Music Recommendation

Matthew Blessing

Introduction

Purpose

To aid music listeners in expanding their tastes and discovering new music, which in turn will keep users engaged in streaming platforms.

Aim

To see how the use of a hybrid recommender system affects the relevance and diversity of recommendations, which are important in expanding user tastes.

To see if these aspects can be improved using graph-based approaches.

Dataset

Combination of two datasets:

- A Spotify dataset – contains song metadata and audio content [1]
- The Spotify Million Playlist Dataset – contains user-song interactions data

Song Content Features

- Track ID
- Track Name
- Artist Name
- Track Popularity
- Artist Popularity
- Genre List

- Key
- Loudness
- Mode
- Tempo

- Danceability
- Energy
- Speechiness
- Acousticness
- Instrumentalness
- Liveness
- Valence

User-Song Interaction Features

- User ID
- Track ID
- Playlist Name

Implicit Feedback

The dataset contains no explicit user feedback.
It only tells us how many times a user has added a song to a playlist.

This is **positive-only implicit feedback**:

- If a user has added a song to a playlist, this is a positive preference.

  How can we know which songs a user likes more than others?

- If a user has not added a song to any playlists, this is not necessarily a negative preference.

  Either:

  - The user has heard the song before and doesn't want it in a playlist (negative preference), or
  - The user has never heard the song before.

Data Preparation

I grouped the interactions data by user ID and playlist name, then randomly sampled playlists containing at least 5 songs until I reached 100,000 user-song interactions.

[This subset contained interactions involving 3,060 users and 14,757 songs]

I also created some smaller datasets using the same approach of sizes 25,000 and 50,000.

Pre-processing:

For the song content:

- Key - one-hot encoding
- Loudness and tempo - min-max scaling so all columns have values in the range [0,1]
- Genres list and artist name – TF-IDF
- Drop popularity columns (don't want popularity bias)

I also converted the data to an interactions matrix.
The number in row u, column i is the number of times user u has saved song i to a playlist.
Sparsity of this matrix = 99.8%.

RS1: A Hybrid Approach

My first recommender system is a hybrid consisting of three components.
The first component is:

Content-Based Filtering (CBF)

If a user likes lots of songs with a particular feature, then they will most likely enjoy other songs with this feature.

- This approach is suitable as there is lots of song content.

(+) Recommendations are explainable
(+) New items are not victim to the cold start problem (doesn't need ratings)

(-) New users are victim to the cold start problem (we don't know what they like!)
(-) Lack of serendipity

RS1: A Hybrid Approach

My first recommender system is a hybrid consisting of three components.
The first component is:

Content-Based Filtering (CBF)

Each song is represented by a vector containing the numerical audio data and the genre and artist information in TF-IDF form.

The vectors are very sparse, and all vector elements are between 0 and 1.

- The chosen similarity measure was cosine similarity, which can be efficiently computed in Python for sparse data quicker than other measures.
  The formula for cosine similarity is:

$$sim(i,j) = \cos(\vec{i},\vec{j}) = \frac{\vec{i}\cdot\vec{j}}{\|\vec{i}\|_2 * \|\vec{j}\|_2}$$

- User profiles were calculated by taking the mean of the profiles of all songs that the user has interacted with.

RS1: A Hybrid Approach

My first recommender system is a hybrid consisting of three components. The first component is:

Content-Based Filtering (CBF)

The hyperparameters for this model are weights to multiply the song features by to prevent certain features from dominating the recommendations.

The model performed best with the following weightings:

- Multiplying the key columns by 0.3
- Not including the mode (multiplying the column by 0)
- Multiplying the loudness and tempo columns by 0.9
- Multiplying the genre columns by 0.7

RS1: A Hybrid Approach

My first recommender system is a hybrid consisting of three components. The second component is:

Model-Based Collaborative Filtering (CF)

If user A and user B like lots of the same music, then songs that user B likes but user A has not heard before make good recommendations for user A.

