Implicit RecSys in PySpark on the Million Song Dataset

Farris Atif (fda239), Zafir Momin (zm2114)

Intro

The Million Song Dataset (MSD) was part of a Kaggle challenge that explored building collaborative filter models to recommend songs to users that they would prefer to listen to. The MSD is built upon implicit feedback, or feedback that provides passive information about its users, for example play counts as compared to explicit 5 star. This paper discusses building a recommender system in PySpark using an implicit Alternating Least Squares algorithm, or ALS for short. Briefly, the data is shaped as a User-Item Matrix, R, and the ALS model aims to create a user-matrix, U, and a item matrix, V. These matrices are optimized one at a time while holding the other constant to converge on an approximation for R which is then used to make recommendations!

Project Initialization

The data given in the train, validation, and test set splits was of the form shown in Figure 1.

	user_id	count	track_id
0	b80344d063b5ccb3212f76538f3d9e43d87dca9e	1	TRIQAUQ128F42435AD
1	b80344d063b5ccb3212f76538f3d9e43d87dca9e	1	TRIRLYL128F42539D1

Figure 1: *The schema of the data in the three parquet files*

The *user_id* and *track_id* columns shown are hashed to uniformly unique and unknown users/artists. For the ALS model, these must be formatted to integers. However, the hashing must be consistent across the different datasets to ensure recommendations were executed on the corresponding user's preferences. PySpark's StringIndexer module was used to hash each of the columns to unique float values which were then casted to integers. To maintain consistency and accuracy, a pipeline was built to perform this operation fitted to the training set. This training model pipeline was then saved and loaded to transform the validation and test sets. As we are not dealing with the cold start case, the "invalid handles", or handles that the StringIndexer has not seen before, were skipped, and dropped from the test and validation sets. Each of the new parquet files were then stored to personal HDFS paths.

ALS Model Build

Spark's ML & mllib library both contain classes which are optimized for Alternating Least Squares (ALS) factorization and subsequent recommender predictions. While both libraries offer similar functionality, this implementation utilizes the ML library. This was chosen over the legacy MLLIB because the former makes use of Spark DataFrames - which are inherently easier to interpret and manipulate. The ease-of-use that the ML library ALS offers are immediately apparent when initializing the ALS class, which allows the user to specify which DataFrame columns correspond to user/track/count data, as well as the hyperparameters of the ALS method.

ALS CLASS

```
1  als = ALS( rank = i[3],  maxIter=i[2] ,  regParam=i[1],  userCol="userId",  itemCol="trackId",
       ratingCol="count",  alpha=  i[0] ,  implicitPrefs = True,  coldStartStrategy="drop")
3  model = als.fit(train)
```

Once the class is initialized & the model is fit to our training data , track recommendations are ready to be made for previously unseen/empty entries of the utility matrix using this learned representation. The code block below summarizes this process, wherein a DataFrame of users and recommended tracks is created, then joined onto the validation set using the “userId " column as a join key! (Lines 1-3)

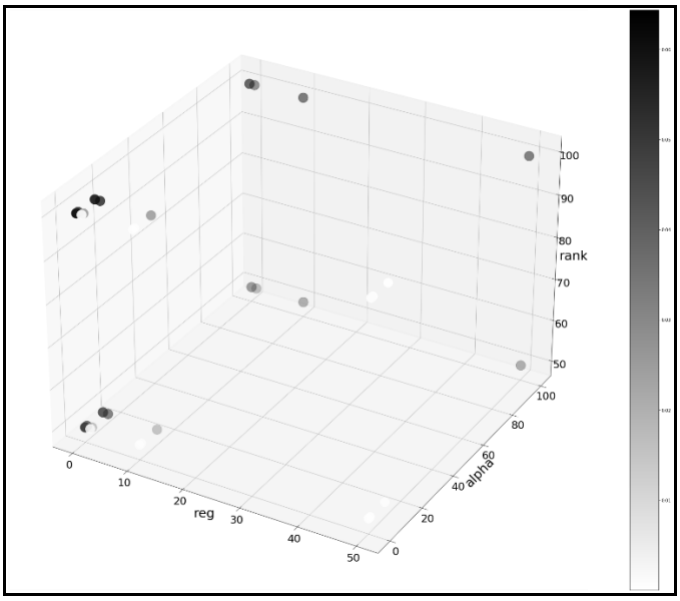
Lastly, this ground truth/validation DataFrame is converted to an RDD, as the scoring metric that was chosen is only compatible through Spark as an RDD (via the MLLIB library). (Lines 4-6)

PREDICTIONS AND SCORING

```
1  userSubsetRecs = model.recommendForUserSubset(users, 500)
2  userSubsetRecs.select("userId","recommendations.trackId")
3  k = userSubsetRecs.join(val1,"userId")
4  k = k.rdd.map(lambda row: (row[1], row[2]))
5  metrics = RankingMetrics(k)
6  precision.append(metrics.meanAveragePrecision)
```

Model Tuning & Results

To optimize the performance of the model, a comprehensive grid search was created to identify optimal hyperparameters. While there are 4 hyperparameters within the ALS matrix factorization space (alpha, regularization parameter, rank, and no. iterations) - only the first three were chosen for the initial grid search. This was done assuming that once optimal alpha, regularization parameter, and rank values were obtained - then drilling down on the iterations for the resulting hyperparameter set would indeed produce the best results of them all. Moreover, this allowed us to expedite the grid search, since the initial search was holding the number of iterations constant & at a low value (3). The 3D plot below shows our grid search mapped into space. Additionally, the table shows the same grid search as well as the results of drilling down the number of iterations on the best hyperparameter set. The best Mean Average Precision (MAP) achieved was .065.



