1004 Big Data Final Project

Team SKlearn: Sheetal Laad (sl7054), Kelly Sooch (kas1080)

Implementation

Transformations

The transformations themselves were relatively simple to implement, however with the amount of data provided, we ran into memory issues. In the section “Downsampling Data / Partitioning” we describe how we worked around that issue. Since the ALS model that we wanted to implement only supports integer fields for user and item ids, we had to transform the user_id string column and the track_id string column. We used StringIndexer, which encodes a string column of labels to a column of label indices. We did the following two transformations:

- Applying a StringIndexer on the user_id column
- Applying a StringIndexer on the track_id column

Downsampling Data/partitioning

With roughly 49 million records in the training set parquet file, it was not possible to use the entire dataset without getting a memory error. In order to work around that we experimented with a few alternative techniques learned in class, a few listed below:

- default partitions
- 1000 partitions by count
- 1000 partitions by user
- 10000 partitions by user
- 50000 partitions by user

After trying the methods above and still running into memory issues, we decided to try downsampling to more rapidly prototype our model. While downsampling the data, we had to be mindful that the downsampled data includes enough users from the validation set to test our model. We took the following high-level steps to downsample accurately:

1. Obtained all the records from the train dataset of the users that exist in the union of the test data and validation data
2. Randomly sample an additional 5% of the records generated from the users in train that do not exist in validation or test data
3. Combine the two sets of data to use as our new training set

By downsampling this way, we could ensure that data for all the users in the validation and test data would be used to train the model.

Model

Collaborative filtering is a common technique used for recommender systems, in which the goal is to fill missing entries of a user-item association matrix. In this method, users and items are described by a small set of latent factors that can be used to predict missing entries. We used the alternating least squares (ALS) algorithm to learn the latent factors. Our data is a form of implicit feedback, as it is the number of times a user has listened to a track (“count”). Instead of trying to model the matrix ratings directly, ALS uses the ratings to represent strength in observations of user actions, which are then related to the level of confidence in observed user preference. Then the model tries to find latent factors that can be used to predict the expected

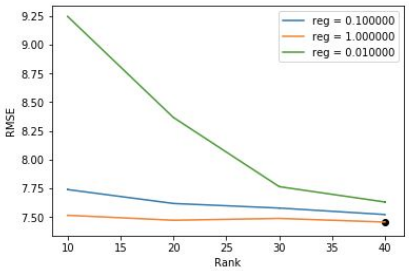
preference of a user for an item. In ALS, this can be done very easily by setting implicitPrefs = True and setting nonnegative = True when building the model.

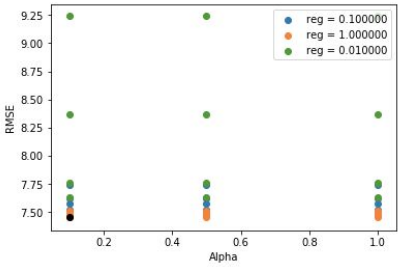
Model Selection

An important task in machine learning is model selection / parameter tuning. In past homeworks, we weren't given a separate validation file, so we were able to use the CrossValidator split the data into training and validation sets, and then ParamGrid to help construct the parameter grid. Since we were given a validation dataset for this project, we didn't need to use the CrossValidator and decided to build our own grid search. Our process was to train model on a given set of parameter combinations, and validate on the validation set, retrieving the root mean square error (RMSE) value as an indication of good our model is, which we obtained using the RegressionEvaluator of Spark mllib. Our first-pass parameter values were:

- Rank: 10, 20, 30, 40
- Regularization: .01, 0.1, 1.0, 10
- Alpha: 0.1, 0.5, 1.0

(results for implicit = False)





Results for the parameter values listed above can be seen in the graphs above. Our best model (Rank = 40, Regularization = 1.0, Alpha = .10), determined by the model that gave the lowest RMSE value, gave an RMSE value of 7.457310 on the validation set. The graphs also demonstrate that Rank and Regularization seem to have the most effect on RMSE values. Once we got validation results for the best model, we wanted to narrow the ranges of the rank, regularization, and alpha around those from the best model in order to get an even stronger model. Our second-pass parameter values were:

