Movie Recommendation System

Big Data Final Project

GitHub Repository: https://github.com/nyu-big-data/final-project-group43

Kristin Mullaney
Center for Data Science
New York University
New York, NY, USA
kmm9492@nyu.edu

INTRODUCTION

Modern movie recommendation systems use advanced algorithms to suggest films based on a user's viewing history and preferences, making it easier for viewers to find their next favorite films.

This project delved into the efficacy of several of these algorithms by creating various movie recommendation systems using the MovieLens dataset collected by F. Maxwell Harper and Joseph A. Konstan [1]. Two versions of the dataset were used: a small sample (ml-latest-small, 9000 movies and 600 users) and a larger sample (ml-latest, 58000 movies and 280000 users) to generate three different models.

The first model was a popularity model that solely used PySpark functionality [2]. It was designed to identify the 100 most popular movies and assign them to all users. The second and third models were both latent factor models: one implemented solely with PySpark and the other utilizing the LightFM package [3]. These models went beyond the traditional method of recommending the same 100 movies to all users and instead, generated user and movie vector representations to make personalized recommendations.

In this project, four commonly used ranking metrics - precision at k, mean average precision at k, NDCG at k, and mean average precision - were employed to evaluate the performance of the model's recommendations. Additionally, a custom metric was developed to further assess a model's overall performance, which incorporated all four other metrics. The results indicated that the PySpark latent factor model performed slightly better than the popularity model in all four ranking metrics for the small dataset. However, it underperformed in all four ranking metrics for the large dataset. Furthermore, the LightFM implementation was found to have higher precision and speed compared to the PySpark version for both the small and large datasets.

DATA

Two samples from the MovieLens dataset were utilized - the compact 'ml-latest-small' containing 9000 films and 600 users, and the extensive 'ml-latest' featuring 58000 movies and 280000 users. Both samples included user ratings and tags, with the larger sample also providing additional 'tag genome' information for each film. The primary focus was on the movie ratings data, which includes a user's identification, movie identification, a rating out of 5, and a timestamp for each rating.

TRAIN-TEST-SPLIT

In order to train the models effectively, it was necessary to divide the ratings dataset into three distinct subsets: training, validation, and testing data. To ensure that each of these subsets included ratings from every user, the ratings data was first imported into a PySpark dataframe and grouped by individual user. Utilizing PySpark's built-in sample function, each viewer's group of ratings was then randomly allocated to the training, validation, and testing subsets, with roughly 70% of each user's ratings being designated for training and the remaining 30% being divided equally between validation and testing. It should be noted that timestamps were not taken into account during this process. The rationale behind including all users in all three subsets was to ensure that the model was capable of accurately learning information about each user, rather than making blind recommendations. This process is demonstrated in Figure 1 below.

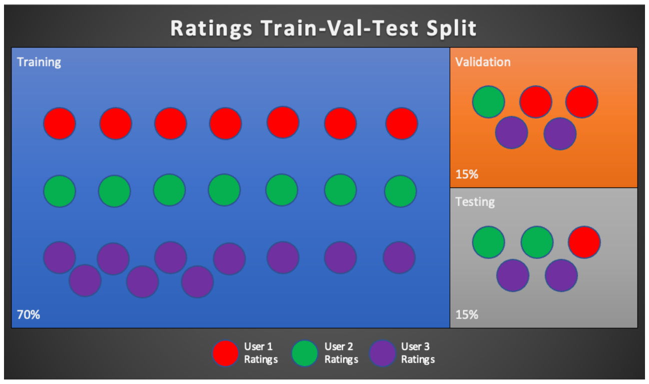

Figure 1: Example of how several users ratings would be split between the training, validation, and test sets.

MODEL SKELETON

A parent class named "Model" was created to serve as the foundation for the subsequent popularity and latent factor model classes. This was done to exploit the similarities across different model types and to establish a consistent and organized structure for future modeling endeavors. The parent "Model" class accepts train, validation, and test datasets and includes three methods: “Build”, “Recommend”, and “Score”. The “Build” and “Recommend” methods in the parent "Model" class are left unimplemented, as each child class of model requires its own specific functionality for generating and applying movie recommendations. The "Build" method for each "Model" object is responsible for establishing the underlying structure by which the model generates its movie recommendations. Subsequently, the "Recommend" method applies this built model to a given dataset. Finally, the "Score" method evaluates the accuracy of the model's recommendations by comparing them to the true ratings provided by viewers. This parent class allows for easy and consistent evaluations of the accuracy of movie recommendations made by the models.

