A Two-Model Reciprocal Rank Fusion Recommender for Amazon Industrial & Scientific Products

Recommender System Challenge (Final Project) - Group 43

Kevin Bretz
Leiden University

Michail Protonotarios
Leiden University

Maftukhatul Q. Virati
Leiden University

Abstract

For the Recommender System Challenge we built a next-item recommender on a 147k-interaction subset of the Amazon Industrial & Scientific catalogue. Our final submission is a Reciprocal Rank Fusion (RRF) of two sequential recommenders, SASRec and BERT4Rec, with the RRF constant set to k=45 and equal weights. On the Kaggle leaderboard we reach a Recall@10 score of 0.02013 using this blend. Originally we tried several heavier alternatives such as graph-based CF and gradient-boosted re-rankers. However, these experiments either did not fit the data well, or improved local validation but got a lower score on the leaderboard. We then focused on tuning only the two sequential models with an RRF blend, which gave us our best results.

Keywords

Recommender systems; SASRec; BERT4Rec; reciprocal rank fusion; sequential recommendation; ablation analysis

1 Introduction

Our main task is to predict, for each of 2,255 users, the next ten items they are most likely to interact with. For training, we use 147,267 chronologically ordered interactions with user, item and timestamp fields. These interactions cover a total of 23,284 users and 13,441 items. The hidden ground-truth set is held by Kaggle and we have a limit of 25 submissions per day to get the best result on the Recall@10 metric.

The pipeline is designed based on two features of the dataset. First, by doing some data exploration we found that the user histories are short as the interactions have a median of 5 and a mean of 6.3 and there are also 17 % of users that only have one or two interactions. This helped us rule out methods that need a long sequence context. Second, the catalogue has a total of 13,441 items, which is pretty small. This means it is still viable to score every item for every user without worrying about retrieval efficiency. The bottleneck in this project will not be how fast we generate the ranking, but the quality of the ranking on the top-200 candidates.

Our final pipeline is small on purpose. First, we train SASRec and BERT4Rec to convergence on the user-item sequences, then merge their ranked lists with weighted Reciprocal Rank Fusion (RRF). Sections 2–4 describe the steps we took to get the winning recipe, Section 5 covers the hyperparameter sweeps, Section 6 explains in detail what we went through in the early experiments and the failure modes that pushed us back to a minimal configuration and lastly, Section 7 reflects on what we would try next.

2 Dataset Preprocessing

The inputs are three CSV files. `train.csv` has user, item and timestamp data. `test.csv` has the same schema but is only used for context. `item_meta.csv` covers 56.6 % of training items with 15 columns. Some examples are `title`, `features`, `main_category`, `categories`, `store` (which we use as brand, more detail in Section 6), `average_rating`, `rating_number` and `price`. We checked that all 2,255 submission users appear in the training set. As a result, cold-user handling was not required and recommendations could be generated without resorting to a global-popularity fallback for unseen users.

For Rule 3 compliance, we never use `test.csv` as a training signal and no model in our pipeline loads a pretrained checkpoint, so both SASRec and BERT4Rec are trained from random initialisation.

We re-index users and items densely by their first appearance in the training data and build a per-user history sorted by timestamp. Less than 1 % of rows are duplicates and are dropped, but we keep the same item appearing at different timestamps because it is useful for sequential models.

Since the winning 2-way RRF only uses the user–item interaction sequences from `train.csv`, missing values in `item_meta.csv` (the 43.4 % of items without metadata coverage, plus individual null fields) never enter the pipeline. The exploratory content-aware variants discussed in Section 6 median-imputed numeric fields and used a fixed "Unknown" bucket for categorical fields.

For the validation split we started with leave-one-out that held out each user's last interaction, but this caused the model to not generalize well. We then switched to a time-based split where the latest 10 % of interactions by global timestamp are held out and used as the validation target set. This was closer to how Kaggle actually splits its test data and gave us a more reliable val-to-leaderboard (LB) transfer on the sequential models.

