Project Report

Part II Deep Dive

IEOR E4571 Personalization Theory and Application

Tian Pan	tp2517
Yiying Huang	yh2870
Zhichao Yang	zy2280

1. Abstract

After writing a simple collaborative filtering algorithms from scratch earlier in the course, we sought to implement a recommendation system at scale. The data we use is the Amazon electronics product data spanning May 1996-July 2014. We used PySpark to analyze the data and implemented three machine learning algorithms.

2. Data Analysis

The raw rating dataset contains 4,201,696 users, 476,002 items and 7,824,482 ratings in total, along with the timestamp of the purchasing time. The rating scale is from 1.0 (hate the product) to 5.0 (love the product). Table 1 is a snapshot of our raw data set. To note that, after examining the data in the training set, we found that no user bought an item twice based on the timestamp. Therefore, we assume each user will only buy a product once, and therefore we will not recommend a product that has been purchased by the user. (Since we won't use the timestamp column in this following modeling process, it will be removed.)

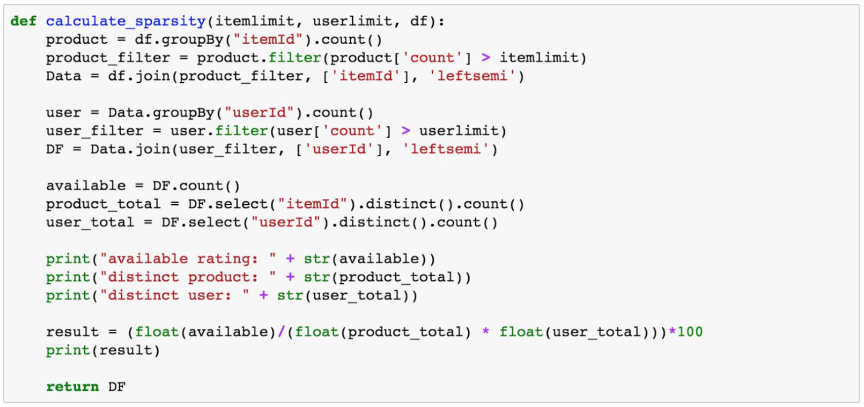
Table 1: Sample of raw data set

```
+--------------+----------+------+----------+
|        userId|    itemId|rating| timestamp|
+--------------+----------+------+----------+
| AKM1MP6P0OYPR|0132793040|   5.0|1365811200|
|A2CX7LUOHB2NDG|0321732944|   5.0|1341100800|
|A2NWSAGRHCP8N5|0439886341|   1.0|1367193600|
|A2WNBOD3WNDNKT|0439886341|   3.0|1374451200|
|A1GI0U4ZRJA8WN|0439886341|   1.0|1334707200|
|A1QGNMC6O1VW39|0511189877|   5.0|1397433600|
|A3J3BRHTDRFJ2G|0511189877|   2.0|1397433600|
|A2TY0BTJOTENPG|0511189877|   5.0|1395878400|
|A34ATBPOK6HCHY|0511189877|   5.0|1395532800|
| A89DO69P0XZ27|0511189877|   5.0|1395446400|
+--------------+----------+------+----------+
```

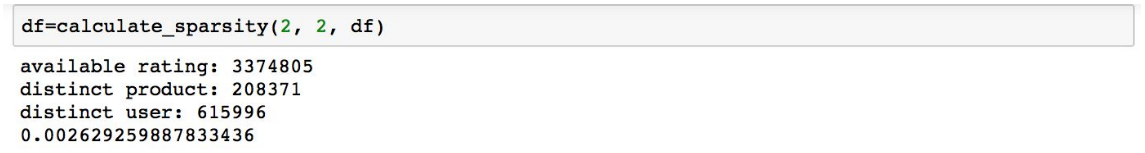
Table 2: Top rated/purchased items

```
+----------+-----+---------------+--------------------+--------------------+
|    itemId|count|          brand|               title|         description|
+----------+-----+---------------+--------------------+--------------------+
|B007WTAJTO| 4983|        SanDisk|SanDisk Ultra 64G...|Perfect for today...|
|B00DR0PDNE| 4315|         Google|Google Chromecast...|                    |
|B003ES5ZUU| 4009|           null|AmazonBasics High...|                    |
|B0019EHU8G| 3697|    Mediabridge|Mediabridge ULTRA...|Mediabridge ULTRA...|
|B0074BW614| 3307|           null|                null|                null|
|B003ELYQGG| 3114|      Panasonic|Panasonic RPHJE12...|Panasonic In-Ear ...|
|B006GWO5WK| 3065|           null|                null|                null|
|B0002L5R78| 2706|       DVI Gear|DVI Gear HDMI Cab...|                    |
|B002WE6D44| 2687|           null|Transcend 8 GB Cl...|                    |
|B009SYZ8OC| 2602|           null|AmazonBasics Appl...|                    |
|B000LRMS66| 2409|           null|                null|                null|
|B00BGGDVOO| 2184|           Roku|Roku 3 Streaming ...|                    |
|B00622AG6S| 2070|           null|PowerGen 2.4Amps ...|The PowerGen 2.4A...|
|B005HMKKH4| 2006|Western Digital|WD My Passport 2T...|                    |
|B002V88HFE| 1855|           null|                null|                null|
|B005FYNSPK| 1785|        SanDisk|SanDisk Cruzer Fi...|With its low-prof...|
|B004QK7HI8| 1766|           Mohu|Mohu Leaf Paper-T...|                    |
|B0041Q38NU| 1735|           null|Kingston Digital ...|                    |
|B000QUUFRW| 1714|        SanDisk|SanDisk 4GB Extre...|The SanDisk Extre...|
|B008OHNZI0| 1660|           null|Tech Armor Ultima...|WHY BUY A TECH AR...|
+----------+-----+---------------+--------------------+--------------------+
only showing top 20 rows
```

We used a function, *calculate_sparsity,* to control the size of our dataset, through this function we can filter out users who rated more than *userlimit* times and item which was purchased more than *itemlimit* times. For our initial analysis, we choose *itemlimit* and *userlimit* both equals to 2, which means each user has rated more than 2 items and each item has been purchased more than twice, and get the subset with 615,996 users, 208,371 products and 3,374,805 ratings. The sparsity of the set is 0.0026%.

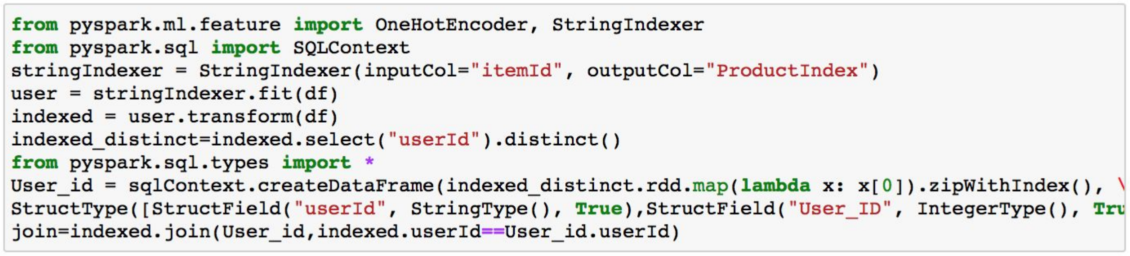


```
def calculate_sparsity(itemlimit, userlimit, df):
    product = df.groupBy("itemId").count()
    product_filter = product.filter(product['count'] > itemlimit)
    Data = df.join(product_filter, ['itemId'], 'leftsemi')

    user = Data.groupBy("userId").count()
    user_filter = user.filter(user['count'] > userlimit)
    DF = Data.join(user_filter, ['userId'], 'leftsemi')

    available = DF.count()
    product_total = DF.select("itemId").distinct().count()
    user_total = DF.select("userId").distinct().count()

    print("available rating: " + str(available))
    print("distinct product: " + str(product_total))
    print("distinct user: " + str(user_total))

    result = (float(available)/(float(product_total) * float(user_total)))*100
    print(result)

    return DF
```

