# Recommender System for User Listening Data via Collaborative Filtering and Alternating Least Squares

Micaela Flores
NYU CDS
mrf444@nyu.edu

Ziv Schwartz
NYU CDS
zs1349@nyu.edu

May 18, 2019

## 1 Introduction

For this project, we are tasked with using Spark to build and evaluate a recommender system for user listening data. In general, recommendation systems can either employ explicit data or implicit data to make predictions. Explicit data is feedback voluntarily provided by users such as a star rating of a movie or a "like" on Facebook. Conversely, implicit data is drawn from user behavior such as clicks on an ad or types of items purchased online. The use of implicit data in recommendation systems is becoming increasingly popular and needed as explicit feedback from users is often not readily available in large quantities. Thus, this collaborating filtering approach with Alternating Least Squares makes predictions using data from users and the items with which they have interacted.

## 2 Data

The data contains three files split into training, validation, and testing data. Each file contains a table with columns *user_id, count, track_id*, and a superfluous *index* column. The training data, `cf_train.parquet`, contains full histories for around a million users, and partial histories for 110,000 users. The validation data set, `cf_validation.parquet`, contains the remainder of histories for 10,000 users, and the test data set, `cf_test.parquet`, contains the remaining history for 100,000 users.

## 3 Implementation

### 3.1 Baseline

For the baseline, we attempted to perform grid search to find the best set of the three hyperparameters. We chose a range of 5 values for each hyperparameter:

$$regParam \in \{0.001, 0.01, 0.1, 0.3, 0.5\}$$
$$alpha \in \{0.5, 1, 5, 10, 20\}$$
$$rank \in \{8, 10, 12, 15, 20\}$$

Due to the large size of the training file, memory errors, killed jobs, and general unpredictability of the DUMBO cluster, we were unable to execute a single successful job on the entire training file. We did, however, attempt to downsample the training data to 10% of the original size, but it still took overnight to train, and the resulting ALS model performed best with the default *regParam*, *alpha*, and *rank* parameters. This, coupled with the suggestion from the professor, prompted us to remove all hyperparameter tuning and focus on getting a single ALS model with the default parameters trained on 10% of the data.

We begin in `als_train.py` by reading in the parquet file and, with a Pipeline, convert both the *user_id* and *track_id* columns into integer indices using `StringIndexer`. Then, using the `ml.redcommendation` package, we create an ALS model with the default parameters, fit it to our transformed training data, and save the model in HFS. We likewise save the *user_id* and *track_id* `StringIndexer` objects to preserve the mapping when we transform the test data in `als_test.py`.

In `als_test.py`, we load in the trained ALS model and `StringIndexer` objects, and use them to transform the test data in the same way we did above. After this, we take the test data and use a group by and aggregation query to collect all the *track_ids* per user into a single list. Thus, each row in the resulting DataFrame contains all the tracks a particular user actually listened to. In a similar vein, we extract the same information from the trained ALS model, but instead use the method `.recommendForUserSubset(data, n)` which returns the top $n$ track predictions for each user in `data`. For this project, we perform evaluations on the top 500 track predictions for each user. We join the predicted recommendations with the actual track preferences and pass this information into `RankingMetrics`. Our evaluation metric of choice is Mean Average Precision, or MAP, which is the mean of the average precision of track recommendations for each user.

### 3.2 Extension: Alternative Model Formulations

Beyond the baseline implementation of the ALS model for recommendations, we decided to dig deeper into the particular model formulations. Given the original structure of the data, we are able to accomplish this by modifying the count data and observing how it affects our MAP estimate.

We conducted a thorough evaluation of the distribution of the count data and decided on three alternative formulations on which to train our ALS model. The first was to run a standard log compression on the entire count data column. The play counts span a very large range per user per track, from single listens all the way to 9,667 listens. Given the expansive range, log compression can be used to create more linear data and provide a better evaluation space for our recommender system.