TUNING FUNCTION

Using a parent "Model" class has a major advantage in that the "Score" method can be applied to different types of models. This allows for a single tuning function to optimize hyper-parameter values for various models. The tuning function inputs hyper-parameter names and possible values to test. It can only adjust one hyper-parameter at a time and builds models with each value to determine the best performance. To account for multiple metrics (precision at k, NDCG at k, MAP at k, and precision), a system calculates an overall performance score by normalizing results for each hyper-parameter against the best performing value for that metric. These values are then summed and divided by four to generate a final score, with a score of 1 indicating the highest performance in all metrics. The best performing hyper-parameter value is set as the model's new default and the process repeats for the next hyper-parameter to be tuned.

POPULARITY MODEL

The popularity model serves as a benchmark for evaluating our latent factor implementation. This model simply recommends the 100 most popular movies to every user without considering personal preferences. The only hyper-parameter that was fine-tuned in this model was the minimum allowed average rating for a movie. This was implemented to account for variations in what could be considered popular. For example, ranking popularity by highest number of views would favor higher budget films that received more exposure, even if the films received low ratings from audiences. On the other hand, ranking popularity by average ratings alone would favor small budget films that were only rated a handful of times and may not be well-received by broader audiences.

In order to ensure that the recommended movies would be widely seen and enjoyed, it was determined that a popular film would be one that received the highest number of ratings, provided that it met a minimum average rating threshold. Positive ratings were considered to be those above 3, and ratings were transformed into binary values of 1 for movies that were reviewed positively or 0 for movies that were either not seen or received a rating below 3. Through the use of the tuning function, it was discovered that the optimal minimum average rating cutoff was 3.46 for the small dataset and 3.38 for the larger dataset. This led to the recommendation of films such as Forrest Gump and Shawshank Redemption, which have gained widespread acclaim and are highly regarded by audiences. The final metrics for the validation and test data for both datasets are illustrated in the accompanying figure.

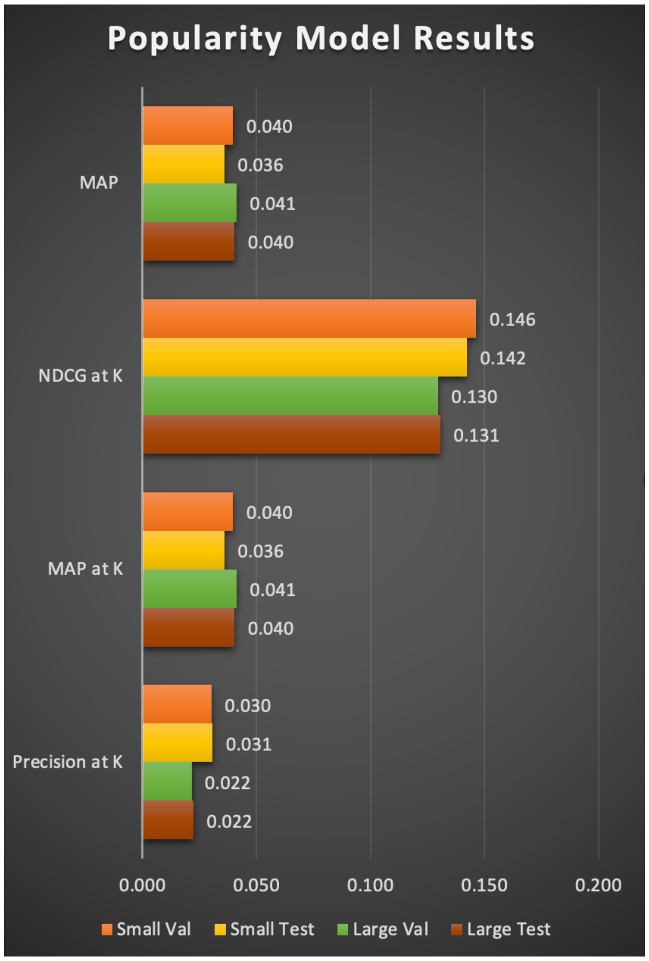

Figure 2: Results of the popularity model on the validation and test sets for both the small and large datasets.

