On the Generalizability and Predictability of Recommender Systems

Duncan McElfresh[*1] **Sujay Khandagale**[*1] **Jonathan Valverde**[*2] **John P. Dickerson**[2]
Colin White[1]

[1]Abacus.AI [2]University of Maryland

Abstract While other areas of machine learning have seen more and more automation, designing a high-performing recommender system still requires a high level of human effort. Furthermore, recent work has shown that modern recommender system algorithms do not always improve over well-tuned baselines. A natural follow-up question is, "how do we choose the right algorithm for a new dataset and performance metric?" In this work, we start by giving the first large-scale study of recommender system approaches by comparing 18 algorithms and 100 sets of hyperparameters across 85 datasets and 315 metrics. We find that the best algorithms and hyperparameters are highly dependent on the dataset and performance metric, however, there are also strong correlations between the performance of each algorithm and various meta-features of the datasets. Motivated by these findings, we create RecZilla, a meta-learning approach to recommender systems that uses a model to predict the best algorithm and hyperparameters for new, unseen datasets. By using far more meta-training data than prior work, RecZilla is able to substantially reduce the level of human involvement when faced with a new recommender system application. We not only release our code and pretrained RecZilla models, but also all of our raw experimental results, so that practitioners can train a RecZilla model for their desired performance metric: `https://github.com/naszilla/reczilla`.

1 Introduction

While some areas of machine learning have benefitted greatly from repurposing existing computation through pretrained models [20, 50, 32, 21, 44], recommender system (rec-sys) research has followed a different trajectory: despite their widespread usage across many e-commerce, social media, and entertainment companies such as Amazon, YouTube, and Netfix [11, 26, 52], there is far less work in reusing models. Many rec-sys techniques are designed and optimized with just a *single* dataset in mind [26, 31, 11, 40, 55]. Intuitively, this might be because each rec-sys application is highly unique based on the dataset and the target metric. For example, a typical user session looks very different among e.g. YouTube, Home Depot, and AirBnB [11, 40, 31]. However, this intuition has not been formally established. Furthermore, recent work has shown that neural recommender system algorithms do not always improve over well-tuned baselines such as k-nearest neighbor and matrix factorization [18]. A natural question is then, "how do we choose the right algorithm for a new dataset and performance metric?"

In this work, we show that the best algorithm and hyperparameters are highly dependent on the dataset and user-defined performance metric. Specifically, we run the first large-scale study of rec-sys approaches by comparing 18 algorithms across 85 datasets and 315 metrics. For each dataset and algorithm pair, we test up to 100 hyperparameters (given a 10 hour time limit per pair). The codebase that we release, which includes a unified API for a large, diverse set of algorithms, datasets, and metrics, may be of independent interest. We show that the algorithms do not *generalize* – the

[*]Equal contribution. This work was extended to a full-length paper here: `https://arxiv.org/abs/2206.11886`.

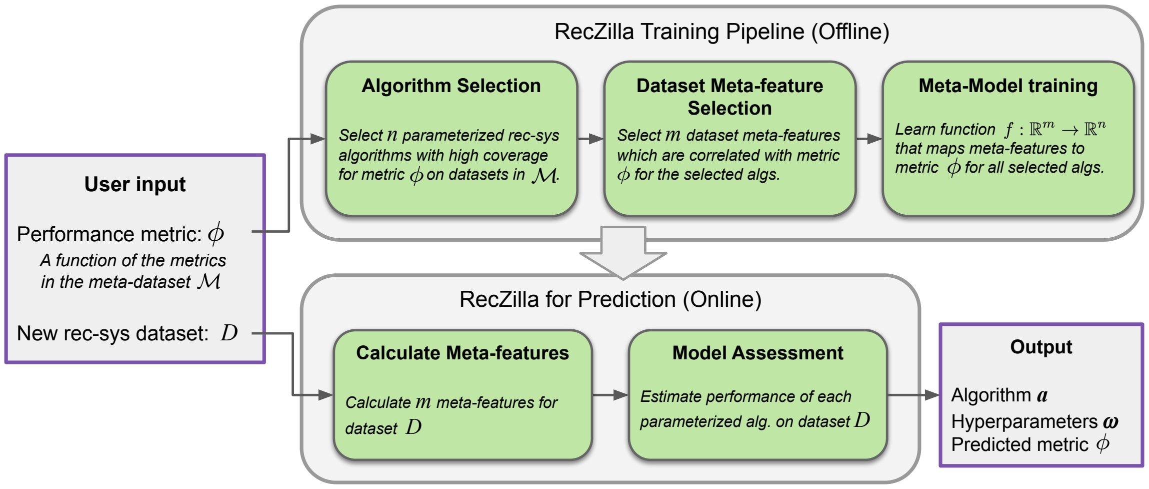
RecZilla Training Pipeline (Offline)
Algorithm Selection
Select n parameterized rec-sys algorithms with high coverage for metric ϕ on datasets in $\mathcal{M}$.
Dataset Meta-feature Selection
Select m dataset meta-features which are correlated with metric ϕ for the selected algs.
Meta-Model training
Learn function $f : \mathbb{R}^m \to \mathbb{R}^n$ that maps meta-features to metric ϕ for all selected algs.
User input
Performance metric: ϕ
A function of the metrics in the meta-dataset $\mathcal{M}$
New rec-sys dataset: D
RecZilla for Prediction (Online)
Calculate Meta-features
Calculate m meta-features for dataset D
Model Assessment
Estimate performance of each parameterized alg. on dataset D
Output
Algorithm $\boldsymbol{a}$
Hyperparameters $\boldsymbol{\omega}$
Predicted metric ϕ

Figure 1: RecZilla recommends a parameterized rec-sys algorithm for a user-provided dataset and performance metric. The RecZilla pipeline is built using a meta-dataset $\mathcal{M}$ that includes many different performance metrics evaluated on many different rec-sys algorihtms on many different datasets; we estimate algorithm performance using dataset meta-features.

performance of algorithms changes substantially across datasets and across performance metrics. Furthermore, the best hyperparameters on one dataset often perform significantly worse than the best hyperparameters on a different dataset. On the other hand, we *do* show that various meta-features of the dataset can be used to *predict* the performance of rec-sys algorithms.

Motivated by these findings, we introduce RecZilla, a meta-learning-based algorithm selection approach (see Figure 1) inspired by SATzilla [57]. At the core of RecZilla is a model that, given a user-defined performance metric, predicts the best rec-sys algorithm and hyperparameters for a new dataset based on meta dataset features such as number of users and items, and spectral properties of the interaction matrix. We show that RecZilla quickly finds high-performing algorithms on datasets it has never seen before. While there has been prior work on meta-learning for recommender systems [16, 17], no prior work is metric-independent, searches for hyperparameters as well as algorithms, or considers more than nine dataset families. By running an ablation study on the number of meta-training datasets, we show that more dataset families are crucial to the success of RecZilla. We release ready-to-use, pretrained RecZilla models for common test metrics, and we release the raw results from our large-scale study, along with code so that practitioners can easily train a new RecZilla model for their specific performance metric of interest.

Our contributions. We summarize our main contributions below.

- We run a large-scale study of recommender systems, showing that the best algorithm and hyperparameters are highly dependent on the dataset and user-defined performance metric. We also show that dataset meta-features are predictive of the performance of algorithms.
- We create RecZilla, an algorithm selection approach which, given a performance metric, efficiently predicts the best algorithm and set of hyperparameters on new datasets.
- We release a public repository containing 85 datasets and 18 rec-sys algorithms, accessed through a unified API. Furthermore, we release both pretrained RecZilla models, and raw data so that users can train a new RecZilla model on their desired metric.

Related work. Recommender systems are a widely studied area of research [8]. Common approaches include k-nearest neighbors [1], matrix factorization [39, 43], and deep learning approaches [11, 26, 52]. For a survey on recommender systems, see [8, 4]. A recent meta-study showed that of the

12 neural rec-sys approaches published at top conferences between 2015 and 2018, 11 performed worse than well-tuned baselines (e.g. nearest neighbor search or linear models) [18].

Algorithm selection for recommender systems was first studied in 2011 [34] by using a graph representation of item ratings. Follow-up work used dataset meta-features to select the best nearest neighbor and matrix factorization algorithms [23, 3, 28]. Subsequent work focused on improving the model and framework [17] including studying 74 meta-features systematically [13]. More recent approaches from 2018 run meta-learning for recommender systems by casting the meta-problem itself as a collaborative filtering problem. Performance is then estimated with subsampling landmarkers [14, 16, 15]. No prior work in algorithm selection for rec-sys includes open-source Python code. There is also work on automated machine learning (AutoML) for recommender systems, without meta-learning [56, 6, 29, 30]. To the best of our knowledge, no meta-learning or AutoML rec-sys papers have run experiments on more than nine dataset families or four test metrics, and no prior work predicts hyperparameters in addition to algorithms.

2 Analysis of Recommender Systems

In this section, we present a large-scale empirical study of rec-sys algorithms across a diverse set of datasets and metrics. We assess the following two research questions.

1. **Generalizability**. If a rec-sys algorithm or set of hyperparameters performs well on one dataset and metric, will it perform well on other datasets or on other metrics?
2. **Predictability**. Given a metric, can various dataset meta-features be used to predict the performance of rec-sys algorithms?

Experimental design. We run 18 rec-sys algorithms, including clustering-based, matrix factorization, linear, and baseline methods. We run these algorithms on 85 datasets from 19 dataset "families": a family refers to an original dataset (such as Movielens), while "dataset" refers to a single train-test split drawn from the original dataset, which may be a small subset of the original. We use 21 different base metrics (such as precision, recall, NDCG) computed at 15 different cutoff values. For full details of the algorithms, datasets, and metrics, see Appendix A.

For each dataset, we compute a train and test split based on leave-last-k-out (and our repository also includes splits based on global timestamp). For eac algorithm, we expose several hyperparameters and give ranges based on common values. For each dataset, we run each algorithm on a random sample of up to 100 hyperparameter sets. Each algorithm is allocated a 10 hour limit for each dataset split; we train and test the algorithm with at most 100 hyperparameter sets on an `n1-highmem-2` CPU, until the time limit is reached. Each algorithm is trained on the train split, and the performance metrics are computed on the test split. By running 18 algorithms, up to 100 hyperparameters, and 85 datasets, this resulted in 84 769 successful experiments, and by computing 315 metrics, our final meta-dataset of results includes more than 26 million evaluations.

