CSCI 5123 Replication Assignment

Anirudh Ganesh

1 Abstract

Replication is extremely important in the field of science. Results in a vaccumm are valuable, but become much more valuable when they are corroborated and confirmed. With that in mind, in this paper I both replicate and extend a recent paper in the field of recipe recommendation. “HUMMUS: A Linked, Healthiness-Aware, User-centered and Argument-Enabling Recipe Data Set for Recommendation" poses a new dataset with consolidated recipes, reviews, and features, that can be used to recommend nutritious recipes. After constructing the dataset, researchers ran several experiments, training and evaluating models from 3 different libraries. In this paper, I replicate the dataset and experiments, recreating graphs, tables, and key metrics to confirm the original paper’s results. Additionally, I extend the original work by adding two new metrics, tuning model hyperparameters, and introducing 5 new models.

2 Original Paper[1]

The original paper, entitled "HUMMUS: A Linked, Healthiness-Aware, User-centered and Argument-Enabling Recipe Data Set for Recommendation", was written by Felix Bölz, Diana Nurbakova, Sylvie Calabretto, Lionel Brunie, Armin Gerl, and Harald Kosch, submitted on September 29, 2023. The paper attempts to address the problems of world-wide obesity by designing a dataset that can be used by Recommender Systems to recommend healthy, nutritious options. Currently, there are million people suffering from health complications related to obesity, an alarming figure that illustrates the importance of the problem.

Current datasets used for food recommendation suffer from a lack of features related to *health awareness* and a lack of meaningful user-item interactions. Furthermore, most datasets and codebases are often not publicly available, or contain very little data, making healthy food recommendastion a challenging task. To combat this, the researchers created and published a linked, healthiness-aware, and argument-enabling data set called *HUMMUS*, which stands for Health-aware User-centered recoMMendation and argUment enabling data Set). The dataset was constructed by crawling the web and scraping information from Food-related repositories like the *Food.com* website and the *FoodKG* knowledge graph. Along with the existing features, like text reviews, ratings, and tags, researchers also added additional features that accurately capture the nutrition value of recipes.

The final, published, *HUMMUS* dataset improves on existing datasets in several key ways. For one, the data is more recent, and thus provides more data. Moreover, the data includes more features, and provides a direct mapping to a subgraph of the *FoodKG* that enables semantic reasoning. There are a few drawbacks to the dataset, the main one being sparsity. The dataset’s sparsity is near the 99% mark, much higher than the *MovieLens100k* set, which has a sparsity of 93.69%

After the dataset was constructed, researchers ran several experiments using popular models. Notably, these models were **not** tuned in any way, and simply used out of the box, resulting in relatively poor performance. The point of the experiments was not to construct models that use the dataset optimally, but rather to show some of the possibilities of the dataset. Researchers provide several possible extensions, some of which will be implemented in this paper.

3 Replication

Replicating the results from this paper can be split up into two main parts. First, the dataset must be constructed, and the figures associated with the dataset (including density of nutrient values in the dataset, heatmap of a subset of features in the dataset, healthiness score distribution) must be replicated to match the figures in the original paper. Next, the models shown in the original paper must be trained, tested, and evaluated, to construct the results table seen in the original paper.

Fortunately, the underlying code behind the paper was published by the researchers on GitLab. Replication may seem like a trivial task because of this, but in reality I encountered several problems that I will describe in this section.

3.1 Crawling

The first step to replicating this paper was using the code in the "crawling" folder to get the raw data used to construct the *HUMMUS* dataset. There is no ipynb file provided (as there was for the experiment/dataset modification section), so I had to run the raw files using the terminal. I didn’t have any problems using the crawler code to get the raw data, and since the GitLab already featured the raw data, I had a comparison point I could use to verify the crawling code worked.

3.2 Dataset Construction + Exploration

The code used for this section is found in the dataset_scripts file, which is in the .ipynb (python notebook) format. I ran the notebook directly in VSCode, which works similar to using Google Colab.

To load the dataset, I had to unzip the raw CSV files stored in data/food_kg into the hummus_data folder. The core line of code used to read the data is:

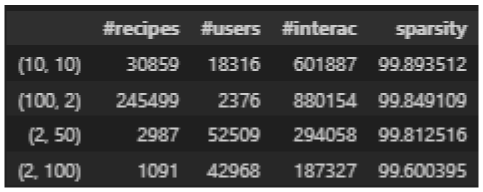
```
utils.load_and_clean_data(data_location,
          additional_location, k_user=10, k_recipe=10
          add_recipe_columns=['food_kg_locator'],
          authorship_relations=6, recipe_tags=True,
               debug=True)
```

Two files are used to create the data – one that features much of the core information about the recipes (recipes.csv), and one that contains tags (RAW_recipes.csv). The key parameters to note are k_user and k_recipe, which define how many users and recipes we want to measure. This greatly affects the number of recipes, users, user-item interactions and sparsity. In order to create the sparsity table, I had to load the data with a variety of different parameters, but notably, **all experiments use (10,10)** for (k_user, k_recipe), so for the figures and experiments I used these parameters as well.