The next alternative formulation considers how low user play counts affect recommendations and their corresponding MAP. We predict that listening to a song only once should have a very minimal, if any, effect on future recommendations. As such, we drop all counts of one from the training data. The entire training data consists of 49,824,519 rows of users, counts, and tracks. Of those, 29,595,102 instances have a count of one (about 60% of the full data), so we remove those values and train our ALS model on the remaining instances to see how the MAP value differs.

We continue exploring how low play counts affect recommendations by dropping all instances from the training data that contain two or fewer counts. For any one track, a play count of two suggests that a user may have been indifferent to a track after the first listen, but listened again an additional time. If after the second listen the user does not enjoy the song, it should not be considered as a positive interaction by our recommender system. Of the 49,824,519 total instances in our training data, 37,145,639 instances contain two or fewer counts (about 74.5% of the full data), so we drop those rows entirely. Therefore, in this alternative formulation, only the instances where users listened to tracks three times or more were used to train our ALS model.

Within the `ext_als.py` file we add a command line parameter, *model_formulation* (defaults to *None*), and the specific alternative formulation is generated depending on which parameter is passed in. For the log formulation, pass `log`; for the counts greater than one formulation, pass `ct1`; and for the counts greater than two formulation, pass `ct2`.

# 4 Evaluation Results

Due to time constraints, we trained ALS models on 10% of the original data set and on 10% of the data with all three of the aforementioned alternative implementations. The results are detailed below.

| Implementation | Validation MAP | Test MAP |
|---|---|---|
| Baseline | 0.01147 | 0.01164 |
| Log Compression | 0.01418 | 0.01404 |
| Drop $count \leq 1$ | 0.01080 | 0.01106 |
| Drop $count \leq 2$ | 0.00607 | 0.00472 |

Again, it should be noted that the relatively low MAP values of each variation are likely due the reduction of data and the lack of any hyperparameter tuning. In light of a suggestion from the professor, we test our trained ALS model on the validation set and test set separately since we do not perform hyperparameter tuning on the validation set. We see that the Log Compression alternative formulation slightly outperformed all other formulations on the validation set and test set. This is likely due to the fact that taking the log of highly skewed data (as we have since about 74.5% of our data has very low counts) will make it less skewed, and therefore, more interpretable by standard statistical methods.

# 5 Conclusion

In this project, we build and evaluate a recommender system on user listening data using the Alternating Least Squares (ALS) algorithm to train our models. Due to the lack of sufficient computational resources[1], our trained models and evaluation metrics reflect performance on a 10% sample of the full training data. We train a baseline ALS model and three alternative model formulations, (1) log compression, (2) drop $counts \leq 1$, and (3) drop $counts \leq 2$. Using `RankingMetrics` to yield the MAP of each our models, we evaluate on the top 500 track predictions for each user on the validation and test set. Out of the four ALS model formulations, the Log Compression alternative formulation slightly outperformed all other formulations on both data sets, with MAP values of 0.01418 and 0.01404, respectively.

# Contributions

The writing and testing of the code was done very collaboratively and evenly throughout the entire process. Micaela wrote the script for `als_train.py`, including the hyperparameter grid search using `RankingMetrics` (not used in the final code, as detailed in the paper). She also wrote the script for `als_test.py`. Ziv wrote the script for the extension, `ext_als.py`, including all prior research and thorough evaluations of alternate model formulations, and helped in debugging and

[1] please click here: `https://tenor.com/ShUg.gif`

optimizing the baseline and extension scripts for deployment. Both partners ran, verified, and tested the implementation extensively and contributed equally to the writing and editing the final paper.

## References


[1] Victor. "ALS Implicit Collaborative Filtering." Medium, 23 Aug. 2017, medium.com/radon-dev/als-implicit-collaborative-filtering-5ed653ba39fe.

[2] "Spark ML cookbook (Python)." I failed the Turing Test, 15 May 2017, vinta.ws/code/spark-ml-cookbook-pyspark.html