- More efficient and scalable than a memory-based approach.
- Can help diversify recommendations

(+) Serendipity – counteracts the lack of this in CBF

(-) Cold start problem for both users and items
(-) Potential overfitting to the data

RS1: A Hybrid Approach

My first recommender system is a hybrid consisting of three components.
The second component is:

Model-Based Collaborative Filtering (CF)

- I used the Alternating Least Squares (ALS) matrix factorization algorithm as described in [2], which was designed for implicit feedback datasets.
- Prediction occurs by computing the dot product between the user factors and item factors. Bias terms are not needed as we have implicit feedback.
- As described in [2], we use a confidence matrix instead of an interactions matrix, where confidence in user u liking item i is given by:

$$c_{ui} = 1 + \alpha r_{ui},$$

where r_{ui} is value (u, i) in the interactions matrix.

The model performed best with 240 latent factors, a regularization term of 0.01 and an alpha value of 40.

RS1: A Hybrid Approach

My first recommender system is a hybrid consisting of three components.
The third component is:

Constraint-Based Knowledge-Based Filtering (KBF)

Planned to extract contextual information from playlist names as in [3], but a large proportion of them only contained artist/genre/album names.
→ Instead, KBF allows user to get context-dependent recommendations.

- The context of the user will influence their tastes, and instead of inferring the context as in a context-aware recommender system, this can be expressed by the user in terms of constraints.

(+) Produces recommendations more relevant to what user wants at that moment in time

(-) Requires an extra step from the user (knowledge input)

RS1: A Hybrid Approach

My first recommender system is a hybrid consisting of three components.
The third component is:

Constraint-Based Knowledge-Based Filtering (KBF)

Users will be able to constrain recommendations to particular artists and genres, as well as keys, modes, or tempos.

Other song features such as "danceability" are somewhat subjective, so they won't be included.

Implementation: pre-filtering approach, where the only songs that can be recommended are those that match the user constraints.

RS1: A Hybrid Approach

The Hybrid:

I combined CBF and CF in a weighted approach with weights 0.6 and 0.4 respectively, and to improve user experience, I made the KBF component of the hybrid optional.

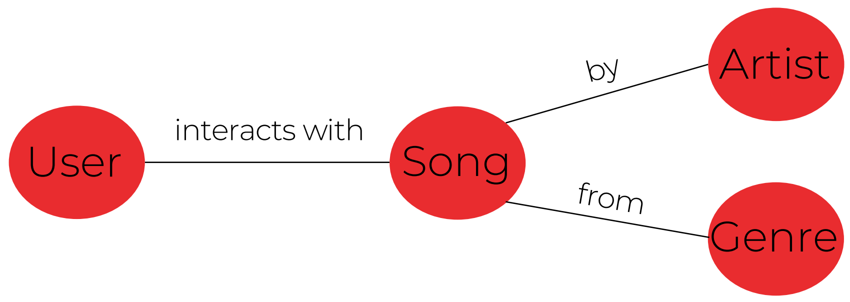
I tried a weighted approach, mixed approach, and cascade approach, and chose the weighted approach as it gave the best results.

The combination of CBF, CF and KBF is suitable for my objective since:

- CBF doesn't rely on ratings, so new items don't suffer from the cold start problem.
- CF provides diversity and serendipity.
- KBF can help provide better recommendations to users suffering from the cold start problem, or users who want specific recommendations at a particular time.

RS2: Graph-Based Hybrid Approach

The dataset includes relationships that naturally form a graph and are shown below:



Content-based POV:

- Use the song content to create a knowledge graph.

Collaborative POV:

- Collaborative graph of users and songs.
- Link prediction: find the best edges that we could add to the graph to connect users and songs.

So far, the methods used have only considered first-order connections in these graphs, but the overall graph structure can be used to improve recommendations.

RS2: Graph-Based Hybrid Approach

How can we do this?

For the CBF:

- I created a knowledge graph consisting of nodes for every song, artist, and genre in the data.
- I used Node2Vec to get a set of graph embeddings for the songs, which I used in the song profiles instead of the TF-IDF vectors used previously [4].

For the CF:

- I used a model called LightGCN to learn a set of embeddings for the users and songs in the system [5].
- Scores were calculated via the dot product between user and song embeddings.

The KBF process remained the same.