We also built a content representation per item during exploration by concatenating title, features and description. Next, we reduced it using TF-IDF with the settings of 50k features, unigrams and bigrams, `min_df`=3, `max_df`=0.5 and sublinear TF. We then truncated with SVD to 128 dense components. These features were then fed to several re-ranker variants that we discuss in Section 6. However, our best model, the 2-way RRF with SASRec and BERT4Rec, does not use them.

3 Model Design

In this section, we explain the submitted pipeline, which has two retrievers plus the RRF merge.

3.1 SASRec

SASRec [1] is a causal self-attentive transformer trained next-item-prediction style. Items are embedded into a d=64 space with a learned positional embedding and two transformer blocks (with 2 attention heads and dropout 0.2) produce a contextual representation at every sequence position. The training we used is the standard BPR loss against a uniformly sampled negative per position. We also left-pad and truncate sequences to length 50.

At inference time we encode the user's history, take the dot product of the last position's hidden state with the item embedding table, mask the items they have already seen and return the top 200.

3.2 BERT4Rec

BERT4Rec [2] swaps the causal transformer for a bidirectional one trained on a masked-language-model objective, items are replaced with a `[MASK]` token during training and the model has to recover them via full-softmax cross-entropy over the item vocabulary. We use the same width and depth as SASRec (d=64, 2 blocks, 2 heads, dropout 0.2, max length 50) and a mask probability of 0.2. At inference we append a `[MASK]` token to the user's history and read off the logits at that position.

Because both models share width and depth, the only thing that actually differs is the training objective. SASRec captures "what comes after" while BERT4Rec captures "what fits in this sequence." On short sequences like in our dataset, BERT4Rec is empirically slightly stronger because the mask can land anywhere, not just at the end. The blend benefits from the small disagreement between the two.

3.3 Reciprocal Rank Fusion

Each model gives us, per user, a ranked list of 200 candidates. Reciprocal Rank Fusion merges them by scoring each item as a weighted sum of its inverse-ranked positions:

$$\mathrm{RRF}(u, i) \;=\; \sum_{m} \frac{w_m}{k_{\mathrm{rrf}} + \mathrm{rank}_m(u, i) + 1}, \tag{1}$$

where $\mathrm{rank}_m(u, i)$ is the zero-indexed rank of item i in model m's list. The k_{rrf} constant controls how much the top of each list dominates. A small value lets the top items dominate, while a larger value flattens the contribution across ranks. The standard choice is k_{rrf}=60 [3]. After scoring, we drop items already in the user's training history and return the top 10.

For our final submission we use $w_{\mathrm{sasrec}} = w_{\mathrm{bert4rec}} = 1.0$ and $k_{\mathrm{rrf}} = 45$. We give a more detailed explanation in Section 5 on why we picked $k = 45$.

4 Training Procedure

We train SASRec for 20 epochs at batch size 512 and BERT4Rec for 30 epochs at batch size 512. Both use PyTorch with Adam with settings η=10^{-3}, β=(0.9, 0.98), weight decay 10^{-5} and a cosine-annealing schedule that decays the learning rate from 10^{-3} to 10^{-5} over the run. For the validation runs we fit on `train_sparse_fit` with val targets removed, while for the submission run we fit on the full training data.

SASRec uses one uniformly sampled negative per position with a BPR loss. BERT4Rec uses full softmax over the item vocabulary, which is not best practice, but we only have 13k items so it is doable. We also tried using in-batch InfoNCE for SASRec since it is a popular retrieval objective, but it did not work for us. We explain this in more detail in Section 6.

We use a default seed of 42. From our experience there is some variance between seeds (±0.001 Recall@10), but when we tried using the average from multi-seed, the score on the LB dropped. Based on this, the per-seed differences in ranking might be a useful signal that RRF can exploit, rather than noise to average over.

Validation items are removed from `train_sparse_fit` before we fit either retriever. In the end, the validation Recall@10 we report is a measure of generalization. However, the scores from validation and leaderboard are not proportional. We explore this in more detail in Section 6.

