Introduction

Over the years, datasets have increased in size and some have become too large to fit in memory on a single machine. Various methods for parallelizing and/or distributing computation across multiple machines have been proposed to deal with this challenge (Dean & Ghemawat, 2008; Rocklin, 2015). One such solution is Spark, a cluster computing framework that supports parallel operations and the analysis of data that is split across multiple machines while maintaining scalability and fault tolerance (Zaharia et al., 2010). Spark relies on a resilient distributed dataset (RDD) abstraction, which is a read-only collection of objects partitioned across multiple machines that has the capability of being rebuilt if a partition is lost (Zaharia et al., 2010). RDDs are also well-suited for a variety of applications.

For this project, we built a song recommender system based on the Million Song Dataset (MSD) using PySpark, a Python language implementation of Spark. The MSD is an open-access database that comprises metadata, audio features, and user listening statistics for 1 million songs (Bertin-Mahieux et al., 2011). The user-item information is provided as (user, song, count) tuples, while the item-level data contains additional features, including tags, similar songs, lyrics, acoustic data, genres, and more. We built a recommender system using the alternating least squares (ALS) algorithm, a collaborative filtering technique that learns latent factors and predicts missing entries in a user-item (in this case song) association matrix. We tested this implementation against a baseline popularity model and a single-machine ALS implementation with lenskit, a tool for conducting experiments on recommender algorithms and datasets (Ekstrand et al., 2011).

Data

The datasets for this project were pulled from the MSD and provided as three files in Parquet format: cf_train.parquet (training data), cf_validation.parquet (validation data) and cf_test.parquet (test data). Each file consists of tuples of the form (user_id, count, track_id), and indicates how many times a user has listened to a track.

Evaluation metrics

We evaluated our models via the following metrics: 1) length of time to perform computations, and 2) mean average precision (MAP). MAP is defined as:

$$\mathrm{MAP} = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{|Q_i|}\sum_{q\in Q_i}\mathrm{Precision}(R_{iq})$$

where n denotes the number of users in the dataset, Q_i is the set of indices of positive recommendations (i.e. recommended tracks that are also in the ground truth set) for user i, and R_{iq} is the set of ranked recommended tracks from the top to q^{th} result for user i. In this sort of application we care about more than just the first positive interaction (i.e. reciprocal rank); we prioritize having many positive interactions ranked well. Hence MAP was the most suitable evaluation metric.

Exploratory data analysis

Basic characteristics of the train and validation datasets (cf_train and cf_validation, respectively) were explored prior to building the model. **Table 1** shows the total number of users and unique tracks included in the train and validation sets. In-depth analysis of the train set revealed that all users listened to at least five songs, but there were 31,253 songs that had only one listener, which represents ~8% of the total number of songs in the dataset. Given the high percentage, we expected that if used to train

our model, the model would choose some of these unpopular songs as future recommendations. Since we were primarily interested in validating our model against the observed data in the validation set, tracks with only one listener in the train set were removed to improve the likelihood of recommending tracks that were actually observed in the validation set.

Table 1: Exploratory data analysis for train and validation datasets.

	Train	Validation
Number of users (user_id)	1,129,318	10,000
Number of unique tracks (track_id)	385,371	49,994

Hyperparameter tuning

Once we built a working version of the model, we tuned the hyperparameters using the validation set. Detailed results of our hyperparameter tuning are shown in **Table 2**.

Table 2: Hyperparameter tuning. A) Tuning the rank and regularization parameters. The default value of alpha=1 was used for these tests. B) Tuning the alpha parameter with ranks of 100 and 200.

A

Rank	Regularization	MAP
10	0.001	0.03302
	0.01	0.03307
	0.1	0.03325
	1	0.02344
	10	3.88E-5
50	0.001	0.04510
	0.01	0.04493
	0.1	0.04662
	1	0.03941
100	0.01	0.05010
	0.1	0.05255
	1	0.04610
200	0.001	0.05472
	0.01	0.05445
	0.1	0.05804
	1	0.05280
400	0.01	0.05742
	0.1	0.06204
	1	0.05933

B

Rank	Regularization	Alpha	MAP
100	0.1	0.001	0.05438
		0.01	0.05449
		0.1	0.05151
		1	0.05255
		5	0.05989
		10	0.06319
		20	0.06574
		50	0.06815
		100	0.06857
200	0.1	0.001	0.06221
		0.01	0.06191
		0.1	0.05756
		1	0.05804
		5	0.06619
		10	0.07004
		20	0.07369
		50	0.07742
		100	0.07873

Since *rank* and *regularization* are two of the most important parameters for the ALS model, we tested a wide range of values for these parameters. *Rank* was tested at 10, 50, 100, 200, and 400. Although a rank of 400 resulted in the highest MAP, the improvement was not significant enough to justify the dramatic increase in time to train the model. Thus, we chose a *rank* of 200 for the final model. The *regularization* parameter was initially tested at 0.001, 0.01, 0.1, 1, and 10 with *rank* set to 10. In this case, the best MAP results were seen at values of 0.01, 0.1, and 1, so these values were used for subsequent tuning.

Once the optimal *rank* and *regularization* parameters were chosen, the *alpha* parameter, which represents the confidence in the weights for the song counts, was also tuned on a wide range of values: 0.001, 0.01, 0.1, 1, 5, 10, 20, 50, and 100. We found that MAP scores were best at the extremes of this range, with the highest scores corresponding to the highest *alpha* parameter values. We chose *alpha* to be 50 for our final model as *alpha* of 100 resulted in only a minimal improvement in model performance.

