A Multi-Effect Model for the Netflix Challenge

Gautham Gorti
ggorti3@stanford.edu
006280603

Brendon Gu
bkgu@stanford.edu
006659577

Abstract

Matrix factorization and neighborhood models have both achieved excellent results on collaborative filtering problems. As these methods model different types of effects, combining them allows them to complement each other and has been shown to yield improved performance in practice. In this project, we explore these two approaches to collaborative filtering and design and implement a computationally efficient model for the Netflix challenge in which latent factor and neighborhood information are considered simultaneously to make predictions. We compare the performance of this combined model with those of baseline implementations of matrix factorization models and neighborhood models, along with published baselines for the Netflix challenge.

1 Introduction

The Netflix challenge was a Netflix-sponsored competition where the task was predicting user ratings for movies on a scale of 1 to 5 stars. This task is an example application of collaborative filtering, in which past user behavior is used to implicitly learn relationships between users and between items to then identify new user-item associations.

The importance and prominence of recommender systems has only increased since then, as companies have recognized the importance of personalized recommendations as part of the overall user experience. The widespread nature of these systems encompasses video platforms such as YouTube and Netflix, social media platforms such as Facebook, Twitter, and Instagram, and e-commerce platforms such as Amazon, among many others.

In this project, we explore two popular approaches to collaborative filtering: latent factor models [7] and neighborhood models [6]. For the Netflix challenge, latent factor models characterize users and movies by transforming both into the same latent space, allowing predictions for a user's rating for a given movie to be generated by an inner product of the corresponding user and movie latent representations. Neighborhood models compute relationships between users or between movies to then estimate a user's rating for a given movie based on either ratings for similar users on that movie, or ratings for similar movies by that user.

Previous work [3] has suggested that latent factor and neighborhood approaches address different levels of structure in the data. Therefore, our final system will merge the models to improve the performance of the system. Our project scope is twofold: first, we design a collaborative filtering system that employs the two approaches for collaborative filtering described above to model both regional and local effects on top of global effects, and second, we make our model computationally efficient and implement it in a way that allows us to learn on the entire dataset, rather than on a subset of the data.

The specific methods we investigate are alternating least squares matrix factorization and a neighborhood model with interpolation weights. We are able to combine these two models to achieve a significant performance improvement over Netflix's original CineMatch recommendation system.

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

In the remainder of this paper, we start with a description of the problem and related work in Section 2. We provide a detailed description of our methodology, as well as of the Netflix challenge dataset and the model implementation in Section 3. Finally, we list and discuss our experimental results in Section 4, and provide concluding thoughts and areas for future work in Section 5.

2 Problem Formulation and Related Work

The problem setup is as follows:

Let $R = \{r_{ij}\}$ be a user-movie rating matrix, with entry r_{ij} representing the rating of movie j by user i (entries may be missing). The (i, j) pairs for which r_{ij} is known are stored in the index set $\mathcal{I} = \{(i, j) \mid r_{ij} \text{ is known}\}$. Usually the vast majority of ratings are unknown.

Let $\mathcal{D}, \mathcal{T}$ be a train, test partition of $\mathcal{I}$ so that $\mathcal{D} \cup \mathcal{T} = \mathcal{I}$ and $\mathcal{D} \cap \mathcal{T} = \emptyset$. We wish to train a model using $\mathcal{D}$ to make predictions $\hat{r}_{ij}$ that minimize the root mean squared error (RMSE) over the test set:

$$\text{RMSE} := \sqrt{\frac{1}{|\mathcal{T}|} \sum_{(i,j) \in \mathcal{T}} (r_{ij} - \hat{r}_{ij})^2}$$

Neighborhood Models Neighborhood methods generate predictions for unknown ratings by taking a weighted average of the neighborhood of the most similar known ratings, given some similarity metric. For a user-item pair (i, j), we can either use user-user similarity – which will create the neighborhood using the users i_2 most similar to user i who have also rated item j – or we can use item-item similarity – which will create the neighborhood using the items j_2 most similar to item j which were also rated by user j. In this problem, there are far more users than items, so item-item similarity is more typically used [6], [1]. This is a very intuitive approach: "Because you liked movie j_1, you may also like the similar movie j_2."

