Replication of *Performance of Recommender Algorithms on Top-N Recommendation Tasks*

Michael Dunn
University of Minnesota
Minneapolis, Minnesota
dunn0562@umn.edu

ABSTRACT

Across the various domains that recommender systems operate within, the problem of recommending multiple items to the user at once is common. This problem is known as the "top-N recommendation problem," and improving the quality of the *N* items shown to the user is advantageous from both a business and consumer perspective.

Often, the algorithms behind these systems are evaluated using error metrics (like MAE or RMSE), which measure the error between the algorithm-predicted ratings and actual user ratings. However, in many systems, the user is not shown these predicted ratings, rather, they are only given a list of recommendations. In these cases, it does not matter how well the algorithm can predict values if the top-N list of movies does not contain quality results. For this reason, evaluating recommender algorithms with accuracy statistics, like recall or precision, can instead measure how many "good" recommendations were provided in the top-N list.

Another issue encountered while generating top-N lists is the frequent inclusion of highly popular items. This set of popular items, identified as the items that fall in the "*short-head"* of the popularity distribution, often end up in the top-N recommendation lists provided to users. However, these items are uninteresting to the users, as they are likely to have already seen them. For this reason, testing the ability of recommender algorithms to produce recommendations over the "*long-head"* items, the items that are considered "unpopular", is important.

Through the replication of the data preparation, algorithms, and evaluation frameworks of the article titled "Performance of recommender algorithms on top-n recommendation tasks" by Paolo Cremonesi, Yehuda Koren, and Roberto Turrin, I verified that their findings were robust. However, with the minimal inclusion of tuning and implementation details, I was unable to obtain their exact results throughout my own reimplementation.

KEYWORDS

Recommender Systems, Professor Kluver, Top-N Reccomendation, Replication

1 Introduction

The practice of evaluating recommender algorithms with error metrics, like MAE (mean average error) and RMSE (root mean squared error) is commonly used in the field of recommender systems. By calculating the error between algorithm-predicted ratings and actual user ratings, these metrics are a simplistic way to determine the predictive accuracy of an algorithm. However, accuracy metrics, like precision and recall, can measure the quality of top-N predictions directly, which is especially helpful when evaluating the ability of an algorithm to generate top-N lists.

Another important factor to consider while evaluating top-N lists is which items are included. Items with the highest count of ratings, the most popular items, are not useful recommendations, since the likelihood that a user has already heard of the recommended item, and decided not to rate it, is high. Removing popular items, specifically the items in the "long-tail" of the popularity distribution, from the dataset before evaluating the algorithm can outline which algorithm is best suited for lesser-known item recommendation. The methodology used to replicate the long-tailed item selection is discussed further in Section 2.1.

The work done in [1] outlines the comparison of non-personalized, collaborative, and latent-factor algorithms in the creation of top-N recommendation lists over the complete set and long-tailed Movielens ratings. Although detailed processes and tuning were not included in [1], their explanation of algorithms and evaluation frameworks was enough to implement their processes and recreate their results myself.

2 Testing Methodology

To avoid algorithmic overfitting, in [1], the Movielens dataset was split into train, probe, and test sets. To create the probe set, 1.4% of the total Movielens ratings were sampled, leaving the remaining 98.6% of the data to the training set *M*. The test set *T* was created by filtering only 5-star ratings from the probe set, ensuring "that *T* only contains items relevant to the respective users" (Cremonesi et al., 2010). Following this protocol on the Movielens dataset, the train and test sets were |M| = 3704 and |T| = 1098 respectively.

2.1 Short head and long tail test set creation

To test the ability of each of the algorithms to recommend lesser-known items, items in the dataset were sorted into *the long-tail* and *short-head* of the item popularity distribution. Within the Movielens dataset, it is observed that "the short-head (33% of ratings) involves the 5.5% of popular items" (Cremonesi et al., 2010). Creating an additional test set, T_{long} that only contains items found in the long-tail of the distribution allows us to measure the ability of each algorithm to create non-trivial recommendations.

However, in my recreation of the T_{long} set, I was unable to match the exact size. Although [1] describes the short-head to contain 5.5% of popular items (or 213 items), 5.5% of my test set *T* is 204 items. Therefore, I found that $|T_{short}|$ = 204, and $|T_{long}|$ = 894. Despite the minor variance in test set sizes, my replicated algorithmic output matches the findings in the study.