It's worth noting that the metrics used in this project were affected by the fact that a user who viewed a movie and didn't enjoy it was treated identically to a user who never saw the movie at all. This skews the results towards false negative reviews. Additionally, since all of the ratings used in this project were historical and no real recommendations were made, it's not possible to differentiate between non-viewers who consciously decided not to watch a movie and those who had never been recommended the film in any way.

In terms of performance, the Mean Average Precision metrics performed better with the larger dataset, while precision and NDCG performed better with the smaller dataset. The reason for this discrepancy is unclear, but it's worth noting that NDCG (at around .13 and .146 respectively) is much higher than the other metrics (between .021 and .041 across all fields). The fact that the NDCG metric takes the order of recommendations into account, implies that the increase in NDCG score can be indicative of the model's improved ability to correctly rank the recommendations, particularly for the most sought-after movies. The model's proficiency in identifying the most popular movies could be attributed to the significant overlap in the films that people watch, as well as the tendency for humans to act as one another's recommendation systems, resulting in a disproportionate level of exposure for blockbuster films among the general public.

LATENT FACTOR MODEL

The primary objective of the latent factor model is to generate personalized recommendations by harnessing embedding representations of users and movies. Ideally, this model should outperform the popularity model.

To accomplish this, PySpark's alternating least squares algorithm was employed to construct the latent factor model. Three of the algorithms built-in hyper-parameters were used, namely rank, number of iterations, and regularization hyper-parameter. The rank hyper-parameter determines the length of the embedding vectors, and in theory, longer vectors can capture more intricate details about an object. The number of iterations hyper-parameter sets the number of times the alternating least squares algorithm should run during training, and the regularization hyper-parameter helps the model generalize to new data.

An additional hyper-parameter, minimum ratings count, was added to check if movies with too few ratings were affecting the model's performance. To start the tuning process, default hyper-parameter values were randomly assigned, and the tuning function was used to adjust each one, one at a time. Once the best value for a hyper-parameter was found, it was set as the new default value, and the next hyper-parameter was tuned. The process of tuning each of the four hyper-parameters in sequence is referred to as a complete tuning iteration for the remainder of this report.

To avoid running an excessive number of hyper-parameter variations on an already large model, several techniques were implemented to reinforce that the located values were likely optimal. Firstly, the model continually tuned the hyper-parameters until two complete iterations were in total agreement about the optimal values. Secondly, to account for the possibility that the previous and current iterations might be inclined to agree (as the results of the first iterations were used as the starting hyper-parameter settings for the second iteration), the entire tuning process was run again with different default values and with a different hyper-parameter tuning order. If both rounds were in agreement about the optimal values, the tuning process was complete, and final metrics on the validation and test sets were collected.

For the initial tuning, several values were tried for each of the hyper-parameters. For rank, values of 5, 10, 50, 100, and 200 were tried. For the number of iterations, values of 1, 5, 10, 15, and 20 were tried. For the regularization term, values of 0.15, 0.20, 0.25, 0.30, and 0.35 were tried. In addition, values of 60, 70, 80, 90, and 100 were tried for the minimum number of ratings allowed for a movie to be considered for recommendations.

The default settings for the first tuning section were as follows: number of iterations set to 5, rank set to 1, minimum rating count set to 60, and regularization term set to 0.15. The order in which the hyper-parameters were tuned was: number of iterations, rank, minimum rating count, and then regularization term.

The default settings for the second tuning section were as follows: number of iterations set to 20, rank set to 200, minimum rating count set to 100, and regularization term set to 0.35.

