Million Songs Dataset – Recommender System

Building a music recommender system using the million-song dataset

Amey Ramakant Mhadgut, arm994@nyu.edu

MS CS Student, Courant Institute of Mathematical Sciences - New York University

1 INTRODUCTION

With a wide variety of multimedia options available to users, it has become important to suggest items to users that they likely may like. As a part of the project, we are providing top 500 music suggestions for each user that the user may like. For this, the baseline recommendation system that we are building is a latent factor model using Collaborative filtering which is implemented using Spark's Alternating Least Square (ALS) module. We will look at dataset first, then talk about preprocessing and then talk about modeling. Lastly, we will look at the results.

2 BASELINE MODEL

The next subsections we will look at the dataset, data preprocessing steps and building and testing the model and results.

2.1. ABOUT THE DATASET

The dataset consists of one million users with their count for tracks listens.
On Peel's HDFS, we have the count data in the format (user_id, count, track_id) spilt into train, validation, and test dataset as below:

- cf_train.parquet: To train our model
- cf_validation.parquet: To tune hyperparameters for our model
- cf_test.parquet: To test the final model

We should note that the dataset consists of listen count for each user for each track. This is implicit feedback in terms of recommender system. An example of explicit feedback includes song ratings or song liked/disliked which directly contributes to if a user liked/disliked the song.
To explore the dataset more, I created a data_viewer.py and data_explorer.py which were used to look at the dataset and run aggregation queries to understand the dataset.

Dataset	No. of Unique Users
Train set	1129318
Validation set	10000
Test set	100000

2.2. PREPROCESSING THE DATA

The major requirement for ALS is to have the user column and item column in integer form (numeric form). The dataset that we have currently has the user_id and track_id column in alphanumeric format. So, we preprocess the dataset to index these columns into numeric using StringIndexer from PySpark.ml.feature module.

```
# Alphanumeric  to numeric conversion
userid_indexer = StringIndexer(inputCol='user_id', outputCol='user_id_num', handleInvalid = 'skip')
userid_indexer_model = userid_indexer.fit(dataframe)
trackid_indexer = StringIndexer(inputCol='track_id', outputCol='track_id_num', handleInvalid='skip')
trackid_indexer_model = trackid_indexer.fit(dataframe)
dataframe = userid_indexer_model.transform(dataframe)
dataframe = trackid_indexer_model.transform(dataframe)
```

I also down sampled the training dataset to 25% of the users (~250k) and then trained and tuned the model. This was done because the cluster was heavily loaded with jobs and this exercise was enough to get me a range of hyperparameter values to explore.

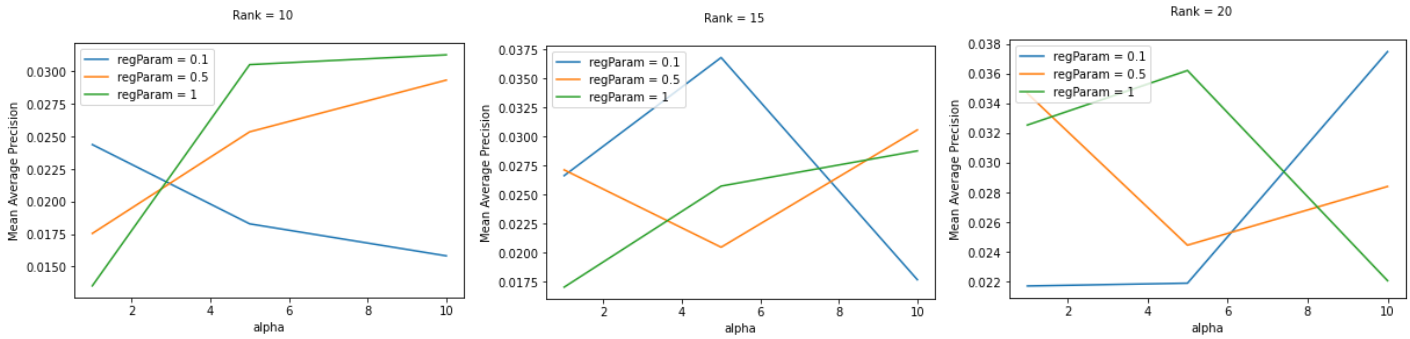
2.3. ABOUT THE MODEL

For the baseline model, I used the Alternating Least Square (ALS) module from PySpark's ML module. The model has two variant – Implicit & Explicit. We have set the parameters as below -

