Employing Implicit and Explicit Feedback Data to Construct a Synergistic Collaborative Filtering System for Goodreads Users

Lauren D'Arinzo (lhd258) and Alene Rhea (akr435)
Center for Data Science, New York University

1. Overview

Background:

Algorithmic recommender systems utilize feedback data to model the relevance of an item for a unique user. In the age of big data, they exist ubiquitously across domains, and are especially useful in contexts where implicit feedback and explicit ratings occur frequently and at high volume. Goodreads is a social media platform for book reviews, where users have the ability to document content that they have read and subsequently receive recommendations based on their interactions with those books. Users are able to provide feedback explicitly, in the form of a 5-star rating system, free text reviews, and flagging books they have read before. The platform's success relies on recommendation algorithms that are able to handle hundreds of millions of user-book interactions.

Objective:

We sought to first implement and evaluate a baseline collaborative filtering recommender system using only user-book interactions containing rating data. Subsequently, we expanded our approach to include additional interaction data via an implicit is_reviewed flag, in order to implement a more comprehensive system that takes into account two different types of feedback data.

Dataset:

The dataset we used was scraped from goodreads.com in 2017 by Wan and Macauly as part of the UCSD Book Graph.[1] The deidentified file of all publicly displayed user-book-rating interactions contains data from 876,145 unique users, and a total of 228,648,342 user-book-rating interactions. Each instance represents a unique user-book interaction and its rating (INT 0-5), an is_read flag (0/1), and an is_reviewed flag (0/1). From user experience experimentation on the goodreads platform, we deduced that rating a rating of 1-5 is an explicit feedback mechanism -- users take an action to select the five star rating for a book. 'Is read' is also an explicit feedback mechanism -- users must click to indicate that they have read a book and the field is not auto-filled just by having rated a book. On the other hand, 'is_reviewed' is a form of implicit feedback; when users write a free text review (explicit feedback), this flag becomes 1 without any further action required by the user.

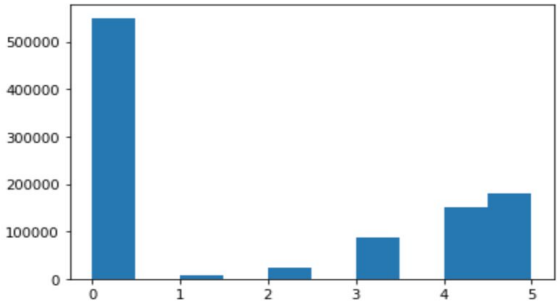
2. Data Processing

Data Profiling

In profiling the distribution of ratings across all interactions, we observed a high frequency of instances where the rating was 0. Surprisingly, these instances represent ~51% of all interactions (Figure 1), despite the dataset documentation suggesting that possible rating values are 1-5. We contacted Wan (author of the UCSD Book Graph), who confirmed that the interactions where rating = 0 represent cases where users have not yet rated a book. Thus, we decided to drop these instances for the baseline implementation, as they do not meaningfully convey a numeric rating.

[1] https://sites.google.com/eng.ucsd.edu/ucsdbookgraph/home

Figure 1. Frequency distribution of rating values show '0' as most common rating on subset of 1 million interactions



Quality Checking and Synthetic Data

In order to debug and perform quality checks on our pipeline, we created synthetic datasets using the same schema as the Goodreads data. This allowed us to manually inspect entire dataframes, and to thoroughly track functions' behavior. We integrated boolean 'synthetic' options into several of our functions to prevent synthetic results from being written to HDFS or saved to results files. We wrote a detailed *quality_check*() function which tracked progress and timing of the data preparation process, and compared results and aggregations to our expectations. We incorporated 'debug' options into the pipeline as well.

Preprocessing:

Our first step was to save the entire interactions dataset, ordered by user_id, to Parquet. This was a time-intensive step, however we only had to do it once to greatly speed up the rest of our data processing.

Because the latent factor model used here relies on matrix factorization of user and item vectors, users that do not have many book interactions may not provide adequate information to successfully recommend 500 other books. Consequently, we selected users having 10 or fewer interactions, and dropped all of their interactions, after having removed instances where rating = 0.

