Hybrid Recommendation System with Alternating Least Squares and Clusters of Users using PySpark.

Mauricio S.C. Hernandes, Ph.D. *

Abstract

In this report we lay out a hybrid recommendation system for items in a general e-commerce website based on past purchases of users and users profile information. The system is built in a general way to allow easier integration with real user profile data and item purchase history.

The Recommendation System in this report consists on two main parts: 1) A Collaborative Filter based on the Alternating Least Square method that derives from past purchases recommendations to a given user; and 2) A Content Based recommendation that cluster users in groups and assigns recommendations to users based on which groups they belong to.

The system is implemented in pySpark (using spark 3.0.0) and Python 3.8. It was developed on Ubuntu 20.04 (LTS) x64 with: 256 GB Memory, 32 Cores and 800 GB of SSD Disk.

Keywords: Hybrid Recommendation System, ALS, Clustering, PySpark, Spark

1 Introduction

This report intend to lay out an easy to understand hybrid system combining users past purchases and users profile information to recommend items that they are most likely to buy. A hybrid system is a system that combines collaborative and content based filtering, where **collaborative filter** is a technique that employs all user's past purchases to recommend new items to each user. One of the issues with collaborative filters (or any recommendation system in general) is what is called **cold start**, or when a user that has never done any purchase needs an item to be recommended. Among other ways to circumvent this issue, the user profile information can be used to recommend an item. The technique that employs users information for recommendations is called **content based filtering**. Of course the cold start problem can also happen with content based recommend systems. The way that the proposed system handles users without past purchases or no profile information will be recommending the most popular items at that moment.

2 System Diagram

Figure 1 shows a broad overview of the recommendation system using ALS and kMeans modules for building a user recommended list. ALS comes from the **Alternating Least Square** method.[1] kMeans is the name of a popular clustering method and that we apply to our cluster of users.[2] Both methods will be

*mau@SolvingTheHumanProblem.com

[1]This method was popularized by the Netflix prize in the Paper 'Large-scale parallel collaborative filtering for the netflix prize' by Y Zhou at al. from 2008.

[2]Wikipedia's article about kMeans is a good first introduction to the topic: https://en.wikipedia.org/wiki/K-means_clustering

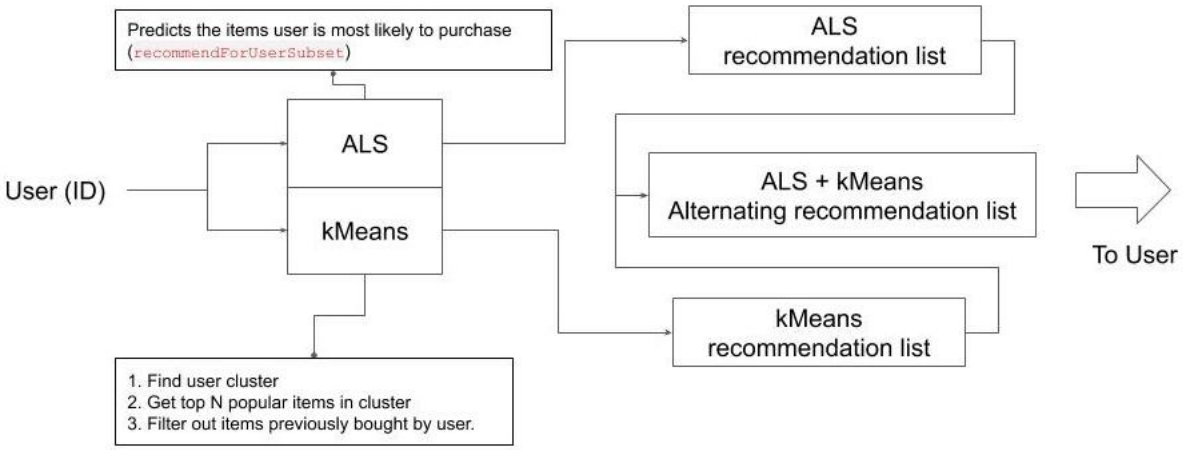

Figure 1: *Hybrid recommendation system. A userid is supplied to the* ***ALS*** *module and to the* ***kMeans*** *module. The kMeans module assigns a cluster to the user based on their profile, then it builds the list of most popular items among users in the cluster, filters out the items already bought by user and returns a list of suggested items. ALS module creates a recommendation list from the user factor calculated from all the users past purchase history. The final list is either the list provided by the ALS list if it is not empty, otherwise it is the kMeans recommendation list. If necessary we complete the the 72 recommended items with the most popular items ranked by past purchase.*

explained later as we walk through the steps to optimize their parameters.