RS2: Graph-Based Hybrid Approach

Hyperparameter values

CBF:
- Multiply key column by 0.5 and remove mode column (0 weight)
- Node2Vec parameters used in [6]

CF – hyperparameter values as specified in [5]:
- Embedding size of 64
- 3 propagation layers used to update the embeddings
- Training for 1000 epochs with a batch size of 1024 and a regularization coefficient of 0.0001

I did not perform hyperparameter tuning, but this could be used in the future to improve performance.

Weights remained the same: 0.6 for CBF and 0.4 for CF.

Evaluation

Metric 1:

We have implicit feedback, so a metric concerning ranking accuracy is more suitable than one for rating accuracy. The accuracy metric I have chosen to use is recall@k:

$$recall@k = \frac{number\ of\ recommendations\ that\ are\ relevant}{number\ of\ relevant\ songs\ in\ the\ test\ set}$$

- This evaluates the proportion of relevant test items that appear in the top k recommendations and is therefore a measure of ranking accuracy.
- Doesn't depend on relative positions of each recommendation, which is not so important in song recommendation.
- Precision not appropriate as no interaction is not necessarily a negative preference.

Evaluation

Metric 1:

Note, if a user has more than k relevant songs in the test set, then their maximum recall@k will be less than 1.

Therefore, I normalised the recall@k for each user by dividing it by their maximum recall@k

$$recall_{max}@k = \frac{\min(k, number\ of\ relevant\ songs\ in\ the\ test\ set)}{number\ of\ relevant\ songs\ in\ the\ test\ set},$$

to get

$$normalised\ recall@k = \frac{recall@k}{recall_{max}@k}.$$

Values of k used: 5, 10, and 20.

Evaluation

Metric 2:

Given that I want to improve user preference prediction and diversify user recommendations, my additional metric of choice is novelty.

- Measures how good a recommender system is at recommending items that many users have not seen before.
- This is extremely important since we want the recommender to expand users' tastes.

Novelty is defined as follows:

$$Novelty(R) = \frac{\sum_{i \in R} -\log_2 p(i)}{|R|},$$

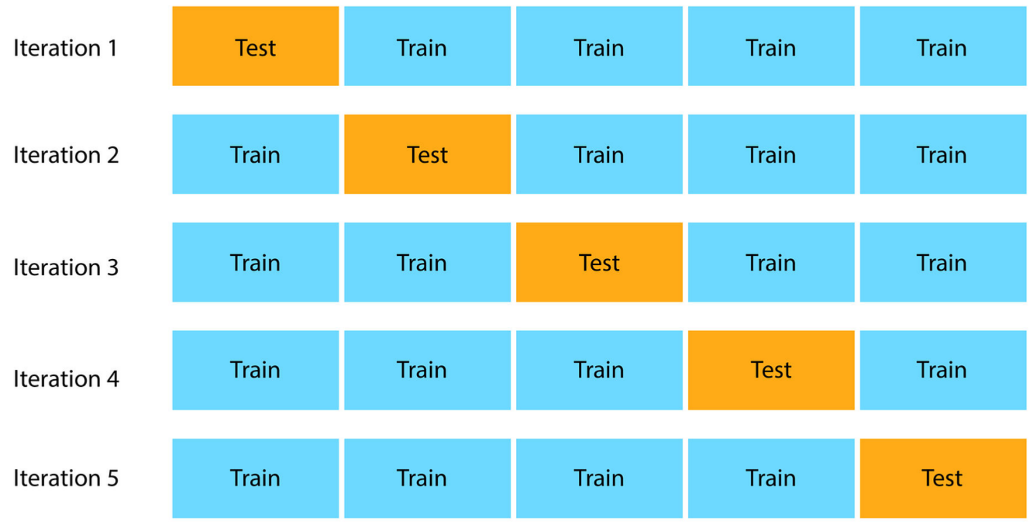
where R is a set of recommendations and $p(i)$ is the proportion of users who have interacted with item i.

$Novelty(R)$ can range from 0 to infinity and the larger the value, the better.

Evaluation

Data Splitting:

- I used 5-fold cross validation to evaluate my system. This was stratified by user ID to evenly distribute user interactions throughout the folds.
- Each fold involved at least one user interaction due to the data preparation.