Alpha	Reg. Parameter	Rank	Iterations	MAP
10	1	100	3	0.0600
0.01	0.1	100	3	0.0606
100	10	100	3	0.0640
100	10	125	20	0.0653

Figure 2: *Grid Search Plot & Tabulated Format of Results*

Extension

The results we achieved from the parallel computation performed on the cluster were compared to a single machine implementation in Lenskit on a local machine. Without the parallel computation, the local machine is not able to run ALS algorithms on such large datasets. For this reason, 0.05% of the data was first randomly sampled from the cluster from the training and test sets which created our local train and test datasets. Since Lenskit does not have an evaluation metric for MAP, the precision metric was used as a comparison. After creating our model using Lenskit, the runtime and precisions were calculated for various sizes of the training dataset. This is shown in Figure 3a and b.

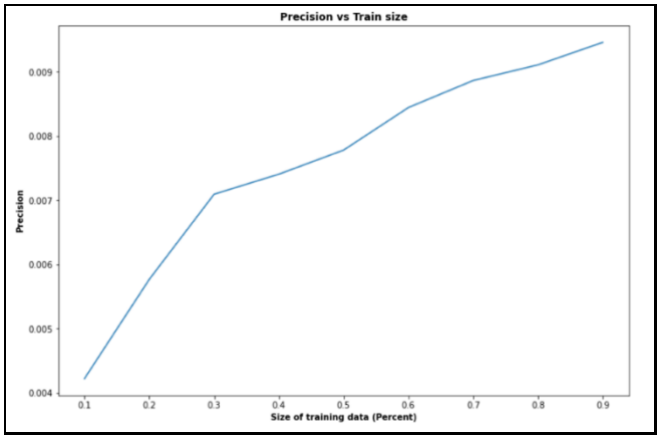


Figure 3a: *Precision as a function of the size of the training dataset*

Here, the precision increases at a high rate as our training data size increases. It would be safe to assume that further increasing the size of the training data would increase the precision of our model. However, the local machine struggled to run the larger levels of the training dataset above indicating that the upper limit of dataset size is not much higher than the current full training set.

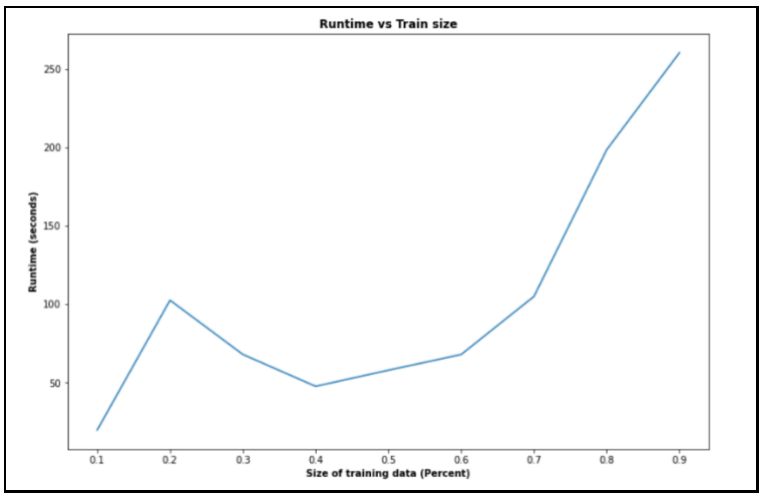


Figure 3b: *Runtime as a function of the size of the training dataset*

The runtime mostly increases with the increasing size of the training dataset. Further increasing the dataset size would make the runtime increase significantly assuming the data is small enough to be run locally. Overall, the best precision was approximately 0.0098 showing that the smaller sample size drastically reduces the performance of the model.

Conclusion

Pyspark's ALS functionality allowed us to explore the nuances of creating an implicit recommender system while also reaping the benefits of Spark's efficiency with handling vast quantities of data. While the final metric was not as accurate as hoped for, the process of creating a model & tuning it provided significant understanding as to how these systems are initialized and perform. Moreover, the low accuracy and slower run times of the Lenskit package highlight the efficiencies of Spark, column-oriented files, and working on a cluster. Lastly, building this baseline model provides us with a foundation to further tune and possibly improve the accuracy of the predictions in the future.

References:

https://lkpy.readthedocs.io/en/stable/index.html
https://spark.apache.org/docs/2.4.7/ml-collaborative-filtering.html
https://spark.apache.org/docs/2.4.7/mllib-evaluation-metrics.html#ranking-systems

*Note that **all work** was completed together by both partners Zafir & Farris. We worked together on everything by screen sharing on Zoom. Individual github branches are available as well, for further examination.