The key point of this section is that no out-of-the box recommendation system exists that integrates these two things 'user past purchase' and 'user information.' The examples readily available often focus in one of these two fronts. The current system is an intuitive and simple way to merge both methods in one system.

3 Evaluation and Model Selection

3.1 Measuring the right Recommendation

In order to select the best models from our candidates we will benchmark each other's model using the **normalized discounted cumulative gains at level 72** (nDCG@72), given by the following formula:

$$nDCG@72(R_u) = \frac{1}{|U|}\sum_{u\in U}\frac{DCG@72(R_u)}{iDCG@72(R_u)}$$

$$DCG@72(R_u) = \sum_{i=1}^{72}\frac{R_u^i}{log_2(i+1)}$$

where:

- U is the set of all users, and $|U|$ is the number of users.
- R_u is the relevance list of the recommended items. R_u^i is the i-th item in the list and $R_u^i \in \{0, 1\}$. A recommendation value is 1 when the user actually purchased the item in the testing data, and 0 otherwise.
- $iDCG@72$ is the **ideal** DCG calculated as if all the recommendations were relevant, i.e., their recommendation value were equal to 1.

[3]Normalized discounted cumulative gains metrics is a standard measure for document retrieval systems. For a

The reference code to this report has implementations of the $nDCG@k$ in the `metrics.py` file.[3]

3.2 Measuring variety

To ensure a certain variety on the recommendation list we can measure the entropy of all the recommendations. The entropy is calculated as follows:

$$entropy@72 = -\sum_{i}^{I} \frac{c_i}{z} log_2(\frac{c_i}{z})$$

where:

- I is a set of recommended items.
- c_i is the number of times item i appeared in the recommendation list fo all users.
- z is the total number of items recommended to all users (in our case we have that $z = |U| \times 72$).

For our purpose in this report we will consider that an entropy of at least 10 will ensure that a large number of items are evenly recommended.

4 Data Structure

The data structure we will assume for this report is as follows:

3 files with the data fields given by:

- `user_master.tsv` is the file containing the user profile and behaviour information: **1,640,956** data points, i.e., users.

  Categorical data fields:

  - `feature1` ranging $\{A1, A2, A3\}$
  - `feature2` ranging $\{B_i : i \in 49\}$
  - `feature3` ranging $\{C_i : i \in 10\}$
  - `feature4` ranging $\{D1, D2\}$
  - `feature5` ranging $\{E_i : i \in 11\}$
  - `feature6` ranging $\{F_i : i \in 10\}$

- `item_history.tsv` is the file containing all the purchases for a certain period of time: **6,319,946** data points.

  Data fields:

  - `user_id` - Unique user ID.

  - `item_id` - Unique item ID.

  - `latest_timestamp` - Timestamp of the item purchase.

  - `frequency` - Cumulative total frequency of purchases.

- `target_users.tsv` a file containing only user IDs for test and evaluation of our models: **250,343** data points.

4.1 Splitting the Data. Train, Test and Validate

We will separate the data as follows for training, testing and validation.

We will split the `item_history.tsv` into 3 sets: 70% of original file to `ihist_train.tsv`, 20% to `ihist_test.tsv` and the remaining 10% to `ihist_val.tsv` for training, testing and validation. The split is done before the data exploration phase to avoid any bias.

The validation step is common in many machine learning system, and it has some issues for recommendation systems, mainly due to the presence of users that never appeared in the training (causing the cold start effect). In other words, slicing the data for validation in our system is not as valuable as it would be for other machine learning system, but we will keep it as a sanity check at the end.

The files `user_master.tsv` and `target_users.tsv` will not be split.

simple example of the this metrics in action, see Pranay Chandekar's article at https://towardsdatascience.com/evaluate-your-recommendation-engine-using-ndcg-759a851452d1

5 Searching for optimum ALS parameters

The idea of a Collaborative filtering is that users that have similar purchase pattern should be interested in similar items.