Central to most item-item approaches is a similarity measure between items. Note that computation of all item-item similarities is an $\mathcal{O}(n_u n_m s)$ operation, where n_u is number of users, n_m is number of items, and s is the cardinality of the largest set among sets of movies rated by any single user. Given our dataset and computing resources, this computation is very expensive, and we cannot implement a true neighborhood model. We will discuss a workaround in our methods section.

Items with many ratings have well-defined similarity scores, so predictions involving these items are generally strong. Also, users that have rated many items will provide a large support from which to select neighbors, which also yields strong predictions. However, the neighborhood model can fail when trying to predict ratings for less popular items, because those items' corresponding similarity scores may rely on only a few other ratings and hence have high variance. The model can also fail when making predictions involving users who have rated few items, as the available neighborhood will consist of few ratings, again resulting in variance issues. In general, neighborhood models struggle when density in the training set around a desired user-item pair is low.

Latent Factor Models Latent factor models, assign for each user and each item a k-dimensional vector in the latent space $\mathbb{R}^k$ – the prediction for a given user-item pair is given by the inner product of the respective latent factors.

We can imagine that each dimension of the latent space learns to describe some important underlying attribute of each item. In the case of movies, perhaps the first dimension measures the amount of comedy in the movie, the second dimension measures the amount of action in the movie, and so on (in reality, the latent dimensions will not have such straightforward interpretations, this is just an illustration). Then, the corresponding latent vector for each user describes how much that particular user likes or dislikes movies along each given trait.

As opposed to neighborhood models, latent factor models provide a low-rank representation of users and movies which imposes some structure on the data space. When an appropriate rank and L2 regularization strength is selected, these models are good at correctly predicting many ratings – not limited to ratings involving highly represented users and movies as in the case of the neighborhood model [2][7]. Again, literature [3] seems to indicate that latent factor models and neighborhood models succeed in predicting different types of ratings and fail in predicting different types of ratings – and from that, ensembling both methods together can yield strong results.

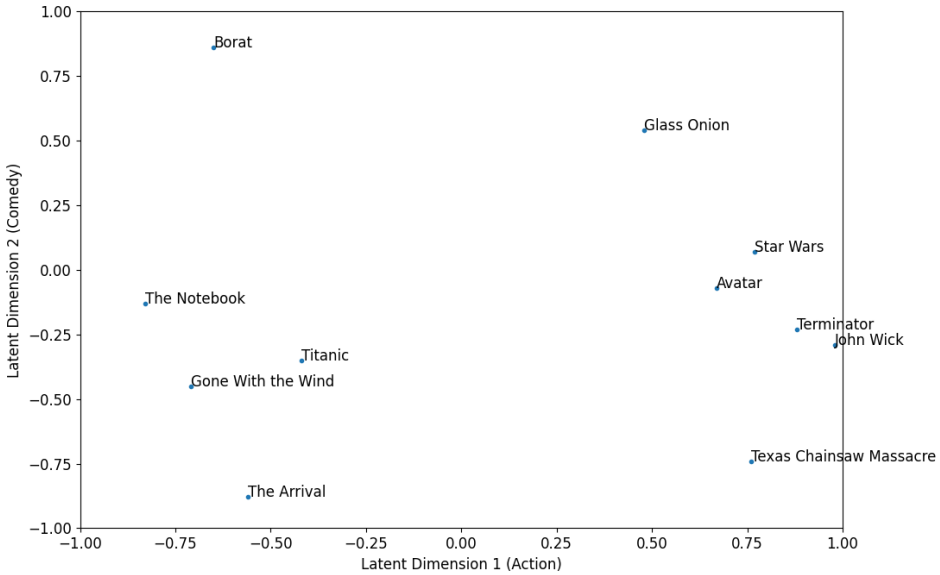

Figure 1: Toy example of possible movie latent factors in 2-dimensional latent space.

