Bayesian SVD++ with Graph Convolutional Networks

Group: *Bayesians*

Arda Arslan
aarslan@student.ethz.ch

Gökberk Özsoy
goezsoy@student.ethz.ch

Sven Wiesner
swiesner@student.ethz.ch

Rajat Prince Thakur
rthakur@student.ethz.ch

ABSTRACT

In this paper, we develop a novel collaborative filtering approach by using a matrix factorization algorithm as the base, and empowering it with an idea from graph convolutional network research. On top of these, we employ Bayesian approach for defining network weights, which gives us a chance to understand how certain we are for a missing rating prediction. We optimize our model end-to-end, and conduct extensive experiments for showing its effectiveness. As *Bayesians* on Kaggle, we are in the top ranks in the competition.

1 INTRODUCTION

Online retailing has soared globally with easy access to the internet since last few decades. With millions of users and items, e-commerce companies need to match users with products of their taste to increase sales and customer satisfaction. On this direction, a recommendation system analyzes the user behaviour, and seeks to provide a specific rating that a user would give to an item. Here, the user behaviour can be deduced via combining the user's ratings to a set of items, and other descriptive features about users and items. Companies with millions of users tend to have huge volume of ratings data available, which triggers high quality research to develop robust recommender systems. Advancements in research returns as higher profits, which explains why this field is popular.

Collaborative filtering is a branch of recommendation systems. It only uses past ratings which means no other features about users or items are required. It assumes if a user x has same rating for an item with another user y, then x is more likely to have similar ratings for other items with y than a random user. In other words, for predicting a missing rating, the user collaborates with other users. Therefore, it aims to collectively understand relationships between different users, and between users and items.

We propose a collaborative filtering model which combines SVD++ [5], with two ideas from deep learning research, namely Graph Convolutional Nets (GCN) [1], and Bayesian Neural Nets (BNN) [2]. SVD++ is one of matrix factorization methods, where they learn low dimensional user and item embeddings using observed ratings data. They are proven to be simple and effective, and they exist with bunch of different variations [6]. In our model, we used SVD++ [5], where the author combines explicit and implicit ratings to learn better user and item embeddings. We used GCN to extend this idea further, by learning user and item embeddings from a graph network based autoencoder. Redefining the problem within graph framework, a more complex collaborative signal is created, which increases richness of embeddings. As the last extension, we converted our network to BNN for outputting uncertainty information for each rating prediction. This is promising, as any company would like to be sure if the item they promote is related to that particular user or not.

2 MODELS AND METHODS

In this section, we will first introduce our notation for the paper. Then, we will explain each fundamental idea embedded into our model, and finally we will present our model.

2.1 Notation

For this project, we are given a set of $N = 1,176,952$ integer movie ratings, ranging from 1 to 5, that are assigned by $m = 10,000$ users to $n = 1,000$ movies. Let $R \in R^{mxn}$ be the rating matrix, where rating R_{ui} indicates the preference by user u of item i. Unobserved user-item entries are simply blank. Also, let $\Omega = \{(u, i) : R_{ui} \text{ is known}\}$ be the set of user and movie indices for which the ratings are known.

2.2 Building Blocks of Our Model

Now, let us briefly mention about each fundamental idea one-by-one, combination of which will lead to our model.

2.2.1 SVD++[5]. Matrix factorization aims to find low dimensional embeddings of size k for each user and item with $k \ll min(m, n)$ such that user-item interactions are modeled as inner products of these embeddings. Let $p_u \in \mathrm{R}^k$ be a specific user's embedding and $q_i \in \mathrm{R}^k$ be a specific item's embedding. Thus, a rating can be estimated as

$$\hat{R}_{ui} = q_i^T p_u \tag{1}$$

SVD++ extends this idea by adding user and item specific biases ($b_i \in R$ and $u_i \in R$) and implicit item embeddings ($y_j \in R^k$). Thus a rating can be estimated as

$$\hat{R}_{ui} = \mu + b_i + b_u + q_i^T \left(p_u + |I_u|^{-0.5} \sum_{j \in I_u} y_j \right) \tag{2}$$

where μ is global average rating, and I_u is set of items user u rated. Regardless of the rating value, if a user provided a rating about an item, this is a clue about the user's preference pattern and hence needs to be exploited via adding additional parameters (y_j). The purpose of biases are capturing inherent rating behaviour of each user, and popularity of each item.