Parameter	Value	Comment
implicitPrefs	True	Since our dataset has implicit data as highlighted in Section 2.1
nonnegative	True	Since our count column ranges from 1 to 9667 and is positive
userCol	user_id_num	Numeric indexed user_id
itemCol	track_id_num	Numeric indexed track_id
ratingCol	count	Listen count for songs
coldStartStrategy	drop	Drop any rows in the DataFrame of predictions that contain NaN values
maxIter	10	Iterate for 10 count (Tried more but cluster fails sometimes for more data)
rank*	**TBD**	**To be tuned**
regParam*	**TBD**	**To be tuned**
Alpha*	**TBD**	**To be tuned**

I tuned the hyperparameters for various values which are marked with * above using the validation dataset. I considered Mean Average Precision and Precision at 500 as the metric to evaluate the model's ranking performance. Initially, I also tried another model approach to predict the count data and evaluate using RegressionEvaluator wherein I used Root Mean Squared Error as the metric. But later I read the research paper[1]

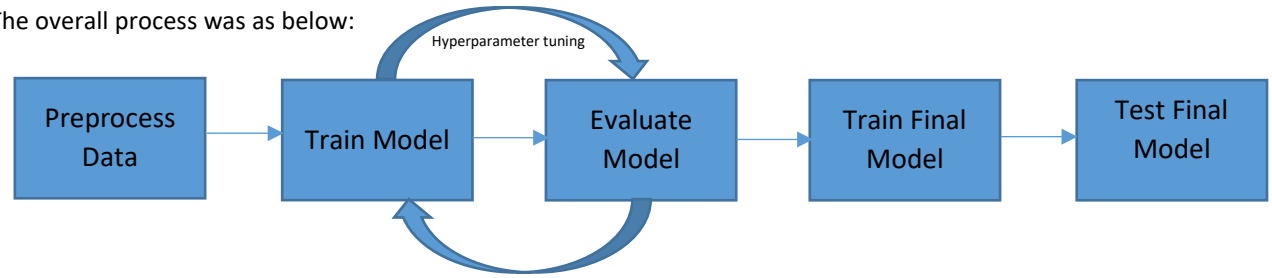
Observed the below results which I plotted using Matplotlib:



I observed the best result in terms of mAP against the validation dataset for rank: 20, regParam: 0.1, alpha: 10.
For more details on the results, refer the Hyperparameter Tuning Plots.ipynb notebook.
Using the above hyper parameters, I trained the final model and evaluated the model against the test dataset, got the results as:
Mean Average Precision is 0.03466062211007827, Precision at k = 500 is 0.007259999999999998
I could measure the evaluation only on 5000 users from the test set because of cluster limited and being occupied.

2.4. SUMMARY

The overall process was as below:



Below are some key lessons:
- Start with subset of users (understanding sampling correctly)
- For tuning, grid search can be tricky with large data and limited resources
- Ranking evaluation can be time consuming and needs resources. RankingEvaluator is available in Spark 3 but the cluster available to us had Spark 2.4

3 FAST SEARCH USING ANNOY [EXTENSION]

I am an active Spotify user and curious about their recommendation engine. So, I choose this extension to explore the area. Another reason to choose is I have done a similar exercise in Computer Graphics wherein I used Binary Spatial Partitioning tree to improve the search performance for collision detection. So, I wanted to explore something similar here.