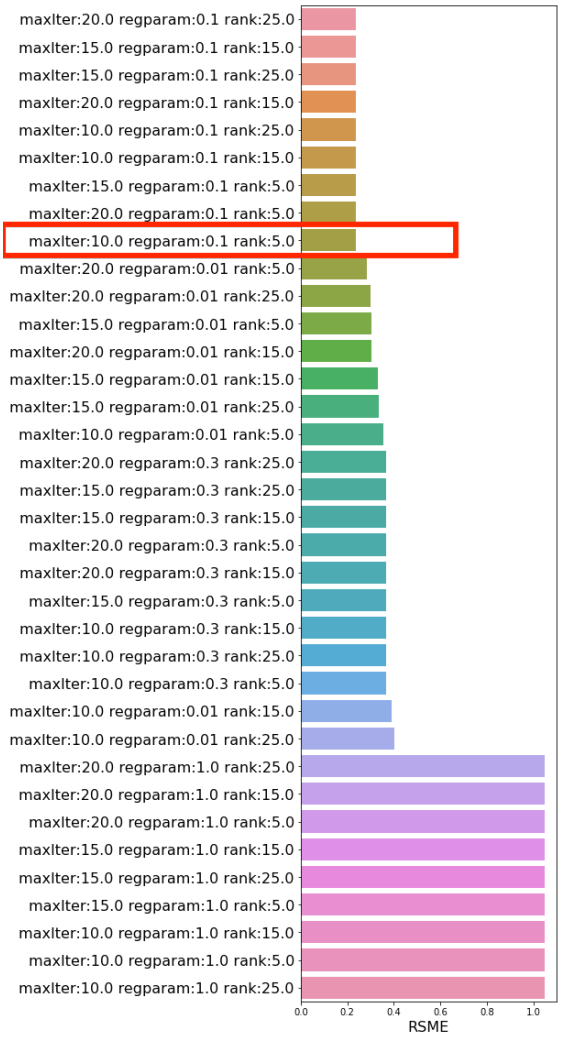

Figure 2: *ALS RSME grid-search result. This analysis suggest the parameters values rank=5, regParam=0.1 and maxIter=10 are good values for our model. However, later in the report we will benchmark these parameters against the nDCG@72 in the test data and we will see that a much larger rank yields a better practical result.*

What the Alternating Least Squares method does is to efficiently estimate what items a particular user would like based on their past purchases. The way it does is in a few steps. First, it builds a similarity matrix, where each column represents an item and each row a user, the cells in the matrix are the ratings that a user explicitly enters in the system (in our case we do not have ratings, but rather number of purchases, this is called implicit rating), call this similarity matrix R. The goal of ALS is to find matrices U and I such that their product approximates the similarity matrix, i.e., $R \approx U \times I$. The matrix U is called **user factors** while the values in I are called **item factors**. We call **rank** the dimension of the user factors space (the number of columns), which is the same as the number of rows in the item factors matrix.

For our purposes we will perform a grid-search to find the best values for the three parameters:

- `maxIter`: Maximum number of iterations performed by the ALS algorithm before stoping
- `regParam`: Regularization parameter, used to avoid overfitting
- `rank`: The dimension of the user factor space (the number of columns in the U matrix as explained before).

The way we measure the performance of each set of parameters is through a 10-fold cross-validation, i.e.,

- 1) Split the training data in ten disjoint sets,
- 2) Traini the ALS model in nine of the ten datasets,
- 3) Calculate the Root-square mean error (RSME) in the remaining tenth dataset.
- 4) Repeat steps 1) - 3) leaving aside each time a different dataset.
- 5) Output the mean of the RSME's obtained by step 4).

Figure 3 shows the result for our cross-validation and grid-search as described. Note that `regParam = 0.1` had the best performance. The main takeaway from this analysis is that a `regParam=0.1` should yield a

good regularization parameter. However, the grid-search plot suggest that a `rank=5` and `maxIter=10` can be good fit to our curve, but it is important to notice that this analysis was the first criteria for parameter tuning and it was ran against the root square mean error metrics. Later in the report we will analise different ranks against the nDCG@72 metrics.

6 KMeans: Users Clusters

The idea of a clustering method for a recommendation system is that similar users should have similar preferences for items to purchase. For that reason, we will use a clustering algorithm to group users in different clusters.

Next we see how to determine an efficient and fair number of groups on a large number of users with sparse features.