Here, p_u, q_i, b_u, b_i, and y_j are all parameters of the model, and to learn them, regularized squared error on the set of known ratings is minimized using stochastic gradient descent (SGD).

$$\underset{p_u, q_i, b_u, b_i, y_j}{\arg\min} \sum_{R_{ui} \in \Omega} \left(R_{ui} - \hat{R}_{ui} \right)^2 + \lambda \left(b_i^2 + b_u^2 + ||q_i||^2 + ||p_u||^2 + \sum_{j \in I_u} ||y_j||^2 \right) \tag{3}$$

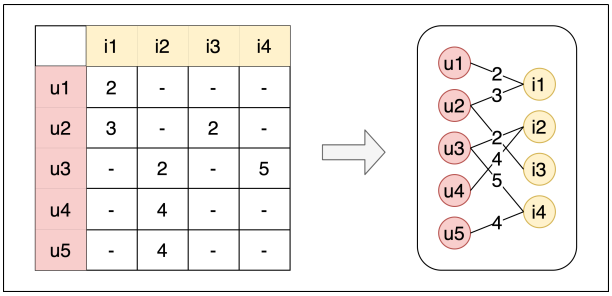


Figure 1: Conversion of rating matrix R to bipartite graph $\mathcal{G}$

2.2.2 Graph Convolutional Networks. Matrix factorization objectives (such as SVD++) end up having suboptimal embeddings because they lack an explicit encoding of the collaborative signal, which is latent in user-item interactions to reveal the behavioral similarity between users or items [7]. At this point, graph neural networks can be used to integrate user-item interactions into embedding process.

For this aim, first we need to convert the rating matrix R to a bipartite graph between user and item nodes, with observed ratings represented by links. With user nodes $u_i \in \mathcal{U}$ where i=[1:m], item nodes $v_j \in \mathcal{V}$ where j=[1:n], and edges (u_i, rating, v_j) $\in \mathcal{E}$, bipartite graph $\mathcal{G} = (\mathcal{U}, \mathcal{V}, \mathcal{E})$, which is shown in Fig. 1. After this step, numerous solutions exist in the literature, but we will follow Graph Convolutional Matrix Completion idea [1]. They propose a graph-based auto-encoder framework (Fig. 2) which produces latent features of user and item nodes through a form of message passing on $\mathcal{G}$. Then, the decoder uses latent features for users and items to reconstruct ratings. For our model, we will only use encoder part, thus we omit to describe decoder part further.

The encoder part holds different parameters for each edge type (i.e. number of ratings, 5 in our case), and shares these weights across all locations of the graph. Graph convolution idea is combining all incoming messages from neighbors to create embedding for the corresponding node (either user or item). We can define an edge type specific message as this:

$$\mu_{j \to i, r} = \frac{1}{c_{ij}} W_r x_j \tag{4}$$

where W_r is edge type specific parameter, and x_j is initial feature vector of node j, and c_{ij} is normalizer computed as $\sqrt{N_i N_j}$ where N_i is number of neighbors of node i. User and item nodes are not differentiated here and they are all processed the same way. Combining each of these edge type specific messages can be done as this:

$$h_i = \sigma\left(accum\left(\sum_{j \in N_{i,1}} \mu_{j \to i,1}, ..., \sum_{j \in N_{i,R}} \mu_{j \to i,R}\right)\right) \tag{5}$$

where *accum* is either stack or sum. With intermediate embedding h_i, we are only one step behind of arriving final embedding for a node (again either user or item). The authors designed to add one more nonlinear layer as this:

$$u_i = \sigma(W h_i) \tag{6}$$

With this, we finally have user embedding $p_u \in R^k$ or item embedding $q_i \in R^k$. This concludes the encoder part of the model which is enough for the scope of our paper.