Sparsity Table: The "sparsity table" seen in the original paper depicts how different values of *k* affect sparsity. Unfortunately, I ran into a memory issue when trying to run the code. Specifically, (1, 1), (2, 2), (50, 2) all yielded the following error message – "Unable to allocate 1.12 TiB for an array with shape (X, X) and data type float64". Since I can only use my laptop to run code, I am not sure how to overcome this limitation. Fortunately, this table is not critical to the rest of the code, since (10,10) is the value chosen for the experiments, and the interactions matrix spawned from this value **does** fit into memory.

k	#recipes	#users	#interac.	sparsity [%]
(1, 1)	507 335	302 412	1 916 424	99.99875
(2, 2)	270 284	83 881	1 476 615	99.99348
(10, 10)	30 859	18 316	601 887	99.89351
(50, 2)	254 741	4 614	1 015 399	99.91361
(100, 2)	245 499	2 376	880 154	99.84910
(2, 50)	2 987	52 509	294 058	99.81251
(2, 100)	1 091	42 968	187 327	99.60039

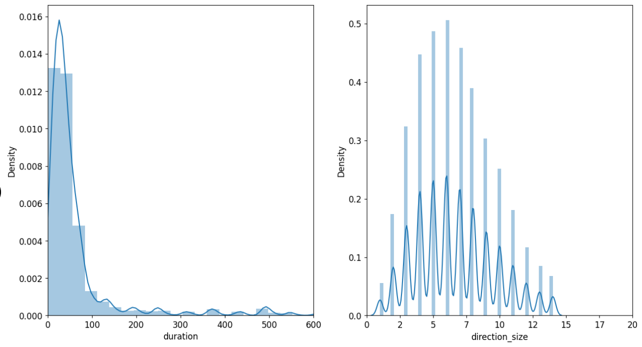
Original Sparsity Table



	#recipes	#users	#interac	sparsity
(10, 10)	30859	18316	601887	99.893512
(100, 2)	245499	2376	880154	99.849109
(2, 50)	2987	52509	294058	99.812516
(2, 100)	1091	42968	187327	99.600395

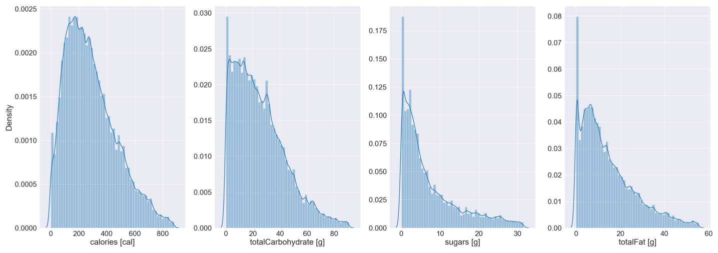
Replicated Sparsity Table

Duration and Direction Size vs. Density: I was able to recreate this figure with the code provided. This figure is not in the original paper, but is in the codebase:

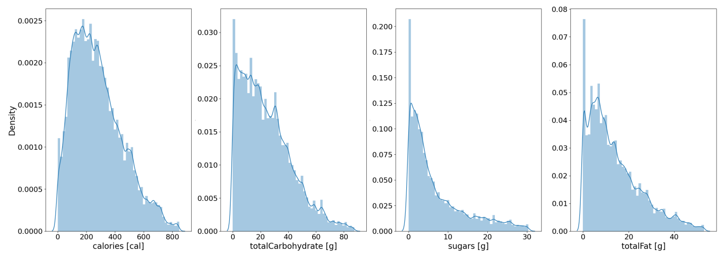


Replicated Graph

Density of nutrient values in dataset: I was able to recreate this figure with the code provided.

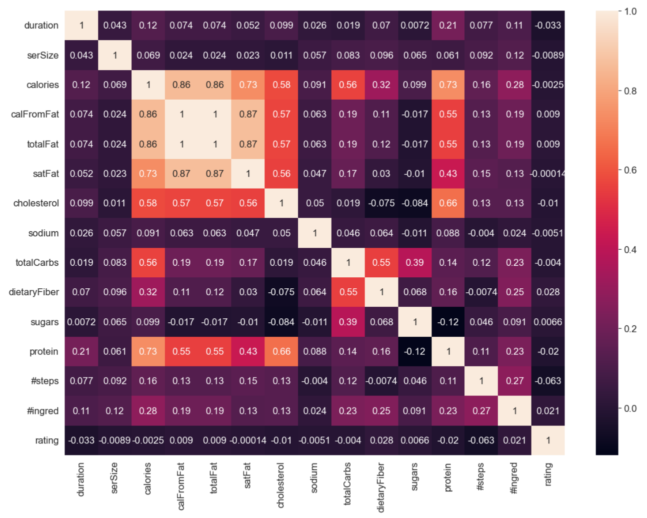


Original Graph

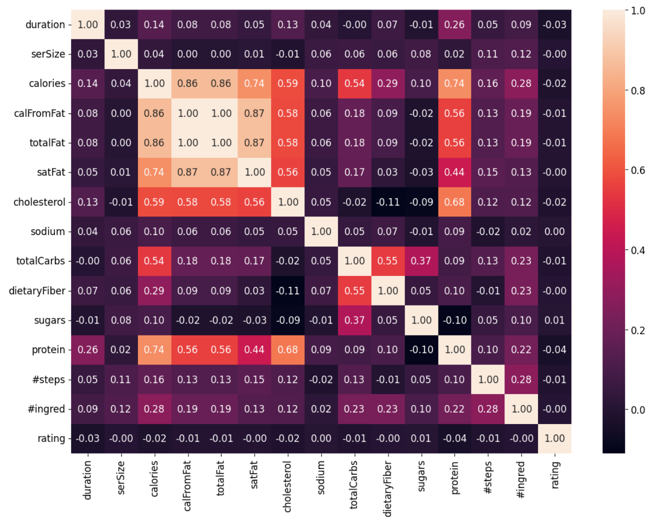


Replicated Graph

Ingredient Correlation HeatMap: I was able to recreate this figure with the code provided.