3 Methods

3.1 Dataset

We use the movie ratings dataset provided by Netflix for the Netflix challenge. The dataset consists of 100,480,507 {user, movie, rating, date} tuples, generated from $n_m = 17,770$ distinct movies and $n_u = 480,189$ distinct users. In the provided dataset, these tuples are stored across four separate .txt files. Netflix also provided a "probe" dataset of around 1.4 million examples – a subset of the training set that roughly matches the distribution of the final unseen test set. It is important to note that the distribution of the probe set is different (and more challenging) than the distribution of a naive uniform random sampling of the training set. The probe set contains many more ratings by users who do not rate much and are harder to predict. In a way, this represents a real-world requirement of collaborative filtering systems, which need to predict well on all users, not just heavy raters. As is convention for the Netflix challenge, we will use the probe set as a hold-out validation set, and the training set will consist of all ratings not included in the probe set. A qualifying set, on which competition participants were supposed to make predictions, is available, but there are no true ratings for this dataset so we did not use it.

We preprocess the data by aggregating the four .txt files into one .csv file, with columns representing {user, movie, rating, date}, and each row representing a rating.

The massive size of this dataset provides a big challenge. Any models we train must be implemented as efficiently as possible in terms of both time and space complexity. We also have limited computing resources. To meet these needs, we will use implementations in both PySpark, which can read/write to disk and perform computations efficiently, and in C++, which gives the fastest possible computation speeds and most efficient memory management among common programming languages. All code can be found at the repository https://github.com/ggorti3/netflix-challenge.

3.2 Normalization

Before training models, we normalize the data to remove global effects. An example of a global effect would be if a given movie tends to be rated 1 point higher than other items – instead of relying on our model to learn this relation, we could simply normalize all ratings of a given movie, for example,

by subtracting away the item-wise mean rating from each movie. Similar ideas can be applied to user-wise mean. The normalization process we use is as follows:

Let μ denote the average of all ratings. Then, for every user i, let γ_i denote the average of adjusted ratings $r_{ij} - \mu$ across all movies j. Lastly, for each movie j, let τ_j denote the average of adjusted ratings $r_{ij} - \mu - \gamma_i$ across all users i. We then define

$$g_{ij} := \mu + \gamma_i + \tau_j$$

$$r'_{ij} := r_{ij} - g_{ij}.$$

We will train our algorithms to predict normalized ratings $\hat{r}'_{ij}$ and then add means back to recover $\hat{r}_{ij}$.

One can even generate a very basic global effects prediction using the normalization parameters,

$$\hat{r}_{ij} = g_{ij},$$

which provides surprisingly good results for such a simple procedure.

3.3 Models

3.3.1 Latent Factorization (ALS-WR)

We assign to each user i a k-dimensional factor $u_i \in \mathbb{R}^k$ and to each movie j a k-dimensional factor $m_j \in \mathbb{R}^k$. Using the training set, we wish to minimize

$$\sum_{(i,j)\in\mathcal{D}} (r'_{ij} - u_i^T m_j)^2 - \lambda(\sum_i \|u_i\|^2 + \sum_j \|m_j\|^2)$$

where λ is L2 regularization strength.

To solve this optimization problem, we use the Alternating Least Squares with Weighted λ Regularization (ALS-WR) [7] algorithm. The algorithm rotates between fixing the u_i and optimizing the m_j using a closed form least squares solution, and then vice versa with m_j fixed and u_i optimized.

Once the model is trained, we can generate the predictions

$$\hat{r}_{ij} = u_i^T m_j + g_{ij}.$$

We used the pyspark.ml.recommendation.ALS class from PySpark's pyspark.ml library to implement this algorithm. We were able to perform a brief hyperparameter search to select k and λ using coordinate descent, shown in tables 1 and 2. The parameters selected for this model were $k = 10$ and $\lambda = 0.05$.