The results of the tuning are presented below. It is worth noting that the small dataset's hyper-parameters all converged towards the center of the available range, whereas the regularization term for the large dataset initially settled at 0.15, the minimum value tested. Thus, another round of tuning was conducted on the large dataset, utilizing a wider range of values for the regularization term (0.01, 0.05, 0.10, 0.15, and 0.20). The results below are from the final round of tuning.

	Small	Large
Rank	100	100
Number of Iterations	5	5
Regularization Term	0.2	0.1
Minimum Ratings Count	70	100

Figure 3: Hyper-parameter tuning results on the small and large datasets. Only the validation sets were used during tuning.

It is intriguing that both models seemed to achieve the highest performance with a rank of 100 and a mere 5 iterations. The small dataset achieved its best performance with a minimum rating count of 70 views, while the larger dataset performed optimally with a slightly higher threshold of 100 views. This suggests that movies with too few ratings may not possess enough generalizability to the broader population to be beneficial for the model. The reason for the optimal performance with only 5 iterations remains uncertain, however, it could be attributed to the potential for overfitting with an excessive number of iterations. Furthermore, the discrepancy in the optimal regularization term for the small and large datasets is noteworthy. The optimal regularization term drops from 0.2 to 0.1 for the larger dataset. This could be an outcome of the training datasets for the larger group being a more accurate representation of the overall population, and thus requiring less regularization to generalize effectively. The metrics for models constructed using these hyper-parameter settings with the small and large datasets are illustrated in the figure below.

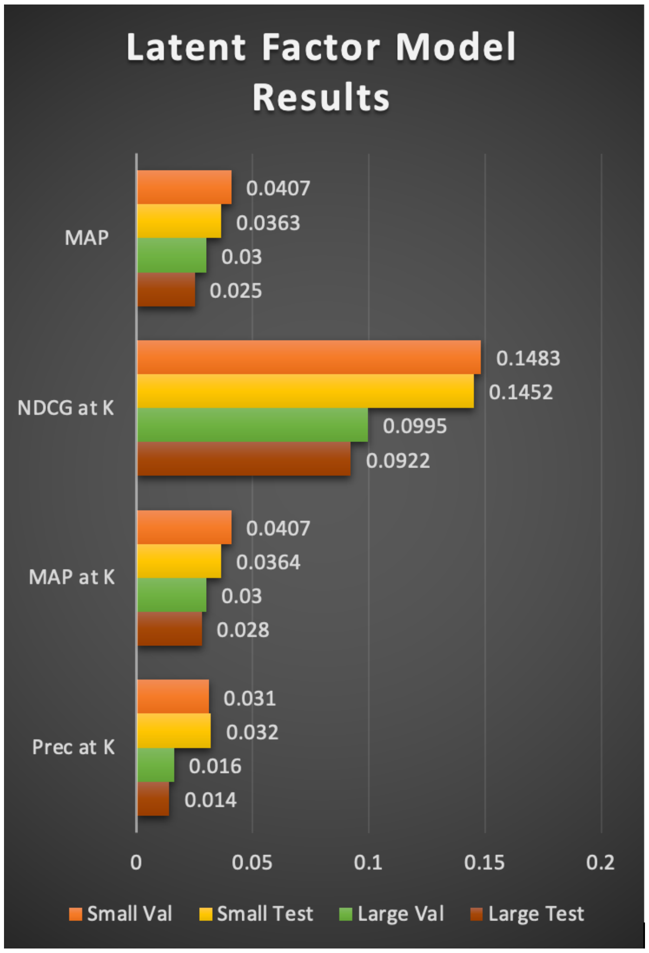

Figure 4: Results of the latent factor model on the validation and test sets for both the small and large datasets.

The results indicated that while the latent factor model exhibited a slight advantage over the popularity model for both validation and test sets across all four metrics for the small dataset, it performed inferiorly to the popularity model on every metric for the large dataset. The reason for this discrepancy remains uncertain, but it could partially be attributed to the model's inability to differentiate between movies that were not well received and those that had not yet been recommended. This could have led the model to detect signals that were not truly present and incorporate them into the latent factor representations. It is possible that this effect became more pronounced with the larger dataset, which had a greater number of movies to recommend and was therefore sparser. The effect of the error might have inadvertently been reduced by simply recommending Forrest Gump to everyone, as the popularity model did, rather than attempting more elaborate methods.