2.2 Recall and precision evaluation

Top-N recommendation comparison of each of the eight algorithms required the replication of the recall and precision evaluation algorithms described in the study. The first step for implementing the evaluation framework was to train the model on all ratings in the training set *M*. Next, for each 5-star rating of items within the test set *T*, a randomly selected list of 1000 items unrated by the creator of the rating was selected. Then, using the model trained on *M*, a list of predictions for each of the 1000 unrated items, combined with a prediction for the 5-star rated item, was generated. Finally, the items were sorted in descending order, and a top-N list was made by taking the first *N* items in the sorted list. If the rated item was contained in the top-N list, a "hit" is recorded, otherwise, the next rating in the test set would be tested. The following equations for recall and precision were given in [1], and I implemented them in my replication. Note that |T| is the number of total test ratings:

$$recall(N) = \frac{\#\ hits}{|T|}$$

$$precision(N) = \frac{\#\ hits}{N * |T|} = \frac{recall(N)}{N}$$

By modifying the precision/recall function provided in the Surprise documentation [4], I was able to replicate the testing criteria used in [1]

3 Algorithms

Demonstrated in [1] are several non-personalized, collaborative, and latent-factor algorithms, all of which follow the testing methodology outlined in Section 2, except the Top Popular algorithm, whose specific evaluation criteria are outlined in Section 3.1.

3.1 Non-personalized algorithms

The first two algorithms that I implemented and evaluated from [1] are the *Movie Average* (MovieAvg) and *Top Popular* (TopPop). Movie Average generates a prediction for an item by a user as "the mean rating expressed by the community on item *i,* regardless of the ratings given by *u*" (Cremonesi et al., 2010). The Top Popular algorithm generates the same top-N list of most popular items as recommendations for every user, regardless of their rating history. Because the Top Popular algorithm generates a top-N list of item recommendations, and not a predicted rating or association value between users and items, the recall evaluation does not generate 1000 unrated items, and instead, returns a "hit" if the test item is contained within the top-N list.

3.2 Collaborative Algorithms

The first collaborative algorithm that is described in [1] is an item-item collaborative filtering algorithm that uses Pearson similarity to implement a k-nearest-neighbor approach. This model is named the *Correlation Neighborhood* (CorNgbr), and the equation for generating predicted ratings by user *u* on item *i* listed is as:

$$\hat{r}_{ui} = b_{ui} + \frac{\sum_{j \in D^k(u;i)} d_{ij}(r_{uj} - b_{uj})}{\sum_{j \in D^k(u;i)} d_{ij}} \quad (1)$$

Here, $D^k(u;i)$ is "the set of most similar items", where d_{ij} is the shrunk Pearson similarity between two items calculated using this equation:

$$d_{ij} = \frac{n_{ij}}{n_{ij} + \lambda_1} s_{ij} \quad (2)$$

To replicate the usage of this algorithm, I used the Surprise library [4], which has a built-in item-item k-nearest neighbor algorithm with the equation

$$\hat{r}_{ui} = b_{ui} + \frac{\sum_{j \in N_u^k(i)} sim(i,j) \cdot (r_{uj} - b_{uj})}{\sum_{j \in N_u^k(i)} sim(i,j)} \quad (3)$$

where $N_u^k(i)$ is the equivalent set to $D^k(u; i)$. Additionally, to ensure that the shrunk similarity is used, the parameter "shrinkage" in "sim_options" is set to a value of $\lambda_1 = 100$, which follows the recommended value λ_1 listed in [1].

However, aside from a recommended shrinking factor, and the measure for similarity being used, no other tuning parameters were listed within the study. For this reason, the default parameters outlined in the Surpirse documentation were used in my replication, leading to results that varied slightly from [1].

The other neighborhood model that was outlined in the paper was a *Non-normalized Cosine Neighborhood*. The equation for this algorithm is very similar to (1), with the difference coming from the removal of the normalization factor and the use of Cosine similarity instead of Pearson correlation.:

$$\hat{r}_{ui} = b_{ui} + \sum_{j \in D^k(u;i)} d_{ij}(r_{uj} - b_{uj}) \quad (4)$$

In both [1] and my replication, this model is referred to as NNCosNgbr. Due to the removal of the normalization in this algorithm, predicted ratings $\hat{r}_{ui}$ no longer fall in the range [1…5], and instead, $\hat{r}_{ui}$ is a measure of the relationship between *u* and *i*.

According to Cremonesi et al. (2010), the benefit of removing the normalization is a "higher ranking for items with many similar neighbors… where we have a higher confidence in the recommendation".

Similar to CorNgbr, specific tuning information was not listed for this algorithm, and I was unable to replicate the exact results of the study. However, using a modified version of the item-item k-nearest neighbor algorithm provided in the Surprise library [4], I was still able to produce results with this algorithm.