Table 1: Probe RMSE at various ranks (latent vector dimensions) with $\lambda = 0.1$

Rank	Probe RMSE
Rank 5	0.94452
Rank 10	**0.93688**
Rank 20	0.93701

Table 2: Probe RMSE at various regularization strengths with rank 10

λ	Probe RMSE
$\lambda = 0.01$	0.98735
$\lambda = 0.05$	**0.93798**
$\lambda = 0.1$	0.93830
$\lambda = 0.8$	1.27270

3.3.2 "Neighborhood" Model using Interpolation Weights

Traditionally, neighborhood models use similarity measures to make predictions by using weighted averages of ratings for the most similar users or movies. However, as mentioned in the background discussion on neighborhood methods, explicit computation of similarities can be very difficult as the dataset grows in size, and storage of these similarities requires significant memory. The computational cost of computing all item-item similarities is $\mathcal{O}(n_u n_m s)$, and the space required to store them is $\mathcal{O}(n_m^2)$.

To receive some benefits of the neighborhood approach while reducing computation, we can alternatively learn our own weighting using "interpolation weights" [1]. Suppose we have a user-movie pair (i, j). Let J_i denote the set of movies rated by the user. We can train a model that predicts

$$\hat{r}_{ij} = g_{ij} + \sum_{j_2 \in J_i - \{j\}} w_{jj_2} r'_{ij_2}.$$

We tune the weights parameters w by optimizing

$$\sum_{(i,j)\in\mathcal{D}} \left(r'_{ij} - \sum_{j_2 \in J_i - \{j\}} w_{jj_2} r'_{ij_2} \right)^2 - \lambda \left(\sum_{j_1, j_2} \|w_{j_1 j_2}\|^2 \right)$$

We can view this optimization procedure as learning the optimal similarities between movies from the data, rather than computing them using an arbitrary similarity metric. Alternatively, we can interpret the interpolation weights w as scaling factors for the residuals from global estimates. For two related movies j and j_2, we expect w_{jj_2} to be high. Thus, whenever user i rates j_2 higher than expected ($r'_{ij_2} = r_{uj_2} - g_{uj_2}$ is high), we would like to increase our estimate for i's rating of j by adding $(r_{uj_2} - b_{uj_2}) w_{jj_2}$ to the global estimate. On the other hand, if user i rates j_2 as expected ($r_{uj_2} - b_{uj_2}$ close to zero), or j_2 isn't similar to j, (w_{jj_2} close to zero), our estimate will not deviate much from the global estimate.

We implemented our own stochastic gradient descent algorithm in C++ to train this model. Training times were lengthy, so instead of using a large hyperparameter search, we selected hyperparameters near the hyperparameters from literature [3], taking $\lambda = 0.05$ and using learning rate 0.0004 for 6 epochs with a decay of 0.9 after each epoch. There are some adjustments as our model is not exactly the same as in the reference.

3.3.3 Factorization with "Neighborhood"

To address the shortcomings of our individual models, we can learn them simultaneously. This can be done by simply adding the objectives of the previous two models; our new objective is

$$\sum_{(i,j)\in\mathcal{D}} \left(r'_{ij} - \left(\sum_{j_2 \in J_i - \{j\}} w_{jj_2} r'_{ij_2} \right) - u_i^T m_j \right)^2 + \lambda \left(\sum_{j_1, j_2} \|w_{j_1 j_2}\|^2 \right) + \lambda_2 \left(\sum_i \|u_i\|^2 + \sum_j \|m_j\|^2 \right)$$

and the prediction is

$$\hat{r}_{ij} = g_{ij} + \left(\sum_{j_2 \in J_i} w_{jj_2} r'_{ij_2} \right) + u_i^T m_j$$

We use a two-step training scheme. First, we use our SGD algorithm to optimize the weights w alone, without including latent factors in the model. Then, we fit the latent factors model using the ALS-WR algorithm on the residuals left from the "neighborhood" model. We use the same hyperparameters to tune the neighbors model as in the previous section. We take $k = 50$ and $\lambda_2 = 0.08$ for the latent factors model. We chose a higher latent dimensionality for this combined model due to suggestions from literature [3].

