Scalable Techniques for Personalized Movie Recommendations

MARTIN KEENAN, New York University, USA

1 INTRODUCTION

This report documents the development of machine learning models for recommending movies based on a user's partial viewing history.[1] In this section we introduce the MovieLens datasets, discuss their division into training, validation and test sets, and explain our choice of evaluation metric. Section 2 describes the construction of a simple baseline popularity model. Section 3 details the training of a latent factor model that offers personalized recommendations. Section 4 proposes an alternative formulation of the test set for better modeling real-world user interactions. Finally, Section 5 explores a method for reducing inference time using ScaNN, a state-of-the-art similarity search method developed by Google Research.

1.1 MovieLens Dataset

The MovieLens datasets, collected by the GroupLens research lab at the University of Minnesota, contain movie ratings provided by users from the mid-1990s to 2018. Each observation contains a user ID, a movie ID, a timestamp, and a rating between 0.5 and 5.0 stars. The first dataset contains ratings on 9,000 unique movies provided by 600 users, who each contributed at least twenty and an average of 160 ratings. The second dataset contains ratings on 58,000 movies provided by 280,000 users, who each contributed at least one and an average of 100 ratings [3]. Hereafter, the datasets are referred to as the "small dataset" and "large dataset."

1.2 Train-Test Split

Before training, each dataset is divided into training observations and held-out observations for measuring the models' ability to generalize to new data. First, users in each dataset are segmented into groups of training users, validation users, and test users, using a 60/20/20 random split. Based on the timestamp, the most recent thirty percent of each validation and test user's observations become the validation set, used for hyperparameter tuning, and test set, used to measure model performance. This hold-out method most closely simulates real-world circumstances, where a model makes predictions about a user's future interactions based solely on their history. The remaining observations from the validation and test users are combined with those from the training users to become the training set.

1.3 Evaluation Criteria

The latent factor model discussed in Section 3 is trained to minimize the square distance between observed training ratings and predictions. This makes root mean square error (RMSE) a natural choice of evaluation metric because it measures how close our predictions are to a user's actual ratings.

However the purpose of a recommendation system is not just to predict ratings, but to provide a set of items the user will enjoy. We also consider two ranking metrics which are used to evaluate a set of recommendations: mean average precision at K (MAP) and normalized discounted cumulative gain at K (NDCG). The MAP is calculated as the fraction of the first K recommended items that are identified as positive interactions. Here, a positive interaction is equivalent to the hold-out set containing a record of the user watching the movie. NDCG compares an ordered list of

[1]The codes accompanying this report are provided at https://github.com/nyu-big-data/final-project-group_29.

recommendations to the held-out set and gives more weight to higher ranked items. NDCG is calculated by first finding the discounted cumulative gain and the ideal discounted cumulative gain,

$$\text{DCG} = \sum_{i=1}^{K} \frac{\mathbb{1}(\text{movie}_i)}{\log_2(i+1)} \qquad \text{and} \qquad \text{IDCG} = \sum_{i=1}^{|H|} \frac{1}{\log_2(1+i)},$$

where $\mathbb{1}(\text{movie}_i)$ is an indicator for whether movie_i is in the user's held-out observations and $|H|$ is the user's number of held-out observations. NDCG is then calculated as the ratio of DCG over IDCG. For evaluating the baseline and latent factor models, we set K equal to 100 for both ranking metrics.

Both MAP and NDCG evaluate the real-world interaction between users and movies more realistically than RMSE. However, they are still somewhat limited because they treat all observations in the held-out set equally as positive interactions. NDCG can be modified to give higher weight to items with higher observed explicit ratings [9], however the `pyspark.ml.evaluation` module only supports the implicit implementation of NDCG [6].

To simulate positive interactions between users and movies, we also evaluate our models on alternative validation and test sets where we exclude all ratings below four stars. By preserving only observations with high ratings, the held-out data is a better representation of movies that users enjoyed.