Original Graph



Replicated Graph

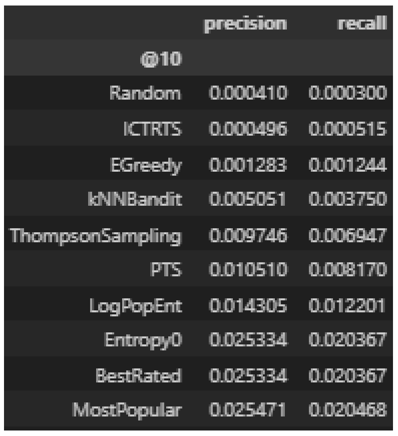
3.3 Experiments

As previously mentioned, the researchers conducted experiments to get a baseline on the dataset. Specifically, the researchers trained 10 models from the *irec* Library [9], 8 models from the *implicit* library [3], and 2 models from the *recommenders* library [7]. They measured 4 metrics – MAP, nDCG@10, Precision@10, and Recall@10.

Irec Models: There were 10 models created using the *irec* library: Random, ICTR, ϵ-greedy, kNNBandit, ThompsonSampling, PTS, LogPopEnt, Entropy, BestRated, and MostPopular. I did not need to change any code to run the models, but I did struggle with importing the *irec* library. Specifically, I was unable to pip install the library, since the underlying pipfile used "import sklearn" instead of "import scikit-learn". To overcome this, I simply downloaded the *irec* library off the Github, and put it in the same folder as the experiment code. This approach ended up working well, so I used it for the *implicit* and *recommenders* libraries as well. In total, the training and predictions step took 10h 37 minutes on my local machine.

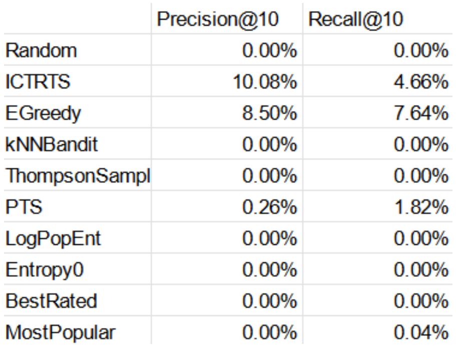
Model	*Library*	Precision@10	Recall@10
Random	*irec*	0.000410	0.000300
ICTR	*irec*	0.000546	0.000491
ϵ-Greedy	*irec*	0.001392	0.001149
kNNBandit	*irec*	0.005051	0.003750
ThompsonSampling	*irec*	0.009746	0.006947
PTS	*irec*	0.010483	0.008021
LogPopEnt	*irec*	0.014305	0.012201
Entropy	*irec*	0.025334	0.020367
BestRated	*irec*	0.025334	0.020367
MostPopular	*irec*	0.025471	0.020468

Original Irec Models



	precision	recall
@10		
Random	0.000410	0.000300
ICTRTS	0.000496	0.000515
EGreedy	0.001283	0.001244
kNNBandit	0.005051	0.003750
ThompsonSampling	0.009746	0.006947
PTS	0.010510	0.008170
LogPopEnt	0.014305	0.012201
Entropy0	0.025334	0.020367
BestRated	0.025334	0.020367
MostPopular	0.025471	0.020468

Replicated Irec Models



	Precision@10	Recall@10
Random	0.00%	0.00%
ICTRTS	10.08%	4.66%
EGreedy	8.50%	7.64%
kNNBandit	0.00%	0.00%
ThompsonSampl	0.00%	0.00%
PTS	0.26%	1.82%
LogPopEnt	0.00%	0.00%
Entropy0	0.00%	0.00%
BestRated	0.00%	0.00%
MostPopular	0.00%	0.04%

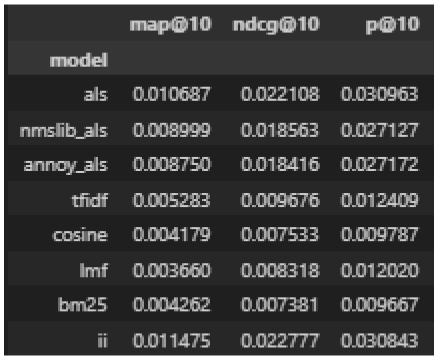
Percent Difference Between Observed and Replicated Values

The above table indicates that the results from the original paper were replicated quite well, as there is very little percent difference from the replicated results and the original. An acceptable margin error is 5-6%, and only 3 entries (precision and recall10 for EGreedy, precision10 for ICTR) exceed this margin.

Implicit Models: There were 8 models created using the *implicit* library: ALS, ALS_nmslib, ALS_annoy, Tf-idf, Cosine, LMF, BM25, and Item-item. As with *irec*, I put the entire *implicit* library in the codebase. I also had to put the nmslib library in the codebase, to get the nmslib_als model to work. In total, the training and predictions step took 4 minutes.