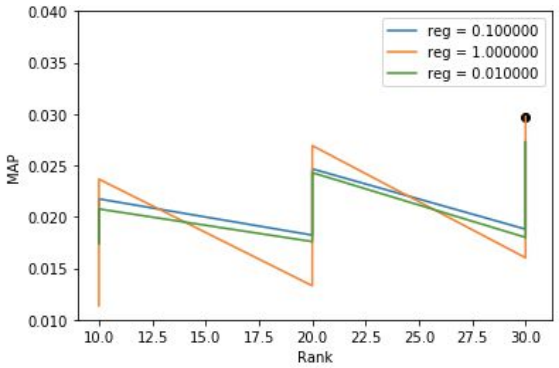
- Rank: 38, 40, 42, 44
- Regularization: .05, .1, .2, .3
- Alpha: .10

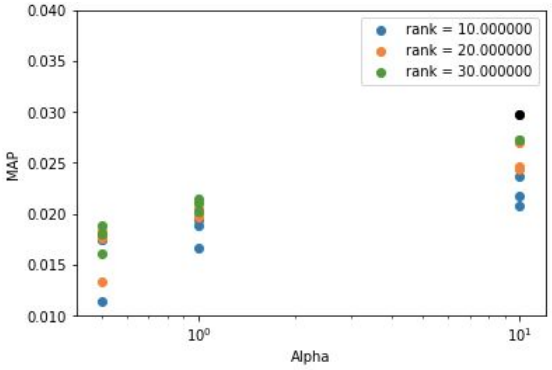
Upon further review of our results, we recognized that rank metrics would be better to tune the ALS model, since we have implicit feedback in our data. In order to complete this, we have to use predictions from the ALS model's recommendForUserSubset method for the top 50 items. Here, our evaluation metric was the mean average precision (MAP) for top 50 items. MAP is a measure of how many of the recommended items are in the set of true relevant items, where the order of the recommendations is taken into count. In this scenario, we were looking for the parameters that maximized the MAP value. The parameter values were:

- Rank: 10, 20, 30
- Regularization: .01, .1, 1.0
- Alpha: .5, 1, 10

However, we tried to run this hyperparameter grid search starting on Thursday (5/9) and our jobs that once were working, started to fail very often. Even after downsampling further, and

allocating memory, our grid search continued to fail. We decided to utilize Google Cloud, since Dumbo's cluster was overallocated.





Results for the parameter values listed above can be seen in the graphs above. Our best model (Rank = 30, Regularization = 1.0, Alpha = 10), determined by the model that gave the lowest RMSE value, gave an MAP value of 0.029750 on the validation set.

Evaluator

Once our model was trained, we wanted to evaluate it on the test set using the RankingMetrics function that is provided by Spark mllib. Given that the requirement was to evaluate on predictions of the top 500 items for each user, we opted to use recommendForUserSubset() method of the ALS model to generate the top 500 recommended items for each user in the test set. Initializing the RankingMetrics was a challenge, as we had to join the recommended set of items with the truth set, which was computationally expensive. In order to utilize the RankingMetrics, we had to transform the test data so that each row included the user with the list tracks relevant to that user in descending order. Then we joined this dataframe to the set of recommended items for each user, and transformed the resulting dataframe into an rdd. Then we mapped each row of the rdd to a tuple of (recommended set of items, truth set of items) pairs for each user, and passed the resulting rdd to RankingMetrics. While we primarily used the meanAveragePrecision metric to evaluate the model on our test data, we also looked at precisionAt(k) and ndcgAt(k) values. The precisionAt function computes the average precision of all the queries, truncated at ranking position k, while ndcgAt computes the average normalized discounted cumulative gain value of all the queries, truncated at ranking position k. We set k = 500 for these values.

Evaluation Results

As mentioned above, due to constraints of the cluster, we were unable to tune our model with MAP as the evaluator, and instead used our best model that was tuned based on RMSE (Rank = 30, Regularization = 1.0, Alpha = 10). The following were our results:

- MAP: 0.034869
- Precision: 0.007446
- NDCG: 0.143481

While we had hoped to get better results, the following likely resulted in our low MAP, precision, NDCG:

- Due to downsampled training data, we are likely to have lost information about a user and/or item that could have been helpful for the model to have.
- We would have done more hyper-parameter tuning, with higher rank values and alpha values, if we hadn't run into computational issues.