3.3 Latent Factor Algorithms

Alongside the non-personalized and collaborative algorithms listed in [1], three latent factor models were outlined. For each of the following algorithms, specific tuning values were not provided, and the default values listed in the Surprise documentation [4] were chosen unless otherwise specified.

The first latent factor model described in [1] is *Asymmetric-SVD* (AsySVD), and is a biased version of a Singular Value Decomposition algorithm, computed with the equation:

$$\hat{r}_{ui} = b_{ui} + p_u q_i^T \quad (5)$$

To replicate the implementation of this algorithm, I again used the Surprise library [4], which has a biased SVD algorithm implementation, which I ran with *n_factors = 200* to replicate the results from the study.

The next latent factor algorithm that was used in [1] was the SVD++ algorithm. A specific algorithm was not provided within [1], but a reference to Koren (2018) outlines the following equation for the SVD++ model [2]:

$$\hat{r}_{ui} = b_{ui} + q_i^T(p_u + |N(u)|^{-\frac{1}{2}} \sum_{j \in N(u)} y_j) \quad (6)$$

An implementation for the SVD++ model is provided in the Surprise library [4], with the caveat that an extra term μ is added to $\hat{r}_{ui}$. To closer match the procedure listed in the study, I overrode the estimate() function in the Suprise implementation for SVDpp() and removed the value of μ. Similar to *Asymmetric-SVD,* the SVD++ model was run with 200 factors.

3.3.1 Pure SVD

The third latent factor model experimented in [1] is *PureSVD*, a true singular value decomposition modeled by this equation:

$$\hat{R} = U \cdot \Sigma \cdot Q^T \quad (7)$$

where the item factor matrix Q and the user factor matrix P is defined as $P = U \cdot \Sigma$. Because the goal of top-N recommendation is not precise predicted rating values, using *PureSVD* is possible, since we can "consider all missing values in the user rating matrix as zeroes, despite being out of the 1-to-5 star rating range" (Cremonesi et al., 2010). With this in mind, Cremonesi et al. derived the following equation for generating a user-item association metric, in place of a predicted rating:

$$\hat{r}_{ui} = r_u \cdot Q \cdot q_i^T \quad (7)$$

where r_u is the *u*-th row of the user-item rating matrix.

To obtain the user and item factor matrices, I used the SVDLIBC library [3]. Cloning the SVDLIBC Git repository, converting the user-item rating matrix into a text file, and running the following commands allowed me to create U and Q in the same manner outlined in [1]:

./svd -o result_50 -d 50 -rdt rating_matrix.txt

./svd -o result_150 -d 150 -rdt rating_matrix.txt

Two commands were used because the study compares two *PureSVD* models, one with 50 latent factors, and one with 150 latent factors.

Unlike the previous two latent factor algorithms, the Surprise library [4] does not have a direct implementation of *PureSVD*, and I implemented it myself using the "algorithm base class (AlgoBase)" from the Surprise documentation [4] and (7).

4 Results

Figure 1 is a graph of recall for each of the different algorithms at the various sizes of *N*. When compared to the graph displayed in [1], we can see the similarities in the results produced by my replication. Firstly, the non-personalized and collaborative algorithms were outperformed by each of the latent factor algorithms (AsySVD, SVD++, and PureSVD). Although PureSVD "[is] not sensible from an error minimization viewpoint and cannot be assessed by an RMSE measure", we see that it outperforms SVD++, the algorithm which "represents [the] highest quality in RMSE-optimized factorization methods" (Cremonesi et al., 2010). Specifically, my implementation of PureSVD reaches a recall of 0.5208 at N = 10, the same as the recall value provided in [1].

Figure 2 shows similar results for each algorithm when trained over items in the long-tail of the rating popularity distribution. Overall, the recall curves are less steep in FIgure 2 as compared to the algorithms trained over all items, and the trend of latent factor models being superior is consistent. These less-steep curves align with the idea that "recommending less known items… is usually a more difficult task" (Cremonesi et al., 2010).

Furthermore, my replication shows the relationship between the lower-performing models, specifically between CorNgbr and MovieAvg. Although [1] refers to CorNgbr as "widely used", the results in my replication emphasize its underperformance when compared to PureSVD.

Finally, in an effort to keep my replication paper concise, precision values and graphs were omitted. But since precision is calculated by dividing recall(*N*) by *N*, the precision values calculated by my replication are in line with the precision values in [1].