2 BASELINE MODEL

The baseline model makes non-personalized recommendations by taking the average rating for each movie in the training set and recommending movies in order of their average rating. One downside of this method is that there are many movies with few observed ratings, which diminishes confidence that the average in the training data is reflective of quality. To reduce the influence of low frequency movies, we include β artificial observations with zero-star ratings for each movie. Sparsely observed movies will therefore have scores closer to zero. For each of the small and large datasets, we evaluate the model on the validation set with different values of the hyperparameter β. The results are shown in Table 1, where the best result for each metric is indicated by an asterisk. The bottom row shows the test performance using the value of β with the best validation performance for the given metric. Note that RMSE is not calculated with a penalty term because the penalty only serves to improve ordered recommendations, not the accuracy of ratings predictions.

Table 1: Validation and Test Metrics for Baseline Model

	Small Dataset			Large Dataset		
β	**MAP**	**NDCG**	**RMSE**	**MAP**	**NDCG**	**RMSE**
0	0.000	0.000	1.006*	0.000	0.000	0.968*
5	0.029	0.120	-	0.009	0.046	-
10	0.030*	0.124*	-	0.010	0.051	-
100	0.028	0.123	-	0.019	0.072	-
1,000	0.027	0.120	-	0.024	0.092	-
10,000	0.027	0.119	-	0.028	0.103	-
100,000	0.027	0.119	-	0.028	0.105	-
1,000,000	0.027	0.119	-	0.029*	0.106*	-
Test Results:	**0.023**	**0.108**	**0.968**	**0.029**	**0.107**	**0.962**

Interestingly, the best validation results on the large dataset occur when using a very high value of β. With such a high penalty, the movies' rankings are more dependent on how many times they have been watched and less on their

average rating. This suggests that the best baseline predictor of whether a user has watched a movie in the hold-out set is whether many other people have watched it and not necessarily whether it is a good movie.

3 LATENT FACTOR MODEL

The latent factor model provides more flexibility than the baseline model by allowing us to make personalized predictions about users' ratings. This model assumes there is some number of unobserved attributes representing each movie, which we call latent factors. We refer to the number of latent factors in our model as the "rank." The users and movies's affinities for each factor are modeled by a real number, and the inner product between user and movie vectors measures the strength of a user's affinity for a particular movie.

The model is trained using the Alternating Least Squares (ALS) algorithm implemented in the `pyspark.ml` module [1]. ALS minimizes the training RMSE by repeatedly updating the user and item factors. First, the algorithm computes the user factors as the coefficients of linear least squares regression on the item factors and observed training ratings. Then it updates the item factors in the same way by fixing the user factors, and repeats this process until convergence. The model also takes a regularization parameter, λ, which is used in each iteration of least squares regression.

Table 2 displays the results of the latent factor model on the validation set for several ranks between 10 and 200 and values of λ between 0.001 and 0.1. The best result for each metric is indicated by an asterisk. The bottom row shows the test performance using the hyperparameters with the best validation performance for the given metric.

Table 2: Validation and Test Metrics for Latent Factor Model

		Small Dataset			Large Dataset		
Rank	**λ**	**MAP**	**NDCG**	**RMSE**	**MAP**	**NDCG**	**RMSE**
10	0.001	0.001	0.012	2.052	0.000	0.000	0.925
10	0.005	0.001	0.012	1.363	0.000	0.000	0.880
10	0.01	0.001	0.013	1.207	0.000	0.000	0.859
10	0.05	0.002	0.021	0.987	0.000	0.001	0.822
10	0.1	0.002	0.015	0.927	0.000	0.000	0.830
75	0.001	0.015	0.084	1.704	0.000	0.002	1.186
75	0.005	0.017	0.090	1.341	0.002	0.014	0.984
75	0.01	0.017	0.089	1.181	0.004	0.023	0.915
75	0.05	0.010	0.072	0.959	0.005	0.030	0.808
75	0.1	0.005	0.043	0.923	0.000	0.001	0.827
200	0.001	0.019*	0.097*	1.709	0.003	0.017	1.169
200	0.005	0.018	0.095	1.321	0.007	0.042	0.978
200	0.01	0.017	0.093	1.167	0.008*	0.048*	0.916
200	0.05	0.011	0.074	0.958	0.007	0.041	0.807*
200	0.1	0.005	0.042	0.923*	0.000	0.001	0.827
Test Results:		**0.018**	**0.087**	**0.890**	**0.008**	**0.046**	**0.808**

