Final_code

November 6, 2019

1 Project 1 - Recommender system for movies

1.1 Data and methods

Let us start by reading the data:

[1]:
```
from pyspark.sql.session import SparkSession
```

[2]:
```
spark = SparkSession.builder.getOrCreate()
```

[3]:
```
data = spark.read.csv("ratings.csv", header='true').drop('timestamp')
data.show()
```

```
+------+-------+------+
|userId|movieId|rating|
+------+-------+------+
|     1|      2|   3.5|
|     1|     29|   3.5|
|     1|     32|   3.5|
|     1|     47|   3.5|
|     1|     50|   3.5|
|     1|    112|   3.5|
|     1|    151|   4.0|
|     1|    223|   4.0|
|     1|    253|   4.0|
|     1|    260|   4.0|
|     1|    293|   4.0|
|     1|    296|   4.0|
|     1|    318|   4.0|
|     1|    337|   3.5|
|     1|    367|   3.5|
|     1|    541|   4.0|
|     1|    589|   3.5|
|     1|    593|   3.5|
|     1|    653|   3.0|
|     1|    919|   3.5|
+------+-------+------+
only showing top 20 rows
```

How many entries are there in the data?

[4]:
```
data.count()
```

[4]: 20000263

[5]:
```
data.describe("rating").show()
```

```
+-------+------------------+
|summary|            rating|
+-------+------------------+
|  count|          20000263|
|   mean|3.5255285642993797|
| stddev| 1.051988919294227|
|    min|               0.5|
|    max|               5.0|
+-------+------------------+
```

Converting the data into numeric values:

[6]:
```
from pyspark.sql.types import DoubleType, IntegerType

data = data.withColumn("movieId", data["movieId"].cast(IntegerType()))
data = data.withColumn("userId", data["userId"].cast(IntegerType()))
data = data.withColumn("rating", data["rating"].cast(DoubleType()))
data.printSchema()
```

```
root
 |-- userId: integer (nullable = true)
 |-- movieId: integer (nullable = true)
 |-- rating: double (nullable = true)
```

1.2 Sampling the data

[7]:
```
import pandas as pd
data = pd.read_csv('ratings.csv').drop('timestamp', axis=1)
```

[8]:
```
data.head()
```

[8]:

	userId	movieId	rating
0	1	2	3.5
1	1	29	3.5
2	1	32	3.5
3	1	47	3.5
4	1	50	3.5

In order to train our models, we need to sample our data into a smaller dataset. We plan to create a user-based collaborative filtering, so we will need to compute the similarities between the users. We believe it is appropriate to select the most popular movies as it would be easier to evaluate the similarities in behavior between users. Let us start by selecting the movies that have more than 5,000 ratings:

[9]:
```
counts = data['movieId'].value_counts()
data = data[data['movieId'].isin(counts.index[counts > 5000])]
data.groupby('movieId').nunique().count()
```

[9]:
```
userId     1005
movieId    1005
rating     1005
dtype: int64
```

[10]:
```
data.groupby('userId').nunique().count()
```

[10]:
```
userId     138476
movieId    138476
rating     138476
dtype: int64
```

We end up with 1,005 distinct movies and 138,476 distinct users. Let us randomly select 10,000 users to build our dataset.

[11]:
```
import numpy as np
np.random.seed(1)
sample = data['userId'].unique()
sample_users = np.random.choice(sample, 10000)
dataset = data.loc[data['userId'].isin(sample_users)]
dataset
```

[11]:

	userId	movieId	rating
922	10	1	4.0
923	10	11	4.0
924	10	25	4.0
925	10	260	4.0
926	10	356	3.0
…	…	…	…
19999279	138484	733	3.0
19999280	138484	736	4.0
19999281	138484	748	3.0
19999282	138484	780	5.0
19999283	138484	802	5.0

[890576 rows x 3 columns]

[12]:
```
dataset.groupby('movieId').nunique().count()
```

[12]:
```
userId     1005
movieId    1005
rating     1005
dtype: int64
```

[13]:
```
dataset.groupby('userId').nunique().count()
```

[13]:
```
userId     9672
movieId    9672
rating     9672
dtype: int64
```

Interesting, we only have 9,672 users instead of 10,000... But well, it will be enough for our study!