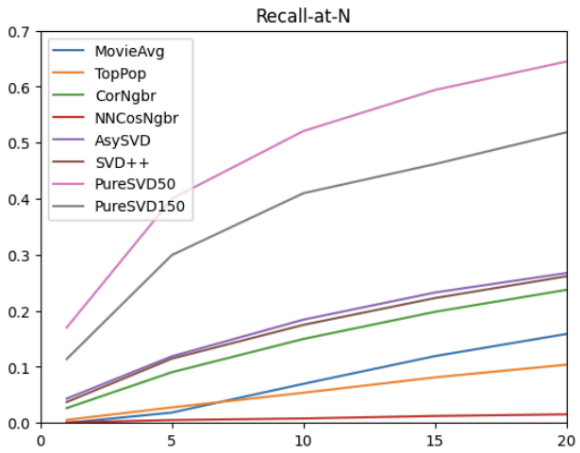

Figure 1: recall-at-*N* on all items.

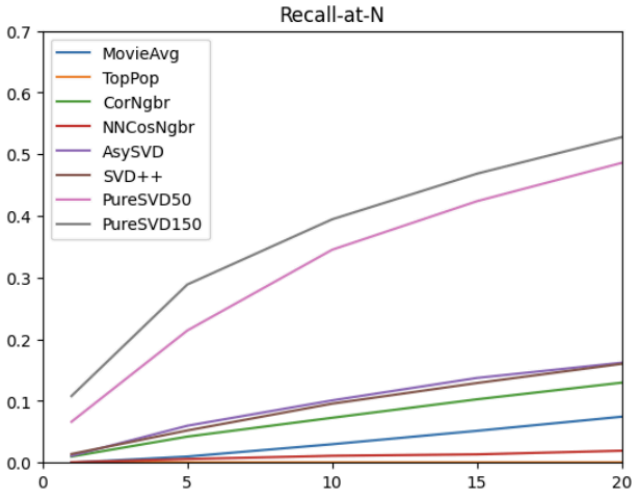

Figure 2: recall-at-*N* on items in the long-tail.

5 Discussion

Due to the minimal tuning details included in [1], obtaining the exact results outlined in the study was difficult. One example of this is my results from SVD++. Although [1] describes their SVD++ results as "competitive with that of PureSVD50 for small values of recall", I could not replicate the tuning used by SVD++ to achieve these values. Following the default values set in Surprise [4] as well as the recommended tuning values in [2], the reference for SVD++ given by the authors only resulted in values competitive with AsySVD.

Additionally, I was unable to recreate the "strange and somewhat unexpected result of TopPop" (Cremonesi et al.,

2010) and NNCosNgbr, both of which were supposed to be competitive with the more powerful latent factor models.

Because TopPop was the only algorithm that was designed specifically to produce top-N lists, without generating a rating or comparison value, I was unable to perfectly apply the recall evaluation steps outlined in the second section of [1], as I could not “predict the ratings for the test item *i* and for the additional 1000 items” (Cremonesi et al., 2010). My TopPop algorithm implementation followed the description in [1] and always recommended the top-N items that had the most reviews. I believe that Cremonesi et al. (2010) may have used a different recall calculation for TopPop that was not included in the study, leaving my best-judgment implementation to result in different values.

For NNCosNgbr, my implementation follows the equation written in [1] but was unable to produce the impressive recall values described in the paper. The equation for CosNgbr () is nearly the same as the equation for NNCosNgbr (?), with the difference of a removed normalization factor in the denominator, so I believe the difference in results is not due to implementation, but rather, due to tuning or specifics not mentioned in [1].

REFERENCES

[1] Paolo Cremonesi, Yehuda Koren, and Roberto Turrin. 2010. Performance of recommender algorithms on top-n recommendation tasks. In Proceedings of the fourth ACM conference on Recommender systems (RecSys '10). Association for Computing Machinery, New York, NY, USA, 39–46. https://doi.org/10.1145/1864708.1864721

[2] 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 (KDD '08). Association for Computing Machinery, New York, NY, USA, 426–434. https://doi.org/10.1145/1401890.1401944

[3] Rohde, D. (2005) *SVDLIBC Library Documentation*, *SVDLIBC*. Available at:https://alize.univ-avignon.fr/svn/LIA_RAL/trunk/LIA_SpkDet/CovIntra/src/SVDLIBC/Manual/index.html (Accessed: 05 May 2025).

[4] Hug, N. (2015) Surprise’ Documentation. Available at: https://surprise.readthedocs.io/en/stable/index.html (Accessed: 06 May 2025).