5 Inference Pipeline

At inference, both trained models score every item for every submission user. The per-user candidate lists are saved as 2,255 × 200 arrays in compressed numpy format. This makes the actual RRF blend a sub-second operation, which is convenient when sweeping blend weights and k_{rrf} values.

The exact submission is generated with:

```
python solution.py --stage infer \
    --out submissions/grid_2way_k45.csv \
    --rrf_blend sasrec+bert4rec \
    --k_rrf 45 --rrf_weights 1.0,1.0
```

The submission CSV has three columns (`ID`, `user_id`, `item_id`) where `item_id` is a comma-separated string of ten item ids, in the same row order as `sample_submission.csv`.

If a user ends up with fewer than 10 items after the seen-item filter, we pad with global popularity so every row still has 10 items.

The winning configuration was trained and run inference on an Intel Core i7-11370H (4 cores) CPU. End-to-end the whole pipeline (preprocess, train both models, blend, write CSV) takes about 70 minutes on that machine, with most of the time spent on the BERT4Rec training. The exploratory experiments and the failed re-ranker variants discussed in Section 6 were run on a separate system with an RTX 2070 SUPER plus a Ryzen 5 3600X (6-core), where the same end-to-end pipeline finishes in roughly 8 minutes; the RRF merge itself is sub-second on either machine. We note that re-running the same code from scratch on the RTX system at the default seed produced Recall@10 values in the 0.017–0.018 range rather than the 0.02013 we shipped, even after disabling CUDA and forcing CPU mode. We did not fully isolate the cause, but it appears to be a mix of system-specific variables (the CPU architectures and PyTorch / BLAS versions on the two machines differ). For exact reproducibility we therefore ship the trained candidate blocks alongside the code in the submission archive.

6 Hyperparameter Analysis

For hyperparameter analysis, we swept several groups, the SASRec and BERT4Rec architecture, the RRF k-constant and the blend weights.

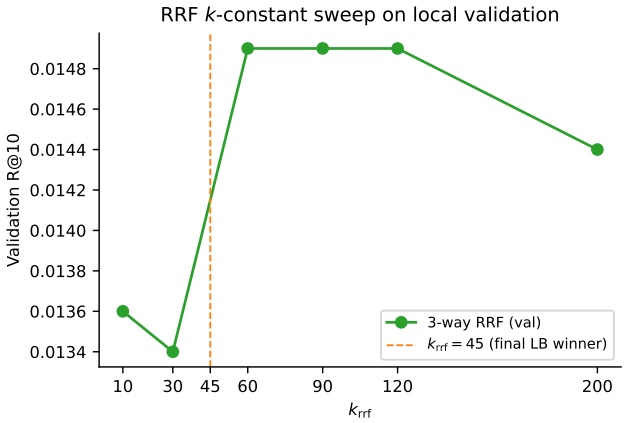

Figure 1: RRF k_{rrf} sweep on local validation (3-way blend). Validation peaks at k=60–120 but the leaderboard winner is k=45. We discuss this val→LB mismatch in Fig. 4.

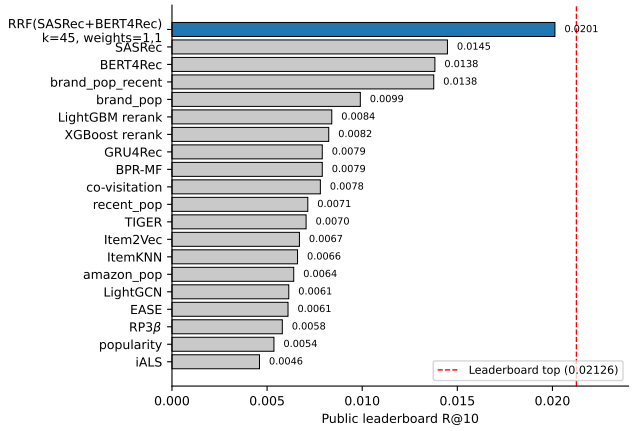

Figure 2: Public leaderboard Recall@10 for every standalone retriever we tried, sorted ascending. SASRec and BERT4Rec lead the standalone bars. The dominant gain is the weighted RRF blend of both at k_{rrf}=45 (orange).