Let us now split our datset in a train set and a test set. We will use both of them for the rest of our study in this homework:

[14]:
```
dataset = spark.createDataFrame(dataset)
(training, test) = dataset.randomSplit([0.8,0.2], seed=0)
print(training.count(), test.count())
```

```
712334 178242
```

1.3 Statistics

We can make some statistics about our new set of movies to better understand it. Here is the number of ratings each movie has:

[15]:
```
stat = dataset.groupBy("movieId").count().sort('count', ascending=False)
stat.describe('count').show()
```

```
+-------+-----------------+
|summary|            count|
+-------+-----------------+
|  count|             1005|
|   mean|886.1452736318408|
| stddev|651.5892025154428|
|    min|              326|
|    max|             4654|
+-------+-----------------+
```

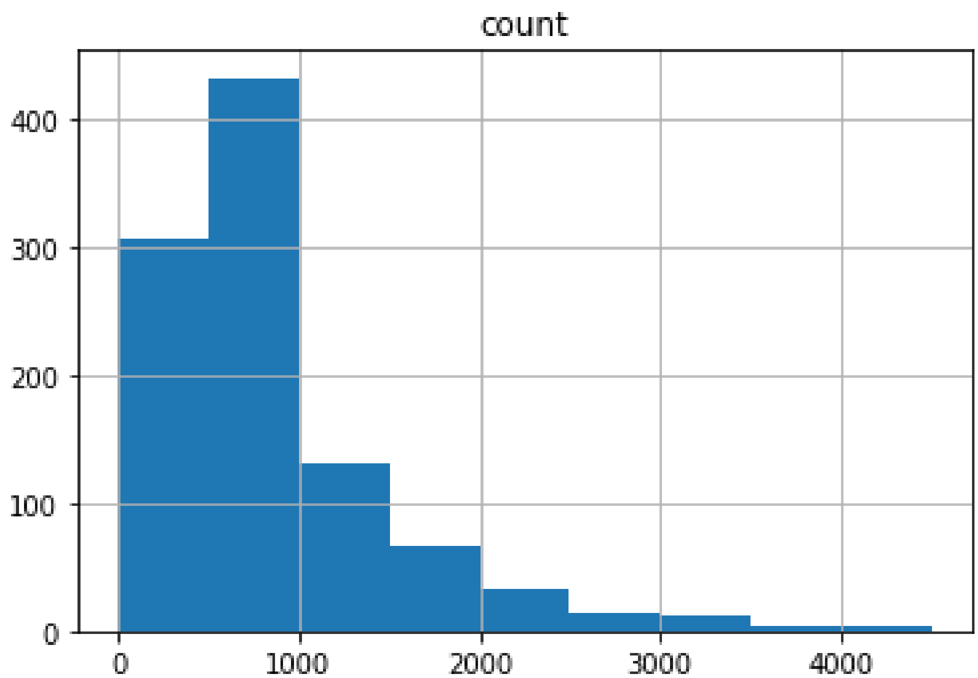
Even though we kept a small part of the users, we still have at least 326 ratings per movie which is enough for our study.

Here we can see the number of ratings with respect to the number of movies:

[16]:
```
%matplotlib inline
import pandas as pd
```

```
panda_stat = stat.toPandas()
bin_ = [i*500 for i in range(10)]
panda_stat.hist(column='count', bins=bin_)
```

[16]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x120465fd0>]],
      dtype=object)



We can observe that the vast majority of the movies have less than 1,000 ratings and just a few of them have more ratings. This is the long tail effect: the popular movies (that represent a small proportion of the overall dataset) are far more rated than the other ones.

2 Metrics

In this section, we will define the metrics functions which we will use later to assess the performances of our models: - MAE - RMSE - precision

MAE and RMSE:

[17]:
```
from sklearn.metrics import mean_absolute_error, mean_squared_error
import numpy as np

def RMSE(y, y_predicted):
    return np.sqrt(mean_squared_error(y, y_predicted))
```

Precision:

[18]:
```
# Let us first create a function to transform our raw predictions into␣
↪appropriate ones.
def round_rating(x):
    dec = x - int(x)
    if (dec >= 0.25) and (dec < 0.75):
        return int(x) + 0.5
    else:
        return int(x) + (dec > 0.5)

# Precision function metric:
def precision(y, y_predicted):
    y_predicted = [round_rating(k) for k in y_predicted]
    TP = 0
    for i in range(len(y)):
        if y[i] == y_predicted[i]:
            TP += 1
    return TP/len(y)
```