3.1. PROBLEM STATEMENT

Our goal is to identify k nearest neighbors for a given vector from a search space of vectors. Using some sort of similarity (or distance like dot, Manhattan, cosine, etc.) to find similarity between our query vector and each vector in the search space and get the k vectors which have highest similarity (least distance).

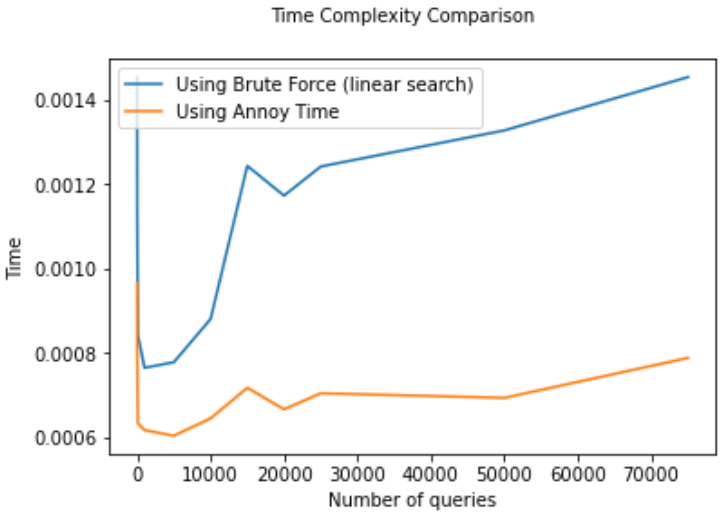
To approach this problem, say we have N vectors in our search space, we can go about in two ways –
1. Using linear search: Calculate distance per vector in the N vectors from the given query vector, sort and return the top k results. This take going through N vectors and the time complexity is O(N).
2. Using trees: Create a tree for the search space once. Search in the tree with time complexity of O(log N)

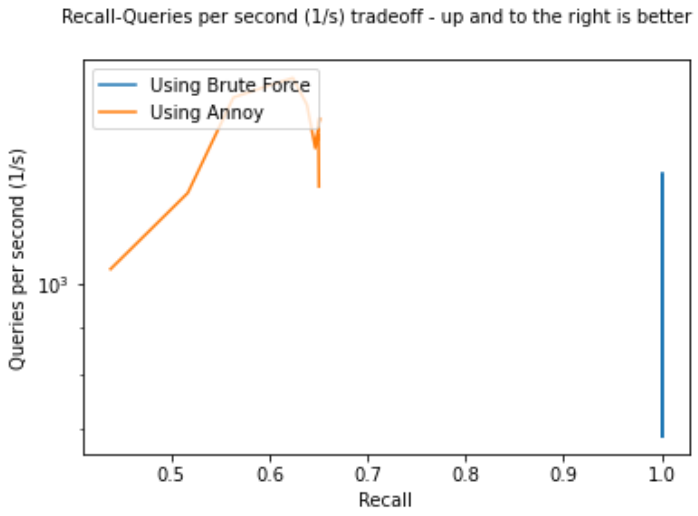
Spotify's Approximate Nearest Neighbor Oh Yeah![2] library to search for user queries in the search space helps to achieve the Point 2 above and has other benefits.

3.2. BENCHMARKING THE ALGORITHM

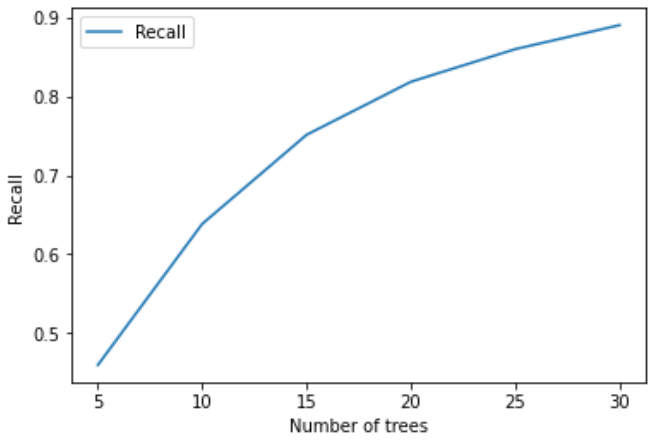
I tried to explore and validate annoy's performance. To do that, I used linear search as the baseline. I used the metric as dot product to calculate distance between vectors. I also tried the angular metric but that took time for the brute force approach.