2.2.3 Bayesian Networks[2]. Ordinary neural networks are unable to reflect uncertainties in the training set, and they are prone to overfitting. Assigning probability distributions to network weights instead of single fixed values helps the networks to cope with these problems.

Given the training set, posterior distribution $p(w|D)$ of weights can be adjusted. However, this is not a straightforward task. With huge numbers of weights and accompanied nonlinearities, calculating the posterior in exact form is intractable. Variational learning provides a solution here by learning the parameters of a known distribution (e.g. Gaussian) that minimizes the KL-divergence with true posterior distribution [3]. More precisely, let known distribution be $q(w|\theta)$, where θ is defining parameters of $q(w)$, and prior distribution of weights be $p(w)$, then:

$$\theta^* = \arg\min_{\theta} \mathbf{KL}[q(w|\theta)||p(w|D)] \tag{7}$$

$$= \arg\min_{\theta} \int q(w|\theta) log\left(\frac{q(w|\theta)}{p(w)p(D|w)}\right) dw \tag{8}$$

$$= \arg\min_{\theta} \mathbf{KL}[q(w|\theta)||p(w)] - \mathbf{E}_{w \sim q(w|\theta)}[log p(D|w)] \tag{9}$$

In (7), we wrote the objective we want to minimize. In (8), we explicitly wrote KL divergence, and in (9) we grouped together expressions which have following meanings: first part says we want θ which makes approximate posterior q(w|θ) as close as possible to prior p(w). This can be seen as a form of regularization, where we do not let parameters to have any arbitrary value. The second part says we want to find θ which increases the data likelihood, which can be interpreted as increasing regression performance (in our case regressing rating values between 1 and 5). In our implementation, we choose approximate posterior q(w) to be a Gaussian distribution with mean μ and standard deviation σ. Hence, θ = (μ,σ). Eqn (9) is minimized by taking Monte Carlo samples from posterior $q(w)$, and calculating gradients of θ with respect to objective estimation (loss value).

Finally, we will explain prediction procedure for unseen samples. Let $\hat{y}$ be label (rating in our case), $\hat{x}$ be input features, and $q(w|\theta^*)$ be estimated approximate posterior. Then, prediction is as follows:

$$p(\hat{y}|\hat{x}) = \mathbf{E}_{w \sim q(w|\theta)}[p(\hat{y}|\hat{x}, w)] \tag{10}$$

Since each weight is a distribution, for each prediction, we need to sample from these distributions. This is an opportunity for infinite size ensemble as each sampling will create different set of weights. However, instead of evaluating expectation exactly, we sample finite (e.g. 5) times and take average of the results. As more data are observed, the uncertainties on weights can decrease, leading to a more deterministic prediction.

2.3 Our Model

In this section, we will explain how we combined the ideas explained above to create our model, along with training procedure and implementation details.