2.1 Baseline Model

We first need a baseline model to later compare the performances of our personalized models. We will use this one:

$$r_{ui} = \mu$$

with μ: global average rating

[19]:
```
training.describe("rating").show()
```

```
+-------+------------------+
|summary|            rating|
+-------+------------------+
|  count|            712334|
|   mean| 3.639545494108101|
| stddev|1.0158713154695678|
|    min|               0.5|
|    max|               5.0|
+-------+------------------+
```

[20]:
```
mu = 3.639545494108101
```

[21]:
```
def baseline_rating(userId, movieId):
    return mu
```

[22]:
```
test_baseline = test.toPandas()
```

[23]:
```
import time
```

```
t0 = time.time()
y_predicted = []
for i in range(len(test_baseline.index)):
    movieId = test_baseline.iloc[i, 1]
    userId = test_baseline.iloc[i, 0]
    y_predicted.append(baseline_rating(userId, movieId))
t_baseline = time.time() - t0
print ("Time of the training:", t_baseline)
```

```
Time of the training: 2.4853639602661133
```

[24]:
```
y = test_baseline['rating'].values
```

[25]:
```
test_baseline['baseline_rating'] = y_predicted
test_baseline.head()
```

[25]:

	userId	movieId	rating	baseline_rating
0	10	1	4.0	3.639545
1	10	260	4.0	3.639545
2	10	527	5.0	3.639545
3	10	1250	4.0	3.639545
4	10	1304	3.0	3.639545

[26]:
```
mae_baseline = mean_absolute_error(y, y_predicted)
rmse_baseline = RMSE(y, y_predicted)
precision_baseline = precision(y, y_predicted)

print('MAE: ', mae_baseline, '\nRMSE: ', rmse_baseline, '\nprecision: ',␣
 ↪precision_baseline, '\nRunning time: ', t_baseline)
```

```
MAE:  0.8195743288512253
RMSE:  1.0181109771997578
precision:  0.1013453619236768
Running time:  2.4853639602661133
```

We will compare the performances of our models with these values.

2.2 Memory Based: user-based model

2.2.1 Computing Similarity matrix

Now we will focus on the training set to build an user similarity matrix, using the adjusted cosine similarity.

Let's first adjust each row to remove the user bias. For that, we first compute the average rating per user, and we substract this average to each rating to obtain a non biased rating column:

[27]:
```
Training = training.toPandas()
Testing = test.toPandas()
```

```
training_ratings_mean = Training.groupby('userId', as_index=False).mean().
↪rename(columns = {'rating': 'rating_mean'})[['userId','rating_mean']]
training_ratings_mean.head()
```

[27]:

	userId	rating_mean
0	10	3.931034
1	28	2.818182
2	49	3.820000
3	53	4.000000
4	81	4.800000

[28]:
```
training_adjusted_ratings = pd.merge(Training, training_ratings_mean,␣
↪on='userId', how='left', sort=False)
training_adjusted_ratings['rating_adjusted'] =␣
↪training_adjusted_ratings['rating'] -␣
↪training_adjusted_ratings['rating_mean']
training_adjusted_ratings.head()
```

[28]:

	userId	movieId	rating	rating_mean	rating_adjusted
0	10	11	4.0	3.931034	0.068966
1	10	25	4.0	3.931034	0.068966
2	10	356	3.0	3.931034	-0.931034
3	10	858	5.0	3.931034	1.068966
4	10	912	4.0	3.931034	0.068966

[29]:
```
training_adjusted_ratings = training_adjusted_ratings[['movieId', 'userId',␣
↪'rating_adjusted']]
training_adjusted_ratings.head()
```

[29]:

	movieId	userId	rating_adjusted
0	11	10	0.068966
1	25	10	0.068966
2	356	10	-0.931034
3	858	10	1.068966
4	912	10	0.068966

Here, the last column of the dataframe contains the adjusted rating on each user. We will then use this column to compute the similarity between each users using cosine similarity: - firstly, select two distinct users - secondly, isolate items that have been rated by both users - thirdly, apply similarity computation, which is defined as the following:

$$sim(u,v) = \frac{\sum_{i \in I} R_{u,i} R_{v,i}}{\sqrt{\sum_{i \in I} R_{u,i}^2 R_{v,i}^2}}$$

Where: - $R_{u,i}$ is the mean centered rating of item i given by user U - I is the set of items that are rated by u and v

In order to compute the similarity matrix, we will use the cosine_similarity method from the Sklearn library.