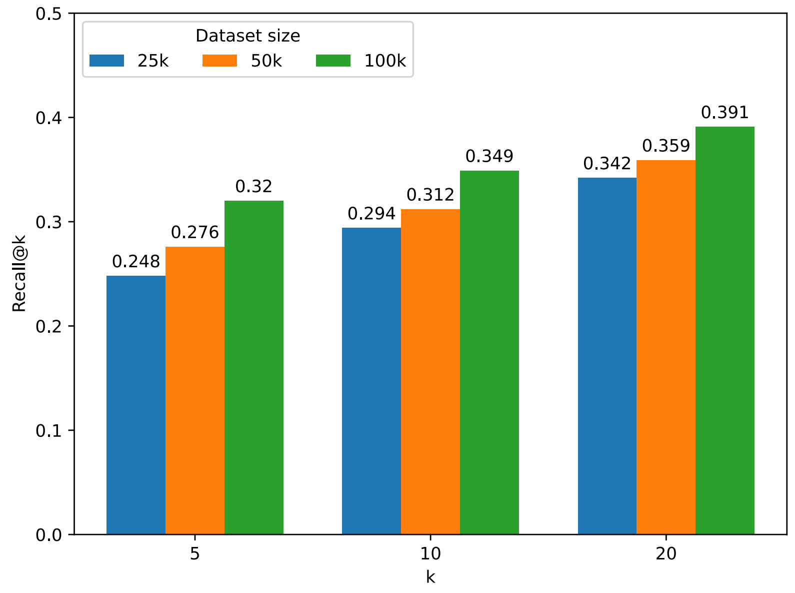
Iteration 1	Test	Train	Train	Train	Train
Iteration 2	Train	Test	Train	Train	Train
Iteration 3	Train	Train	Test	Train	Train
Iteration 4	Train	Train	Train	Test	Train
Iteration 5	Train	Train	Train	Train	Test

Figure 1: 5-fold cross validation.
Source: [7]

Evaluation – RS1

Recall@k:

The recall increased with both dataset size and the value of k.