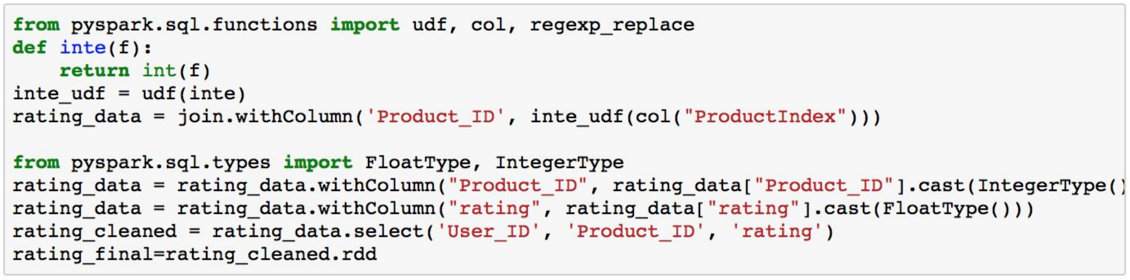


```
df=calculate_sparsity(2, 2, df)
```

```
available rating: 3374805
distinct product: 208371
distinct user: 615996
0.002629259887833436
```



```
from pyspark.ml.feature import OneHotEncoder, StringIndexer
from pyspark.sql import SQLContext
stringIndexer = StringIndexer(inputCol="itemId", outputCol="ProductIndex")
user = stringIndexer.fit(df)
indexed = user.transform(df)
indexed_distinct=indexed.select("userId").distinct()
from pyspark.sql.types import *
User_id = sqlContext.createDataFrame(indexed_distinct.rdd.map(lambda x: x[0]).zipWithIndex(), \
StructType([StructField("userId", StringType(), True),StructField("User_ID", IntegerType(), Tru
join=indexed.join(User_id,indexed.userId==User_id.userId)
```



```
from pyspark.sql.functions import udf, col, regexp_replace
def inte(f):
    return int(f)
inte_udf = udf(inte)
rating_data = join.withColumn('Product_ID', inte_udf(col("ProductIndex")))

from pyspark.sql.types import FloatType, IntegerType
rating_data = rating_data.withColumn("Product_ID", rating_data["Product_ID"].cast(IntegerType()
rating_data = rating_data.withColumn("rating", rating_data["rating"].cast(FloatType()))
rating_cleaned = rating_data.select('User_ID', 'Product_ID', 'rating')
rating_final=rating_cleaned.rdd
```

Since PySpark doesn't support for Long IDs and alphabetical IDs for the ALS model, we used *StringIndexer* and *zipWithIndex()* to assign distinct int values as new user id and item id respectively in our dataset. Table 2 shows the cleaned sample piece of out data.

Table 3: Sample of cleaned data set

```
+-------+----------+------+
|User_ID|Product_ID|rating|
+-------+----------+------+
|      0|     13093|   4.0|
|      0|     10620|   5.0|
|      0|        53|   5.0|
|      1|      2208|   5.0|
|      1|     38247|   5.0|
|      1|      1857|   5.0|
|      1|     45751|   4.0|
|      1|     33783|   2.0|
|      1|      5416|   5.0|
|      2|     14873|   5.0|
+-------+----------+------+
```

We then randomly splitted user-product rating pairs into train set, validation set and test set with percentage 60%, 20%, 20% of the cleaned dataset respectively.

3. Objective

In this project, our ultimate goal is to build a recommender system that will provide at least one valid recommendation for each user. We use the ALS collaborative filtering model and Frequent pattern mining to recommend relative electronic products for users which he/she never purchased before based on ratings and their purchase pattern respectively. And then recommend relevant products to the specific user. We want to find and select the best model as our final model by calculating the "purchasing rate" on the validation set. The "purchasing rate" is defined by:

$$\frac{\textit{number of users made at least one purchase from our recommendations}}{\textit{number users in both (validation or test) and training set}}$$

Since we only selected users and products both having more than 2 ratings to reduce the sparsity of our data, which means our dataset is still very sparse. Therefore, in further analysis, we tried to apply locality sensitive hashing to our data to build the recommendation system.

4. Benchmark

We used baseline bias model as the benchmark in our project. In this model, we recommended top 10 products with the highest predicted rating to users. The model takes into account the mean rating given by each user and the mean rating received by each product in order to remove the presence of any possible bias when predicting because some users may have tendency to rate everything higher or lower:

$$r_{ui} = \bar{x} + b_u + b_i$$

$$\bar{x} = \textit{global average rating}$$

$$b_u = \textit{user bias}$$

$$b_i = \textit{product bias}$$

$$r_{ui} = \textit{predicted rating for user u to product i}$$

We used both train set and validation set together as the training data to train the baseline model. And firstly calculated the mean rating value of the whole training data set.

```
mean=train_valid_df.select('rating').groupBy().avg("rating").take(1)[0][0]
mean
```

```
4.159511466811338
```

Then calculated the user bias and product bias. Table 3 shows the sample user bias data and table 4 shows the sample product bias data.

```
user_bias=train_valid_df.groupBy("User_ID").avg('rating')
user_bias=user_bias.select('User_ID',(((user_bias['avg(rating)'])-mean).alias("user_bias")))
```

```
item_bias=train_valid_df.groupBy("Product_ID").avg('rating')
item_bias=item_bias.select('Product_ID',(((item_bias['avg(rating)'])-mean).alias("item_bias")))
```

Table 4: Sample of user bias

```
+-------+--------------------+
|User_ID|           user_bias|
+-------+--------------------+
|      0|  0.5071551998553288|
|      1|  0.8404885331886618|
|      2| -0.4928448001446717|
|      3|  0.8404885331886618|
|      4|  0.8404885331886618|
|      5|  0.8404885331886618|
|      6|-0.32617813347800473|
|      7| 0.09048853318866179|
|      8| -0.6595114668113382|
|      9|  0.8404885331886618|
+-------+--------------------+
```

Table 5: Sample of item_bias

```
+----------+-------------------+
|Product_ID|          item_bias|
+----------+-------------------+
|         0|0.40516743227123087|
|         1|-0.1884565409177026|
|         2| 0.6097337101880997|
|         3| 0.6250537948461652|
|         4| 0.4643045007123421|
|         5|0.23327605755513314|
|         6| 0.2898029308380252|
|         7|0.47174355058229267|
|         8|0.42012370879443406|
|         9| 0.2307535101471867|
+----------+-------------------+
```

For a specific user, the value of the predicted rating depends only on the product bias (since mean and user bias are the same with regard to a specific user). So in order to recommend the top 10 relevant products, we only need to take top 10 products into consideration, i.e., the products with highest product bias. To do so, we also reduced the model running time significantly.

