DSGA 1004 - BIG DATA
FINAL PROJECT REPORT

Shreyas Chandrakaladharan (sc6957)
Shradha Chhabra (sc7242)

1. Introduction

A recommendation system is a type of information filtering system that attempts to predict the preferences of a user, and make suggestions based on these preferences. These systems passively track different sorts of user behavior, such as purchase history, listening/watching habits and browsing activity, in order to model user preferences. The technology behind those systems is based on profiling *users* and *items*, and finding how to relate them.

2. Problem Setting

In the given problem, we want to build a recommender system that recommends new songs to users. We have been given *users*, *tracks*, and the number of times users have listened to tracks. We record this data by observing user behavior and this is a form of implicit feedback. Unlike the much more extensively researched explicit feedback, we do not have any direct input from the users regarding their preferences. Therefore, user behavior in this case is profiled by the number of tracks a user has listened to, and we can analyze this using **collaborative filtering (CF)**. Collaborative filtering analyzes relationships between users and interdependencies among items, in order to identify new user-item associations. The data is represented as a **utility matrix**, giving for each user-item pair, a value that represents what is known about the degree of preference of that user for that item. We assume that the matrix is sparse, meaning that most entries are "unknown." The goal of a recommendation system is to predict the blanks in the utility matrix. We are predicting count for each <user_id, track_id> pair.

3. Data

There are in total 7 data files that are given to us [cf_train.parquet, cf_validation.parquet, cf_test.parquet, metadata.parquet, features.parquet, tags.parquet, lyrics.parquet], out of which the first three contain training, validation, and testing data which we used for our baseline model. Specifically, each file contains a table of triplets (user_id, count, track_id) which measure implicit feedback derived from listening behavior. The four additional files consist of supplementary data for each track (item) in the dataset. Moreover, the data files are of column-based Parquet format, which could have several benefits over conventional row-based files:

1. Files stored in Parquet format can be split across multiple disks, which lends themselves to scalability and parallel processing.
2. By their very nature, column-oriented data stores are optimized for read-heavy analytical workloads, while row-based databases are best for write-heavy transactional workloads.

Files size:
cf_train contains ~50M records for ~1M distinct users
cf_validation contains ~136K records for 10K distinct users
cf_test contains ~1.3M records for 100K distinct users.

4. Baseline Recommender System

So far we have understood that the interactions between two classes of entities– users and items are represented using *Utility Matrix*. Users have preferences for certain items, and these preferences have to

be teased out of the data. One of the most popular algorithms to solve CF problems is Matrix Factorization (MF) and out of many MF algorithms we are going to use Alternating Least Square (ALS).

4.1 Indexing

- The user and item identifiers were encoded (strings) into numerical index representations for them to be consumed by the ALS model.
- We built two separate StringIndexers. One for transforming User ID into U_ID and another for transforming Track ID into T_ID. We do it step by step for clarity and lesser memory consumption. We built the user ID indexer using data from the training set since it contains all users IDs. We built the track ID indexer using data from the metadata file since it contains all track IDs.
- *pyspark.ml.feature.StringIndexer* is used to create the indexers. Indexers were then saved (./sc2_final_u_indexer has the user indexer and ./sc2_final_t_indexer has the track indexer) after fitting them to be reused. We then used *StringIndexerModel* function to load these saved model in our program.
- Finally we stored the results to new files in HDFS in order to save the overhead costs of transforming the data each time the code is run.

```
data = spark.read.parquet(data_file)
data = data.sample(False,0.001)
u_model = StringIndexerModel.load(u_idx_model_file)
t_model = StringIndexerModel.load(t_idx_model_file)

transformed_data = u_model.transform(data)
transformed_data = t_model.transform(transformed_data).select('u_id','t_id','count')
```

Code Snippet: String Indexing

4.2 Downsampling the data

Scale of the dataset was a major engineering blocker in this project. Training with the entire training data was both computationally expensive, ridiculously slow and majorly redundant. Thus, in order to more rapidly prototype the model, we downsampled the data. Downsampling is based on the distinct users from validation and test sets. We discard the rest of the data.

4.3 Modelling

4.3.1 ALS model

Alternating Least Square (ALS) is a matrix factorization algorithm which runs in a parallel fashion. ALS is implemented in Apache Spark ML and built for a large-scale collaborative filtering problems. ALS does a pretty good job at solving scalability and sparseness of the Ratings data, and it's simple and scales well to very large datasets. The implementation in our recommendation system has the following parameters:

> *maxIter* has been set to 5, this is the default number of iterations to run.

> *implicitPrefs* flag has been set to 'True' in order to use the ALS variant adapted for implicit feedback data

> *alpha* is a parameter applicable to the implicit feedback variant of ALS that governs the baseline confidence in preference observations. This parameter was tuned in our training.

> *regParam* specifies the regularization parameter in ALS. This parameter was tuned in our training.

> *rank* is the number of latent factors in the model. This parameter was tuned in our training.