Let's first create our rating matrix. We put the userId in rows because we want to create an **user based** model. We can see that it is a very sparse matrix:

[30]:
```
# First the original ratings matrix, used to compute the predicted ratings
training_rating_matrix = Training.pivot_table(values='rating',
                                              index='userId',
                                              columns='movieId')
training_rating_matrix.head()
```

[30]:
```
movieId  1      2      3      5      6      7      10     11     14     16     \
userId
10         NaN    NaN    NaN    NaN    NaN    NaN    NaN    4.0    NaN    NaN
28         NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
49         NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
53         4.0    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
81         NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN

movieId  …  70286  71535  72998  73017  74458  78499  79132  80463  81591  \
userId   …
10       …    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
28       …    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
49       …    NaN    NaN    NaN    NaN    NaN    NaN    4.5    NaN    NaN
53       …    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
81       …    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN

movieId  81845
userId
10         NaN
28         NaN
49         NaN
53         NaN
81         NaN

[5 rows x 1005 columns]
```

[31]:
```
# Second the adusted rating matrix, used to compute the similarities between␣
 ↪movies
training_adjusted_rating_matrix = training_adjusted_ratings.
 ↪pivot_table(values='rating_adjusted',
                                                                          ␣
 ↪index='userId',
                                                                          ␣
 ↪columns='movieId')
training_adjusted_rating_matrix.head()
```

[31]:
```
movieId  1      2      3      5      6      7      10         11     14     \
userId
10         NaN    NaN    NaN    NaN    NaN    NaN    NaN  0.068966    NaN
28         NaN    NaN    NaN    NaN    NaN    NaN    NaN       NaN    NaN
49         NaN    NaN    NaN    NaN    NaN    NaN    NaN       NaN    NaN
53         0.0    NaN    NaN    NaN    NaN    NaN    NaN       NaN    NaN
81         NaN    NaN    NaN    NaN    NaN    NaN    NaN       NaN    NaN

movieId  16     …  70286  71535  72998  73017  74458  78499  79132  80463  \
userId          …
10         NaN  …    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
28         NaN  …    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
49         NaN  …    NaN    NaN    NaN    NaN    NaN    NaN   0.68    NaN
53         NaN  …    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
81         NaN  …    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN

movieId  81591  81845
userId
10         NaN    NaN
28         NaN    NaN
49         NaN    NaN
53         NaN    NaN
81         NaN    NaN

[5 rows x 1005 columns]
```

[32]:
```
# Creating a dummy matrix with 0 instead of Nan

dummy_training_adjusted_rating_matrix = training_adjusted_rating_matrix.copy().
 ↪fillna(0)
dummy_training_adjusted_rating_matrix.head()
```

[32]:
```
movieId  1      2      3      5      6      7      10         11     14     \
userId
10         0.0    0.0    0.0    0.0    0.0    0.0    0.0  0.068966    0.0
28         0.0    0.0    0.0    0.0    0.0    0.0    0.0  0.000000    0.0
49         0.0    0.0    0.0    0.0    0.0    0.0    0.0  0.000000    0.0
53         0.0    0.0    0.0    0.0    0.0    0.0    0.0  0.000000    0.0
81         0.0    0.0    0.0    0.0    0.0    0.0    0.0  0.000000    0.0

movieId  16     …  70286  71535  72998  73017  74458  78499  79132  80463  \
userId          …
10         0.0  …    0.0    0.0    0.0    0.0    0.0    0.0   0.00    0.0
28         0.0  …    0.0    0.0    0.0    0.0    0.0    0.0   0.00    0.0
49         0.0  …    0.0    0.0    0.0    0.0    0.0    0.0   0.68    0.0
53         0.0  …    0.0    0.0    0.0    0.0    0.0    0.0   0.00    0.0
81         0.0  …    0.0    0.0    0.0    0.0    0.0    0.0   0.00    0.0
```

```
movieId  81591  81845
userId
10         0.0    0.0
28         0.0    0.0
49         0.0    0.0
53         0.0    0.0
81         0.0    0.0

[5 rows x 1005 columns]
```