We take top 20 popular products and recorded the product id so that if the user already purchased a product (rated a product) in our recommendation set, we will remove that item and refill the set with the 11th popular product and so on. Table 5 shows the sample of users in the training set and corresponding recommendation list.

```
popular_item=item_bias.orderBy('item_bias',ascending=False).select('Product_ID').take(20)
popular_item_id=[]
for i in range(len(popular_item)):
    popular_item_id.append(popular_item[i][0])
```

```python
def recommend(*param):
    global popular_item_id
    user_item=[param[0][0]]
    value=[]
    count=0
    for i in range(20):
        if popular_item_id[i] not in param[0][1]:
            value.append(popular_item_id[i])
            count+=1
            if count==10:
                break
    user_item.append(value)
    return user_item
```

```python
recommend_list=[]
for i in range(len(train_list)):
    recommend_list.append(recommend(train_list[i]))
```

```python
train_recommend=sc.parallelize(recommend_list).toDF().select(col('_1').alias('User_ID'),
                col('_2').alias('Recommend_Item_train')).cache()
```

Table 6: Sample of users and corresponding 10 recommend products in training set

```
+-------+--------------------+
|User_ID|Recommend_Item_train|
+-------+--------------------+
|      0|[128593, 167071, ...|
|      1|[128593, 167071, ...|
|      2|[128593, 167071, ...|
|      3|[128593, 167071, ...|
|      5|[128593, 167071, ...|
|      6|[128593, 167071, ...|
|      7|[128593, 167071, ...|
|      8|[128593, 167071, ...|
|      9|[128593, 167071, ...|
|     10|[128593, 167071, ...|
+-------+--------------------+
```

To check the accuracy of the baseline model, we compared our recommendation result with the truly purchase in the test set (we consider user rated a product as the user purchased that product). And then count the true predicted number. We get the result that there are only 9 effective recommendation out of 627056 purchase which means the true positive rate of the baseline bias model is 0.0014%

```python
recommend_num=[]
for i in range(len(final_test_list)):
    recommend_num.append(recommend_count(final_test_list[i]))
```

```python
count=0
for i in range(len(recommend_num)):
    if recommend_num[i][1]!=0:
        count+=1
print(count)
```

```
9
```

5. Algorithms

5.1 ALS collaborative filtering

Alternating least squares collaborative filtering model (ALS) is a two-step iterative optimization process. In every iteration it first fixes P (product matrix) and solves for U (user matrix), and following that it fixes U and solves for P. (Original matrix $R \approx U \times P$) In each step the cost function can either decrease or stay unchanged, but never increase. Alternating between the two steps guarantees reduction of the cost function, until convergence. According to the two-step process, the cost function can be broken down into two cost functions:

$$\forall u_i : J(u_i) = \left\| R_i - u_i \times P^T \right\|_2 + \lambda \cdot \left\| u_i \right\|_2$$

$$\forall p_j : J(p_j) = \left\| R_i - U \times P_j^T \right\|_2 + \lambda \cdot \left\| p_j \right\|_2$$

And the matching solutions for u_i and p_j are:

$$u_i = (P^T \times P + \lambda I)^{-1} \times P^T \times R_i$$

$$p_j = (U^T \times U + \lambda I)^{-1} \times U^T \times R_j$$

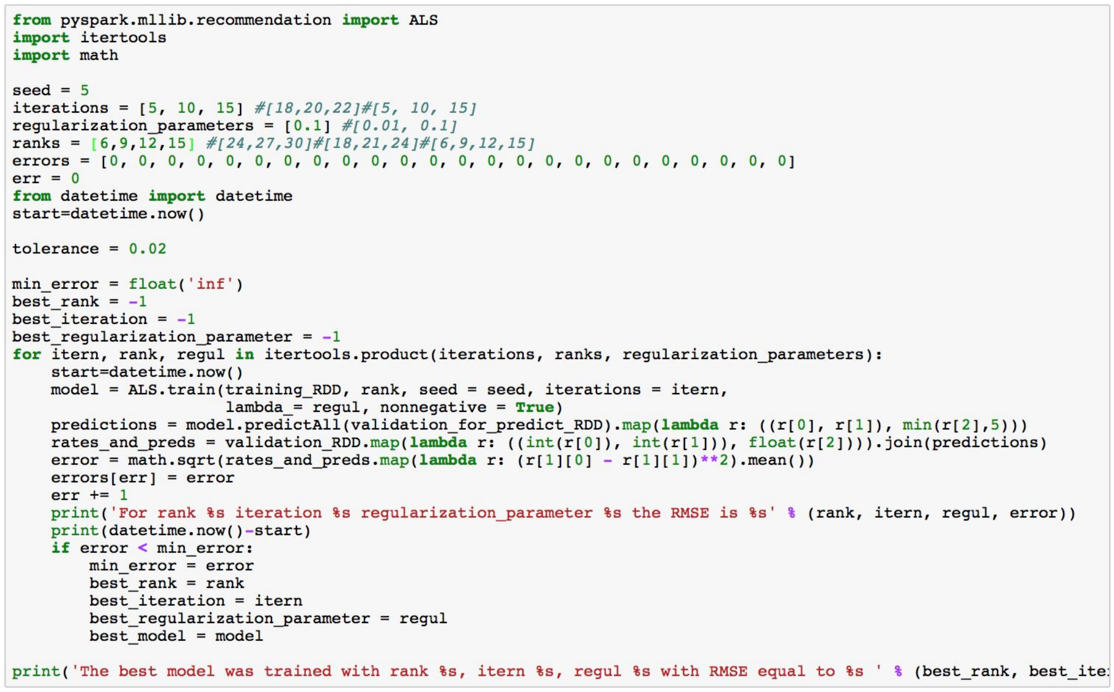
And since each u_i doesn’t depend on other $u_{j \neq i}$ vectors, each step can potentially be introduced to massive parallelization. PySpark’s machine learning library *MLlib* provides an implementation of ALS algorithm. So we implement it to train our model on the train set.

To find the best parameters for our ALS algorithm, we further divide our training set into 75% and 25% sub-training set and sub_validation set respectively. We will try inputting different parameters as number of iterations, value of regularization parameters and ranks to find the best set of parameter that minimize the RMSE on the sub-training set.

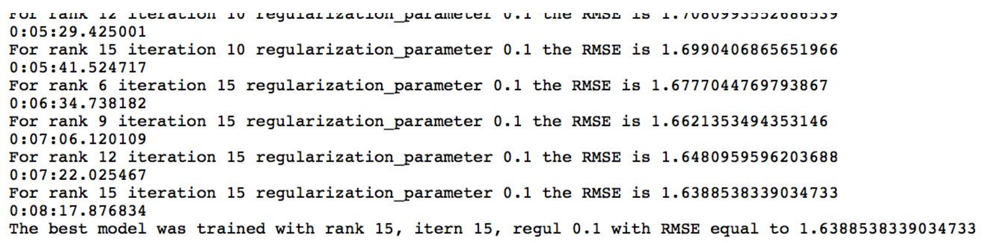
```
als_df = train_df.cache()
als_rdd = als_df.rdd.cache()
```

```
training_RDD, validation_RDD = als_rdd.randomSplit([3, 1], seed=0)
validation_for_predict_RDD = validation_RDD.map(lambda x: (x[0], x[1]))
```

We first try to use the entire sub-training set to fit the ALS algorithm. But the RMSE on the sub-validation set is 1.6389 (with rank=15, iterations=15 and regularization parameter=0.1) which is really high even after grid search for the best parameters.