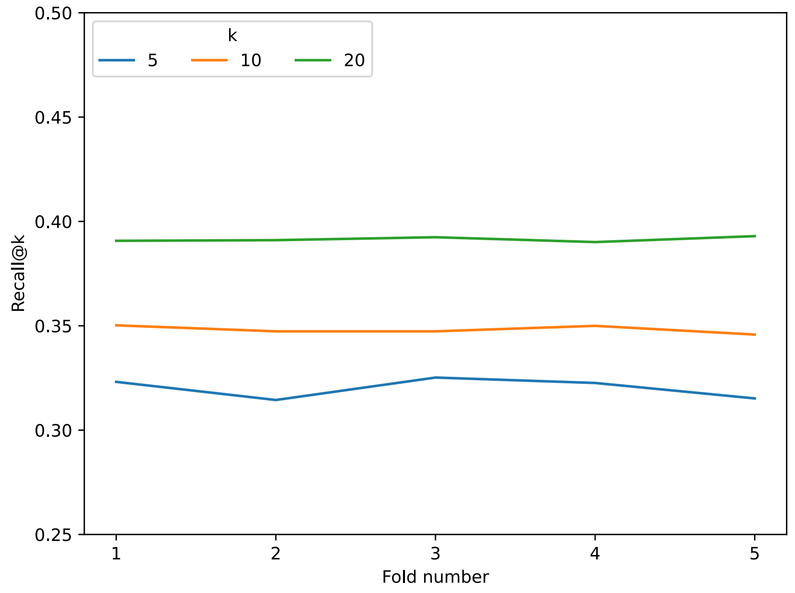


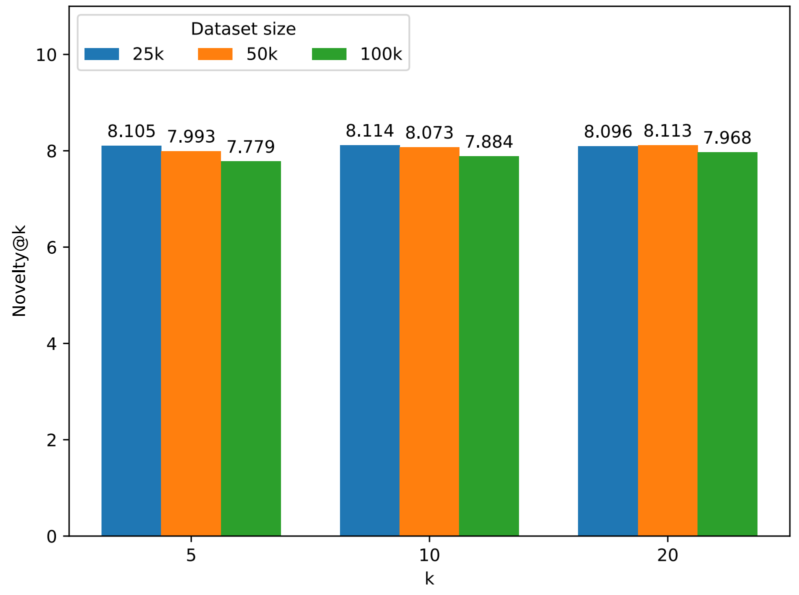
Figure 2: (Left) A grouped bar graph of the recall@k (k = 5, 10, and 20) on the three datasets for RS1. (Right) A line graph showing the recall@k (k = 5, 10, and 20) for each fold of the 100,000 dataset.

Evaluation – RS1

Novelty@k:

The novelty (mostly) decreased as dataset size increased and (mostly) increased with k.

The changes were very small.



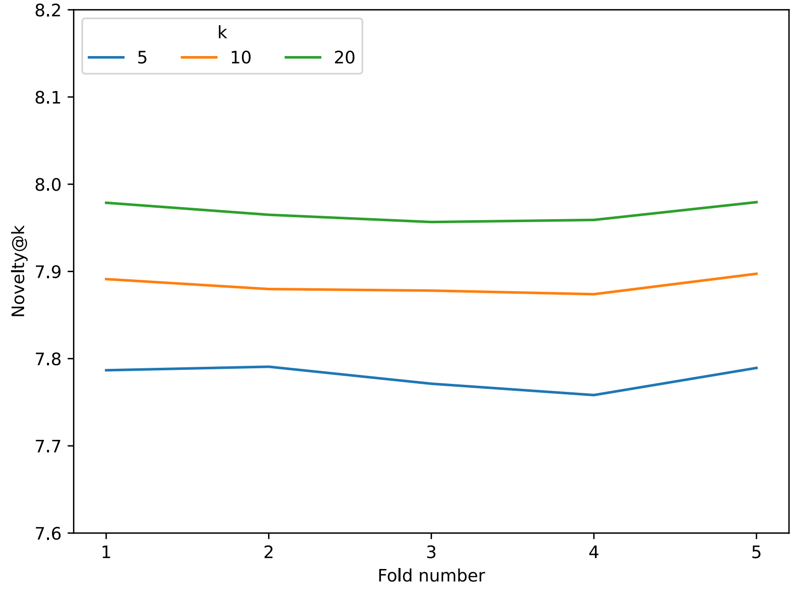
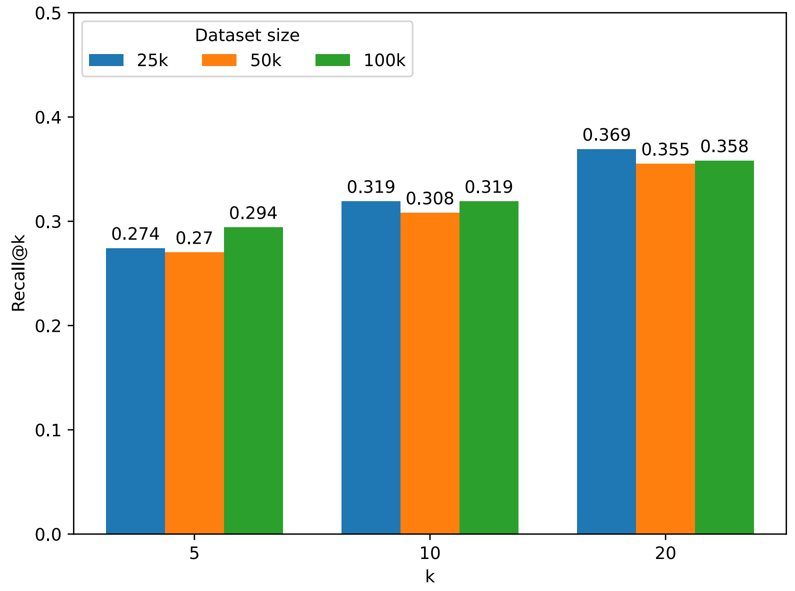