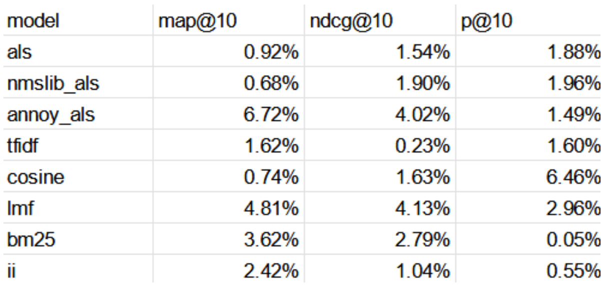
Model	*Library*	MAP	nDCG@10	Precision@10
ALS	*implicit*	0.010786	0.022453	0.031556
ALS_nmslib	*implicit*	0.009061	0.018922	0.027670
ALS_annoy	*implicit*	0.009380	0.019188	0.026773
Tf-idf	*implicit*	0.005370	0.009654	0.012213
Cosine	*implicit*	0.004210	0.007412	0.009193
LFM	*implicit*	0.003492	0.007988	0.011675
BM25	*implicit*	0.004422	0.007593	0.009672
Item-item	*implicit*	0.011759	0.023017	0.030674

Original Implicit Models



model	map@10	ndcg@10	p@10
als	0.010687	0.022108	0.030963
nmslib_als	0.008999	0.018563	0.027127
annoy_als	0.008750	0.018416	0.027172
tfidf	0.005283	0.009676	0.012409
cosine	0.004179	0.007533	0.009787
lmf	0.003660	0.008318	0.012020
bm25	0.004262	0.007381	0.009667
ii	0.011475	0.022777	0.030843

Replicated Implicit Models



model	map@10	ndcg@10	p@10
als	0.92%	1.54%	1.88%
nmslib_als	0.68%	1.90%	1.96%
annoy_als	6.72%	4.02%	1.49%
tfidf	1.62%	0.23%	1.60%
cosine	0.74%	1.63%	6.46%
lmf	4.81%	4.13%	2.96%
bm25	3.62%	2.79%	0.05%
ii	2.42%	1.04%	0.55%

Percent Difference Between Observed and Replicated Values

The above table indicates that the results from the original paper were replicated quite well, as there is very little percent difference from the replicated results and the original. An acceptable margin error is 5-6%, and only one entry (map10 for annoy) exceeds this margin.

Recommenders Models: There were 2 models created using the *recommenders* library: BPR and BiVAE. I was able to get the code working, but ran into the same memory issues as before when trying to make model predictions. I ran some tests where I subsetted a portion of the interactions matrix (i.e pp_interactions = pp_interactions.iloc[:N] where N is some number) to see how many entries could fit without exceeding memory and found that at most, I could get 30,000 entries. Since the entire user-interactions matrix is 600,000 entries, I would only be able to train the model on 5% of the data. As a result, I unfortunately don't believe I can replicate the 2 Recommenders Models on my local machine. Since the underlying code works on a smaller number of entries, however, I do believe the metrics would yield similar results to those researchers found given a computer with sufficient memory. **Note:** I tried running the Recommenders Models code in both VSCode Colab and Google Colab, but both yielded the same Memory Overflow issues (Google Colab offers even less memory than my local machine).

3.4 Analysis:

Overall, my replication was mostly successful. I was able to replicate the dataset using the crawling code, replicate the EDA figures exactly, and get metric scores very close to the original paper for both the *irec* and *implicit* models. However, memory limitations on my local machine prevented me from replicating the Recommenders Models succesfully, which leaves me unable to evaluate the error between replicated and original results for these models.

4 Extension

With Replication out of the way, I can move onto extension, where I add to the original body of work in several key ways. The original authors noted that the models they ran were more of a baseline than an optimized work.

4.1 Metrics

Metrics are extremely important for evaluating a recommender system, allowing researchers to compare different models and find the best option. In the original paper, all models were tested on one shared metric, Precision@10. This metric measures the number of relevant results in the top 10 recommender items, essentially evaluating how well the model recommends relevant items. Additionally, the *implicit* models were measured using Mean Average Precision and Normalized Discounted Cumulative Gain@10, and the *irec* models were measured using Recall@10. While these metrics are strong, additional metrics could help evaluate models in new ways. Furthermore, there is only one metric common to all three libraries, which makes comparing the models across libraries difficult.

To combat these issues and extend the original work, I introduce two new metrics to both the *irec* and *implicit* libraries – **AUC** and **Hit Ratio**. Notably, I am unable to run these metrics on the *recommenders* models due to memory constraints, but I will describe how they can be implemented.

AUC: AUC, or Area under ROC Curve, measures the likelihood that a random relevant item is ranked higher than a random irrelevant item [6]. The higher the AUC score (bounded between 0 and 1), the better the recommendation system generally.

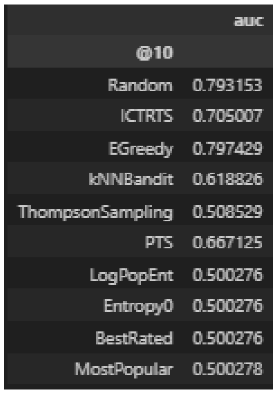
The *implicit* module already includes an AUC metric, via the “AUC_at_k" function in the implicit.evaluation folder. The *irec* module does not have the AUC metric directly, but does have a “Gini Coefficient" Metric. The Gini Coefficient is related to AUC by the following formula:

$$G = 2 \cdot AUC - 1$$

which can be rearranged to solve for AUC:

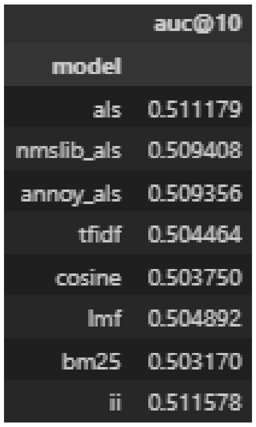
$$AUC = \frac{(G + 1)}{2}$$

To measure AUC, I modified the irec_util.py file by getting the gini values, then individually getting AUC values before averaging the set of AUC values. The *recommenders* library also includes an AUC function, which can be directly used to calculate AUC scores for the models.