4 Results

Our results on the Netflix dataset are shown below. We use the global effects model (based on the normalization parameters explained in Section 3.2) and the Netflix's original CineMatch recommendation system as baselines, and each model is evaluated based on RMSE on the probe set.

Model	Probe RMSE
Baseline: Global Effects	0.9945
Baseline: CineMatch	0.9514
Factorization (ALS-WR)	0.9379
"Neighborhood"	0.9410
Factorization + "Neighborhood"	**0.9341**

Table 3: Comparison of RMSEs computed on probe dataset.

We observe that the normalization parameters alone can provide a reasonable prediction for unseen ratings, indicating that global effects make up a large part of a predicted rating. The latent factorization method greatly reduces RMSE, indicating its strength as a collaborative filtering method. The "neighborhood" model is also able to beat the CineMatch baseline on its own.

Lastly, combining the factorization method with neighborhood information achieved the best results.

This makes sense intuitively, as the model spaces of the factorization model and neighborhood model are both subsets of the model space of the combined model. That is, both model architectures can be represented as specific cases of the more general combined model. A different interpretation of the improved performance of the combined model is that we can view the model as a three-tiered system for predictions.

Recalling the prediction for the model,

$$\hat{r}_{ij} = g_{ij} + \left(\sum_{j_2 \in J_i} w_{jj_2} r'_{ij_2} \right) + u_i^T m_j,$$

the first tier, g_{ij}, describes general properties of the user and movie, without accounting for any relevant interactions, i.e. the global effects. The next tier, $u_i^T m_j$, provides the interaction between the latent representations of the user and movie, i.e. regional effects. Finally, the "neighborhood tier," $\sum_{j_2 \in J_i} w_{jj_2} r'_{ij_2}$, contributes specific adjustments to the prediction that are hard to quantity generally, i.e. local effects.

This lends credence to the idea that both models are able to learn relationships that the other alone cannot, and that using both together achieves more accurate predictions by learning both regional and local effects.

5 Conclusions and Future Work

We implement matrix factorization models and neighborhood models for the Netflix challenge, as well as a combined model in which latent factor and neighborhood information are both incorporated. We show that this combined model outperforms models that are based on only the former or latter as well as other baseline implementations, resulting in a significant improvement over Netflix's original CineMatch recommendation system. One main takeaway is that recommender system performance on this problem heavily depends on utilizing different aspects of the data. Through this project, we also gained an increased understanding of techniques for efficient applied statistics on large datasets, including distributed computing, memory management, and iterative optimization.

We are also thankful to have worked on a well-studied problem with an extensive history of published research. While we focused on fairly traditional methods for collaborative filtering in this project, these methods have since been extended to incorporate other aspects of the data, such as using the provided date information to add time-varying parameters to a model, which may help explain shifts in user preferences [5], or using implicit feedback, in which we view user ratings as implicitly conveying information about their preferences because they chose to submit a rating, rather than not submit one [3][4][5]. In recent years, the strength of deep learning-based approaches compared to traditional techniques on complex problems has also motivated the application of several deep learning architectures, including autoencoders, recurrent neural networks, and convolutional neural networks, to collaborative filtering problems.

References

[1] Robert M. Bell and Yehuda Koren. Scalable collaborative filtering with jointly derived neighborhood interpolation weights. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, pages 43–52, Oct 2007.

[2] Simon Funk. Netflix update: Try this at home, Dec 2006.

[3] Yehuda Koren. Factorization meets the neighborhood: A multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, page 426–434, New York, NY, USA, 2008. Association for Computing Machinery.

[4] Yehuda Koren. Factor in the neighbors: Scalable and accurate collaborative filtering. *ACM Trans. Knowl. Discov. Data*, 4(1), jan 2010.

[5] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[6] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, page 285–295, New York, NY, USA, 2001. Association for Computing Machinery.

[7] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In Rudolf Fleischer and Jinhui Xu, editors, *Algorithmic Aspects in Information and Management*, pages 337–348, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.