[33]:
```python
# Importing the cosine similarity method from the Sklearn library
from sklearn.metrics.pairwise import cosine_similarity

# Computing the similarities
cosine_similarity = cosine_similarity(dummy_training_adjusted_rating_matrix,␣
↪dummy_training_adjusted_rating_matrix)
```

[34]:
```python
# converting the cosine similarity into a similarity matrix
cosine_similarity = pd.DataFrame(cosine_similarity,
                                 index=dummy_training_adjusted_rating_matrix.
↪index,
                                 columns=dummy_training_adjusted_rating_matrix.
↪index)
cosine_similarity.head()
```

[34]:
```
userId    10        28        49        53        81        124       127     \
userId
10      1.000000 -0.069696  0.001287  0.035488 -0.054441  0.009338  0.001382
28     -0.069696  1.000000  0.166682 -0.069757  0.022414 -0.001541 -0.026634
49      0.001287  0.166682  1.000000 -0.012216  0.005586  0.076581 -0.046541
53      0.035488 -0.069757 -0.012216  1.000000 -0.009941 -0.047513  0.000486
81     -0.054441  0.022414  0.005586 -0.009941  1.000000  0.049579  0.009777

userId    163       167       180     …    138376    138382    138387  \
userId                                …
10      0.062998  0.000000  0.000000  …  0.015494  0.013785  0.004675
28     -0.024768  0.000000  0.124142  …  0.055146 -0.056902  0.024929
49      0.017672  0.013294  0.104890  …  0.083366  0.001380  0.109074
53     -0.019090 -0.019356 -0.029822  … -0.004245  0.003400  0.032787
81      0.006285  0.000000  0.000000  …  0.009261  0.024872  0.003093

userId    138397    138422    138425    138481    138482    138483    138484
userId
10     -0.045728 -0.015694  0.000000 -0.021299  0.000000  0.001268 -0.052079
28     -0.014355  0.058112  0.000000  0.000000  0.000000  0.170082  0.094211
49      0.074606 -0.001911  0.024065  0.169790  0.055434  0.081052  0.096710
```

```
53     -0.047263  0.008900  0.000000  0.065448 -0.007524 -0.001845 -0.025489
81      0.009733  0.008238  0.013055  0.196376 -0.027524  0.000000  0.034499

[5 rows x 9671 columns]
```

2.2.2 Computing predicted ratings

Now we can complete the rating matrix by computing each missing rating using weighted mean ratings. More precisely, each missing entry is given by :

$$r_{u,i} = \frac{\sum_{i \in N} Similarity_{u,v} * r_{v,i}}{\sum_{j \in N} || \; Similarity_{u,v} \; ||}$$

where N is the set of users that have rated i.

[35]:
```
import heapq
def user_based_collab(userId, movieId, k=10):

    # First check if the user is in the training set. Otherwise we do not have␣
↪computed similarities for her.
    if userId in training_rating_matrix.index:

        # get the similarities between the movie and the other movies
        user_similarities = cosine_similarity[userId]
        #print('User Similarities: ', user_similarities)

        # Get all the other users' ratings for this movie
        other_users_ratings = training_rating_matrix[movieId]
        #print('other users ratings: ', other_users_ratings)

        # Remove the NaN from the users' ratings and from the similarity vector
        nan_index = other_users_ratings[other_users_ratings.isnull()].index
        other_users_ratings = other_users_ratings.dropna()
        user_similarities = user_similarities.drop(nan_index)
        #print('DROPING NAN')
        #print('User Similarities: ', user_similarities)
        #print('other users ratings: ', other_users_ratings)

        # take k nearest neighbors
        k_index = user_similarities[user_similarities.isin(heapq.nlargest(k,␣
↪user_similarities))].index
        other_users_ratings = other_users_ratings[k_index]
        user_similarities = user_similarities[k_index]
        #print('TAKING K NEAREST')
        #print('User Similarities: ', user_similarities)
        #print('other users ratings: ', other_users_ratings)
```

```
        # Compute the predicted rating
        s = 0
        for sim in user_similarities:
            s += abs(sim)
        if s > 0:
            return np.dot(user_similarities, other_users_ratings) / s
        else:
            return mu


    # If the user or the movie were not in the training set, return the average␣
↪rating
    else:
        return mu
```