Generalizability. Our first observation is that *all* algorithms perform *well* on some datasets, and poorly on others. First we identify the best-performing hyperparameter set for each (algorithm, dataset) pair—to simulate hyperparameter optimization using our meta-dataset. We then rank all algorithms for each dataset, according to several performance metrics. If we focus on a single metric, then most algorithms are ranked first according to this metric on at least one dataset.

Average performance is more varied: some algorithms tend to perform better than others. Table 7 shows the mean, min (best) and max (worst) ranking of all 18 algorithms over all dataset and all accuracy and hit-rate metrics. Nearly all algorithms are ranked first for at least one metric, on at least one dataset; the only exception is Random, which has a minimum rank 2. Many algorithms perform very well on average; interestingly, the three algorithms with the highest average ranking each come from different algorithm families: Item-KNN is a similarity-based metric, SLIM-BPR is based on linear models, and SVD is a matrix factorization method. Furthermore, most algorithms perform very poorly in some cases: the maximum rank is at least 15 (out of 18) for all algorithms.

Predictability. We calculate 383 different meta-features to characterize each dataset. These meta-features include statistics on the rating matrix—including basic statistics, the distribution-based features of Cunha et al. [13], and landmark features [14]—which measure the performance of simple rec-sys algorithms on a subset of the training dataset. Since these meta-features are used for algorithm selection in Section 3, they are calculated using only the training split of each dataset. For more details on the dataset meta-features, see Appendix A.4.

We find that some meta-features are highly correlated with the performance of algorithms. For example, "mean of item rating count distribution" has a correlation of 0.941 with SlopeOne, and "median of item rating count distribution" has a correlation of 0.933 with CoClustering. See Table 8 for more details. This experiment motivates the design of RecZilla in the next section, which trains a model using dataset meta-features to predict the performance of algorithms on new datasets.

In Appendix A, we train three different meta-learner functions (XGBoost, KNN, and linear regression) using our meta-dataset, to predict performance metric PREC@10 for 10 rec-sys algorithms with high average performance. MAE decreases as more dataset families are added, suggesting that it is possible to estimate rec-sys algorithm performance using dataset meta-features.

3 RecZilla: Automated Algorithm Selection

In the previous section, we found that *(1)* the best algorithm and hyperparameters strongly depend on the dataset and user-chosen performance metric, and *(2)* the performance of algorithms can be predicted from dataset meta-features. Points *(1)* and *(2)* naturally motivate an algorithm selection approach to rec-sys powered by meta-learning.

In this section, we present *RecZilla*, which is motivated by a practical challenge: given a performance metric and a new rec-sys dataset, quickly identify an algorithm and hyperparameters that perform well on this dataset. This challenge arises in many settings—e.g., when selecting good baseline algorithms for academic research, or when developing high-performing rec-sys algorithms for a commercial application. We begin with an overview and then formally present our approach.

Overview. As alluded to earlier, *RecZilla* is an algorithm selection approach powered by meta-learning. We use the results from the previous section as the meta-training dataset. Given a user-specified performance metric, we train a meta-model that predicts the performance of each of a set of algorithms and hyperparameters on a dataset, by using meta-features of the dataset. Given a new, unseen dataset, we compute the meta-features of the dataset, and then use the meta-model to predict the performance of each algorithm, returning the best algorithm according to the user-selected performance metric. See Figure 1, and see Appendix B for the full details of RecZilla.

Experimental setup. Focusing on the performance metric PREC@10, we build a meta-dataset $\mathcal{M}$ using all rec-sys datasets, algorithms, and meta-features described in Section 2. All meta-learners are evaluated using leave-one-dataset-out evaluation: we iteratively select each dataset *family* as the meta-test dataset, and run the full RecZilla pipeline using the remaining datasets as the meta-training data. Splitting on dataset families rather than datasets ensures that there is no test data leakage. Then for each dataset D in the test set, we compare the performance metric of the predicted best parameterized algorithm to the performance metric of the ground-truth best algorithm using `%Diff`: the percentage difference of PREC@10 on the predicted best algorithm vs. the ground-truth best algorithm.

Comparisons to existing methods. We compare RecZilla to polynomial SVM with 74 meta-features (the best approach from a 2018 analysis [17]) and CF4CF-META [15], which combines CF4CF [16] with earlier meta-learning approaches. Due to their basis on all prior work in the area, these two methods can be seen as representative of all prior work on algorithm selection for recommender systems. We refer to them by `cunha2018` and `cf4cf-meta`. Note that `cunha2018` has no open-source code, and `cf4c4-meta` only has code in R. Furthermore, in order to give a more fair empirical study, we implement both approaches directly within our codebase. Each model uses the same meta-training datasets, algorithm selection procedure, and base algorithms. Since a main novelty

Table 1: Comparison between RecZilla and two representative prior algorithm selection approaches. We report the mean and standard deviation across 50 trials for 19 test sets, for 950 total trials. The runtime is the average time it takes to output predictions on the meta-test dataset.

Approach	Runtime (sec)	%Diff (↓)	PREC@10 of best pred. (↑)
`cunha2018` [17]	**0.39**	52.9 ± 23.0	0.00813 ± 0.0113
`cf4cf-meta` [15]	6.68	43.5 ± 21.8	0.00808 ± 0.00773
RecZilla	6.69	**35.1± 24.1**	**0.00884± 0.00848**

of RecZilla is predicting hyperparameters as well as algorithms, the other two approaches are only given the algorithms with the default hyperparameters.

We compare RecZilla (with an XGBoost model) to `cunha2018` and `cf4c4-meta`. The algorithms are given all 18 dataset families not in the test set, to use as training data. We run 50 trials for all 19 possible test sets in the leave-one-dataset-out evaluation, for a total of 950 trials. See Table 1. RecZilla outperforms the other two approaches in both %Diff and in terms of the PREC@10 value of the rec-sys algorithm outputted by each meta-learning algorithm.

In Appendix B, we give an ablation study on the number of training meta-datapoints and meta-features used by RecZilla, as well as the meta-model of RecZilla.

4 Conclusions, Limitations, and Broader Impact

In this work, we conducted the first large-scale study of rec-sys approaches: we compared 18 algorithms and 100 sets of hyperparameters across 85 datasets and 315 metrics. We showed that for a given performance metric, the best algorithm and hyperparameters highly depend on the dataset. We also find that various meta-features of the datasets are predictive of algorithmic performance and runtimes. Motivated by these findings, we created RecZilla, the first metric-independent, hyperparameter-aware algorithm selection approach to recommender systems. Through empirical evaluation, we show that given a user-defined metric, RecZilla effectively predicts high-performing algorithms and hyperparameters for new, unseen datasets, substantially reducing the need for human involvement. We release our code and pretrained RecZilla models, as well as raw experimental results so that users can train new RecZilla models on their own test metrics of interest.

Limitations. While our work progresses prior work along several axes, there are still avenues for improvement. First, the meta-learning problem in RecZilla is low-data. Although we added nearly all common rec-sys research datasets into RecZilla, the result is still only 85 meta-datapoints (datasets). While we guarded against over-fitting to the training data in numerous ways, RecZilla can still be improved by more training data. Therefore, as new recommender system datasets are released in the future, our hope is to add them to our API, so that RecZilla continuously improves over time. Furthermore, the magnitude of our evaluation (78 929 rec-sys models trained) leaves our meta-data susceptible to biases based on experiment success/failures. Therefore, RecZilla may have higher uncertainty for the datasets and algorithms that are more likely to fail.

Broader impact. Our work is "meta-research": there is not one specific application that we target, but our work makes it substantially easier for researchers and practitioners to quickly train recommender system models when given a new dataset. On the research side, this is a net positive because researchers can much more easily include baselines, comparisons, and run experiments on large numbers of datasets, all of which lead to more principled empirical comparisons. On the applied side, our day-to-day lives are becoming more and more influenced by recommendations generated from machine learning models, which comes with pros and cons. These recommendations connect users with needed items that they would have had to spend time searching for [36]. Although these recommendations may lead to harmful effects such as echo chambers [24, 37], techniques to identify and mitigate harms are improving [27, 45].

5 Reproducibility Checklist

1. For all authors…

   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] [The main claims in the abstract and introduction reflect the paper's contributions and scope.]

   (b) Did you describe the limitations of your work? [Yes] [See Section 4.]

   (c) Did you discuss any potential negative societal impacts of your work? [Yes] [See Section 4.]

   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes] [We read the ethics review guidelines and ensured our paper conforms to them.]

2. If you are including theoretical results…

   (a) Did you state the full set of assumptions of all theoretical results? [N/A] [We did not include theoretical results.]

   (b) Did you include complete proofs of all theoretical results? [N/A] [We did not include theoretical results.]

3. If you ran experiments…

   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results, including all requirements (e.g., `requirements.txt` with explicit version), an instructive `README` with installation, and execution commands (either in the supplemental material or as a URL)? [Yes] [We include the code, data, and instructions to reproduce the results here: `https://github.com/naszilla/reczilla`.]

   (b) Did you include the raw results of running the given instructions on the given code and data? [Yes] [We include our raw results; see `https://github.com/naszilla/reczilla`.]

   (c) Did you include scripts and commands that can be used to generate the figures and tables in your paper based on the raw results of the code, data, and instructions given? [Yes] [We include scripts to generate our exact results. See the `scripts` folder in `https://github.com/naszilla/reczilla`.]

   (d) Did you ensure sufficient code quality such that your code can be safely executed and the code is properly documented? [Yes] [We included multipe documentation files, and put in a reasonable effort to make our code as easy to use as possible.]

   (e) Did you specify all the training details (e.g., data splits, pre-processing, search spaces, fixed hyperparameter settings, and how they were chosen)? [Yes] [See Sections 2 and 3.]