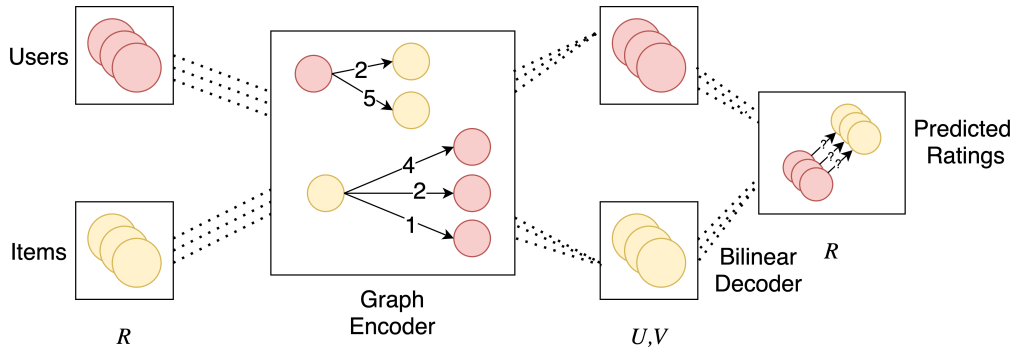


Figure 2: Graph Convolutional Matrix Completion Framework. In our model, we will only use graph encoder part. [U,V] = Enc(R), $\hat{R}$ = Dec(U,V), where U and V are matrices of user and item embeddings, respectively.

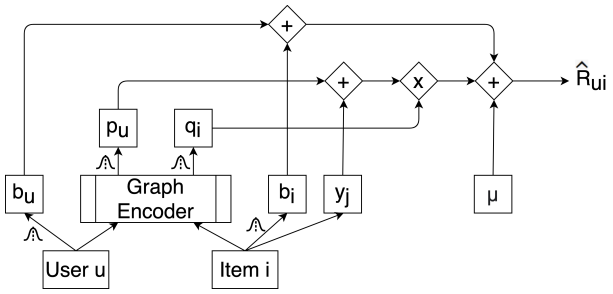


Figure 3: Schema of our model.

2.3.1 Objective Function. In our model, we used SVD++ as skeleton, where we have user and item embeddings (p_u,q_i), user and item biases (b_u,b_i), and item implicit embeddings (y_j). However, to better represent item and user interactions, we used encoder part of graph based autoencoder framework. This allows us richer embeddings p_u and q_i. Finally, we converted all weights to be normal distributions instead of fixed single values so that we would get uncertainty estimation about a predicted ranking, and regularization effect on the network. One critical point to mention here is the output of the graph encoder is not direct embeddings (p_u, q_i), but the parameters of Gaussian whose sampling will give p_u and q_i. Thus, for both p_u and q_i, the output space of the encoder is divided into two, one half is for μ, and the other half is for σ. For each forward pass, we sample from $\mathcal{N}(\mu,\sigma)$ to get p_u and q_i. Having uncertainty about a predicted ranking has practical advantages, as one now can choose whether to show a product as recommended to a user or not. Composing all these extensions together, we name our model as "Bayesian SVD++ with Graph Convolutional Networks", and it can be trained end-to-end smoothly. The following equation is objective function containing the idea represented above.

$$\theta^* = \arg\min_{\theta} \mathbf{KL}[q(w|\theta)||\mathcal{N}(0,I)] - \mathbf{E}_{w\sim q(w|\theta)}[logp(R_{ui}|u,i,w)] \tag{11}$$

$$where \quad p(R_{ui}|u,i,w) \sim \mathcal{N}(\hat{R}_{ui}, I) \tag{12}$$

where $\hat{R}_{ui}$ is calculated as in Figure 3.

2.3.2 Training. Training procedure is as follows (intended to be closer to a pseudocode):

(1) Sample a user-item entry whose rating is known.
(2) Feed graph encoder with one-hot user(u) and item(i) vectors.
(3) Sample weights $w \sim q(w|\theta)$ to get b_u, b_i, p_u and q_i.
(4) Compute prediction as using Eqn. (2).
(5) Calculate $l = \sqrt{(R_{ui} - \hat{R}_{ui})^2} + \sum_{\forall\theta} \mathbf{KL}[q(w|\theta)||p(w)]$ where R_{ui} is ground truth rating, **KL** is KL divergence between each weight's distribution and unit Gaussian prior.
(6) Compute partial derivates $\frac{\partial l}{\partial \theta}$ and update each θ via Adam optimizer.

GCN is used in step 2, SVD++ in step 4, and BNN in steps 3 and 5.