Evaluation of model on test set

The final values obtained through hyperparameter tuning were *rank*=200, *regularization*=0.1, and *alpha*=50. In order to successfully fit and evaluate the model on the Peel high performance cluster (HPC), the number of user and item blocks were increased to 100 and the checkpoint interval was disabled. The MAP value for the PySpark model trained on the test set was 0.07752.

Extension 1: Baseline popularity model

We implemented the baseline popularity on each of the three datasets to recommend the 500 most popular songs to each user. To accomplish this, each unique track was sorted by its frequency in the dataset and indexed to a nonnegative integer corresponding to its rank, such that the top 500 most popular tracks were indexed 0 - 499. These tracks were recommended to each user in the dataset and the MAP values computed. The MAP values on the train, validation, and test datasets were 0.02830, 0.01995, and 0.02075 respectively.

Extension 2: Lenskit single machine model

We implemented the lenskit single-machine ALS model on the Greene cluster and compared it to PySpark's parallel ALS model. Unlike PySpark, the lenskit implementation of ALS does not have an *alpha* hyperparameter. We therefore limited our hyperparameter tuning to the *number of features* and the *regularization* parameter. Since we had already performed a full grid search in PySpark on the equivalent model, we did not do so on the single-machine implementation. We focused instead on exploring how the *rank* and *fraction of the data used* affected our runtime, with MAP as a secondary priority. The runtimes (fitting times) on the cluster were obtained by averaging three runs and calculating the standard deviation, as cluster times are highly dependent on the total volume of other jobs. The results are shown in **Table 3**. Interestingly, the lenskit model had a MAP of 0.0349 for the test dataset, which was significantly lower than the MAP of 0.07752 using the PySpark model. However, the lenskit model also had significantly lower fitting times.

Table 3: Exploring lenskit runtimes in comparison to PySpark. All trials used 10 iterations and a regularization parameter of 0.1. The PySpark model runtimes were calculated using the most optimal hyperparameters.

Rank	Fraction	Fitting Time (s)	Validation MAP	Spark Fitting Time (s)
200	0.5	1815	0.0145	3978 ± 131
	0.625	2084	0.0190	
	0.75	2392	0.0243	4695 ± 580
	0.875	2695	0.0320	
	1.0	3378	0.0357	5576 ± 690
150	1.0	2360	0.0331	
100	1.0	1493	0.0303	
50	1.0	989	0.0257	

Discussion

Overall, the personalized recommender system built in PySpark outperformed the other two in terms of predictions. With the most optimal hyperparameters (*rank*=200, *regularization*=0.1, and *alpha*=50), the MAP for the PySpark ALS model was 0.07752 for the test set. In comparison, the baseline popularity model MAP for the test set was 0.02075, indicating that personalizing the recommendations adds a lot of value to the model. Using equivalent hyperparameters for the lenskit single-machine ALS model, the MAP was 0.0349 for the test set. However, the fitting times were surprisingly better than those obtained when running the PySpark ALS model, perhaps due to the high volume of jobs performed on the cluster, as implied by the high standard deviation.

Conclusions

In this report, we have presented three recommender systems of songs based on the Million Song Dataset. To evaluate each of the models, we have compared their fitting times (when appropriate) as well as their MAP for making predictions. The first recommender system is based on the ALS algorithm, a collaborative filtering technique, and built in PySpark, which allows for parallel computation on a cluster. The second recommender system is a baseline popularity model, which recommends a set of the most popular songs to each user. The third is a single-machine ALS model that was implemented using the lenskit tool. The first two recommender systems were run on the Peel cluster, while the third one was run on the Greene cluster. Overall, the PySpark ALS model outperformed the other two models based on MAP, though it did have longer fitting times, which may be partially explained by the volume of jobs that were concurrently running on the cluster.

Code and contributions

Github repository: https://github.com/nyu-big-data/final-project-team_rex/tree/main

Emily Mui: Exploratory data analysis, Spark hyperparameter tuning, attempted Spark model evaluation
Adhham Zaatri: Lenskit eval + tuning code, preliminary Spark code (basic data loading, model fitting)
Yechan Lew: Spark model evaluation code, baseline model extension
Julia Manasson: Lenskit fitting code, attempted Spark model evaluation, many report improvements
Everyone: Contributed to discussions of project strategy, conceptual design of the algorithms, brainstorming and troubleshooting, and writing of the report.

References

Bertin-Mahieux, T., Ellis, D. P. W., Whitman, B., & Lamere, P. (2011). The million song dataset. *Proceedings of the 12th International Society for Music Information Retrieval Conference, ISMIR 2011*.

Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, *51*(1). https://doi.org/10.1145/1327452.1327492

Ekstrand, M. D., Ludwig, M., Konstan, J. A., & Riedl, J. T. (2011). Rethinking the recommender research ecosystem: Reproducibility, openness, and LensKit. *RecSys'11 - Proceedings of the 5th ACM Conference on Recommender Systems*. https://doi.org/10.1145/2043932.2043958

Rocklin, M. (2015). Dask: Parallel Computation with Blocked Algorithms and Task Scheduling. *Proceedings of the 14th Python in Science Conference*. https://doi.org/10.25080/majora-7b98e3ed-013

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster computing with working sets. *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2010*.