	auc
@10	
Random	0.793153
ICTRTS	0.705007
EGreedy	0.797429
kNNBandit	0.618826
ThompsonSampling	0.508529
PTS	0.667125
LogPopEnt	0.500276
Entropy0	0.500276
BestRated	0.500276
MostPopular	0.500278

AUC scores for irec models

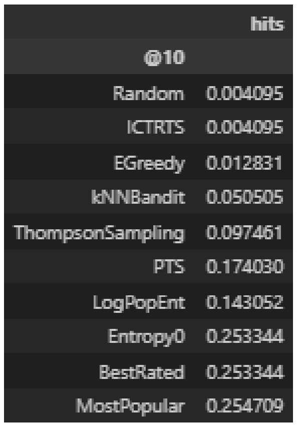


	auc@10
model	
als	0.511179
nmslib_als	0.509408
annoy_als	0.509356
tfidf	0.504464
cosine	0.503750
lmf	0.504892
bm25	0.503170
ii	0.511578

AUC scores for implicit models

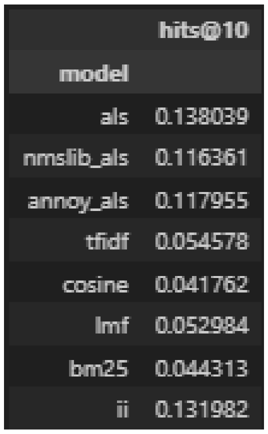
Hit Ratio@K: Hit ratio (or Hit Rate) @ K measures the share of users for which at least one relevant item is present in the recommendation list of size K. In this case, K = 10. Hit ratio measures both relevance and user coverage, evaluating how well the recommender system serves up relevant items to all users.[6]

The *irec* module already implements the Hit Ratio metric, via the “Hits" class in the Evaluator module. However, the *implicit* module does not include a “Hits" function/class, and does not offer a way to extract it from another metric. As a result, to measure Hit Ratio for the *implicit* models, I needed to implement a procedure from scratch. This procedure, defined as the “hit" function in implicit_util.py, goes through the test matrix batch by batch, making predictions using the recommend function on the train matrix. Every time a user has a liked item in their recommendations list, the number of hits is increased. The function returns the total number of hits divided by the total number of users, effectively measuring the hit ratio. This same function could be implemented in the cornac_util.py file to calculate hit ratio for the *recommenders* models.



	hits
@10	
Random	0.004095
ICTRTS	0.004095
EGreedy	0.012831
kNNBandit	0.050505
ThompsonSampling	0.097461
PTS	0.174030
LogPopEnt	0.143052
Entropy0	0.253344
BestRated	0.253344
MostPopular	0.254709

Hit Ratio scores for irec models



	hits@10
model	
als	0.138039
nmslib_als	0.116361
annoy_als	0.117955
tfidf	0.054578
cosine	0.041762
lmf	0.052984
bm25	0.044313
ii	0.131982

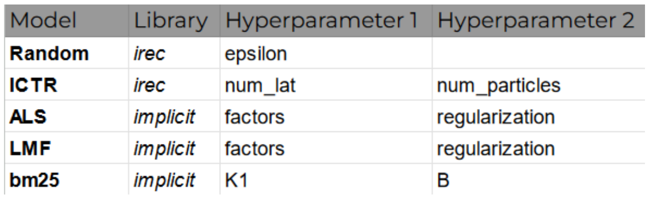
Hit Ratio scores for implicit models

Analysis: Overall, these metrics are valuable for both evaluating models in a vaccum and comparing models from different libraries. Previously, there was only one metric shared across all models, so introducing two additional metrics helps to make evaluation more accurate.

4.2 Hyperparameter Tuning[8]:

Hyperparameter tuning is the process of finding optimal values for parameters that affect a model's performance and behavior. These parameters are set prior to the training process and influence how the model learns from the data. Hyperparameter tuning is critical for optimizing the performance of a model, since parameter values can significantly impact the model's accuracy, generalization ability, and overall effectiveness in solving a particular task. In the original paper, researchers did **not** tune hyperparameters, resulting in potentially suboptimal model performance. This offers a chance for extension and improvement, as refining hyperparameters could yield better metric results.

In total, there are 5 models present in the *irec* and *implicit* libraries that have hyperparameters I can tune:

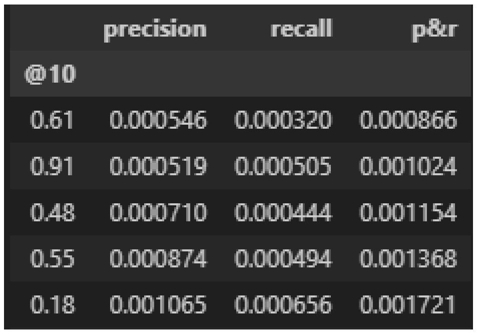


Model	Library	Hyperparameter 1	Hyperparameter 2
Random	*irec*	epsilon	
ICTR	*irec*	num_lat	num_particles
ALS	*implicit*	factors	regularization
LMF	*implicit*	factors	regularization
bm25	*implicit*	K1	B

Model Hyperparameters

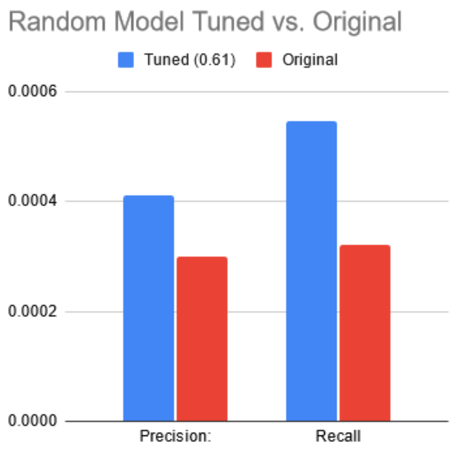
There are many methods that can be used to tune hyperparameters, but the two most common are **Grid-Search** and **Random Search**. In both methods, a grid of possible parameter values/combinations is generated. In Grid-Search, the entire grid is searched, and every possible parameter value/combination is tested. This method is exhaustive (guaranteed to find optimal parameter values), but is extremely computationally intensive. Random Search also searches the grid, but does so randomly, picking random combinations for a pre-defined number of iterations. Random Search is non-exhaustive, but much more realistic given the time and computation power constraints. As a result, I opted for Random Search to tune hyperparameters for the models.