2.3.3 Prediction. Given a missing entry for a user-item interaction, we predict the rating by taking samples from posterior weights. This is a natural outcome of treating the problem as Bayesian, and leads to different predictions each time for the same input features due to randomness. As stated before, we can take average of each prediction to declare final rating prediction. This is an example of infinite ensemble. In our model, we used 5 different sampling for a prediction. Uncertainty comes from calculating standard deviations of prediction set.

2.3.4 Implementation Details. We used the code from [8] for preparing data for the graph-encoder, the GCN architecture and the overall training and evaluation pipeline. We implemented SVD++ and Bayesian approach on top of this code using PyTorch. At the end, we have a pipeline that is end-to-end trainable. In addition, our implementation is modular as we can add any feature we want on top of plain SVD++ implementation. Models which can be created easily via our code is SVD++, Bayesian SVD++, SVD++ with GCN, and finally our model Bayesian SVD++ with GCN. On the other hand, we have an objective function of two parts: first part detects prediction quality(RMSE) and the second part detects if our posterior distributions are close to priors(KL). In our implementation we give importance coefficient λ to KL part, for decreasing its contribution to loss throughout training. We also implemented hyperparameter tuning which we will discuss in the next section.

Method	Val Error	Test Error	Time(s)
General Average	1.120	1.117	8
SVD [4]	1.065	1.064	16
SVD++ [4]	0.9788	0.9785	85189
Bayesian SVD++	0.9989	0.99719	100242
SVD++ GCN	0.9744	0.9709	210
Bayesian SVD++ GCN	0.9781	0.9781	344

Table 1: Performance comparison of our model with baselines.

3 RESULTS

3.1 Hyperparameter Tuning

Hyperparameters in our model are from Bayesian and GCN sides. For Bayesian, we should choose prior μ and σ, as well as KL coefficient λ. For GCN, we need to choose intermediate and final output sizes, dropout rate, *accum* function type. For these, we splitted given train set into 3 fold: 80% for training, 10% for hyperparameter tuning, and 10% for testing before submission to Kaggle. We used Adam optimizer, employed learning rate scheduling and early stopping.

3.2 Performance Comparison

We enlisted performance of our model against different baselines in Table 1. In that table, 'Val Error' indicates each model's performance with best hyperparameters selected on our test split. 'Test Error' means each models' performance on Kaggle public test set. Finally, time is in seconds. General average is the average of all known ratings served as prediction. SVD and SVD++ are from surprise library as a sanity check for our model. Bayesian SVD++ and SVD++ GCN are baselines implemented by ourselves discussed in previous section. Finally, Bayesian SVD++ GCN is our proposed model.

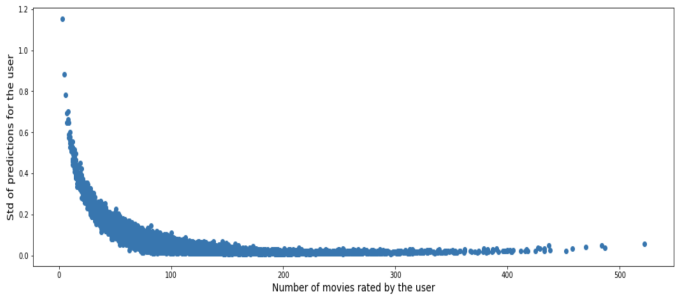
3.3 Uncertainty on Predictions

We advertised being Bayesian as an advantage as the company can see how much our model is certain about an item that will be offered to the user. In Fig.4, we can see its realization. As the number of items rated by a user increases, the model will become more certain whether or not a specific item is user's taste. Similarly, as more users rate a specific item, the model becomes more certain whether or not to offer that item to a user who has not used it.