6.1 Choosing the number of Clusters

As mentioned before, we will split the users using a traditional clustering method called **kMeans**.[4] Given a dataset with features shown in section 4 we will perform a silhouette analysis to decide what is the best number of clusters for the data based on the silhouette score. The silhouette score is an average of the euclidian distance, in the sparse features space, between each data point and the center of its closest neighbouring cluster (called b_i for data point i) minus the distance from the data point to its cluster's center (called a_i for data point i), divided by the maximum between a_i and b_i. See the formula below, for $s(i)$, the **silhouette score** of data point i:

$$s(i) = \frac{b_i - a_i}{max(a_i, b_i)}$$

We define $s(i)$ to be zero when there is only one point in i's cluster. One way to see how the silhouette score would change if the number k of clusters were to be too high is as follows:

"...suppose that we have set k too high. Then some natural clusters have to be divided in an artificial way, in order to conform to the specified number of groups. However, these artificial fragments will typically also show up through their narrow silhouettes. Indeed, the objects in such a fragment are on the average very close to the remaining part(s) of their natural cluster, and hence the 'between' dissimilarities b_i will become very small, which also results in small $s(i)$ values.[5]"

	K	silhouete score
0	100	0.246800
1	200	0.296600
2	300	0.324200
3	400	0.387600
4	500	0.412800
5	600	0.459400
6	700	0.477700
7	800	0.509100
8	900	0.545300
9	1,000	0.565200
10	2,000	0.737595
11	3,000	0.827398
12	4,000	0.889325
13	10,000	0.982678
14	100,000	0.997158

Figure 3: Silhouette Analysis of KMeans clustering over our dataset.

Figure 3 shows our silhouette score for k clusters for k from to 100 to 100,000. Note that the score approaches 1 asymptotically which means that there is little to no advantage to choose a k equal to 100,000 over 10,000. The reason for this increasing silhou-

[4] Once again, the wikipedia article is a good introduction for the reader new to this unsupervised classification method: https://en.wikipedia.org/wiki/K-means_clustering

[5] Extracted from 'Silhouettes: A graphical aid to the interpretation and validation of cluster analysis' by Peter J.Rousseeuw, 1987 https://reader.elsevier.com/reader/sd/pii/0377042787901257

ette value might be explained by the sparsity of the feature space. If that is the case, the distance between points is so large, that the b_i term overshadows the relevance of a_i in the $s(i)$ formula, in essence transforming the silhouette formula to $s(i) \approx b_i/b_i = 1$. For a more proper analysis one should plot the silhouette plots for each cluster and see what is causing this increasing value. But since 100,000 classes would be impractical to visualise, we leave this analysis as is and select a k between 3,000 to 5,000 as their model fitted to the data reasonably fast and have similar performance to a k equal to 10,000 or above.

7 Limitations and Honorable Mentions

One of the key issues during the development of this system was the scalability for the Content Based Filtering. Most standard libraries were taking too long for training and/or predicting as the number of entries increased. Some attempts worth mentioning are XGBoost[6] and SKlearn's stochastic gradient descent (SGD) Classifier.[7]

SGD at a first glance seemed a good candidate as it allows partial fitting, i.e., we were have to load only chunks of the data in memory at each step. However, the training time increased to unpractical levels as the training data increased, so much so that training with the whole dataset could have taken a week just for the first epoch. Hence, this method was abandoned.

XGBoost had an issue with the prediction time. This parallel boosting model has been growing in popularity in the past few years as it has been performing at high levels in many data science competitions, however, as the number of features grew predicting the classes took longer, making this method inefficient for the amount of items present in the training data.

Therefore, due to performance we chose kMeans for the the Content Based Filtering since its training and prediction were more scalable in practice.

8 Our Results

In this section we will present our empirical results.

8.1 Hardware Spec and Version

Digital Ocean[8] Virtual Machine with:
Ubuntu 20.04 (LTS) x64
- 256 GB Memory
- 32 Cores and
- 800 GB of SSD Disk
Running Spark 3.0.0

8.2 Single Models

As we fitted ALS and kMeans models with different parameters we measured their performance with the nDCG@72 and entropy@72. The results for the best *individual* models within reasonable training time[9] are presented in table 1.

At first, looking at the table one can see that the ALS models performed better than the kMeans models. However, note that this does not mean that we should deploy a recommendation system with an ALS model as its single recommendation technique. Reason being that the nDCG@72 as presented is calculated only on the users that received a recommendation, i.e., ALS in our training 'dropped' users that had no previous purchase during training (that is the meaning of

[6] https://github.com/dmlc/xgboost
[7] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html
[8] https://digitalocean.com
[9] Given the goal of this report was to write and develop the system within 10 days, we ignored models that would require longer than one hour for training to keep the work to schedule. If this report were to be taken further, one natural step would be to increase the training time of some of the most promising models illustrated in this report.

the parameter `coldStartStrategy="drop"` in our `als_trainer.py` file). We still need a strategy for the remaining users, hence the kMeans model.