[ ]:
```
t0 = time.time()
predictions_user_based =[]
for row in Testing.index:
    userId = Testing.loc[row, 'userId']
    movieId = Testing.loc[row, 'movieId']
    rating = user_based_collab(userId, movieId)
    predictions_user_based.append(rating)
t_user_based = time.time() - t0
print ("Time of the training:", t_user_based)
```

[37]:
```
Testing['User_based_predictions'] = predictions_user_based
Testing
```

[37]:

	userId	movieId	rating	User_based_predictions
0	10	1	4.0	4.117869
1	10	260	4.0	4.091967
2	10	527	5.0	4.011499
3	10	1250	4.0	4.002198
4	10	1304	3.0	4.360314
…	…	…	…	…
178237	138484	555	5.0	3.859160
178238	138484	587	4.0	2.977625
178239	138484	589	5.0	3.340871
178240	138484	593	3.0	4.322535
178241	138484	608	5.0	4.515801

[178242 rows x 4 columns]

[59]:
```
y_1 = Testing['rating']
y_predicted_1 = Testing['User_based_predictions']
```

```
mae_user_based = mean_absolute_error(y_1, y_predicted_1)
rmse_user_based = RMSE(y_1, y_predicted_1)
precision_user_based = precision(y_1, y_predicted_1)

print('MAE: ', mae_user_based, '\nRMSE: ', rmse_user_based, '\nprecision: ',␣
↪precision_user_based, '\nRunning time: ', t_user_based)
```

```
MAE:  0.709629915655458
RMSE:  0.9118947788296244
precision:  0.2285488268758205
Running time:  767.0439291000366
```

Fortunately, we can observe an improvement in our metrics by … This model is more effective than the baseline model because it is more personalized. Indeed, it takes into account the users behaviors and their similarities. However, due to an important sparsity, this is not the best that can be done: the performances would be poorer if we had taken less popular movies with fewer ratings in our sample dataset.

2.3 Model Based: Matrix Factorization

2.3.1 1. Simple ALS method

We will first create a simple ALS method to fit the training set, without fitting the hyper parameters. We start with a rank of 20, 20 iterations, $\lambda = 0.01$ and we set the constraint of non-negativity to True.

Let's first import the required packages:

[40]:
```
from pyspark.ml.recommendation import ALS, ALSModel
from pyspark.ml.tuning import TrainValidationSplit, ParamGridBuilder
from pyspark.ml.evaluation import RegressionEvaluator
```

[41]:
```
# Create ALS Model
SimpleAls = ALS(rank=20, maxIter=20, regParam=0.01, userCol="userId",␣
↪itemCol="movieId", ratingCol="rating",
          coldStartStrategy="drop", nonnegative=True)
```

[42]:
```
import time

t0 = time.time()
SimpleModel = SimpleAls.fit(training)
t_simpleALS = time.time() - t0
print ("Time of the training:", t_simpleALS)
```

```
Time of the training: 14.102512836456299
```

We can therefore use the simple model to compute the predictions:

[43]:
```
# Define evaluator as RMSE
```

```
evaluator_rmse = RegressionEvaluator(metricName='rmse', labelCol='rating',␣
↪predictionCol='prediction')
evaluator_mae = RegressionEvaluator(metricName='mae', labelCol='rating',␣
↪predictionCol='prediction')
```

[44]:
```
# Generate predictions and evaluate RMSE
predictions_training_1 = SimpleModel.transform(training)
predictions_test_1 = SimpleModel.transform(test)

rmse_training_1 = evaluator_rmse.evaluate(predictions_training_1)
rmse_test_1 = evaluator_rmse.evaluate(predictions_test_1)

mae_training_1 = evaluator_mae.evaluate(predictions_training_1)
mae_test_1 = evaluator_mae.evaluate(predictions_test_1)
```