For both MAP and NDCG, the baseline model performs better than the latent factor model and the difference is more significant on the larger dataset. For example, the baseline NDCG metrics on the small and large datasets are 0.108 and 0.107, and the latent factor metrics are 0.087 and 0.046. This could be due to the fact that the latent factor does not explicitly model the number of times a movie has been watched, whereas the baseline model does so through its β parameter.

However, we do see a significant decrease in RMSE, from 0.968 and 0.962 with the baseline model to 0.890 and 0.808 with the latent factor model. This shows that the latent factor model is indeed creating personalized recommendations as it is able to more accurately predict each user's ratings. While this may not be the best measure of whether a recommendation is good, it significantly improves upon an area where the baseline model was lacking.

4 ALTERNATIVE TEST SET

As discussed in Section 1.3, we also evaluate the baseline and latent factor models using alternative validation and test sets, which exclude any observations with a rating less than four stars. This limits observed positive interactions to those where a user watched *and enjoyed* a movie. Table 3 shows each model's performance using the hyperparameters associated with the best result for the given ranking metric on the validation set. On the small dataset, the latent factor model now has a slightly higher MAP. However, the latent factor model still lags behind the baseline in the other dataset-metric combinations.

Table 3: Ranking Metrics Using Held-Out Observations with Four or More Stars

	Small Dataset		Large Dataset	
	MAP	**NDCG**	**MAP**	**NDCG**
Baseline	0.024	0.109	0.033	0.110
Latent Factor	0.024	0.089	0.010	0.054

Further approaches to improve modeling the interaction between users and movies may include a broader hyperparameter search and an implementation of ranking metrics that give higher weight to items with high ratings, rather than the implicit implementation in the `pyspark.ml.evaluation` module.

5 IMPROVING INFERENCE TIME WITH SCANN

One of the biggest challenges when building an efficient recommendation system is responding quickly to user queries. Inefficiencies at inference time can be aggravating for users and also have compounding effects if the system is not built to handle many queries at a time.

ScaNN (Scalable Nearest Neighbors) is a method developed by Google Research for efficient similarity search in large, high-dimensionality datasets. It is currently one of the most efficient open source approximate nearest neighbor search algorithms, as measured by ANN Benchmarks [2].

ScaNN operates in three main phases, of which the first and third are optional [8]. The first phase recursively partitions the data into a tree so that the tree's leaves contain similar vectors. The partitioning includes hyperparameters for the number of leaves in the tree and the number of leaves to search at query time. The second phase uses a novel methodology called anisotropic quantization to compute an approximate similarity score between the query vector and all the data points in the relevant partitions [4]. The third phase computes the similarity between the query and the K' datapoints with the highest scores by brute force and returns the datapoints with the K highest scores. In our implementation of ScaNN, we include both optional phases and set K' equal to 1,000 and K equal to 100.

Figure 1 shows the relationship between inference speed and recall when using ScaNN on the parameters from a latent factor model with rank 200 trained on the small dataset, which contains 9,000 movies. We generate the top 100 movie recommendations for 500 users and compute recall as the mean fraction of movies that are in the top 100 recommendations from a brute force implementation using `numpy`. The points on each line represent varying numbers

of leaves in the partitioning phase. The different lines represent different proportions of leaves searched during query time.