Model and Parameter	nDCG@72	Entropy@72
k-Means k=3000	0.0010	14.29
k-Means k=5000	0.0011	14.30
ALS rank=100	0.013	10.16
ALS rank=200	0.014	10.62

Table 1: *Entropy@72 and nDCG@72 for each individual model with different parameters. Both ALS models are set with maxIter=20 and regParam=0.1 varying only the number of ranks.*

Remember that in section 5 we had a small RSME for ALS models with lower ranks (like 5 or 10). However, when measured against the nDCG metrics, their values were 100 times lower than higher ranks (100 to 200), hence we decided for higher ranks. The performance seemed to increase as we increased ranks, which makes sense when you think on the size of dimension of the problem (number of users and items), however a rank of 200 was at the limit of one hour training time, hence we stoped there. Finally, the analysis of section 5 helped us to feel confident with the other parameters, like `regParam=0.1` and a `maxIter=20`.

Next, we combine kMeans and ALS models into a complete recommendation system.

8.3 Complete System

Follows our results on the test data of different parameters for our kMeans and ALS models. The models are pre-trained in the `models` folder accompanying this report.

Parameters	nDCG@72	Entropy@72
k=3000 rank=100	0.007964	12.20s
k=3000 rank=200	0.007957	12.43
k=5000 rank=100	0.007972	12.21
k=5000 rank=200	0.007965	12.45

Table 2: *Entropy@72 and nDCG@72 for our recommendation system where 'K' is the number of clusters in the kMeans model and 'rank' is the dimension of user features for the ALS model. The difference between all the options is not expressive, but we will choose the values of k=5000 and rank=200 as our parameters of choice. This choice is not based on the 'bigger parameters is better', but from the idea that we might be under-fitting our models due to the limit of training time we have.*

The main contributor for the low score were users that had zero items correctly recommended. See figure 4 for a picture of how the distribution of nDCG's look like. An interesting fact was that when we calculated and plotted the nDCG's when the model predicted to users in the *training* set, we saw a very similar distribution (very heavy zero correct recommendations). This suggested that the model had not fully learned the complexity of the data, i.e., under-fitting due to the low value of the parameters (cluster number and rank).

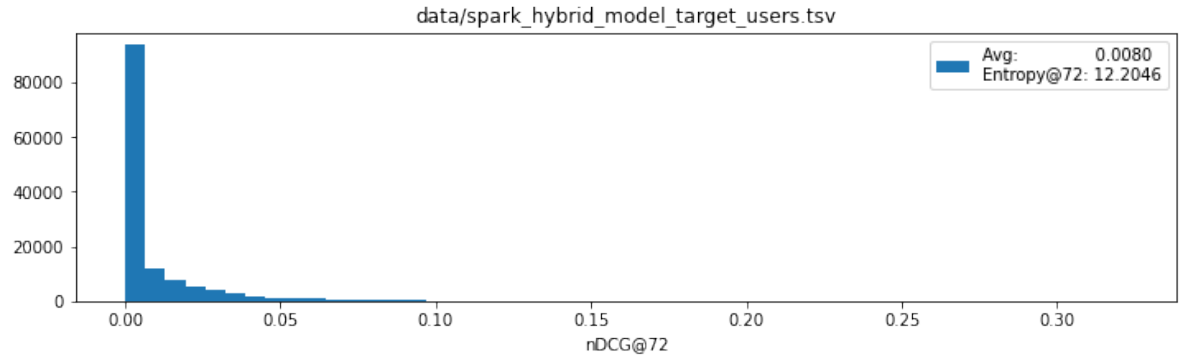

Figure 4: *Distribution of nDCG@72 of the test dataset for the complete recommendation system (ALS + kMeans) with K=3000 (clusters) and rank=100. The 'culprit' for the low nDCG score is due to the high number of users that the system keeps getting zero correct recommendations. As we increased the number of ranks and clusters, we could see more and more users shifting from the zero threshold to the right on the plot.*

9 Conclusion

The present report proposed a simple hybrid recommendation system for items to be purchased developed on Spark 3.0.0. The system used a Content and Collaborative filtering approach in a natural and quick to train way.

Among some of the limitations for the current system are the limited number of features utilized for the Content based filtering and the number of executors in spark. Both limitations go hand-in-hand as the decision of leaving features out was based on the time to train. As our single machine spark 'cluster' had only 32 cores we were limited at the number of permutations and features we could add to the model and still train in a reasonable time. As we were to add more machines (and more cores) to our cluster, one would expect the model complexity and the quality of the recommendations to increase.

Finally, despite the low average score from the resulting system, the current report lays a clear path to increase performance given more resources (time and computation). Given the tight schedule and sparse data, the results have a clear trend upwards.