Figure 3: (Left) A grouped bar graph of the novelty@k (k = 5, 10, and 20) on the three datasets for RS1. (Right) A line graph showing the novelty@k (k = 5, 10, and 20) for each fold of the 100,000 dataset.

Evaluation – RS2

Recall@k:

The recall increased with k, but there was no pattern regarding dataset size.



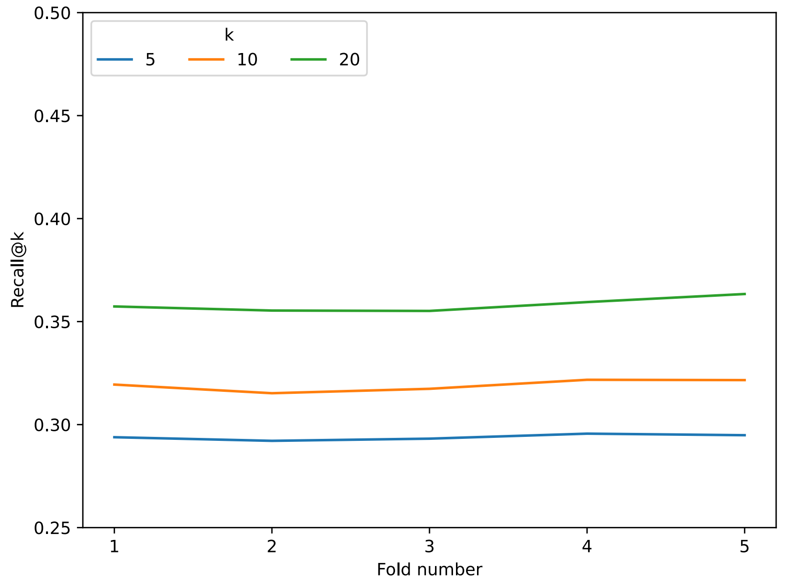
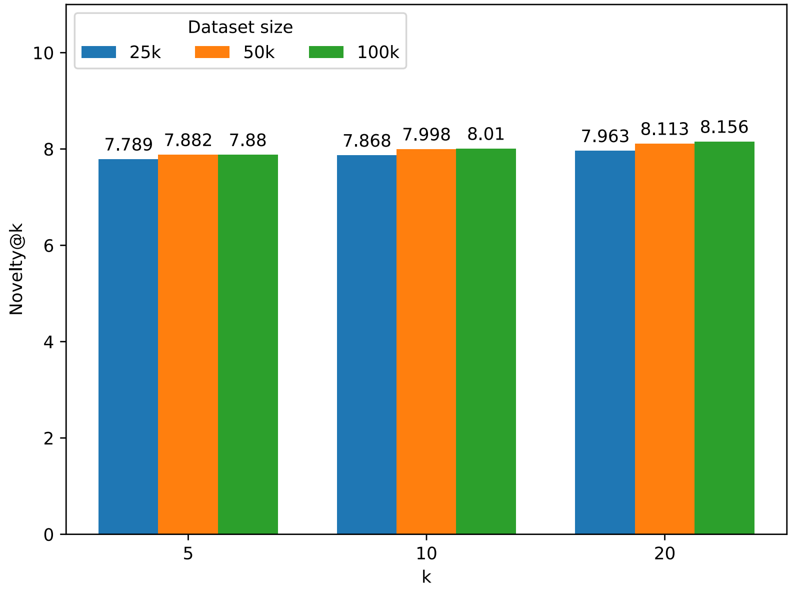


Figure 4: (Left) A grouped bar graph of the recall@k (k = 5, 10, and 20) on the three datasets for RS2. (Right) A line graph showing the recall@k (k = 5, 10, and 20) for each fold of the 100,000 dataset.