We additionally generated random downsample splits of 1%, 5%, and 25% of the dataset in order to optimize a working pipeline efficiently before attempting to tune hyperparameters on 200+ million observations. Our downsample function used *sample()* with a random seed; quality checking on synthetic and real data revealed that passing the same random seed to the sample function was not sufficient to ensure identical results. Research revealed that this is due to the fact that Spark does not guarantee the order of dataframe rows. To ensure that our samples were consistent, we used *persist()*.

Training/Validation/Test Split:

We constructed train, validation, and test splits such that all users who appear in the validation and test sets also have interactions in the training set; this is essential because collaborative recommender systems are unable to make successful predictions on users they have never encountered before. This was implemented by first generating a list of distinct user ids, then using the *randomSplit()* method to assign 60% of users to train, 20% to validation, and 20% to test. As with sample(), we used *persist()* after the call to *randomSplit(),* in addition to passing a consistent random seed. Next, to ensure that all users in the validation and tests were in fact observed, we used *sampleBy('user_id')* to move approximately 50% of each validation and test user's interactions back to training, and to keep as the difference between the original validation/test set and the set moving back to training, as the validation/test set. Using a differencing operation (i.e. EXCEPT in SQL) was crucial, so as not to lose or duplicate any tuples at this stage. Once again, quality checking revealed that it was necessary to use *persist()* in addition to a random seed after the call to *sampleBy('user_id')*. Because it uses Bernoulli sampling, *sampleBy()* does not guarantee an exact fraction. This meant there were actually some users for whom all their interactions ended up staying in validation/test. We moved all the interactions for those users to train. Finally, to make sure that all the books in validation/test are also observed by train (for the same reason we had to ensure this for users), we used an INNER JOIN to drop all interactions of unobserved books from validation and test.

Storage Optimization:

Our implementation pipeline allows for the separation of data preparation and modeling, by providing an option to write the downsampled training, validation, and test sets to Parquet, ordered by user_id. This means that we only need to run the data preparation steps one time per downsample fraction size, and that we can consequently reap benefits that come with column oriented storage, like predictable access patterns and faster performance on aggregation queries.

3. Collaborative Modeling with Alternating Least Squares

The alternating least squares method:

Alternating Least Squares (ALS) is a bi-linear method for matrix factorization. It utilizes latent user and item vectors, alternating between holding one vector set constant while minimizing the loss function with respect to the other vector set, and then iteratively switching which vector is held constant while repeating the process until convergence. We utilized the ALS function from pyspark.ml.recommendation to implement this lactor factor model with our training set.

Hyperparameters:

The hyperparameters of interest for tuning the basic model were rank and the lambda regularization parameter. Rank specifies the number of latent factors in the user and item matrices and regParam is the level of regularization. Values of [10, 20, 100, 500] and [0.01, 0.1, 1, 10] were tested for rank and regParam, respectively. Additional parameters to ALS include:

- *numBlock*s: number of partition blocks in user and item vectors, left to default of 10. We would have liked to experiment with altering this parameter, however time constraints did not allow for this.
- *maxIter*: maximum iterations, left to default of 10. A low number of iterations helps keep fitting time down, although it introduces optimization error into the model.
- *implicitPrefs*: left to default of False for baseline model which uses only explicit feedback, and set to True for the extension using implicit is_reviewed data.
- *Alpha*: baseline confidence in observations for implicit feedback, unused in baseline model and left to default of 1.0 in hybrid model. Again, we would have liked to experiment with altering this parameter, if time constraints had allowed.
- *Non-negativ*e: whether or not non-negative for least squares optimization are applied, left to default of False.