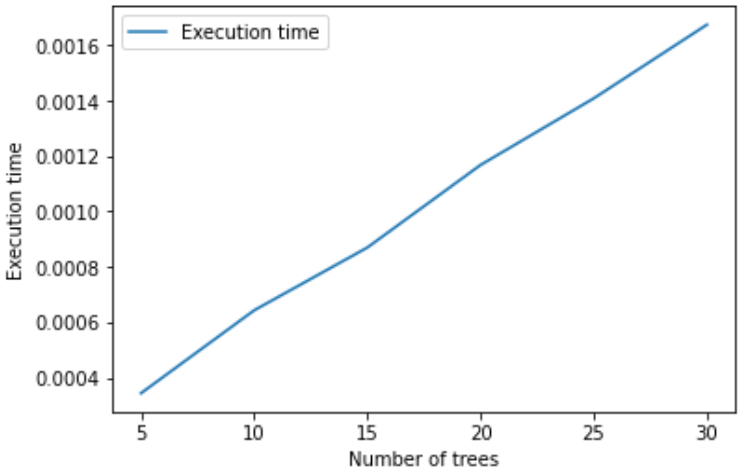
I recorded the below factors:
1. Time complexity: Compare time complexities for both the linear search and annoy across number of queries and the results are below. Annoy, being tree based, performed certainly well.
2. Recall-Queries per second (1/s): I found this benchmarking idea from ann-benchmark[3] which is also displayed on annoy's readme. I calculated the recall and across queries per sec. Recall was calculated by considering the brute force results as ground truth. Got the results as below. I tried this with number of trees as 10. The recall was sufficiently good for the performance. With a greater number of trees, the recall may increase.





3. Number of trees: We can increase the number of trees in annoy to get better recall. I measured recall and time complexity across a range of number of trees and got the below result. There is a tradeoff between better results and performance as observed.





3.3. BENFITS OF ANNOY

As stated in the readme, the benefits in addition to performance include –
1. Ability to use static files as indexes
2. Low Memory usage
3. Memory can be shared across multiple processes

4 RUN INSTRUCTION AND CODE ARTIFACTS

Below are the code artifacts and their use –
1. Baseline Model
   - project_starter.py: Pipeline to train, validate and test model
2. Fast Search Extension
   - Fast Search.ipynb: Notebook for fast search using annoy
   - utilities.py: Contains utilities used in the Fast Search notebook
   - user_factors.csv and values.csv: CSV files with vectors
3. Initialize
   - shell_setup.sh: Shell file to initialize HDFS and Spark commands
4. Hyperparameter tuning plots
   - Hyperparameter Tuning Plots.ipynb: Notebook to generate plots
5. Others
   - data_explorer.py, data_sampler.py, data_viewer.py: Code to explore dataset
   - project_starter_explicit_model.py and output.txt: Explicit model pipeline and its output (which was discarded)

To run,
1. Initialize HDFS and Spark by running: *source shell_setup.sh*
2. To run the baseline model, run: *spark-submit --driver-memory=8g --executor-memory=8g --executor-cores=25 project_starter.py*
3. To run the fast search, run the blocks in the Notebook

5 CITATIONS/REFERENCES

[1] The Million Song Dataset Challenge - https://brianmcfee.net/papers/msdchallenge.pdf
[2] Credits - https://github.com/spotify/annoy
[3] Credits - https://github.com/erikbern/ann-benchmarks