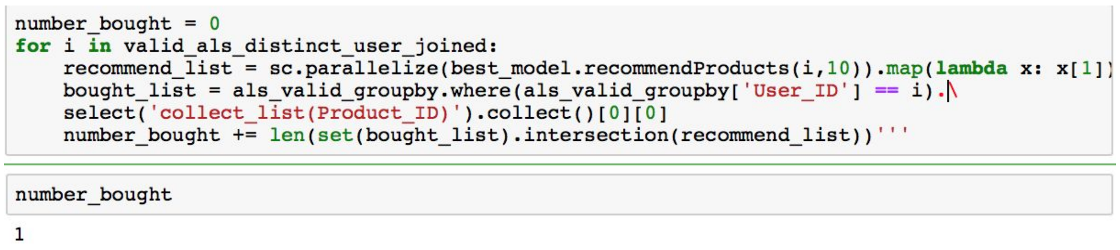


```
from pyspark.mllib.recommendation import ALS
import itertools
import math

seed = 5
iterations = [5, 10, 15] #[18,20,22]#[5, 10, 15]
regularization_parameters = [0.1] #[0.01, 0.1]
ranks = [6,9,12,15] #[24,27,30]#[18,21,24]#[6,9,12,15]
errors = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
err = 0
from datetime import datetime
start=datetime.now()

tolerance = 0.02

min_error = float('inf')
best_rank = -1
best_iteration = -1
best_regularization_parameter = -1
for itern, rank, regul in itertools.product(iterations, ranks, regularization_parameters):
    start=datetime.now()
    model = ALS.train(training_RDD, rank, seed = seed, iterations = itern,
                      lambda_= regul, nonnegative = True)
    predictions = model.predictAll(validation_for_predict_RDD).map(lambda r: ((r[0], r[1]), min(r[2],5)))
    rates_and_preds = validation_RDD.map(lambda r: ((int(r[0]), int(r[1])), float(r[2]))).join(predictions)
    error = math.sqrt(rates_and_preds.map(lambda r: (r[1][0] - r[1][1])**2).mean())
    errors[err] = error
    err += 1
    print('For rank %s iteration %s regularization_parameter %s the RMSE is %s' % (rank, itern, regul, error))
    print(datetime.now()-start)
    if error < min_error:
        min_error = error
        best_rank = rank
        best_iteration = itern
        best_regularization_parameter = regul
        best_model = model

print('The best model was trained with rank %s, itern %s, regul %s with RMSE equal to %s ' % (best_rank, best_ite
```



```
For rank 12 iteration 10 regularization_parameter 0.1 the RMSE is 1.7060993552686539
0:05:29.425001
For rank 15 iteration 10 regularization_parameter 0.1 the RMSE is 1.6990406865651966
0:05:41.524717
For rank 6 iteration 15 regularization_parameter 0.1 the RMSE is 1.6777044769793867
0:06:34.738182
For rank 9 iteration 15 regularization_parameter 0.1 the RMSE is 1.6621353494353146
0:07:06.120109
For rank 12 iteration 15 regularization_parameter 0.1 the RMSE is 1.6480959596203688
0:07:22.025467
For rank 15 iteration 15 regularization_parameter 0.1 the RMSE is 1.6388538339034733
0:08:17.876834
The best model was trained with rank 15, itern 15, regul 0.1 with RMSE equal to 1.6388538339034733
```

Using this model to recommend products for a sample of users in the validation set also gives us a really bad "purchasing rate". We are only able to get user to purchase at least one of our recommended product out of 14,000 users that we give recommendations to while each user are given 10 recommended products.



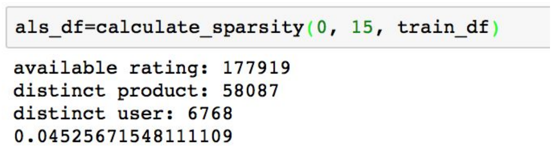
```
number_bought = 0
for i in valid_als_distinct_user_joined:
    recommend_list = sc.parallelize(best_model.recommendProducts(i,10)).map(lambda x: x[1]
    bought_list = als_valid_groupby.where(als_valid_groupby['User_ID'] == i).\
    select('collect_list(Product_ID)').collect()[0][0]
    number_bought += len(set(bought_list).intersection(recommend_list))'''
```

```
number_bought
```

```
1
```

Our large RMSE on the sub-validation dataset with the ALS algorithm and bad "purchasing rate" is due to the sparsity of our dataset. In order to see if ALS algorithm can be better applied for frequent users, we will split the users into 2 types of users: Frequent user and infrequent users and try to apply ALS algorithm only on the frequent users and apply Frequent patterning mining(in the next session) to infrequent users. We will then compare the "purchasing rate" using both approaches for frequent users only, if the Frequent patterning mining model performs better than ALS, we will use FPM for the entire dataset, otherwise, we will combine 2 models and use ALS for frequent users and FPM for infrequent users.

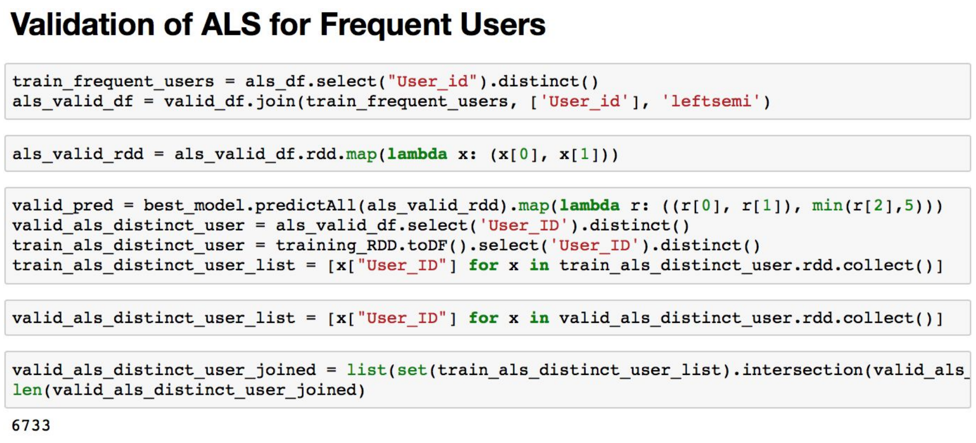
Now we will implement ALS algorithm to only frequent users:
We define users that bought >15 items as frequent users. Using the *calculate_sparsity* function we defined before, we will get a new training set for ALS with only frequent users.



```
als_df=calculate_sparsity(0, 15, train_df)
```

```
available rating: 177919
distinct product: 58087
distinct user: 6768
0.04525671548111109
```

Similar to the process we used above, we found the best model using grid search and the RMSE on the new sub-validation set is 1.2824 (with rank=35, iterations=20 and regularization parameter=0.1).
To apply the best model to the validation dataset, we first filter out users that are not in the training set and we get 6733 users in the validation set:



Validation of ALS for Frequent Users

```
train_frequent_users = als_df.select("User_id").distinct()
als_valid_df = valid_df.join(train_frequent_users, ['User_id'], 'leftsemi')
```

```
als_valid_rdd = als_valid_df.rdd.map(lambda x: (x[0], x[1]))
```

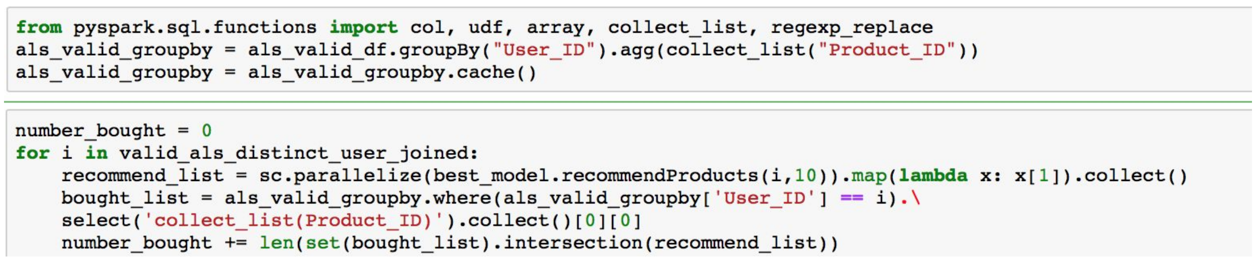
```
valid_pred = best_model.predictAll(als_valid_rdd).map(lambda r: ((r[0], r[1]), min(r[2],5)))
valid_als_distinct_user = als_valid_df.select('User_ID').distinct()
train_als_distinct_user = training_RDD.toDF().select('User_ID').distinct()
train_als_distinct_user_list = [x["User_ID"] for x in train_als_distinct_user.rdd.collect()]
```

```
valid_als_distinct_user_list = [x["User_ID"] for x in valid_als_distinct_user.rdd.collect()]
```

```
valid_als_distinct_user_joined = list(set(train_als_distinct_user_list).intersection(valid_als_
len(valid_als_distinct_user_joined)
```

```
6733
```