MODEL EXTENSION: LIGHTFM IMPLEMENTATION

To gain a comprehensive understanding of both the benefits and drawbacks of PySpark's implementation of a latent factor recommender system, an alternate collaborative filtering recommendation system was implemented using the LightFM package for Python. PySpark's latent factor implementation uses the Alternating Least Squares (ALS) algorithm and is optimized for distributed computing, making it well-suited for handling large datasets and scaling to billions of interactions. On the other hand, LightFM utilizes the Stochastic Gradient Descent (SGD) algorithm on a single machine, making it more efficient and faster, but less suitable for handling large datasets. The goal was to see how model speed and performance compared between the two implementations.

However, due to the different built-in hyper-parameters in each package, a direct comparison was difficult. Thus, the chosen approach was to optimize the hyper-parameters for each model and then compare speed and accuracy across models. Since the available metrics for each implementation also varied, only precision at k was used to judge model performance.

To optimize the hyper-parameters of the LightFM latent factor model, a similar method to the tuner function previously used was implemented, where each hyper-parameter was tuned in sequence. The chosen hyper-parameters to tune were the learning rate, loss function, number of components and the user regularization term. The optimal values were found to be 0.01 for the learning rate, logistic or WARP for the loss function, 50 for the number of components and 0.1 for the user regularization term. The results of the LightFM model on the test set, which were obtained after implementing the optimized hyper-parameters, are presented below in comparison to the corresponding values from PySpark's implementation. It's worth noting that eight nodes were available for PySpark's distributed computation.

Small Dataset		
	PySpark	LightFM
Time to Fit (in Seconds)	2.8	0.8
Precision	0.32	0.035

Figure 4: Results of the speed and precision of both latent factor models on the test set. Results are an average of three runs.

Large Dataset		
	PySpark	LightFM
Time to Fit (in Seconds)	153	4
Precision	0.015	0.002

Figure 5: Results of the speed and precision of both latent factor models on the test set. Results are an average of three runs.

The results clearly indicate that LightFM's collaborative filtering algorithm performed faster and with higher precision than PySpark's in this case. It is worth mentioning that the difference in speed and precision was less pronounced for the small datasets. Though LightFM was expected to perform better for the smaller dataset, it is not immediately obvious why it vastly outperformed the PySpark implementation on the large dataset. One possibility is that shuffling data during PySpark's distributed computation may have slowed the model down, although it seems unlikely that this would be the sole cause of the delay. Another possibility is that the stochastic gradient descent method is simply a faster and more efficient training method than alternating least squares, and LightFM is only limited by the capabilities of a single machine. It is possible that as long as a single machine can process the data, it will perform faster than alternating least squares with distributed computation.

CONCLUSION

In conclusion, it was found that each of the three models had their unique strengths and weaknesses. The popularity model, which recommended films similar to "Forrest Gump", performed almost as well as the PySpark latent factor model on the small dataset and even outperformed it on the large dataset. On the other hand, the LightFM stochastic gradient descent implementation managed to achieve comparable precision to PySpark's alternating least squares method in a much shorter timeframe. While all three models showed promise, LightFM really stood out. It could be said that LightFM is the "Forrest Gump" of movie recommendation systems, it’s generally well-liked and it can run really fast.

REFERENCES

[1]
F Maxwell Harper and Joseph A Konstan. 2015. The movielens datasets: History and context. Acm transactions on interactive intelligent systems (tiis) 5, 4 (2015), 1–19.

[2]
Maciej Kula. 2015. Metadata Embeddings for User and Item Cold-start Recommendations. In Proceedings of the 2nd Workshop on New Trends on Content-Based Recommender Systems co-located with 9th ACM Conference on Recommender Systems (RecSys 2015), Vienna, Austria, September 16-20, 2015. (CEUR Workshop Proceedings), CEUR-WS.org, 14–21. Retrieved from http://ceur-ws.org/Vol-1448/paper4.pdf

[3]
Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. Communications of the ACM 59, 11 (2016), 56–65. DOI:https://doi.org/10.1145/2934664