For the sequential models we tried $d_{model} \in \{32, 64, 128\}$, blocks $\in \{1, 2, 4\}$ and training epochs $\in \{10, 20, 40, 80\}$. d=64 with 2 blocks and 20–30 epochs was consistently the best setting. Doubling either the width or the depth produced a clear overfitting collapse on the leaderboard. SASRec at d=128 for 80 epochs scored 0.0061 LB versus 0.0145 for the d=64 baseline. The short-history regime simply does not have enough information per user to support a wider model. Smaller, more regularised models transferred better.

For the RRF k-constant the standard value is k_{rrf}=60. On our local validation $k \in \{60, 90, 120\}$ were essentially tied at R@10= 0.0149 in our 3-way blend with SASRec, BERT4Rec and brand, while smaller values like $\{10, 30\}$ hurt sharply (Fig. 1). On the actual leaderboard, though, we needed a much sharper choice of k_{rrf}=45. Once both sequential models were trained to convergence, the top of their ranked lists had high precision, so a smaller k_{rrf} that does not flatten the rank curve was better. This is where most of our jump from 0.01536 (at k_{rrf}=60) to 0.02013 came from.

For the blend weights, on the 3-way blend the brand weight 1.5 was the LB peak (0.01772), with the fine-grain weights 1.4, 1.45, 1.55, 1.6 all scoring 0.01728. Once the sequential models were retrained to convergence, however, dropping the brand term entirely and using a pure 2-way SASRec+BERT4Rec at equal weights with k_{rrf}=45 beat every 3-way variant. The brand popularity term had been a useful crutch for the under-trained models but became a dilution once both sequence models converged.

We also tried half-lives of $\{14, 30, 60, 90, 180, 365, 730, 1500\}$ days on the `brand_pop_recent` retriever. Our default of 365 days tied with $\{180, 730, 1500\}$ on local validation. The very short half-lives ($\leq$ 90 days) under-weighted the user's actual brand profile. None of them improved the leaderboard once we moved to the 2-way blend.

7 Performance Analysis

7.1 Leaderboard results of all submitted models

Figure 2 aggregates the LB scores for every standalone retriever we submitted to Kaggle. The two strongest single models are SASRec and BERT4Rec at 0.0145 and 0.0138 respectively and the brand-aware popularity retriever `brand_pop_recent` reaches 0.0138 by itself. The pure 2-way RRF of SASRec and BERT4Rec with the right k_{rrf} is about 39 % above either of them alone.

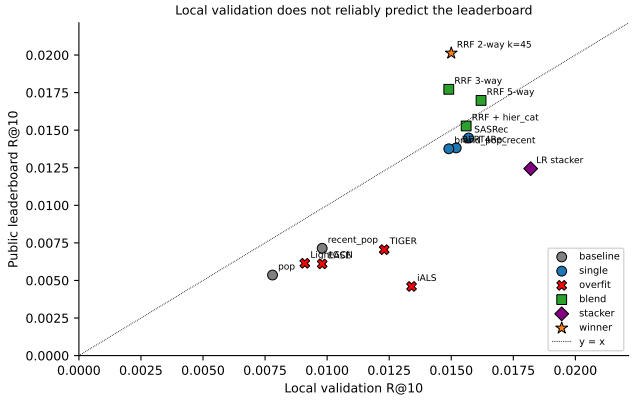

Figure 3: Local val Recall@10 vs public LB Recall@10 for selected submissions. The dotted line is y=x. Linear-CF models and the heavily parameterised re-rankers sit below the line, where the validation is too optimistic. Simple RRF blends sit above the line, with a pessimistic validation score.