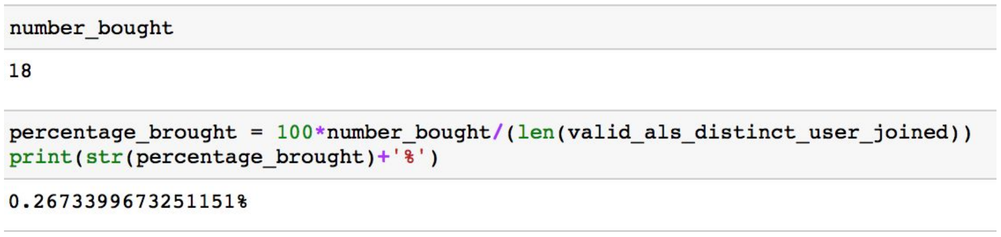
For each distinct user in the validation set, we group by each User_ID and aggregate all the items they rated in the validation set(we assume a user must have brought the product if the user have rated the product), we compare the list of products the user brought and the list of products that we use the best model to recommend for the users, and calculate the number of users that brought at least one on our recommendations.



```
from pyspark.sql.functions import col, udf, array, collect_list, regexp_replace
als_valid_groupby = als_valid_df.groupBy("User_ID").agg(collect_list("Product_ID"))
als_valid_groupby = als_valid_groupby.cache()
```

```
number_bought = 0
for i in valid_als_distinct_user_joined:
    recommend_list = sc.parallelize(best_model.recommendProducts(i,10)).map(lambda x: x[1]).collect()
    bought_list = als_valid_groupby.where(als_valid_groupby['User_ID'] == i).\
    select('collect_list(Product_ID)').collect()[0][0]
    number_bought += len(set(bought_list).intersection(recommend_list))
```

Out of 6733 users, 18 users actually brought recommended product. Therefore, using ALS algorithms for frequent users, the purchasing rate is 0.2673%.



```
number_bought
```

```
18
```

```
percentage_brought = 100*number_bought/(len(valid_als_distinct_user_joined))
print(str(percentage_brought)+'%')
```

```
0.2673399673251151%
```

5.2 Frequent pattern mining

In Spark 2.2.0, *spark.ml* provides a parallel implementation of FP-growth based, a popular algorithm to mining frequent itemsets using spark dataframe. It works in a divide-and-conquer way. The first scan of the database derives a list of frequent items in which items are ordered by frequency descending order. According to the frequency descending list, the database is compressed into a frequent-pattern tree, or FP-tree, which retains the itemset association information. The FP-tree is mined by starting from each frequent length-1 pattern (as an initial suffix pattern), constructing its conditional pattern base, then constructing its conditional FP-tree, and performing mining recursively on such a tree. The pattern growth is achieved by the concatenation of the suffix pattern with the frequent patterns generated from a conditional FP-tree.

The two main parameters of the model are minimum support and minimum confidence. Minimum support is used to define frequent itemset with respect to all transactions. In this project, each user's purchasing history is defined as one transaction. A minimum support of 0.5 means that the itemset appears in half of the users' purchasing history. Minimum confidence is used when deriving association rules.

Figure 1:FP-Tree Construction

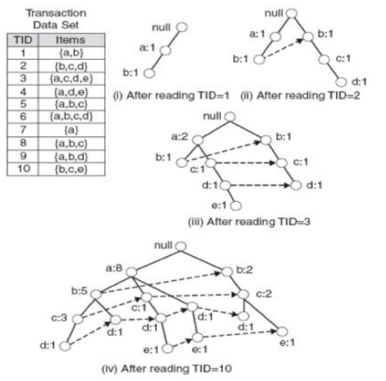


Figure 2:Prefix path sub-trees

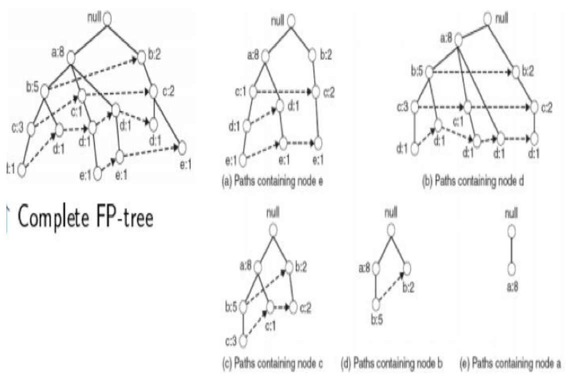

Complete FP-tree
(a) Paths containing node e
(b) Paths containing node d
(c) Paths containing node c
(d) Paths containing node b
(e) Paths containing node a

Figure 3: Frequent itemset generation

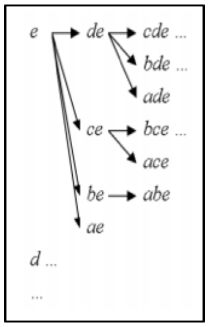
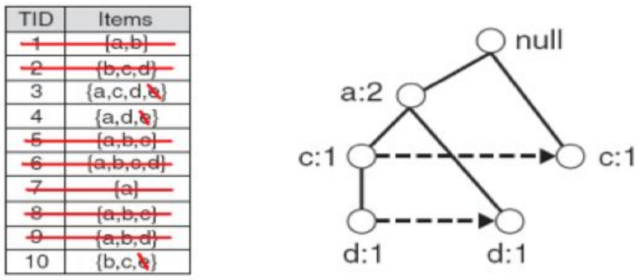


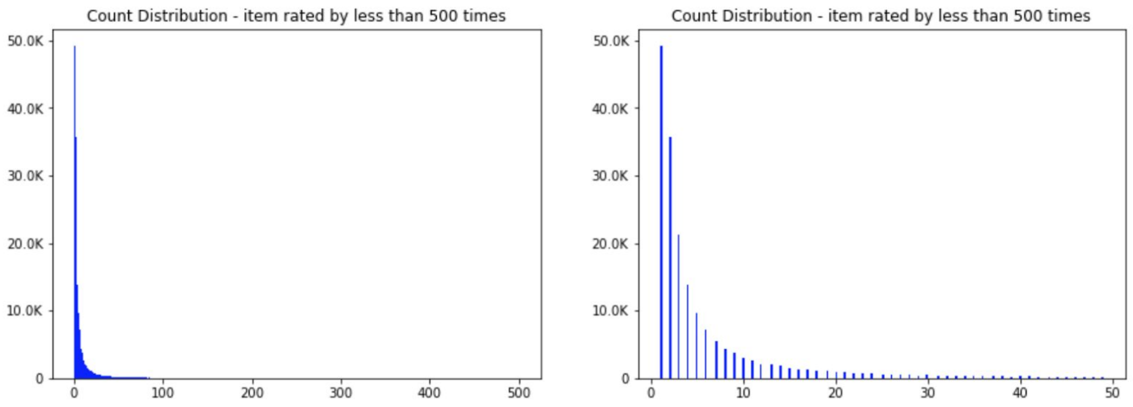
Figure 4: FP-Tree conditional on e



TID	Items
~~1~~	~~{a,b}~~
~~2~~	~~{b,c,d}~~
3	{a,c,d,$\cancel{e}$}
4	{a,d,$\cancel{e}$}
~~5~~	~~{a,b,c}~~
~~6~~	~~{a,b,c,d}~~
~~7~~	~~{a}~~
~~8~~	~~{a,b,c}~~
~~9~~	~~{a,b,d}~~
10	{b,c,$\cancel{e}$}

According to the training set, most of the items were rated by less than 30 times (Figure 5). Therefore, it is reasonable to select a smaller minimum support. we started with some random minimum support of 0.0000085 to explore the model in order to have more frequent itemsets. The training set contains 595,383 distinct users/transactions. Therefore, an itemset is classified as frequent if it appears at least 5 times among all the transactions. And this gave us 9417 itemsets and most of them contains only two items. This number dropped dramatically when the minimum support increases to 0.00002. Therefore, we initially expected to see a better prediction from a smaller minSupport since it provides more possibilities.