   (f) Did you ensure that you compared different methods (including your own) exactly on the same benchmarks, including the same datasets, search space, code for training and hyperparameters for that code? [Yes] [Yes, see Section 3.]

   (g) Did you run ablation studies to assess the impact of different components of your approach? [Yes] [We gave an ablation study in Section B.]

   (h) Did you use the same evaluation protocol for the methods being compared? [Yes] [In our ablation, the same evaluation protocol was used.]

   (i) Did you compare performance over time? [Yes] [See Section 3.]

   (j) Did you perform multiple runs of your experiments and report random seeds? [Yes] [We ran 50 trials of our experiments in Section 3.]

(k) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] [All of our experiments have error bars.]

(l) Did you use tabular or surrogate benchmarks for in-depth evaluations? [N/A] [There do not exist tabular or surrogate benchmarks for recommender systems.]

(m) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] [We include this information in Section 3].

(n) Did you report how you tuned hyperparameters, and what time and resources this required (if they were not automatically tuned by your AutoML method, e.g. in a NAS approach; and also hyperparameters of your own method)? [Yes] [We explained hyperparameter tuning in Section 3.]

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...

   (a) If your work uses existing assets, did you cite the creators? [Yes] [See Section A.]

   (b) Did you mention the license of the assets? [N/A] Our experiments were conducted only on publicly available datasets.

   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A] We did not include new assets.

   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A] Our experiments were conducted only on publicly available datasets.

   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] Our experiments were conducted only on publicly available datasets.

5. If you used crowdsourcing or conducted research with human subjects...

   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A] [We did not conduct research with human subjects.]

   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A] [We did not conduct research with human subjects.]

   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A] [We did not conduct research with human subjects.]

Acknowledgments

Dickerson was supported in part by NSF CAREER Award IIS-1846237, NIST MSE Award #20126334, DARPA GARD #HR00112020007, and DoD WHS Award #HQ003420F0035.

References

[1] David Adedayo Adeniyi, Zhaoqiang Wei, and Y Yongquan. Automated web usage data mining and recommendation system using k-nearest neighbor (knn) classification method. *Applied Computing and Informatics*, 12(1):90–108, 2016.

[2] Gediminas Adomavicius and YoungOk Kwon. Improving aggregate recommendation diversity using ranking-based techniques. *IEEE Transactions on Knowledge and Data Engineering*, 24(5):896–911, 2011.

[3] Gediminas Adomavicius and Jingjing Zhang. Impact of data characteristics on recommender systems performance. *ACM Transactions on Management Information Systems (TMIS)*, 3(1):1–17, 2012.

[4] Charu C Aggarwal et al. *Recommender systems*, volume 1. Springer, 2016.

[5] Fabio Aiolli. Efficient top-n recommendation for very large scale binary rated datasets. In *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, page 273–280, New York, NY, USA, 2013. Association for Computing Machinery.

[6] Rohan Anand and Joeran Beel. *Auto-Surprise: An Automated Recommender-System (AutoRecSys) Library with Tree of Parzens Estimator (TPE) Optimization*, page 585–587. Association for Computing Machinery, New York, NY, USA, 2020.

[7] Krisztian Balog, Filip Radlinski, and Shushan Arakelyan. Transparent, scrutable and explainable user models for personalized recommendation. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'19, page 265–274, New York, NY, USA, 2019. Association for Computing Machinery.

[8] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez. Recommender systems survey. *Knowledge-Based Systems*, 46:109–132, 2013.

[9] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[10] Colin Cooper, Sang Hyuk Lee, Tomasz Radzik, and Yiannis Siantos. Random walks in recommender systems: exact computation and simulations. In *Proceedings of the 23rd international conference on world wide web*, pages 811–816, 2014.

[11] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198, 2016.

[12] Paolo Cremonesi, Yehuda Koren, and Roberto Turrin. Performance of recommender algorithms on top-n recommendation tasks. In *Proceedings of the fourth ACM conference on Recommender systems*, pages 39–46, 2010.

[13] Tiago Cunha, Carlos Soares, and André CPLF de Carvalho. Selecting collaborative filtering algorithms using metalearning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 393–409. Springer, 2016.

[14] Tiago Cunha, Carlos Soares, and André CPLF de Carvalho. Recommending collaborative filtering algorithms using subsampling landmarkers. In *International Conference on Discovery Science*, pages 189–203. Springer, 2017.

[15] Tiago Cunha, Carlos Soares, and André CPLF de Carvalho. Cf4cf-meta: Hybrid collaborative filtering algorithm selection framework. In *International Conference on Discovery Science*, pages 114–128. Springer, 2018.

[16] Tiago Cunha, Carlos Soares, and André CPLF de Carvalho. Cf4cf: recommending collaborative filtering algorithms using collaborative filtering. In *Proceedings of the 12th ACM Conference on Recommender Systems*, pages 357–361, 2018.

[17] Tiago Cunha, Carlos Soares, and André CPLF de Carvalho. Metalearning and recommender systems: A literature review and empirical study on the algorithm selection problem for collaborative filtering. *Information Sciences*, 423:128–144, 2018.

[18] Maurizio Ferrari Dacrema, Simone Boglio, Paolo Cremonesi, and Dietmar Jannach. A troubling analysis of reproducibility and progress in recommender systems research. *ACM Transactions on Information Systems (TOIS)*, 39(2):1–49, 2021.

[19] Maurizio Ferrari Dacrema, Paolo Cremonesi, and Dietmar Jannach. Are we really making much progress? A worrying analysis of recent neural recommendation approaches. In Toine Bogers, Alan Said, Peter Brusilovsky, and Domonkos Tikk, editors, *Proceedings of the 13th ACM Conference on Recommender Systems, RecSys 2019, Copenhagen, Denmark, September 16-20, 2019*, pages 101–109. ACM, 2019.

[20] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[22] Lee R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.

[23] Michael Ekstrand and John Riedl. When recommenders fail: predicting recommender failure for algorithm selection and combination. In *Proceedings of the sixth ACM conference on Recommender systems*, pages 233–236, 2012.

[24] Yingqiang Ge, Shuya Zhao, Honglu Zhou, Changhua Pei, Fei Sun, Wenwu Ou, and Yongfeng Zhang. Understanding echo chambers in e-commerce recommender systems. In *Proceedings of the 43rd international ACM SIGIR conference on research and development in information retrieval*, pages 2261–2270, 2020.

[25] Thomas George and Srujana Merugu. A scalable collaborative filtering framework based on co-clustering. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*, pages 4–pp. IEEE, 2005.

[26] Carlos A Gomez-Uribe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)*, 6(4):1–19, 2015.

[27] Pietro Gravino, Bernardo Monechi, and Vittorio Loreto. Towards novelty-driven recommender systems. *Comptes Rendus Physique*, 20(4):371–379, 2019.

[28] Josephine Griffith, Colm O'Riordan, and Humphrey Sorensen. Investigations into user rating information and predictive accuracy in a collaborative filtering domain. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 937–942, 2012.

[29] Garima Gupta and Rahul Katarya. Enpso: An automl technique for generating ensemble recommender system. *Arabian Journal for Science and Engineering*, 46(9):8677–8695, 2021.

[30] Srijan Gupta. Auto-caserec: A novel automated recommender system framework. 2020.

[31] Malay Haldar, Mustafa Abdool, Prashant Ramanathan, Tao Xu, Shulin Yang, Huizhong Duan, Qing Zhang, Nick Barrow-Williams, Bradley C Turnbull, Brendan M Collins, et al. Applying deep learning to airbnb search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1927–1935, 2019.

[32] Kaiming He, Ross Girshick, and Piotr Dollár. Rethinking imagenet pre-training. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4918–4927, 2019.

[33] Yifan Hu, Yehuda Koren, and Chris Volinsky. Collaborative filtering for implicit feedback datasets. In *2008 Eighth IEEE international conference on data mining*, pages 263–272. Ieee, 2008.

[34] Zan Huang and Daniel Dajun Zeng. Why does collaborative filtering work? transaction-based recommendation model validation and selection by analyzing bipartite random graphs. *INFORMS Journal on Computing*, 23(1):138–152, 2011.

[35] Nicolas Hug. Surprise: A python library for recommender systems. *Journal of Open Source Software*, 5(52):2174, 2020.

[36] Dietmar Jannach, Markus Zanker, Alexander Felfernig, and Gerhard Friedrich. *Recommender systems: an introduction.* Cambridge University Press, 2010.

[37] Ray Jiang, Silvia Chiappa, Tor Lattimore, András György, and Pushmeet Kohli. Degenerate feedback loops in recommender systems. In *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society*, pages 383–390, 2019.

[38] 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.

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

[40] Pigi Kouki, Ilias Fountalis, Nikolaos Vasiloglou, Xiquan Cui, Edo Liberty, and Khalifeh Al Jadda. From the lab to production: A case study of session-based recommendations in the home-improvement domain. In *Fourteenth ACM conference on recommender systems*, pages 140–149, 2020.

[41] Daniel Lemire and Anna Maclachlan. Slope one predictors for online rating-based collaborative filtering. In *Proceedings of the 2005 SIAM International Conference on Data Mining*, pages 471–475. SIAM, 2005.

[42] Mark Levy and Kris Jack. Efficient top-n recommendation by linear regression. 2013.

[43] Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. *Advances in neural information processing systems*, 27, 2014.

[44] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[45] Virginia Morini, Laura Pollacci, and Giulio Rossetti. Toward a standard approach for echo chamber detection: Reddit case study. *Applied Sciences*, 11(12):5390, 2021.

[46] Allan H. Murphy. The finley affair: A signal event in the history of forecast verification. *Weather and Forecasting*, 11(1):3 – 20, 1996.

[47] Bibek Paudel, Fabian Christoffel, Chris Newell, and Abraham Bernstein. Updatable, accurate, diverse, and scalable recommendations for interactive applications. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 7(1):1–34, 2016.

[48] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. Bpr: Bayesian personalized ranking from implicit feedback. *arXiv preprint arXiv:1205.2618*, 2012.

[49] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: An open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*, CSCW '94, page 175–186, New York, NY, USA, 1994. Association for Computing Machinery.