7.2 Local validation does not predict the leaderboard

The single most useful thing we learned in this project is that our local validation isn't a reliable predictor of the leaderboard and the direction of the bias depends on the kind of model. Figure 3 plots local validation Recall@10 against public LB Recall@10 for a selection of submissions. iALS, EASE, LightGCN, TIGER and the logistic-regression stacker all live well below the y=x diagonal as

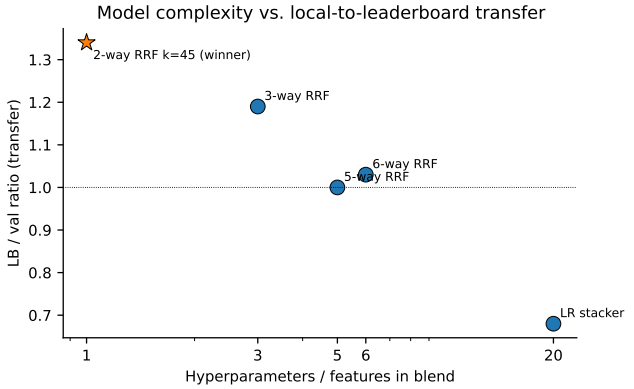

Figure 4: Local-to-leaderboard transfer (LB/val ratio) versus the number of hyperparameters or features the blend contains. The relationship is roughly monotonic, simpler blends transfer better.

their validation scores in the 0.01–0.018 range crashed to 0.005–0.012 on the actual leaderboard. On the other side, the simple 3-way RRF (val 0.0149, LB 0.01772, ratio 1.19) and the final 2-way RRF (val $\approx$ 0.015, LB 0.02013, ratio $\approx$ 1.34) actually sit above the diagonal, validation *understated* how well they would do.

Figure 4 makes the same point more directly as we add hyperparameters or features to a blend, the val→LB transfer ratio drops, from 1.34 at the 2-way RRF (which has one tuned constant) down to 0.68 at the LR stacker (20 features). Our working interpretation is that this is just classical generalisation. Our validation distribution differs from the hidden test in subtle ways. The most likely culprit is the temporal shift, because our val cut is the latest 10 % by timestamp but the test set is even later and the bigger the model, the more sensitive it is to that shift.

In hindsight, this helps explain why the winning recipe is so small. For the first couple of weeks we kept adding things such as popularity variants, content-aware SASRec, brand-aware SASRec, LightGCN, TIGER, LightGBM re-rankers, an LR stacker, MMR diversification, per-segment routing and adaptive blending and almost everything either lifted validation while hurting the leaderboard score, or didn't help either. What finally produced the jump from 0.01772 to 0.02013 was the opposite experimental move, instead of using a 3-way blend, we used only the 2-way, retraining the sequential models to convergence and tuning a single k_{rrf} constant.

7.3 Failure modes

We summarise the cleanest failure modes we ran into.

Linear collaborative filters. We tried multiple linear collaborative filters such as iALS [7], EASE [5], LightGCN [6], RP3β and ItemKNN. Our catalogue is small enough that these methods are tractable, but their parameter counts grow with the catalogue ($n\times n$ for EASE/ItemKNN, $n\times d$ for iALS/LightGCN), which lets them memorise the training interaction matrix. Our spot-checks on EASE and iALS scored above the popularity baseline on local validation but dropped sharply on the leaderboard. The time-based test split exposes that these models do not generalise to future interactions, which matches the broader critique of [9].

TIGER (generative retrieval). We also looked at the generative-retrieval approach of Rajput et al. [4], but it isn't really suited to this dataset. The published method seeds the RQ-VAE codebooks with Sentence-T5 embeddings, which Rule 3 forbids. We tried substituting our own from-scratch TF–IDF+SVD embedding, but the codebook capacity ($256^3 \approx 16$M slots) is three orders of magnitude larger than the 13k items in our catalogue, so most semantic IDs end up empty and the decoder cannot disambiguate items that collide on the same ID. A small-scale test confirmed that this approach collapsed the leaderboard recall.

In-batch InfoNCE SASRec. Swapping BPR for an in-batch contrastive loss with cosine similarity and a temperature is a fairly standard retrieval technique, but it failed here. Training loss converged, but the val Recall@10 stayed essentially at random (0.005). Our best guess is that the temperature interacted badly with our heavily skewed item-popularity distribution and the cosine pre-normalisation throws away the magnitude signal that popularity-aware retrievers rely on.