Figure 5: Training set item frequency distribution



```
fpGrowth = FPGrowth(itemsCol="Product_ID", minSupport= 0.0000085, minConfidence = 0.5 )
model = fpGrowth.fit(groupby)
```

```
+-------------+-----+
|item_set_size|count|
+-------------+-----+
|            3|  114|
|            4|    1|
|            2| 9302|
+-------------+-----+
fpGrowth2 = FPGrowth(itemsCol="Product_ID", minSupport= 0.00002, minConfidence = 0.5)
model2 = fpGrowth2.fit(groupby)
item_set2 = model2.freqItemsets.filter(size('items') > 1)
count_itemset2 = item_set2.select('*',size('items').alias('item_set_size'))
```

```
count_itemset2.groupBy("item_set_size").agg(count("items").alias("count")).show()
```

```
+-------------+-----+
|item_set_size|count|
+-------------+-----+
|            3|   15|
|            2| 1695|
+-------------+-----+
```

After running the FPM, the results indicated that the model only recommended a small fraction of the products and did not provide recommendation for all users. The model is further extended to incorporate a random combination of popular item (> 100 ratings) and other items to cover more unpopular items and add serendipity. The model randomly selected 8 popular products and 5 other products to recommend. If there is an overlap between the set with original prediction and purchasing history, the overlapped items will not be recommended.

```
import random
def extension(array):  # Randomly recommend 8 items from popular products and 5 items from other produc
    if array[0] == None:
        lst = []
    else:
        lst =  array[0]

    for i in range(8):
        index = random.randint(0,2803)
        if F_item[index].Product_ID not in lst and F_item[index].Product_ID not in array[1] : #and F_it
            lst.append(F_item[index].Product_ID)
    for j in range(5):
        index2 = random.randint(0,184154)
        if NF_item[index2].Product_ID not in lst and NF_item[index].Product_ID not in array[1]  : #and
            lst.append(NF_item[index2].Product_ID)
    return lst

extension_udf = udf(extension)
prediction_ext = prediction.withColumn('prediction', extension_udf(array('prediction','Product_ID')))
```

5. Model Quality

5.1 Alternating least square model

The model was validated with the following parameters.

```
iterations = [5, 10, 15, 20]
regularization_parameters = [0.1 , 0.01]
ranks = [5, 10, 15, 20, 25, 30, 35, 40]
```

We first tested for regularization parameters and noticed that when it is equal to 0.01, the RMSE on the sub-validation set is always worse than when it is equal to 0.1. So we tested other parameter combinations with 0.1 as the regularization parameters. Rest of the parameters and RMSE values are:

```
For rank 5 iteration 5 regularization_parameter 0.1 the RMSE is 1.3096884294501052
For rank 10 iteration 5 regularization_parameter 0.1 the RMSE is 1.3465036993923258
For rank 15 iteration 5 regularization_parameter 0.1 the RMSE is 1.361442008063377
For rank 20 iteration 5 regularization_parameter 0.1 the RMSE is 1.3722740534637663
For rank 25 iteration 5 regularization_parameter 0.1 the RMSE is 1.3785868887655046
For rank 5 iteration 10 regularization_parameter 0.1 the RMSE is 1.310342540917748
For rank 10 iteration 10 regularization_parameter 0.1 the RMSE is 1.3167519254909124
For rank 15 iteration 10 regularization_parameter 0.1 the RMSE is 1.3131183738490348
For rank 20 iteration 10 regularization_parameter 0.1 the RMSE is 1.3096347371106494
For rank 25 iteration 10 regularization_parameter 0.1 the RMSE is 1.3092144823858045
For rank 5 iteration 15 regularization_parameter 0.1 the RMSE is 1.3126894219290142
For rank 10 iteration 15 regularization_parameter 0.1 the RMSE is 1.3080668872068546
For rank 15 iteration 15 regularization_parameter 0.1 the RMSE is 1.3003247869571188
For rank 20 iteration 15 regularization_parameter 0.1 the RMSE is 1.295483679420156
For rank 25 iteration 15 regularization_parameter 0.1 the RMSE is 1.292340227165509
For rank 5 iteration 20 regularization_parameter 0.1 the RMSE is 1.3130607552783002
For rank 10 iteration 20 regularization_parameter 0.1 the RMSE is 1.3036324928789071
For rank 15 iteration 20 regularization_parameter 0.1 the RMSE is 1.294961095080365
For rank 20 iteration 20 regularization_parameter 0.1 the RMSE is 1.290141191964345
For rank 25 iteration 20 regularization_parameter 0.1 the RMSE is 1.2855870840830876
For rank 30 iteration 20 regularization_parameter 0.1 the RMSE is 1.2826146580152509
For rank 35 iteration 20 regularization_parameter 0.1 the RMSE is 1.2824038700152223
For rank 40 iteration 20 regularization_parameter 0.1 the RMSE is 1.283351682960428
```

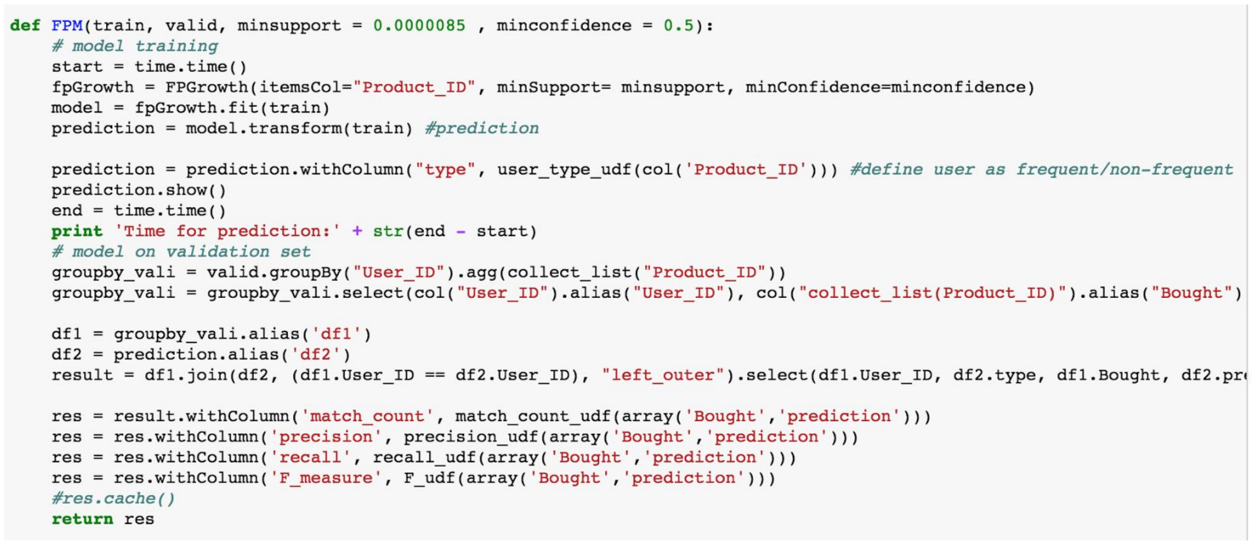
Finally, we found the best model was trained with rank 35, iterations 20, regularization parameter 0.1 with RMSE equal to 1.2824. We will then use these parameter to train the best model to make recommendations for users.

5.2 Frequent Pattern Mining

The model was validated with the following parameter combinations. Users in the training set were labeled as frequent users (bought >15 items) and infrequent users for the purpose of comparing with the ALS model. Two measures match count and F-measure were used for evaluation. Match count indicates the number of user in the validation set that got at least one correct prediction, which means the user bought one of the predicted items. F-measure shows a combination of recall and precision.