[45]:
```
predictions_test_1.show()
```

```
+------+-------+------+----------+
|userId|movieId|rating|prediction|
+------+-------+------+----------+
| 88599|    471|   3.0| 3.1912675|
|133898|    471|   3.0| 4.3473053|
| 92406|    471|   5.0| 4.8634644|
| 94243|    471|   3.0| 1.8905424|
|115718|    471|   5.0| 4.2721243|
| 49769|    471|   3.5| 3.1854076|
| 72096|    471|   4.0| 2.4613588|
|125339|    471|   3.0| 3.7363334|
|113982|    471|   4.0| 3.7423837|
| 24253|    471|   4.0| 3.5454612|
| 41389|    471|   5.0|  4.758245|
|115672|    471|   4.0| 4.1487803|
|130987|    471|   2.5| 3.5283122|
| 48392|    471|   4.0|  3.999699|
| 45750|    471|   3.0| 2.2895908|
| 48542|    471|   5.0|  4.637712|
| 27050|    471|   4.0| 3.3200185|
|  4386|    471|   4.0| 3.0404818|
| 58440|    471|   3.0| 3.8718536|
| 21145|    471|   5.0| 3.1520758|
+------+-------+------+----------+
only showing top 20 rows
```

[46]:
```
df_predictions_training_1 = predictions_training_1.toPandas()
df_predictions_test_1 = predictions_test_1.toPandas()
```

```
precision_simpleALS_training = precision(df_predictions_training_1['rating'],␣
 ↪df_predictions_training_1['prediction'])
precision_simpleALS_test = precision(df_predictions_test_1['rating'],␣
 ↪df_predictions_test_1['prediction'])

print('For the training set:')
print('MAE: ', mae_training_1, '\nRMSE: ', rmse_training_1, '\nprecision: ',␣
 ↪precision_simpleALS_training)
print()
print('For the test set:')
print('MAE: ', mae_test_1, '\nRMSE: ', rmse_test_1, '\nprecision: ',␣
 ↪precision_simpleALS_test)
print()
print('Running time', t_simpleALS)
```

```
For the training set:
MAE:  0.4655872217135966
RMSE:  0.6148227744526578
precision:  0.3659322733436843

For the test set:
MAE:  0.6414211006720851
RMSE:  0.8454902219527436
precision:  0.2686082326737395

Running time 14.102512836456299
```

This model is quite simple, we did not try to fit the hyperparameters, namely the dimension of the latent vectors (rank of the matrix), the number of ALS iterations and the regularization parameter. Despite its simplicity, it is way better than the user-based model. The running is much lower which means that we can afford to often compute new predictions for a larger dataset. We can now generate Top 10 user recommendations:

[47]:
```
userRecs = SimpleModel.recommendForAllUsers(10).toPandas().set_index('userId')
userRecs.head()
```

[47]:
```
                                           recommendations
userId
15790   [(1101, 5.523297309875488), (377, 5.2211279869…
78120   [(34405, 6.2117156982421875), (2005, 6.1617794…
83250   [(924, 5.376289367675781), (60684, 5.315775394…
113000  [(47, 5.905139446258545), (2959, 5.84366178512…
18051   [(3988, 6.995851516723633), (18, 6.91307926177…
```

2.3.2 2. Better ALS method: fitting the hyperparameters

One way to improve our previous model is to fit the hyperparameters to the model. For this, we are using the *pyspark.ml.tuning* library and its ParamGridBuilder function.

Here we are offering differents possibilites for the hyperparameters and the model will try them all to find the best one.

[48]:
```
# Create ALS Model
FitALS = ALS(userCol="userId", itemCol="movieId", ratingCol="rating",
          coldStartStrategy="drop", nonnegative=True)
```

[49]:
```
# Tune model using ParamGridBuilder
param_grid = ParamGridBuilder().addGrid(FitALS.rank, [15,17,20]).addGrid(FitALS.
↪maxIter, [15,20,25]).addGrid(FitALS.regParam, [.01,.05,.1]).build()
```

Let us use a cross-validation set up to train and tune our model. We will split the training set into a train set and a tune set, with ratio 0.8, 0.2 respectively:

[50]:
```
# Build cross validation using TrainValidationSplit
tvs = TrainValidationSplit(estimator=FitALS, estimatorParamMaps=param_grid,␣
↪evaluator=evaluator_rmse, trainRatio=0.8)
```