Evaluation:

Since the output of a deployed system of this sort is a ranked list of recommended books for a given user, we consider three ranking evaluation metrics, all part of the mllib.evaluation.RankingMetrics class:

1. Mean Average Precision: captures proportion of recommended books that appear in the true utility set, averaged across all users M, where order of the recommendations in comparison to their true relevance is considered

$$MAP = \frac{1}{M}\sum_{i=0}^{M-1}\frac{1}{|D_i|}\sum_{j=0}^{Q-1}\frac{rel_{D_i}(R_i(j))}{j+1}$$

2. Normalized Discounted Cumulative Gain (NDCG at k): Also takes order into account, and captures proportion of the top k=500 recommended books that appear in the true utility set, averaged across all users M

$$NDCG(k) = \frac{1}{M}\sum_{i=0}^{M-1}\frac{1}{IDCG(D_i,k)}\sum_{j=0}^{n-1}\frac{rel_{D_i}(R_i(j))}{\ln(j+1)}$$

Where

$$n = \min\left(\max\left(|R_i|,|D_i|\right),k\right)$$

$$IDCG(D,k) = \sum_{j=0}^{\min(|D|,k)-1}\frac{1}{\ln(j+1)}$$

3. Precision (at k): captures proportion of the top k=500 recommended books that appear in the true utility set, averaged across all users M, where order of the recommendations is not considered

$$p(k) = \frac{1}{M}\sum_{i=0}^{M-1}\frac{1}{k}\sum_{j=0}^{\min(|D|,k)-1} rel_{D_i}(R_i(j))$$

We never had to choose between these metrics, as they always agreed on which model was superior.

Results of Hyperparameter Tuning:

Due to memory constraints, we were only able to perform an exhaustive grid search for optimal hyperparameters on 1% of the data. We would not expect the optimal hyperparameters identified here (green) to be the combination that yields the best performance on the entire dataset. In particular, we would anticipate that higher rank values perform better, with the intuition that more latent features can meaningfully be utilized to make good recommendations when there are more interactions to train with.

Table I. Search for optimal Rank and Lambda parameters on 1% of all interactions

Rank	λ	MAP	NGDP at k=500	Precision at k=500
10	0.01	0.00013	0.00328	0.00066
20	0.01	0.00037	0.00818	0.00167
100	0.01	0.00477	0.06175	0.00855
500	**0.01**	**0.01190**	**0.10850**	**0.01449**
10	0.1	0.00005	0.00250	0.00043
20	0.1	0.00023	0.00707	0.00115
100	0.1	0.00052	0.00128	0.00128
500	0.1	0.00055	0.01092	0.00136
10	1	0.00010	0.00077	0.00007
20	1	0.00009	0.00083	0.00009
100	1	0.00009	0.00077	0.00007
500	1	0.00009	0.00081	0.00008
10	10	0.00001	0.00035	0.00004
20	10	0.00002	0.00055	0.00006
100	10	0.00007	0.00070	0.00007
500	10	0.00012	0.00106	0.00011

Results on Full Dataset:

Although we were not able to execute a full hyperparameter tuning loop on the full dataset, we were able to successfully fit a model on 100% of the training set and evaluate it on 100% of the validation set. For this run we used a rank of 10 and a regularization parameter of 0.01. This run achieved a MAP of 6.7988e-07, NDCG at 500 of 2.6867e-05, and precision at 500 of 5.9444e-06.

Results on Test Set:

Though we ideally would have liked to test the optimal rank and lambda combination from Table I (Rank=500, Lambda=0.01), we were unable to carry out evaluation for such a high rank due to heavy burden on the cluster in the days leading up to the project due date. As an alternative, we ran the trained ALS on train+validation with a rank of 10 and lambda of 0.01. Evaluating this model on the 1% downsample held out test data yields a MAP of 0.00297, NGDP at 500 of 0.01193 and Precision at 500 of 0.00191

Methods for Increasing Evaluation Speed:

We experimented with methods to increase evaluation speed on the dumbo cluster, including decreasing the downsample percentage, decreasing the number of recommendations, and decreasing rank. In general, we observed that under the memory and core constraints of this cluster, we were limited to consideration of a higher rank and less data, or a lower rank and more data.

We also experimented with grid search to find the optimal num_partitions when repartitioning/coalescing different dataframes, the rationale being that fewer partitions allow work to be done in larger chunks. The results of the coalesce grid search were largely inconclusive; we found that the results were hard to comparatively evaluate because network factors made more of a difference in run times. Further complicating matters, different downsample sizes are optimized at different num_partitions. We settled on using int (200 * [0.25+downsample fraction]) for num_partitions in all cases. We found that this greatly improved speeds in general, although it shifted the bottleneck from predicting recommendations to modeling. This indicates that we may have used too few partitions for train.