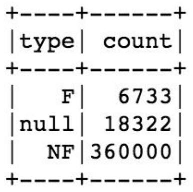
Validation Test	minSupport	minConfide-nce	Frequent User Match Count	Infrequent User Match Count	Infrequent F-measure	Non-Frequent User F-measure	Run time
test1	0.0000085	0.5	36	43	0.0886732	0.0094926	161.29926
test2	0.0000085	0.3	106	453	0.2757530	0.0998886	162.22275
test3	0.0000085	0.1	582	3648	1.30678433	0.6757563	158.85099
test4	0.00002	0.5	5	11	0.0103605	0.0026378	124.37937

test5	0.00002	0.3	42	291	0.1153669	0.0666136	132.55952
test6	0.00002	0.1	287	2528	0.74080625	0.48411485	122.63916
test7	0.000005	0.5	Model Failed	Model Failed	Model Failed	Model Failed	-
test8	**0.000007**	**0.1**	**671**	**3992**	**1.434658562**	**0.727751064**	**192.48248**
test9	0.0000065	0.1	Model Failed	Model Failed	Model Failed	Model Failed	-

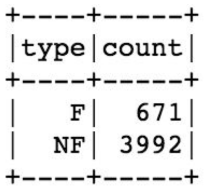


```
def FPM(train, valid, minsupport = 0.0000085 , minconfidence = 0.5):
    # model training
    start = time.time()
    fpGrowth = FPGrowth(itemsCol="Product_ID", minSupport= minsupport, minConfidence=minconfidence)
    model = fpGrowth.fit(train)
    prediction = model.transform(train) #prediction

    prediction = prediction.withColumn("type", user_type_udf(col('Product_ID'))) #define user as frequent/non-frequent
    prediction.show()
    end = time.time()
    print 'Time for prediction:' + str(end - start)
    # model on validation set
    groupby_vali = valid.groupBy("User_ID").agg(collect_list("Product_ID"))
    groupby_vali = groupby_vali.select(col("User_ID").alias("User_ID"), col("collect_list(Product_ID)").alias("Bought")

    df1 = groupby_vali.alias('df1')
    df2 = prediction.alias('df2')
    result = df1.join(df2, (df1.User_ID == df2.User_ID), "left_outer").select(df1.User_ID, df2.type, df1.Bought, df2.pr

    res = result.withColumn('match_count', match_count_udf(array('Bought','prediction')))
    res = res.withColumn('precision', precision_udf(array('Bought','prediction')))
    res = res.withColumn('recall', recall_udf(array('Bought','prediction')))
    res = res.withColumn('F_measure', F_udf(array('Bought','prediction')))
    #res.cache()
    return res
```

After the FPM model, among all users in the validation set, 6733 of them are frequent users, 360000 of them are infrequent users and 18322 of them are not in the training set. The best model turned out to be the one with minimum support of 0.000007 and minimum confidence of 0.1, which successfully recommended products to 4663 users in the validation sets (671 frequent users and 3992 infrequent users). The purchasing rate defined above was then 1.27%. Both the match count and F-measures were the highest among all the models.

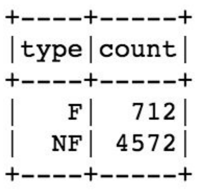


```
+----+------+
|type| count|
+----+------+
|   F|  6733|
|null| 18322|
|  NF|360000|
+----+------+
```



```
+----+-----+
|type|count|
+----+-----+
|   F|  671|
|  NF| 3992|
+----+-----+
```

The training set contains 186,959 items and 2804 of them have been rated for over 100 times. The model recommended 1004 distinct items in total for the test set, which is 0.5% of the total products available. The model, therefore, does not to cover a broad range of products. Therefore, an extension mentioned in 5.2 was added to further improve the model. After applying the second step, the match count increased slightly to 712 for frequent users and 4572 for infrequent users, the purchasing rate was increased to 1.44%.



```
+----+-----+
|type|count|
+----+-----+
|   F|  712|
|  NF| 4572|
+----+-----+
```

5.3 Test Set Results

Since the ALS model did not outperform the frequent pattern mining (FPM) model for frequent users, FPM was used as the final model applied to the test set. The test set contains 385887 users and 367508 of them are also in the training set. Among those users, the model successfully recommended at least one items to 5182 of them. That is the model successfully recommended products to 1.34% of the users in the test set. To exam this result in more detail, 4465 out of 360781 infrequent users (1.23%) purchased recommended products and 717 out of 6727 frequent users (10.65%) purchased recommended products. The purchasing rate was 1.43%. The model worked well for frequent users and also provide some coverage to infrequent users. The performance of both infrequent user and frequent user were improved after the second step in the model.

6. Further Analysis

6.1 Locality sensitive hash algorithm

We implemented locality sensitive hashing (LSH) to further speed up the item similarity calculations. Under LSH, rough similarity measures were used to partition the data into buckets, and more precise similarity measures were calculated within each bucket. With this method we lost some information by partitioning the data into separate buckets, but we reduced the overall runtime in the similarity calculations because it can run in parallel on each bucket.

MinHash is an LSH family for Jaccard distance where input features are sets of natural numbers. Jaccard distance of two sets is defined by the cardinality of their intersection and union:

$$d(A,B) = 1 - \frac{|A \cap B|}{|A \cup B|}$$

Min Hash applies a random hash function g to each element in the set and take the minimum of all hashed values:

$$h(\mathbf{A}) = \min_{a \in \mathbf{A}}(g(a))$$

The input sets for MinHash are represented as binary vectors where the vector indices represent the elements themselves and the non-zero values in the vector represent the presence of that element in the set. So we firstly need to change our data set to sparse vectors for efficiency.

```
train_rdd = train_rdd.map(lambda x: (x.User_ID, [(x.Product_ID, x['rating'])]))
train_rdd = train_rdd.reduceByKey(lambda a, b: a + b)
```

```
import numpy as np
import scipy.sparse as sps
from pyspark.mllib.linalg import Vectors

train_new = train_rdd.map(lambda x: (x[0], Vectors.sparse(maxIndex*2, x[1])))

train_new.take(5)
```

```
[(0, SparseVector(373918, {10620: 5.0, 13093: 4.0})),
 (512000, SparseVector(373918, {641: 4.0, 10429: 1.0})),
 (8200, SparseVector(373918, {44888: 4.0})),
 (16400, SparseVector(373918, {2887: 5.0})),
 (378200,
  SparseVector(373918, {6769: 3.0, 32209: 4.0, 126412: 5.0, 131053: 3.0, 151806: 3.0}))]
```

After changing the data format of our training set, we imported the *MinHashLSH* package in *pyspark.ml* to implement the algorithm. Below shows the code and sample results for finding hashes and Jaccard distance for each user.

```
from pyspark.ml.feature import MinHashLSH
from pyspark.ml.linalg import Vectors
from pyspark.sql.functions import col


dataA = train_new
dfA = spark.createDataFrame(dataA, ["id", "features"])

dataB = train_new
dfB = spark.createDataFrame(dataB, ["id", "features"])

mh = MinHashLSH(inputCol="features", outputCol="hashes", numHashTables=5)

model = mh.fit(dfA)

print("The hashed dataset where hashed values are stored in the column 'hashes':")
model.transform(dfA).show(5)

print("Approximately joining dfA and dfB on distance smaller than 0.9:")
model.approxSimilarityJoin(dfA, dfA, 0.9, distCol="JaccardDistance")\
    .select(col("datasetA.id").alias("idA"),
            col("datasetB.id").alias("idB"),
            col("JaccardDistance")).show(5)
```

```
The hashed dataset where hashed values are stored in the column 'hashes':
+------+--------------------+--------------------+
|    id|            features|              hashes|
+------+--------------------+--------------------+
|     0|(373918,[10620,13...|[[-1.267356126E9]...|
|512000|(373918,[641,1042...|[[-3.43236794E8],...|
|  8200|(373918,[44888],[...|[[1.155334799E9],...|
| 16400|(373918,[2887],[5...|[[-1.343943744E9]...|
|378200|(373918,[6769,322...|[[-1.328428554E9]...|
+------+--------------------+--------------------+
only showing top 5 rows
```

```
Approximately joining dfA and dfB on distance smaller than 0.9:
+---+------+---------------+
|idA|   idB|JaccardDistance|
+---+------+---------------+
|174| 86960|           0.75|
|174|231779|            0.8|
|174|239717|            0.8|
|174|275803|           0.75|
|261| 29816|            0.8|
+---+------+---------------+
only showing top 5 rows
```