Model 1 - Random: The Random model only has one hyperparameter – "epsilon". The epsilon value controls how the agent acts – when ϵ=1 the agent acts randomly, and When ϵ=0, the agent always takes the current greedy actions. As such, when tuning I set the range from 0.01 to 1.0, with 0.01 step size. I ran the tuning for 20 epochs, and specifically measured **precision** and **recall** since these were metrics measured in the original paper. The top 5 epsilon values sorted by (precision + recall) are:

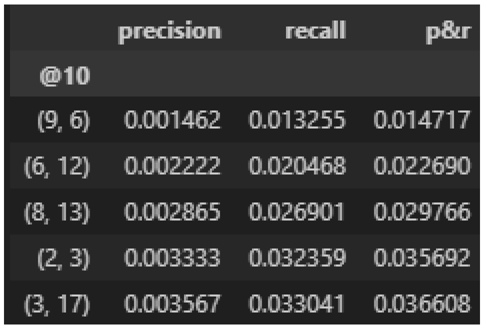


	precision	recall	p&r
@10			
0.61	0.000546	0.000320	0.000866
0.91	0.000519	0.000505	0.001024
0.48	0.000710	0.000444	0.001154
0.55	0.000874	0.000494	0.001368
0.18	0.001065	0.000656	0.001721

With tuned hyperparameters, I was able to achieve a better precision and recall than the original paper:

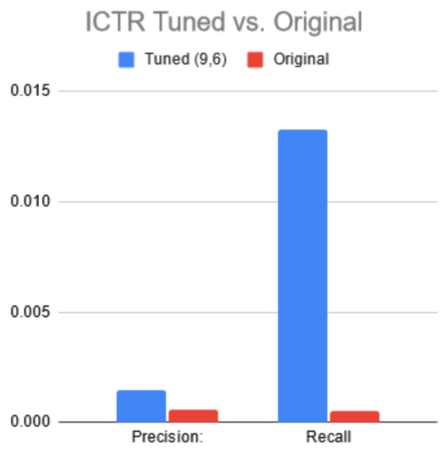


Model 2 - ICTR The ICTR model has two hyperparameters – num_lat and num_particles. The *irec* module does not directly document what these hyperparameters affect, but from my understanding the ICTR model uses particle-based learning (hence num_particles), and latent factors (num_lat controls number of latent factors). I selected a range of 1 to 10 and 1 to 20 for num_particles and num_lat respectively, and ran tuning for 20 epochs. Notably, this model is much more computationally intensive than the Random model, and running it on the entire dataset for 20 epochs would take multiple days. To combat this, I ran the tuning on a subset of the dataset, sectioning off 100,000 entries of the total 610,000 to greatly reduce traning time. The top 5 parameter combination values sorted by (precision + recall) are:

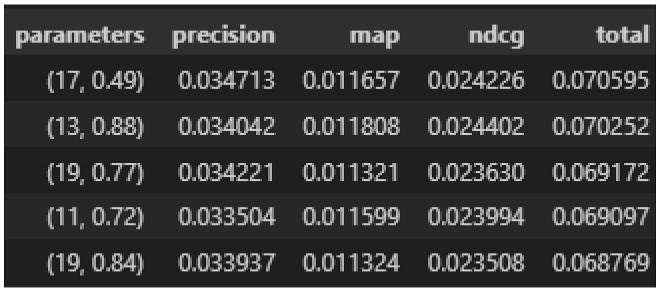


	precision	recall	p&r
@10			
(9, 6)	0.001462	0.013255	0.014717
(6, 12)	0.002222	0.020468	0.022690
(8, 13)	0.002865	0.026901	0.029766
(2, 3)	0.003333	0.032359	0.035692
(3, 17)	0.003567	0.033041	0.036608

With tuned hyperparameters, I was able to achieve better precision and recall than the original:

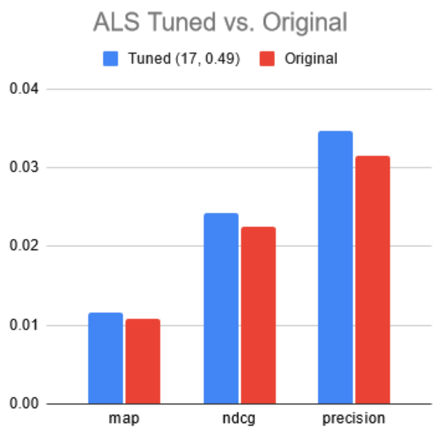


Model 3 - ALS The ALS (Alternating Least Squares) model has two hyperparameters – factors and regularization. Factors refers to the number of latent factors (variables that can only be inferred indirectly), and regularization refers to the regularization term, which is used to reduce bias and overfitting. The original values used are factors = 8 and regularization = 0.1, so I chose a range of 1 to 20 for factors and 0.01 to 1 (step size 0.01) for regularization. With 20 epochs, the top 5 parameter combination values sorted by (precision + map + ndcg) are:

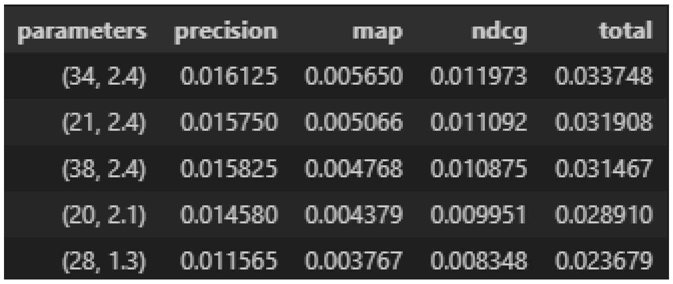


parameters	precision	map	ndcg	total
(17, 0.49)	0.034713	0.011657	0.024226	0.070595
(13, 0.88)	0.034042	0.011808	0.024402	0.070252
(19, 0.77)	0.034221	0.011321	0.023630	0.069172
(11, 0.72)	0.033504	0.011599	0.023994	0.069097
(19, 0.84)	0.033937	0.011324	0.023508	0.068769

I was able to achieve better precision, map, and ndcg than the original paper:

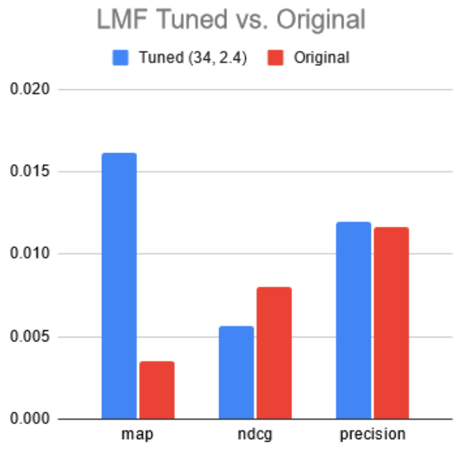


Model 4 - LMF: The LMF (Latent Matrix Factorization) model has the same two hyperparameters (factors and regularization) as the ALS model. However, they have a different range, since the original values used are factors = 30 and regularization = 1.5. Based on these values, I chose a range of 20 to 40 for factors and 0.5 to 2.5 (step size 0.1) for regularization. With 20 epochs, the top 5 parameter combination values sorted by (precision + map + ndcg) are:

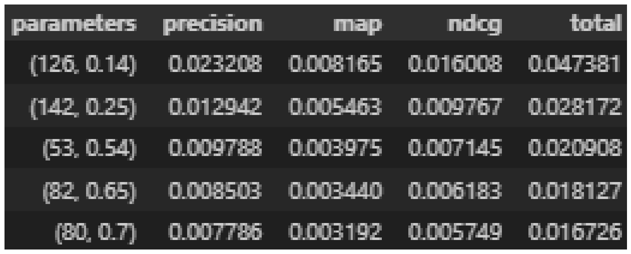


parameters	precision	map	ndcg	total
(34, 2.4)	0.016125	0.005650	0.011973	0.033748
(21, 2.4)	0.015750	0.005066	0.011092	0.031908
(38, 2.4)	0.015825	0.004768	0.010875	0.031467
(20, 2.1)	0.014580	0.004379	0.009951	0.028910
(28, 1.3)	0.011565	0.003767	0.008348	0.023679

I was able to achieve slightly better precision, much better map, and slightly worse ndcg than the original paper:

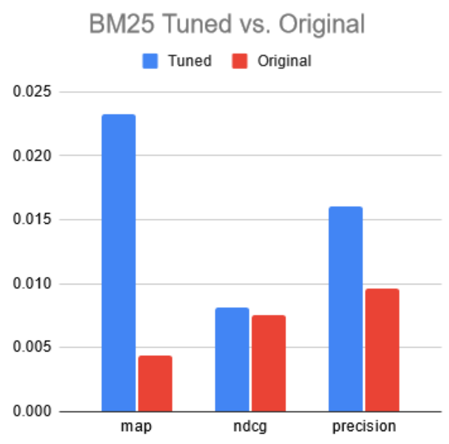


Model 5 - BM25: The BM25 (Best Match 25) model has two hyperparameters – K1 and B. These factors control the impact of term frequency saturation and document length normalization. In the original paper, the values chosen are K1=100 and B=0.5. Based on these values, I chose a range of 50 to 150 for K1 and 0.01 to 1.0 (step size 0.01) for B. With 20 epochs, the top 5 parameter combination values sorted by (precision + map + ndcg) are:



parameters	precision	map	ndcg	total
(126, 0.14)	0.023208	0.008165	0.016008	0.047381
(142, 0.25)	0.012942	0.005463	0.009767	0.028172
(53, 0.54)	0.009788	0.003975	0.007145	0.020908
(82, 0.65)	0.008503	0.003440	0.006183	0.018127
(80, 0.7)	0.007786	0.003192	0.005749	0.016726

I was able to achieve better precision, map, and ndcg than the original paper:



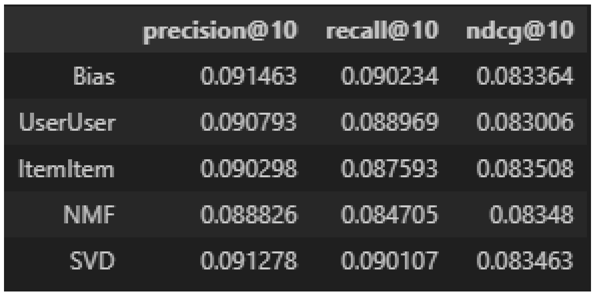
Analysis: Overall, I think the hyperparameter tuning was a strong success, since I managed to beat all of the original paper's metrics. However, there is also definitely room for improvement. Specifically, increasing the number of epochs, testing out different ranges of hyperparameter values, and performing Grid-Search CV could help improve the metrics even more.