Figure 1: ScaNN Efficiency with Varying Number of Leaves on the Small Dataset

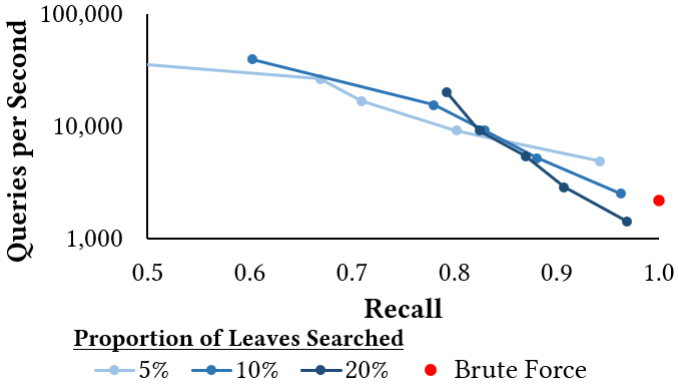
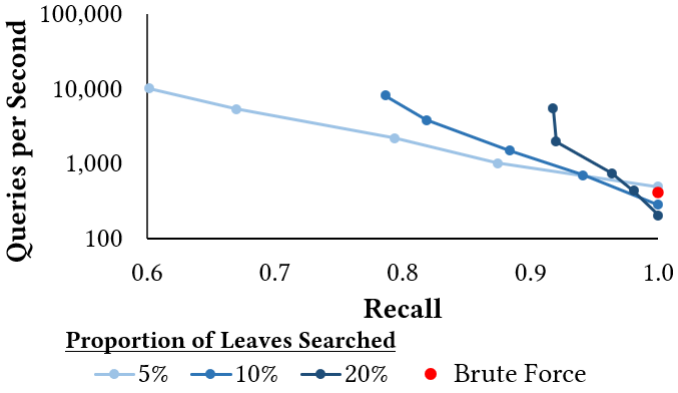


Figure 2 shows the efficiency gains of ScaNN on the parameters from a model with rank 200 trained on the large dataset, which contains 58,000 movies. Both datasets show a similar pattern in terms of efficiency gains with each of ScaNN's hyperparameters. With a low number of partitions, the increase in recall from searching a greater proportion of leaves is more significant than the cost to speed. However when there is a large number of partitions, it is better to search a smaller proportion of leaves to achieve significantly higher speed for a nearly identical recall. This is likely because it is easier to find high quality nearest neighbor candidates when the tree is partitioned at a finer-grained level.

Figure 2: ScaNN Efficiency with Varying Number of Leaves on the Large Dataset



At recall close to one, the ScaNN method achieves a similar speed to the brute force method. It is worth noting that there are efficiencies not seen here that may make ScaNN faster than brute force at scale. First, ScaNN can make use of a GPU to parallelize its search. Second, with higher rank models, greater numbers of movies, and more concurrent queries, the brute force implementation may run into memory limits and computational constraints. ScaNN mitigates these issues through its efficient partition tree structure and quantization.

REFERENCES

[1] "Alternating Least Squares (ALS) matrix factorization," available at https://spark.apache.org/docs/3.0.1/api/python/pyspark.ml.htmlmodule-pyspark.ml.recommendation.

[2] ANN Benchmarks, "Benchmarking Results," available at http://ann-benchmarks.com/.

[3] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4: 19:1–19:19. https://doi.org/10.1145/2827872

[4] Guo, Ruiqi, et al. "Accelerating large-scale inference with anisotropic vector quantization." International Conference on Machine Learning. PMLR, 2020.

[5] Koren, Yehuda, Robert Bell, and Chris Volinsky. "Matrix factorization techniques for recommender systems." Computer 42.8 (2009): 30-37.

[6] "MLlib - Evaluation Metrics," available at https://spark.apache.org/docs/1.5.0/mllib-evaluation-metrics.html.

[7] Rajaraman, Anand, and Jeffrey David Ullman. Mining of massive datasets. Cambridge University Press, 2011.

[8] "ScaNN Algorithms and Configurations," available at https://github.com/google-research/google-research/blob/master/scann/docs/algorithms.md.

[9] Wang, Yining, et al. "A theoretical analysis of NDCG ranking measures." Proceedings of the 26th annual conference on learning theory (COLT 2013). Vol. 8. 2013.