[50] Tal Ridnik, Emanuel Ben-Baruch, Asaf Noy, and Lihi Zelnik-Manor. Imagenet-21k pretraining for the masses. *arXiv preprint arXiv:2104.10972*, 2021.

[51] 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.

[52] Brent Smith and Greg Linden. Two decades of recommender systems at amazon. com. *Ieee internet computing*, 21(3):12–18, 2017.

[53] Harald Steck. Embarrassingly shallow autoencoders for sparse data. In *The World Wide Web Conference*, pages 3251–3257, 2019.

[54] Amos Tversky. Features of similarity. *Psychological review*, 84(4):327, 1977.

[55] Fan Wang, Xiaomin Fang, Lihang Liu, Yaxue Chen, Jiucheng Tao, Zhiming Peng, Cihang Jin, and Hao Tian. Sequential evaluation and generation framework for combinatorial recommender system. *arXiv preprint arXiv:1902.00245*, 2019.

[56] Ting-Hsiang Wang, Xia Hu, Haifeng Jin, Qingquan Song, Xiaotian Han, and Zirui Liu. *AutoRec: An Automated Recommender System*, page 582–584. Association for Computing Machinery, New York, NY, USA, 2020.

[57] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, June 2008.

[58] Tao Zhou, Zoltán Kuscsik, Jian-Guo Liu, Matúš Medo, Joseph Rushton Wakeling, and Yi-Cheng Zhang. Solving the apparent diversity-accuracy dilemma of recommender systems. *Proceedings of the National Academy of Sciences*, 107(10):4511–4515, 2010.

A Experiment Details

This appendix outlines the algorithms, datasets, metrics, and hyperparameter selection used in RecZilla, as well as the details of our procedure for generating the meta-dataset. This codebase is publicly available[1], and is written in Python. The RecZilla codebase builds on another public Github repository[2].

A.1 Generating Meta-Datasets

We generate the meta-dataset for RecZilla using 18 rec-sys algorithms and 85 datasets. We use a leave-one-out training/validation split for each dataset: for each user, the last interaction is held out for validation, and all remaining interactions are used for training. Each algorithm-dataset pair is given a 10 hour time limit for training and validation and trains/validates with up to 100 random hyperparameter sets (see Appendix A.5), on a single a "n1-highmem-2" instance on Google Cloud (2 vCPUs, 13GB memory). During validation, we calculate 21 different performance metrics at 15 different cutoffs, for a total of 315 different metrics (see Appendix A.6). Out of all 1 530 dataset-algorithm combinations tested in our experiments, 1 404 of them completed the train/validation procedure with at least one hyperparameter set within the 10 hour time limit. Most failed experiments failed due to invalid hyperparameter values, and some failed due to memory errors.

Algorithm runtime varied substantially across algorithm family and dataset. Table 2 shows runtime statistics over all experiments and all algorithms, for experiments that completed within the 10-hour time limit.

A.2 Rec-sys Algorithms Implemented in RecZilla

Our experiments use 18 rec-sys algorithms. Algorithms with hyperparameters are associated with a hyperparameter space, as well as a set of "default" hyperparameters. Table 3 lists each implemented algorithm, along with its hyperparameter space and default parameters.

In the current implementation, we define several versions of User-KNN and Item-KNN, with one version for each similarity metric. This is done for convenience, since different KNN similarity metrics are associated with different hyperparameters. However, in our experiment results we treat all versions of User-KNN and Item-KNN as the same algorithm.

All algorithms here use the interface from [19] (their codebase is publicly available[3]). All but two of our 18 algorithms use the implementation from this codebase; two algorithms (CoClustering and SlopeOne) use the implementation of Surprise [35], which is also publicly available.[4]

Table 3: Description of all algorithms implemented in RecZilla.

Algorithm Name	Reference/ Description	Hyperparameter Space
CoClustering	Clusters users and items. Uses their average ratings to predict new ratings. [25] [35]	num-control-users : Int(1, 1000) num-control-items : Int(1, 1000)

[1]https://github.com/naszilla/reczilla

[2]https://github.com/MaurizioFD/RecSys2019_DeepLearning_Evaluation

[3]https://github.com/MaurizioFD/RecSys2019_DeepLearning_Evaluation/

[4]http://surpriselib.com/

EASE-R	Linear model designed for sparse data. Simplified version of an autoencoder [53].	l2-norm : [1, 1e7]
GlobalEffects	Rating predictions are based on a global score for each item and each user.	-
iALS	Matrix factorization method. Leverages alternating least squares for optimization and uses regularization [33].	num-factors : Int(1, 200) confidence-scaling : {lin., log.} alpha : [1e-3, 50] epsilon : [1e-3, 10] reg : [1e-5, 1e-2]
ItemKNN-Asymmetric	k-nearest neighbors, item-based. [51, 18] Similarity between items is calculated using the asymmetric cosine similarity [5].	top-K : Int(5, 1000) shrink : Int(0, 1000) alpha : [0, 2]
ItemKNN-Cosine	k-nearest neighbors, item-based. [51, 18] Similarity between items is calculated using the cosine similarity.	top-K : Int(5, 1000) shrink : Int(0, 1000) normalize : Bool feature-weighting : {none, BM25, TF-IDF}
ItemKNN-Dice	k-nearest neighbors, item-based. [51, 18] Similarity between items is calculated using the Sørensen-Dice coefficient. [22]	top-K : Int(5, 1000) shrink : Int(0, 1000) normalize : Bool
ItemKNN-Euclidean	k-nearest neighbors, item-based. [51, 18] Similarity between items is calculated using the euclidean distance (l2 distance).	top-K : Int(5, 1000) shrink : Int(0, 1000) normalize : Bool normalize-avg-row : Bool similarity-from-distance : {lin., log., exp.}
ItemKNN-Jaccard	k-nearest neighbors, item-based. [51, 18] Similarity between items is calculated using the Jaccard index. [46]	Same as ItemKNN-Dice

ItemKNN-Tversky	k-nearest neighbors, item-based. [51, 18] Similarity between items is calculated using the Tversky index. [54]	top-K : Int(5, 1000) shrink : Int(0, 1000) alpha : [0, 2] beta : [0, 2]
MF-AsySVD	Matrix factorization model that replaces user factors with the factors of items rated by that user. Items have multiple corresponding factors. [38]	sgd-mode : {sgd, adagrad, adam} use-bias : Bool num-factors : Int(1, 200) item-reg : [1e-5, 1e-2] user-reg : [1e-5, 1e-2] learning-rate : [1e-4, 1e-1] negative-interactions-quota : [0, 0.5]
MF-BPR	Uses the Bayesian Personalized Ranking loss to learn a matrix factorization model [48].	sgd-mode : {sgd, adagrad, adam} num-factors : Int(1, 200) batch-size : {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024} positive-reg : [1e-5, 1e-2] negative-reg : [1e-5, 1e-2] learning-rate : [1e-4, 1e-1]
MF-FunkSVD	A modified version of the matrix factorization algorithm proposed in a blog post[5] [18].	sgd-mode : {sgd, adagrad, adam} use-bias : Bool batch-size : {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024} num-factors : Int(1, 200) item-reg : [1e-5, 1e-2] user-reg : [1e-5, 1e-2] learning-rate : [1e-4 1e-1] negative-interactions-quota : [0, 0.5]

NMF	Non-negative matrix factorization [12].	num-factors : Int(1, 350)
		solver : {coordinate-descent, multiplicative-update.}
		init-type : {random, nndsvda}
		beta-loss : {frobenius, kullback-leibler}

P3alpha	Computes the relevance between users and items based on random walks in a graph containing both users and items [10].	top-K : Int(5, 1000) alpha : [0, 2] normalize-similarity : Bool
PureSVD	Matrix factorization method based on SVD.	num-factors : Int(1, 200)
Random	Predicts random ratings.	-

[5]https://sifter.org/~simon/journal/20061211.html

RP3beta	Similar to P3alpha, but uses a reweighing scheme to compensate for item popularity [47].	top-K : Int(5, 1000) alpha : [0, 2] beta : [0, 2] normalize-similarity : Bool
SLIM-BPR	Uses a Sparse Linear Method (SLIM) optimized for Bayesian Personalized Ranking (BPR) loss. [7, 18]	top-K : Int(5, 1000) symmetric : Bool sgd-mode : {sgd, adagrad, adam} lambda-i : [1e-5, 1e-2] lambda-j : [1e-5, 1e-2] learning-rate : [1e-4, 1e-1]
SLIMElasticNet	Sparse Linear Method (SLIM) [42, 18]	top-K : Int(5, 1000) symmetric : Bool l1-ratio : [1e-5, 1] alpha : [1e-3, 1e-2]
SlopeOne	Uses linear functions to predict ratings for an item based on those from other items. [41] [35]	-
TopPop	Recommends items based on global popularity regardless of user.	-
UserKNN-Asymmetric	k-nearest neighbors, item-based, using the asymmetric cosine similarity. [49, 18]	Same as ItemKNN-Asymmetric
UserKNN-Cosine	k-nearest neighbors, user-based, using the cosine similarity. [49, 18]	Same as ItemKNN-Cosine
UserKNN-Dice	k-nearest neighbors, user-based, using the Sørensen-Dice coefficient. [49, 18]	Same as ItemKNN-Dice
UserKNN-Euclidean	k-nearest neighbors, user-based, using the euclidean distance. [49, 18]	Same as ItemKNN-Euclidean
UserKNN-Jaccard	k-nearest neighbors, user-based, using the Jaccard index. [49, 18]	Same as ItemKNN-Jaccard
UserKNN-Tversky	k-nearest neighbors, user-based, using the Tversky index. [49, 18]	Same as ItemKNN-Tversky

Table 2: Min, mean, and max runtime for each algorithm, over all experiments, for both training and evaluation. The rightmost column shows the number of experiments collected for each algorithm. These runtime statistics only include experiments that completed within the 10 hour time limit, so they are skewed to be small, and should be interpreted as general trends. GlobalEffects, SlopeOne, Random, and TopPop do not have hyperparameters and therefore completed a maximum of 85 experiments. Furthermore, we tested multiple distance metrics for Item-KNN and User-KNN, resulting in more experiments.