Evaluation – RS2

Novelty@k:

The novelty increased with both dataset size and the value of k.



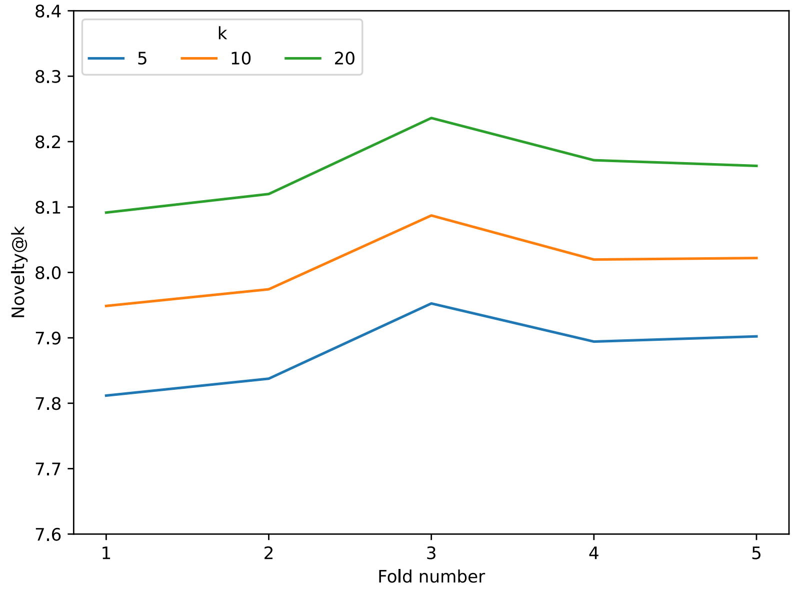
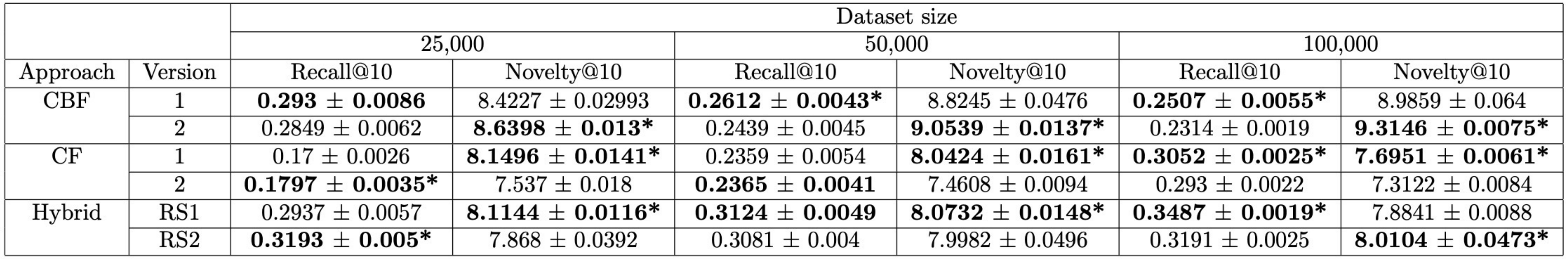


Figure 5: (Left) A grouped bar graph of the novelty@k (k = 5, 10, and 20) on the three datasets for RS2. (Right) A line graph showing the novelty@k (k = 5, 10, and 20) for each fold of the 100,000 dataset.

Evaluation – Comparison

Observations:

- CBF1 had better recall than CBF2, but the improved version had better novelty.
- CF1 had better novelty than CF2, but CF2 mostly had better recall.
- RS1 mostly outperformed RS2.



