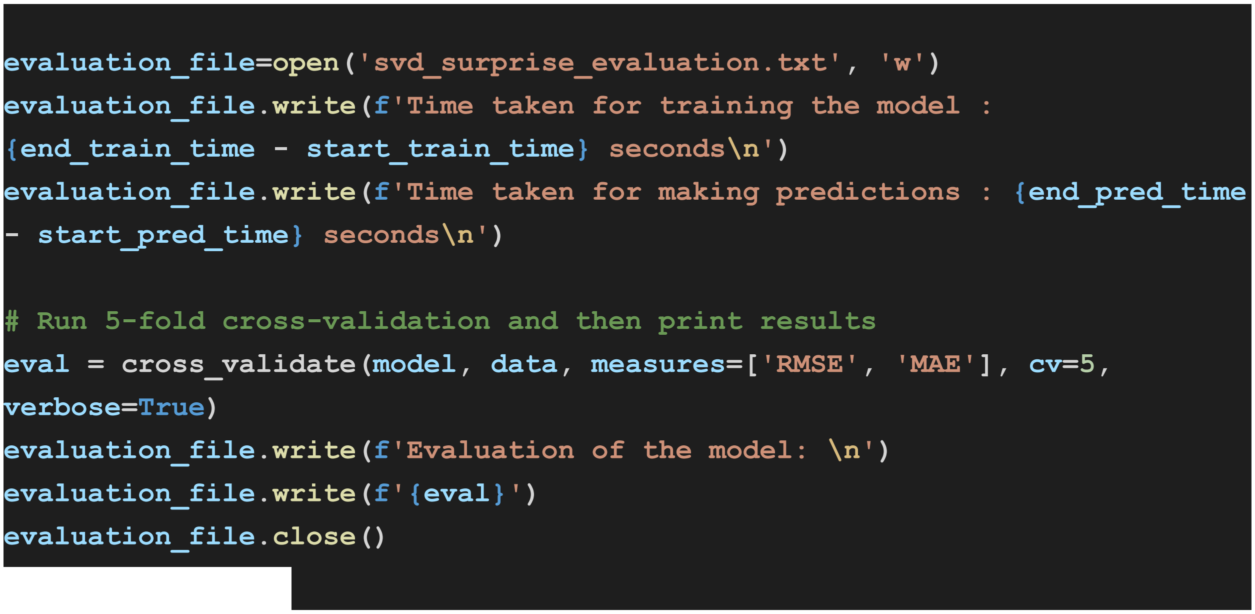
Algorithms used were:

- SVD

  - In linear algebra, the Singular Value Decomposition (SVD) of a matrix is a factorization of that matrix into three matrices.
  - A very helpful library that implemented SVD is **surprise library from scikit.**
  - The final implementation of SVD gave an **RMSE score of 1.0009 on the 50% of the Test set when the number of factors used in decomposition was 50**
  - **Code Implementation for SVD**

Svd_surprise.py

```
# Import libraries
import pandas as pd
import time
from surprise import Reader, Dataset, SVD
from surprise.model_selection import cross_validate, train_test_split

# Predict function
def predict_ratings(df):
  rating_file=open('svd_surprise_prediction.dat', 'w')

  for ind in df.index:
    userId = df['userId'][ind]
    itemId = df['itemId'][ind]
    pred = round(model.predict(userId, itemId).est)
    rating_file.write(str(pred))
    if ind < len(df.index) - 1:
      rating_file.write('\n')

  rating_file.close()

# Reading training file
trainDF = pd.read_csv('train.dat', sep='\t', names=["userId", "itemId",
"rating", "timestamp"])
trainDF.drop('timestamp', axis=1, inplace=True)
# Reading test file
testDF = pd.read_csv('test.dat', sep='\t', names=["userId", "itemId"])

# Cleaning the Test file
iId_819 = int(trainDF.query("userId == 819").sample(n=1)["itemId"])
iId_218_1 = int(trainDF.query("userId == 218").sample(n=1)["itemId"])
iId_218_2 = int(trainDF.query("userId == 218").sample(n=1)["itemId"])

testDF['itemId'] = testDF['itemId'].replace([375], iId_819)
testDF['itemId'] = testDF['itemId'].replace([30], iId_218_1)
testDF['itemId'] = testDF['itemId'].replace([35], iId_218_1)

# Using surprise for SVD
# Data Preparation
reader = Reader(rating_scale=(1,5))
data = Dataset.load_from_df(trainDF, reader)
trainset = data.build_full_trainset()

# Training the model
start_train_time = time.time()
model = SVD(n_factors=50)
model.fit(trainset)
end_train_time = time.time()

start_pred_time = time.time()
predict_ratings(testDF)
end_pred_time = time.time()
```



```
evaluation_file=open('svd_surprise_evaluation.txt', 'w')
evaluation_file.write(f'Time taken for training the model : {end_train_time - start_train_time} seconds\n')
evaluation_file.write(f'Time taken for making predictions : {end_pred_time - start_pred_time} seconds\n')

# Run 5-fold cross-validation and then print results
eval = cross_validate(model, data, measures=['RMSE', 'MAE'], cv=5, verbose=True)
evaluation_file.write(f'Evaluation of the model: \n')
evaluation_file.write(f'{eval}')
evaluation_file.close()
```

- Linear Regression
  - Linear regression analysis is used to predict the value of a variable based on the value of another variable. The variable we want to predict is called the dependent variable. The variable we are using to predict the other variable's value is called the independent variable.
  - Here, for this implementation, I created the Vector Assembler to see how the linear regression works in Pyspark
  - Although it is not the matrix decomposition, the result in predicting the ratings was good on the Test Case.
  - **Linear Regression gave the best RMSE result of 1.1821 on the 50% of the Test case, and it was one of the fastest to predict the ratings**
- Random Forest
  - Another library I utilized which was not matrix decomposition was the Random Forest Regressor of Pyspark.
  - The random forest is a classification algorithm consisting of many decision trees. Because the ratings were categorized from 1 to 5, Random Forest could have been used for multi-class classification.
  - Though Random Forest was the slowest, it gave good results when submitting the test case predictions.

Evaluation of the models on different Numbers of Nodes, CPUs per task, and Tasks per node

Algo	Num of Nodes	CPUs per task	Tasks per node	RMSE one 50% of Test Set	Time to train	Time to predict	Time to find the closest neighbor
LSH	1	2	2	1.5819	0.720	2.7539	0.09870
LF SVD	1	2	2	0.9991	5.29339	0.24752	
LF LR	1	2	2	1.1817	7.3828	0.0627	
LF RF	1	2	2	1.1659	13.8485	0.1282	
LSH	3	3	2	1.5804	0.82884	2.61340	0.07966
LF SVD	3	3	2	1.0009	2.9172	0.08403	
LF LR	3	3	2	1.1821	7.229	0.03140	
LF RF	3	3	2	1.1659	10.5240	0.08602	

Closing

- Increasing the Number of Nodes, and CPUs per task reduced the training time of the various models.
- It also helped in faster prediction.
- However, because multiple nodes were allocated to one task, the slurm-spark job took a lot of time to start.

References

- https://spark.apache.org/docs/3.1.2/api/python/reference/api/pyspark.ml.feature.BucketedRandomProjectionLSH.html
- https://surprise.readthedocs.io/en/stable/matrix_factorization.html