	Training time (seconds)			Evaluation time (seconds)			Num. experiments
	min	mean	max	min	mean	max	size
Alg. family							
CoClustering	0.02	167.45	25766.23	0.05	51.09	11393.36	5106
EASE-R	<0.01	13.85	454.14	0.05	14.99	341.97	4376
GlobalEffects	<0.01	0.43	12.73	0.05	639.58	8191.36	85
iALS	0.69	369.30	24283.54	0.05	10.26	3558.97	3502
Item-KNN	<0.01	100.28	13617.51	0.05	159.03	8192.90	12847
MF-AsySVD	0.03	207.76	28463.53	0.05	40.37	13293.74	4254
MF-BPR	0.02	82.19	9635.26	0.06	69.99	12117.41	5659
MF-FunkSVD	0.02	172.61	15650.11	0.05	50.49	12793.54	4938
NMF	0.01	221.50	20369.45	0.06	87.42	11471.30	2957
P3alpha	<0.01	78.17	6264.47	0.05	62.40	6563.46	5816
SVD	<0.01	3.58	353.12	0.05	99.06	10393.31	6132
RP3beta	<0.01	80.45	7043.17	0.05	61.24	7067.75	5900
Random	<0.01	0.09	2.35	0.06	937.67	16529.64	85
SLIME-lasticNet	0.06	142.95	25731.74	0.05	11.63	1816.69	4706
SLIM-BPR	0.03	82.57	31063.87	0.05	21.89	1962.25	5176
SlopeOne	0.05	17.73	470.27	0.07	45.77	803.25	48
TopPop	<0.01	0.13	3.66	0.05	626.50	7501.84	85
User-KNN	<0.01	73.28	30975.03	0.05	158.07	6327.46	13097

A.3 RecZilla Datasets

The RecZilla codebase implements 88 datasets (3 additional datasets were added after our experiments on 85 datasets), derived from 20 dataset families. All datasets are listed in Table 4.

Table 4: Summary of datasets used to train and evaluate RecZilla.

Dataset Name	# Interactions	# Items	# Users	Density
AmazonAllBeauty	232	6,357	139	2.60E-04
AmazonAllElectronics	235	7,437	124	2.50E-04
AmazonAlternativeRock	328	3,842	120	7.10E-04
AmazonAmazonFashion	331	20,800	253	6.30E-05
AmazonAmazonInstantVideo	75,673	23,965	29,756	1.10E-04
AmazonAppliances	2,252	11,402	1,581	1.20E-04
AmazonAppsforAndroid	840,985	61,275	240,933	5.70E-05
AmazonAppstoreforAndroid	19	152	16	7.80E-03
AmazonArtsCraftsSewing	97,022	112,334	30,712	2.80E-05
AmazonAutomotive	300,532	320,112	100,163	9.40E-06
AmazonBaby	236,392	64,426	71,826	5.10E-05
AmazonBabyProducts	10,481	9,475	5,327	2.10E-04

AmazonBeauty	489,929	249,274	146,995	1.30E-05
AmazonBlues	98	896	25	4.40E-03
AmazonBooks	11,498,997	2,330,066	1,686,577	2.90E-06
AmazonBuyaKindle	6,312	1,858	2,715	1.30E-03
AmazonCDsVinyl	1,705,140	486,360	245,080	1.40E-05
AmazonCellPhonesAccessories	588508	319,678	245,110	7.51E-06
AmazonChristian	1,155	7,512	428	3.60E-04
AmazonClassical	528	2,301	152	1.50E-03
AmazonClothingShoesJewelry	1,615,940	1,136,004	496,837	2.86E-06
AmazonCollectiblesFineArt	1,066	5,705	230	8.12E-04
AmazonComputers	51	4,266	26	4.60E-04
AmazonCountry	151	1,677	47	1.90E-03
AmazonDanceElectronic	686	4,763	211	6.80E-04
AmazonDavis	38	58	28	2.30E-02
AmazonDigitalMusic	238,151	266,414	56,814	1.60E-05
AmazonElectronics	2,302,922	476,002	651,680	7.40E-06
AmazonFolk	236	2,366	38	2.60E-03
AmazonGiftCards	237	345	144	4.80E-03
AmazonGospel	105	1,616	45	1.40E-03
AmazonGroceryGourmetFood	335,994	166,049	86,400	2.30E-05
AmazonHardRockMetal	156	1,063	41	3.60E-03
AmazonHealthPersonalCare	661,968	252,331	205,704	1.30E-05
AmazonHomeImprovement	45	3,855	32	3.60E-04
AmazonHomeKitchen	1,029,164	410,243	327,439	7.70E-06
AmazonIndustrialScientific	16,784	45,383	7,779	4.75E-05
AmazonInternational	608	5,544	193	5.70E-04
AmazonJazz	490	2,917	109	1.50E-03
AmazonKindleStore	1,387,653	430,530	213,192	1.50E-05
AmazonKitchenDining	81	3,658	63	3.50E-04
AmazonLatinMusic	21	613	13	2.60E-03
AmazonLuxuryBeauty	1,564	1,798	717	1.20E-03
AmazonMagazineSubscriptions	1,257	1,422	560	1.58E-03
AmazonMiscellaneous	416	5,262	164	4.80E-04
AmazonMoviesTV	1,894,519	200,941	319,406	3.00E-05
AmazonMP3PlayersAccessories	19	1,657	14	8.19E-04
AmazonMusicalInstruments	92,628	83,046	29,040	3.80E-05
AmazonNewAge	132	1,276	44	2.40E-03
AmazonOfficeProducts	166,878	130,006	59,858	2.10E-05
AmazonOfficeSchoolSupplies	41	3,229	21	6.05E-04
AmazonPatioLawnGarden	134,727	105,984	54,196	2.30E-05
AmazonPetSupplies	291,543	103,288	93,336	3.00E-05
AmazonPop	435	5,622	118	6.60E-04
AmazonPurchaseCircles	17	33	11	4.70E-02
AmazonRapHipHop	32	779	19	2.20E-03
AmazonRB	136	2,253	69	8.70E-04
AmazonRock	519	4,464	97	1.20E-03
AmazonSoftware	29,434	18,187	9,097	1.80E-04
AmazonSportsOutdoors	751,440	478,898	238,090	6.60E-06
AmazonToolsHomeImprovement	751,440 412,401	260,659	132,013	1.20E-05
AmazonToysGames	549,347	327,698	164,590	1.00E-05
AmazonVideoGames	308,086	50,210	84,273	7.30E-05

AmazonWine	215	1,228	84	2.10E-03
Anime	7,669,090	11,200	69,521	9.80E-03
BookCrossing	323,443	340,556	22,568	4.20E-05
CiaoDVD	47,102	16,121	4,743	6.20E-04
Dating	17,088,628	168,791	135,359	7.50E-04
Epinions	592,236	139,738	28,487	1.50E-04
FilmTrust	32,586	2,071	1,336	1.20E-02
Frappe	17,022	4,082	777	5.40E-03
GoogleLocalReviews	4,867,954	3,116,785	818,824	1.90E-06
Gowalla	3,735,522	1,247,095	91,846	3.30E-05
Jester2	1,640,712	140	56,333	2.10E-01
LastFM	89,058	17,632	1,883	2.70E-03
MarketBiasAmazon	32,511	9,560	20,335	1.70E-04
MarketBiasModCloth	40,633	1,020	6,866	5.80E-03
Movielens100K	98,114	1,682	943	6.20E-02
Movielens10M	9,833,849	10,680	69,878	1.30E-02
Movielens1M	986,002	3,882	6,039	4.20E-02
Movielens20M	19,723,277	27,278	138,493	5.20E-03
MovielensHetrec2011	851,372	10,109	2,113	4.00E-02
MovieTweetings	808,662	38,018	31,917	6.70E-04
NetflixPrize	99,521,398	17,770	476,694	1.20E-02
Recipes	819,642	231,637	35,464	1.00E-04
Wikilens	26,316	5,111	275	1.90E-02
YahooMovies	195,947	11,916	7,642	2.15E-03
YahooMusic	77,764,403	98,213	1,647,758	4.81E-04

A.4 Dataset Meta-Features & Meta-Feature Selection

We calculate a total of 383 meta-features for each rec-sys dataset, consisting of a few general meta-features, meta-features describing the distribution of ratings, and performance of landmarkers.

For each dataset, we extract the number of users, number of items, number of ratings, and the ratio of items to users. Furthermore, following the approach outlined in [13], we compute the sparsity of the matrix of interactions, and we systematically obtain a series of meta-features based on different distributions that can be obtained by aggregating the ratings in several ways.

Distribution meta-features. The distribution meta-features are obtained in two steps, as described in [13]. First, we obtain a distribution. We do this in one of seven ways. We either take all of the ratings at once or we aggreggate ratings for either items or users in one of three different ways: sum, count, or mean. For each of these seven distributions, we then compute ten different descriptive statistics: mean, maximum, minimum, standard deviation, median, mode, Gini index, skewness, kurtosis, and entropy. This results in 70 distribution meta-features.

Landmarkers. For landmarkers, we evaluate the performance of several baseline algorithms on a subset of the training set. We first select the subsample. Next, we partition this subsample into two sets: a “sub-training set” and a “sub-validation set”. We train each landmarker on the sub-training set and compute performance metrics on the sub-validation set. We compute the 19 performance metrics described in Section A.6, plus three more algorithm-independent metrics:

- **Items in Evaluation Set**: measures the fraction of items with at least one rating in the evaluation set. This metric is algorithm-independent and only serves to describe the dataset and its split.
- **Users in Evaluation Set**: measures the fraction of users with at least one rating in the evaluation set. This metric is algorithm-independent and only serves to describe the dataset and its split.

- **Item Coverage**: fraction of items that are ranked within the top K for at least one user.

All 22 metrics are evaluated at cutoffs 1 and 5, to create the meta-features.

The subsampling scheme is designed to satisfy several constraints. We limit the number of users to 100 and the number of items to 250. We also need to ensure there are at least 2 items rated per user so that the subsequent data split (holding out one item rating per user) does not result in cold users. Furthermore, we ensure the number of items selected is at least 6 so that we can evaluate the performance metrics with the cutoff of 5.