[51]:
```
# Fit ALS model to train data
import time

t0 = time.time()
FitModel = tvs.fit(training)
t_fitALS = time.time() - t0
print ("Time of the training:", t_fitALS)
```

```
Time of the training: 554.3457682132721
```

Let's save the model:

[52]:
```
FitModel.save("FitModelALS")
```

[53]:
```
# Extract the best model from the tuning of Hyperparameters
best_model = FitModel.bestModel
```

[54]:
```
# Generate predictions and evaluate RMSE
predictions_training_2 = best_model.transform(training)
predictions_test_2 = best_model.transform(test)

rmse_training_2 = evaluator_rmse.evaluate(predictions_training_2)
rmse_test_2 = evaluator_rmse.evaluate(predictions_test_2)

mae_training_2 = evaluator_mae.evaluate(predictions_training_2)
mae_test_2 = evaluator_mae.evaluate(predictions_test_2)
```

We can print the metrics for this new model:

[61]:
```
df_predictions_training_2 = predictions_training_2.toPandas()
df_predictions_test_2 = predictions_test_2.toPandas()
```

```python
precision_simpleALS_training = precision(df_predictions_training_2['rating'],␣
↪df_predictions_training_2['prediction'])
precision_simpleALS_test = precision(df_predictions_test_2['rating'],␣
↪df_predictions_test_2['prediction'])

print('Best hyperparameters found: ')
print('Rank:', best_model.rank)
print('MaxIter:', best_model._java_obj.parent().getMaxIter())
print('RegParam:', best_model._java_obj.parent().getRegParam())
print()
print('Metrics: ')
print('For the training set:')
print('MAE: ', mae_training_2, '\nRMSE: ', rmse_training_2, '\nprecision: ',␣
↪precision_simpleALS_training)
print()
print('For the test set:')
print('MAE: ', mae_test_2, '\nRMSE: ', rmse_test_2, '\nprecision: ',␣
↪precision_simpleALS_test)
print()
print('Running time', t_fitALS)
```

```
Best hyperparameters found:
Rank: 20
MaxIter: 25
RegParam: 0.1

Metrics:
For the training set:
MAE:  0.5717943020148085
RMSE:  0.7328727913464498
precision:  0.28216398487226496

For the test set:
MAE:  0.6223925941501726
RMSE:  0.799148429671974
precision:  0.2613203471704041

Running time 554.3457682132721
```

We see that the **error** is much lower than the previous one. Fitting the hyperparameters is effective.

Let's see how our model predict the actual ratings of the test set:

[56]:
```python
predictions_test_2.show()
```

```
+------+-------+------+----------+
|userId|movieId|rating|prediction|
+------+-------+------+----------+
```

```
| 88599|    471|   3.0|  3.347707|
|133898|    471|   3.0| 4.2234397|
| 92406|    471|   5.0|  4.281761|
| 94243|    471|   3.0|  2.298056|
|115718|    471|   5.0| 4.0113745|
| 49769|    471|   3.5| 3.0017989|
| 72096|    471|   4.0| 3.1719604|
|125339|    471|   3.0| 3.8096318|
|113982|    471|   4.0| 3.1445296|
| 24253|    471|   4.0| 3.6033697|
| 41389|    471|   5.0| 4.7256317|
|115672|    471|   4.0|   4.13637|
|130987|    471|   2.5| 3.3635993|
| 48392|    471|   4.0|  3.668394|
| 45750|    471|   3.0| 2.9343634|
| 48542|    471|   5.0| 3.9762886|
| 27050|    471|   4.0| 3.2472942|
|  4386|    471|   4.0| 3.4065237|
| 58440|    471|   3.0| 3.8583205|
| 21145|    471|   5.0| 3.5195282|
+------+-------+------+----------+
only showing top 20 rows
```

We can now generate the top 10 recommendations for all users:

[57]:
```
user_recs_fit = best_model.recommendForAllUsers(10).toPandas().
↪set_index('userId')
user_recs_fit.head()
```

[57]:
```
                                           recommendations
userId
15790   [(3753, 4.374968528747559), (457, 4.2822256088…
78120   [(2324, 4.926319122314453), (318, 4.8594355583…
83250   [(5971, 5.053318500518799), (541, 5.0453348159…
113000  [(296, 3.426694869995117), (2959, 3.4147636890…
18051   [(318, 4.68931770324707), (527, 4.631686210632…
```