Extensions

The extension that we chose to do was the *Alternative model formulations*. We experimented with 3 different datasets:

- Removed all training records with counts of 1
  Of the 49824519 records in the unique set, 29595102 had count 1. After removing the counts, we wanted to make sure that even after downsampling, we had roughly the same count of records as we did in the original downsampling scenario, so that we could stay consistent when training our model. In order to do that we experimented with the random sampling of the additional training data (Step 2 described in Implementation). Roughy a 25% sample brought us to the right amount of data. Parameter values we tested are given in the table below.
- Removed all training records with counts of 5 or less
  Of the 49824519 records in the unique set, 44640416 records had count 5 or lower. Through experimentation described above, 30% brought us to roughly the right count. Parameter values we tested are given in the table below.
- Applied log compression on the counts
  We applied the following log compression transformation on our data set: log(count + 1). Because no changes were made to the original size of the training set through this method, we downsampled the same way we originally downsampled. Parameter values we tested are given in the table below.

	Parameters Test	Optimal Parameter	Test Results
Original	Rank: 10, 20, 30 Reg.: .01, .1, 1.0 Alpha: 0.5, 1, 10	Rank: 30 Reg.: 1.0 Alpha: 10 MAP: 0.029750	MAP: 0.034869 Precision: 0.007446 NDCG: 0.143481
Remove count <= 1	Rank: 10, 20, 30 Reg.: .01, .1, 1.0 Alpha: 0.5, 1, 10	Rank: 30 Reg.: 1.0 Alpha: 10 MAP: 0.031432	MAP: 0.037004 Precision: 0.008231 NDCG: 0.153341
Remove count <= 5	Rank: 10, 20, 30 Reg.: .01, .1, 1.0 Alpha: 0.5, 1, 10	Rank: 30 Reg.: 1.0 Alpha: 10 MAP: 0.03207	MAP: 0.037042 Precision: 0.008197 NDCG: 0.152759
Log10(count + 1)	Rank: 10, 20, 30 Reg.: .01, .1, 1.0 Alpha: 0.5, 1, 10	Rank: 30 Reg.: 1.0 Alpha: 10 MAP: 0.0237	MAP: 0.027862 Precision: 0.006697 NDCG: 0.124521

As can be seen from the table above, the model trained on data with removed counts (removed counts 1 and counts greater than or equal to 5) performed best, and we found slightly better MAP on our test set than our original model. It's likely that our results might differ if we were able to test on a wider range for Rank and Alpha.

Team Member Contributions:
Kelly - downsample data, test/validation metrics, extension
Sheetal - hyper-parameter tuning, extension

Code Appendix

- *downsample.py*: downsamples training data
- *recommender_train_rmse.py*: training model on and validating on rmse, prints all rmse values and print the best rank, regularization, alpha combination based on those tested
- *recommender_train_map.py*: training model on and validating on MAP, prints all MAP values and print the best rank, regularization, alpha combination based on those tested
- *best_model_train.py*: takes as input a rank, regularization, alpha - used to train model based on results from recommender_train
- *map_test_file.py*: takes as inputs the test/validation data, string indexer models for the user_id and track_id columns, and transforms the test set to return a dataframe of the format row([user, [list of relevant tracks for that user]) and saves it as a parquet file
- *recommender_test.py*: evaluates model on test data, giving MAP, NCGD, Precision
- Code used for extension was done directly through pyspark
  ds = spark.read.parquet('hdfs:/user/bm106/pub/project/cf_train.parquet')
  ds.createOrReplaceTempView('ds')
  removed_count1 = spark.sql('select * from ds where count > 1')
  removed_count1.write.parquet('downsample_removed_count1_6_del')
  removed_count5 = spark.sql('select * from ds where count > 5')
  removed_count5.write.parquet('downsample_removed_count5_6')
  ds_old = spark.read.parquet('hdfs:/user/kas1080/train_downsampled_all.parquet')
  ds_old.createOrReplaceTempView('ds_old')
  log = spark.sql('select user_id, log10(count + 1) as count, track_id, __index_level_0__ from ds_old')
  log.write.parquet('train_log_downsample_2')