> we set *coldStartStrategy* to 'drop' to ensure we don't get NaN evaluation metrics

> Data Columns: *userCol* = "u_id", *itemCol* = "t_id", *ratingCol* = "count"

Then we saved the model files.

```
model_file = './models/als_{0}_{1}_{2}.model'.format(l,a,r)
print('Working on {0}...'.format(model_file))
als = ALS(maxIter=5, implicitPrefs = True, alpha = a,regParam=l, rank=r,userCol="u_id",
    itemCol="t_id",ratingCol="count",coldStartStrategy="drop")
model = als.fit(data)
model.save(model_file)
```

Code Snippet: Fitting ALS Model

4.3.2 Training

Hyper Parameters tuning: We tried 27 combinations, that is, three values for each of the 3 hyperparameters. We chose the range for each parameter based on intuition from *Collaborative Filtering for Implicit Feedback Datasets by Hu. et al.* From the paper, the best parameters that worked were alpha = 4- and regStrength = 120. Performance increased as rank increases but after rank=40 negligible increases were observed. We expected our problem to have best performance around the same parameters. While this is not true and we need a wider search to empirically optimize performance due to the scale of the problem, we restricted to the following values: Alpha = [0,20,40], Regularization Strength = [0,50,100], Rank = [5,13,21]

For rank, additionally we considered the number of features in features.parquet file. There were 13 features for each song. This gives a rough idea of how many dimensions the song embedding might have. This was our intuition behind picking 13. We manually tested out each parameter combination in an iterative manner. We did not cross validate as there was already lots of data to train. Cross validation proves most useful only when there is very less data to train on. Moreover we had an explicit validation set.

4.3.3 Evaluation

Evaluation of implicit-feedback recommender requires appropriate measures. In the traditional setting where a user is specifying a numeric score, there are clear metrics such as mean squared error or root mean square error to measure success in prediction. However with implicit models we have to take into account availability of the item, competition for the item with other items, and repeat feedback. Thus, **precision based metrics, such as RMSE, MSE MAE, are not very appropriate**, as they are based on the difference between the rating that we predicted for an item and the rating that a user gave that item. They require knowing which items users dislike for them to make sense. For this reason, we are reporting both Precision at 500 (P500) and Mean Average Precision (MAP).

RankingMetrics: This takes as an argument an RDD of *predicted rankings* and *ground truth rankings*. Ground truth in this context is held-out interactions, contained in cf_validation/cf_test.

recommendForUserSubset: We used this method of the ALSModel for getting the top 500 recommended items. A point to note is that it does not list top 500 *new* items, that is, it may recommend items that are already known positives. But in the given problem, since each user has, on average, a small set of positive interactions, this won't contribute much to the overall score.

We transformed the recommended items into predicted rankings in the format required to get ranking metrics on it. To transform the data into the required format for RankingMetrics, we first used explode() and collect_set() functions to do the transformation required to build the prediction set for each user. However, this ended up destroying the order of predictions (set of recommended items for each user) i.e. they were not ranked by count (this work is shown in V1). So, V1 worked for calculating P500 which does not required ordered predictions. V2 however is simpler and just uses a direct map transformation to build the predictions. This does not destroy the order and worked for Mean Average Precision which required ordered prediction.

```
model = ALSModel.load(model_file)
userSubsetRecs = model.recommendForUserSubset(testusers, 500)
joined_table = labels.join(userSubsetRecs,labels.u_id==userSubsetRecs.u_id)
reqdPredsLabels = joined_table.rdd.map(lambda l: ([t.t_id for t in l.recommendations],l.ranked_labels))
metrics = RankingMetrics(reqdPredsLabels)
```

Code Snippet: Evaluate ALS Model _v2

4.4 Results

Our best MAP of 0.028 and best P500 of 0.007 observed from two different combinations of 3 parameters. Best MAP combination was (alpha=20,regParam=0,rank=21) and Best P500 combination was (alpha=40, regParam=0, rank=21). We then tested both models on test data and results are printed in actual_metrics_v2.out

5. Extension

As part of our extension, we choose to do the first extension option which was to try out different transformations of the implicit feedback value to see if it improves performance. We tried two methods: log transformation and dropping low count values. Due to resource scarcity, we trained the extension models only for one hyperparameter configuration which was our best combination from the baseline model, (alpha = 20, regParam = 0, rank = 21).

Log Transform: When we apply log transform to the count, we convert the counts from a large scale to a small scale. We squeeze all the information into a smaller window, so in a sense, this brings the data closer and more points are near the decision boundary. This way we hope the model will be able to define the decision boundary better because it has more points near it. Observations: The approach failed and did not improve results. It worsened the results significantly. This gives us the intuition that our data has to be separated further than brought closer. So we tried dropping low counts.

Filters (dropping low counts): The intuition behind this is that we are dropping values close to the decision boundary. This is the reverse intuition of log-transform. We have only large counts, which are far away from the decision boundary, so the classifier should be able to make the correct decision easily. Observations: This did not work as we expected. This might be due to the fact that when we drop low counts (drop values with counts <=1) we lose about half the data. We do not have sufficient data to train. Results: **(alpha = 20, regParam = 0, rank = 21)**

	Precision at 500	Mean Average Precision
Log Transform	0.0005504999999999999	0.000299870818464979
Filters	0.005694955620486247	0.01707338204552185

Team Members	Contributions
Shreyas Chandrakaladharan	Write-ups for: Problem Understanding, Data Understanding, Implementations (Indexing, Downsampling, Modelling, Evaluate, Results) Coding: String Indexing, Modelling, Evaluation
Shradha Chhabra	Write-ups for: Implementations (Indexing, Downsampling, Modelling, Evaluate, Results), Extension implementation and results Coding: EvaluationTest, Extension