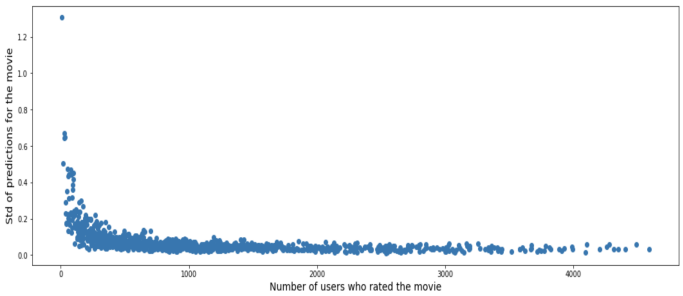
4 DISCUSSION

Increasing model complexity does not always help. One needs to understand a model's deficits to reinforce it with others. We thought that matrix factorization models only care about a specific user's and item's embedding during SGD as only they will be updated. However, this is not enough, and that is why we turned to GCN for richer embeddings, and Bayesian for reporting standard deviation in predictions. Original GCN autoencoder gives embeddings, but we converted it to give mean and variance of weights which will create embeddings. This was a nice twist.

Making the network Bayesian adds KL divergence term to loss which increases it. It also serves as regularization on weights, and its effect is apparent when we compare SVD++ GCN with Bayesian SVD++ GCN, as the latter has worse performance by a small margin. This makes the SVD++ GCN as our best performing model in Kaggle, however it cannot output uncertainty information which is a huge drawback in real world commerce. On the other hand, addition of GCN has a positive effect as the best two models has it.



(a) Relation between number of movies rated by the user and standard deviation of that user's predictions



(b) Relation between number of users rated the movie and standard deviation of that movie's predictions

Figure 4: Standard deviation on predictions for users and items

Variance of predictions decrease dramatically as we observe more data. This is common outcome of Fig. 4 for both users and items. Results are parallel with what we expect by looking to the theory. Epistemic uncertainty is the uncertainty caused by lack of data, and more data should decrease it. We cannot do anything about noise inherent in the data which makes aleatoric uncertainty, but reduction in epistemic uncertainty proves that our implementation and mathematical modelling is correct.

Discrepancy in computation time between Bayesian SVD++ and other models with GCN is that in the former we used mini-batch processing, but on the latter we used full-batch processing as the data can fit in GPU. We wanted to keep the former to mention the performance difference here.

5 CONCLUSION

In this paper, we proposed a novel collaborative filtering model, with recent learning ideas. SVD++ is simple and effective, yet it provides suboptimal embeddings. We supported it with GCN to squeeze more information from user-item relations. Lastly, Bayesian idea adds a new perspective to recommendation. With modular implementation, smooth end-to-end training, we ended up in top 5 in Kaggle, and showed the power of our approach.

REFERENCES

[1] Rianne van den Berg, Thomas N Kipf, and Max Welling. 2017. Graph convolutional matrix completion. *arXiv preprint arXiv:1706.02263* (2017).

[2] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. 2015. Weight uncertainty in neural network. In *International Conference on Machine Learning*. PMLR, 1613–1622.

[3] Alex Graves. 2011. Practical Variational Inference for Neural Networks. In *Proceedings of the 24th International Conference on Neural Information Processing Systems (NIPS'11)*. Curran Associates Inc., Red Hook, NY, USA, 2348–2356.

[4] Nicolas Hug. 2020. Surprise: A Python library for recommender systems. *Journal of Open Source Software* 5, 52 (2020), 2174. https://doi.org/10.21105/joss.02174

[5] Yehuda Koren. 2008. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 426–434.

[6] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8, 30–37. https://doi.org/10.1109/MC.2009.263

[7] Xiang Wang, Xiangnan He, Meng Wang, Fuli Feng, and Tat-Seng Chua. 2019. Neural graph collaborative filtering. In *Proceedings of the 42nd international ACM SIGIR conference on Research and development in Information Retrieval*. 165–174.

[8] Jiani Zhang. 2020. https://github.com/dmlc/dgl/tree/master/examples/pytorch/gcmc.