We start the subsampling process by filtering by the users that have at least 2 ratings. We next filter by the items all those users rated. If this results in less than 6 items, we add a random sample of the remaining (cold) items back to the set to have 6 items in total. Next, if the number of users is larger than 100, we take a random subsample of 100 of them and filter by those users. As before, we filter by the items those users rated, and ensure this results in at least 6 items by adding random items back if needed. If the number of items is still greater than 250, we must build an item subsample that results in at least two ratings per user. For each user, we randomly choose two of the items the user rated, and we take the union of these item choices. If this results in less than 250 items, we take a random sample of the remaining items to make the total number of items equal to 250.

Once this subsample is built, we split the subsample into the sub-training and sub-validation set by leaving 1 random item out for each user. We are then ready to run our landmarkers on those sets.

Our landmarkers consist of TopPop, ItemKNN, UserKNN, and PureSVD. For ItemKNN and UserKNN, we use cosine similarity and set the number of neighbors k to 1 and 5. For PureSVD, we use 1 and 5 for the number of latent factors. This results in a total of 7 landmarkers.

Running all 7 landmarkers and computing all 22 metrics at the 2 different cutoffs results in a total of 308 landmarker meta-features.

A.5 Hyperparameter Sampling

For each algorithm with hyperparameters, we test up to 100 parameter sets, limited by the 10-hour time limit used in our experiments. The first evaluated hyperparameter set is the default hyperparameters [6] The remaining 99 hyperparameters are sampled using the ranges specified in Table 3, using Sobol sampling.

A.6 Evaluation Metrics

Each of the evaluation metrics that we use during validation measures the quality of ranking based on the item ratings. For each user, we generate predicted ratings for all items and rank the items according to the predicted rating (in descending order). We trim these ranked lists at a given cutoff, which we denote by K. We then compute different metrics using these user-wise top K items. Some metrics also consider the set of relevant items within these top K, defined as those for which the user rated the item in the evaluation set.

We use 21 different metrics, and we compute them using cutoffs in

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 40, 50\}.$$

This results in 315 different metric/cutoff combinations total. Only a small subset of these metrics are used in our analysis; however, any user-chosen metric (or combination of metrics) can be used to define a performance function for RecZilla.

The metrics are calculated using the implementation in the public repository[7] that our codebase is built on. Below is a list of metrics calculated during model evaluation:

[6] See `https://github.com/naszilla/reczilla`.

[7] `https://github.com/MaurizioFD/RecSys2019_DeepLearning_Evaluation`

- **Average Popularity**: measures the popularity of the recommended items. The popularity for each item is the frequency with which it was rated in the training set. These popularities are normalized by the largest popularity. Next, for each user, we compute the normalized popularity of each of the items within its top K and take their mean. Finally, we average across all users.
- **Average Reciprocal Hit-Rank (ARHR)**: similar to MRR, except the reciprocal ranks for all relevant items (not just the first) are summed together.
- **Diversity (Gini)**: computes the Gini diversity index [8] of the global distribution of items ranked within the top K across all users. Higher values indicate higher diversity.
- **Diversity (Herfindahl)**: computes the Herfindahl index [2] of the global distribution of items ranked within the top K across all users. Higher values indicate higher diversity.
- **Diversity (Shannon)**: computes the Shannon entropy of the global distribution of items ranked within the top K across all users.
- **F1 Score**: the harmonic mean between precision and recall.
- **Hit Rate**: fraction of users for which at least one relevant item is present within the top K.
- **Item Coverage (Hit)**: fraction of relevant items that are ranked within the top K for at least one user.
- **Mean Average Precision (mAP)**: the mean of the average precision across all users. The average precision for a user is computed as follows: for any position $i \leq K$ occupied by a relevant item, we compute the precision at i. We sum all of these precision values and divide the total by K.
- **Mean Average Precision - Min Den**: similar to mAP, but using a modified version of the average precision. If the number of test items for the user is smaller than K, we divide the sum (in the last step of the average precision computation) by this number instead of by K.
- **Mean Inter-List Diversity**: measures how different the top K lists are for all users, as originally proposed in [58]. For each pair of users, we compute the fraction of items in their top K items that are not present in both lists. Taking the average across all users yields the mean inter-list diversity. The codebase implements a more efficient but equivalent way of computing this metric.
- **Mean Reciprocal Rank (MRR)**: the mean of the reciprocal rank across all users. The reciprocal rank for a user is the reciprocal of the rank of the most highly-ranked relevant item or 0 if there is none.
- **Normalized Discounted Cumulative Gain (NDCG)**: first, the discounted cumulative gain (DCG) of the top K ranking is computed by adding, for all relevant items in the top K, a gain discounted logarithmically in terms of the rank. The NDCG is obtained by dividing the DCG of the ranking by that of an ideal ranking (a ranking ordered by relevance).
- **Novelty**: a metric that rewards recommending items that were not popular in the training set [58]. For each item, we compute the fraction of ratings in the training set that correspond to the item. The novelty contributed by the item is computed by taking the negative logarithm of that fraction and dividing by the total number of items, so that items that were seldom seen in the training set result in high contributions. Now, for any user, we compute the novelty as the sum of these contributions for the top K items. Finally, we average the metric across all users.
- **Precision**: the fraction of items in the top K that are relevant, computed across all users.
- **Precision Recall Min**: similar to precision, but if there are less test items than K, the fraction is computed with respect to the number of test items.
- **Recall**: the fraction of relevant items that were placed within the top K, computed across all users.

[8] https://www.statsdirect.com/help/default.htm#nonparametric_methods/gini.htm

Table 5: Highest absolute correlations between algorithm running time (default hyperparameters) and meta-features.

Abs. Correlation	Algorithm Family	Meta-feature
0.999	MF-FunkSVD	Number of interactions
0.997	MF-BPR	Number of users
0.993	GlobalEffects	Number of interactions
0.990	TopPop	Number of interactions
0.986	ItemKNN	Kurtosis of item rating sum distribution

Table 6: Highest absolute correlations between dataset hardness (negative of maximum PREC@10 achieved over all algorithms) and meta-features.

Abs. Correlation	Pos. or Neg. Corr.	Meta-feature
0.752	Negative	Entropy of ratings
0.668	Negative	Mode of user rating count distribution
0.655	Negative	Landmarker, UserKNN ($k = 5$), Item Coverage @ 1
0.652	Negative	Landmarker, TopPop, Recall @ 5
0.652	Negative	Landmarker, TopPop, Hit Rate @ 5

- **User Coverage**: fraction of users both present in the evaluation set and for which the model is able to generate recommendations. In practice, all the algorithms used were able to generate recommendations for all users, since our splitting procedures did not result in cold users, so this metric was always 1 for our datasets and algorithms.
- **User Coverage (Hit)**: fraction of users both present in the evaluation set and for which the model is able to generate at least one relevant recommendation within the top K items. It is equal to the product of the hit rate and the user coverage. Because the latter was always equal to 1, the user coverage (hit) was the same as the hit rate in our experiments.

A.7 Additional details and experiments from Section 2 (generalizability)

Recall that in Table 8, we showed the meta-features that are most highly correlated with the performance (PREC@10) of each algorithm, using their default parameters. In Table 5, we run the same analysis using "training time" instead of PREC@10 as the metric. We see that for some algorithms, the runtime is very highly correlated with certain meta-features such as "number of interactions".

Next, we compute a simple measure of *dataset hardness*, which we compute as, given a performance metric, the negative of the maximum value achieved for that dataset across all algorithms. For example, if all 18 algorithms do not perform well on the MovieTweetings dataset, then we can expect that the MovieTweetings dataset is "hard". In Table 6, we show the meta-features that are most highly correlated with the *hardness* of each algorithm, where hardness is calculated as -PREC@10. We find that the entropy of the rating matrix is most correlated with dataset hardness.

To illustrate the changes in algorithm performance across datasets, Figure 2 shows the normalized metric values for eight algorithms across 17 dataset splits. Some algorithms tend to perform well (Item-KNN and SLIM-BPR) and others poorly (Random, TopPop), but no algorithm clearly dominates for all metrics and datasets. This is a primary motivation for our meta-learning pipeline descirbed in Section 3: different algorithms perform well for different datasets on different metrics, so it is important to identify appropriate algorithms for each setting.

Table 7: The relative performance of each rec-sys algorithm depends on the dataset and metric. This table shows the mean, min (best) and max (worst) rank achieved by all algorithms over all 85 datasets, over 10 accuracy and hit-rate metrics at all cutoffs tested. This includes metrics NDCG, precision, recall, Prec.-Rec.-Min-density, hit-rate, F1, MAP, MAP-Min-density, ARHR, and MRR.