Had there not been time constraints, we would have liked to additionally experiment with column compression techniques, considering that columns like rating and is_reviewed have integrity constraints with low cardinalities (1-5 and 0-1, respectively).

Attempts to allocate additional resources from dumbo were cut short, as we did not want to contribute more than necessary to the overburdening of the shared resource.

Another time-saving method was to cache dataframes/RDDs that would be used more than once, and to save the results of particularly computationally intense transformations to HDFS, for reuse later (e.g. fitted models and recommendations). The latter strategy required careful naming protocol for HDFS files.

4. Hybrid Approach Using Extended Interactions

Concept

For our extension, we chose to use the binary is_reviewed attribute to build a new collaborative filtering model. We then produced a weighted sum of the recommendations of the implicit model with those of the explicit model.

Combining models with weighted sums is a winning strategy because of how flexible and modular it can be. For example, if we wanted to produce recommendations for users who had not rated very many books, we might decide to keep the 0-ratings, and to increase the weight of the implicit model in the hybrid recommendations. We can also continue to layer more and more models on top of this implementation.

Implementation

As discussed above, is_reviewed is a form of implicit feedback, so we set the *implicitPrefs* parameter of ALS to True here. All other inputs to the model were the same, except that the *ratingCol* became is_reviewed (as opposed to rating).

For the purpose of this assignment, we were looking to see whether our hybrid model could improve on the basic recommender system. For consistency's sake, we therefore decided to use the same interactions as we did above in the training, validation, and test splits, and to evaluate on the same truth set as we did above. Note that the implicit model would bring more to the table if it weren't also dropping 0-ratings, or if the "truth" set were based on is_reviewed. The current implementation does not preclude its ability to perform these functions in future extensions.

The problem of how to produce a weighted sum of rankings was non-trivial. Our algorithm is summarized in Figure 2 below. First we used recommendforUserSubset to produce k recommendations for each subset, where each user_id in the validation sets gets 1 row consisting of their user_id and an array of length k consisting of their top

scored (book_id, prediction score) tuples. We exploded the arrays to create dataframes where each user-book pair gets its own row, as in the original interactions dataset. We then split the tuples to recreate a book_id attribute and a prediction score attribute. At this stage we performed a full outer join on the two dataframes, matching on user_id and book_id. Note that all validation user_ids will be present in both sets, but the book_ids will vary based on which ones are recommended by each model. We fill Null values in the score attributes to 0, and then create a new attribute to hold PS_E + isrev_weight*PS_I where PS_E is the predicted score of the user-book pair by the explicit model, and PS_I is the predicted score of the implicit model. Note that these two scores are on different scales, and isrev_weight is treated as a hyperparameter. The last stage of the process is to reduce the long dataframe back to a wide one with ordered arrays of the k book_ids with the highest weighted sums. We accomplished this by using a Window function.

From there, the ranked lists of books are evaluated against the true list of books, as above.

Figure 2. Flowchart of Hybrid Recommender System Combining *Rating* and *Is_Reviewed* Data