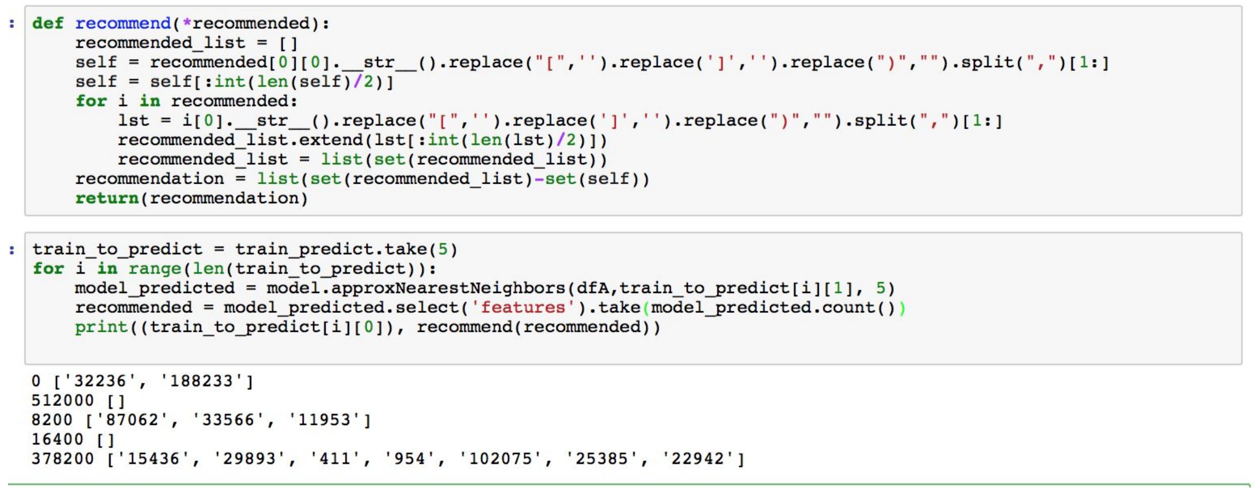
In order to do the recommendation, we then found the nearest neighbour for each user, for example, we took user 512000 as the key to find his/her 50 nearest neighbors.

```
key = Vectors.sparse(512000, [641, 10429], [4.0, 1.0])
#key = Vectors.sparse(0, [10620, 13093], [5.0, 4.0])
model_df = model.approxNearestNeighbors(dfA, key, 50)
```

```
model_df.show()
```

```
+------+--------------------+--------------------+------------------+
|    id|            features|              hashes|           distCol|
+------+--------------------+--------------------+------------------+
|512000|(373918,[641,1042...|[[-3.43236794E8],...|               0.0|
|310373|(373918,[641],[5.0])|[[-3.43236794E8],...|               0.5|
|290078|(373918,[641],[5.0])|[[-3.43236794E8],...|               0.5|
|201958|(373918,[641],[4.0])|[[-3.43236794E8],...|               0.5|
| 33607|(373918,[641],[5.0])|[[-3.43236794E8],...|               0.5|
|248038|(373918,[641],[3.0])|[[-3.43236794E8],...|               0.5|
|558228|(373918,[641],[5.0])|[[-3.43236794E8],...|               0.5|
|100215|(373918,[641],[5.0])|[[-3.43236794E8],...|               0.5|
|479884|(373918,[641],[4.0])|[[-3.43236794E8],...|               0.5|
|551804|(373918,[641],[5.0])|[[-3.43236794E8],...|               0.5|
|396375|(373918,[641],[5.0])|[[-3.43236794E8],...|               0.5|
|591221|(373918,[641,4743...|[[-3.43236794E8],...|0.6666666666666667|
|436616|(373918,[641,4501...|[[-3.43236794E8],...|0.6666666666666667|
|103055|(373918,[641,1612...|[[-1.214341393E9]...|0.6666666666666667|
| 70626|(373918,[143,641]...|[[-3.43236794E8],...|0.6666666666666667|
|400621|(373918,[641,881]...|[[-1.735217066E9]...|0.6666666666666667|
|  8247|(373918,[392,641]...|[[-1.905534369E9]...|0.6666666666666667|
| 63831|(373918,[10,641],...|[[-3.43236794E8],...|0.6666666666666667|
|437810|(373918,[641,5894...|[[-1.79940787E9],...|0.6666666666666667|
|  4840|(373918,[641,1239...|[[-3.43236794E8],...|0.6666666666666667|
+------+--------------------+--------------------+------------------+
only showing top 20 rows
```

Finally, we implemented a function to recommend products that the user's neighbors have purchased. Within the recommended product list, we also removed products that the user have already purchased before.To test our recommendation list, we tried to recommend products for the first 5 users in the training set from their 5 nearest neighbors.



```
: def recommend(*recommended):
      recommended_list = []
      self = recommended[0][0].__str__().replace("[",'').replace(']','').replace(")","").split(",")[1:]
      self = self[:int(len(self)/2)]
      for i in recommended:
          lst = i[0].__str__().replace("[",'').replace(']','').replace(")","").split(",")[1:]
          recommended_list.extend(lst[:int(len(lst)/2)])
          recommended_list = list(set(recommended_list))
      recommendation = list(set(recommended_list)-set(self))
      return(recommendation)
```

```
: train_to_predict = train_predict.take(5)
  for i in range(len(train_to_predict)):
      model_predicted = model.approxNearestNeighbors(dfA,train_to_predict[i][1], 5)
      recommended = model_predicted.select('features').take(model_predicted.count())
      print((train_to_predict[i][0]), recommend(recommended))
```

```
0 ['32236', '188233']
512000 []
8200 ['87062', '33566', '11953']
16400 []
378200 ['15436', '29893', '411', '954', '102075', '25385', '22942']
```

Unfortunately, it took 2 minutes just to recommend products for 5 users from nearest neighbors. We estimated that it would take hundreds of hours to recommend products for all users in the validation set. Therefore, due to the extreme long run time, we will not continue with implementing this model.

6.2 Potential Further improvement

In this dataset, we assume a user has bought a product only if the user rated this product. However, many user purchased the product without giving a rating. Since we are lacking of purchasing data of each user, we might underestimate the "purchasing rate" -- the accuracy of our predictions. If we have more time and the purchasing data available, we can further improve our model and get better prediction accuracy.

7. Conclusion

The prediction accuracy of a recommender system heavily depends on various factors. The properties of a dataset play a major role as well as the parameters of the algorithms. As the results shown in previous chapters, it is obvious that both the ALS model and FPM model are significantly better than the baseline model which accords with our expectation. Furthermore, FPM model outperforms the ALS model for both frequent and infrequent users. Since the FP-growth requires only two scans for mining patterns, the approach is more efficient than the ALS model. Also, given the high sparsity of our dataset, it was no surprise that the FPM techniques performed a lot better than the ALS model and baseline model, since FPM model can find implicit purchasing habits among users. So we conclude that the FPM algorithm is seemingly more accurate than the ALS algorithm.

8. Reference

[1]http://jmcauley.ucsd.edu/data/amazon/
[2]http://spark.apache.org/docs/2.2.0/mllib-frequent-pattern-mining.html
[3]https://spark.apache.org/docs/2.2.0/mllib-collaborative-filtering.html
[4]https://github.com/mrsqueeze/spark-hash
[5]http://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html#pyspark.ml.feature.MinHashLSH
[6]R. He, J. McAuley. Modeling the visual evolution of fashion trends with one-class collaborative filtering. WWW, 2016
[7] J. McAuley, C. Targett, J. Shi, A. van den Hengel. Image-based recommendations on styles and substitutes. SIGIR, 2015