Rank	Item-KNN	SLIM-BPR	SVD	P3alpha	iALS	User-KNN	NMF	RP3beta	EASE-R	MF-FunkSVD	SLIM-ElasticNet	MF-AsySVD	MF-BPR	TopPop	CoClustering	GlobalEffects	SlopeOne	Random
mean	3.4	5.0	5.7	5.7	6.0	6.2	6.4	6.7	6.7	7.5	7.8	8.2	8.6	9.9	11.8	12.1	13.2	14.2
min	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2
max	17	14	16	17	17	16	15	16	17	16	16	15	16	18	17	18	18	18

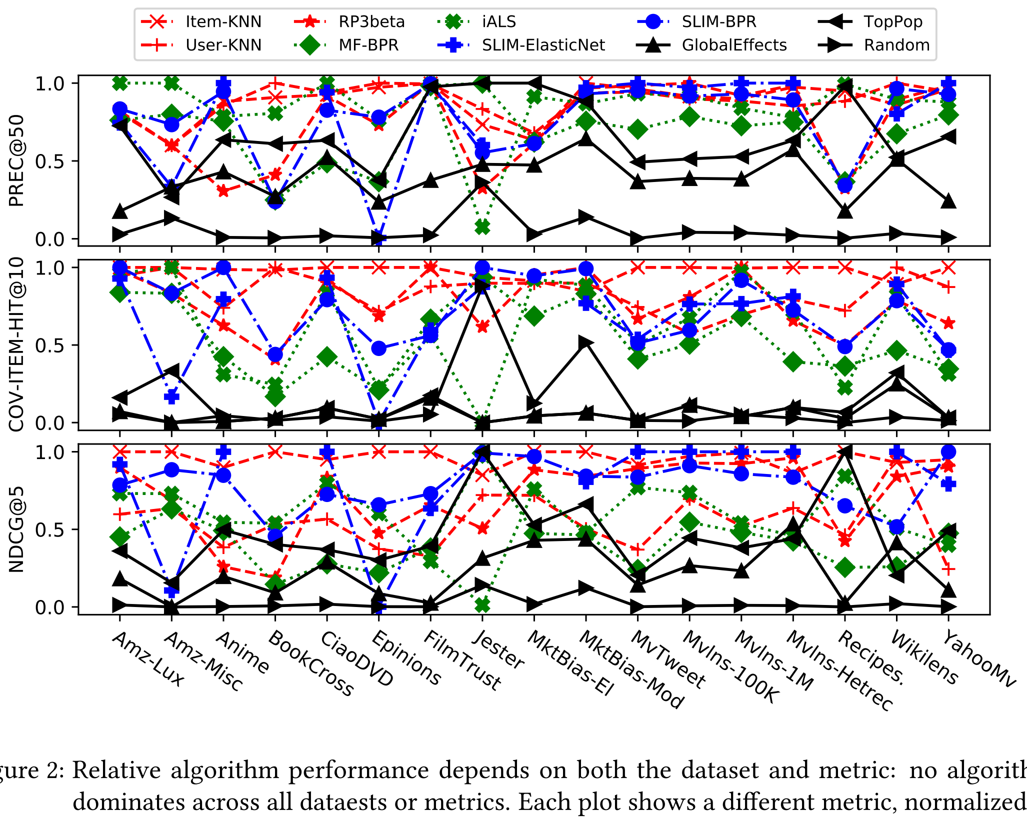

Figure 2: Relative algorithm performance depends on both the dataset and metric: no algorithm dominates across all dataests or metrics. Each plot shows a different metric, normalized to $[0, 1]$ for each dataset; the horizontal axis shows different dataset, ordered alphabetically. Each series corresponds to a different algorithm: similarity-based methods are red, matrix factorization methods are green, and baseline methods are black.

Table 8: Highest absolute correlations computed across 85 datasets and weighed inversely proportional to dataset family frequency, over all pairs of algorithm families and meta-features, for the PREC@10 performance metric and the default algorithm hyperparameters.

Abs. Correlation	Algorithm Family	Meta-feature
0.941	SlopeOne	Mean of item rating count distribution
0.933	CoClustering	Median of item rating count distribution
0.887	MF-BPR	Sparsity of rating matrix
0.855	RP3beta	Mean of item rating count distribution
0.846	UserKNN	Landmarker, Pure SVD, mAP@5

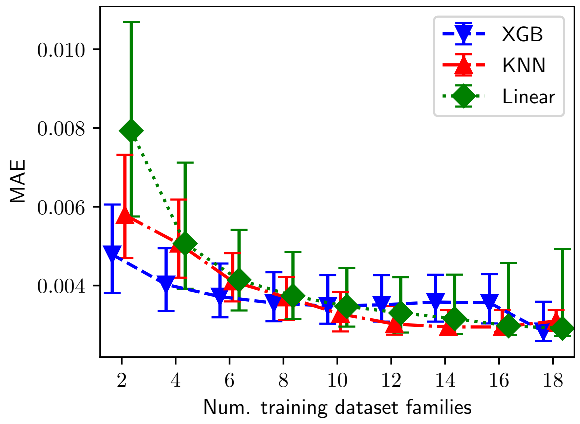

Figure 3: Three basic meta-learners (KNN, linear regression, and XGB) are trained randomly-selected dataset families to predict performance metric PREC@10. As more dataset families are added, the meta-learners are better able to predict rec-sys algorithm performance, suggesting that our dataset meta-features are useful for predicting rec-sys algorithm performance. Vertical axis shows mean absolute error (MAE), over all folds of leave-one-out validation, and 200 random trials; in each trial a different set of training datasets are chosen. Error bars show the 40th and 60th percentile.

Additional details from Section 2 (predictability). As a toy-model version of RecZilla, we train three different meta-learner functions (XGBoost, KNN, and linear regression) using our meta-dataset, to predict performance metric PREC@10 for 10 rec-sys algorithms with high average performance. We use leave-one-out evaluation for each meta-learner: one dataset family is held out for testing, while *m* are used for training. Figure 3 shows the mean absolute error (MAE) of each meta-learner; these results are aggregated over 200 random samples of randomly-selected training dataset families. MAE decreases as more dataset families are added, suggesting that it is possible to estimate rec-sys algorithm performance using dataset meta-features.

We also find that performance metrics are not the only values that can be predicted with dataset meta-features. In particular, we find that the *runtime* of rec-sys algorithms is also highly correlated with different meta-features: the runtimes of MF-FunkSVD, MF-BPR, GlobalEffects, and TopPop all have greater than 0.99 correlation simply with “number of interactions” (see Table 2). Furthermore, we compute a simple measure of *dataset hardness*, which we compute as, given a performance metric, the maximum value achieved for that dataset across all algorithms. For example, if all 18 algorithms do not perform well on the MovieTweetings dataset, then we can expect that the MovieTweetings dataset is “hard”. Once again, we find that certain dataset meta-features are highly correlated with dataset hardness, with “entropy of ratings” having the highest correaltion at 0.752. See Table 6.

The fact that dataset meta-features are correlated with algorithm performance, algorithm runtimes, and dataset hardness is a strong positive signal that meta-learning is worthwhile and useful in the context of recommender systems. We explore this direction further in the next section.

Generalizability of hyperparameters.. While the previous section assessed the generalizability of pairs of (algorithm, hyperparameters), now we assess the generalizability of the hyperparameters themselves while keeping the algorithms fixed. For a given rec-sys algorithm, we can tune it on a dataset i, and then evaluate the normalized performance of the tuned method on a dataset j, compared to the normalized performance of the best hyperparameters from dataset j. In other words, we compute the performance of tuning a method on one dataset and deploying it on another.

In Figure 4, we run this experiment for all pairs of datasets (one dataset per dataset family). We plot the hyperparameter transfer for three different algorithms, as well as the average over all algorithms which completed sufficiently many experiments across the set of hyperparameters. For each given i, j, we create the set of hyperparameters that completed for the given algorithm on both datasets i and j, and then we use min-max scaling for the performance metric values of these hyperparameters on i and on j separately. Therefore, all matrix values are between 0 and 1; a value close to 1 indicates that the best hyperparameters from dataset i are also nearly the best on dataset j. A value close to 0 indicates that the best hyperparameters from dataset i are nearly the worst for dataset j. Across all algorithms, the majority of pairs of datasets do not have strong hyperparameter transfer, and it is particularly hard for some datasets such as Gowalla and Jester2. Overall, these experiments give evidence that tuning the hyperparameters of an algorithm on one dataset and transferring to another dataset does not give high performance, motivating our RecZilla approach which predicts the best hyperparameters for a given algorithm and dataset.

B RecZilla Meta-Learning Pipeline

In this section, we give more details of the RecZilla pipeline, and we give an additional experiment in which we compare RecZilla to other existing rec-sys meta-learning approaches.

The RecZilla pipeline consists of the following components: initial algorithm selection, meta-feature selection, and finally the meta-learner for algorithm selection. The purpose of the first two components is to reduce the dimensionality of the dataset and reduce the risk of overfitting for the classifier. We describe each of these components in the following sections. In all that follows, we assume that the user provides (a) a performance metric function ϕ, (b) the number of parameterized algorithms to be considered by the meta-learner n, and (c) the number of dataset meta-features to be considered by the meta-learner m. In addition, we assume access to a meta-dataset $\mathcal{M}$, such as the one described (and already pre-computed) in this paper.

B.1 Initial algorithm selection

We first select n parameterized algorithms which have high *coverage* over all datasets in meta-dataset $\mathcal{M}$. Since data is relatively scarce in this meta-learning task, we select a subset of n algorithms to reduce the dimensionality of the meta-learner prediction target.

B.2 Meta-feature selection

Since our meta-dataset $\mathcal{M}$ includes hundreds of features, we restrict our meta-learner to m features to avoid over-fitting. This is the same approach taken by prior work [17]. It is computationally infeasible to find the set of $m = 10$ best meta-features out of a set of 383, since we would need to check $\binom{383}{10} \approx 10^{19}$ combinations of meta-features. Instead, we iteratively grow a set of m meta-features which are highly correlated with the user-specified performance metric, without selecting redundant features, by using a "greedy" approach (similar in spirit to prior work [17]). Here we require that the (a) performance metric function ϕ is chosen ahead of time, and (b) a set of n parameterized algorithms have been selected. We introduce some additional notation for this