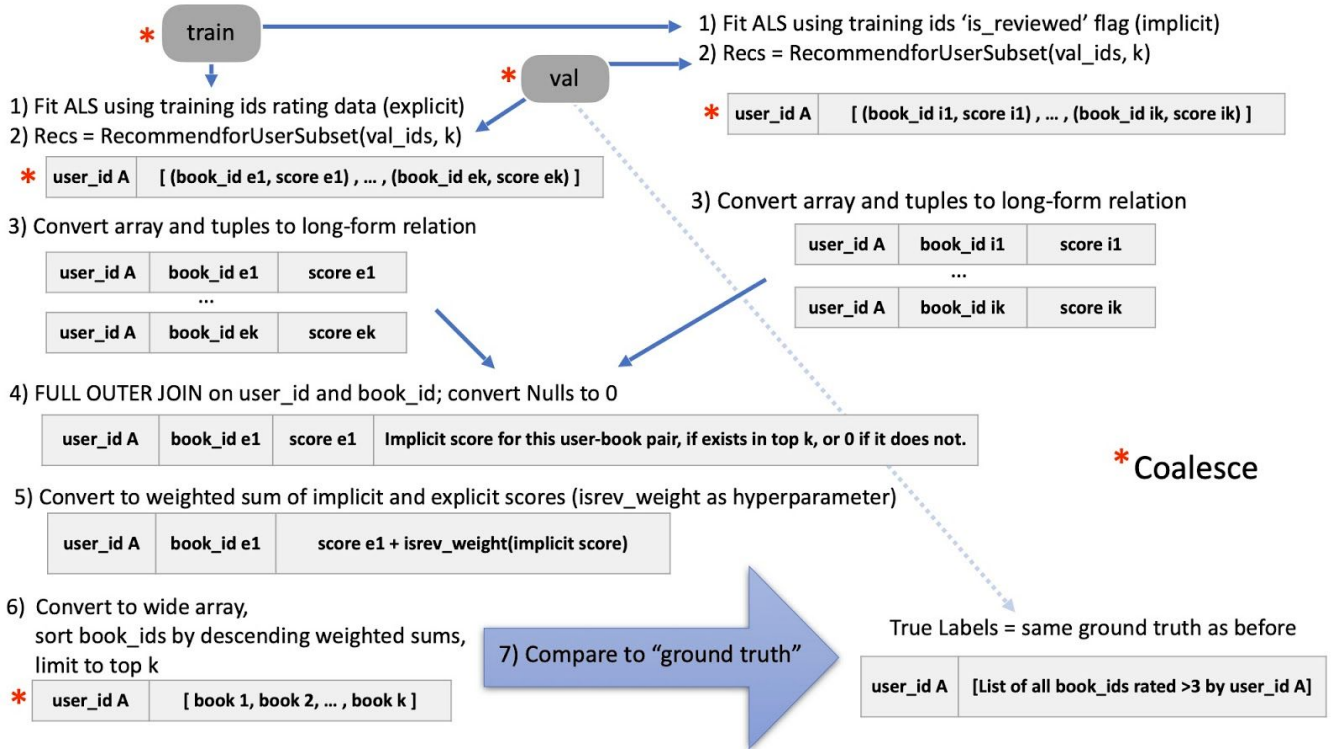


Table II: Search for optimal isrev_weight parameter on 1% of all interactions (fixed rank=10, lambda=0.01)

	isrev_weight				
	-1	**0**	**0.5**	**1**	**5**
MAP	**0.00010**	0.00008	0.00008	0.00008	0.00008
NGDP at k=500	**0.00322**	0.00241	0.00241	0.00241	0.00241
Precision at k=500	**0.00066**	0.00029	0.00029	0.00029	0.00029

We found that the optimal value of isrev_weight was -1, for 1% of the dataset and a fixed rank of 10 and lambda of 0.01. This unexpected result may indicate that the interaction between rating and is_reviewed is more complicated that it appears.

Test Set Results

Table III. Test Set Result Comparison of Hybrid and Baseline Models on %1 of data (lambda= 0.01, rank=10)

	Basic Model isrev_weight= 0	Hybrid Model isrev_weight= -1
MAP	0.00297	0.00113
NGDP at k=500	0.01193	0.01024
Precision at k=500	0.00191	0.00191

Overall, our hybrid implementation performs slightly worse than our basic model, though the two models were not tuned exhaustively for all parameters simultaneously. It's possible that tuning isrev_weight along with the other parameters might produce a superior model.

5. Contributions:

Lauren: data_prep, baseline model, report
Alene: data_prep, baseline model, extension, report

6. References

1. https://www.goodreads.com/about/us
2. https://sites.google.com/eng.ucsd.edu/ucsdbookgraph/home
3. Mengting Wan, Julian McAuley, "Item Recommendation on Monotonic Behavior Chains", RecSys 2018.
4. https://github.com/Samimust/my-yelper/blob/master/My%20Yelper%20Project%20Report.pdf
5. https://blog.insightdatascience.com/explicit-matrix-factorization-als-sgd-and-all-that-jazz-b00e4d9b21ea
6. https://spark.apache.org/docs/1.5.0/mllib-evaluation-metrics.html
7. https://spark.apache.org/docs/2.2.0/ml-collaborative-filtering.html