		Dataset size					
		25,000		50,000		100,000	
Approach	Version	Recall@10	Novelty@10	Recall@10	Novelty@10	Recall@10	Novelty@10
CBF	1	**0.293 ± 0.0086**	8.4227 ± 0.02993	**0.2612 ± 0.0043***	8.8245 ± 0.0476	**0.2507 ± 0.0055***	8.9859 ± 0.064
	2	0.2849 ± 0.0062	**8.6398 ± 0.013***	0.2439 ± 0.0045	**9.0539 ± 0.0137***	0.2314 ± 0.0019	**9.3146 ± 0.0075***
CF	1	0.17 ± 0.0026	**8.1496 ± 0.0141***	0.2359 ± 0.0054	**8.0424 ± 0.0161***	**0.3052 ± 0.0025***	**7.6951 ± 0.0061***
	2	**0.1797 ± 0.0035***	7.537 ± 0.018	**0.2365 ± 0.0041**	7.4608 ± 0.0094	0.293 ± 0.0022	7.3122 ± 0.0084
Hybrid	RS1	0.2937 ± 0.0057	**8.1144 ± 0.0116***	**0.3124 ± 0.0049**	**8.0732 ± 0.0148***	**0.3487 ± 0.0019***	7.8841 ± 0.0088
	RS2	**0.3193 ± 0.005***	7.868 ± 0.0392	0.3081 ± 0.004	7.9982 ± 0.0496	0.3191 ± 0.0025	**8.0104 ± 0.0473***

Table 1: The recall@10 and novelty@10 for each recommender when evaluated on three datasets of increasing size.
Bold text indicates the better performance for each approach.
* indicates a significant difference in the mean recall/novelty across folds between the two versions of a particular approach based on a t-test with a 5% significance level.

References

[1] E. Chang, 2021, “[Data Extracted from Spotify API]”, GitHub. [Online]. Available: https://github.com/enjuichang/PracticalDataScience-ENCA/blob/main/data/allsong_data.csv

[2] Y. Hu, Y. Koren, and C. Volinsky. (15-19 Dec. 2008). Collaborative Filtering for Implicit Feedback Datasets. Presented at the 8th IEEE Int. Conf. on Dat. Min., Pisa, Italy. [Online]. Available: http://yifanhu.net/PUB/cf.pdf

[3] M. Pichl, E. Zangerle, and G. Specht, “Improving Context-Aware Music Recommender Systems Beyond the Pre-filtering Approach,” in *Proc. of the ACM ICMR,* 2017, pp. 201-208, [Online]. Available: https://doi.org/10.1145/3078971.3078980

References

[4] A. Grover, J. Jeskovec, “node2vec: Scalable Feature Learning in Networks,” in *Proc. of the 22nd ACM SIGKDD Int. Conf. on Kno. Dis. and Dat. Min.*, 2016, [Online]. Available: https://cs.stanford.edu/~jure/pubs/node2vec-kdd16.pdf

[5]] X. He et al. (Jul. 2020). LightGCN: Simplifying and Powering Graph Convolutio n Network for Recommendation. Presented at the 43rd Int. ACM SIGIR Conf. on Res. and Dev. in Inf. Ret. [Online]. Available: https://arxiv.org/pdf/2002.02126v4.pdf

[6] Elior Cohen, “Node2Vec”, node2vec [Online]. Available: https://github.com/elior c/node2vec

[7] C. Y. Wijaya, “Cross-Validation to Improve Model Reliability.” https://cornellius.s ubstack.com/p/cross-validation-to-improve-model (accessed Jan. 17, 2024).

Code References

[8] E. Chang (2021) Content-based Filtering Spotify Song Recommendation System [Source Code]. Available: https://github.com/enjuichang/PracticalDataScience-ENCA/blob/main/notebooks/content_based_recsys.ipynb

[9] Gabriel Moreira (2020) Recommender Systems in Python 101 (Version 4) [Source Code]. Available: https://www.kaggle.com/code/gspmoreira/recommender-systems-in-python-101

[10] Ben Frederickson, "AlternatingLeastSquares", Implicit [Online]. Available: https://benfred.github.io/implicit/api/models/cpu/als.html

[11] The Matplotlib development team, Grouped bar chart with labels [Source Code]. Available: https://matplotlib.org/stable/gallery/lines_bars_and_markers/barchart.html

Code References

[12] Dipanjan Das (2021) LightGCN->Pytorch(From Scratch) (Version 3) [Source Code]. Available: https://www.kaggle.com/code/dipanjandas96/lightgcn-pytorch-from-scratch