Gradient-boosted re-rankers (LightGBM LambdaRank [8], XGBoost). These ranked items using popularity, brand match and content cosine features. Val Recall@10 reached 0.014 but the LB stayed near 0.008, below the sequential models alone. The trees ended up over-weighting popularity at training time and lost SASRec's finer-grained sequential signal in the candidate pool.

Logistic-regression stacker. A plain LR on per-retriever inverse-rank features over 10 retrievers reached val 0.0182 (held-out 0.0182, vs 0.0118 for the 3-way baseline on the same split), but it scored only 0.01244 on the leaderboard. That was the most extreme val→LB collapse we ever saw and a clean case of overfitting with 20 features over 4,543 training users with only 1,094 positive labels.

Score-level averaging across seeds. 5-seed averaging of SASRec scored val 0.0151 but LB 0.01318, worse than a single seed at 0.01448. The per-seed variance in the ranking turns out to be informative for the fusion step, not noise to average away.

Metadata-derived popularity variants. Beyond the brand- and category-weighted popularity retrievers we used in earlier blends, we also tried several other heuristics that are based on `item_meta.csv` fields such as Amazon review counts, Best Sellers Rank, release date and price tier. None of them added signal that wasn't already captured by SASRec and BERT4Rec on the raw interaction sequences and all of them hurt the leaderboard when we added them as a third RRF teammate to the converged sequential pair.

8 Reflections

The single biggest decision we made was committing, late in the project, to a fully retrained pair of sequential models plus the simplest possible fusion. Every detour into heavier models resulted in a lower leaderboard score. The simple 2-way rank-fusion recipe is cheap, easy to debug and surprisingly robust. The second-biggest decision was to stop using the leave-one-out validation split in favour of a time-based 90th-percentile cutoff, which gave us a more reliable val-to-LB signal on the sequential models, even though it still under-predicts the 2-way RRF.

Some things we would do differently with more time:

- A held-out 2-fold time-based cross-validation that better mimics the Kaggle temporal shift. Our single time-based split is still just one sample of a distribution.
- Tuning hyperparameters against the leaderboard feedback signal rather than local validation, for example a Bayesian bandit over a small space of $\{k_{\text{rrf}}, \text{epochs}, d_{\text{model}}\}$ with the 25-submission daily quota as the evaluation budget. We did this implicitly in the final week of the project. Making it explicit much earlier would have saved us a lot of time.

The biggest limitation we're aware of is that the 2-way blend doesn't enforce any diversity in the top-10. We can't guarantee that the recommendations span different brands or categories, only that they span the two sequence models' top picks. We did try MMR diversification with λ=0.7 over the top-30 and it cost us about 0.001 on LB. In hindsight a more targeted diversification constraint, especially for the cold-user fallback, might still be worth trying.

References

[1] W.-C. Kang, J. McAuley. Self-attentive sequential recommendation. *ICDM*, 2018.
[2] F. Sun et al. BERT4Rec: sequential recommendation with bidirectional encoder representations from transformer. *CIKM*, 2019.
[3] G.V. Cormack, C.L.A. Clarke, S. Büttcher. Reciprocal rank fusion outperforms Condorcet and individual rank learning methods. *SIGIR*, 2009.
[4] S. Rajput et al. Recommender systems with generative retrieval. *NeurIPS*, 2023.
[5] H. Steck. Embarrassingly shallow autoencoders for sparse data. *The Web Conference (WWW)*, 2019.
[6] X. He et al. LightGCN: simplifying and powering graph convolution network for recommendation. *SIGIR*, 2020.
[7] Y. Hu, Y. Koren, C. Volinsky. Collaborative filtering for implicit feedback datasets. *ICDM*, 2008.
[8] G. Ke et al. LightGBM: A highly efficient gradient boosting decision tree. *NeurIPS*, 2017.
[9] M. Ferrari Dacrema, P. Cremonesi, D. Jannach. Are we really making much progress? A worrying analysis of recent neural recommendation approaches. *RecSys*, 2019.