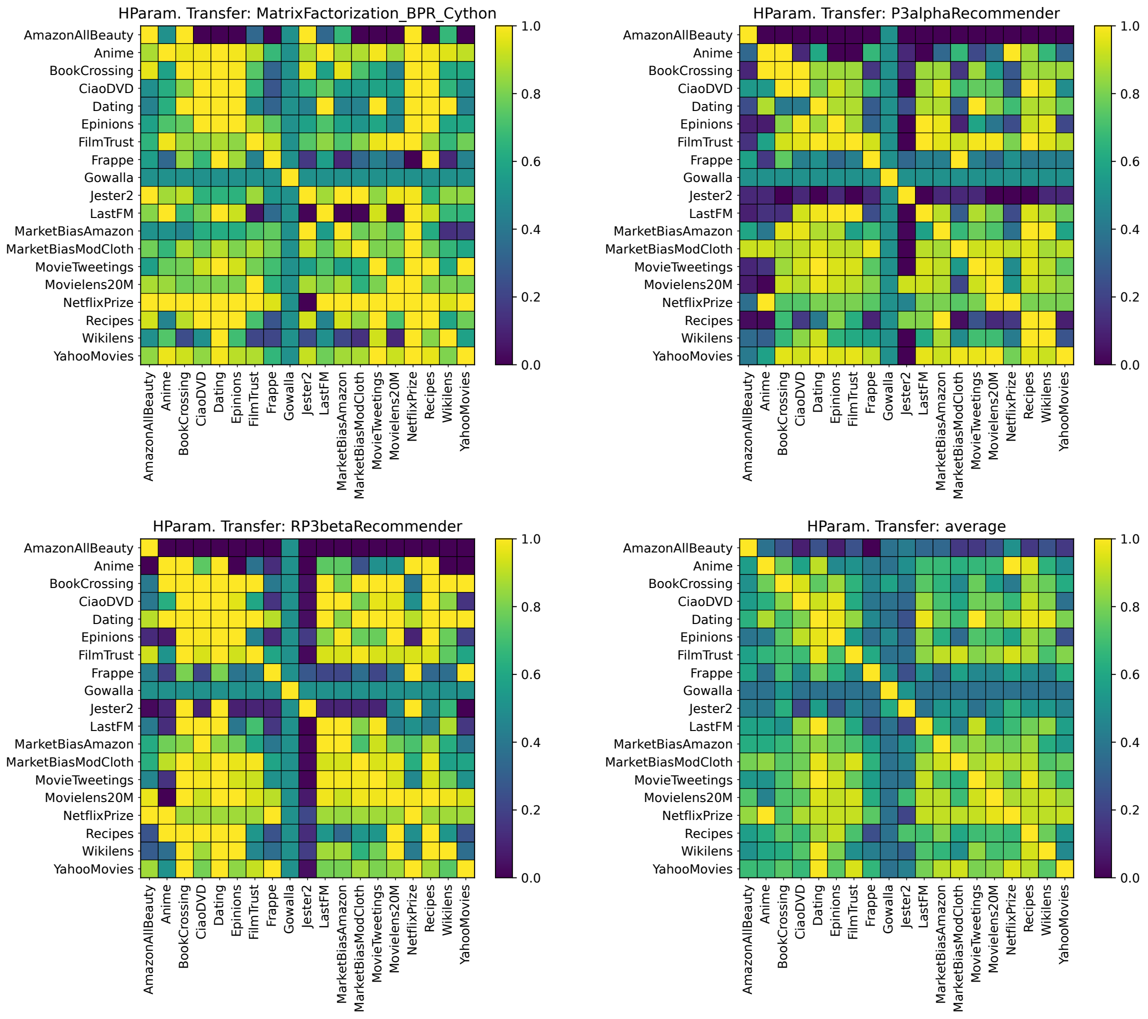

Figure 4: Transferability of hyperparameters across datasets, for three different algorithms, and the average of all algorithms (bottom right). For each plot, row i, column j denotes the relative performance of an algorithm tuned on dataset i and then evaluated on dataset j. A value close to 1 indicates that the hyperparameters transfer well from i to j, while a value close to 0 indicates that the hyperparameters transfer poorly.

section to describe our meta-feature selection process. Let $\boldsymbol{y}^i \in \mathbb{R}^{|\mathcal{D}|}$ be the vector of performance metric ϕ for parameterized algorithm $i \in 1, \ldots, n$, for all datasets in $\mathcal{D}$. Let $\boldsymbol{d}^j \in \mathbb{R}^{|\mathcal{D}|}$ be the vector of meta-feature j for all datasets in $\mathcal{D}$, and let J be the total number of meta-features. Let F denote a set of feature indices that corresponds to selected features.

For each (algorithm, meta-feature pair), we first compute the absolute value of the Pearson correlation between the meta-feature and the performance of each parameterized algorithm i, across all datasets: $c_{ij} \leftarrow |\text{corr}(\boldsymbol{y}^i, \boldsymbol{d}^j)|$ for all $i \in \{1, \ldots, n\} = [n]$ and $j \in \{1, \ldots, J\} = [J]$. When computing this correlation, each sample is weighed inverse-proportionally to the size of the dataset family it corresponds to—to prevent large dataset families (such as Amazon) from dominating the correlation computation.

We select the first feature by finding the largest absolute correlation coefficient between any of the meta-features and parameterized algorithms, and we choose the meta-feature corresponding to it:

$$j_0 \leftarrow \arg\max_{j \in [J]} \left(\max_{i \in [n]} c_{ij} \right).$$

All remaining $(m-1)$ meta-features are selected such that we maximize the *improvement* in the absolute correlation between the selected features and the selected algorithms' performance. This way, we avoid selecting highly correlated features. Algorithm 1 gives a pseudocode description of this feature-selection process.

Algorithm 1 RecZilla Meta-feature selection

```
Require: $\boldsymbol{d}^j \in \mathbb{R}^{|\mathcal{D}|}, \forall j \in [J]$        ▷ $J$ vectors of meta-features for each dataset
Require: $\boldsymbol{y}^i \in \mathbb{R}^{|\mathcal{D}|}, \forall i \in [n]$        ▷ $n$ vectors of performance metrics for each algorithm
Require: $m > 0$        ▷ number of features to select
  $F \leftarrow \{\}$        ▷ indices of selected features
  $x_i = 0, \forall i \in [n]$        ▷ max abs. correlation between any selected meta-feature and $\boldsymbol{y}^i$
  $c_{ij} \leftarrow |\mathrm{corr}(\boldsymbol{y}^i, \boldsymbol{d}^j)|, \forall i \in [n], j \in [J]$
  while $|F| < m$ do
      $j' \leftarrow \arg\max_{j \in [J]} \left[ \max_{i \in [n]} (c_{ij} - x_i) \right]$
      $F \leftarrow F \cup \{j'\}$
      $x_i \leftarrow \max\{x_i, c_{ij'}\}, \forall i \in [n]$        ▷ update the max. abs. correlation for each alg.
  end while
  return $F$
```

B.3 Metalearner for algorithm selection

The goal of the metalearner is to predict the performance of all n selected parameterized algorithms on a new dataset. The input to this meta-learner is the set of m meta-features selected in the previous selection, and the output is an n-dimensional vector of performance metrics for all selected algorithms. We treat this as a multi output regression problem, and our experiments test three different models: a RegressiorChain with XGBoost as the base model, KNN with $k = 5$ and L_2 distance, and multinomial linear regression. For training these meta-learner models, we used the squared error cost function.

To train the meta-learner we construct a final meta-dataset consisting of one tuple $(\boldsymbol{d}, \boldsymbol{y})$ for each dataset represented in $\mathcal{M}^{train}$, where $\boldsymbol{d}$ is a vector of m meta-features for the dataset, and $\boldsymbol{y}$ is a vector of the performance of n parameterized rec-sys algorithms in the set of selected parameterized algorithms $\mathcal{S}'$.

B.4 Ablation study.

We vary both the number of training meta-datapoints and meta-features used by RecZilla; the datapoints and features are randomly selected over 50 random trials. We also compare four different meta-learning functions within RecZilla: XGBoost [9], linear regression, k-nearest neighbors, and uniform random. For KNN, we set $k = 5$ and use the L_2 distance from the selected meta-features.

Figure 5 (left) shows `%Diff` vs. the size of the meta-training set, and Figure 5 (right) shows the results of an ablation study on the number of selected meta-features m; all results are aggregated over all leave-one-out folds and 50 random trials. Generally, XGBoost and KNN outperform the linear model and the random baseline, with XGBoost achieving top performance when using 10 meta-features and the maximum number of training datasets. Furthermore, the number of datasets in the meta-training set matters more than the meta-learning model itself. For example, the improvement of XGBoost from 4 to 10 and to 18 training datasets is larger than the difference

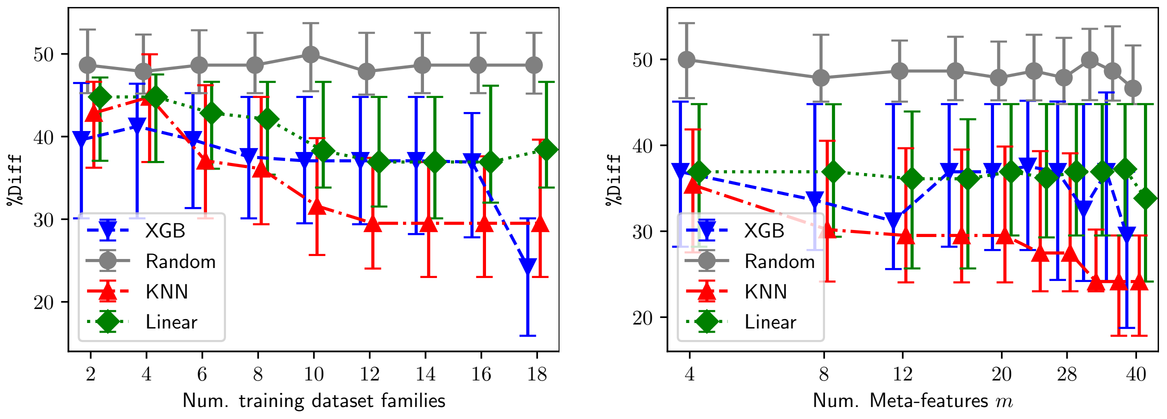

Figure 5: Performance of the RecZilla pipeline improves as we add more training meta-datapoints, and more meta-features m. Subsets of training meta-datapoints and meta-features are selected randomly, over 50 random trials. Points show the median `%Diff`, and error bars show the 40th and 60th percentile over all folds and random trials. (Left) $m = 10$ meta-features, while the number of training dataset families varies. (Right) All training data is used, while m varies.

in performance between XGBoost and KNN at 4 and 10 training datasets, respectively. Finally, we find that the optimal number of meta-features for XGBoost and KNN peaks between 10 and 40.

Pre-trained RecZilla models. We release pre-trained RecZilla models for PREC@10, NDCG@10, and Item-hit Coverage@10, trained with XGBoost on all 85 datasets, with algorithms $n = 10$ and meta-features $m = 10$. We also include a RecZilla model that predicts the Pareto-front of PREC@10 and model training time, so that users can select their desired trade-off between performance and runtime. Finally, we include a pipeline so that users can choose a metric from the list of 315 (or any desired combination of the 315 metrics) and train the resulting RecZilla model.