4.3 Additional Models:

The original paper included 22 models from 3 different libraries, *implicit*, *irec*, and *recommenders*. In this section I extend the paper by introducing 5 models from the *surprise* library [5] – Bias, kNN User-User, kNN Item-Item, SVD, and NMF. These models are evaluated using the following three metrics – precision@10, recall@10, and ndcg@10.

Although *surprise* does not feature a precision or recall metric in the codebase, the documentation provides the code for a function that calculates precision and recall @ k. NDCG is also not present in the *surprise* codebase, but I was able to reuse my ndcg code from Assignment 2 to calculate the score. The pp_interactions matrix is split into train and test sets (test size of 20%) that are used to fit the models and make predictions.

The following table summarizes the results from running the 5 *surprise* models:



	precision@10	recall@10	ndcg@10
Bias	0.091463	0.090234	0.083364
UserUser	0.090793	0.088969	0.083006
ItemItem	0.090298	0.087593	0.083508
NMF	0.088826	0.084705	0.08348
SVD	0.091278	0.090107	0.083463

As the table indicates, the surprise models performed very well in comparison to the models from the *implicit* and *irec* libraries. In fact, the surprise models seem to also beat the *recommenders* models, which performed the best in the original paper. The model parameters are currently untuned, offering room for additional metric improvement via hyperparameter tuning.

5 Conclusions and Future Work

Overall, the replication and extension process ended up going well. I was able to replicate the HUMMUS dataset, generate the associated figures, and run most models with very similar results to the original paper. However, the lack of memory and computational power of my local machine was a bottleneck factor that prevented me from successfully replicating the *recommenders* models. Along with replication, I extended the paper in 3 key ways. First, I added two new metrics, AUC and Hit-Ratio@k, both of which provide valuable information that can be used to compare models across libraries. I also performed hyperparameter tuning using Random Grid Search, achieving better metrics than the original paper. Finally, I added 5 new models from the *surprise* library, achieving better precision, recall, and ndcg than the original paper.

There are a few ways my work can be improved, as well as a few more ways the original paper can be extended. For one, with a better computer, the *recommenders* models could be successfully replicated in reasonable time. Additionally, Grid-Search Hyperparameter Tuning could be performed with a powerful enough computer, which would yield even better metric performance. Tuning the *surprise* models I added, and potentially adding more (e.g. a custom FunkSVD model) could also yield more valuable information. Along with refining my models, future work could involve extending the original paper in new ways. For example, adding a hybrid recommender that takes recipe tags, food categories, cooking steps, and more into consideration could solve the cold-start problem and improve recommender performance. Additionally, implementing *personalized* scores liked BMI, height, weight, age, etc. could yield more relevant recommendations.

6 Code:

All the code written for this paper is available on my GitHub[4]. In total, I modified 3 files – dataset_scripts.ipynb, implicit_util.py, and irec_util.py. Whenever I modified code, I clearly documented it in comments. Additionally, I added new MarkDown section headers to indicate my added code sections in the ipynb file. I did not upload the entire HUMMUS codebase to my GitHub, since there are many files, and I only modified 3 of them. To replicate my code, simply download the original codebase from the HUMMUS GitLab [2], and replace the 3 aforementioned files with my versions of them.

References

[1] Sylvie Calabretto Lionel Brunie Armin Gerl Harald Kosch Felix Bölz, Diana Nurbakova. 2023. HUMMUS: A Linked, Healthiness-Aware, User-centered and Argument-Enabling Recipe Data Set for Recommendation. *In Seventeenth ACM Conference on Recommender Systems (RecSys '23)* (2023). https://doi.org/10.1145/3604915.3609491

[2] felix134. 2023. HUMMUS GitLab Repository. (2023). https://gitlab.com/felix134/connected-recipe-data-set

[3] Ben Fred. 2022. Implicit: Fast Python Collaborative Filtering for Implicit Feedback Datasets. (2022). https://github.com/benfred/implicit

[4] Anirudh Ganesh. 2024. Replication Assignment GitHub Repository. (2024). https://github.umn.edu/ganes099/CSCI-5123-Replication-Assignment/tree/main

[5] Nicolas Hug. 2022. Surprise: A Python scikit for building and analyzing recommender systems. (2022). https://github.com/NicolasHug/Surprise

[6] Sangram Kapre. 2021. Common metrics to evaluate recommendation systems. *Medium* (2021). https://flowthytensor.medium.com/some-metrics-to-evaluate-recommendation-systems-9e0cf0c8b6cf

[7] Microsoft. 2023. Microsofts Recommenders GitHub Page. (2023). https://github.com/recommenders-team/recommenders

[8] Shanthababu Pandian. 2022. A Comprehensive Guide on Hyperparameter Tuning and its Techniques. *Analytics Vidhya* (2022). https://www.analyticsvidhya.com/blog/2022/02/a-comprehensive-guide-on-hyperparameter-tuning-and-its-techniques/

[9] Werneck et. al. Silva, Silva. 2022. IRec: An Interactive Recommendation Framework. *In Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval* (2022). https://doi.org/10.1145/3477495.3531754